```
# python --optimize interview
from typing import TypeVar, Protocol
from functools import lru_cache
import heapq, asyncio
```

TECHNICAL INTERVIEW PREPARATION

Python Under Pressure

A Complete Interview Guide — written from the interviewer's side of the table.

Ahmed Aly

Technical Lead · Software & ML/AI Systems · 15+ Years

AA

Python Under Pressure Free Sample

Ahmed Aly

Chapter 32: Python Gotchas – What Trips Everyone Up

Part VII begins here. The previous six parts covered language internals, performance, production tooling, DSA, and design patterns. This part is the Interview Playbook – direct preparation for the Python technical interview.

Chapter 32 covers the gotchas: the behaviors that are correct by Python's rules but counterintuitive to most developers. These appear in coding interviews both as direct questions ("what does this print?") and as bugs you write under pressure. Knowing them cold is a filter that separates candidates who have done serious Python from those who have dabbled.

Fourteen gotchas. Each one explained, demonstrated with live output, and fixed.

Gotcha 1: Mutable Default Argument

```
#  The trap
def bad_append(item, lst=[]):
    lst.append(item)
    return lst

print(bad_append(1))   # [1]
print(bad_append(2))   # [1, 2]  <- expected [2]
print(bad_append(3))   # [1, 2, 3]  <- expected [3]
```

Why: Default argument values are evaluated once at function definition time, not at each call. The `[]` is created once and reused – the same list object is shared across all calls.

Verify:

```
print(bad_append.__defaults__)   # ([1, 2, 3],) -- the accumulated list
```

Fix: Use `None` as the sentinel and create a new list inside the function:

```
#  The fix
def good_append(item, lst=None):
    if lst is None:
        lst = []     # fresh list every call
    lst.append(item)
    return lst

print(good_append(1))   # [1]
print(good_append(2))   # [2]  <- correct
```

This applies to any mutable default – `{}`, `set()`, or custom objects. Only immutable defaults (`None`, `0`, `""`, tuples) are safe.

Gotcha 2: Late Binding Closures

```
#  The trap
funcs = [lambda x: x * i for i in range(4)]
print([f(2) for f in funcs])   # [6, 6, 6, 6] -- all use i=3
```

Why: Python closures capture variables by reference, not by value. By the time any `f` is called, the loop has finished and `i` is 3. All four lambdas share the same `i`.

Fix 1: Capture the current value as a default argument:

```
#  Fix 1 -- default argument captures value at definition time
funcs = [lambda x, i=i: x * i for i in range(4)]
print([f(2) for f in funcs])   # [0, 2, 4, 6]
```

Fix 2: Use `functools.partial`:

```
#  Fix 2 -- functools.partial
from functools import partial

def multiply(x, factor): return x * factor
funcs = [partial(multiply, factor=i) for i in range(4)]
print([f(2) for f in funcs])   # [0, 2, 4, 6]
```

Gotcha 3: `is` vs `==`

```
#  The trap -- using 'is' for value comparison
x = 257
y = 257
print(x is y)    # False -- different objects above interning range
print(x == y)    # True  -- always use == for value comparison
```

Why: `is` tests **identity** (same object in memory), not **equality** (same value). Python interns (caches) small integers from -5 to 256 and commonly used strings, so `is` accidentally works for those values – but fails unpredictably for larger integers and other objects.

```
a = 256; b = 256
print(a is b)    # True  -- interned
c = 257; d = 257
print(c is d)    # False -- not interned (CPython implementation detail)
print(c == d)    # True  -- correct comparison

# Only correct use of 'is': checking for None, True, False (singletons)
value = None
print(value is None)    #  correct
print(value == None)    # works but triggers linting warnings
```

Rule: Use `==` for value comparison. Use `is` only for singleton checks: `x is None`, `x is True`, `x is False`.

Gotcha 4: UnboundLocalError

```
#  The trap
x = 10

def confusing():
    print(x)   # UnboundLocalError -- even though x=10 exists globally
    x = 20     # this assignment makes x LOCAL throughout the function

confusing()   # UnboundLocalError: cannot access local variable 'x'
```

Why: If a variable is assigned anywhere in a function body, Python treats it as local throughout that function – even before the assignment. The `print(x)` at the top tries to read a local `x` that has not been assigned yet.

```
#  Fix 1 -- use global keyword (when you actually want to modify the global)
x = 10
def read_global():
    global x
    x += 1
    return x

#  Fix 2 -- pass as parameter (preferred)
def increment(x):
    return x + 1

result = increment(x)
```

Gotcha 5: Exception Variable Scope

```
#  The trap
try:
    raise ValueError("something went wrong")
except ValueError as e:
    message = str(e)    # save before the block ends!

# In Python 3, 'e' is DELETED after the except block
print(e)   # NameError: name 'e' is not defined
```

Why: In Python 3, the exception variable (`e`) is explicitly deleted after the `except` block completes – to break potential reference cycles. This is a deliberate change from Python 2 behavior.

```
#  Fix -- save to a different variable inside the except block
try:
    raise ValueError("something went wrong")
except ValueError as e:
    error_message = str(e)    # save to a different name
    error_type    = type(e).__name__

# error_message and error_type are available here
print(error_message)   # something went wrong
print(error_type)      # ValueError
```

Gotcha 6: Class Variable vs Instance Variable

```
#  The trap -- self.count += 1 creates an instance variable
class Counter:
    count = 0   # class variable -- shared across all instances

    def bad_increment(self):
        self.count += 1   # looks like it modifies the class variable...

c1 = Counter()
c2 = Counter()
c1.bad_increment()
c1.bad_increment()

print(Counter.count)    # 0  -- unchanged! c1.count shadows it
print(c1.count)         # 2  -- instance variable created on c1
print(c2.count)         # 0  -- c2 still reads the class variable
```

Why: `self.count += 1` is equivalent to `self.count = self.count + 1`. The right side reads `Counter.count` (class level), but the left side creates a new instance attribute `self.count` on `c1`. The class variable is never modified.

```
#  Fix -- explicitly reference the class
class Counter:
    count = 0

    def increment(self):
        Counter.count += 1   # or: type(self).count += 1
```

Gotcha 7: Modifying a Collection During Iteration

```
#  The trap -- modifying a list while iterating over it
items = [1, 2, 3, 4, 5]
for x in items:
    if x % 2 == 0:
        items.remove(x)   # skips elements -- list shifts under the iterator

print(items)   # [1, 3, 5] -- looks right but 4 was skipped!
# (It works here by accident -- in other cases you lose elements)
```

```
#  Dict: raises RuntimeError
d = {"a": 1, "b": 2, "c": 3}
try:
    for k in d:
        if d[k] == 2:
            del d[k]   # RuntimeError: dictionary changed size during iteration
except RuntimeError as e:
    print(f"Error: {e}")
```

Why: The iterator holds an index into the underlying data structure. Removing elements shifts the data, causing elements to be skipped or visited twice. Python detects dict modification and raises `RuntimeError`.

```
#  Fix 1 -- list comprehension (cleanest)
items = [1, 2, 3, 4, 5]
items = [x for x in items if x % 2 != 0]

#  Fix 2 -- iterate over a copy
for x in items[:]:           # items[:] creates a copy
    if x % 2 == 0:
        items.remove(x)

#  Fix for dict -- iterate over a copy of keys
d = {"a": 1, "b": 2, "c": 3}
for k in list(d.keys()):     # list() materializes the keys
    if d[k] == 2:
        del d[k]
```

Gotcha 8: Generator Exhaustion

```
#  The trap -- generators are single-use
gen = (x**2 for x in range(5))
print(sum(gen))   # 30  -- correct
print(sum(gen))   # 0   -- generator is exhausted!
print(list(gen))  # []  -- nothing left
```

Why: A generator computes values on demand and maintains its position. Once exhausted (all values yielded), it stays at the end – there is no automatic reset.

```
#  Fix 1 -- use a function that returns a fresh generator each call
def squares(n):
```

```
    return (x**2 for x in range(n))

print(sum(squares(5)))   # 30
print(sum(squares(5)))   # 30 -- fresh generator

#  Fix 2 -- materialize to a list if you need to consume multiple times
squares_list = [x**2 for x in range(5)]
print(sum(squares_list))   # 30
print(max(squares_list))   # 16
```

Gotcha 9: Tuple Requires a Trailing Comma

```
#  The trap -- parentheses alone do not make a tuple
t1 = (1)      # int 1, NOT a tuple
t2 = ("hi")   # str "hi", NOT a tuple
t3 = ()       # empty tuple -- exception to the rule

#  The fix -- trailing comma is required for single-element tuples
t4 = (1,)      # tuple with one element
t5 = ("hi",)   # tuple with one element
t6 = 1,        # also a tuple -- parens optional for multi-element

print(type(t1))   # <class 'int'>
print(type(t4))   # <class 'tuple'>
```

Gotcha 10: `None` Is Not `False`

```
#  The trap -- None and False are both falsy but not equal
print(None == False)        # False -- different values
print(None is not False)    # True  -- different objects

# Both are falsy:
if not None:  print("None is falsy")    # prints
if not False: print("False is falsy")   # prints

# But None == False is False:
print(None == 0)    # False
```

```
print(False == 0)   # True -- bool is a subclass of int!
print(True == 1)    # True
print(True + True)  # 2
print(True * 5)     # 5

#  Fix -- be explicit
value = None
if value is None:    #  checks for None specifically
    pass

if not value:        #  checks for any falsy value (None, 0, [], "")
    pass

if value is False:   #  checks specifically for False, not 0 or None
    pass
```

Gotcha 11: `sort()` Returns `None`

```
#  The trap -- assigning the result of sort()
items = [3, 1, 4, 1, 5]
result = items.sort()      # common mistake
print(result)              # None -- sort() modifies in-place and returns None!
print(items)               # [1, 1, 3, 4, 5] -- sorted in place

#  Fix -- use sorted() if you want the return value
items = [3, 1, 4, 1, 5]
result = sorted(items)     # returns a new sorted list
print(result)              # [1, 1, 3, 4, 5]
print(items)               # [3, 1, 4, 1, 5] -- original unchanged
```

Gotcha 12: Float Precision

```
#  The trap -- floating-point arithmetic is not exact
print(0.1 + 0.2 == 0.3)   # False!
print(0.1 + 0.2)            # 0.30000000000000004

#  Fix 1 -- math.isclose for approximate comparison
```

```
import math
print(math.isclose(0.1 + 0.2, 0.3))   # True

#  Fix 2 -- pytest.approx in tests
# assert 0.1 + 0.2 == pytest.approx(0.3)

#  Fix 3 -- decimal.Decimal for exact decimal arithmetic
from decimal import Decimal
print(Decimal("0.1") + Decimal("0.2") == Decimal("0.3"))   # True
```

Gotcha 13: Chained Comparisons Are Not What You Think

```
# Python chained comparisons are mathematical -- usually what you want
print(1 < 2 < 3)    # True  -- equivalent to (1 < 2) and (2 < 3)
print(1 < 2 > 1)    # True  -- equivalent to (1 < 2) and (2 > 1)
print(0 < x < 10)   # range check -- very Pythonic, use it

# But this surprises people from other languages
x = 2
print(1 < x < 3)    # True  -- in Python
# In C/Java: (1 < x) < 3 -> True < 3 -> 1 < 3 -> True (but for wrong reason!)

#  The trap -- chaining with 'in' and 'not in'
nums = [1, 2, 3]
print(1 in nums)         # True
print(4 not in nums)     # True
# 4 not in nums in [nums] -- this chains! (4 not in nums) and (nums in [nums])
```

Gotcha 14: Augmented Assignment on Immutable Types

```
#  The apparent contradiction
t = (1, 2, 3)
try:
    t += (4, 5)    # tuples are immutable -- should fail, right?
except TypeError:
    print("TypeError!")
# Actually succeeds! t now refers to a NEW tuple (1,2,3,4,5)
```

```python
# t is rebound, not mutated

t = (1, 2, 3)
original_id = id(t)
t += (4, 5)
print(t)                        # (1, 2, 3, 4, 5)
print(id(t) == original_id)     # False -- t is a NEW object

# In contrast, list += mutates IN PLACE
lst = [1, 2, 3]
original_id = id(lst)
lst += [4, 5]
print(id(lst) == original_id)  # True -- same object, mutated

#  The REAL trap -- tuple containing a mutable object
t = ([1, 2], [3, 4])
try:
    t[0] += [5]    # TypeError: 'tuple' object does not support item assignment
except TypeError as e:
    print(f"  Error: {e}")
# But! The += actually ran before the assignment failed:
print(t[0])    # [1, 2, 5] -- list WAS modified!
# This is a known Python quirk: the augmented op on the list succeeds,
# then the tuple assignment fails -- leaving the list modified.
```

Quick Reference: The Fourteen Gotchas

#	Gotcha	The Fix
1	Mutable default argument	Use `None`; create inside function
2	Late binding closures	Default arg `i=i` or `functools.partial`
3	`is` vs `==`	`==` for values; `is` only for singletons
4	`UnboundLocalError`	Pass as parameter; use `global` if needed
5	Exception variable deleted	Save to different name in `except` block

#	Gotcha	The Fix
6	Class vs instance variable	Reference `ClassName.attr` explicitly
7	Modify during iteration	List comprehension; iterate a copy
8	Generator exhaustion	Wrap in function; use list if multi-use
9	Single-element tuple	Trailing comma: `(x,)`
10	`None` is not `False`	Use `is None`; use `is False` for bool checks
11	`sort()` returns `None`	Use `sorted()` to get a new sorted list
12	Float precision	`math.isclose()`; `Decimal` for exactness
13	Chained comparisons	They work as math – but `in` chains can surprise
14	Augmented assign on immutable	Rebinds the variable; tuple mutation quirk

---

Interview Lens

Q1: What does this print, and why?

```
def f(x, data=[]):
    data.append(x)
    return data

print(f(1))
print(f(2))
print(f(3))
```

Answer: `[1]`, `[1, 2]`, `[1, 2, 3]`. The default `[]` is created once at function definition and shared across calls. Fix: use `data=None` and create a new list inside.

Q2: What does this print?

```
funcs = [lambda x: x + i for i in range(3)]
print([f(0) for f in funcs])
```

Answer: `[2, 2, 2]`. Late binding: all lambdas close over the same `i`, which is 2 after the loop completes. Fix: `lambda x, i=i: x + i`.

Q3: What is the output?

```
x = 10
def f():
    print(x)
    x = 20
f()
```

Answer: `UnboundLocalError`. The assignment `x = 20` makes `x` local to `f()` throughout the function body – including before the assignment. The `print(x)` tries to read a local `x` that is not yet assigned.

Q4: What does this print?

```
try:
    raise ValueError("oops")
except ValueError as e:
    saved = e

print(type(saved))
```

Answer: `<class 'ValueError'>`. `saved` is assigned inside the except block before `e` is deleted, so `saved` persists and refers to the exception object. If you had tried to use `e` outside the block, it would be `NameError`.

Q5: What is the output?

```
lst = [3, 1, 4]
result = lst.sort()
print(result, lst)
```

Answer: `None [1, 3, 4]`. `list.sort()` sorts in place and returns `None`. Use `sorted(lst)` to get the sorted list as a return value.

Knowing the gotchas cold prevents you from writing bugs under interview pressure – and signals deep Python familiarity to interviewers. The next chapter provides the Top 50 Python Interview Questions and Answers.

End of Free Sample

You just read **Chapter 32: Python Gotchas** from *Python Under Pressure*.

The full book includes:

- **35 chapters** across 7 parts
- CPython internals, GIL, memory model, descriptors, metaclasses
- NumPy vectorization, asyncio, pybind11, mypy –strict
- Complete DSA – arrays, linked lists, trees, heaps, dynamic programming
- 16 design patterns + system design
- **50 interview Q&A with senior-level answers**
- Live coding strategy and Junior/Senior/Staff rubrics

Get the full book:

leanpub.com/python-under-pressure

Ahmed Aly – Technical Lead · Software & ML/AI Systems · 15+ Years