**System Requirements Specification Index**

**For**

**Date and Time Processor**

**Version 1.0**

**IIHT Pvt. Ltd.**

**fullstack@iiht.com**

**Table of Contents**

1 Project Abstract

2 Business Requirements

3 Constraints

4 Template Code Structure

5 Execution Steps to Follow

**Date and Time Processor**

**System Requirements Specification**

**1** **Project Abstract**

A healthcare scheduling system needs reliable date and time processing capabilities to manage patient appointments, medication schedules, and staff rotations. This assignment focuses on implementing fundamental datetime operations using Python's datetime module to create a practical scheduling tool. Students will learn how to perform essential datetime manipulations including conversions, calculations, formatting, and timezone handling for real-world scheduling scenarios.

**2** **Business Requirements:**

| Screen Name | Console input screen |
| --- | --- |
| Problem Statement | 1. System must implement core datetime manipulation functions  2. Functions must handle date arithmetic, formatting, and timezone conversions  3. All functions require proper documentation (docstrings)  4. System must demonstrate practical applications of datetime processing  5. Error handling must validate inputs and handle edge cases |

**3** **Constraints**

**3.1** **Input Requirements**

1. Date and Time Formats:

o String dates must follow ISO format (YYYY-MM-DD)

o String datetimes must follow format (YYYY-MM-DD HH:MM:SS)

o Example: `"2025-03-19 14:30:00"`

2. Function Parameters:

o Date strings must be validated before processing

o Timezone values must be integer offsets (hours from UTC)

o Duration inputs must be integers for days, hours, minutes

o Date ranges must have valid start and end dates

**3.2** **Function Constraints**

1. Function Definition:

o Each function must have a specific datetime manipulation purpose

o Must include docstrings with examples

o Example: `def convert\_string\_to\_datetime(date\_string):`

2. Datetime Operations:

o Must use Python's `datetime` module

o Must handle string parsing and formatting

§ `convert\_string\_to\_datetime()`: Parse string to datetime object

§ `format\_datetime()`: Format datetime to specified string pattern

§ `calculate\_date\_difference()`: Find days between two dates

§ `add\_time\_duration()`: Add specified time to a datetime

§ `get\_day\_of\_week()`: Return weekday name for given date

§ `convert\_timezone()`: Convert datetime between timezone offsets

3. Return Values:

o Functions must return appropriate types (datetime objects, strings, integers)

o Functions should return None or empty results when inputs are invalid

o The calculate\_date\_difference function must return a dictionary with different time units

4. Error Handling:

o Functions should validate input types and formats

o Handle invalid dates, times, and timezone offsets

o Raise appropriate exceptions with clear error messages

**3.3** **Output Constraints**

1. Display Format:

o Console output must be clearly labeled

o Example: `Time difference: 7 days, 5 hours, 30 minutes`

o Results should be formatted for readability

o Main function should demonstrate each datetime operation with labeled results

**4. Template code Structure:**

**1.** Datetime Conversion Functions:

o `convert\_string\_to\_datetime(date\_string)` - converts string to datetime object

o `format\_datetime(dt, format\_string)` - formats datetime to specified string format

**2.** Datetime Calculation Functions:

o `calculate\_date\_difference(start\_date, end\_date)` - calculates time between dates

o `add\_time\_duration(dt, days=0, hours=0, minutes=0)` - adds time to datetime

o `get\_day\_of\_week(date\_string)` - returns weekday name for date

**3.** Timezone Functions:

o `convert\_timezone(dt, source\_offset, target\_offset)` - converts datetime between timezone offsets

**4.** Program Control:

o `main()` - demonstrates all datetime functions with sample inputs

o Should display results clearly with appropriate formatting

# **5. DETAILED FUNCTION IMPLEMENTATION GUIDE**

## **5.1 String to Datetime Conversion Functions**

1. **Write a Python function to convert string dates to datetime objects.** Define: convert\_string\_to\_datetime(date\_string) The function should:
   * Accept a string parameter in ISO format: "YYYY-MM-DD" or "YYYY-MM-DD HH:MM:SS"
   * Validate that input is a string type, raise TypeError for non-string inputs
   * Use datetime.strptime() to parse the string with appropriate format patterns
   * Try parsing with time format "%Y-%m-%d %H:%M:%S" first
   * If that fails, try parsing date-only format "%Y-%m-%d"
   * For date-only format, set time components to 00:00:00
   * Raise ValueError with clear message for invalid date formats
   * Handle edge cases like leap years (2024-02-29) and century boundaries
   * Return datetime.datetime object representing the parsed date
   * Example: convert\_string\_to\_datetime("2025-03-19") returns datetime(2025, 3, 19, 0, 0)
   * Example: convert\_string\_to\_datetime("2025-03-19 14:30:00") returns datetime(2025, 3, 19, 14, 30)

## **5.2 Datetime Formatting Functions**

1. **Write a Python function to format datetime objects as strings.** Define: format\_datetime(dt, format\_string="%Y-%m-%d %H:%M:%S") The function should:
   * Accept a datetime.datetime object as first parameter
   * Accept optional format string parameter with default "%Y-%m-%d %H:%M:%S"
   * Validate that dt is a datetime object, raise TypeError for invalid types
   * Validate that format\_string is a string, raise TypeError for invalid format types
   * Use datetime.strftime() method to format the datetime object
   * Support all standard strftime format codes (%Y, %m, %d, %H, %M, %S, %A, %B, etc.)
   * Handle edge dates like minimum date (0001-01-01) and maximum date (9999-12-31)
   * Return formatted string representation of the datetime
   * Example: format\_datetime(datetime(2025, 3, 19, 14, 30), "%B %d, %Y at %I:%M %p") returns "March 19, 2025 at 02:30 PM"
   * Example: format\_datetime(datetime(2025, 3, 19), "%Y-%m-%d") returns "2025-03-19"

## **5.3 Date Calculation Functions**

1. **Write a Python function to calculate time differences between dates.** Define: calculate\_date\_difference(start\_date, end\_date) The function should:
   * Accept two parameters that can be either datetime objects or date strings
   * Convert string inputs to datetime objects using convert\_string\_to\_datetime()
   * Validate that both parameters are datetime objects after conversion
   * Calculate time difference using datetime subtraction (end\_date - start\_date)
   * Handle both positive and negative differences (future and past dates)
   * Use timedelta.total\_seconds() to get precise time difference
   * Calculate days using timedelta.days property
   * Calculate total hours as total\_seconds // 3600
   * Calculate total minutes as total\_seconds // 60
   * Return dictionary with keys: "days", "hours", "minutes", "total\_seconds"
   * All values should be integers (use int() conversion for calculated values)
   * Handle same date inputs (should return all zeros)
   * Support mixed input types (string + datetime combinations)
   * Example: calculate\_date\_difference("2025-03-19", "2025-03-26") returns {"days": 7, "hours": 168, "minutes": 10080, "total\_seconds": 604800}
2. **Write a Python function to add time durations to datetime objects.** Define: add\_time\_duration(dt, days=0, hours=0, minutes=0) The function should:
   * Accept datetime object or string as first parameter
   * Accept optional integer parameters for days, hours, and minutes (default 0)
   * Convert string input to datetime object using convert\_string\_to\_datetime()
   * Validate that dt is datetime object after conversion
   * Validate that all duration parameters are integers, raise TypeError for non-integers
   * Support negative duration values for subtracting time
   * Use timedelta(days=days, hours=hours, minutes=minutes) to create duration
   * Add timedelta to datetime using + operator
   * Handle date boundary crossings (day, month, year changes)
   * Handle large duration values that span multiple years
   * Return new datetime object with duration added
   * Preserve original datetime object (don't modify in place)
   * Example: add\_time\_duration(datetime(2025, 3, 19), days=2, hours=5) returns datetime(2025, 3, 21, 5, 0)
   * Example: add\_time\_duration("2025-03-19 10:00:00", days=-1, hours=-2) returns datetime(2025, 3, 18, 8, 0)

## **5.4 Date Analysis Functions**

1. **Write a Python function to determine the day of week for any date.** Define: get\_day\_of\_week(date\_string) The function should:
   * Accept either string date or datetime object as parameter
   * Convert string input to datetime object using convert\_string\_to\_datetime()
   * Validate that parameter is datetime object after conversion
   * Use strftime("%A") to get full weekday name in English
   * Return string with full weekday name (not abbreviation)
   * Handle all edge cases including leap years and century boundaries
   * Support both date-only and datetime inputs
   * Ignore time component if present (focus only on date for weekday)
   * Return one of: "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
   * Handle historical dates and far future dates consistently
   * Example: get\_day\_of\_week("2025-03-19") returns "Wednesday"
   * Example: get\_day\_of\_week(datetime(2025, 3, 19, 14, 30)) returns "Wednesday"

## **5.5 Timezone Conversion Functions**

1. **Write a Python function to convert datetime between timezone offsets.** Define: convert\_timezone(dt, source\_offset, target\_offset) The function should:
   * Accept datetime object or string as first parameter
   * Accept integer source timezone offset in hours (-12 to +14 range)
   * Accept integer target timezone offset in hours (-12 to +14 range)
   * Convert string input to datetime object using convert\_string\_to\_datetime()
   * Validate that dt is datetime object after conversion
   * Validate that offsets are integers, raise TypeError for non-integers
   * Validate offset ranges: -12 ≤ offset ≤ +14, raise ValueError for invalid ranges
   * Calculate hour difference: target\_offset - source\_offset
   * Use timedelta(hours=hours\_diff) to create time adjustment
   * Add timedelta to original datetime to get converted time
   * Handle date boundary crossings when timezone conversion changes the date
   * Handle year boundary crossings (December 31 → January 1)
   * Preserve minute and second components exactly
   * Return new datetime object in target timezone
   * Handle same timezone conversion (should return identical datetime)
   * Example: convert\_timezone("2025-03-19 14:30:00", -5, -8) returns datetime(2025, 3, 19, 11, 30) (Eastern to Pacific)
   * Example: convert\_timezone(datetime(2025, 3, 19, 23, 0), -5, 0) returns datetime(2025, 3, 20, 4, 0) (crosses to next day)

## **5.6 Main Demonstration Function**

1. **Write a Python function to demonstrate all datetime processing capabilities.** Define: main() The function should:
   * Print clear section headers for each demonstration
   * Use provided sample data: appointment\_date, surgery\_date, staff\_shift\_start
   * Demonstrate string to datetime conversion with clear before/after output
   * Show datetime formatting with custom format patterns including:
     + Full text date format: "%A, %B %d, %Y at %I:%M %p"
     + Date-only format: "%Y-%m-%d"
     + Time-only format: "%H:%M:%S"
   * Calculate and display time differences between appointments with all units
   * Demonstrate duration addition for follow-up appointments (add 7 days, 1 hour, 30 minutes)
   * Show day of week determination for multiple dates
   * Demonstrate timezone conversion between Eastern (UTC-5) and Pacific (UTC-8)
   * Format all output for readability with descriptive labels
   * Handle any potential errors gracefully
   * Print completion message at the end
   * Use proper string formatting to display results clearly
   * Example output sections:
     + "1. String to Datetime Conversion:"
     + "2. Datetime Formatting:"
     + "3. Date Difference Calculation:"
     + "4. Time Duration Addition:"
     + "5. Day of Week Determination:"
     + "6. Timezone Conversion:"

## **5.7 Input Validation and Error Handling Requirements**

All functions must implement comprehensive error handling:

**Type Validation:**

* Check input types before processing
* Raise TypeError with descriptive messages for invalid types
* Handle both string and datetime inputs where specified

**Value Validation:**

* Validate date format strings match expected patterns
* Check timezone offset ranges (-12 to +14)
* Ensure duration parameters are integers
* Validate that dates are within Python's datetime range

**Format Validation:**

* Accept ISO format dates: "YYYY-MM-DD" and "YYYY-MM-DD HH:MM:SS"
* Reject other date formats with clear error messages
* Handle leading zeros in date components
* Support both date-only and datetime strings

**Edge Case Handling:**

* Process leap year dates correctly (February 29)
* Handle century and millennium boundaries
* Support minimum date (0001-01-01) and maximum date (9999-12-31)
* Manage timezone conversions that cross date boundaries
* Handle large duration additions that span multiple years

**Return Value Consistency:**

* Always return specified types (datetime objects, strings, dictionaries)
* Maintain precision for time calculations
* Preserve all datetime components unless specified otherwise
* Return consistent data structures with required keys for dictionaries

**6. Execution Steps to follow:**

1. Run the program

2. Observe how each datetime function processes date and time data

3. Test with sample inputs for string parsing, calculations, and formatting

4. Experiment with different datetime operations for scheduling scenarios

Execution Steps to Follow:

● 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) then you need to use CTRL+Shift+B -command compulsorily on code IDE. This will push or save the updated contents in the internal git/repository. Else the code will not be available in the next login.

● 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

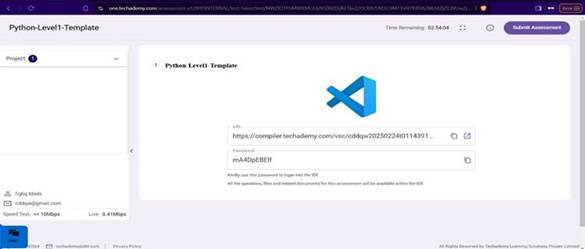
● Before Final Submission also, you need to use CTRL+Shift+B - command compulsorily on code IDE, before final submission as well. This will push or save the updated contents in the internal git/repository, and will be used to evaluate the code quality.

Screen shot to run the program

To run the application

Python3 filename.py

To run the testcase python -m unittest



● 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.