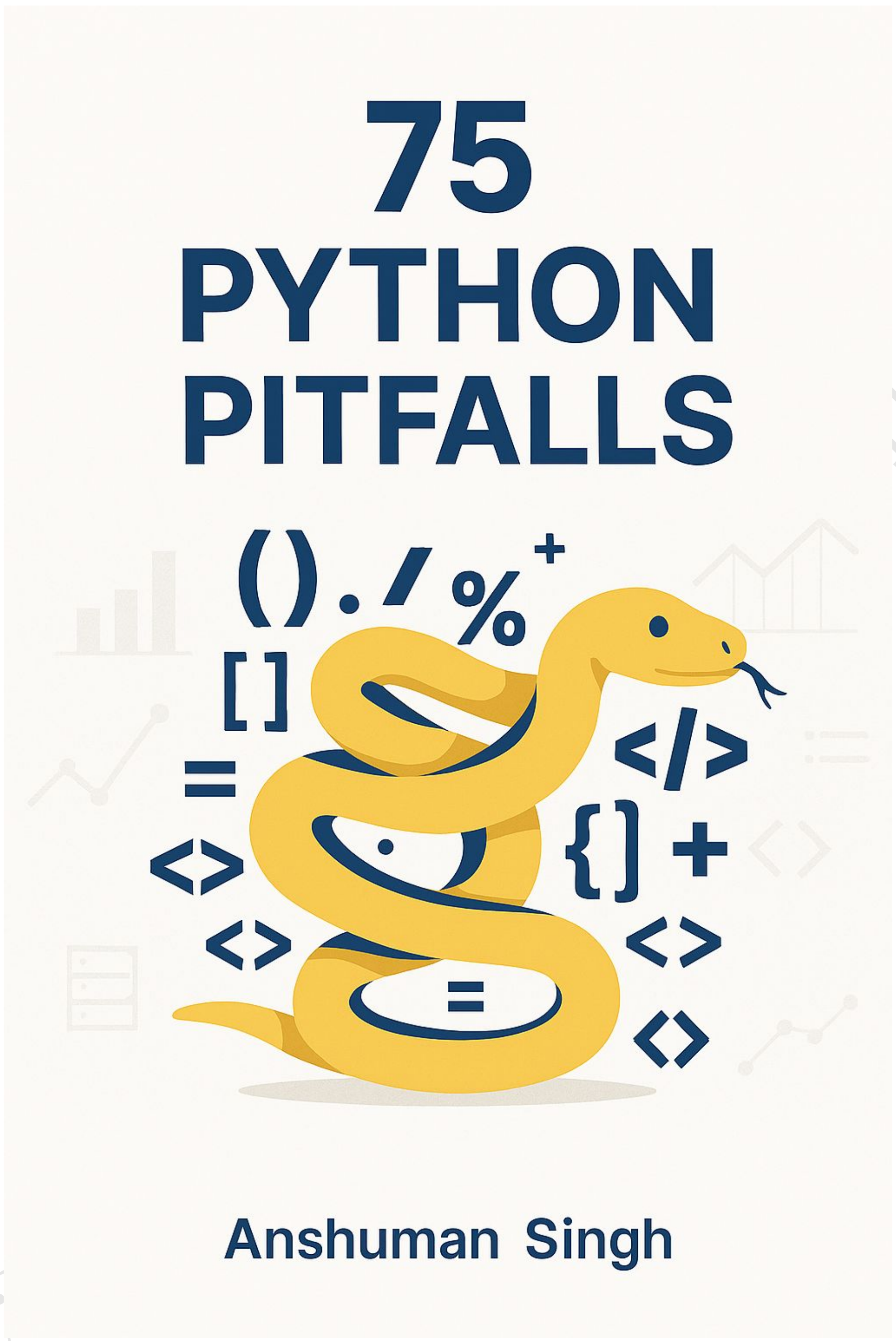

75 Python Pitfalls

Why Beginners Stumble and How to Avoid Them

Anshuman Singh

November 1, 2025

Contents

Chapter 1

Introduction: Welcome to Python Pitfalls

Problem

Why This Book? Python is beautiful but full of hidden traps! This book exposes 75 common mistakes that frustrate beginners. Each problem is designed to make you think and learn the "why" behind Python's behavior.

1.1 How to Use This Book

- **Try Each Problem First** - Don't peek at solutions immediately!
- **Understand the Why** - It's not about memorizing, but understanding
- **Experiment** - Modify the code and see what happens
- **Check Solutions** - Detailed explanations at: `https://anshuman365.github.io/assets/pdf/75_problem_solution.pdf`

Think About This

Pro Tip: The best way to learn is to make mistakes and understand why they happen. Embrace the confusion - it means you're learning!

Chapter 2

The Syntax Trap: Where Beginners Get Stuck

2.1 Mutable Default Arguments

Problem

Problem #1: The Mysterious Growing List

What's wrong with this function? Why does it behave unexpectedly?

```python
def add_item(item, items=[]):
    items.append(item)
    return items

print(add_item(1))  # Output: [1]
print(add_item(2))  # Output: [1, 2] - Wait, what?!
print(add_item(3))  # Output: [1, 2, 3] - It remembers!
```

Expected: Each call should return a new list with one item.
Actual: The list keeps growing between calls!

Think About This

Think about when Python evaluates default arguments. Is it every time the function is called, or just once?

2.2 Variable Scope Confusion

Problem

Problem #2: The Ghost Variable

Why does this code raise an UnboundLocalError when it seems like ‘x‘ should be accessible?

```python
x = 10  # Global variable

def modify():
    print(x)  # UnboundLocalError: local variable 'x' referenced before assignment
    x = 5     # This assignment makes x local to the function

modify()
```

Error: UnboundLocalError: local variable ’x’ referenced before assignment

Think About This

Python decides variable scope at compile time, not runtime. The mere presence of an assignment makes a variable local to the function!

2.3 String Comparison Surprise

Problem

Problem #3: Identical But Different Strings

These strings look identical but compare differently. Why?

```python
# These look the same but...
s1 = "cafe"         # Using single character
s2 = "cafe"         # Using combining character

print("String 1:", s1)
print("String 2:", s2)
print("Are they equal?", s1 == s2)  # False - why?
print("Length s1:", len(s1), "Length s2:", len(s2))  # 4 vs 5
```

Observation: Both strings look identical when printed but have different lengths and don’t compare as equal.

2.4 Integer Identity Puzzle

Problem

Problem #4: The Integer Identity Mystery

Why do small integers behave differently from large integers with the ‘is‘ operator?

```python
a = 256
b = 256
print(a is b)  # True

x = 257
y = 257
print(x is y)  # False - Wait, why?
```

Question: Why does the behavior change at 256?

Think About This

Python caches small integers for performance. The exact range may vary between implementations!

2.5 List Comprehension Scope

Problem

Problem #5: The Leaking Variable

What happens to the variable ‘x‘ after this list comprehension?

```python
x = "hello"
numbers = [x for x in range(5)]
print(x)  # What will this print?
```

Think: Does the list comprehension create a new scope for ‘x‘?

2.6 Boolean Operator Quirk

Problem

Problem #6: The AND-OR Surprise

What does this expression evaluate to?

```python
result = True or False and False
print(result)  # True or False?
```

Remember: Python operator precedence might surprise you!

2.7 Chained Comparison

Problem

Problem #7: Mathematical Looking Comparisons

Is this valid Python? What does it evaluate to?

```python
x = 5
result = 1 < x < 10
print(result)  # True or False?
```

Question: Does Python support mathematical-style chained comparisons?

2.8 Multiple Assignment

Problem

Problem #8: The Swapping Trick

How does this swap work without a temporary variable?

```python
a = 1
b = 2
a, b = b, a
print(a, b)  # 2, 1 - How?
```

Think: What's happening behind the scenes with tuple packing/unpacking?

Chapter 3

Data Structure Disasters

3.1 List Mutation in Loop

Problem

Problem #9: The Disappearing Items

Why do some items get skipped when modifying a list while iterating?

```python
numbers = [1, 2, 3, 4, 5, 6]

# Try to remove all even numbers
for i, num in enumerate(numbers):
    if num % 2 == 0:
        del numbers[i]  # Dangerous!

print(numbers)  # Output: [1, 3, 5] - Wait, 4 is still there?
# Actually it's [1, 3, 5, 6] - 6 wasn't removed!
```

Expected: [1, 3, 5]
Actual: [1, 3, 5, 6] - The last even number wasn't removed!

Think About This

When you delete an item, the list shifts left, but the loop counter keeps moving right. This causes items to be skipped!

3.2 Dictionary Key Mutability

Problem

Problem #10: The Unhashable Type Error

Why can't we use lists as dictionary keys, but tuples work fine?

```python
# This works:
valid_dict = {(1, 2): "tuple key"}  # OK

# This fails:
invalid_dict = {[1, 2]: "list key"}  # TypeError: unhashable type: 'list'

print(valid_dict)
```

Error: TypeError: unhashable type: 'list'

3.3 Shallow Copy Surprise

Problem

Problem #11: The Connected Lists

Why does modifying one list affect the other when I clearly made a copy?

```python
original = [[1, 2], [3, 4]]
copy = original[:]  # Make a slice copy

# Modify the copy
copy[0][0] = 999

print("Original:", original)  # [[999, 2], [3, 4]]
print("Copy:", copy)          # [[999, 2], [3, 4]]
# Both changed! Why?
```

Observation: Changing the copy also changed the original!

3.4 Set Uniqueness

Problem

Problem #12: The Duplicate Mystery

Why does this set have fewer items than expected?

```python
numbers = [1, 1, 2, 2, 3, 3, True, False, 1.0, 1]
unique_set = set(numbers)
print(unique_set)  # {0, 1, 2, 3} - Wait, where are the others?
print(len(unique_set))  # 4, not 10!
```

Question: Why are True, 1, and 1.0 considered the same in a set?

3.5 Dictionary Order

Problem

Problem #13: The Ordered Dictionary

Are dictionaries guaranteed to maintain insertion order?

```python
d = {}
d['a'] = 1
d['b'] = 2
d['c'] = 3

print(list(d.keys()))  # What order?
```

Think: Is this behavior consistent across all Python versions?

3.6 Default Dictionary

Problem

Problem #14: The Missing Key

How can we avoid KeyError when accessing missing dictionary keys?

```python
d = {'a': 1, 'b': 2}
print(d['c'])  # KeyError: 'c'

# What's a better way?
```

Solution: Explore defaultdict or get() method!

3.7 List vs Tuple

Problem

Problem #15: The Mutable Tuple

Wait, can tuples really be changed?

```python
my_tuple = (1, 2, [3, 4])
my_tuple[2].append(5)
print(my_tuple)  # (1, 2, [3, 4, 5]) - It changed!
```

Question: Is the tuple really immutable?

3.8 String Concatenation

Problem

Problem #16: The Expensive Operation

Why is this considered inefficient?

```python
result = ""
for i in range(1000):
    result += str(i)
```

Think: What happens in memory with each concatenation?

Chapter 4

Function Frustrations

4.1 Late Binding Closures

Problem

Problem #17: The Loop Variable Trap

Why do all functions in this list return the same value?

```
functions = []
for i in range(3):
    functions.append(lambda: i)

# What will these return?
print(functions[0]())  # 2
print(functions[1]())  # 2
print(functions[2]())  # 2 - All return 2!
```

Expected: 0, 1, 2
Actual: 2, 2, 2 - All functions return the final value of i!

Think About This

Python closures capture variables by name, not by value. All lambdas see the same 'i' variable at its final value!

4.2 Argument Unpacking Confusion

Problem

Problem #18: The Asterisk Anxiety

What's the difference between *args and **kwargs, and when to use which?

```python
def simple_func(a, b, c):
    return a + b + c

numbers = [1, 2, 3]
# These two calls are equivalent:
result1 = simple_func(*numbers)  # Unpacks list
result2 = simple_func(1, 2, 3)   # Direct arguments

# But this fails:
def wrapper(*args):
    return simple_func(args)  # TypeError: missing 2 required positional arguments
```

Error: Understand the difference between passing a tuple and unpacking it.

4.3 Function Attribute

Problem

Problem #19: The Function Object

Can functions have attributes like objects?

```python
def my_func():
    return "hello"

my_func.counter = 0
my_func.counter += 1
print(my_func.counter)  # 1 - It works!
```

Question: Is this a good practice?

4.4 Generator Exhaustion

Problem

Problem #20: The One-Time Use Generator

Why does this generator work only once?

```python
def squares(n):
    for i in range(n):
        yield i * i

gen = squares(3)
print(list(gen))  # [0, 1, 4]
print(list(gen))  # [] - Empty! Why?
```

Think: What happens to a generator after it's been exhausted?

4.5 Decorator Timing

Problem

Problem #21: The Early Execution

When are decorators executed?

```python
def my_decorator(func):
    print("Decorator executed!")
    return func

@my_decorator
def my_function():
    print("Function called")

# When is "Decorator executed!" printed?
```

Question: At import time or when the function is called?

Chapter 5

Object-Oriented Oddities

5.1 Class vs Instance Variables

Problem

Problem #22: The Shared Variable

Why are all instances sharing the same list?

```python
class MyClass:
    items = []  # Class variable

    def add_item(self, item):
        self.items.append(item)

obj1 = MyClass()
obj2 = MyClass()

obj1.add_item(1)
obj2.add_item(2)

print(obj1.items)  # [1, 2] - Shared!
print(obj2.items)  # [1, 2] - Shared!
```

Expected: Each instance should have its own list
Actual: All instances share the same list!

5.2 Method Binding

Problem

Problem #23: The Self Surprise

What happens when we forget self?

```python
class MyClass:
    def __init__(self):
        self.value = 10

    def get_value():  # Forgot self!
        return self.value

obj = MyClass()
print(obj.get_value())  # TypeError!
```

Error: TypeError: get_value() takes 0 positional arguments but 1 was given

5.3 Private Variables

Problem

Problem #24: The "Private" Myth

Are double-underscore variables really private?

```python
class MyClass:
    def __init__(self):
        self.__secret = "hidden"

    def get_secret(self):
        return self.__secret

obj = MyClass()
print(obj._MyClass__secret)  # "hidden" - Not so private!
```

Question: Is there true privacy in Python?

5.4 Inheritance MRO

Problem

Problem #25: The Diamond Problem

Which method gets called in multiple inheritance?

```python
class A:
    def method(self):
        return "A"

class B(A):
    def method(self):
        return "B"

class C(A):
    def method(self):
        return "C"

class D(B, C):
    pass

obj = D()
print(obj.method())  # "B" or "C"?
```

Think: Understand Python's Method Resolution Order (MRO)!

Chapter 6

Module Mayhem

6.1 Circular Imports

Problem

Problem #26: The Import Loop

What happens with circular imports?

```python
# module_a.py
import module_b
def func_a():
    return module_b.func_b()

# module_b.py
import module_a
def func_b():
    return module_a.func_a()
```

Question: Will this work? What error might occur?

6.2 Name == Main

Problem

Problem #27: The Import vs Execution

Why do we use ‘if __name__ == "__main__"‘?

```python
# my_module.py
def important_function():
    print("This should always be available")

if __name__ == "__main__":
    print("This should run only when executed directly")
```

Think: What’s the difference between importing and executing a module?

Chapter 7

Exception Enigmas

7.1 Broad Except

Problem

Problem #28: The Silent Failure

What's wrong with this exception handling?

```python
try:
    risky_operation()
except:  # Bare except!
    print("Something went wrong")
```

Issue: This can catch even SystemExit and KeyboardInterrupt!

7.2 Else in Try

Problem

Problem #29: The Try-Else Mystery

When does the 'else' clause execute in try-except?

```python
try:
    result = safe_operation()
except ValueError:
    print("Value error occurred")
else:
    print("No exception occurred!")
```

Question: Is this different from putting code after the try-except?

7.3 Exception Message Capture

Problem

Problem #30: The Lost Exception Message

Why can't I access the exception message in the correct way?

```python
try:
    int("not_a_number")
except ValueError as e:
    print(e.message)  # AttributeError: 'ValueError' object has no attribute 'message'
```

Error: AttributeError: 'ValueError' object has no attribute 'message'

Think About This

In Python 3, exceptions don't have a .message attribute. Use str(e) or the args tuple instead!

7.4 Finally vs Return

Problem

Problem #31: The Final Override

Which value does this function return?

```python
def test_function():
    try:
        return "from try"
    finally:
        return "from finally"

result = test_function()
print(result)  # "from finally" - The return in finally wins!
```

Question: Why does the finally block override the return value?

Chapter 8

Advanced Python Pitfalls

8.1 Iterators vs Iterables

Problem

Problem #32: The Exhausted Iterator

Why does this work only once?

```python
numbers = [1, 2, 3]
iterator = iter(numbers)

print(list(iterator))  # [1, 2, 3]
print(list(iterator))  # [] - Empty! Why?
```

Think: What's the difference between an iterable and an iterator?

Think About This

An iterable can be iterated over multiple times, but an iterator can only be used once!

8.2 Context Manager Exception

Problem

Problem #33: The Silent Context Failure

What happens if both the context manager and the block raise exceptions?

```python
class MyContext:
    def __enter__(self):
        return self
    def __exit__(self, exc_type, exc_val, exc_tb):
        raise ValueError("Error in exit")

with MyContext():
    raise RuntimeError("Error in block")
```

Question: Which exception will be raised?

8.3 Metaclass Confusion

Problem

Problem #34: The Meta Inheritance

Why does this metaclass not affect the child class?

```python
class Meta(type):
    def __new__(cls, name, bases, dct):
        dct['meta_attr'] = 'from_meta'
        return super().__new__(cls, name, bases, dct)

class Base(metaclass=Meta):
    pass

class Child(Base):
    pass

print(hasattr(Child, 'meta_attr'))  # True or False?
```

Think: Do child classes inherit their parent's metaclass?

8.4 Descriptor Protocol

Problem

Problem #35: The Property vs Descriptor

Why does this descriptor behave differently from a property?

```python
class MyDescriptor:
    def __get__(self, obj, objtype=None):
        return "descriptor value"

class MyClass:
    desc = MyDescriptor()
    @property
    def prop(self):
        return "property value"

obj = MyClass()
print(obj.desc)    # "descriptor value"
print(obj.prop)    # "property value"
print(MyClass.desc) # "descriptor value" - Wait, called without instance!
```

Observation: Descriptors work on classes too!

8.5 Async/Await Gotcha

Problem

Problem #36: The Forgotten Await

Why doesn't this coroutine run?

```python
import asyncio

async def slow_operation():
    await asyncio.sleep(1)
    return "done"

# This just returns a coroutine object, doesn't run it
result = slow_operation()
print(result)  # <coroutine object slow_operation at 0x...>
```

Error: Forgot to await or run the coroutine!

Think About This

Coroutines are just objects until you await them or run them in an event loop!

8.6 Threading Race Condition

Problem

Problem #37: The Counter Race

Why doesn't this counter reach the expected value?

```python
import threading

counter = 0

def increment():
    global counter
    for _ in range(100000):
        counter += 1

threads = []
for i in range(10):
    t = threading.Thread(target=increment)
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print(counter)  # Less than 1000000 - Why?
```

Expected: 1000000
Actual: Some smaller number due to race conditions!

8.7 GIL Misunderstanding

Problem

Problem #38: The Global Interpreter Lock

Does the GIL prevent all threading issues?

```python
import threading

# This is still not thread-safe despite the GIL!
shared_list = []

def append_numbers():
    for i in range(1000):
        shared_list.append(i)

threads = []
for i in range(10):
    t = threading.Thread(target=append_numbers)
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print(len(shared_list))  # Might be less than 10000
```

Question: Why isn't the GIL enough to make this thread-safe?

8.8 Memory Management

Problem

Problem #39: The Memory Leak

Why does this code keep growing in memory?

```python
import weakref

class Data:
    def __init__(self, value):
        self.value = value
        self.cache = []

def process_data():
    data = Data("hello")
    # Circular reference!
    data.cache.append(data)
    return data

# Each call creates data that can't be garbage collected
for i in range(100000):
    process_data()
```

Issue: Circular references prevent garbage collection!

8.9 Garbage Collection Timing

Problem

Problem #40: The Del Mystery

When is __del__ actually called?

```python
class Resource:
    def __init__(self, name):
        self.name = name
    def __del__(self):
        print(f"Cleaning up {self.name}")

def create_and_forget():
    r = Resource("temporary")
    # r goes out of scope here

create_and_forget()
print("Function finished")
# When is "Cleaning up temporary" printed?
```

Question: Is __del__ called immediately when object goes out of scope?

8.10 Pickling Limitations

Problem

Problem #41: The Unpicklable Object

Why can't I pickle this object?

```python
import pickle

def outer_function():
    x = 10
    def inner_function():
        return x
    return inner_function

my_func = outer_function()

# This fails!
pickled = pickle.dumps(my_func)  # PicklingError!
```

Error: Can't pickle local functions or closures!

8.11 Serialization Alternatives

Problem

Problem #42: The JSON Serialization

Why can't JSON serialize this Python object?

```python
import json
from datetime import datetime

data = {
    'time': datetime.now(),
    'set': {1, 2, 3},
    'complex': 1 + 2j
}

json_str = json.dumps(data)  # TypeError!
```

Error: JSON only supports basic types: str, int, float, bool, None, dict, list

8.12 Data Classes Defaults

Problem

Problem #43: The Mutable Default in Data Class

Why does this data class have shared lists?

```python
from dataclasses import dataclass

@dataclass
class Inventory:
    items: list = []  # Dangerous mutable default!

inv1 = Inventory()
inv2 = Inventory()

inv1.items.append("sword")
print(inv2.items)  # ['sword'] - Shared!
```

Issue: Same mutable default problem as regular functions!

8.13 Type Hints Runtime

Problem

Problem #44: The Runtime Type Ignorance

Do type hints affect runtime behavior?

```python
def add_numbers(a: int, b: int) -> int:
    return a + b

# This works fine at runtime despite type violation
result = add_numbers("hello", "world")
print(result)  # "helloworld"
```

Observation: Type hints are just hints - no runtime enforcement!

8.14 Walrus Operator Scope

Problem

Problem #45: The Assignment Expression Scope

What's the scope of walrus operator variables?

```python
# This works:
if (n := len([1,2,3])) > 2:
    print(f"Length is {n}")

# But what about here?
results = [x for x in range(10) if (y := x % 2) == 0]
print(y)  # What is y here? Does it exist?
```

Question: Do assignment expressions leak into outer scope?

Chapter 9

Expert Level Pitfalls

9.1 Decorator Parameters

Problem

Problem #46: The Decorator Factory Confusion

Why does this decorator with parameters not work?

```python
def decorator_with_args(arg1, arg2):
    def real_decorator(func):
        def wrapper(*args, **kwargs):
            print(f"Decorator args: {arg1}, {arg2}")
            return func(*args, **kwargs)
        return wrapper
    return real_decorator

# Wrong usage:
@decorator_with_args
def my_function():
    pass

# Correct usage:
@decorator_with_args("hello", "world")
def my_function():
    pass
```

Error: Missing parentheses when decorator takes arguments!

9.2 Class Decorators

Problem

Problem #47: The Class Modification Decorator

How can a decorator modify a class?

```python
def add_method(cls):
    def new_method(self):
        return "added by decorator"
    cls.new_method = new_method
    return cls

@add_method
class MyClass:
    def original_method(self):
        return "original"

obj = MyClass()
print(obj.new_method())  # "added by decorator"
```

Think: Decorators can work on classes too, not just functions!

9.3 Method Decorators

Problem

Problem #48: The Staticmethod Decorator Order

Why does this decorator combination not work?

```python
class MyClass:
    @staticmethod
    @my_decorator  # This order is wrong!
    def my_method():
        return "hello"
```

Issue: @staticmethod must be the outermost decorator!

Think About This

The decorator closest to the function is applied first. @staticmethod needs to see the undecorated function!

9.4 Property Setters

Problem

Problem #49: The Property Validation

Why doesn't this property setter get called?

```python
class BankAccount:
    def __init__(self):
        self._balance = 0

    @property
    def balance(self):
        return self._balance

    # Forgot to create the setter!
    # @balance.setter
    # def balance(self, value):
    #     if value < 0:
    #         raise ValueError("Balance cannot be negative")
    #     self._balance = value

account = BankAccount()
account.balance = -100  # This works! No validation.
```

Error: Without a setter, the property becomes read-only but assignment creates a new instance attribute!

9.5 Abstract Base Classes

Problem

Problem #50: The Abstract Method Enforcement

When are abstract methods checked?

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def speak(self):
        pass

# This fails only when instantiated, not when defined
class Dog(Animal):
    pass  # Forgot to implement speak()

dog = Dog()  # TypeError: Can't instantiate abstract class Dog
```

Observation: Abstract method checking happens at instantiation time, not class definition time!

9.6 Protocol Classes

Problem

Problem #51: The Structural Typing

What's the difference between Protocol and ABC?

```python
from typing import Protocol

class Flyer(Protocol):
    def fly(self) -> str: ...

class Bird:
    def fly(self):
        return "flying"

class Airplane:
    def fly(self):
        return "flying high"

def make_it_fly(f: Flyer) -> str:
    return f.fly()

# Both work without explicit inheritance!
print(make_it_fly(Bird()))
print(make_it_fly(Airplane()))
```

Think: Protocols use structural typing (duck typing) while ABCs use nominal typing!

9.7 Monkey Patching

Problem

Problem #52: The Surprising Method Addition

Can I add methods to existing classes?

```python
class Original:
    def existing_method(self):
        return "original"

def new_method(self):
    return "monkey patched"

# Monkey patching!
Original.new_method = new_method

obj = Original()
print(obj.new_method())  # "monkey patched"
```

Question: Is monkey patching a good practice?

9.8 Metaprogramming Magic

Problem

Problem #53: The Dynamic Class Creation

How can I create classes dynamically?

```python
def create_class(class_name, base_classes, attributes):
    return type(class_name, base_classes, attributes)

# Creates a class dynamically!
MyClass = create_class('MyClass', (), {'x': 10, 'method': lambda self: self.x})

obj = MyClass()
print(obj.method())  # 10
```

Think: Classes are created by the type metaclass - we can use this directly!

9.9 Import Hooks

Problem

Problem #54: The Custom Importer

Can I customize how Python imports modules?

```python
import sys
import types

class VirtualImporter:
    def find_module(self, fullname, path=None):
        if fullname == "virtual_module":
            return self
        return None

    def load_module(self, fullname):
        module = types.ModuleType(fullname)
        module.message = "Hello from virtual module!"
        sys.modules[fullname] = module
        return module

# Install the hook
sys.meta_path.append(VirtualImporter())

# Now we can import our virtual module!
import virtual_module
print(virtual_module.message)
```

Advanced: You can completely customize Python's import system!

9.10 Sys.path Manipulation

Problem

Problem #55: The Runtime Path Modification

Can I add to Python's import path at runtime?

```python
import sys
import os

# Add a directory to the import path
new_path = "/path/to/my/modules"
if new_path not in sys.path:
    sys.path.append(new_path)

# Now I can import from that directory
import my_custom_module
```

Warning: This affects the entire Python process - use carefully!

9.11 Dynamic Attribute Access

Problem

Problem #56: The Magic Getattr

How can I handle missing attributes dynamically?

```python
class DynamicAttributes:
    def __getattr__(self, name):
        if name.startswith("fake_"):
            return f"dynamically created: {name}"
        raise AttributeError(f"'{self.__class__.__name__}' object
   has no attribute '{name}'")

obj = DynamicAttributes()
print(obj.fake_attr)  # "dynamically created: fake_attr"
print(obj.real_attr)  # AttributeError
```

Think: __getattr__ is only called for missing attributes!

9.12 Getattr vs Getattribute

Problem

Problem #57: The Infinite Recursion

Why does this code crash with recursion?

```python
class BadClass:
    def __getattribute__(self, name):
        # This causes infinite recursion!
        return self.__dict__[name]

obj = BadClass()
obj.x = 10
print(obj.x)  # RecursionError!
```

Error: Accessing self.__dict__ inside __getattribute__ calls __getattribute__ again!

Think About This

Use super().__getattribute__('__dict__') inside __getattribute__ to avoid recursion!

9.13 Slots Usage

Problem

Problem #58: The Memory Optimization

What's the trade-off with __slots__?

```python
class WithSlots:
    __slots__ = ['x', 'y']
    def __init__(self, x, y):
        self.x = x
        self.y = y

obj = WithSlots(1, 2)
obj.z = 3  # AttributeError: 'WithSlots' object has no attribute 'z'
```

Trade-off: __slots__ saves memory but prevents dynamic attribute creation!

9.14 Weak References

Problem

Problem #59: The Circular Reference Breaker

How can I break circular references?

```python
import weakref

class Node:
    def __init__(self, value):
        self.value = value
        self.parent = None  # Will be a weak reference

parent = Node("parent")
child = Node("child")

# Use weakref to avoid circular reference
child.parent = weakref.ref(parent)

# Now parent can be garbage collected even if child exists
```

Solution: Weak references don't prevent garbage collection!

9.15 Cyclic References

Problem

Problem #60: The Garbage Collection Savior

How does Python handle cyclic references?

```python
import gc
import sys

class A:
    def __init__(self):
        self.b = None

class B:
    def __init__(self):
        self.a = None

a = A()
b = B()
a.b = b
b.a = a  # Cyclic reference!

# Delete references
del a
del b

# Will they be collected?
collected = gc.collect()
print(f"Collected {collected} objects")
```

Answer: The cyclic garbage collector can handle this!

9.16 Del Method Dangers

Problem

Problem #61: The Resurrection Danger

Can an object come back to life during __del__?

```python
class Zombie:
    zombies = []
    def __del__(self):
        Zombie.zombies.append(self)  # Resurrecting the object!
        print("I'm being deleted... but coming back!")

obj = Zombie()
del obj  # Object is "deleted" but added to zombies list
print(f"Zombies: {len(Zombie.zombies)}")  # 1 - It's alive!
```

Warning: Don't resurrect objects in __del__ - it leads to undefined behavior!

9.17 Interpreter Shutdown

Problem

Problem #62: The Module Cleanup Race

What happens during interpreter shutdown?

```python
import atexit

def cleanup():
    print("Cleaning up...")

atexit.register(cleanup)

# During interpreter shutdown, module globals might be None
import sys
sys.exit(0)  # What order do things happen in?
```

Complexity: Interpreter shutdown has complex ordering dependencies!

9.18 C Extensions

Problem

Problem #63: The Reference Counting

How do C extensions manage memory?

```python
# C extension code example (conceptual)
"""
PyObject* create_list() {
    PyObject* list = PyList_New(0);
    // Must manage reference counts correctly!
    Py_INCREF(list);  // If we're returning it
    return list;
}
"""
```

Challenge: Manual reference counting in C extensions is error-prone!

9.19 FFI Issues

Problem

Problem #64: The C Data Conversion

What happens with C data types in Python?

```python
from ctypes import *

# C function that returns a string
libc = CDLL("libc.so.6")
getenv = libc.getenv
getenv.restype = c_char_p

result = getenv(b"HOME")
print(result)  # b'/home/user' - bytes, not str!
```

Issue: C interfaces often return bytes that need decoding!

9.20 Performance Profiling

Problem

Problem #65: The Wrong Optimization Target

How do I find what to optimize?

```python
import time

def slow_function():
    total = 0
    for i in range(1000000):
        total += i
    return total

# Time the whole function
start = time.time()
result = slow_function()
end = time.time()
print(f"Time: {end - start:.4f}s")
```

Better: Use cProfile to find the real bottlenecks!

9.21 Memory Profiling

Problem

Problem #66: The Memory Bloat

How do I find memory leaks?

```python
def create_big_list():
    return [i for i in range(1000000)]

def process_data():
    data = create_big_list()
    # Forgot to clean up!
    return sum(data)

result = process_data()
# The big list is still in memory until garbage collected
```

Solution: Use memory_profiler or tracemalloc to track memory usage!

9.22 Debugging Techniques

Problem

Problem #67: The Post-Mortem Debugging

How can I debug crashed programs?

```python
import pdb

def buggy_function():
    x = 1
    y = 0
    return x / y  # ZeroDivisionError

# Set trace to enter debugger on error
try:
    buggy_function()
except:
    pdb.post_mortem()  # Debug the traceback!
```

Technique: Post-mortem debugging lets you inspect the state when an error occurred!

9.23 Testing Corner Cases

Problem

Problem #68: The Boundary Condition

What edge cases should I test?

```
def divide(a, b):
    return a / b

# Test these cases:
# divide(1, 1)     # Normal case
# divide(1, 0)     # Division by zero
# divide(0, 1)     # Zero numerator
# divide(-1, 1)    # Negative numbers
# divide(1, -1)    # Negative denominator
# divide(1.5, 2.5) # Floats
```

Advice: Always test boundary conditions and error cases!

9.24 Packaging Issues

Problem

Problem #69: The Relative Import

Why does this import work in development but not when packaged?

```
# my_package/
#   __init__.py
#   module_a.py
#   module_b.py

# In module_a.py:
from .module_b import some_function  # Relative import

# This works when package is installed, but not when running
    module_a directly!
```

Issue: Relative imports only work within a package!

9.25 Virtual Environment Problems

Problem

Problem #70: The Wrong Python Interpreter

Why can't my script find the installed package?

```python
#!/usr/bin/env python3
import requests  # ModuleNotFoundError even though it's installed!

# Check which Python is being used:
import sys
print(sys.executable)  # Might be system Python, not virtualenv
    Python
```

Solution: Always activate your virtual environment!

9.26 Dependency Conflicts

Problem

Problem #71: The Version Hell

Why do I get different behavior on different machines?

```
# requirements.txt
package-a==1.0
package-b==2.0
# package-b requires package-a>=1.5, but we pinned 1.0!

# This creates a conflict that pip may or may not resolve
```

Problem: Dependency version conflicts are common in complex projects!

9.27 Version Compatibility

Problem

Problem #72: The Python Version Check

How do I write code that works across Python versions?

```python
import sys

if sys.version_info >= (3, 8):
    # Use walrus operator
    if (n := len(data)) > 10:
        print(f"Big data: {n}")
else:
    # Fallback for older Python
    n = len(data)
    if n > 10:
        print(f"Big data: {n}")
```

Best Practice: Always check Python version for version-specific features!

9.28 Cross-Platform Issues

Problem

Problem #73: The Path Separator

Why does my file path work on Windows but not Linux?

```python
# Windows style path (fails on Linux)
path = "C:\\Users\\name\\file.txt"

# Platform-independent path
import os
path = os.path.join("Users", "name", "file.txt")
```

Solution: Always use os.path.join for cross-platform compatibility!

9.29 Unicode Handling

Problem

Problem #74: The Encoding Nightmare

Why do I get Unicode errors with file I/O?

```python
# This may fail with non-ASCII characters
with open("file.txt", "r") as f:
    content = f.read()  # UnicodeDecodeError!

# Always specify encoding
with open("file.txt", "r", encoding="utf-8") as f:
    content = f.read()  # Safe!
```

Rule: Always specify encoding when reading/writing text files!

9.30 The Final Pitfall

Problem

Problem #75: The Over-Engineering Trap

Am I solving the right problem?

```python
# Complex solution:
from abc import ABC, abstractmethod
from typing import Protocol, Generic, TypeVar

T = TypeVar('T')

class ComplexSystem(ABC, Generic[T]):
    @abstractmethod
    def process(self, data: T) -> T: ...

# Simple solution:
def process_data(data):
    return data * 2  # Does the job!

# Sometimes the simplest solution is the best one!
```

Wisdom: Don't over-engineer! Solve the actual problem you have, not imaginary future problems.

Chapter 10

Conclusion and Next Steps

Problem

Congratulations! You've explored 75 common Python pitfalls. Remember, encountering these issues is a sign that you're learning and growing as a Python developer.

Get Complete Solutions!

Think About This

Detailed explanations, fixes, and deep dives for all 75 problems are available in the solutions PDF:

`https://anshuman365.github.io/assets/pdf/75_problem_solution.pdf`

Visit: `https://anshuman365.github.io/resources.html` for more Python resources!

Created by Anshuman Singh

10.1 Keep Learning

- Practice regularly and read Python code
- Contribute to open source projects
- Follow Python PEPs and documentation
- Join Python communities
- Never stop being curious!

Bibliography

[1] Python Documentation. *The Python Tutorial.* https://docs.python.org/3/tutorial/

[2] PEP 8 – Style Guide for Python Code. https://pep8.org/

[3] Ramalho, Luciano. *Fluent Python.* O'Reilly Media, 2022.

[4] Slatkin, Brett. *Effective Python.* Addison-Wesley, 2019.

[5] Real Python Tutorials. https://realpython.com

[6] Anshuman Singh. Programming Resources. https://anshuman365.github.io/