



Chapter 12

🧠 Chapter 12: Advanced Python 1 – Ultimate Notes & Explanations

1 Walrus Operator (`:=`)

✅ Concept:

The **Walrus operator** (`:=`) allows you to **assign a value** to a variable **as part of an expression** — combining assignment and condition check in one line.

🔍 Syntax:

```
if (n := len(my_list)) > 3:  
    print(f"List is too long ({n} elements)")
```

🔄 Old Way vs New Way:

```
# Old Way  
n = len(my_list)  
if n > 3:  
    ...  
  
# With Walrus  
if (n := len(my_list)) > 3:  
    ...
```

📄 Code:

```
# walrus.py
if(n := len([1,2,3,4,5])) > 3:
    print(f"List is too long({n} elements, expected < 3)")
else:
    print("List is shorter than 3 elements")
```

2 Type Hints

✓ Concept:

Type Hints provide **information to the reader and IDEs** about what type a variable/function expects. It doesn't enforce anything but makes the code easier to understand and maintain.

🔧 Syntax:

```
def func(a: int, b: int) → int:
    return a + b
```

📁 Code:

```
# typehints.py
from typing import Tuple

n: int = 5
name: str = "Prathamesh"

def sum(a: int, b: int) → int:
    return a + b

person: tuple[int, str] = ("Prathamesh", 100)

print(sum(5, 5))
print(person, type(person))
```

3 Match Case (Switch Statement in Python)

✓ Concept:

Python's `match-case` is similar to **switch-case** in other languages. It's more powerful, supporting **patterns**, **classes**, **destructuring**, etc.

⚠ **Works only in Python 3.10+.**

📁 Code:

```
# matchcase.py
def http_status(status: int):
    match status:
        case 200:
```

```

        return "OK"
    case 400:
        return "Bad Request"
    case 401:
        return "Unauthorised"
    case 404:
        return "Not Found"
    case 429:
        return "Too Many Requests"
    case 500:
        return "Internal Server Error"
    case _:
        return "Unknown Status"

print(http_status(200))    # OK
print(http_status(429))    # Too Many Requests
print(http_status(29378))  # Unknown Status

```

4 Dictionary Merge & Multiple File Context Managers

✓ Concept:

- Merging dictionaries using `|` operator ([Python 3.9+](#))
- Opening multiple files using **single** `with` **statement**

📁 Code:

```

# dictmerge_and_contextmanager.py
dict1 = {"a": 3, "b": 4}
dict2 = {"b": 2, "c": 7}
merged = dict1 | dict2
print(merged) # {'a': 3, 'b': 2, 'c': 7}

with open("file1.txt") as f1, open("file2.txt") as f2:
    a = f1.read()
    b = f2.read()
    print("Both files are same" if a == b else "Both files are different")

```

5 Exception Handling

✓ Concept:

Use `try`, `except` to **gracefully handle errors** and prevent the program from crashing.

📁 Code:

```

# exception.py
try:
    a = int(input("Enter a number: "))
    print(f"The number is {a}")

```

```
except ValueError:
    print("Please enter a valid number")
except Exception as e:
    print("Error:", e)

print("Thank you, this program has not crashed")
```

6 Raising Custom Exceptions

✓ Concept:

Use `raise` to **manually throw exceptions** when a condition violates logic or constraints.

📁 Code:

```
# raising_exception.py
a = int(input("Enter 1st Number: "))
b = int(input("Enter 2nd Number: "))

if b == 0:
    raise ZeroDivisionError("Division by zero is not fundamentally defined")
else:
    print(f"The value of a/b is {a/b}")
```

7 `try-except-else` Block

✓ Concept:

- `else` block runs **only if** `try` **succeeds**
- Helps to separate **normal flow** from **error flow**

📁 Code:

```
# try_else.py
try:
    a = int(input("Enter a number: "))
    print(f"The number is {a}")
except Exception as e:
    print(e)
else:
    print("Thank you, this program has not crashed")
```

8 `try-finally` Block

✓ Concept:

- The `finally` block **always runs**, whether an exception occurred or not.
- Useful for cleanup, closing files, etc.

Code:

```
# try_finally.py
def main():
    try:
        a = int(input("Enter a number: "))
        print(f"The number is {a}")
        return
    except Exception as e:
        print(e)
        return
    finally:
        print("I am finally, I always run")

main()
```

9 `__name__ == "__main__"` & Module Reuse

Concept:

- Helps differentiate if a file is **run directly** or **imported as a module**.
- Common for reusable scripts.

Code (module.py):

```
def name():
    print("I am class from module.py")

name()
print(__name__) # "__main__" if run directly

if __name__ == "__main__":
    print("We are directly running this code")
else:
    print("We are not directly running this code")
```

Code (main.py):

```
from module import name
name() # Output: I am class from module.py
```

10 `global` Keyword

Concept:

- Used to **declare a global variable** inside a function
- Allows updating it in global scope

Code:

```
# global.py
def fun():
    global a
    a = 434

fun()
print(a) # Output: 434
```

enumerate() Function

✓ Concept:

- Simplifies iterating over lists **with indices**

Code:

```
# enumerate.py
l = [3443, 45, 5, 4]

for index, item in enumerate(l):
    print(f"The number at index {index} is {item}")
```

List Comprehension

✓ Concept:

- Shorter syntax to **create new lists** using loops
- Very clean and Pythonic

Code:

```
# list_comprehension.py
myList = [2, 3, 4, 5, 6]
squaredList = [i * i for i in myList]
print(squaredList)
```

Chapter 12: Advanced Python 1 — Deep Dive Notes

1. Walrus Operator (:=)

| "Assignment expressions: saving one line at a time."

What Is It?

Introduced in **Python 3.8**, the **walrus operator** allows you to **assign a value** to a variable **within an expression**. Saves memory, boosts readability.

When To Use:

- In **loops** when filtering
- While checking the **result of a function**
- To **reduce redundancy**

Real-World Use:

```
while (line := input("Enter a line: ")) != "exit":
    print(f"You typed: {line}")
```

Pros:

- Removes redundant assignments
- Keeps logic compact

Cons:

- Can harm readability if overused in complex conditions

2. Type Hints (`typing` module)

| "Python becomes semi-typed — the best of both worlds."

What Are Type Hints?

Python is dynamically typed, but **type hints** (introduced in PEP 484) let you annotate functions and variables with **expected types**.

Example:

```
from typing import List, Tuple, Dict

def process(scores: List[int]) → float:
    return sum(scores) / len(scores)
```

Why Use?

- Improves **code readability**
- Helps with **editor autocomplete (IntelliSense)**
- Makes code easier to understand for teams

Tools That Use It:

- `mypy` for type checking
- `Pyright`, `Pylance` in VSCode
- Big in large-scale Python systems (ML pipelines, APIs)

3. Match Case (Pattern Matching)

| Python's take on switch-case, but more powerful!

What Is It?

Introduced in **Python 3.10**, it allows **structural pattern matching** — not just values, but **shapes and data structures**.

Deep Match Example:

```
def greet(person):
    match person:
        case {"name": str(name), "age": int(age)}:
            return f"Hello {name}, age {age}!"
```

Real Use:

- Processing **JSON or API responses**
- Handling **abstract syntax trees**
- Smart decision trees

4. Dictionary Merge (`|` operator)

| Python finally got dictionary union!

Syntax:

```
dict3 = dict1 | dict2
```

Behavior:

- Merges dicts
- If keys overlap, **right dict wins**

Advanced Use:

```
config = default_config | user_config | env_config
```

5. Context Managers (Multiple files)

| Clean resource management with `with` statement

Why Use?

Avoids:

- Open file locks
- Memory/resource leaks
- Manual `.close()`

Multi-context Example:


```
with open("a.txt") as a, open("b.txt") as b:  
    ...
```

✓ Custom Context Manager:

```
class Custom:  
    def __enter__(self): print("Entered")  
    def __exit__(self, *args): print("Exited")  
  
with Custom():  
    print("Inside block")
```

🔒 6. Exception Handling: try-except-else-finally

| Bulletproof your code

🔄 Why So Many Blocks?

- `try`: Run risky code
- `except`: Handle error
- `else`: Run if no error
- `finally`: Always run (cleanup/logging)

📌 Real Use:

- File operations
- Database queries
- Network calls

💥 Raising Custom Errors:

```
if not valid:  
    raise ValueError("Invalid Input!")
```

🧩 7. `__name__ == "__main__"` Idiom

| Makes Python scripts reusable as modules

🧠 Purpose:

When a Python file is run:

- Its `__name__` becomes `"__main__"`
- If it's imported, `__name__` is the file name

🔍 Best Practice:

```
def main():
    ...

if __name__ == "__main__":
    main()
```

8. Global Variable Usage

Carefully used for shared state

Global Pitfall:

Too much use of `global` leads to spaghetti code. But in **simple scripts or quick hacks**, it's fine.

```
def set_value():
    global config
    config = "dark-mode"
```



9. `enumerate()`

Elegant way to get index + value in a loop

```
for i, item in enumerate(["apple", "banana"]):
    print(i, item)
```

Clean Alternative To:

```
index = 0
for item in items:
    ...
    index += 1
```

10. List Comprehensions

Fast and readable list creation

Syntax:

```
squares = [x * x for x in range(10) if x % 2 == 0]
```

Advanced:

```
matrix = [[i * j for j in range(3)] for i in range(3)]
```

Bonus: Dict Comprehension

```
squared = {x: x * x for x in range(5)}
```



Real-World Usage Tips

Feature	Use-Case
Walrus Operator	Input loops, lazy evaluation
Type Hints	API design, large codebases
Match Case	Data routing, config parsing
Dict Merge	Chaining config or overrides
Context Managers	Database, file handling, sockets
try/except/else/finally	Error-proof systems
<code>__name__ == "__main__"</code>	Build CLI tools, test files
<code>global</code>	Lightweight config or state in scripts
<code>enumerate()</code>	Index-aware iteration (e.g., UI rendering)
List Comprehension	Data transformation, cleaning



Bonus Insights



In Interviews:

- You might be asked:

| "Explain how context managers work behind the scenes."

- OR:

| "When would you use raise instead of return?"

- OR:

| "Why use `__name__ == '__main__'`?"








In Projects:

- Writing reusable modules? Use `__main__`.
- Building CLI tools? Use type hints + exception handling.
- Working on APIs? Match-case + exception blocks.



Summary (TL;DR)

Concept	Skill
Walrus	One-liner assignments
Type Hints	Code clarity & safety
Match-Case	Clean decision logic
Dict Merge	Combine configs

 Context Manager	Safe file & resource handling
 Exceptions	Error-proof logic
 Raise	Intentional crashes
 Enumerate	Indexed loops
 List Comp	Fast & clean list creation

Advanced Python: Deep Dive — Level 2 Concepts

Beyond Basics: Real Power, Real Use Cases

1. Iterable vs Iterator vs Generator

All loops rely on this trio. If you understand this, you understand Python's looping core.

➤ Iterable

Any object capable of returning its members one at a time.

✓ Examples:

```
[1, 2, 3], "Hello", (1, 2), range(10)
```

➤ Iterator

An object with `__next__()` and `__iter__()` methods.

```
l = [1, 2, 3]
it = iter(l)
print(next(it)) # Output: 1
```

➤ Generator

A **lazy iterator**: does not store all values in memory, creates them on the fly using `yield`.

```
def gen():
    for i in range(3):
        yield i

g = gen()
print(next(g)) # Output: 0
```

Real Use:

- Efficient memory handling (e.g., reading large files, training models)
- Streams and pipelines (data processing)

2. Decorators (with and without parameters)

Functions that modify other functions — like plugins for code.

Basic Decorator:

```
def my_decorator(func):
    def wrapper():
        print("Before call")
        func()
        print("After call")
    return wrapper

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

Decorator With Arguments:

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

@repeat(3)
def greet(name):
    print(f"Hi {name}")

greet("Prathamesh")
```

Used in:

- Flask/Django routes
- Logging, authentication
- Time measurement

3. Lambda, `map()`, `filter()`, `reduce()`

Functional programming techniques that let you write ultra-compact, readable logic.

Lambda (Anonymous Functions)

```
square = lambda x: x**2
print(square(5)) # 25
```

map()

```
nums = [1, 2, 3]
doubled = list(map(lambda x: x*2, nums))
```

filter()

```
evens = list(filter(lambda x: x % 2 == 0, nums))
```

reduce() (from `functools`)

```
from functools import reduce
product = reduce(lambda x, y: x * y, [1, 2, 3, 4]) # Output: 24
```

 Useful in:

- Data processing pipelines
- One-liner transformations
- Pandas-style logic in pure Python

4. Closures and Factory Functions

| A closure is when a function remembers the values from its enclosing scope.

```
def outer(x):
    def inner(y):
        return x + y
    return inner

add5 = outer(5)
print(add5(3)) # 8
```

 Real Use:

- Function factories (like `@retry`, `@cache`)
- Encapsulation without using classes

5. Metaclasses

| The class of a class. Rare but powerful.

What They Are:

- In Python, everything is an object.
- Classes themselves are instances of **metaclasses**.

```
class Meta(type):
    def __new__(cls, name, bases, dct):
```

```
print(f"Creating class {name}")
return super().__new__(cls, name, bases, dct)
```

```
class MyClass(metaclass=Meta):
    pass
```

🧠 Use in:

- Frameworks (like Django's ORM)
- Customizing class behavior
- Singleton, autoloading, etc.

6. Contextlib for Custom Context Managers

| Build your own with statements — super clean and Pythonic.

```
from contextlib import contextmanager
```

```
@contextmanager
def open_file(name):
    f = open(name)
    try:
        yield f
    finally:
        f.close()
```

```
with open_file("test.txt") as f:
    data = f.read()
```

💡 Real Use:

- Managing DB connections, network sockets
- Replacing long `try...finally` chains

7. **args** and ***kwargs** — and Argument Unpacking

| Flexible function definitions for dynamic or unknown input.

🔍 Example:

```
def show(*args, **kwargs):
    print(args) # Tuple
    print(kwargs) # Dict
```

```
show(1, 2, 3, name="Prathamesh", age=17)
```

💡 Also useful in:

```
data = {"name": "P", "age": 17}
def intro(name, age): ...
intro(**data)
```

8. Dynamic Imports and `importlib`

You can import modules during runtime! 🔥

```
import importlib

module_name = "math"
math = importlib.import_module(module_name)
print(math.sqrt(25))
```

🧠 Used in:

- Plugin systems
- Lazy loading large codebases
- Dynamic toolkits

9. Dunder Methods (`__str__` , `__repr__` , `__len__` , `__getitem__`)

Python allows operator overloading and customization using “double underscore” methods.

```
class MyList:
    def __init__(self, data): self.data = data
    def __getitem__(self, i): return self.data[i]
    def __len__(self): return len(self.data)

ml = MyList([1,2,3])
print(len(ml))    # 3
print(ml[0])      # 1
```

💡 You can define:

- `__add__` , `__mul__` , `__contains__` , etc.

10. Property Decorator (`@property`)

Define a method that acts like an attribute.

```
class Circle:
    def __init__(self, radius):
        self._radius = radius
```



```
@property
def area(self):
    return 3.14 * self._radius ** 2
```

```
c = Circle(5)
print(c.area)
```

🧠 Use:


- Read-only attributes
- Encapsulate logic cleanly
- Don't break object interface

🔍 Summary Table (At a Glance)

Concept	Real Use Case	Core Keyword/Idea
Iterable vs Iterator	Efficient looping	<code>__iter__</code> , <code>__next__</code>
Generators	Lazy data pipelines	<code>yield</code>
Decorators	Plugins, auth, logging	<code>@decorator</code>
Lambda + map/filter/reduce	Compact logic in pipelines	Functional programming
Closures	Function factories	Lexical scope
Metaclasses	Customize class creation	<code>type()</code>
contextlib	Build <code>with</code> -like blocks	<code>@contextmanager</code>
*args/**kwargs	Flexible APIs	Var-args
Dynamic imports	Plugin systems, optional deps	<code>importlib</code>
Dunder Methods	Operator overloads, class control	<code>__str__</code> , <code>__getitem__</code>
<code>@property</code>	Clean calculated fields	<code>@property</code>

💣 30+ Common Mistakes in Advanced Python (Chapter 12)

#	❌ Common Mistake	🐛 What Goes Wrong	✅ Correct Usage or Fix
1	Using walrus in wrong context	Confusing assignment with comparison	<code>if (n := len(lst)) > 5:</code>
2	Forgetting parentheses with walrus	SyntaxError	<code>if (x := some_func()):</code> ✅
3	Using <code>type hints</code> as enforcement	Type hints don't enforce types	Use static type checkers like <code>mypy</code>
4	Using <code>match-case</code> without Python 3.10+	Code doesn't run on older versions	Ensure Python ≥ 3.10
5	Not handling <code>match-case</code> fallback	<code>match</code> only checks first match	Use <code>case _:</code> as default
6	Overriding built-in keywords like <code>sum</code>	Breaks built-in functions	Use names like <code>calc_sum()</code>
7	Using <code>`dict1</code>	<code>dict2`</code> in Python < 3.9	This merge operator fails

8	Reading both files from same handle	<code>f1.read()</code> twice = second read is empty	Read separately or reset file pointer
9	Not closing files without <code>with</code>	File handle leak	Always use <code>with open()</code>
10	Thinking <code>try-except</code> covers all errors	Doesn't catch syntax or indentation errors	Use carefully, not as blanket
11	Using <code>except:</code> without specifying error	Hides real issues	Use <code>except ValueError:</code> etc.
12	Raising exception without custom message	Hard to debug	<code>raise ValueError("Custom message")</code>
13	Misusing <code>raise</code> without exception	<code>raise</code> alone fails outside <code>except</code>	Use like: <code>raise SomeError()</code>
14	Using <code>else</code> block wrongly in <code>try</code>	Putting code that might error into <code>else</code>	Use <code>else</code> only for guaranteed safe code
15	Assuming <code>finally</code> stops code	Doesn't cancel crash, just runs	Use with care and understand its limits
16	Calling <code>__name__</code> instead of checking it	Confuses module execution check	Use <code>if __name__ == "__main__":</code> 
17	Forgetting to use <code>global</code> keyword	Variable defined locally instead	Use <code>global x</code> inside function
18	Overwriting <code>global</code> unintentionally	Global <code>a = 5</code> , local <code>a = 10</code> — shadowing	Use different names or <code>nonlocal</code>
19	Using <code>enumerate()</code> without unpacking	Gets <code>(index, item)</code> tuple, not value	Use <code>for i, val in enumerate(lst):</code> 
20	Using list comprehension for side effects	It's meant for building new lists	Use loops if you're not storing result
21	Using <code>lambda</code> for complex logic	Hard to read and debug	Use named functions instead
22	Forgetting <code>yield</code> in generators	No values are yielded, returns <code>None</code>	Use <code>yield</code> instead of <code>return</code> for each value
23	Misusing decorators with arguments	Forgetting extra wrapper layer	Use 3-level structure for param decorators
24	Confusing <code>*args</code> and <code>**kwargs</code> order	<code>SyntaxError</code>	<code>def func(a, *args, **kwargs)</code> 
25	Using mutable defaults like <code>[]</code> in function args	Shared across function calls	Use <code>None</code> , then assign inside
26	Misusing <code>@property</code> setter/getter	Forgetting to use <code>@<property>.setter</code>	Use correct decorator syntax
27	Treating <code>@property</code> like regular method	Causes confusion in calls	Call like <code>obj.prop</code> , not <code>obj.prop()</code>
28	Creating recursive decorators without base case	Infinite recursion	Always ensure a termination condition
29	Using <code>import</code> inside loops unnecessarily	Slows down performance	Move <code>import</code> statements to top of file
30	Ignoring error types while debugging	Using <code>except Exception</code> blindly	Log error with <code>traceback</code> or <code>e.__class__</code>
31	Using walrus with expressions that return <code>None</code>	Assigns <code>None</code> and fails comparisons	Avoid: <code>if (x := print()) > 5</code> 
32	Not using <code>contextlib</code> for custom context managers	Writing verbose <code>__enter__ / __exit__</code> code	Use <code>@contextmanager</code> for simpler syntax

33	Overusing try-finally for non-critical sections	Code becomes complex for no reason	Use only when cleanup is essential
----	---	------------------------------------	------------------------------------

✓ Chapter 12 – Practice Problem Notes (Advanced Python)

Problem 1 – Safe File Opening with Context Manager

```
# Open 1.txt, 2.txt, and 3.txt
# Show an error if any file doesn't exist — but the program must not crash

try:
    with (
        open('1.txt') as f1,
        open('2.txt') as f2,
        open('3.txt') as f3
    ):
        print("Opened All files")
except Exception as e:
    print("Error:", e)

print("This program has not crashed!")
```

Concepts Covered:

- `with` statement for multiple file handling
- Graceful error handling with `try-except`
- Prevents crash using exception fallback

Edge Case:

- One missing file breaks the entire `with` block

Suggestion:

Open each file individually or iterate through a list if you want to open only valid files.

Problem 2 – Print Specific Elements Using `enumerate()`

```
# Print 3rd, 5th, and 7th elements using `enumerate`

l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 323, 3, 23, 23, 23]

for index, item in enumerate(l):
    if index in [2, 4, 6]: # 3rd, 5th, and 7th elements (0-based index)
        print(item)
```

Concepts Covered:

- `enumerate()` for cleaner loop with index
- Selective item access via index

💡 **Tip:** Avoid hardcoding indices if positions are dynamic — consider using `range(start, stop, step)`.

Problem 3 – Multiplication Table Using List Comprehension

```
# List comprehension to print multiplication table of user input


user_no = int(input("Enter the number: "))
table = [user_no * i for i in range(1, 11)]
print(table)
```

Concepts Covered:

- List comprehension
- Dynamic list creation

💡 **Edge Case:**

- Negative or zero input → still gives valid table

 **Example Output** for 5 → `[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]`

Problem 4 – Safe Division (a / b) with Zero Division Handling

```
# Display a/b. Show "Infinity" if b == 0. Round to integer

try:
    a = int(input("Enter 1st Number: "))
    b = int(input("Enter 2nd Number: "))
    c = a / b
    print(int(c))
except ZeroDivisionError:
    print("Infinity")
except Exception as e:
    print("Error")

print("Program not crashed")
```

Concepts Covered:

- Exception handling with specific error types
- Safe division
- Type conversion

💡 **Improvement:**

- You can show float result or keep it rounded, based on use case.

Problem 5 – Save Multiplication Table to File

```
# Save the multiplication table into a file named table.txt

user_no = int(input("Enter the number: "))
table = [user_no * i for i in range(1, 11)]
```

```
print(table)
```

```
# Convert list to string and write
with open("CH12_PS//table.txt", "a") as f:
    f.write(str(table) + "\n")
```

🔍 Concepts Covered:

- File writing using `with open()`
- List to string conversion
- Appending mode (`"a"`)

🧠 Edge Case:

- Ensure "CH12_PS" folder exists or `FileNotFoundError` will occur.

💡 Improvement:

Format output line-by-line using:

```
for i in range(1, 11):
    f.write(f"{user_no} x {i} = {user_no*i}\n")
```

🧠💡 Chapter 12: Advanced Python – 🔥 All-in-One Summary Table

📌 #	🌱 Topic / Feature	🧠 Description	🔥 Concepts Involved	⚠️ Common Mistakes	🔧 Practice Problem(s)
1	Walrus Operator <code>(:=)</code>	Assignment inside expressions	Efficient loops, <code>if</code> checks	Using before Python 3.8, Confusing syntax	N/A
2	Type Hints / Annotations	Declare variable/function types	Clean code, IDE hinting, <code>typing</code> module	Thinking it enforces type (Python is dynamic)	N/A
3	Match-Case (Structural Pattern Matching)	Modern <code>switch</code> - like logic	Clean conditional chains	Forgetting <code>case</code> - default	N/A
4	Dictionary Merge	.	Merge two dictionaries	Shorter syntax, dict union	Overwriting values unknowingly
5	Multi-File Context Manager <code>with</code>	Open multiple files in one <code>with</code> block	Context managers	All fail if one file missing	Problem 1
6	Exception Handling	Try-except to prevent crashes	ValueError, ZeroDivisionError, etc.	Using general <code>except:</code> carelessly	Problem 4
7	<code>raise</code> Custom Exceptions	Intentionally crash if conditions fail	Input validation, design patterns	Raising without meaning, wrong exception type	N/A

#	Topic / Feature	Description	Concepts Involved	Common Mistakes	Practice Problem(s)
8	<code>try-else</code> Block	Run <code>else</code> if no exception	Cleaner success logic	Expecting it to always run	Problem 4
9	<code>finally</code> Block	Always runs — even after <code>return</code>	Logging, cleanup, final confirmation	Misusing <code>return</code> before <code>finally</code>	Problem 4
10	<code>if __name__ == "__main__":</code>	Know when file is imported vs run	Reusability, modularity	Using <code>if name == "__main__"</code> (typo)	<code>module.py</code> + <code>main.py</code>
11	Global Variables	Modify global from inside function	Use <code>global</code> keyword	Forgetting <code>global</code> → variable not created	<code>global.py</code>
12	<code>enumerate()</code>	Track index in <code>for</code> loop	Cleaner alternative to manual counter	Index mismatch with wrong logic	Problem 2
13	List Comprehension	Compact way to create lists	One-liners, filtering, transformations	Making code unreadable, nesting too much	Problems 3 & 5
14	Writing to Files	Using <code>with open(..., "a")</code> to store data	Appending data	Not checking file path/folder	Problem 5
15	Custom Errors	Manual exception messages	User-defined error logic	Forgetting to import or define class	Problem 4 (partially)
16	Dynamic Type Systems	Python is dynamic despite hints	Flexibility in duck typing	Assuming type enforcement	<code>typehints.py</code>
17	<code>__name__</code> Special Variable	Helps modularize scripts	<code>"__main__"</code> logic	Using as a string directly	<code>module.py</code>
18	Reading Multiple Files	Compare contents of 2 files	File I/O, with-as context	Reading same file twice instead of both	<code>dictmerge_and_context.py</code>
19	Operator Overloading	Customize <code>+</code> , <code>*</code> , etc.	Classes like <code>vector</code> , <code>complex</code>	Forgetting <code>__str__</code> , <code>__add__</code> methods	Ch11 Problem 4, 5, 6
20	Exception Flow Control	Raise → Try → Catch → Finally	Full flow of error handling	Nested exceptions not caught properly	Problems 1 & 4
21	Typing Tuple, List	<code>Tuple[int, str]</code> for readability	<code>typing</code> module	Using native <code>tuple[...]</code> before Python 3.9	<code>typehints.py</code>