System Requirements Specification Index

For

Customer Support Ticket Tracking Console Application

Version 1.0

IIHT Pvt. Ltd.

fullstack@iiht.com

TABLE OF CONTENTS

Customer Support Ticket Tracking Console

System Requirements Specification

1 Project Abstract

TechHelp Inc. requires a ticket tracking system for their customer support operations. The system will track customer issues, assign priorities, and monitor resolution status using Python's list operations and methods. This console application demonstrates fundamental list operations while allowing support agents to manage customer requests effectively. The system performs critical functions including categorizing tickets by type, sorting by priority, and tracking resolution progress. By implementing these operations with Python lists, the system provides an efficient way for support teams to organize their workload.

2 Business Requirements:

Screen Name	Console input screen
Problem Statement	1. System needs to store and categorize different types of tickets (billing, technical, feature, account) 2. Application must allow tickets to be sorted by different attributes (priority, id) 3. System must support filtering tickets by type, status, or keyword 4. Console should handle different list operation like Addition (+) for combining ticket queues, Slicing for selecting subset of ticket ,List comprehension for filtering tickets , Advanced list methods (sort, append, pop, clear)

3 CONSTRAINTS

3.1 INPUT REQUIREMENTS

1. Ticket Records:
    - Must be stored as dictionaries with fields for id, title, type, priority, status
    - Must be stored in list variable `tickets`
    - Example: `{"id": "T001", "title": "Payment not processing", "type": "billing", "priority": 2, "status": "open"}`
2. Ticket Types:
    - Must be one of: "billing", "technical", "feature", "account"
    - Must be stored as string in ticket's "type" field
    - Example: "billing"
3. Priority Levels:
    - Must be stored as integer in "priority" field
    - Must be between 1-4 (1=critical, 4=low)
    - Example: 2
4. Ticket Status:
    - Must be one of: "new", "open", "resolved", "closed"
    - Must be stored as string in ticket's "status" field
    - Example: "open"
5. Active Queues:
    - Must be stored as list in variable `active_queue`

- Must contain references to tickets from main list
- Must not exceed 5 tickets
- Example: `[tickets[0], tickets[3]]`

6. Predefined Tickets:
   - Must use these exact predefined tickets in the initial ticket list:
     - `{"id": "T001", "title": "Payment not processing", "type": "billing", "priority": 2, "status": "open"}`
     - `{"id": "T002", "title": "Reset password", "type": "account", "priority": 3, "status": "new"}`
     - `{"id": "T003", "title": "Application crashes", "type": "technical", "priority": 1, "status": "open"}`
     - `{"id": "T004", "title": "Add dark mode", "type": "feature", "priority": 4, "status": "new"}`
     - `{"id": "T005", "title": "Renewal failed", "type": "billing", "priority": 2, "status": "open"}`
7. Escalated Tickets:
   - Must use these exact predefined items in the escalated list:
     - `{"id": "E001", "title": "Security breach", "type": "technical", "priority": 1, "status": "new"}`
     - `{"id": "E002", "title": "Double-charged", "type": "billing", "priority": 1, "status": "new"}`

3.2 Operations Constraints

1. Ticket Queue Combination:
   - Must use exact list addition operator `+`
   - Example: `tickets + escalated` combines current tickets with escalated tickets

2. Ticket Subset Selection:
   - Must use exact `tickets[start:end:step]` notation

- Example: `tickets[0:5]` selects first 5 tickets

3. Ticket Filtering:
    - Must use list comprehension
    - Example: `[ticket for ticket in tickets if ticket["type"] == "technical"]`
4. Priority Queue Management:
    - Must use list indexing and operations
    - Must validate against queue constraints
    - Example: Adding a ticket: `active_queue.append(tickets[3])`
5. Ticket Status Updates:
    - Must use list and dictionary operations to update status
    - Example: `tickets[index]["status"] = "resolved"`

3.3 OUTPUT CONSTRAINTS

1. Display Format:
    - - Show ticket ID, title, type, priority, status
    - - Format priority with level indicator
    - - Format status with appropriate visual indicator
    - - Each ticket must be displayed on a new line

2. Required Output Format:
    - - Show "===== TICKET TRACKING SYSTEM ====="
    - - Show "Total Tickets: {count}"
    - - Show "Critical Tickets: {count}"
    - - Show "Current Ticket List:"
    - - Show tickets with format: "{id} | {title} | {type} | Priority: {priority} | {status}"

- - Show "Active Queue:" when displaying active queue
- - Show "Filtered Results:" when displaying search results

4. TEMPLATE CODE STRUCTURE:

1. Data Management Functions:

- `initialize_data()` - creates the initial ticket and escalated lists
- `add_ticket(tickets, ticket)` - adds a ticket to the list
- `remove_ticket(tickets, index)` - removes a ticket at the specified index

2. List Operation Functions:

- - `sort_tickets(tickets, key)` - sorts tickets by key
- - `filter_tickets(tickets, filter_type, value)` - filters tickets by criteria
- - `combine_queues(tickets1, tickets2)` - combines two ticket lists

3. Queue Management:

- - `manage_queue(tickets, active_queue, operation, index)` - handles queue operations

4. Ticket Management:

- - `update_ticket(tickets, index, field, value)` - updates ticket fields

5. Display Functions:

- - `get_formatted_ticket(ticket)` - formats a ticket for display
- - `display_data(data, data_type)` - displays tickets or queue

6. Program Control:

- - `main()` - main program function

5. DETAILED FUNCTION IMPLEMENTATION STRUCTURE

5.1 Data Initialization Functions

1. Write a Python function to initialize ticket tracking system data. Define: `initialize_data()`

The function should:

- Create predefined ticket list with **exact variable name** `tickets`
- Create escalated ticket list with **exact variable name** `escalated`
- Return tuple containing `(tickets, escalated, [])`

Must include these exact predefined tickets:
tickets = [{"id": "T001", "title": "Payment not processing", "type": "billing", "priority": 2, "status": "open"}, {"id": "T002", "title": "Reset password", "type": "account", "priority": 3, "status": "new"}, {"id": "T003", "title": "Application crashes", "type": "technical", "priority": 1, "status": "open"}, {"id": "T004", "title": "Add dark mode", "type": "feature", "priority": 4, "status": "new"}, {"id": "T005", "title": "Renewal failed", "type": "billing", "priority": 2, "status": "open"}]
Must include these exact escalated tickets:
escalated = [{"id": "E001", "title": "Security breach", "type": "technical", "priority": 1, "status": "new"}, {"id": "E002", "title": "Double-charged", "type": "billing", "priority": 1, "status": "new"}]

- Return statement: `return tickets, escalated, []`
- Dictionary structure: Each ticket must contain exactly these keys: "id", "title", "type", "priority", "status"
- Data types: id/title/type/status as strings, priority as integer (1-4)

5.2 Basic List Operations Functions

2. Write a Python function to add a ticket to the list. Define: `add_ticket(tickets, ticket)`

The function should:

- Accept parameters: `tickets` (list), `ticket` (dictionary)
- Validate required fields: `["id", "title", "type", "priority", "status"]`
- Check field existence: `if field not in ticket: raise ValueError(f"Ticket is missing required field: {field}")`

- Validate ticket type: `valid_types = ["technical", "billing", "general", "account", "feature"]`
- Type validation: `if ticket["type"] not in valid_types: raise ValueError("Invalid ticket type")`
- Validate priority range: `if not isinstance(ticket["priority"], int) or ticket["priority"] < 1 or ticket["priority"] > 4:`
- Priority error: `raise ValueError("Priority must be an integer between 1 and 4")`
- Validate status: `valid_statuses = ["new", "open", "resolved", "closed"]`
- Status validation: `if ticket["status"] not in valid_statuses: raise ValueError("Invalid status")`
- **Use append() method**: `tickets.append(ticket)`
- Return updated list: `return tickets`
- Example: add_ticket(tickets, {"id": "T006", "title": "Test", "type": "technical", "priority": 2, "status": "new"})

3. Write a Python function to remove a ticket by index. Define: `remove_ticket(tickets, index)`

The function should:

- Accept parameters: `tickets` (list), `index` (integer)
- Validate index range: `if index < 0 or index >= len(tickets):`
- Raise index error: `raise IndexError("Index out of range")`
- **Use pop() method**: `return tickets.pop(index)`
- Return removed ticket as dictionary
- Example: remove_ticket(tickets, 0) removes first ticket and returns it
- List modification: Original list is modified, ticket is removed permanently

5.3 List Sorting and Filtering Functions

4. Write a Python function to sort tickets by specified key. Define: `sort_tickets(tickets, key)`

The function should:

- Accept parameters: `tickets` (list), `key` (string)
- Validate sort key: `valid_keys = ["id", "priority", "status", "type"]`
- Key validation: `if key not in valid_keys: raise ValueError("Invalid sort key")`

- **Use copy() method**: `sorted_tickets = tickets.copy()` (preserve original)
- **Use sort() method with lambda**: `sorted_tickets.sort(key=lambda ticket: ticket[key])`
- Return sorted list: `return sorted_tickets`
- Sorting behavior: Ascending order (low to high for priority, alphabetical for strings)
- Example: sort_tickets(tickets, "priority") returns list sorted by priority (1, 2, 3, 4)
- Lambda function: Extracts the specified field value for comparison

5. Write a Python function to filter tickets by criteria. Define: `filter_tickets(tickets, filter_type, value)`

The function should:

- Accept parameters: `tickets` (list), `filter_type` (string), `value` (varies)
- Validate filter type: `valid_filters = ["type", "status", "priority", "keyword"]`
- Filter validation: `if filter_type not in valid_filters: raise ValueError("Invalid filter type")`

Use list comprehension for keyword filtering:
 if filter_type == "keyword":   value = value.lower()   return [ticket for ticket in tickets if value in ticket["title"].lower()]

Use list comprehension for priority filtering:
 elif filter_type == "priority":   if isinstance(value, str):   value = int(value) # Convert string to int   return [ticket for ticket in tickets if ticket["priority"] == value]

Use list comprehension for type/status filtering:
 else: return [ticket for ticket in tickets if ticket[filter_type] == value]

-
- Case handling: Keyword search is case-insensitive using .lower()
- Type conversion: Priority values converted from string to int if needed
- Example: filter_tickets(tickets, "type", "billing") returns all billing tickets

5.4 List Combination and Priority Functions

6. Write a Python function to combine two ticket queues. Define: `combine_queues(tickets1, tickets2)`

The function should:

- Accept parameters: `tickets1` (list), `tickets2` (list)
- **Use addition operator (+)**: `return tickets1 + tickets2`
- Create new list: Does not modify original lists
- Maintains order: tickets1 items first, then tickets2 items
- Example: combine_queues(tickets, escalated) creates combined list
- List concatenation: Joins two lists into single new list

7. Write a Python function to get tickets by priority level. Define: `get_priority_tickets(tickets, priority_level)`

The function should:

- Accept parameters: `tickets` (list), `priority_level` (integer)
- Validate priority: `if not isinstance(priority_level, int) or priority_level < 1 or priority_level > 4:`
- Priority error: `raise ValueError("Priority level must be between 1 and 4")`
- **Use list comprehension**: `return [ticket for ticket in tickets if ticket["priority"] == priority_level]`
- Filter by exact match: Only tickets with exact priority level
- Example: get_priority_tickets(tickets, 1) returns all critical (priority 1) tickets

5.5 Queue Management Functions

8. Write a Python function to manage active ticket queue. Define: `manage_queue(tickets, active_queue, operation, index=None)`

The function should:

- Accept parameters: `tickets` (list), `active_queue` (list), `operation` (string), `index` (optional integer)
- Validate operation: `valid_operations = ["add", "remove", "clear"]`
- Operation validation: `if operation not in valid_operations: raise ValueError("Invalid operation")`

Add operation implementation:
if operation == "add":   if index is None: raise ValueError("Index is required for add operation")   if index < 0 or index >= len(tickets):   raise IndexError("Ticket index out of range")   if len(active_queue) >= 5:   raise ValueError("Active queue is at maximum capacity (5 tickets)")   active_queue.append(tickets[index])

Remove operation implementation:
 elif operation == "remove":   if index is None: raise ValueError("Index is required for remove operation")   if index < 0 or index >= len(active_queue):   raise IndexError("Queue index out of range")   return active_queue.pop(index)

Clear operation implementation:
 elif operation == "clear":   active_queue.clear()

- Return value: `return active_queue` (except for remove, which returns removed ticket)
- Queue capacity: Maximum 5 active tickets at once
- Reference management: Active queue contains references to tickets from main list

5.6 Ticket Update Functions

9. Write a Python function to update ticket fields. Define: `update_ticket(tickets, index, field, value)`

The function should:

- Accept parameters: `tickets` (list), `index` (integer), `field` (string), `value` (varies)
- Validate index: `if index < 0 or index >= len(tickets): raise IndexError("Ticket index out of range")`
- Validate field: `valid_fields = ["status", "priority", "type"]`
- Field validation: `if field not in valid_fields: raise ValueError("Invalid field")`
- Get ticket reference: `ticket = tickets[index]`

Status update implementation:
 if field == "status":   valid_statuses = ["new", "open", "resolved", "closed"]   if value not in valid_statuses: raise ValueError("Invalid status")   ticket["status"] = value

Priority update implementation:
 elif field == "priority":   try: priority = int(value)   if priority < 1 or priority > 4:   raise ValueError()   except ValueError: raise ValueError("Priority must be an integer between 1 and 4")   ticket["priority"] = priority

-

Type update implementation:
 elif field == "type":   valid_types = ["technical", "billing", "general", "account", "feature"]   if value not in valid_types: raise ValueError("Invalid type")   ticket["type"] = value

- Return updated ticket: `return ticket`
- Direct modification: Updates the ticket in the original list

5.7 Display Functions

10. Write a Python function to format ticket for display. Define: `get_formatted_ticket(ticket)`

The function should:

- Accept parameter: `ticket` (dictionary)
- Create priority indicators: `priority_indicators = ["CRITICAL", "HIGH", "MEDIUM", "LOW"]`
- Get priority text: `priority_text = priority_indicators[ticket["priority"] - 1]`
- Format status: `status_text = ticket["status"].upper()`

Return formatted string:
return f"{ticket['id']} | {ticket['title']} | {ticket['type']} | Priority: {priority_text} ({ticket['priority']}) | Status: {status_text}"

- String formatting: Uses f-strings for clean, readable output
- Priority mapping: 1=CRITICAL, 2=HIGH, 3=MEDIUM, 4=LOW
- Status formatting: Converts status to uppercase for visibility
- Example output: "T001 | Payment not processing | billing | Priority: HIGH (2) | Status: OPEN"

11. Write a Python function to display ticket data. Define: `display_data(data, data_type="tickets")`

The function should:

- Accept parameters: `data` (list), `data_type` (string, default "tickets")
- Handle empty data: `if not data: print("No tickets to display."); return`

Display headers based on type:
if data_type == "tickets":   print("Current Ticket List:")elif data_type == "queue":   print("Active Queue:")elif data_type == "filtered":   print("Filtered Results:")

- **Display each ticket**: `for ticket in data: print(get_formatted_ticket(ticket))`
- Header customization: Different headers for different data types

- Formatted output: Uses get_formatted_ticket() for consistent formatting

5.8 Main Program Implementation

12. Write a Python main program for ticket tracking system. Define: `main()`

The program should:

- **Initialize system data**: Call initialize_data() and assign to variables `tickets`, `escalated`, `active_queue`
- **Display welcome information**: Show system title, total ticket count, and critical ticket count
- **Implement menu-driven interface**: Create a loop with numbered options for different operations
- **Handle user input**: Get user choices and call appropriate functions based on selection
- **Provide error handling**: Use try-catch blocks to handle invalid inputs and function errors
- **Exit gracefully**: Include option to terminate the program and cleanup

Menu System Requirements:

- **Option 1**: View all tickets using display function
- **Option 2**: Add or remove tickets with user input validation
- **Option 3**: Sort tickets by user-selected criteria (id, priority, status, type)
- **Option 4**: Filter tickets by various criteria with user input
- **Option 5**: Manage active queue operations (add, remove, view, clear)
- **Option 6**: Process escalated tickets by combining with main list
- **Option 7**: Update specific ticket fields with validation
- **Option 0**: Exit the program

5.9 Implementation Requirements

List Operations Usage Requirements:

- **append()**: Used in add_ticket() and manage_queue() for adding items to lists
- **pop()**: Used in remove_ticket() and manage_queue() for removing specific items
- **clear()**: Used in manage_queue() for emptying the active queue
- **copy()**: Used in sort_tickets() to preserve original list order
- **sort()**: Used in sort_tickets() with lambda function for custom sorting
- **Addition (+)**: Used in combine_queues() for list concatenation
- **List comprehension**: Used in filter_tickets() and get_priority_tickets() for filtering
- **Indexing**: Used throughout for accessing specific tickets by position

Data Structure Requirements:

- **Ticket dictionary structure**: Must contain exactly these keys: "id", "title", "type", "priority", "status"
- **List variable names**: Must use exact names: `tickets`, `escalated`, `active_queue`
- **Parameter names**: Must use exact parameter names in function definitions
- **Return values**: Functions must return appropriate data types (list, dict, or modified list)

6. Execution Steps to Follow:

1. Run the program

2. View the main menu

3. Select operations:

- Option 1: View Tickets
- Option 2: Add/Remove Ticket
- Option 3: Sort Tickets
- Option 4: Filter Tickets
- Option 5: Queue Operations
- Option 6: Process Escalated
- Option 7: Update Ticket
- Option 0: Exit

4. Perform operations on the ticket list

5. View results after each operation

6. Exit program when finished

<u>Execution Steps to Follow:</u>

● All actions like build, compile, running application, running test cases will be through Command Terminal.

● To open the command terminal the test takers, need to go to Application menu (Three horizontal lines at left top) -> Terminal -> New Terminal

● This editor Auto Saves the code

● If you want to exit(logout) and continue the coding later anytime (using Save & Exit option on Assessment Landing Page)

- These are time bound assessments the timer would stop if you logout and while logging in back using the same credentials the timer would resume from the same time it was stopped from the previous logout.

- To launch application: python3 filename.py

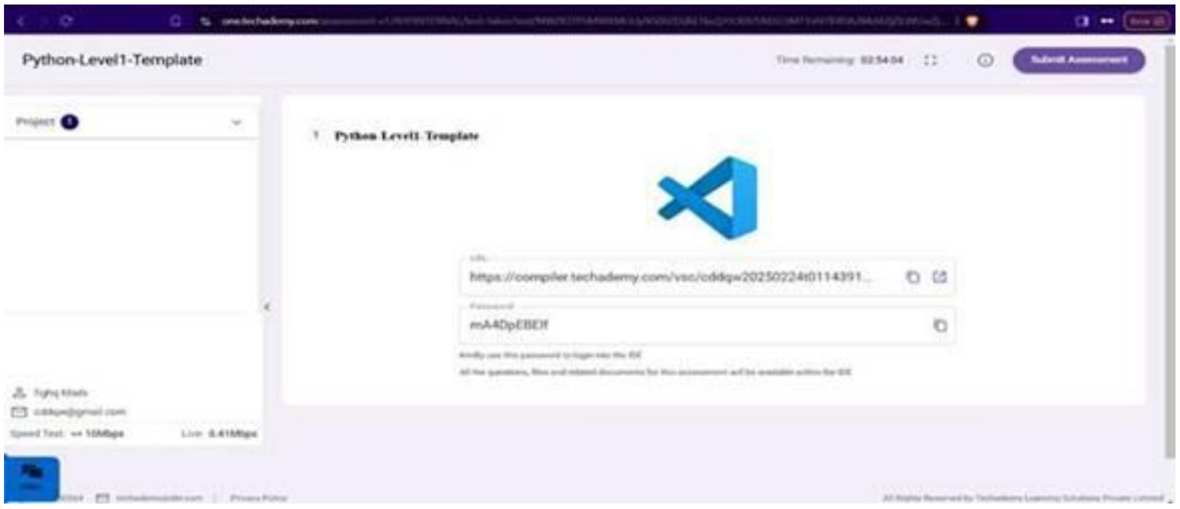
- To run Test cases: python3 -m unittest

<u>Screen shot to run the program</u>

To run the application

python3 filename.py

To run the testcase python3-m unittest



Python-Level1-Template

Time Remaining: 02:54:04

Submit Assessment

Project 1

1 Python Level1 Template

URL
https://compiler.techademy.com/vsc/cddqw20250224t0114391...

Password
mA4DpEBEIf

Kindly use this password to login into the IDE

All the questions, files and related documents for this assessment will be available within the IDE

Speed Test: 10Mbps   Live: 0.41Mbps

- Once you are done with development and ready with submission, you may navigate to the previous tab and submit the workspace. It is mandatory to click on “Submit Assessment” after you are done with code.