Python Programming Mock Exam - Solution Guide

Comprehensive Answer Key with Explanations

Author: Manus AI
Date: December 2024

Part I: Fundamentals and Control Structures (30 points)

Task 1: Basic Python Operations (8 points)

Subtask 1.1 (3 points) - Grade Statistics Function

```
def calculate_grade_statistics(scores):
    """Calculate statistics for a list of exam scores."""
    if not scores:  # Handle empty list
        return {
            'average': 0,
            'highest': 0,
            'lowest': 0,
            'passing_count': 0
        }

    total = sum(scores)
    average = total / len(scores)
    highest = max(scores)
    lowest = min(scores)
    passing_count = sum(1 for score in scores if score >= 60)

    return {
        'average': round(average, 2),
        'highest': highest,
        'lowest': lowest,
        'passing_count': passing_count
    }

# Test with example
scores = [85, 92, 78, 65, 88, 45, 90]
result = calculate_grade_statistics(scores)
print(result)
```

```
# {'average': 77.57, 'highest': 92, 'lowest': 45,
'passing_count': 5}
```

Key Points: - Handle edge case of empty list - Use built-in functions `sum()`, `max()`, `min()` for efficiency - Use generator expression for counting passing grades - Round average to 2 decimal places

Subtask 1.2 (3 points) - Email Validation

```
def validate_email(email):
    """Validate email address according to specified rules."""
    # Check for exactly one '@' symbol
    if email.count('@') != 1:
        return False

    # Split by '@' to get local and domain parts
    parts = email.split('@')
    local_part = parts[0]
    domain_part = parts[1]

    # Check if both parts have at least one character
    if len(local_part) == 0 or len(domain_part) == 0:
        return False

    # Check for valid domain extensions
    valid_extensions = ['.com', '.org', '.edu', '.net']
    has_valid_extension = any(domain_part.endswith(ext) for ext
in valid_extensions)

    return has_valid_extension

# Test cases
print(validate_email("user@example.com"))     # True
print(validate_email("test@university.edu"))  # True
print(validate_email("invalid@domain.xyz"))   # False
print(validate_email("no-at-symbol.com"))     # False
print(validate_email("@missing-local.com"))   # False
```

Key Points: - Use `count()` method to check for exactly one '@' - Split email into local and domain parts - Use `any()` with generator expression for extension checking - Handle edge cases (missing parts, invalid extensions)

Subtask 1.3 (2 points) - Temperature Conversion

```
def convert_temperature(temp, scale):
    """Convert temperature between Celsius and Fahrenheit."""
    if scale.upper() == 'C':
```

```
        # Convert Celsius to Fahrenheit
        fahrenheit = (temp * 9/5) + 32
        return round(fahrenheit, 2)
    elif scale.upper() == 'F':
        # Convert Fahrenheit to Celsius
        celsius = (temp - 32) * 5/9
        return round(celsius, 2)
    else:
        raise ValueError("Scale must be 'C' or 'F'")

# Test cases
print(convert_temperature(25, 'C'))   # 77.0 (Celsius to
Fahrenheit)
print(convert_temperature(77, 'F'))   # 25.0 (Fahrenheit to
Celsius)
print(convert_temperature(0, 'C'))    # 32.0 (Freezing point)
```

Key Points: - Use `upper()` to handle both 'c'/'C' and 'f'/'F' - Apply correct conversion formulas - Round result to 2 decimal places - Raise appropriate exception for invalid scale

Task 2: Control Structures and Loops (12 points)

Subtask 2.1 (4 points) - Number Pyramid Pattern

```
def print_pattern(n):
    """Print a number pyramid pattern."""
    for i in range(1, n + 1):
        # Calculate spaces for centering
        spaces = ' ' * (n - i)

        # Build ascending numbers
        ascending = ''.join(str(j) for j in range(1, i + 1))

        # Build descending numbers (excluding the peak)
        descending = ''.join(str(j) for j in range(i - 1, 0,
-1))

        # Combine and print
        line = spaces + ascending + descending
        print(line)

# Test with n=4
print_pattern(4)
# Output:
#    1
#   121
```

```
#  12321
# 1234321
```

Key Points: - Use nested loops or string operations - Calculate proper spacing for centering - Build ascending and descending number sequences - Join sequences without separators

Subtask 2.2 (4 points) - Prime Number Finder

```
def find_prime_numbers(start, end):
    """Find all prime numbers between start and end
(inclusive)."""
    def is_prime(n):
        """Helper function to check if a number is prime."""
        if n < 2:
            return False
        if n == 2:
            return True
        if n % 2 == 0:
            return False

        # Check odd divisors up to sqrt(n)
        for i in range(3, int(n ** 0.5) + 1, 2):
            if n % i == 0:
                return False
        return True

    primes = []
    for num in range(start, end + 1):
        if is_prime(num):
            primes.append(num)

    return primes

# Test case
result = find_prime_numbers(10, 30)
print(result)  # [11, 13, 17, 19, 23, 29]
```

Key Points: - Implement efficient prime checking algorithm - Check divisors only up to square root of number - Handle edge cases (numbers less than 2) - Use helper function for code organization

Subtask 2.3 (4 points) - Number Guessing Game

```
import random

def guess_number_game():
```

```
    """Number guessing game with hints and attempt counting."""
    target = random.randint(1, 100)
    attempts = 0

    print("Welcome to the Number Guessing Game!")
    print("I'm thinking of a number between 1 and 100.")

    while True:
        try:
            guess = int(input("Enter your guess: "))
            attempts += 1

            if guess == target:
                print(f"Congratulations! You guessed it in
{attempts} attempts!")
                break
            elif guess < target:
                print("Too low! Try again.")
            else:
                print("Too high! Try again.")

        except ValueError:
            print("Please enter a valid number.")
            # Don't increment attempts for invalid input

# To run the game:
# guess_number_game()
```

Key Points: - Use `random.randint()` for number generation - Implement input validation with try-except - Provide appropriate hints based on comparison - Count and display number of attempts

Task 3: String Processing (10 points)

Subtask 3.1 (4 points) - Text Analysis

```
def analyze_text(text):
    """Analyze text and return comprehensive statistics."""
    # Word count
    words = text.split()
    word_count = len(words)

    # Character count (excluding spaces)
    char_count = len(text.replace(' ', ''))

    # Vowel count
    vowels = 'aeiouAEIOU'
    vowel_count = sum(1 for char in text if char in vowels)
```

```
    # Most common character (excluding spaces)
    char_frequency = {}
    for char in text:
        if char != ' ':
            char_frequency[char] = char_frequency.get(char, 0) +
1

    if char_frequency:
        most_common_char = max(char_frequency,
key=char_frequency.get)
    else:
        most_common_char = None

    return {
        'word_count': word_count,
        'char_count': char_count,
        'vowel_count': vowel_count,
        'most_common_char': most_common_char
    }

# Test case
result = analyze_text("Hello World Programming")
print(result)
# {'word_count': 3, 'char_count': 18, 'vowel_count': 5,
'most_common_char': 'r'}
```

Key Points: - Use `split()` for word counting - Filter out spaces for character counting - Use dictionary to track character frequencies - Handle edge case of empty text

Subtask 3.2 (3 points) - Phone Number Formatting

```
def format_phone_number(phone):
    """Format 10-digit phone number as (XXX) XXX-XXXX."""
    # Remove all non-digit characters
    digits = ''.join(char for char in phone if char.isdigit())

    # Check if we have exactly 10 digits
    if len(digits) != 10:
        return "Invalid phone number"

    # Format as (XXX) XXX-XXXX
    formatted = f"({digits[:3]}) {digits[3:6]}-{digits[6:]}"
    return formatted

# Test cases
print(format_phone_number("1234567890"))      # "(123) 456-7890"
print(format_phone_number("123-456-7890"))    # "(123) 456-7890"
print(format_phone_number("(123) 456-7890"))  # "(123) 456-7890"
print(format_phone_number("123 456 7890"))    # "(123) 456-7890"
```

```
print(format_phone_number("12345"))           # "Invalid phone
number"
```

Key Points: - Extract only digits using `isdigit()` - Validate exactly 10 digits - Use string slicing for formatting - Handle various input formats

Subtask 3.3 (3 points) - Palindrome Sentence Check

```
def is_palindrome_sentence(sentence):
    """Check if sentence is palindrome, ignoring spaces,
punctuation, and case."""
    # Keep only alphanumeric characters and convert to lowercase
    cleaned = ''.join(char.lower() for char in sentence if
char.isalnum())

    # Check if cleaned string equals its reverse
    return cleaned == cleaned[::-1]

# Test cases
print(is_palindrome_sentence("A man a plan a canal Panama"))  #
True
print(is_palindrome_sentence("race a car"))                   #
False
print(is_palindrome_sentence("Was it a rat I saw?"))          #
True
print(is_palindrome_sentence("Madam, I'm Adam"))              #
True
```

Key Points: - Use `isalnum()` to filter alphanumeric characters - Convert to lowercase for case-insensitive comparison - Use string slicing `[::-1]` for reversal - Handle punctuation and spaces properly

---

Part II: Functions, OOP, and Advanced Concepts (40 points)

Task 4: Advanced Functions and Recursion (15 points)

Subtask 4.1 (5 points) - Fibonacci Implementations

```
def fibonacci_sequence_recursive(n):
    """Generate first n Fibonacci numbers using recursion."""
    def fib(num):
        if num <= 0:
            return 0
```

```
        elif num == 1:
            return 1
        else:
            return fib(num - 1) + fib(num - 2)

    return [fib(i) for i in range(n)]

def fibonacci_sequence_iterative(n):
    """Generate first n Fibonacci numbers using iteration."""
    if n <= 0:
        return []
    elif n == 1:
        return [0]
    elif n == 2:
        return [0, 1]

    sequence = [0, 1]
    for i in range(2, n):
        next_fib = sequence[i-1] + sequence[i-2]
        sequence.append(next_fib)

    return sequence

# Test both implementations
print(fibonacci_sequence_recursive(8))
# [0, 1, 1, 2, 3, 5, 8, 13]
print(fibonacci_sequence_iterative(8))
# [0, 1, 1, 2, 3, 5, 8, 13]
```

Key Points: - Recursive version: Define helper function with base cases - Iterative version: More efficient, builds sequence step by step - Handle edge cases (n <= 0, n == 1, n == 2) - Both should produce identical results

Subtask 4.2 (5 points) - Data Processing with args and *kwargs

```
def process_data(*args, **kwargs):
    """Process numeric data with specified operation and
formatting."""
    # Get operation and format from kwargs
    operation = kwargs.get('operation', 'sum')
    format_type = kwargs.get('format', 'int')

    # Validate that we have numeric arguments
    if not args:
        return "No data provided"

    # Perform the specified operation
    if operation == 'sum':
        result = sum(args)
    elif operation == 'product':
```

```
            result = 1
            for num in args:
                result *= num
        elif operation == 'average':
            result = sum(args) / len(args)
        elif operation == 'max':
            result = max(args)
        elif operation == 'min':
            result = min(args)
        else:
            return f"Unknown operation: {operation}"

        # Format the result
        if format_type == 'int':
            return int(result)
        elif format_type == 'float':
            return float(result)
        elif format_type == 'string':
            operation_name = operation.capitalize()
            return f"{operation_name}: {result}"
        else:
            return result

    # Test cases
    print(process_data(1, 2, 3, 4, operation='sum',
    format='string'))     # "Sum: 10"
    print(process_data(2, 4, 6, operation='average',
    format='float'))     # 4.0
    print(process_data(1, 2, 3, 4, 5, operation='product',
    format='int')) # 120
```

Key Points: - Use `*args` to accept variable number of arguments - Use `**kwargs` to accept operation and format parameters - Implement all required operations with proper logic - Handle different format types appropriately

Subtask 4.3 (5 points) - Timing Decorator

```
    import time
    from functools import wraps

    def timing_decorator(func):
        """Decorator to measure and print function execution
    time."""
        @wraps(func)  # Preserves original function metadata
        def wrapper(*args, **kwargs):
            start_time = time.time()
            result = func(*args, **kwargs)
            end_time = time.time()
            execution_time = end_time - start_time
```

```
        print(f"{func.__name__} took {execution_time:.4f}
seconds")
        return result
    return wrapper

@timing_decorator
def slow_function():
    """A function that takes some time."""
    time.sleep(1)
    return "Done"

@timing_decorator
def fast_function(x, y):
    """A fast function for testing."""
    return x + y

# Test the decorator
result1 = slow_function()  # Prints: "slow_function took 1.0023
seconds"
result2 = fast_function(5, 3)  # Prints: "fast_function took
0.0001 seconds"
```

Key Points: - Use `@wraps(func)` to preserve function metadata - Measure time before and after function execution - Handle both `*args` and `**kwargs` in wrapper - Return the original function's result

Task 5: Object-Oriented Programming (25 points)

Subtask 5.1 (10 points) - Library Class

```
class Library:
    """A class representing a library with book management
capabilities."""

    # Class variables
    total_books = 0
    library_count = 0

    def __init__(self, name):
        """Initialize a library with a name."""
        self.name = name
        self.books = []

        # Update class variables
        Library.library_count += 1
        self.library_id = Library.library_count

    def add_book(self, title, author, isbn):
        """Add a book to the library."""
```

```
        book = {
            'title': title,
            'author': author,
            'isbn': isbn
        }
        self.books.append(book)
        Library.total_books += 1
        print(f"Added '{title}' by {author} to {self.name}")

    def remove_book(self, isbn):
        """Remove a book by ISBN."""
        for i, book in enumerate(self.books):
            if book['isbn'] == isbn:
                removed_book = self.books.pop(i)
                Library.total_books -= 1
                print(f"Removed '{removed_book['title']}' from
{self.name}")
                return True
        print(f"Book with ISBN {isbn} not found in {self.name}")
        return False

    def find_books_by_author(self, author):
        """Find all books by a specific author."""
        author_books = [book for book in self.books if
book['author'].lower() == author.lower()]
        return author_books

    def get_library_info(self):
        """Return formatted library information."""
        return f"Library: {self.name} (ID: {self.library_id}),
Books: {len(self.books)}"

    @classmethod
    def get_total_books(cls):
        """Get total books across all libraries."""
        return cls.total_books

    @classmethod
    def get_library_count(cls):
        """Get total number of libraries."""
        return cls.library_count

# Test the Library class
lib1 = Library("Central Library")
lib2 = Library("Branch Library")

lib1.add_book("1984", "George Orwell", "978-0-452-28423-4")
lib1.add_book("Animal Farm", "George Orwell",
"978-0-452-28424-1")
lib2.add_book("Brave New World", "Aldous Huxley",
"978-0-06-085052-4")
```

```
print(lib1.get_library_info())
print(f"Total books: {Library.get_total_books()}")
print(f"Orwell books: {lib1.find_books_by_author('George
Orwell')}")
```

Key Points: - Implement class variables that track across all instances - Use proper instance variable initialization - Implement all required methods with appropriate functionality - Use class methods for accessing class-level data

Subtask 5.2 (8 points) - BankAccount Class

```
class BankAccount:
    """A class representing a bank account with secure balance
management."""

    def __init__(self, account_number, account_holder,
initial_balance=0):
        """Initialize a bank account."""
        self.account_number = account_number
        self.account_holder = account_holder
        self._balance = initial_balance  # Private attribute

    @property
    def balance(self):
        """Get account balance (read-only property)."""
        return self._balance

    def deposit(self, amount):
        """Deposit money to the account."""
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")
        self._balance += amount
        print(f"Deposited ${amount:.2f}. New balance: $
{self._balance:.2f}")

    def withdraw(self, amount):
        """Withdraw money from the account."""
        if amount <= 0:
            raise ValueError("Withdrawal amount must be
positive")
        if amount > self._balance:
            raise ValueError("Insufficient funds")
        self._balance -= amount
        print(f"Withdrew ${amount:.2f}. New balance: $
{self._balance:.2f}")

    def transfer(self, amount, target_account):
        """Transfer money to another account."""
        if not isinstance(target_account, BankAccount):
            raise TypeError("Target must be a BankAccount
```

```
instance")

        # Withdraw from this account
        self.withdraw(amount)
        # Deposit to target account
        target_account.deposit(amount)
        print(f"Transferred ${amount:.2f} to account
{target_account.account_number}")

    def __str__(self):
        """Return formatted account information."""
        return f"Account {self.account_number}:
{self.account_holder}, Balance: ${self._balance:.2f}"

    def __eq__(self, other):
        """Compare accounts by account number."""
        if isinstance(other, BankAccount):
            return self.account_number == other.account_number
        return False

# Test the BankAccount class
account1 = BankAccount("ACC001", "Alice Johnson", 1000)
account2 = BankAccount("ACC002", "Bob Smith", 500)

print(account1)  # Uses __str__
account1.deposit(200)
account1.withdraw(150)
account1.transfer(100, account2)
print(f"Alice's balance: ${account1.balance}")
print(f"Bob's balance: ${account2.balance}")
```

Key Points: - Use private attribute `_balance` with property getter - Implement proper validation in deposit/withdraw methods - Handle transfer between accounts with error checking - Implement special methods `__str__` and `__eq__`

Subtask 5.3 (7 points) - Vehicle Inheritance Hierarchy

```
class Vehicle:
    """Base class for all vehicles."""

    def __init__(self, make, model, year):
        """Initialize a vehicle."""
        self.make = make
        self.model = model
        self.year = year
        self.fuel_level = 0
        self.engine_running = False

    def start_engine(self):
```

```
        """Start the vehicle's engine."""
        if self.fuel_level <= 0:
            print("Cannot start engine: No fuel!")
            return False
        self.engine_running = True
        print(f"{self.year} {self.make} {self.model} engine
started.")
        return True

    def stop_engine(self):
        """Stop the vehicle's engine."""
        self.engine_running = False
        print(f"{self.year} {self.make} {self.model} engine
stopped.")

    def refuel(self, amount):
        """Add fuel to the vehicle."""
        if amount <= 0:
            raise ValueError("Fuel amount must be positive")
        self.fuel_level += amount
        print(f"Added {amount} units of fuel. Current level:
{self.fuel_level}")

class Car(Vehicle):
    """Car class inheriting from Vehicle."""

    def __init__(self, make, model, year, num_doors):
        """Initialize a car."""
        super().__init__(make, model, year)
        self.num_doors = num_doors

    def start_engine(self):
        """Start car engine with door check."""
        print(f"Checking {self.num_doors} doors...")
        if self.num_doors < 2:
            print("Warning: Unusual number of doors!")
        return super().start_engine()

    def __str__(self):
        return f"{self.year} {self.make} {self.model}
({self.num_doors}-door car)"

class Motorcycle(Vehicle):
    """Motorcycle class inheriting from Vehicle."""

    def __init__(self, make, model, year, has_sidecar=False):
        """Initialize a motorcycle."""
        super().__init__(make, model, year)
        self.has_sidecar = has_sidecar

    def start_engine(self):
        """Start motorcycle engine with safety check."""
```

```
        print("Performing safety check...")
        if self.has_sidecar:
            print("Sidecar detected - extra stability check
complete.")
        else:
            print("Remember to wear a helmet!")
        return super().start_engine()

    def __str__(self):
        sidecar_info = "with sidecar" if self.has_sidecar else
"without sidecar"
        return f"{self.year} {self.make} {self.model}
(motorcycle {sidecar_info})"

# Test the vehicle hierarchy
car = Car("Toyota", "Camry", 2020, 4)
motorcycle = Motorcycle("Harley-Davidson", "Street 750", 2019,
False)

print(car)
print(motorcycle)

car.refuel(50)
car.start_engine()

motorcycle.refuel(20)
motorcycle.start_engine()
```

Key Points: - Proper use of `super().__init__()` in derived classes - Override methods while calling parent implementation - Add class-specific attributes and behavior - Implement `__str__` methods for readable output

---

Part III: Data Structures and Advanced Topics (30 points)

Task 6: List Comprehensions and Lambda Functions (12 points)

Subtask 6.1 (4 points) - Converting For-Loops to Comprehensions

```
# Original for-loop code converted to list comprehensions

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Convert first for-loop
result1 = [num ** 2 for num in numbers if num % 2 == 0]
print(f"Even squares: {result1}")  # [4, 16, 36, 64, 100]
```

```
# Convert second for-loop
words = ["hello", "world", "python", "programming"]
result2 = [word.upper() for word in words if len(word) > 5]
print(f"Long words uppercase: {result2}")  # ['PYTHON',
'PROGRAMMING']
```

Key Points: - Basic comprehension syntax: `[expression for item in iterable if condition]` - Combine filtering (`if` condition) with transformation (expression) - More concise and often more readable than equivalent for-loops

Subtask 6.2 (4 points) - Map, Filter, and Lambda Functions

```
# Given list of dictionaries
students = [
    {"name": "Alice", "grade": 85, "age": 20},
    {"name": "Bob", "grade": 92, "age": 19},
    {"name": "Charlie", "grade": 78, "age": 21},
    {"name": "Diana", "grade": 96, "age": 20}
]

# 1. Extract all names using map and lambda
names = list(map(lambda student: student["name"], students))
print(f"Names: {names}")  # ['Alice', 'Bob', 'Charlie', 'Diana']

# 2. Filter students with grade >= 90 using filter and lambda
high_achievers = list(filter(lambda student: student["grade"] >=
90, students))
print(f"High achievers: {high_achievers}")
# [{'name': 'Bob', 'grade': 92, 'age': 19}, {'name': 'Diana',
'grade': 96, 'age': 20}]

# 3. Create list of formatted strings using map and lambda
formatted_grades = list(map(lambda student:
f"{student['name']}: {student['grade']}", students))
print(f"Formatted: {formatted_grades}")
# ['Alice: 85', 'Bob: 92', 'Charlie: 78', 'Diana: 96']
```

Key Points: - `map()` applies function to each element - `filter()` selects elements based on condition - Lambda functions provide anonymous function capability - Convert results to lists since map/filter return iterators

Subtask 6.3 (4 points) - Nested List Comprehension

```
# Generate 5x5 multiplication table using nested list
comprehension
```

```
multiplication_table = [[i * j for j in range(1, 6)] for i in
range(1, 6)]

print("5x5 Multiplication Table:")
for row in multiplication_table:
    print(row)

# Output:
# [1, 2, 3, 4, 5]
# [2, 4, 6, 8, 10]
# [3, 6, 9, 12, 15]
# [4, 8, 12, 16, 20]
# [5, 10, 15, 20, 25]

# Alternative with better formatting
print("\nFormatted table:")
for i, row in enumerate(multiplication_table, 1):
    print(f"Row {i}: {row}")
```

Key Points: - Nested comprehension syntax: `[[inner_expr for inner_var in inner_range] for outer_var in outer_range]` - Outer loop creates rows, inner loop creates columns - Each element is the product of row and column indices

Task 7: File Operations and Exception Handling (10 points)

Subtask 7.1 (5 points) - Safe File Operations

```
def safe_file_operations(filename, data):
    """Safely write data to a file with comprehensive error
handling."""
    try:
        # Use context manager for proper file handling
        with open(filename, 'w', encoding='utf-8') as file:
            if isinstance(data, str):
                file.write(data)
            elif isinstance(data, list):
                for line in data:
                    file.write(str(line) + '\n')
            else:
                file.write(str(data))

        return (True, f"Successfully wrote data to {filename}")

    except FileNotFoundError:
        return (False, f"Directory not found for file:
{filename}")

    except PermissionError:
        return (False, f"Permission denied: Cannot write to
```

```
{filename}")

    except OSError as e:
        return (False, f"OS error occurred: {e}")

    except Exception as e:
        return (False, f"Unexpected error: {e}")

# Test the function
success, message = safe_file_operations("test.txt", "Hello,
World!")
print(f"Success: {success}, Message: {message}")

success, message = safe_file_operations("/root/forbidden.txt",
"This will fail")
print(f"Success: {success}, Message: {message}")

# Test with list data
data_list = ["Line 1", "Line 2", "Line 3"]
success, message = safe_file_operations("list_data.txt",
data_list)
print(f"Success: {success}, Message: {message}")
```

Key Points: - Use context manager (`with` statement) for automatic file closing - Handle multiple exception types specifically - Return tuple with success status and descriptive message - Handle different data types (string, list, other)

Subtask 7.2 (5 points) - CSV Data Processing

```
def process_csv_data(filename):
    """Process CSV file with student grades and calculate
averages."""
    try:
        student_averages = {}

        with open(filename, 'r', encoding='utf-8') as file:
            # Skip header if present
            lines = file.readlines()

            for line_num, line in enumerate(lines, 1):
                line = line.strip()
                if not line:  # Skip empty lines
                    continue

                try:
                    # Parse CSV line
                    parts = line.split(',')
                    if len(parts) != 4:
                        print(f"Warning: Line {line_num} has
```

```
incorrect format, skipping")
                        continue

                    name = parts[0].strip()
                    grade1 = float(parts[1].strip())
                    grade2 = float(parts[2].strip())
                    grade3 = float(parts[3].strip())

                    # Calculate average
                    average = (grade1 + grade2 + grade3) / 3
                    student_averages[name] = round(average, 2)

                except ValueError as e:
                    print(f"Warning: Invalid grade data on line
{line_num}, skipping: {e}")
                    continue

        return student_averages

    except FileNotFoundError:
        print(f"Error: File '{filename}' not found")
        return {}

    except PermissionError:
        print(f"Error: Permission denied reading '{filename}'")
        return {}

    except Exception as e:
        print(f"Error: Unexpected error reading '{filename}':
{e}")
        return {}

# Test with sample data
# Create a test CSV file first
test_data = """Alice,85,90,88
Bob,92,89,94
Charlie,78,82,75
Diana,96,98,95"""

with open("students.csv", "w") as f:
    f.write(test_data)

# Process the CSV
averages = process_csv_data("students.csv")
print("Student averages:")
for name, avg in averages.items():
    print(f"{name}: {avg}")
```

Key Points: - Handle file reading errors gracefully - Parse CSV data manually (or could use `csv` module) - Validate data format and handle conversion errors - Calculate

averages and round appropriately - Return empty dictionary on errors rather than crashing

Task 8: Modules and Type Annotations (8 points)

Subtask 8.1 (4 points) - Adding Type Annotations

```
from typing import List, Union, Optional

def calculate_statistics(data: List[Union[int, float]],
operation: str) -> Optional[float]:
    """
    Calculate statistics on a list of numbers.

    Args:
        data: List of numbers (int or float)
        operation: String indicating operation ('mean',
'median', 'mode')

    Returns:
        Calculated statistic value or None if invalid operation
    """
    if operation == 'mean':
        return sum(data) / len(data)
    elif operation == 'median':
        sorted_data = sorted(data)
        n = len(sorted_data)
        if n % 2 == 0:
            return (sorted_data[n//2-1] + sorted_data[n//2]) / 2
        else:
            return sorted_data[n//2]
    elif operation == 'mode':
        from collections import Counter
        counts = Counter(data)
        return counts.most_common(1)[0][0]
    else:
        return None

# Test the function
numbers = [1, 2, 3, 4, 5, 5, 6]
print(f"Mean: {calculate_statistics(numbers, 'mean')}")      #
3.714...
print(f"Median: {calculate_statistics(numbers, 'median')}")  #
4.0
print(f"Mode: {calculate_statistics(numbers, 'mode')}")      # 5
```

Key Points: - Use `List[Union[int, float]]` for list of numbers - Use `Optional[float]` for return type that can be None - Import necessary types from `typing` module - Maintain original function logic while adding type safety

Subtask 8.2 (4 points) - Module Import Statements

```
# Import statements for using math_utils module

# 1. Import entire module
import math_utils

# Calculate factorial of 5
fact_5 = math_utils.factorial(5)
print(f"5! = {fact_5}")

# Check if 17 is prime
is_17_prime = math_utils.is_prime(17)
print(f"17 is prime: {is_17_prime}")

# 2. Import only the factorial function with an alias
from math_utils import factorial as fact

# Use the aliased function
result = fact(6)
print(f"6! = {result}")

# 3. Alternative import methods
from math_utils import factorial, is_prime  # Import specific
functions
from math_utils import *                    # Import all (not
recommended)
import math_utils as mu
# Import with module alias

# Examples of usage with different import styles
print(f"Using direct import: {factorial(4)}")
print(f"Using module alias: {mu.is_prime(13)}")
```

Key Points: - Demonstrate different import syntaxes - Show module.function notation - Use aliases for both modules and functions - Explain when each import style is appropriate

---

Bonus Question (5 points) - Context Manager

```
import time
```

```python
class TimedOperation:
    """Context manager to measure execution time of code
blocks."""

    def __init__(self, operation_name="Operation"):
        """Initialize with optional operation name."""
        self.operation_name = operation_name
        self.start_time = None
        self.end_time = None

    def __enter__(self):
        """Enter the context - start timing."""
        self.start_time = time.time()
        print(f"Starting {self.operation_name}...")
        return self  # Return self to allow access to the
context manager

    def __exit__(self, exc_type, exc_value, traceback):
        """Exit the context - stop timing and report."""
        self.end_time = time.time()
        elapsed_time = self.end_time - self.start_time

        if exc_type is None:
            # No exception occurred
            print(f"{self.operation_name} completed in
{elapsed_time:.2f} seconds")
        else:
            # An exception occurred
            print(f"{self.operation_name} failed after
{elapsed_time:.2f} seconds")
            print(f"Exception: {exc_type.__name__}:
{exc_value}")

        # Return False to propagate any exception
        return False

    def get_elapsed_time(self):
        """Get elapsed time if operation is complete."""
        if self.start_time and self.end_time:
            return self.end_time - self.start_time
        return None

# Usage examples
print("Example 1: Successful operation")
with TimedOperation("Database query"):
    time.sleep(2)  # Simulated operation
    result = "Query completed"

print("\nExample 2: Operation with custom name")
with TimedOperation("File processing") as timer:
    time.sleep(1)
    print(f"Processing... (elapsed so far: {time.time() -
```

```
timer.start_time:.1f}s)")

print("\nExample 3: Operation that raises exception")
try:
    with TimedOperation("Risky operation"):
        time.sleep(0.5)
        raise ValueError("Something went wrong!")
except ValueError:
    print("Exception was handled outside context manager")
```

Key Points: - Implement `__enter__` and `__exit__` methods - Use `time.time()` to measure elapsed time - Handle both successful and failed operations - Return `False` from `__exit__` to propagate exceptions - Provide meaningful output with operation names

---

Grading Summary

Total Points: 105 (100 + 5 bonus)

Key Evaluation Criteria:

1. **Correct Syntax (25%)**: Proper Python syntax and structure
2. **Logic Implementation (35%)**: Correct algorithmic approach and logic
3. **Error Handling (15%)**: Appropriate exception handling where required
4. **Code Quality (15%)**: Clean code, good naming, proper comments
5. **Edge Cases (10%)**: Handling of boundary conditions and special cases

Common Points of Emphasis:

- Proper use of Python idioms (list comprehensions, context managers)
- Object-oriented programming principles (encapsulation, inheritance)
- Function design (parameters, return values, documentation)
- Error handling and input validation
- Type annotations and code documentation

This solution guide demonstrates best practices and complete implementations for all exam questions, providing a comprehensive reference for understanding expected solutions and coding standards.