100 Intermediate Python Practice Questions

List Comprehensions & Generator Expressions (Questions 1-10)

1. Square Even Numbers

Create a list comprehension that returns squares of only even numbers from a list.

```python
def square_evens(numbers):
    # Use list comprehension
    pass
```

Test Cases:

- Input: `square_evens([1, 2, 3, 4, 5, 6])` → Output: `[4, 16, 36]`
- Input: `square_evens([1, 3, 5])` → Output: `[]`
- Input: `square_evens([2, 4, 6, 8])` → Output: `[4, 16, 36, 64]`

2. Nested List Comprehension - Matrix Transpose

Write a function using list comprehension to transpose a matrix.

```python
def transpose_matrix(matrix):
    # Use nested list comprehension
    pass
```

Test Cases:

- Input: `transpose_matrix([[1, 2, 3], [4, 5, 6]])` → Output: `[[1, 4], [2, 5], [3, 6]]`
- Input: `transpose_matrix([[1, 2], [3, 4], [5, 6]])` → Output: `[[1, 3, 5], [2, 4, 6]]`

3. Conditional List Comprehension

Create a list comprehension that replaces negative numbers with 0 and keeps positive numbers.

```python
```

```python
def replace_negatives(numbers):
    # Use list comprehension with conditional
    pass
```

Test Cases:

- Input: `replace_negatives([1, -2, 3, -4, 5])` → Output: `[1, 0, 3, 0, 5]`
- Input: `replace_negatives([-1, -2, -3])` → Output: `[0, 0, 0]`

4. Dictionary Comprehension - Character Count

Use dictionary comprehension to count characters in a string.

```python
def char_frequency(s):
    # Use dictionary comprehension
    pass
```

Test Cases:

- Input: `char_frequency("hello")` → Output: `{'h': 1, 'e': 1, 'l': 2, 'o': 1}`
- Input: `char_frequency("aaa")` → Output: `{'a': 3}`

5. Set Comprehension - Unique Lengths

Use set comprehension to get unique word lengths from a list of words.

```python
def unique_lengths(words):
    # Use set comprehension
    pass
```

Test Cases:

- Input: `unique_lengths(["cat", "dog", "bird", "ant"])` → Output: `{3, 4}`
- Input: `unique_lengths(["a", "ab", "abc", "xy"])` → Output: `{1, 2, 3}`

6. Generator Expression - Memory Efficient Sum

Create a generator expression to sum squares of numbers from 1 to n.

```python
```

```
def sum_of_squares(n):
    # Use generator expression
    pass
```

Test Cases:

- Input: `sum_of_squares(5)` → Output: `55` (1+4+9+16+25)
- Input: `sum_of_squares(3)` → Output: `14` (1+4+9)

7. Flatten with List Comprehension

Flatten a 2D list using list comprehension.

```
python

def flatten_2d(matrix):
    # Use list comprehension
    pass
```

Test Cases:

- Input: `flatten_2d([[1, 2], [3, 4], [5, 6]])` → Output: `[1, 2, 3, 4, 5, 6]`
- Input: `flatten_2d([[1], [2, 3]])` → Output: `[1, 2, 3]`

8. Filter and Transform

Create a list comprehension that filters strings by length and converts to uppercase.

```
python

def filter_and_upper(words, min_length):
    # Use list comprehension
    pass
```

Test Cases:

- Input: `filter_and_upper(["hi", "hello", "hey", "python"], 4)` → Output: `['HELLO', 'PYTHON']`
- Input: `filter_and_upper(["a", "ab", "abc"], 2)` → Output: `['AB', 'ABC']`

9. Nested Dictionary Comprehension

Create a multiplication table using dictionary comprehension.

```
python
```

```python
def multiplication_table(n):
    # Return {i: {j: i*j for j in range(1, n+1)} for i in range(1, n+1)}
    pass
```

Test Cases:

- Input: `multiplication_table(3)` → Output: `{1: {1: 1, 2: 2, 3: 3}, 2: {1: 2, 2: 4, 3: 6}, 3: {1: 3, 2: 6, 3: 9}}`

10. Conditional Generator

Create a generator that yields only prime numbers up to n.

```python
def prime_generator(n):
    # Use generator with yield
    pass
```

Test Cases:

- Input: `list(prime_generator(10))` → Output: `[2, 3, 5, 7]`
- Input: `list(prime_generator(20))` → Output: `[2, 3, 5, 7, 11, 13, 17, 19]`

Lambda Functions & Functional Programming (Questions 11-20)

11. Sort by Custom Key

Sort a list of tuples by the second element using lambda.

```python
def sort_by_second(tuples_list):
    # Use sorted with lambda
    pass
```

Test Cases:

- Input: `sort_by_second([(1, 3), (2, 1), (3, 2)])` → Output: `[(2, 1), (3, 2), (1, 3)]`
- Input: `sort_by_second([('a', 5), ('b', 2), ('c', 8)])` → Output: `[('b', 2), ('a', 5), ('c', 8)]`

12. Map with Lambda

Use map and lambda to convert list of strings to their lengths.

```python
def string_lengths(strings):
    # Use map with lambda
    pass
```

Test Cases:

- Input: `string_lengths(["hi", "hello", "hey"])` → Output: `[2, 5, 3]`
- Input: `string_lengths(["a", "ab", "abc"])` → Output: `[1, 2, 3]`

13. Filter with Lambda

Use filter and lambda to get only numbers divisible by both 3 and 5.

```python
def divisible_by_3_and_5(numbers):
    # Use filter with lambda
    pass
```

Test Cases:

- Input: `divisible_by_3_and_5([15, 30, 45, 10, 20])` → Output: `[15, 30, 45]`
- Input: `divisible_by_3_and_5([1, 2, 3, 5, 15])` → Output: `[15]`

14. Reduce to Product

Use reduce to calculate the product of all numbers in a list.

```python
from functools import reduce

def product_of_list(numbers):
    # Use reduce with lambda
    pass
```

Test Cases:

- Input: `product_of_list([1, 2, 3, 4])` → Output: `24`
- Input: `product_of_list([2, 5, 10])` → Output: `100`

15. Multiple Lambda Operations

Chain map operations using lambdas: square each number, then add 10.

```python
def square_and_add(numbers):
    # Use multiple map operations
    pass
```

Test Cases:

- Input: `square_and_add([1, 2, 3])` → Output: `[11, 14, 19]`
- Input: `square_and_add([5, 10])` → Output: `[35, 110]`

16. Sort Dictionary by Value

Sort a dictionary by values using lambda.

```python
def sort_dict_by_value(d):
    # Return sorted list of tuples
    pass
```

Test Cases:

- Input: `sort_dict_by_value({'a': 3, 'b': 1, 'c': 2})` → Output: `[('b', 1), ('c', 2), ('a', 3)]`

17. Filter None Values

Use filter and lambda to remove None values from a list.

```python
def remove_none(lst):
    # Use filter with lambda
    pass
```

Test Cases:

- Input: `remove_none([1, None, 2, None, 3])` → Output: `[1, 2, 3]`
- Input: `remove_none([None, None, 5])` → Output: `[5]`

18. Custom Sort - Multiple Criteria

Sort list of dicts by 'age', then by 'name' if ages are equal.

```python
def sort_people(people):
    # people is list of dicts with 'name' and 'age'
    # Use sorted with lambda
    pass
```

Test Cases:

- Input: `sort_people([{'name': 'Alice', 'age': 30}, {'name': 'Bob', 'age': 25}, {'name': 'Charlie', 'age': 30}])` → Output: `[{'name': 'Bob', 'age': 25}, {'name': 'Alice', 'age': 30}, {'name': 'Charlie', 'age': 30}]`

19. Map with Multiple Lists

Use map with lambda on multiple lists (zip them).

```python
def add_lists(list1, list2):
    # Use map with lambda to add corresponding elements
    pass
```

Test Cases:

- Input: `add_lists([1, 2, 3], [4, 5, 6])` → Output: `[5, 7, 9]`
- Input: `add_lists([10, 20], [5, 10])` → Output: `[15, 30]`

20. Reduce to Find Maximum

Use reduce to find the maximum value in a list.

```python
from functools import reduce

def find_max_reduce(numbers):
    # Use reduce with lambda
    pass
```

Test Cases:

- Input: `find_max_reduce([1, 5, 3, 9, 2])` → Output: `9`
- Input: `find_max_reduce([-5, -1, -10])` → Output: `-1`

Recursion (Questions 21-30)

21. Recursive Factorial

Implement factorial using recursion.

```python
def factorial_recursive(n):
    # Base case and recursive case
    pass
```

Test Cases:

- Input: `factorial_recursive(5)` → Output: `120`
- Input: `factorial_recursive(0)` → Output: `1`
- Input: `factorial_recursive(3)` → Output: `6`

22. Recursive Fibonacci

Return nth Fibonacci number using recursion.

```python
def fibonacci_recursive(n):
    # Base cases and recursive case
    pass
```

Test Cases:

- Input: `fibonacci_recursive(6)` → Output: `8` (0,1,1,2,3,5,8)
- Input: `fibonacci_recursive(1)` → Output: `1`
- Input: `fibonacci_recursive(0)` → Output: `0`

23. Recursive Sum of List

Calculate sum of list elements recursively.

```python
def sum_recursive(lst):
    # Base case: empty list returns 0
    pass
```

Test Cases:

- Input: `sum_recursive([1, 2, 3, 4])` → Output: `10`
- Input: `sum_recursive([5])` → Output: `5`
- Input: `sum_recursive([])` → Output: `0`

24. Recursive Power

Calculate base^exponent using recursion.

```python
def power_recursive(base, exp):
    # Base case: exp = 0
    pass
```

Test Cases:

- Input: `power_recursive(2, 3)` → Output: `8`
- Input: `power_recursive(5, 0)` → Output: `1`
- Input: `power_recursive(3, 4)` → Output: `81`

25. Recursive String Reversal

Reverse a string using recursion.

```python
def reverse_recursive(s):
    # Base case: empty or single char
    pass
```

Test Cases:

- Input: `reverse_recursive("hello")` → Output: `"olleh"`
- Input: `reverse_recursive("a")` → Output: `"a"`
- Input: `reverse_recursive("")` → Output: `""`

26. Recursive Palindrome Check

Check if a string is a palindrome using recursion.

```python
```

```python
def is_palindrome_recursive(s):
    # Base case and recursive case
    pass
```

Test Cases:

- Input: `is_palindrome_recursive("racecar")` → Output: `True`
- Input: `is_palindrome_recursive("hello")` → Output: `False`
- Input: `is_palindrome_recursive("a")` → Output: `True`

27. Recursive GCD

Calculate GCD using Euclidean algorithm recursively.

```python
def gcd_recursive(a, b):
    # Base case: b = 0
    pass
```

Test Cases:

- Input: `gcd_recursive(48, 18)` → Output: `6`
- Input: `gcd_recursive(100, 50)` → Output: `50`
- Input: `gcd_recursive(7, 3)` → Output: `1`

28. Recursive Binary Search

Implement binary search recursively.

```python
def binary_search_recursive(arr, target, left, right):
    # Base case: left > right
    pass
```

Test Cases:

- Input: `binary_search_recursive([1, 2, 3, 4, 5], 3, 0, 4)` → Output: `2`
- Input: `binary_search_recursive([1, 2, 3, 4, 5], 6, 0, 4)` → Output: `-1`

29. Recursive Flatten

Flatten a nested list recursively.

```python
def flatten_recursive(nested_list):
    # Handle nested lists of any depth
    pass
```

Test Cases:

- Input: `flatten_recursive([1, [2, 3], [4, [5, 6]]])` → Output: `[1, 2, 3, 4, 5, 6]`
- Input: `flatten_recursive([[1, 2], [3, 4]])` → Output: `[1, 2, 3, 4]`

30. Recursive Count Occurrences

Count occurrences of an element in a list recursively.

```python
def count_recursive(lst, element):
    # Base case: empty list
    pass
```

Test Cases:

- Input: `count_recursive([1, 2, 3, 2, 2], 2)` → Output: `3`
- Input: `count_recursive([1, 1, 1], 1)` → Output: `3`
- Input: `count_recursive([], 5)` → Output: `0`

Object-Oriented Programming (Questions 31-45)

31. Basic Class - Rectangle

Create a Rectangle class with width, height, area(), and perimeter() methods.

```python
```

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

    def area(self):
        pass

    def perimeter(self):
        pass
```

Test Cases:

- Input: `r = Rectangle(5, 3); r.area()` → Output: `15`
- Input: `r = Rectangle(5, 3); r.perimeter()` → Output: `16`

32. Class with Properties

Create a Circle class with radius property and calculated area/circumference.

```python
class Circle:
    def __init__(self, radius):
        pass

    @property
    def area(self):
        pass

    @property
    def circumference(self):
        pass
```

Test Cases:

- Input: `c = Circle(5); c.area` → Output: `78.53975` (approx)
- Input: `c = Circle(10); c.circumference` → Output: `62.8318` (approx)

33. Inheritance - Animal Classes

Create Animal base class and Dog, Cat subclasses with speak() method.

```python
```

```
class Animal:
    def __init__(self, name):
        pass

class Dog(Animal):
    def speak(self):
        pass

class Cat(Animal):
    def speak(self):
        pass
```

Test Cases:

- Input: `d = Dog("Buddy"); d.speak()` → Output: `"Buddy says Woof!"`
- Input: `c = Cat("Whiskers"); c.speak()` → Output: `"Whiskers says Meow!"`

34. Class with Magic Methods

Create a Vector2D class with **add**, **str**, and **eq** methods.

```
python

class Vector2D:
    def __init__(self, x, y):
        pass

    def __add__(self, other):
        pass

    def __str__(self):
        pass

    def __eq__(self, other):
        pass
```

Test Cases:

- Input: `v1 = Vector2D(1, 2); v2 = Vector2D(3, 4); v3 = v1 + v2; str(v3)` → Output: `"Vector2D(4, 6)"`
- Input: `v1 = Vector2D(1, 2); v2 = Vector2D(1, 2); v1 == v2` → Output: `True`

35. Bank Account Class

Create a BankAccount class with deposit, withdraw, and balance tracking.

```python
class BankAccount:
    def __init__(self, initial_balance=0):
        pass

    def deposit(self, amount):
        pass

    def withdraw(self, amount):
        # Return False if insufficient funds
        pass

    def get_balance(self):
        pass
```

Test Cases:

- Input: `acc = BankAccount(100); acc.deposit(50); acc.get_balance()` → Output: `150`
- Input: `acc = BankAccount(100); acc.withdraw(150)` → Output: `False`

36. Class with Class Variables

Create a Student class that tracks total number of students.

```python
class Student:
    total_students = 0

    def __init__(self, name):
        pass

    @classmethod
    def get_total_students(cls):
        pass
```

Test Cases:

- Input: `s1 = Student("Alice"); s2 = Student("Bob"); Student.get_total_students()` → Output: `2`

37. Private Attributes

Create a Person class with private age attribute and getter/setter.

```python
class Person:
    def __init__(self, name, age):
        pass

    def get_age(self):
        pass

    def set_age(self, age):
        # Only allow positive ages
        pass
```

Test Cases:

- Input: `p = Person("Alice", 30); p.get_age()` → Output: `30`
- Input: `p = Person("Alice", 30); p.set_age(-5); p.get_age()` → Output: `30` (unchanged)

38. Multiple Inheritance

Create classes demonstrating multiple inheritance with a FlyingFish class.

```python
class Fish:
    def swim(self):
        pass

class Bird:
    def fly(self):
        pass

class FlyingFish(Fish, Bird):
    pass
```

Test Cases:

- Input: `ff = FlyingFish(); ff.swim()` → Output: `"Swimming"`
- Input: `ff = FlyingFish(); ff.fly()` → Output: `"Flying"`

39. Abstract Base Class

Create an abstract Shape class with abstract area() method.

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

class Square(Shape):
    def __init__(self, side):
        pass

    def area(self):
        pass
```

Test Cases:

- Input: `s = Square(5); s.area()` → Output: `25`

40. Composition

Create a Car class that has an Engine object.

```python
class Engine:
    def __init__(self, horsepower):
        pass

    def start(self):
        pass

class Car:
    def __init__(self, brand, horsepower):
        pass

    def start_car(self):
        pass
```

Test Cases:

- Input: `car = Car("Toyota", 150); car.start_car()` → Output: `"Engine with 150 HP started"`

41. Static Methods

Create a MathUtils class with static methods.

```python
class MathUtils:
    @staticmethod
    def is_even(n):
        pass

    @staticmethod
    def is_prime(n):
        pass
```

Test Cases:

- Input: `MathUtils.is_even(4)` → Output: `True`
- Input: `MathUtils.is_prime(7)` → Output: `True`

42. Property Decorators

Create a Temperature class with Celsius/Fahrenheit conversion.

```python
class Temperature:
    def __init__(self, celsius):
        pass

    @property
    def celsius(self):
        pass

    @celsius.setter
    def celsius(self, value):
        pass

    @property
    def fahrenheit(self):
        pass
```

Test Cases:

- Input: `t = Temperature(0); t.fahrenheit` → Output: `32.0`
- Input: `t = Temperature(100); t.fahrenheit` → Output: `212.0`

43. Context Manager Class

Create a Timer context manager class.

```python
import time

class Timer:
    def __enter__(self):
        pass

    def __exit__(self, exc_type, exc_val, exc_tb):
        pass
```

Test Cases:

- Usage: `with Timer(): time.sleep(1)` → Should print elapsed time

44. Singleton Pattern

Implement a Singleton class that allows only one instance.

```python
class Singleton:
    _instance = None

    def __new__(cls):
        pass

    def __init__(self):
        pass
```

Test Cases:

- Input: `s1 = Singleton(); s2 = Singleton(); s1 is s2` → Output: `True`

45. Iterator Class

Create a custom iterator for even numbers up to n.

```python
class EvenNumbers:
    def __init__(self, max_num):
        pass

    def __iter__(self):
        pass

    def __next__(self):
        pass
```

Test Cases:

- Input: `list(EvenNumbers(10))` → Output: `[0, 2, 4, 6, 8, 10]`

Exception Handling (Questions 46-55)

46. Basic Try-Except

Create a function that safely divides two numbers.

```python
def safe_divide(a, b):
    # Handle ZeroDivisionError
    pass
```

Test Cases:

- Input: `safe_divide(10, 2)` → Output: `5.0`
- Input: `safe_divide(10, 0)` → Output: `"Cannot divide by zero"`

47. Multiple Exceptions

Handle both ValueError and TypeError in a type conversion function.

```python
```

```python
def safe_int_convert(value):
    # Try to convert to int, handle exceptions
    pass
```

Test Cases:

- Input: `safe_int_convert("123")` → Output: `123`
- Input: `safe_int_convert("abc")` → Output: `"Invalid conversion"`
- Input: `safe_int_convert(None)` → Output: `"Invalid conversion"`

48. Finally Block

Create a function that demonstrates finally block usage.

```python
def read_file_safe(filename):
    # Try to read file, use finally to ensure cleanup
    pass
```

Test Cases:

- Should handle FileNotFoundError and ensure cleanup

49. Custom Exception

Create a custom exception for negative age values.

```python
class NegativeAgeError(Exception):
    pass

def set_age(age):
    # Raise NegativeAgeError if age < 0
    pass
```

Test Cases:

- Input: `set_age(25)` → Output: `25`
- Input: `set_age(-5)` → Raises: `NegativeAgeError`

50. Exception Chaining

Demonstrate exception chaining with raise from.

```python
def process_data(data):
    # Chain exceptions appropriately
    pass
```

51. Assert Statement

Use assertions to validate function inputs.

```python
def calculate_average(numbers):
    # Assert list is not empty
    pass
```

Test Cases:

- Input: `calculate_average([1, 2, 3])` → Output: `2.0`
- Input: `calculate_average([])` → Raises: `AssertionError`

52. Context Manager for Exceptions

Create a context manager that suppresses specific exceptions.

```python
class SuppressException:
    def __init__(self, exception_type):
        pass

    def __enter__(self):
        pass

    def __exit__(self, exc_type, exc_val, exc_tb):
        pass
```

53. Retry Decorator

Create a decorator that retries a function on exception.

```python
def retry(max_attempts=3):
    def decorator(func):
        def wrapper(*args, **kwargs):
            pass
        return wrapper
    return decorator
```

54. Exception Logging

Create a function that logs exceptions before re-raising.

```python
import logging

def divide_with_logging(a, b):
    # Log exception and re-raise
    pass
```

55. Graceful Degradation

Create a function with fallback behavior on exception.

```python
def get_config_value(key, default=None):
    # Try to get from config file, return default on error
    pass
```

File Handling (Questions 56-65)

56. Read Entire File

Read and return contents of a text file.

```python
def read_file(filename):
    # Handle file not found
    pass
```

Test Cases:

- Should return file contents as string or error message

57. Write to File

Write a list of strings to a file, each on a new line.

```python
def write_lines(filename, lines):
    # Write lines to file
    pass
```

58. Count Lines in File

Count the number of lines in a file.

```python
def count_lines(filename):
    pass
```

59. Read CSV File

Read a CSV file and return list of dictionaries.

```python
import csv

def read_csv(filename):
    # Return list of dicts
    pass
```

60. Write CSV File

Write list of dictionaries to CSV file.

```python
import csv

def write_csv(filename, data):
    # data is list of dicts
    pass
```

61. Read JSON File

Read and parse a JSON file.

```python
import json

def read_json(filename):
    pass
```

62. Write JSON File

Write a dictionary to a JSON file.

```python
import json

def write_json(filename, data):
    pass
```

63. Append to File

Append text to an existing file.

```python
def append_to_file(filename, text):
    pass
```

64. Read File with Context Manager

Read file using with statement.

```python
def read_file_safe(filename):
    # Use context manager
    pass
```

65. Process Large File

Read and process a large file line by line (memory efficient).

```python
```

```python
def process_large_file(filename):
    # Yield processed lines one at a time
    pass
```

Advanced Data Structures (Questions 66-75)

66. Stack Implementation

Implement a Stack class with push, pop, peek, is_empty.

```python
class Stack:
    def __init__(self):
        pass

    def push(self, item):
        pass

    def pop(self):
        pass

    def peek(self):
        pass

    def is_empty(self):
        pass
```

Test Cases:

- Input: `s = Stack(); s.push(1); s.push(2); s.pop()` → Output: `2`
- Input: `s = Stack(); s.is_empty()` → Output: `True`

67. Queue Implementation

Implement a Queue class with enqueue, dequeue, is_empty.

```python
```

```
class Queue:
    def __init__(self):
        pass

    def enqueue(self, item):
        pass

    def dequeue(self):
        pass

    def is_empty(self):
        pass
```

Test Cases:

- Input: `q = Queue(); q.enqueue(1); q.enqueue(2); q.dequeue()` → Output: `1`

68. LinkedList Node

Create a basic linked list with append and display methods.

```
python

class Node:
    def __init__(self, data):
        pass

class LinkedList:
    def __init__(self):
        pass

    def append(self, data):
        pass

    def to_list(self):
        pass
```

Test Cases:

- Input: `ll = LinkedList(); ll.append(1); ll.append(2); ll.to_list()` → Output: `[1, 2]`

69. Binary Tree Node

Create a binary tree node with insert and inorder traversal.

```python
class TreeNode:
    def __init__(self, value):
        pass

class BinaryTree:
    def __init__(self):
        pass

    def insert(self, value):
        pass

    def inorder(self):
        pass
```

70. Set Operations

Implement custom set operations (union, intersection, difference).

```python
def set_union(set1, set2):
    pass

def set_intersection(set1, set2):
    pass

def set_difference(set1, set2):
    pass
```

Test Cases:

- Input: `set_union({1, 2}, {2, 3})` → Output: `{1, 2, 3}`
- Input: `set_intersection({1, 2}, {2, 3})` → Output: `{2}`

71. Priority Queue

Implement a simple priority queue using heapq.

```python
```

```python
import heapq

class PriorityQueue:
    def __init__(self):
        pass

    def push(self, item, priority):
        pass

    def pop(self):
        pass
```

72. Graph Representation

Create an adjacency list representation of a graph.

```python
class Graph:
    def __init__(self):
        pass

    def add_edge(self, u, v):
        pass

    def get_neighbors(self, node):
        pass
```

73. Deque Operations

Implement double-ended queue operations.

```python
from collections import deque

def deque_operations():
    # Demonstrate append, appendleft, pop, popleft
    pass
```

74. Counter Usage

Use Counter for frequency analysis.

```python
from collections import Counter

def most_common_elements(lst, n):
    # Return n most common elements
    pass
```

Test Cases:

- Input: `most_common_elements([1,1,1,2,2,3], 2)` → Output: `[(1, 3), (2, 2)]`

75. DefaultDict Usage

Use defaultdict for grouping data.

```python
from collections import defaultdict

def group_by_length(words):
    # Group words by length
    pass
```

Test Cases:

- Input: `group_by_length(["hi", "hello", "hey", "world"])` → Output: `{2: ['hi'], 5: ['hello', 'world'], 3: ['hey']}`

Decorators (Questions 76-85)

76. Simple Timer Decorator

Create a decorator that measures function execution time.

```python
import time

def timer(func):
    def wrapper(*args, **kwargs):
        pass
    return wrapper
```

77. Logging Decorator

Create a decorator that logs function calls.

```python
def log_calls(func):
    def wrapper(*args, **kwargs):
        pass
    return wrapper
```

78. Memoization Decorator

Create a decorator for caching function results.

```python
def memoize(func):
    cache = {}
    def wrapper(*args):
        pass
    return wrapper
```

Test Cases:

- Should cache results of expensive computations

79. Validation Decorator

Create a decorator that validates function arguments.

```python
def validate_positive(func):
    def wrapper(n):
        # Raise error if n <= 0
        pass
    return wrapper
```

80. Retry Decorator

Create a decorator that retries failed operations.

```python
```

```python
def retry(max_attempts=3):
    def decorator(func):
        def wrapper(*args, **kwargs):
            pass
        return wrapper
    return decorator
```

81. Rate Limiting Decorator

Create a decorator that limits function calls per time period.

```python
import time

def rate_limit(max_calls, time_window):
    def decorator(func):
        calls = []
        def wrapper(*args, **kwargs):
            pass
        return wrapper
    return decorator
```

82. Authentication Decorator

Create a decorator that checks user authentication.

```python
def require_auth(func):
    def wrapper(user, *args, **kwargs):
        # Check if user is authenticated
        pass
    return wrapper
```

83. Class Decorator

Create a decorator for classes that adds a method.

```python
def add_str_method(cls):
    # Add __str__ method to class
    pass
```

84. Decorator with Arguments

Create a parameterized decorator for prefix/suffix.

```python
def wrap_text(prefix="", suffix=""):
    def decorator(func):
        def wrapper(*args, **kwargs):
            pass
        return wrapper
    return decorator
```

85. Property Decorator Pattern

Use @property, @setter, @deleter decorators.

```python
class Product:
    def __init__(self, price):
        pass

    @property
    def price(self):
        pass

    @price.setter
    def price(self, value):
        pass
```

Advanced Topics (Questions 86-100)

86. Regular Expressions - Email Validation

Validate email format using regex.

```python
import re

def is_valid_email(email):
    pass
```

Test Cases:

- Input: `is_valid_email("test@example.com")` → Output: `True`
- Input: `is_valid_email("invalid.email")` → Output: `False`

87. Regular Expressions - Extract Phone Numbers

Extract all phone numbers from text.

```python
import re

def extract_phone_numbers(text):
    # Match format: (123) 456-7890 or 123-456-7890
    pass
```

88. Threading - Parallel Execution

Run multiple functions in parallel using threading.

```python
import threading

def parallel_execution(functions):
    # Execute all functions in parallel
    pass
```

89. Multiprocessing - CPU Bound Tasks

Use multiprocessing for CPU-intensive tasks.

```python
from multiprocessing import Pool

def parallel_compute(numbers):
    # Compute squares in parallel
    pass
```

90. asyncio - Async Function

Create an async function with await.

```python
```

```python
import asyncio

async def fetch_data(url):
    # Simulate async operation
    pass

async def main():
    pass
```

91. Contextlib - Custom Context Manager

Create context manager using contextlib.

```python
from contextlib import contextmanager

@contextmanager
def file_handler(filename):
    pass
```

92. Itertools - Combinations

Use itertools to generate combinations.

```python
from itertools import combinations

def get_combinations(items, r):
    pass
```

Test Cases:

- Input: `list(get_combinations([1,2,3], 2))` → Output: `[(1,2), (1,3), (2,3)]`

93. Itertools - Permutations

Generate all permutations of a list.

```python
```

```
from itertools import permutations

def get_permutations(items):
    pass
```

Test Cases:

- Input: `list(get_permutations([1,2,3]))` → Should return all 6 permutations

94. Functools - Partial Functions

Use partial to create specialized functions.

```
python

from functools import partial

def create_multiplier(factor):
    # Return a function that multiplies by factor
    pass
```

Test Cases:

- Input: `double = create_multiplier(2); double(5)` → Output: `10`

95. Enum - Define Constants

Use Enum for defining constants.

```
python

from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

def get_color_name(color):
    pass
```

96. Dataclasses - Simple Data Container

Use dataclass for data containers.

```python
from dataclasses import dataclass

@dataclass
class Point:
    x: float
    y: float

    def distance_from_origin(self):
        pass
```

Test Cases:

- Input: `p = Point(3, 4); p.distance_from_origin()` → Output: `5.0`

97. Type Hints - Annotated Function

Add type hints to a function.

```python
from typing import List, Dict, Optional

def process_data(numbers: List[int]) -> Dict[str, float]:
    # Return dict with 'mean' and 'median'
    pass
```

98. Collections - Named Tuple

Use namedtuple for readable code.

```python
from collections import namedtuple

Point = namedtuple('Point', ['x', 'y'])

def create_point(x, y):
    pass
```

99. Pickle - Object Serialization

Serialize and deserialize objects using pickle.

```python
import pickle

def save_object(obj, filename):
    pass

def load_object(filename):
    pass
```

100. Dynamic Import

Dynamically import and use a module.

```python
def dynamic_import(module_name, function_name):
    # Import module and call function
    pass
```

Practice Tips:

For List Comprehensions:

- Start simple, then nest
- Always consider readability vs. conciseness
- Use generator expressions for large datasets

For OOP:

- Think about real-world relationships
- Use composition over inheritance when possible
- Make use of magic methods for intuitive interfaces

For Recursion:

- Always define base case first
- Ensure progress toward base case
- Consider iterative alternatives for efficiency

For Decorators:

- Use functools.wraps to preserve metadata
- Test with and without arguments
- Consider performance implications

General Tips:

1. Read Python's official documentation
2. Study standard library modules
3. Practice design patterns
4. Write tests for your code
5. Focus on code readability
6. Learn from open-source projects

Good luck with your intermediate Python journey!