System Requirements Specification Index

For

Event Management System

Version 1.0

IIHT Pvt. Ltd.
fullstack@iiht.com

Table of Contents

1 Project Abstract

EventPro Solutions requires a specialized system for managing event planning, guest tracking, and resource allocation. This assignment focuses on implementing functions with appropriate scope management, various argument types, and effective return value handling. Each function will handle specific event management tasks, demonstrating proper variable scope, parameter passing, and structured data returns.

2 Business Requirements:

Screen Name	Console input screen
Problem Statement	1. System needs properly defined functions with appropriate parameters 2. Each function must be properly documented with docstrings 3. Functions must be called with correct arguments 4. System must demonstrate basic nutrition calculations like Calorie calculation, Macronutrient calculation, Body mass index calculation, Water intake recommendation

3 Constraints

3.1 Function Scope Requirements

1. Variable Scope Management:
    - Must use local variables within functions for calculation steps
    - Must demonstrate proper handling of function scope vs. global scope

- Must use nonlocal variables in at least one nested function
- Example: `def analyze_event(attendees): local_total = sum(guest["ticket_price"] for guest in attendees)`

2. Function Nesting:
    - Must implement at least one function that contains nested functions
    - Must demonstrate closure technique for event calculations

      Must maintain proper scope hierarchy

      Example: `def create_pricing_calculator(base_fee): def calculate(guests): return base_fee + (guests * per_person_cost)`
3. Scope Best Practices:
    - Function parameters must be used instead of relying on global variables
    - Results must be returned rather than modifying global state
    - Must demonstrate proper namespace management
    - Must avoid unnecessary global variables

3.2 ARGUMENT REQUIREMENTS

1. Positional Arguements:
    - Must implement functions with required positional arguments
    - Must demonstrate correct positional argument order
    - Example: `def calculate_event_cost(venue_cost, catering_cost, staff_count)`
2. Keyword Arguments:
    - Must implement functions with keyword arguments and defaults
    - Must demonstrate optional parameters with sensible defaults
    - Example: `def create_event(name, date, venue, max_capacity=100, vip_slots=10, budget=5000)`
3. Variable Arguments:
    - Must use *args to handle variable number of attendees or resources
    - Must use **kwargs to handle variable configuration options
    - Example: `def assign_staff(*staff_members, **responsibilities)`
4. Advanced Argument Handling:
    - Must implement at least one function using keyword-only arguments
    - Must implement at least one function using position-only arguments

- Must demonstrate unpacking lists and dictionaries into function arguments
- Example: `def generate_report(event_data, *, format="detailed", include_financials=True)`

3.3 Return Value Constraints

1. Basic Return Values:
    - Functions must return appropriate data types (numbers, strings, booleans)
    - Must demonstrate early returns for special cases
    - Example: `return is_available if not is_available else check_capacity(venue, attendees)`
2. Compound Return Values:
    - Must return tuples for related multiple values
    - Must return dictionaries for named result sets
    - Must return lists for collections of similar results
    - Example: `return (total_revenue, total_expense, net_profit)`
3. Generator Functions:
    - Must implement at least one generator function using yield
    - Must demonstrate lazy evaluation for event data processing
    - Example: `def process_attendee_data(attendees): for attendee in attendees: yield format_attendee_info(attendee)`
4. Return Value Handling:
    - Must demonstrate unpacking returned tuples
    - Must demonstrate accessing returned dictionary values
    - Must implement proper error handling for return values
    - Example: `revenue, expenses, profit = analyze_event_financials(event_data)`

3.4 Input Data Requirements

1. Event Data:

- Input dictionary must contain complete event information
- Each event must have: name, date, venue, capacity, registered_attendees, status
- Example: `{"name": "Tech Conference 2023", "date": "2023-09-15", "venue": "Convention Center", "capacity": 500, "registered_attendees": 350, "status": "upcoming"}`

2. Attendee Data:

   - Input list must contain at least 10 attendee dictionaries
   - Each attendee must have: id, name, email, ticket_type, check_in_status
   - Example: `{"id": "A12345", "name": "Jane Smith", "email": "jane@example.com", "ticket_type": "VIP", "check_in_status": false}`

3. Resource Data:

   - Input list must contain at least 5 resource dictionaries
   - Each resource must have: type, quantity, cost_per_unit, assigned_to
   - Example: `{"type": "Projector", "quantity": 3, "cost_per_unit": 75.50, "assigned_to": "Main Hall"}`

3. Venue Data:
   - Input dictionary must include complete venue information
   - Must include capacity, layout options, and availability
   - Example: `{"name": "Grand Ballroom", "capacity": 250, "hourly_rate": 350, "layout_options": ["theater", "classroom", "banquet"], "availability": {"2023-09-15": true, "2023-09-16": false}}`

3.5 Output Constraints

1. Display Format:

   - Each analysis result must have descriptive labels
   - Financial metrics must be formatted with appropriate precision
   - Monetary values must include currency symbols
   - Percentages must include % symbol
   - Example: `"Event Profit: $2,500.00 (25.0% margin)"`

4. Output Sections:
   - Event summary with registration statistics
   - Resource allocation breakdown
   - Budget analysis and financial projections
   - Scheduling and capacity optimization
   - Staff assignments and responsibilities

4. Template code Structure:

1. Event Management Functions:

   - `calculate_event_capacity(venue, setup_type)` - calculates maximum capacity

- `register_attendee(event, attendee, *, ticket_type="standard")` - registers attendee
- `calculate_registration_stats(event)` - calculates registration percentages
- `create_pricing_calculator(base_fee)` - returns a pricing function

2. Resource Management Functions:

- `allocate_resources(venue, attendees, event_type)` - determines resource needs
- `calculate_resource_costs(resources, rental_period=1)` - calculates costs
- `check_resource_availability(*resources, event_date)` - checks availability
- `generate_resource_report(**options)` - generates resource report

3. Budget Analysis Functions:

- `categorize_expenses(*expenses)` - sorts expenses by category
- `analyze_budget_variance(planned, actual)` - calculates budget variances
- `calculate_event_profitability(revenue, expenses)` - calculates profitability
- `generate_financial_projection(base_cost, attendees, pricing_tiers, /, *, discount_rate=0)` - projects financials

4. Report Generation Functions:

- `format_currency(amount)` - formats numbers as currency
- `format_percentage(value)` - formats numbers as percentages
- `generate_event_summary(event, attendees, resources)` - creates overall summary
- `attendee_check_in_generator(event_data)` - yields check-in statistics

5. Main Program Function:

- `main()` - main program function executing all demonstrations
- Must execute each function with appropriate arguments
- Must handle return values appropriately
- Must present results in structured format

5. EXECUTION STEPS TO FOLLOW:

1. Run the program
2. Observe event management with different argument passing techniques
3. See resource allocation with various return value formats
4. View budget analysis using nested functions and proper scoping
5. Examine the generator function for attendee check-in data
6. Observe function closures in action with the pricing calculator
7. See comprehensive event report generation combining all calculations