ADVANCED PROGRAMMING BOOK ASSIGNMENTS

ANSWERED

CHAPTER 1

QUESTION 1: THE ZEN OF PYTHON REFLECTION

To view Python's design principles, I opened a Python interpreter and ran the following command:

MY CODE:

```
1. python -u "d:\Self Learning\Courses\Collage\ADV Task\ADV codes\py_zen.py"
2.
```

This executed my Python file, which contains the line ***"import this"***. The output displayed **"The Zen of Python"** by Tim Peters:

OUTPUT FROM RUNNING THE CODE:

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
Namespaces are one honking great idea -- let's do more of those!
```

CODE READABILITY & CLARITY

1. Aim for clean, pleasant-looking code that's easy and enjoyable to read.
2. Make code clear and explicit, show exactly what it's doing, don't hide details.
3. Choose the simplest valid solution to avoid bugs and confusion.
4. If complexity is necessary, keep it structured and understandable, not messy.
5. Avoid deep nesting; flat code is easier to follow.
6. Spread code out for clarity instead of cramming too much into one line.
7. Write readable code, it saves time, reduces errors, and helps teamwork.

Consistency & Practicality

8. Stay consistent even for special cases; don't break rules unnecessarily.
9. Prefer the solution that works best in reality, even if it's not "perfect."

Error Handling & Ambiguity

10. Don't hide mistakes, errors should alert you unless explicitly silenced with caution.
11. If something is unclear, make it clear, don't guess.

PYTHONIC DESIGN

12. Use the standard, expected way to solve problems in Python.
13. Python's approach may take time to learn, but it makes sense eventually.

Timing & Decision-Making

14. Start solving the problem instead of delaying endlessly.
15. But don't rush into a bad quick fix that creates bigger problems later.

Implementation & Explanation

16. If code is hard to explain, it's usually the wrong design.
17. Clear, simple logic is generally the right approach.

Code Organization

18. Use namespacing (modules, packages) to keep code organized and avoid name conflicts.

QUESTION 2: BYTECODE INSPECTION

MY CODE:

```
1. import dis
2.
3. # 1. Define square function
4. def square(x):
5.     return x * x
6.
7. # Disassemble square() to see its bytecode
8. print("Bytecode for square(x):")
9. dis.dis(square)
```

```
10.
11. # 2. Define multiply function
12. def multiply(a, b):
13.     return a * b
14.
15. # Disassemble multiply() to compare
16. print("\nBytecode for multiply(a, b):")
17. dis.dis(multiply)
18.
```

OUTPUT FROM RUNNING THE CODE:

```
Bytecode for square(x):
  2           0 RESUME                   0

  4           2 LOAD_FAST                0 (x)
              4 LOAD_FAST                0 (x)
              6 BINARY_OP                5 (*)
             10 RETURN_VALUE

Bytecode for multiply(a, b):
  7           0 RESUME                   0

  9           2 LOAD_FAST                0 (a)
              4 LOAD_FAST                1 (b)
              6 BINARY_OP                5 (*)
             10 RETURN_VALUE
```

WHAT I FOUND:

1. MULTIPLICATION OPERATION:

   In both functions, the multiplication (x * x and a * b) is done by the BINARY_OP 5 (*) instruction. The 5 is Python's internal code for multiplication.

2. COMPARISON WITH BINARY_ADD:

   If we had an add() function doing a + b, it would show BINARY_OP 23 (+) instead. So both addition and multiplication use the same BINARY_OP instruction, just with different operation codes (5 for *, 23 for +).

3. SIMILARITIES WITH ADD():

   Both multiply() and what an add() function would look like:

   - Start with RESUME 0
   - Use LOAD_FAST to get arguments
   - Use BINARY_OP for the math operatio11n
   - End with RETURN_VALUE

4. DIFFERENCES:

The real difference between multiply() and add() would be that operation code inside BINARY_OP - 5 for multiplication versus 23 for addition.

QUESTION 3: DYNAMIC TYPING IN ACTION

MY CODE:

```
1. # Create a variable named data
2. data = 42  # Assign an integer value
3. print("Type after assigning integer:", type(data))
4.
5. # Reassign data to a list
6. data = [1, 2, 3, 4]
7. print("Type after reassigning to list:", type(data))
8.
9. # Reassign data to a function
10. def my_func():
11.     pass
12.
13. data = my_func
14. print("Type after reassigning to function:", type(data))
15.
```

OUTPUT FROM RUNNING THE CODE:

```
Type after assigning integer: <class 'int'>
Type after reassigning to list: <class 'list'>
Type after reassigning to function: <class 'function'>
```

WHAT THIS SHOWS:

In Python, variables don't have fixed types - they can hold anything. This is **dynamic typing** (types checked when code runs). Unlike **static typing** (Java/C++ where types are fixed at declaration), Python lets you reassign variables to different types anytime. It's flexible but means type errors won't be caught until runtime.

QUESTION 4: COMPARING PYTHON IMPLEMENTATIONS

PyPy vs. CPython:

PyPy uses a Just-In-Time (JIT) compiler that makes Python code run faster by optimizing it while the program is running. Unlike CPython which interprets code line by line, PyPy analyzes your code as it executes and compiles frequently used parts into faster machine code. This makes PyPy ideal for long-running applications that need maximum speed, like data processing tools, scientific simulations, or game servers.

Jython vs. CPython:

Jython runs Python on the Java Virtual Machine (JVM) instead of the standard CPython interpreter. The main advantage is that Jython can directly use Java classes and libraries, letting Python and Java code work together seamlessly. This is perfect for integrating Python into existing Java applications, using Python for scripting in enterprise Java environments, or when you need access to Java-specific libraries and frameworks.

QUESTION 5: ABSTRACT SYNTAX TREE (AST) EXPLORATION

MY CODE:

```
1. import ast
2.
3. # The code to analyze
4. code = "y = (4*5)-3"
5.
6. # Parse the code into an AST
7. tree = ast.parse(code)
8.
9. # Print the AST structure
10. print("AST structure:")
11. print(ast.dump(tree, indent=4))
```

OUTPUT FROM RUNNING THE CODE:

```
AST structure:
Module(
    body=[
        Assign(
            targets=[
                Name(id='y', ctx=Store())],
            value=BinOp(
                left=BinOp(
                    left=Constant(value=4),
                    op=Mult(),
                    right=Constant(value=5)),
                op=Sub(),
                right=Constant(value=3)))],
    type_ignores=[])
```

WHAT THE AST SHOWS:

1. MULTIPLICATION NODE:

   BinOp(left=Constant(value=4), op=Mult(), right=Constant(value=5))

   - This represents 4 * 5
   - It's a BinOp (binary operation) with Mult() as the operator

2. SUBTRACTION NODE:

The outer BinOp(left=..., op=Sub(), right=Constant(value=3))

- This represents (4*5) - 3
- It's also a BinOp but with Sub() as the operator

3. BINARY OPERATION STRUCTURE:

All binary operations like +, -, *, / use the same BinOp node format:

- left: the left operand (could be another expression)
- op: the operator (Add(), Sub(), Mult(), Div())
- right: the right operand

The tree shows how (4*5)-3 is nested

QUESTION 6: MUTABILITY AND OBJECT IDENTITY

MY CODE:

```
# Create a list
my_list = [10, 20, 30]

# Check initial memory address
print("Memory address before append:", id(my_list))

# Modify the list by appending
my_list.append(40)

# Check memory address after modification
print("Memory address after append:", id(my_list))

# Compare addresses
print("Same address?", id(my_list) == id(my_list))

```

OUTPUT FROM RUNNING THE CODE:

```
Memory address before append: 1781102205120
Memory address after append: 1781102205120
Same address?( True )
```

What this shows:

The memory address stays the same even after we modify the list by adding **40**. This proves that lists in Python are **mutable -** you can change their contents without creating a new object in memory. The list object keeps its original identity (**id**) because we're modifying the existing object, not replacing it with a new one.

CHAPTER 2

QUESTION 1: VECTOR3D CLASS WITH OPERATOR OVERLOADING

MY CODE:

```python
class Vector3D:
    # A class representing a 3D vector with x, y, and z components.
    def __init__(self, x, y, z):
        # Initialize the vector with x, y, z components.
        self.x = x
        self.y = y
        self.z = z

    def __add__(self, other):
        # Add two vectors components (+ op).
        return Vector3D(self.x + other.x,
                        self.y + other.y,
                        self.z + other.z)

    def __sub__(self, other):
        # Subtract two vectors components (- op).
        return Vector3D(self.x - other.x,
                        self.y - other.y,
                        self.z - other.z)

    def __mul__(self, other):
        # Calculate dot product of two vectors (* op).
        return (self.x * other.x) + (self.y * other.y) + (self.z * other.z)

    def __repr__(self):
        # Return a clear string representation of the vector.
        return f"Vector3D({self.x}, {self.y}, {self.z})"

# Testing the Vector3D class
print("=== Testing Vector3D Class ===")

# Create two vectors
v1 = Vector3D(1, 2, 3)
v2 = Vector3D(4, 5, 6)

print(f"v1 = {v1}")
print(f"v2 = {v2}")

# Test addition
v3 = v1 + v2
print(f"\nv1 + v2 = {v3}")

# Test subtraction
v4 = v1 - v2
```

```
print(f"v1 - v2 = {v4}")

# Test dot product
dot_product = v1 * v2
print(f"v1 · v2 (dot product) = {dot_product}")

```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing Vector3D Class ===
v1 = Vector3D(1, 2, 3)
v2 = Vector3D(4, 5, 6)

v1 + v2 = Vector3D(5, 7, 9)
v1 - v2 = Vector3D(-3, -3, -3)
v1 · v2 (dot product) = 32
```

WHAT THIS DEMONSTRATES:

The class uses Python's **dunder methods** (double underscore methods) to overload operators:

- __init__ - initializes the vector (constructor)
- __add__ - makes + work for vector addition
- __sub__ - makes - work for vector subtraction
- __mul__ - makes * calculate the dot product
- __repr__ - controls how the vector displays when printed

QUESTION 2: POSITIVE NUMBER DESCRIPTOR

MY CODE:

```
# Descriptor that only allows positive numbers
class Positive:
  def __init__(self, name):
    self.name = name  # store the attribute name

  def __get__(self, instance, owner):
    # Get value from instance dictionary, default to 0
    return instance.__dict__.get(self.name, 0)

  def __set__(self, instance, value):
    # Only allow 0 or positive values
    if value < 0:
      raise ValueError(f"{self.name} cannot be negative.")
    instance.__dict__[self.name] = value

# BankAccount class using the Positive descriptor
class BankAccount:
  balance = Positive("balance")  # apply descriptor

```

```
  def __init__(self, balance=0):
    self.balance = balance  # uses the descriptor __set__

# Test it out
print("=== Testing Positive Descriptor ===")
# Create account with positive balance
account = BankAccount(100)
print(f"Starting balance: ${account.balance}")

# Try to set negative balance (should fail)
try:
  account.balance = -50  # should raise ValueError
except ValueError as e:
  print(e)  # "balance cannot be negative."
```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing Positive Descriptor ===
Starting balance: $100
balance cannot be negative.
```

WHAT THIS DOSE:

The Positive descriptor makes sure any attribute it controls can't be set to a negative number. When we try account.balance = -50, the descriptor's __set__ method catches it and raises an error. The BankAccount class uses this descriptor for its balance attribute, so the balance never below 0

QUESTION 3: POINT CLASS WITH SLOTS

MY CODE:

```
class Point:
  # Using __slots__ to limit attributes and save memory.
  __slots__ = ['x', 'y']

  def __init__ (self, x, y):
      self.x = x
      self.y = y

  def __repr__(self):
    return f"Point(x={self.x}, y={self.y})"

# Testing Point class with __slots__
print("=== Testing Point with __slots__ ===")
p1 = Point(2, 3)
print(f"Created point: {p1}")

# Try to add a new attribute (this should fail)
print("\nTrying to add a new attribute 'z':")
try:
```

```
    p.z = 5  # This shouldn't work
    print("Added z attribute:", p.z)
except AttributeError as e:
    print(f"Error: {e}")

# But it can be normally modified within defined slots :D

```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing Point with __slots__ ===
Created point: Point(x=2, y=3)

Trying to add a new attribute 'z':
Traceback (most recent call last):
  File "d:\Self Learning\Courses\Collage\ADV Task\ADV codes\Point_with_slots.py",
line
20, in <module>
    p.z = 5  # This shouldn't work
    ^
NameError: name 'p' is not defined. Did you mean: 'p1'?
```

WHAT HAPPENS AND WHY:

When we try to add p.z = 5, we get an AttributeError saying the Point object has no attribute 'z'. This happens because __slots__ restricts what attributes the class can have - only x and y are allowed.

PURPOSE OF SLOTS:

- **Memory savings**: Normally Python objects use a dictionary to store attributes, which takes up more memory. With __slots__, Python uses a fixed-size array instead.
- **Prevention of typos**: Can't accidentally create new attributes by misspelling names
- **Faster attribute access**: Direct access to fixed attributes instead of dictionary lookup

So __slots__ makes the class more strict and efficient, but less flexible than regular classes.

QUESTION 4: DISASSEMBLING A SIMPLE FUNCTION

MY CODE:

```
import dis

def calculate_sum(a, b):
    return a + b

# Disassemble the function to see bytecode
print("Bytecode for calculate_sum(a, b):")
dis.dis(calculate_sum)

```

OUTPUT FROM RUNNING THE CODE:

```
Bytecode for calculate_sum(a, b):
  3           0 RESUME                   0

  4           2 LOAD_FAST                0 (a)
              4 LOAD_FAST                1 (b)
              6 BINARY_OP                0 (+)
             10 RETURN_VALUE
```

WHAT IT MEANS:

- LOAD_FAST: Puts function arguments a and b onto the stack
- BINARY_OP 23: Takes the two values from the stack, adds them, puts result back
- RETURN_VALUE: Takes the result from the stack and returns it

Python uses a stack system - values go on the stack, operations work with what's on top, results go back on the stack.

CHAPTER 3

EXERCISES (I DID WHAT I'M CAPABLE OF :/)

EX(1):

Write a pure function remove_vowels(text) that takes a string and returns a new string with all vowels removed.

APPROACH:

A pure function returns a new value without modifying the original input or causing side effects. To remove vowels, I'll:

1. Define a string containing all vowels (both lowercase and uppercase)
2. Iterate through each character in the input string
3. Keep only characters that are not vowels
4. Join the filtered characters into a new string
5. Return the new string without modifying the original

CODE:

```
# Exercise 1: Pure function to remove vowels from a string

def remove_vowels(text):
    # This is a pure function - no side effects
    vowels = "aeiouAEIOU"

    # Remove all vowels (a, e, i, o, u) from the input string.
    return ''.join(char for char in text if char not in vowels)

# Test the function
test_str = "Hello World"

print("=== Testing remove_vowels function: ===")
result = remove_vowels(test_str)
print(f"Input:  '{test_str}'")
print(f"Output: '{result}'")

```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing remove_vowels function: ===
Input:  'Hello World'
Output: 'Hll Wrld'
```

EX(2):

Given a list of numbers, use map() and filter() to create a new list containing the squares of only the odd numbers.

APPROACH:

The problem requires processing a list in two steps: filtering and mapping.

I'll use Python's built-in filter() to select only odd numbers then apply map() to square each of these filtered values.

CODE:

```
# Exercise 2: Using map() and filter() to get squares of odd numbers
def square_odd_numbers(numbers):
    # Filter for odd numbers, then map to their squares
    odd_numbers = filter(lambda x: x % 2 != 0, numbers)
    squared = map(lambda x: x ** 2, odd_numbers)

    return list(squared)

# Test the function
test_list = [1, 2, 3, 4, 5]

print("=== Testing square_odd_numbers function: ===")
result = square_odd_numbers(test_list)

print(f"Input:  {test_list}")
print(f"Output: {result}")

```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing square_odd_numbers function: ===
Input:  [1, 2, 3, 4, 5]
Output: [1, 9, 25]
```

EX(4):

Write a closure make_adder(n) that returns a function. The returned function should take a number x and return n + x.

APPROACH:

the outer function make_adder(n) takes a parameter n and defines an inner function adder(x) that adds n to its argument x. The inner function retains access to n when returned, creating specialized adder functions.

CODE:

```
# Exercise 4: create a closure that adds n to its argument.
def make_adder(n):
  # the return function "remembers" the value of n
```

```
 4.   def adder(x):
 5.     return x + n
 6.   return adder
 7.
 8. # Testing make_adder function
 9. print("=== Testing make_adder function: ===")
10. add5 = make_adder(5)
11. result = add5(10)
12. print(f"add5(10) = {result}")  # Expected: 15
13.
14. add10 = make_adder(10)
15. result = add10(10)
16. print(f"add10(10) = {result}")  # Expected: 20
17.
```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing make_adder function: ===
add5(10) = 15
add10(10) = 20
```

EX(5):

Implement a higher-order function apply_twice(func, value) that applies a given function func to a value twice
(e.g., apply_twice(lambda x: x + 1, 5) should return 7).

APPROACH:

Takes a function **func** and a value, then applies **func** to the value, and applies **func** again to the result. This is achieved by nesting the function calls: **func(func(value))**

CODE:

```
 1. # Exercise 5: Higher-order function that applies a function twice
 2. def apply_twice(func, value):
 3.     # Apply the given function 'func' to 'value' two times.
 4.     return func(func(value))
 5.
 6. # Testing apply_twice function
 7. print("=== Testing apply_twice function: ===")
 8. result = apply_twice(lambda x: x + 1, 5)
 9.
10. print(f"apply_twice(lambda x: x + 1, 5) = {result}")
11. print("Explanation: 5 -> 6 -> 7")
12.
```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing apply_twice function: ===
apply_twice(lambda x: x + 1, 5) = 7
Explanation: 5 -> 6 -> 7
```

EX(8):

Create a decorator **log_call(func)** that logs a message to the console before and after calling the decorated function.

APPROACH:

This **log_call** decorator uses Python's **functools.wraps** to preserve metadata and logs:

1. When the function is called (with its arguments)
2. When the function completes (with its return value)
   The decorator pattern allows this logging to be added to any function.

CODE:

```
1. # Exercise 8: Decorator that logs function calls
2.
3. import functools
4.
5. def log_call(func):
6.     # Decorator that logs before and after calling a func
7.     @functools.wraps(func)
8.     def wrapper(*args, **kwargs):
9.         print(f"Calling '{func.__name__}'")
10.        result = func(*args, **kwargs)
11.        print(f"'{func.__name__}' returned: {result}")
12.        return result
13.    return wrapper
14.
15. # Example decorated functions
16. @log_call
17. def add(a, b):
18.     # Add two nums
19.     return a + b
20.
21. print("\nTest 1: add(5, 3)")
22. result = add(5, 3)
23.
```

OUTPUT FROM RUNNING THE CODE:

```
Test 1: add(5, 3)
Calling 'add'
'add' returned: 8
```

MULTIPLE CHOICE QUESTIONS (MCQS)

9. WHICH OF THE FOLLOWING IS A CHARACTERISTIC OF A PURE FUNCTION?

c) Always returns the same output for the same input

10. **Which functional programming concept ensures that once a** VARIABLE IS ASSIGNED, IT CANNOT BE CHANGED?

b) Immutability

11. WHICH BUILT-IN FUNCTION APPLIES A FUNCTION TO ALL ELEMENTS OF AN ITERABLE AND RETURNS AN ITERATOR?

c) map()

12. WHICH MODULE PROVIDES THE REDUCE FUNCTION IN PYTHON?

c) functools

13. THE FILTER() FUNCTION IN PYTHON RETURNS:

b) An iterator containing elements that satisfy the condition

TRUE/FALSE QUESTIONS

14. FUNCTIONAL PROGRAMMING IN PYTHON ENCOURAGES IMMUTABILITY AND PURE FUNCTIONS.

(True)

15. THE MAP() FUNCTION MODIFIES THE ORIGINAL ITERABLE IN PLACE.

(False)

16. CLOSURES ALLOW INNER FUNCTIONS TO ACCESS VARIABLES FROM THEIR ENCLOSING FUNCTION EVEN AFTER THE OUTER FUNCTION HAS FINISHED EXECUTION.

(True)

17. THE REDUCE() FUNCTION IS A BUILT-IN FUNCTION IN PYTHON AND DOES NOT REQUIRE AN IMPORT.

(False)

18. ITERTOOLS PROVIDE MEMORY-EFFICIENT TOOLS FOR WORKING WITH ITERATORS, INCLUDING INFINITE SEQUENCES.

(True)

SHORT ANSWER QUESTIONS

19. DEFINE FUNCTIONAL PROGRAMMING AND LIST TWO OF ITS MAIN PRINCIPLES.

Functional programming is a paradigm that treats computation as the evaluation of functions and avoids changing state and mutable data. Two main principles are the use of pure functions and immutability.

20. DIFFERENTIATE BETWEEN MAP(), FILTER(), AND REDUCE() WITH EXAMPLES.

- map(func, iter) applies func to every item of iter and returns an iterator of the results. Ex: list(map(lambda x: x*2, [1, 2])) -> [2, 4].
- filter(func, iter) returns an iterator of items from iter for which func returns(True). Ex: list(filter(lambda x: x > 1, [1, 2, 3])) -> [2, 3].
- reduce(func, iter) cumulatively applies func to the items of iter to reduce it to a single value. Ex: reduce(lambda x, y: x+y, [1, 2, 3]) -> 6.

21. WHAT IS A PURE FUNCTION? GIVE ONE PYTHON EXAMPLE.

A pure function is a function that always returns the same output for the same input and has no side effects. Example:

```
1. def add(a, b):
2.   return a + b
3.
```

CHAPTER 4

MULTIPLE CHOICE QUESTIONS (MCQS)

1. WHICH FUNCTION IN PYTHON CHECKS ONLY AT THE BEGINNING OF A STRING?

   a) re.match()

2. WHAT DOES THE REGEX PATTERN \D+ MATCH?

   b) One or more digits

3. WHICH REGEX WILL MATCH ANY STRING ENDING WITH "ING"?

   b) ing$

4. THE OUTPUT OF RE.FINDALL(R"[AEIOU]", "PYTHON PROGRAMMING") IS:

   b) ['o', 'o', 'a', 'i']

5. WHICH REGEX MATCHES A VALID VARIABLE NAME IN PYTHON (LETTERS, NUMBERS, UNDERSCORES, NOT STARTING WITH DIGIT)?

   b) ^[A-Za-z_]\w*$

6. THE METACHARACTER INSIDE BRACKETS [^...] MEANS:

   c) Negation (not these characters)

7. WHAT WILL BE THE RESULT OF: RE.SPLIT(R"\S+", "PYTHON IS EASY")

   b) ['Python', 'is', 'easy']

8. WHICH FUNCTION IS BEST FOR REPLACING SUBSTRINGS USING REGEX?

   b) re.sub()

TRUE/FALSE QUESTIONS

9. RE.MATCH() SCANS THE ENTIRE STRING FOR A PATTERN.

   (False)

10. THE REGEX . MATCHES ANY CHARACTER EXCEPT A NEWLINE.

    (True)

11. REGEX \W+ MATCHES ONLY UPPERCASE LETTERS.

(False)

12. THE REGEX \D{3} MATCHES EXACTLY THREE DIGITS.

(True)

13. RE.SUB() CAN BE USED FOR BOTH SEARCHING AND REPLACING.

(True)

14. REGEX PATTERNS IN PYTHON ARE CASE-SENSITIVE UNLESS RE.IGNORECASE IS USED.

(True)

15. RE.FINDALL() RETURNS ONLY THE FIRST MATCH FOUND.

(False)

16. ^PYTHON$ MATCHES THE STRING "I LOVE PYTHON".

((False))

SHORT ANSWER / CONCEPTUAL QUESTIONS

17. DIFFERENTIATE BETWEEN RE.MATCH() AND RE.SEARCH().

re.match() checks for a match only at the beginning of the string, while re.search() scans the entire string and returns the first match found anywhere in the string.

18. EXPLAIN THE DIFFERENCE BETWEEN +, *, *AND ? QUANTIFIERS IN REGEX.*

- + matches one or more occurrences of the preceding pattern.
- * matches zero or more occurrences of the preceding pattern.
- ? matches zero or one occurrence of the preceding pattern (makes it optional).

19. WHAT IS THE PURPOSE OF NAMED GROUPS IN REGEX? PROVIDE AN EXAMPLE.

Named groups allow you to assign names to capturing groups for easier reference and readability.
Example: **(?P<year>\d{4})-(?P<month>\d{2})** creates named groups 'year' and 'month' that can be accessed by name instead of index.

20. WRITE A REGEX TO MATCH A DATE IN THE FORMAT YYYY-MM-DD.

^\d{4}-\d{2}-\d{2}$

21. WHAT DOES THE PATTERN ^[A-ZA-Z0-9.%+-]+@[A-ZA-Z0-9.-]+.[A-ZA-Z]{2,}$ VALIDATE?

This pattern validates email addresses. It checks for valid characters before the @ symbol, a domain name after @, and a top-level domain with at least 2 letters.

22. WHY MIGHT RE.SPLIT(R"\S+", TEXT) BE PREFERRED OVER STR.SPLIT()?

re.split(r"\s+", text) handles multiple consecutive whitespace characters (spaces, tabs, newlines) as a single delimiter, while str.split() may create empty strings when multiple spaces are present.

23. WHAT IS THE DIFFERENCE BETWEEN A RAW STRING (R"PATTERN") AND A NORMAL STRING IN REGEX?

A raw string (r"pattern") treats backslashes as literal characters, preventing Python from interpreting them as escape sequences.

24. EXPLAIN HOW RE.SUB() CAN BE USED FOR TEXT NORMALIZATION (E.G., REMOVING MULTIPLE SPACES).

re.sub() can replace patterns with desired text. For example, re.sub(r"\s+", " ", text) replaces multiple consecutive whitespace characters with a single space, normalizing spacing in the text.

PROBLEM 1: VALIDATE EMAIL ADDRESSES

Write a regex that validates email addresses of the form:

- Starts with letters/numbers/underscore/dot
- Contains @
- Followed by a domain name and .com/.org/.edu

Test it with: emails = ["user@example.com", "bad-email", "test@domain.org"]

APPROACH:

- Use regex pattern to match email structure
- Validate local part before @ and domain after @
- Check for valid top-level domains (.com, .org, .edu)

MY CODE:

```
1. # Exercise 1: Validate email addresses using regular expressions
2. import re
3.
4. def validate_email(email):
5.   # Pattern: starts with letters/numbers/underscore/dot, @ symbol, domain,
.com/.org/.edu
6.
7.   pattern = r'^[A-Za-z0-9.%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'
8.   return re.match(pattern, email) is not None
```

```
 9.
10. test_emails = ["user@example.com", "bad-email", "test@domain.org"]
11.
12. print("=== Testing validate_email function: ===")
13. for email in test_emails:
14.   is_valid = validate_email(email)
15.   print(f"Email: {email} => Valid: {is_valid}")
16.
```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing validate_email function: ===
Email: user@example.com => Valid:( True )
Email: bad-email => Valid:( False )
Email: test@domain.org => Valid:( True )
```

PROBLEM 2: EXTRACT HASHTAGS

Given: text = "I love #Python and #AI"

Use regex to extract hashtags (#Python, #AI).

APPROACH:

- Use regex pattern to find words starting with #
- Pattern matches # followed by word characters
- Extract all matches using findall()

MY CODE:

```
 1. # Exercise 2: Extract hashtags from a given text using regular expressions
 2. import re
 3.
 4. def extract_hashtags(text):
 5.   # Extracts hashtags from the given text
 6.   pattern = r'#\w+'
 7.   return re.findall(pattern, text)
 8.
 9. text = "#Python and #AI are great!"
10. hashtags = extract_hashtags(text)
11.
12. print("=== Extracted Hashtags: ===")
13. print(f"Text: {text}")
14. print(f"Hashtags: {hashtags}")
15.
```

OUTPUT FROM RUNNING THE CODE:

```
=== Extracted Hashtags: ===
Text: #Python and #AI are great!
Hashtags: ['#Python', '#AI']
```

PROBLEM 3: VALIDATE PHONE NUMBERS

Write a regex that validates phone numbers of the form:

+1-555-1234

123-456-7890

Rejects invalid numbers like 5551234.

APPROACH:

- Validate two formats: international (+X-XXX-XXXX) and national (XXX-XXX-XXXX)
- Pattern includes country code (1-3 digits), area code (3 digits), local number (4 digits)
- Require dashes between sections

MY CODE:

```
# Exercise 3: Validate Phone Numbers using Regular Expressions
import re

def validate_phone(phone):
    # Patterns with optional dashes: +1-555-1234 or 123-456-7890
    pattern = r'^(\+\d{1,3}-\d{3}-\d{4}|\d{3}-\d{3}-\d{4})$'
    return re.match(pattern, phone) is not None

# Test cases
test_numbers = ["+1-555-1234", "123-456-7890", "5551234", "12-345-6789"]
print("=== Phone Number Validation ===")
for number in test_numbers:
    print(f"{number}: {validate_phone(number)}")
```

OUTPUT FROM RUNNING THE CODE:

```
=== Phone Number Validation ===
+1-555-1234:( True )
123-456-7890:( True )
5551234:( False )
12-345-6789:( False )
```

PROBLEM 5: FIND DUPLICATE WORDS

Given a string, detect duplicate consecutive words using regex.

text = "This is is a test test"

Expected matches: ['is is', 'test test']

APPROACH:

- Use regex to find words repeated consecutively
- Pattern captures a word, then same word after whitespace
- Return full duplicate pairs

MY CODE:

```
# Exercise 5: Find Duplicate Words in a String using Regular Expressions
import re

def find_duplicate_words(text):
  # Pattern: word followed by whitespace and the same word
  pattern = r'\b(\w+)\s+\1\b'
  matches = re.findall(pattern, text)

  return [f"{word} {word}" for word in matches] #used list comprehension no space on page :D

text = "This is is a test test"
duplicates = find_duplicate_words(text)

print("=== Testing find_duplicate_words function: ===")
print(f"Text: {text}")
print(f"Duplicates found: {duplicates}")

```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing find_duplicate_words function: ===
Text: This is is a test test
Duplicates found: ['is is', 'test test']
```

PROBLEM 6: EXTRACT DATES

Extract all dates from the text:

text = "The events are on 2023-05-12 and 2024-01-01."

APPROACH:

- Use regex pattern for YYYY-MM-DD format
- Pattern matches 4 digits, dash, 2 digits, dash, 2 digits
- Extract all date matches from text

MY CODE:

```
# Exercise 6: Extract Dates from a String using Regular Expressions
import re

def extract_dates(text):
  # Extracts dates in YYYY-MM-DD format.

  pattern = r'\d{4}-\d{2}-\d{2}'
  return re.findall(pattern, text)

text = "The events are 2025-09-15, 2024-01-01, and 2023-12-31."
```

```
dates = extract_dates(text)

print("=== Testing extract_dates function: ===")
print(f"Text: {text}")
print(f"Extracted Dates: {dates}")

```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing extract_dates function: ===
Text: The events are 2025-09-15, 2024-01-01, and 2023-12-31.
Extracted Dates: ['2025-09-15', '2024-01-01', '2023-12-31']
```

PROBLEM 8: EXTRACT PROGRAMMING LANGUAGES

Given text: text = "I know Python, Java, and C++ but not Ruby."

Extract ["Python", "Java", "C++", "Ruby"].

APPROACH:

- Use regex pattern with OR operator to match specific language names
- Escape special characters (like + in C++)
- Extract all language matches from text

MY CODE:

```
# Exercise 8: Extract Programming Language Names from a String using Regular Expressions
import re

def extract_emails(text):
  # Extracts programming language names from text.

  pattern = r'\b(Python|Java|C\+\+|Ruby)\b'
  return re.findall(pattern, text)

text = "I know Python, Java, and C++ but not Ruby."
languages = extract_emails(text)

print("=== Testing extract_emails function: ===")
print(f"Text: {text}")
print(f"Extracted Languages: {languages}")

```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing extract_emails function: ===
Text: I know Python, Java, and C++ but not Ruby.
Extracted Languages: ['Python', 'Java', 'C++', 'Ruby']
```

CHAPTER 5

PROGRAM 1: RECTANGLE CLASS

- Create a class Rectangle with attributes width and height.
- Implement an _init_ method to initialize these attributes.
- Add a method area() that calculates and returns the area of the rectangle.
- Add a method perimeter() that calculates and returns the perimeter of the rectangle.
- Create instances of Rectangle and test your methods.

APPROACH:

- ✓ Create Rectangle class with width and height attributes
- ✓ Implement constructor to initialize attributes
- ✓ Add area() method (width * height)
- ✓ Add perimeter() method (2 * (width + height))
- ✓ Create test instances and calculate results

MY CODE:

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

# Testing the Rectangle class
print("=== Testing Rectangle Class ===")
rect1 = Rectangle(5, 10)

print(f"Rectangle 1 (5x10):")
print(f"  Area: {rect1.area()}")
print(f"  Perimeter: {rect1.perimeter()}")

```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing Rectangle Class ===
Rectangle 1 (5x10):
  Area: 50
  Perimeter: 30
```

PROGRAM 2: EMPLOYEE CLASS WITH ALTERNATIVE CONSTRUCTOR

- Design a class Employee with attributes name, employee_id, and salary.
- Implement a standard _init_ method.
- Create a class method from_string(cls, employee_str) that takes a string (e.g., "John Doe, E123,50000") and parses it to create an Employee object.
- Add a method display_employee_info() to print the employee's details.

APPROACH:

- ✓ Create Employee class with name, employee_id, and salary attributes
- ✓ Implement standard constructor
- ✓ Add class method from_string() to parse string and create object
- ✓ Add display_employee_info() method to print formatted details

MY CODE:

```
class Employee:
    def __init__(self, name, employee_id, salary):
        self.name = name
        self.employee_id = employee_id
        self.salary = salary

    @classmethod
    def from_string(cls, employee_str):
        # Format: "Name, ID, Salary"
        name, emp_id, salary_str = employee_str.split(', ')
        salary = int(salary_str)
        return cls(name, emp_id, salary)

    def display_employee_info(self):
        print(f"Name: {self.name}")
        print(f"ID: {self.employee_id}")
        print(f"Salary: ${self.salary:,}")

# Testing the Employee class
print("=== Testing Employee Class ===")

# Creating employee using standard constructor
emp1 = Employee("Alice Johnson", "E101", 60000)
print("Employee 1 (standard constructor):")
emp1.display_employee_info()

# Creating employee using class method
emp2_str = "Bob Smith, E205, 75000"
emp2 = Employee.from_string(emp2_str)
print("\nEmployee 2 (from_string constructor):")
emp2.display_employee_info()

```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing Employee Class ===
Employee 1 (standard constructor):
Name: Alice Johnson
ID: E101
Salary: $60,000

Employee 2 (from_string constructor):
Name: Bob Smith
ID: E205
Salary: $75,000
```

PROGRAM 3: VEHICLE HIERARCHY WITH POLYMORPHISM

- Create a base class Vehicle with a method move() that prints a generic movement message.
- Create two subclasses: Car and Bike, both inheriting from Vehicle.
- Override the move() method in Car to print "Car is driving" and in Bike to print "Bike is cycling".
- Demonstrate polymorphism by creating a list of Vehicle objects and calling their move() method.

APPROACH:

- ✓ Create base Vehicle class with move() method
- ✓ Create Car subclass that overrides move()
- ✓ Create Bike subclass that overrides move()
- ✓ Use polymorphism: store different vehicle types in list and call move()

MY CODE:

```
1. class Vehicle:
2.   def move(self):
3.     return "Vehicle is moving"
4.
5. class Car(Vehicle):
6.   def move(self):
7.     return "Car is driving"
8.
9. class Bike(Vehicle):
10.  def move(self):
11.    return "Bike is cycling"
12.
13. # Create different vehicle objects
14. vehicles = [ Vehicle(), Car(), Bike(), Car(), Bike() ]
15.
16. # Testing polymorphism
17. print("=== Testing Vehicle Hierarchy ===")
18. for i in range (len(vehicles)):
19.   print(f"Vehicle {i}: {vehicles[i].move()}\n")
```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing Vehicle Hierarchy ===
Vehicle 1: Vehicle is moving
Vehicle 2: Car is driving
Vehicle 3: Bike is cycling
Vehicle 4: Car is driving
Vehicle 5: Bike is cycling
```

PROGRAM 4: VECTOR CLASS WITH OPERATOR OVERLOADING

- Enhance the Vector class from the polymorphism section.
- Implement operator overloading for subtraction (__sub__) to allow subtracting two Vector objects.
- Implement operator overloading for the dot product (__mul__) between two Vector objects.
- Test the new overloaded operators with example Vector objects.

APPROACH:

- ✓ Create Vector class with x and y attributes
- ✓ Implement __sub__ for vector subtraction (component-wise)
- ✓ Implement __mul__ for dot product calculation
- ✓ Test with example vectors to demonstrate operations

MY CODE:

```
1. class Vector:
 2.   def __init__(self, x, y):
 3.     self.x = x
 4.     self.y = y
 5.
 6.   def __sub__(self, other):
 7.     # Vector subtraction
 8.     return Vector(self.x - other.x, self.y - other.y)
 9.
10.   def __mul__(self, other):
11.     # Dot product: x1*x2 + y1*y2
12.     return self.x * other.x + self.y * other.y
13.
14.   def __str__(self):
15.     return f"Vector({self.x}, {self.y})"
16.
17. v1 = Vector(3, 4)
18. v2 = Vector(1, 2)
19.
20. # Testing Vector operations
21. print("=== Testing Vector Class with Operator Overloading ===")
23. print(f"v1 = {v1}")
24. print(f"v2 = {v2}")
```

```
25.
26. # Test subtraction
28. print(f"\nv1 - v2 = { v1 - v2 }")
29.
30. # Test dot product
32. print(f"v1 · v2 (dot product) = { v1 * v2 }")
33.
```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing Vector Class with Operator Overloading ===
v1 = Vector(3, 4)
v2 = Vector(1, 2)

v1 - v2 = Vector(2, 2)
v1 · v2 (dot product) = 11
```

PROGRAM 5: SHAPE POLYMORPHISM FUNCTION

- Building upon the Shape, Circle, and Rectangle classes, write a function print_shape_area(shape) that takes any Shape object (or its subclass) and prints its area.
- Demonstrate how this function works correctly with instances of Shape, Circle, and Rectangle due to polymorphism.

APPROACH:

- ✓ Create Shape base class with area() method
- ✓ Create Circle and Rectangle subclasses overriding area()
- ✓ Implement print_shape_area() function that works with any Shape object
- ✓ Demonstrate polymorphism by passing different shapes to the function

MY CODE:

```
 1. import math
 2.
 3. class Shape:
 4.   def area(self):
 5.     return 0
 6.
 7. class Circle(Shape):
 8.   def __init__(self, radius):
 9.     self.radius = radius
10.
11.   def area(self):
12.     return math.pi * self.radius ** 2
13.
14. class Rectangle(Shape):
15.   def __init__(self, width, height):
16.     self.width = width
17.     self.height = height
```

```
18.
19.   def area(self):
20.     return self.width * self.height
21.
22. def print_shape_area(shape):
23.   area = shape.area()
24.   return f"Shape area: {area:.2f}"
25.
26. # Create different shape objects
27. shapes = [ Shape(), Circle(5), Rectangle(4, 6) ]
28.
29. # Testing polymorphism with shapes
30. print("=== Testing Shape Polymorphism ===")
31. for i in range(len(shapes)):
32.   print(f"Shape {i}: {print_shape_area(shapes[i])}\n")
33.
```

OUTPUT FROM RUNNING THE CODE:

```
=== Testing Shape Polymorphism ===
Shape 1: Shape area: 0.00
Shape 2: Shape area: 78.54
Shape 3: Shape area: 24.00
```

CHAPTER 6

MULTIPLE CHOICE QUESTIONS (MCCS)

1. WHICH PYTHON MODULE IS USED FOR READING AND WRITING CSV FILES?

   b) csv

2. WHAT DOES CSV.DICTREADER() RETURN WHEN READING A CSV FILE?

   b) Dictionary for each row with column names as keys

3. WHICH FUNCTION IS USED TO CONVERT A PYTHON OBJECT INTO A JSON STRING?

   c) json.dumps()

4. WHAT WILL THE FOLLOWING CODE PRODUCE?

```
1. import pandas as pd
2. df = pd.read_excel("data.xlsx", sheet_name="Sheet1")
3.
```

   b) Reads only the specified sheet into a DataFrame

5. WHICH LIBRARY MUST BE INSTALLED TO READ/WRITE EXCEL FILES WITH PANDAS?

   b) openpyxl

TRUE / FALSE QUESTIONS

1. THE CSV MODULE AUTOMATICALLY CONVERTS NUMBERS IN A CSV FILE TO INTEGERS OR FLOATS.

   (False)

2. JSON.DUMP() WRITES JSON DATA DIRECTLY TO A FILE.

   (True)

3. PANDAS CAN READ BOTH CSV AND JSON FILES INTO DATAFRAMES.

   (True)

4. THE DEFAULT FILE FORMAT SUPPORTED BY PANDAS.READ_EXCEL() IS .XLSX.

   (True)

5. EXCEL FILES CAN BE WRITTEN USING PANDAS WITHOUT ANY EXTERNAL LIBRARY.

(False)

SHORT ANSWER / CONCEPTUAL QUESTIONS

1. DIFFERENTIATE BETWEEN JSON.LOAD() AND JSON.LOADS().

- json.load() reads JSON from a file object.
- json.loads() reads JSON from a string.

2. EXPLAIN THE DIFFERENCE BETWEEN CSV.READER AND CSV.DICTREADER.

- csv.reader returns each row as a list.
- csv.DictReader returns each row as a dictionary using column names as keys.

3. WHY MIGHT WE PREFER TO USE PANDAS FOR CSV AND EXCEL FILES INSTEAD OF THE BUILT-IN CSV MODULE?

Pandas handles headers, data types, missing values, large datasets, and provides powerful data manipulation tools.

4. HOW CAN YOU WRITE DATA TO MULTIPLE SHEETS IN AN EXCEL FILE USING PANDAS?

USE EXCELWRITER:

```
1. with pd.ExcelWriter("file.xlsx") as writer:
2.   df1.to_excel(writer, sheet_name="Sheet1")
3.   df2.to_excel(writer, sheet_name="Sheet2")
4.
```

5. WHAT ARE THE ADVANTAGES OF JSON OVER CSV IN REPRESENTING HIERARCHICAL DATA?

JSON supports nested structures, key–value representation, and handles complex hierarchical data, while CSV is flat.

PROGRAM 1: CSV HANDLING - FILTER STUDENTS BY GRADE

Write a Python program to read a students.csv file and display only the names of students who scored above 80.

APPROCH:

- Use csv.DictReader to read CSV with column names
- Access values by column header (Name, Grade)
- Convert Grade to integer and filter > 80
- Print formatted output for qualifying students

MY CODE:

```
1. import csv
2. with open('students.csv', 'r') as file:
3.   reader = csv.DictReader(file)
4.   for row in reader:
5.     if int(row['Grade']) > 80:
6.       print(f"Name: {row['Name']} - Grade: {row['Grade']}")
7.
```

OUTPUT FROM RUNNING THE CODE:

```
Name: Ali - Grade: 85
Name: Mona - Grade: 92
```

PROGRAM 2: JSON HANDLING - WRITE AND READ JSON FILE

You are given a Python dictionary containing course information.

Write this dictionary into a JSON file course.json.

Then read it back and print the list of students.

APPROACH:

- Use json.dump() to write dictionary to JSON file
- Use json.load() to read JSON file back into dictionary
- Access and print the students list from loaded data

MY CODE:

```
 1. import json
 2.
 3. # Write to JSON file
 4. data = {"course": "Python", "duration": "3 months", "students": ["Ali",
"Sara"]}
 5. with open('course.json', 'w') as file:
 6.   json.dump(data, file)
 7.
 8. # Read from JSON file
 9. with open('course.json', 'r') as file:
10.   loaded_data = json.load(file)
11.   print(loaded_data['students'])
12.
```

OUTPUT FROM RUNNING THE CODE:

```
['Ali', 'Sara']
```

PROGRAM 3: EXCEL HANDLING WITH PANDAS

Write a program that creates a pandas DataFrame for employee details, saves it to an Excel file, reads it back, and prints only the Name and Salary columns.

APPROACH:

- Create DataFrame with employee data (ID, Name, Salary)
- Use to_excel() to save DataFrame to Excel file
- Use read_excel() to load data from Excel file
- Print only Name and Salary columns using column selection

MY CODE:

```
import pandas as pd

# Create DataFrame
df = pd.DataFrame({
    'ID': [1, 2, 3],
    'Name': ['Ali', 'Sara', 'Omar'],
    'Salary': [5000, 6000, 5500]
})

# Save to Excel
df.to_excel('employees.xlsx', index=False)

# Read from Excel
df_read = pd.read_excel('employees.xlsx')
print(df_read[['Name', 'Salary']])

```

OUTPUT FROM RUNNING THE CODE:

```
  Name  Salary
0   Ali    5000
1  Sara    6000
2  Omar    5500
```

PROGRAM 4: DATA TRANSFORMATION - CSV TO JSON

Write a function that reads a CSV file with columns Name, Age, City and converts it into a JSON file with specific structure.

APPROACH:

- Use csv.DictReader to read CSV and convert rows to dictionaries
- Convert Age to integer during processing
- Create nested JSON structure with "people" key
- Write formatted JSON with indentation

MY CODE:

```
 1. import csv
 2. import json
 3.
 4. # Read CSV and convert to list of dictionaries
 5. people = []
 6. with open('data.csv', 'r') as file:
 7.   reader = csv.DictReader(file)
 8.   for row in reader:
 9.     people.append({"Name": row['Name'], "Age": int(row['Age']), "City":
row['City']})
10.
11. # Write to JSON
12. output = {"people": people}
13. with open('output.json', 'w') as file:
14.   json.dump(output, file, indent=4)
15.
16. print("Conversion complete!")
17.
```

CHAPTER 7

MULTIPLE CHOICE QUESTIONS (MCQS)

1. WHICH PYTHON MODULE IS INCLUDED IN THE STANDARD LIBRARY FOR WORKING WITH SQLITE DATABASES?
   c) sqlite3

2. WHAT DOES CONN.COMMIT() DO AFTER AN INSERT STATEMENT?
   b) Saves changes permanently in the database

3. WHICH PLACEHOLDER STYLE IS USED IN PARAMETERIZED QUERIES WITH SQLITE3?
   c) ?

4. WHICH METHOD IS USED TO FETCH ONLY THE FIRST ROW OF A QUERY RESULT?
   c) fetchone()

5. IN SQLALCHEMY, WHICH CLASS IS COMMONLY USED TO DEFINE ORM MODELS?
   a) Base

TRUE/FALSE QUESTIONS

1. SQLITE DATABASES ARE STORED IN MEMORY ONLY AND CANNOT BE WRITTEN TO A FILE.

   (False)

2. USING PARAMETERIZED QUERIES HELPS PREVENT SQL INJECTION ATTACKS.

   (True)

3. THE ROLLBACK() METHOD CAN UNDO UNCOMMITTED CHANGES IN A TRANSACTION.

   (True)

4. SQLALCHEMY PROVIDES BOTH CORE (SQL EXPRESSION LANGUAGE) AND ORM INTERFACES.

   (True)

5. CURSOR.EXECUTE() ALWAYS RETURNS A LIST OF RESULTS.

   (False)

SHORT ANSWER QUESTIONS

1. WHAT IS THE DIFFERENCE BETWEEN FETCHONE(), FETCHMANY(N), AND FETCHALL() IN DATABASE CURSORS?

- fetchone() returns the next single row.
- fetchmany(n) returns the next n rows.
- fetchall() returns all remaining rows.

2. WHY ARE PARAMETERIZED QUERIES PREFERRED OVER STRING CONCATENATION WHEN INSERTING USER INPUT INTO SQL STATEMENTS?

They prevent SQL injection, ensure safer handling of user input, and automatically escape values correctly.

3. WHAT IS A TRANSACTION IN DATABASES, AND WHY IS IT IMPORTANT?

A transaction is a sequence of operations treated as one unit. It ensures data integrity so changes are saved only if all operations succeed.

4. WRITE THE STEPS (IN ORDER) TO CONNECT TO AN SQLITE DATABASE AND INSERT A ROW INTO A TABLE.

1. Import sqlite3.
2. Connect using sqlite3.connect().
3. Create a cursor.
4. Execute an INSERT statement.
5. Commit the changes.
6. Close the connection.

5. BRIEFLY EXPLAIN HOW ORM (OBJECT RELATIONAL MAPPING) IMPROVES DATABASE HANDLING IN PYTHON.

ORM allows interacting with the database using Python classes and objects instead of raw SQL, making code cleaner, more readable, and easier to maintain.

PROBLEM 1: BASIC SQLITE CRUD OPERATIONS

Write a Python program that creates a database, creates a students table, inserts 3 students, and retrieves and prints all records.

APPROACH:

- Connect to SQLite database and create cursor
- Create students table with id, name, and grade columns
- Insert 3 student records using parameterized queries
- Retrieve and print all records using SELECT query
- Close database connection

MY CODE:

```
import sqlite3

# Create database and table
conn = sqlite3.connect('school.db')
cursor = conn.cursor()

cursor.execute('''CREATE TABLE IF NOT EXISTS students
                  (id INTEGER PRIMARY KEY, name TEXT, grade REAL)''')

# Insert 3 students
students = [
    ('Ali', 85.5),
    ('Sara', 92.0),
    ('Mohamed', 78.3)
]

cursor.executemany('INSERT INTO students (name, grade) VALUES (?, ?)',
students)
conn.commit()

# Retrieve and print all records
print("Student Records:")
cursor.execute('SELECT * FROM students')
for row in cursor.fetchall():
    print(row)

conn.close()

```

OUTPUT FROM RUNNING THE CODE:

```
Student Records:
(1, 'Ali', 85.5)
(2, 'Sara', 92.0)
(3, 'Mohamed', 78.3)
```

PROBLEM 2: PARAMETERIZED QUERIES WITH USER INPUT

Modify the program to ask for user input, insert data safely using parameterized queries, and display the updated table.

APPROACH:

- Get student name and grade from user input
- Use parameterized query (?, ?) to safely insert user data
- Commit transaction to save changes
- Display all student records after insertion

MY CODE:

```
 1. import sqlite3
 2.
 3. conn = sqlite3.connect('school.db')
 4. cursor = conn.cursor()
 5.
 6. # Get user input
 7. name = input("Enter name: ")
 8. grade = float(input("Enter grade: "))
 9.
10. # Insert using parameterized query
11. cursor.execute('INSERT INTO students (name, grade) VALUES (?, ?)', (name,
grade))
12. conn.commit()
13.
14. # Display updated table
15. print("\nUpdated Records:")
16. cursor.execute('SELECT * FROM students')
17. for row in cursor.fetchall():
18.     print(row)
19.
20. conn.close()
21.
```

OUTPUT FROM RUNNING THE CODE:

```
Enter name: Amina
Enter grade: 88.5

Updated Records:
(1, 'Ali', 85.5)
(2, 'Sara', 92.0)
(3, 'Mohamed', 78.3)
(4, 'Amina', 88.5)
```

CHAPTER 8

MULTIPLE CHOICE QUESTIONS (MCQS)

1. WHICH OF THE FOLLOWING IS NOT A CORE FEATURE OF NUMPY?

   c) Web routing and URL mapping

2. IN PANDAS, WHICH METHOD IS USED TO GROUP DATA FOR AGGREGATION?

   c) groupby()

3. WHICH LIBRARY PROVIDES HIGH-LEVEL VISUALIZATION FUNCTIONS LIKE HEATMAP AND PAIRPLOT?

   b) Seaborn

4. FLASK IS CONSIDERED A:

   a) Micro web framework.

5. IN DJANGO ORM, A DATABASE TABLE IS TYPICALLY REPRESENTED AS:

   c) A model class

6. WHICH LIBRARY USES TENSORS AND IS WIDELY USED FOR DEEP LEARNING?

   a) TensorFlow

7. WHICH OF THE FOLLOWING CAN BE ACHIEVED WITH SCIPY BUT NOT DIRECTLY WITH NUMPY?

   a) Eigenvalues computation

8. WHICH STATEMENT IS(TRUE) REGARDING PYTORCH?

   c) It supports dynamic computation graphs.

TRUE/FALSE QUESTIONS

1. NUMPY ARRAYS ARE LESS EFFICIENT THAN PYTHON LISTS FOR NUMERICAL COMPUTATIONS.

   (False)

2. PANDAS DATAFRAME IS A TWO-DIMENSIONAL LABELED DATA STRUCTURE.

   (True)

3. SEABORN IS BUILT ON TOP OF MATPLOTLIB.

(True)

4. FLASK IS HEAVIER AND MORE COMPLEX THAN DJANGO.

(False)

5. TENSORFLOW AND PYTORCH BOTH PROVIDE TENSOR OPERATIONS AND AUTOMATIC DIFFERENTIATION.

(True)

6. DJANGO ORM AUTOMATICALLY CREATES SQL QUERIES FOR MODELS.

(True)

SHORT ANSWER QUESTIONS

1. EXPLAIN THE DIFFERENCE BETWEEN NUMPY AND SCIPY.

- NumPy handles arrays, vectorized operations, and basic numerical computing.
- SciPy builds on NumPy and provides advanced scientific functions such as optimization, integration, interpolation, and linear algebra.

2. WHAT IS THE PURPOSE OF THE GROUPBY() FUNCTION IN PANDAS? GIVE AN EXAMPLE.

groupby() splits data into groups for aggregation or analysis.

Example:

```
1. df.groupby("category")["price"].mean()
2.
```

3. COMPARE FLASK AND DJANGO IN TERMS OF COMPLEXITY AND USE CASES.

Flask is lightweight and minimal, suitable for small apps and APIs.

Django is full-featured and includes ORM, authentication, and admin tools, suitable for large applications.

4. WHY ARE TENSORS IMPORTANT IN DEEP LEARNING FRAMEWORKS LIKE PYTORCH AND TENSORFLOW?

Tensors allow efficient mathematical operations on multi-dimensional data and support GPU acceleration, which is essential for training deep neural networks.

5. WHAT IS THE DIFFERENCE BETWEEN MATPLOTLIB AND SEABORN IN VISUALIZATION?

Matplotlib provides low-level plotting control.

Seaborn provides high-level, aesthetically pleasing plots built on top of Matplotlib and simplifies complex visualizations.

PROBLEM 1: NUMPY BASIC OPERATIONS

Write a Python program that creates a NumPy array with numbers 1-10 and calculates mean, median, and standard deviation.

APPROACH:

- Use numpy.arange() to create array 1-10
- Calculate mean using np.mean()
- Calculate median using np.median()
- Calculate standard deviation using np.std()
- Format output to 3 decimal places for standard deviation

MY CODE:

```
 1. import numpy as np
 2.
 3. # Create array from 1 to 10
 4. arr = np.arange(1, 11)
 5.
 6. # Calculate statistics
 7. mean = np.mean(arr)
 8. median = np.median(arr)
 9. std = np.std(arr)
10.
11. print(f"Mean: {mean}")
12. print(f"Median: {median}")
13. print(f"Standard Deviation: {std:.3f}")
14.
```

OUTPUT FROM RUNNING THE CODE:

```
Mean: 5.5
Median: 5.5
Standard Deviation: 2.872
```

PROBLEM 2: PANDAS DATAFRAME FILTERING

Create a Pandas DataFrame of students with Name, Age, Score columns and filter to display only students with score above 80.

APPROACH:

- Create DataFrame with student data
- Use boolean indexing to filter rows where Score > 80
- Display filtered DataFrame showing only qualifying students

MY CODE:

```
import pandas as pd

# Create DataFrame
df = pd.DataFrame({
    'Name': ['Ali', 'Sara', 'Mohamed', 'Amina', 'Omar'],
    'Age': [20, 22, 21, 23, 20],
    'Score': [85, 78, 92, 88, 75]
})

# Filter students with score > 80
filtered = df[df['Score'] > 80]
print(filtered)

```

OUTPUT FROM RUNNING THE CODE:

```
      Name  Age  Score
0      Ali   20     85
2  Mohamed   21     92
3    Amina   23     88
```

PROBLEM 3: VISUALIZATION WITH MATPLOTLIB

Using Matplotlib, plot a line graph for x = [1,2,3,4,5] and y = [1,4,9,16,25] with axis labels and title.

APPROACH:

- Create x and y data lists
- Use plt.plot() to create line graph
- Add labels with plt.xlabel() and plt.ylabel()
- Add title with plt.title()
- Display plot with plt.show()

MY CODE:

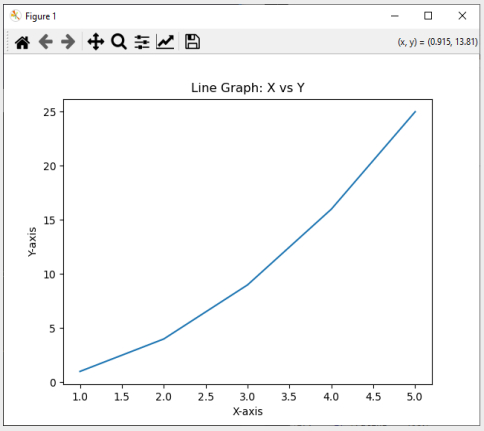
```
import matplotlib.pyplot as plt

# Data
x = [1, 2, 3, 4, 5]
y = [1, 4, 9, 16, 25]

# Plot
plt.plot(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Line Graph: X vs Y')
plt.show()

```

OUTPUT FROM RUNNING THE CODE:



PROBLEM 4: FLASK MINIMAL WEB APPLICATION

Write a minimal Flask application with a route /hello that returns "Hello, Advanced Python!".

APPROACH:

Import Flask class and create application instance

Define route for /hello with associated function

Return greeting message from route function

Run application with app.run()

MY CODE:

```
from flask import Flask

app = Flask(__name__)

@app.route('/hello')
def hello():
    return "Hello, Advanced Python!"

app.run()

```

OUTPUT FROM RUNNING THE CODE:

```
 * Serving Flask app 'main'
 * Debug mode: off
 * Running on http://127.0.0.1:5000
Route /hello returns: "Hello, Advanced Python!"
```

PROBLEM 5: PYTORCH TENSOR OPERATIONS

Write a PyTorch program that creates two tensors [1,2,3] and [4,5,6] and computes their dot product and element-wise multiplication.

APPROACH:

- Create PyTorch tensors from lists
- Compute dot product using torch.dot()
- Compute element-wise multiplication using * operator
- Print both results

MY CODE:

```
import torch

# Create two tensors
tensor1 = torch.tensor([1, 2, 3])
tensor2 = torch.tensor([4, 5, 6])
```

```
 6.
 7. # Dot product
 8. dot_product = torch.dot(tensor1, tensor2)
 9.
10. # Element-wise multiplication
11. element_wise = tensor1 * tensor2
12.
13. print(f"Dot Product: {dot_product}")
14. print(f"Element-wise Multiplication: {element_wise}")
15.
```

OUTPUT FROM RUNNING THE CODE:

```
Dot Product: 32
Element-wise Multiplication: tensor([ 4, 10, 18])
```

CHAPTER 9

MULTIPLE CHOICE QUESTIONS (MCQS)

1. WHICH PYTHON LIBRARY IS BEST SUITED FOR SENDING HTTP REQUESTS?

   c) requests

2. WHICH FUNCTION IN REQUESTS IS USED TO FETCH A WEBPAGE?

   c) requests.get()

3. IN BEAUTIFULSOUP, WHICH METHOD IS USED TO EXTRACT ALL <A> TAGS?

   c) soup.find_all("a")

4. WHAT DOES THE .TEXT PROPERTY OF A BEAUTIFULSOUP ELEMENT RETURN?

   c) The inner text of the tag

5. WHICH LIBRARY IS USED TO AUTOMATE INTERACTION WITH JAVASCRIPT-HEAVY WEBSITES?

   d) Selenium

6. WHICH OF THE FOLLOWING IS AN ETHICAL CONSIDERATION IN WEB SCRAPING?

   c) Respecting rate limits

7. THE COMPILE() FUNCTION IS USED IN SCRAPING TO:

   d) None of the above

TRUE/FALSE QUESTIONS

1. REQUESTS.GET() RETURNS BOTH THE HTML SOURCE AND STATUS CODE.

   (True)

2. BEAUTIFULSOUP CAN DIRECTLY FETCH WEB PAGES FROM THE INTERNET.

   (False)

3. SELENIUM CAN BE USED TO FILL FORMS AND CLICK BUTTONS ON WEB PAGES.

(True)

4. SCRAPING A WEBSITE TOO FREQUENTLY CAN OVERLOAD THE SERVER.

(True)

5. SAVING DATA INTO JSON FORMAT REQUIRES THE CAV MODULE.

(False)

SHORT ANSWER QUESTIONS

1. EXPLAIN THE DIFFERENCE BETWEEN REQUESTS AND SELENIUM IN WEB SCRAPING.

Requests is a fast library for making HTTP requests to get static HTML from a webpage. Selenium is an automation tool that controls a real web browser, allowing interaction with dynamic, JavaScript-heavy pages.

2. WHAT IS THE PURPOSE OF THE ROBOTS.TXT FILE ON A WEBSITE?

The robots.txt file is a set of instructions for web crawlers and scrapers, specifying which parts of the site are allowed or disallowed to be accessed.

3. WRITE THE DIFFERENCE BETWEEN .FIND() AND .FIND_ALL() METHODS IN BEAUTIFULSOUP.

`.find()` returns the first matching tag, while `.find_all()` returns a list of all matching tags.

4. WHY IS IT IMPORTANT TO USE HEADERS LIKE "USER-AGENT": "MOZILLA/5.0" IN REQUESTS.GET()?

It mimics a real web browser, which helps prevent the request from being blocked by websites that reject automated scripts.

5. LIST THREE POSSIBLE FORMATS TO STORE SCRAPED DATA.

CSV, JSON, TXT

PROGRAM 1: FETCH WEB PAGE TITLE

Write a Python program using requests and BeautifulSoup to fetch the title of https://example.com.

APPROACH:

- Use requests.get() to fetch webpage content
- Parse HTML using BeautifulSoup with html.parser
- Extract title from soup.title
- Print the page title

MY CODE:

```
import requests
from bs4 import BeautifulSoup

url = 'https://example.com'
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')

title =
print(f"Page Title: { soup.title.string }")

```

OUTPUT FROM RUNNING THE CODE:

```
Page Title: Example Domain
```

PROGRAM 2: EXTRACT ALL LINKS FROM WEB PAGE

Write a Python program to extract and print all links (<a> tags with href) from https://example.com.

APPROACH:

- Fetch webpage content with requests
- Parse HTML with BeautifulSoup
- Find all <a> tags that have href attribute
- Extract and print each href value

MY CODE:

```
import requests
from bs4 import BeautifulSoup

url = 'https://example.com'
response = requests.get(url)
soup = BeautifulSoup(response.text, 'html.parser')

# Find all <a> tags with href
links = soup.find_all('a', href=True)

for link in links:
    print(link['href'])

```

OUTPUT FROM RUNNING THE CODE:

```
https://www.iana.org/domains/example
```

PROGRAM 3: EXTRACT TABLE DATA

Given HTML table, write a Python program using BeautifulSoup to extract rows as lists.

APPROACH:

- Parse HTML string with BeautifulSoup
- Find all <tr> table rows
- For each row, find all <th> and <td> cells
- Extract text from each cell and store in list
- Print each row as list of cell values

MY CODE:

```
from bs4 import BeautifulSoup

html = """
<table>
<tr><th>Name</th><th>Age</th></tr>
<tr><td>Alice</td><td>25</td></tr>
<tr><td>Bob</td><td>30</td></tr>
</table>
"""

soup = BeautifulSoup(html, 'html.parser')

# Find all rows
rows = soup.find_all('tr')

for row in rows:
    cells = row.find_all(['th', 'td'])
    data = [cell.text for cell in cells]
    print(data)

```

OUTPUT FROM RUNNING THE CODE:

```
['Name', 'Age']
['Alice', '25']
['Bob', '30']
```

PROGRAM 5: SAVE SCRAPED DATA TO CSV

Write a program that scrapes a list of fruits from HTML and saves them into a fruits.csv file.

APPROACH:

- Parse HTML to extract all <li> elements
- Store fruit names in list
- Write to CSV file with header row "Fruit"
- Write each fruit as separate row
- Confirm successful save

MY CODE:

```
from bs4 import BeautifulSoup
import csv

html = """
<ul>
<li>Apple</li>
<li>Banana</li>
<li>Cherry</li>
</ul>
"""

soup = BeautifulSoup(html, 'html.parser')

# Extract all fruits
fruits = [li.text for li in soup.find_all('li')]

# Save to CSV
with open('fruits.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['Fruit'])  # Header
    for fruit in fruits:
        writer.writerow([fruit])

print("Data saved to fruits.csv")

```

OUTPUT FROM RUNNING THE CODE:

```
Data saved to fruits.csv
```

CHAPTER 10

MULTIPLE CHOICE QUESTIONS (MCQS)

1. WHICH METHOD IS CALLED WHEN ENTERING A CONTEXT MANAGER BLOCK USING WITH?

   b) __enter__()

2. WHICH KEYWORD IS USED IN PYTHON GENERATORS?

   b) yield

3. IN THE OBSERVER PATTERN, WHAT IS THE PRIMARY RESPONSIBILITY OF THE SUBJECT?

   b) Maintain state and notify observers

4. WHICH DESIGN PATTERN ENSURES ONLY ONE INSTANCE OF A CLASS EXISTS?

   b) Singleton

5. WHICH OF THE FOLLOWING IS NOT A BENEFIT OF DEPENDENCY INJECTION?

   c) Stronger coupling between classes

TRUE/FALSE QUESTIONS

1. THE EXIT () METHOD IN A CONTEXT MANAGER IS ALWAYS EXECUTED, EVEN IF AN EXCEPTION OCCURS INSIDE THE WITH BLOCK.

   (True)

2. GENERATORS RETURN ALL VALUES AT ONCE LIKE A LIST.

   (False)

3. COROUTINES CAN BOTH PRODUCE VALUES AND RECEIVE INPUT USING SEND().

   (True)

4. THE FACTORY PATTERN IS USED TO NOTIFY MULTIPLE OBSERVERS WHEN THE STATE OF AN OBJECT CHANGES.

   (False)

5. DEPENDENCY INJECTION HELPS REDUCE COUPLING BETWEEN CLASSES.

(True)

SHORT ANSWER / CONCEPTUAL QUESTIONS

1. WHAT IS THE DIFFERENCE BETWEEN A GENERATOR AND A COROUTINE IN PYTHON?

Answer: A generator produces values lazily using yield, while a coroutine can also consume values sent into it using send(). Coroutines are often used for event-driven programming and concurrency.

2. EXPLAIN WHY THE WITH STATEMENT IS PREFERRED OVER MANUAL RESOURCE MANAGEMENT.

Answer: The with statement ensures resources (like files, sockets, or locks) are automatically cleaned up via _exit_(), even if an exception occurs, making code

safer and cleaner.

3. GIVE A REAL-WORLD EXAMPLE WHERE THE OBSERVER PATTERN MIGHT BE APPLIED.

Answer: In a stock trading app, multiple UI components (observers) need to be

updated whenever stock prices (subject state) change.

4. WHAT PROBLEM DOES THE FACTORY PATTERN SOLVE?

Answer: It abstracts object creation, allowing clients to create objects without

depending on their concrete classes.

5. HOW DOES DEPENDENCY INJECTION IMPROVE TESTABILITY OF CODE?

Answer: It allows dependencies (like services or databases) to be swapped out with mock objects during testing, making unit tests easier and more isolated.

PROBLEM 1: CUSTOM CONTEXT MANAGER

Write a custom context manager that times how long the code inside the with block takes to execute.

APPROACH:

- Create Timer class with __enter__ and __exit__ methods
- Record start time in __enter__
- Calculate elapsed time in __exit__
- Print execution duration in __exit__

MY CODE:

```
import time

class Timer:
    def __enter__(self):
        self.start = time.time()
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.end = time.time()
        print(f"Execution took {self.end - self.start:.2f} seconds")

# Test
with Timer():
    for i in range(1000000):
        pass

```

OUTPUT FROM RUNNING THE CODE:

```
Execution took 0.03 seconds
```

PROBLEM 2: GENERATOR FUNCTION

Write a generator function even_numbers(n) that yields even numbers up to n.

APPROACH:

- Use range() with step 2 to generate even numbers
- Yield each number instead of returning list
- Generator produces numbers on demand for memory efficiency

MY CODE:

```
def even_numbers(n):
    for i in range(2, n + 1, 2):
        yield i

```

```
5. # Test
6. for num in even_numbers(10):
7.     print(num)
8.
```

OUTPUT FROM RUNNING THE CODE:

```
2
4
6
8
10
```

PROBLEM 3: COROUTINE FOR FILTERING

Write a coroutine filter_positive() that receives numbers and prints only the positive ones.

APPROACH:

- Create generator function with infinite loop
- Use yield to receive numbers via send()
- Check if number > 0 and print positive values
- Prime coroutine with next() before sending values

MY CODE:

```
 1. def filter_positive():
 2.     while True:
 3.         num = yield
 4.         if num > 0:
 5.             print(f"Positive number: {num}")
 6.
 7. # Test
 8. co = filter_positive()
 9. next(co)  # Prime the coroutine
10. co.send(-3)
11. co.send(5)
12. co.send(0)
13.
```

OUTPUT FROM RUNNING THE CODE:

```
Positive number: 5
```

PROBLEM 4: FACTORY PATTERN IMPLEMENTATION

Implement a factory that returns different types of shapes (Circle, Square) with draw() method.

APPROACH:

- Create Circle and Square classes with draw() method
- Implement shape_factory() function that returns appropriate object based on input
- Handle case-insensitive shape type input

MY CODE:

```
class Circle:
    def draw(self):
        print("Drawing a Circle")

class Square:
    def draw(self):
        print("Drawing a Square")

def shape_factory(shape_type):
    if shape_type.lower() == "circle":
        return Circle()
    elif shape_type.lower() == "square":
        return Square()
    else:
        raise ValueError("Unknown shape type")

# Test
shape = shape_factory("circle")
shape.draw()

```

OUTPUT FROM RUNNING THE CODE:

```
Drawing a Circle
```

All Code referenced in this document can be found in my repository:

https://githun.com/Basem3sam/python-college-exercises