Python Testing & Debugging

Complete Beginner Notes

unittest · pytest · pdb · logging · cProfile · VS Code

10 Practice Tasks Included

From Zero to Confident Tester

Chapter 0

Table of Contents

Chapter 1

Why Testing Matters

Testing is the process of running your code with known inputs and checking that the output is what you expect. It is one of the most important skills a developer can have — it saves hours of debugging, prevents bugs from reaching users, and gives you confidence to change code without fear of breaking things.

Key Benefits of Testing

- **Catch bugs early** — find problems during development, not in production.
- **Safe refactoring** — change code freely, tests will catch regressions.
- **Living documentation** — tests show exactly how functions should behave.
- **Better design** — writing tests forces you to write cleaner, modular code.
- **Team confidence** — everyone can contribute without fear.

Types of Tests

Type	What it tests	Example
Unit Test	A single function or class	test that add(2,3) == 5
Integration Test	Multiple modules working together	test that login + profile both work
End-to-End Test	The whole app from user perspective	click login button → see dashboard
Regression Test	Old bugs don't come back	test that fixed bug stays fixed

■ *Start with unit tests — they are the simplest to write and give the most value per line of code.*

Chapter 2

Writing Tests with unittest

unittest is Python's built-in testing framework. It comes installed with Python — no extra packages needed. It is inspired by Java's JUnit and uses classes to organise tests.

Basic Structure

Every unittest file follows this pattern:

```
import unittest

# 1. Write the function you want to test
def add(a, b):
    return a + b

# 2. Create a test class that inherits from unittest.TestCase
class TestAdd(unittest.TestCase):

    # 3. Each test is a method starting with 'test_'
    def test_add_two_positives(self):
        result = add(2, 3)
        self.assertEqual(result, 5)   # check result == 5

    def test_add_negative(self):
        self.assertEqual(add(-1, -1), -2)

    def test_add_zero(self):
        self.assertEqual(add(0, 5), 5)

# 4. Run the tests
if __name__ == "__main__":
    unittest.main()
```

Running unittest

- **From terminal:** python -m unittest test_add.py
- **Verbose mode:** python -m unittest -v test_add.py
- **Discover all tests:** python -m unittest discover

All Assert Methods Explained

Assert methods are how you **check** that your code did the right thing. If the check fails, the test fails.

Method	Checks that...	Example
`assertEqual(a, b)`	a == b	`assertEqual(add(2,3), 5)`
`assertNotEqual(a, b)`	a != b	`assertNotEqual(1, 2)`
`assertTrue(x)`	x is True/truthy	`assertTrue(is_even(4))`
`assertFalse(x)`	x is False/falsy	`assertFalse(is_even(3))`
`assertIsNone(x)`	x is None	`assertIsNone(find('z'))`
`assertIsNotNone(x)`	x is not None	`assertIsNotNone(find('a'))`
`assertIn(a, b)`	a is in b	`assertIn(3, [1,2,3])`

assertNotIn(a, b)	a is not in b	assertNotIn(9, [1,2,3])
assertRaises(Err, fn)	fn() raises Err	assertRaises(ValueError, int, 'x')
assertAlmostEqual(a,b)	a ≈ b (floats)	assertAlmostEqual(0.1+0.2, 0.3)
assertGreater(a, b)	a > b	assertGreater(5, 3)
assertLess(a, b)	a < b	assertLess(3, 5)
assertIsInstance(a, T)	isinstance(a, T)	assertIsInstance(3, int)

setUp and tearDown — Setup & Cleanup

Sometimes tests need to share setup code (like creating objects). setUp() runs **before each test**, tearDown() runs **after each test**.

```
class TestBankAccount(unittest.TestCase):
    def setUp(self):              # runs before EACH test
        self.account = BankAccount(owner='Alice', balance=100)
    def tearDown(self):           # runs after EACH test
        pass  # cleanup if needed (close files, delete DB, etc.)
    def test_deposit(self):
        self.account.deposit(50)
        self.assertEqual(self.account.balance, 150)
    def test_withdraw(self):
        self.account.withdraw(30)
        self.assertEqual(self.account.balance, 70)
    def test_withdraw_too_much(self):
        with self.assertRaises(ValueError):   # expects an error
            self.account.withdraw(999)
```

Testing for Exceptions

Use **assertRaises** to check that your code correctly raises errors when given bad input:

```
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError('Cannot divide by zero')
    return a / b
class TestDivide(unittest.TestCase):
    def test_normal_division(self):
        self.assertEqual(divide(10, 2), 5.0)
    def test_divide_by_zero(self):
        with self.assertRaises(ZeroDivisionError):   # ← context manager style
            divide(10, 0)
    def test_raises_with_message(self):
        with self.assertRaises(ZeroDivisionError) as ctx:
            divide(5, 0)
        self.assertIn('zero', str(ctx.exception))
```

■ *Naming convention: test files should start with 'test_' and test methods must start with 'test_'. This is how Python's test runner finds them automatically.*

Chapter 3

Writing Tests with pytest

pytest is the most popular Python testing framework. It is simpler to write than unittest (no classes needed!), has better error messages, and has a huge ecosystem of plugins. Install it with: `pip install pytest`

pytest vs unittest — Side by Side

Feature	unittest	pytest
Install needed?	No (built-in)	Yes: pip install pytest
Class required?	Yes (inherit TestCase)	No — just functions!
Assert style	self.assertEqual(a, b)	assert a == b
Fixtures	setUp/tearDown methods	@pytest.fixture decorator
Parametrize	Manual loops	@pytest.mark.parametrize
Output	Basic	Colourful, detailed
Plugins	Limited	Huge ecosystem

Basic pytest — No Classes Needed!

```
# test_math.py  ← file must start with 'test_'
def add(a, b):
    return a + b
def multiply(a, b):
    return a * b
# ── tests ──────────────────────────────────────
def test_add_positive():          # function must start with test_
    assert add(2, 3) == 5         # plain assert — no assertEqual!
def test_add_negative():
    assert add(-1, -1) == -2
def test_multiply():
    assert multiply(3, 4) == 12
def test_multiply_by_zero():
    assert multiply(99, 0) == 0
```

Run with just: `pytest` or `pytest -v` for verbose output.

Fixtures — Reusable Setup Code

A **fixture** is a function decorated with **@pytest.fixture**. pytest automatically passes it to any test that asks for it by name.

```
import pytest
class BankAccount:
    def __init__(self, balance=0):
        self.balance = balance
```

```
    def deposit(self, amount):  self.balance += amount
    def withdraw(self, amount): self.balance -= amount
@pytest.fixture              # ← this creates the fixture
def account():
    return BankAccount(balance=100)   # fresh account for each test
def test_deposit(account):   # ← pytest injects 'account' automatically
    account.deposit(50)
    assert account.balance == 150
def test_withdraw(account):
    account.withdraw(30)
    assert account.balance == 70
```

Parametrize — Test Many Inputs at Once

Instead of writing a separate test for each input, use **@pytest.mark.parametrize** to run the same test with many different values:

```
import pytest
def is_even(n):
    return n % 2 == 0
@pytest.mark.parametrize('number, expected', [
    (2,  True),    # 2 is even
    (3,  False),   # 3 is odd
    (0,  True),    # 0 is even
    (-4, True),    # -4 is even
    (7,  False),   # 7 is odd
])
def test_is_even(number, expected):
    assert is_even(number) == expected  # runs 5 separate tests!
```

■ *@pytest.mark.parametrize saves you from writing repetitive test functions. pytest will run one test per tuple and report each separately.*

Testing Exceptions in pytest

```
import pytest
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError('Cannot divide by zero')
    return a / b
def test_divide_by_zero():
    with pytest.raises(ZeroDivisionError):   # ← pytest.raises context
        divide(10, 0)
def test_divide_error_message():
    with pytest.raises(ZeroDivisionError, match='zero'):   # check msg too
        divide(5, 0)
```

Useful pytest Command-Line Options

Command	What it does
pytest	Run all tests in current directory

`pytest -v`	Verbose: show each test name
`pytest -s`	Show print() output during tests
`pytest test_math.py`	Run only one test file
`pytest test_math.py::test_add`	Run one specific test
`pytest -k 'add'`	Run tests whose name contains 'add'
`pytest --tb=short`	Shorter traceback on failure
`pytest -x`	Stop after first failure
`pytest --lf`	Re-run only last failed tests
`pytest -q`	Quiet mode (minimal output)

Chapter 4

Running Tests in VS Code

VS Code has excellent built-in support for Python testing. You can run and debug tests with a click, see results inline, and get full test output without leaving the editor.

Step-by-Step Setup

Step	Action	Details
1	Install Python Extension	Open Extensions (Ctrl+Shift+X) → search 'Python' → Install the Microsoft Python extension
2	Install pytest	Open terminal in VS Code (Ctrl+`) → type: pip install pytest
3	Open Testing Panel	Click the flask icon (■) in the left sidebar, OR press Ctrl+Shift+P → 'Python: Configure Tests'
4	Choose Framework	Select 'pytest' or 'unittest' → select the folder containing your tests (usually the root)
5	Discover Tests	VS Code will scan for test files and show them in the Testing panel
6	Run Tests	Click the ■ button next to any test, or the ■■ button to run all tests
7	Debug Tests	Click the ■ button next to a test to debug it with breakpoints

settings.json Configuration

You can configure VS Code testing by editing **.vscode/settings.json** in your project:

```
{
    "python.testing.pytestEnabled": true,
    "python.testing.unittestEnabled": false,
    "python.testing.pytestArgs": [
        "tests",        // folder where your tests live
        "-v",           // verbose output
        "--tb=short"    // short tracebacks
    ]
}
```

Setting Breakpoints in VS Code

- Click in the **gutter** (left margin) next to any line number — a red dot appears. That is a breakpoint.
- When you **debug a test**, execution pauses at that line.
- You can then **hover over variables** to see their values, or use the Debug Console.
- Use **F10** to step over, **F11** to step into a function, **F5** to continue.

VS Code Keyboard Shortcuts for Testing

Shortcut	Action
`Ctrl+Shift+P`	Open Command Palette (search any command)
`Ctrl+``	Open integrated terminal

`F5`	Start / continue debugging
`F10`	Step over (next line, don't enter functions)
`F11`	Step into (enter the function)
`Shift+F11`	Step out (return from current function)
`Shift+F5`	Stop debugging
`Ctrl+Shift+D`	Open Run & Debug panel

Chapter 5

Debugging Techniques & Tools

Debugging is the process of finding and fixing errors in your code. Python gives you several tools for this — from simple print statements to powerful interactive debuggers.

Technique 1 — print() Debugging (Simple but Effective)

The simplest debugging technique: add print() statements to see what your variables contain and what path your code is taking.

```
def calculate_total(prices, discount):
    print(f'DEBUG: prices = {prices}')        # see the input
    print(f'DEBUG: discount = {discount}')
    subtotal = sum(prices)
    print(f'DEBUG: subtotal = {subtotal}')    # see intermediate value
    total = subtotal - (subtotal * discount)
    print(f'DEBUG: total = {total}')          # see final value
    return total
result = calculate_total([10, 20, 30], 0.1)
# Output: DEBUG: prices = [10, 20, 30]
#         DEBUG: subtotal = 60
#         DEBUG: total = 54.0
```

■ *Use f-strings for debug prints: print(f'var = {var}') is much clearer than print('var =', var)*

Technique 2 — assert Statements (Early Error Detection)

Use **assert** to check assumptions in your code. If the condition is False, Python raises an AssertionError immediately.

```
def get_percentage(part, whole):
    assert whole != 0, 'whole cannot be zero!'  # guard clause
    assert 0 <= part <= whole, f'part {part} must be between 0 and {whole}'
    return (part / whole) * 100
print(get_percentage(30, 100))   # 30.0 — works fine
print(get_percentage(30, 0))     # AssertionError: whole cannot be zero!
```

Technique 3 — Reading Tracebacks

When Python raises an error, it shows a **traceback**. Reading it correctly is essential for fast debugging:

```
Traceback (most recent call last):          # read from BOTTOM to TOP
  File "app.py", line 12, in main           # outermost call
    result = process(data)
  File "app.py", line 7, in process         # inner call
    return calculate(x)
  File "app.py", line 3, in calculate       # where error happened
    return 10 / x
ZeroDivisionError: division by zero         # ← this is the actual error
# TIPS:
```

```
# 1. Look at the LAST line first — that's the error type and message
# 2. Then look at the line just above it — that's where it happened
# 3. The file and line number tell you EXACTLY where to look
```

Common Python Errors & How to Fix Them

Error	Cause	Fix
`SyntaxError`	Typo, missing colon, bad indent	Read the line indicated, check colons & brackets
`NameError`	Using a variable before defining	Check spelling, check variable is defined first
`TypeError`	Wrong type (e.g. int + str)	Check types with type(), add conversions
`IndexError`	List index out of range	Check list length, use len()
`KeyError`	Dict key doesn't exist	Use dict.get(key) instead of dict[key]
`AttributeError`	Object has no such attribute	Check spelling, check object type
`ValueError`	Right type, wrong value	Validate input before using it
`ZeroDivisionError`	Dividing by zero	Check divisor != 0 before dividing
`ImportError`	Module not found	pip install the package, check spelling
`IndentationError`	Wrong indentation	Use consistent spaces (4 spaces recommended)
`RecursionError`	Infinite recursion	Add a base case to stop the recursion

Chapter 6

Using pdb — Python Debugger

pdb is Python's built-in interactive debugger. It lets you pause your program at any point, inspect variables, and step through code line by line — much more powerful than print() debugging.

How to Start pdb

- **Method 1 — In code:** add import pdb; pdb.set_trace() where you want to pause
- **Method 2 — Python 3.7+:** use breakpoint() (simpler, same effect)
- **Method 3 — Command line:** python -m pdb script.py
- **Method 4 — VS Code:** click the red dot gutter → Debug Test / Run with debugger

pdb in Action — Example

```
def find_max(numbers):
    breakpoint()   # ← execution pauses here, pdb prompt appears
    max_val = numbers[0]
    for num in numbers:
        if num > max_val:
            max_val = num
    return max_val
result = find_max([3, 7, 1, 9, 2])
# When you run this, you see:
# > script.py(3)find_max()
# -> max_val = numbers[0]
# (Pdb) _     ← you type commands here
```

pdb Commands Cheat Sheet

Command	Short	What it does
help	h	Show all available commands
next	n	Execute next line (don't enter functions)
step	s	Step INTO a function call
continue	c	Continue running until next breakpoint
quit	q	Exit the debugger (stop the program)
return	r	Run until current function returns
list	l	Show surrounding source code
print x	p x	Print the value of variable x
pp x	pp	Pretty-print complex objects (lists, dicts)
where	w	Show the current call stack
up	u	Move up one level in the call stack
down	d	Move down one level in the call stack

`break N`	`b N`	Set a breakpoint at line N
`clear`	`cl`	Clear all breakpoints
`args`	`a`	Show arguments of current function
`! expr`	`!`	Execute any Python expression

Interactive pdb Session Example

```
(Pdb) l                    # list code around current line
(Pdb) p numbers            # print value of 'numbers'
# Output: [3, 7, 1, 9, 2]
(Pdb) p max_val            # print current max_val
# Output: 3
(Pdb) n                    # next line
(Pdb) n                    # next line
(Pdb) p num                # see current loop variable
# Output: 7
(Pdb) ! max_val = 100      # change a variable's value mid-run!
(Pdb) c                    # continue to end
```

■ *You can type any Python expression at the pdb prompt. Try: p len(numbers), p type(max_val), p [x for x in numbers if x > 5]*

Chapter 7

Logging for Debugging

The **logging** module is better than print() for real projects. Unlike print(), logging lets you control what gets shown (by level), write to files, include timestamps, and easily turn off debug messages in production.

Log Levels — What They Mean

Level	Number	When to use it
DEBUG	10	Detailed info for diagnosing problems (most verbose)
INFO	20	Confirmation that things are working as expected
WARNING	30	Something unexpected, but code still works
ERROR	40	A serious problem — some functionality failed
CRITICAL	50	A very serious error — the program may not continue

Basic Logging Setup

```
import logging

# Configure logging — do this once at the top of your script
logging.basicConfig(
    level=logging.DEBUG,      # show all messages at DEBUG and above
    format='%(asctime)s | %(levelname)s | %(message)s',
    datefmt='%H:%M:%S'
)

# Now use it anywhere in your code
def process_order(order_id, amount):
    logging.debug(f'Processing order {order_id}, amount={amount}')
    if amount <= 0:
        logging.error(f'Invalid amount {amount} for order {order_id}')
        return False
    if amount > 10000:
        logging.warning(f'Large order: {amount} — manual review needed')
    logging.info(f'Order {order_id} processed successfully')
    return True

process_order('ORD-001', 250)
# 14:30:15 | DEBUG    | Processing order ORD-001, amount=250
# 14:30:15 | INFO     | Order ORD-001 processed successfully
```

Logging to a File

```
import logging

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s | %(levelname)-8s | %(name)s | %(message)s',
    handlers=[
```

```
        logging.FileHandler('app.log'),      # write to file
        logging.StreamHandler()              # also print to console
    ]
)
logger = logging.getLogger(__name__)    # best practice: named logger
logger.info('Application started')
logger.debug('Loading config...')
logger.warning('Config file missing, using defaults')
```

■ *Set level=logging.WARNING in production to hide DEBUG and INFO messages. This avoids performance overhead from excessive logging.*

Chapter 8

Profiling & Optimization

Profiling means measuring where your code spends its time. You should **always profile before optimizing** — guessing which part is slow is almost always wrong. Find the bottleneck first, then fix it.

"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." — Donald Knuth

Method 1 — time module (Simple Timing)

```
import time

def slow_function(n):
    return sum(i**2 for i in range(n))

# Method A: time.time()
start = time.time()
result = slow_function(1_000_000)
end = time.time()
print(f'Took {end - start:.4f} seconds')

# Method B: time.perf_counter() — more precise
start = time.perf_counter()
result = slow_function(1_000_000)
elapsed = time.perf_counter() - start
print(f'Took {elapsed:.6f} seconds')
```

Method 2 — timeit (Accurate Benchmarking)

timeit runs your code many times and gives an average, eliminating random system noise:

```
import timeit

# Compare two approaches
code_1 = '[i**2 for i in range(1000)]'       # list comprehension
code_2 = 'list(map(lambda i: i**2, range(1000)))'  # map

t1 = timeit.timeit(code_1, number=10000)   # run 10000 times
t2 = timeit.timeit(code_2, number=10000)

print(f'List comprehension: {t1:.4f}s')
print(f'Map:                {t2:.4f}s')
print(f'Winner: {"comprehension" if t1 < t2 else "map"}')

# From command line:
# python -m timeit '[i**2 for i in range(1000)]'
```

Method 3 — cProfile (Full Function Profiler)

cProfile shows exactly how many times each function was called and how long each took:

```
import cProfile
import pstats

def slow_search(data, target):
    for item in data:          # O(n) — slow for large lists
        if item == target:
```

```
            return True
    return False
def main():
    data = list(range(100_000))
    for _ in range(100):
        slow_search(data, 99999)
# Profile and print stats
with cProfile.Profile() as pr:
    main()
stats = pstats.Stats(pr)
stats.sort_stats('cumulative')  # sort by total time
stats.print_stats(10)           # show top 10 functions
# Output columns:
# ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
```

Reading cProfile Output

Column	Meaning
`ncalls`	How many times this function was called
`tottime`	Total time spent IN this function (not sub-calls)
`percall`	tottime / ncalls — average time per call
`cumtime`	Cumulative time including all sub-function calls
`percall`	cumtime / ncalls — average cumulative time per call
`filename`	Which file and line the function is in

Common Optimization Techniques

Technique	Before	After	Speedup
List comprehension	`loop + append`	`[x for x in data]`	**2-5x**
Set for lookups	`x in list`	`x in set(data)`	**100x+**
join for strings	`result += str`	`''.join(list)`	**10x+**
Generator vs list	`list(range(1M))`	`range(1M) or (x for x...)`	**Memory**
Dict.get()	`if key in d: d[key]`	`d.get(key, default)`	**Cleaner**
Local var in loops	`global.method() in loop`	`f = obj.method; f() in loop`	**20%**
numpy for math	`for loop on numbers`	`numpy array ops`	**100x+**
Caching repeats	`recalculate each time`	`@functools.lru_cache`	**varies**

lru_cache — Automatic Memoization

```
from functools import lru_cache
import time
# Without cache — recalculates every time
def fib_slow(n):
```

```
    if n < 2: return n
    return fib_slow(n-1) + fib_slow(n-2)
# With lru_cache — remembers previous results
@lru_cache(maxsize=None)    # cache all results
def fib_fast(n):
    if n < 2: return n
    return fib_fast(n-1) + fib_fast(n-2)
start = time.perf_counter()
print(fib_slow(35))   # takes several seconds
print(f'Slow: {time.perf_counter()-start:.2f}s')
start = time.perf_counter()
print(fib_fast(35))   # instant!
print(f'Fast: {time.perf_counter()-start:.6f}s')
```

■ *The golden rule of optimization: MEASURE first, then optimize only the slowest parts. A 10x speedup in code that takes 1% of total time gives you almost nothing.*

Chapter 9

10 Practice Tasks for Beginners

Apply everything you've learned with these hands-on tasks. Each task builds on the concepts in this guide. Try to write the tests **before** the code — this is called Test-Driven Development (TDD)!

Task 01 Temperature Converter Tests

Write a function celsius_to_fahrenheit(c) and fahrenheit_to_celsius(f). Then write unittest tests for: freezing point (0°C = 32°F), boiling point (100°C = 212°F), body temperature, and negative values.

```
import unittest
def celsius_to_fahrenheit(c):
    return (c * 9/5) + 32
class TestTemperature(unittest.TestCase):
    def test_freezing(self):
        self.assertEqual(celsius_to_fahrenheit(0), 32)
    def test_boiling(self):
        self.assertEqual(celsius_to_fahrenheit(100), 212)
    # Add more tests for negative, body temp, etc.
```

Task 02 Palindrome Checker with pytest

Write is_palindrome(s) that returns True if s reads the same forwards and backwards (ignore case and spaces). Use pytest with @pytest.mark.parametrize to test: 'racecar', 'hello', 'A man a plan a canal Panama', ''.

```
import pytest
def is_palindrome(s):
    clean = s.lower().replace(' ', '')
    return clean == clean[::-1]
@pytest.mark.parametrize('text, expected', [
    ('racecar', True),
    ('hello',   False),
    ('A man a plan a canal Panama', True),
    ('',        True),
])
def test_palindrome(text, expected):
    assert is_palindrome(text) == expected
```

Task 03 Shopping Cart with setUp

Create a ShoppingCart class with add_item(name, price), remove_item(name), and get_total() methods. Write unittest tests using setUp to create a fresh cart before each test. Test adding items, removing items, and getting the correct total.

```
class TestShoppingCart(unittest.TestCase):
    def setUp(self):
        self.cart = ShoppingCart()
        self.cart.add_item('Apple', 1.50)
        self.cart.add_item('Bread', 2.00)
    def test_total(self):
        self.assertAlmostEqual(self.cart.get_total(), 3.50)
```

```
    def test_remove_item(self):
        self.cart.remove_item('Apple')
        self.assertAlmostEqual(self.cart.get_total(), 2.00)
```

Task 04 Exception Testing — Input Validator

Write a validate_age(age) function that raises ValueError if age < 0 or age > 150, and TypeError if age is not an integer. Write tests using both assertRaises and pytest.raises to verify these exceptions are raised correctly.

```
def validate_age(age):
    if not isinstance(age, int):
        raise TypeError('Age must be an integer')
    if age < 0 or age > 150:
        raise ValueError(f'Age {age} is out of valid range')
    return True
class TestValidateAge(unittest.TestCase):
    def test_valid_age(self):  self.assertTrue(validate_age(25))
    def test_negative(self):
        with self.assertRaises(ValueError): validate_age(-1)
    def test_wrong_type(self):
        with self.assertRaises(TypeError):  validate_age('25')
```

Task 05 Debug with pdb — Find the Bug

The function below has a bug. Use pdb/breakpoint() to inspect variables and find it. The function should return the average of a list, but it returns wrong results for some inputs.

```
def calculate_average(numbers):
    breakpoint()    # start debugging here
    total = 0
    count = len(numbers)
    for num in numbers:
        total = total + num
    average = total / count    # what if numbers is empty?
    return average
# Bug to find: no handling for empty list (ZeroDivisionError)
# Fix: add  if not numbers: return 0  at the start
# Use pdb commands: p total, p count, p num, n, c
```

Task 06 Logging — File Processor

Write a process_file(filename) function that reads a text file, counts words, and returns the count. Add logging statements at DEBUG level to show each step, INFO for success, WARNING for empty files, and ERROR if the file doesn't exist.

```
import logging
logging.basicConfig(level=logging.DEBUG,
    format='%(levelname)s: %(message)s')
def process_file(filename):
    logging.debug(f'Opening file: {filename}')
    try:
        with open(filename) as f:
            content = f.read()
        words = content.split()
        if not words:
```

```
            logging.warning(f'{filename} is empty')
        logging.info(f'Found {len(words)} words in {filename}')
        return len(words)
    except FileNotFoundError:
        logging.error(f'File not found: {filename}')
        return None
```

Task 07 Profile & Optimize a Search

Write two search functions: linear_search (loop through list) and binary_search (sorted list, divide and conquer). Use timeit to compare them on a list of 100,000 numbers. Then use cProfile to see where time is spent.

```
import timeit, bisect
data = list(range(100_000))
sorted_data = sorted(data)
def linear_search(lst, target):
    for i, val in enumerate(lst):
        if val == target: return i
    return -1
def binary_search(lst, target):
    idx = bisect.bisect_left(lst, target)
    if idx < len(lst) and lst[idx] == target: return idx
    return -1
t1 = timeit.timeit(lambda: linear_search(data, 99999), number=1000)
t2 = timeit.timeit(lambda: binary_search(sorted_data, 99999), number=1000)
print(f'Linear: {t1:.4f}s  |  Binary: {t2:.6f}s')
```

Task 08 pytest Fixtures — Database Simulation

Create a simple UserDatabase class with add_user(name, email), get_user(name), and delete_user(name) methods. Write a pytest fixture that creates a database with test users, then write tests for all three operations.

```
import pytest
@pytest.fixture
def db():
    database = UserDatabase()
    database.add_user('Alice', 'alice@test.com')
    database.add_user('Bob', 'bob@test.com')
    return database
def test_get_user(db):
    user = db.get_user('Alice')
    assert user['email'] == 'alice@test.com'
def test_delete_user(db):
    db.delete_user('Bob')
    assert db.get_user('Bob') is None
def test_add_user(db):
    db.add_user('Carol', 'carol@test.com')
    assert db.get_user('Carol') is not None
```

Task 09 lru_cache Optimization

Write a function that computes the n-th Fibonacci number using recursion (without cache). Measure how long fib(35) takes. Then add @lru_cache and measure again. Compare the results and also check cache_info()

statistics.

```
from functools import lru_cache
import time
def fib(n):           # slow version
    if n < 2: return n
    return fib(n-1) + fib(n-2)
@lru_cache(maxsize=None)
def fib_cached(n):    # fast version
    if n < 2: return n
    return fib_cached(n-1) + fib_cached(n-2)
t1 = time.perf_counter(); fib(35); slow = time.perf_counter()-t1
t2 = time.perf_counter(); fib_cached(35); fast = time.perf_counter()-t2
print(f'Without cache: {slow:.4f}s')
print(f'With cache:    {fast:.8f}s')
print(f'Speedup:       {slow/fast:.0f}x faster')
print(fib_cached.cache_info())
```

Task 10 Full Mini-Project — Calculator with Tests

Build a Calculator class with add, subtract, multiply, divide, and power methods. Write a complete test suite using pytest covering: normal cases, edge cases (divide by zero, zero inputs, negative numbers), and use parametrize for at least one method. Run with pytest -v and make all tests pass.

```
# calculator.py
class Calculator:
    def add(self, a, b):      return a + b
    def subtract(self, a, b): return a - b
    def multiply(self, a, b): return a * b
    def power(self, a, b):    return a ** b
    def divide(self, a, b):
        if b == 0: raise ZeroDivisionError('Cannot divide by zero')
        return a / b
# test_calculator.py
@pytest.fixture
def calc(): return Calculator()
@pytest.mark.parametrize('a,b,expected', [(1,2,3),(0,5,5),(-1,-1,-2)])
def test_add(calc, a, b, expected):
    assert calc.add(a, b) == expected
def test_divide_by_zero(calc):
    with pytest.raises(ZeroDivisionError):
        calc.divide(10, 0)
```

Quick Reference Summary

Tool / Concept	What it's for	Key command / syntax
unittest	Built-in test framework	python -m unittest -v
pytest	Popular test framework	pytest -v --tb=short
assertEqual	Check two values are equal	self.assertEqual(a, b)
assert (pytest)	Check a condition is true	assert result == expected

`assertRaises`	Check exception is raised	`with self.assertRaises(Err):`
`pytest.raises`	Check exception (pytest)	`with pytest.raises(Err):`
`setUp / tearDown`	Per-test setup and cleanup	`def setUp(self): ...`
`@pytest.fixture`	Reusable test setup	`@pytest.fixture def obj():`
`@pytest.mark.parametrize`	Many inputs, one test	`@pytest.mark.parametrize(...)`
`breakpoint()`	Start interactive debugger	`breakpoint() in code`
`pdb commands`	Navigate debug session	`n, s, c, p var, q`
`logging`	Production-safe debug output	`logging.debug/info/warning()`
`time.perf_counter()`	Precise timing	`start = perf_counter()`
`timeit`	Accurate benchmarking	`timeit.timeit(code, number=N)`
`cProfile`	Full function profiler	`python -m cProfile script.py`
`@lru_cache`	Cache function results	`from functools import lru_cache`

Happy Coding & Testing! ■■