UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Workshop Python 3.9, 3.10, 3.11, 3.12

Workshop 1: Developing Functions for User Session Management and Activity Reporting

Context: You are tasked with developing a series of functions for a user session management and activity tracking application. The goal is to structure the code to:

1. Create a user session configuration based on specific preferences.
2. Generate a session report file for each login.
3. Display the last login time according to each user's timezone.
4. Validate and archive session information for future reference.

- The functions should be modular, maintainable, and interact with each other to produce a logical and complete workflow. You must use the built-in collection types (dict, list, etc.) to annotate parameter and return types in the functions, without using imports from the typing module.

Workshop 1: Developing Functions for User Session Management and Activity Reporting

Configuration JSON Object Details

1. Default Configuration (config_default):

- "theme": Application theme, default value "light".
- "notifications": Boolean to enable/disable notifications, default value True.
- "timezone": Time zone, default value "UTC".
- "session_timeout": Session time in minutes, default value 15.

2. User Preferences (config_user):

- "theme": User-chosen theme, e.g., "dark".
- "timezone": User's time zone, e.g., "America/New_York".
- "session_timeout": Customized session time, e.g., 30.

Workshop 1: Developing Functions for User Session Management and Activity Reporting

Step 1: Creating the Session Configuration

- Develop a function named create_session_config that creates a complete session configuration for a user by combining config_default with config_user.

Function Details:

- **Inputs:**
  - A dictionary containing default settings (config_default).
  - A dictionary containing user preferences (config_user).
- **Output: A combined dictionary** that integrates the user's custom values while respecting default settings.

Instruction:

Use dict to annotate the parameter and return types. This function will be used initially to set up the user session configuration before generating the session report.

Workshop 1: Developing Functions for User Session Management and Activity Reporting

Step 2: Generating and Structuring a Session Report File

- Create a function named generate_session_report_filename that generates a structured file name for each user session, using the current date and the timezone defined in the session configuration.

Function Details:

- **Inputs:**
  - The session configuration created in the previous step, including the user's timezone.
- **Output:** A string representing the session report file name, formatted with the connection date and cleaned to display only relevant information.

Example of Expected File Name:

The file name should be unique for each connection day. For example, if the user logs in on November 1, 2024, in the "America/New_York" timezone, the file name could be session_report_2024-11-01_America_New_York.json.
Use dict to annotate the type of the session configuration parameter and str for the return type.

Workshop 1: Developing Functions for User Session Management and Activity Reporting

Step 3: Displaying the Last Login Time Based on Timezone

- Develop a function named display_last_login that takes the last login time in UTC and displays it according to the user's timezone.

Function Details:

- Inputs:
  - A UTC datetime representing the user's last login (e.g., "2024-11-01 14:30:00 UTC").
  - The user's timezone extracted from the session configuration.
- **Output:** A string representing the last login time formatted in the user's timezone.

Example of Expected Result:

If the last login time in UTC is "2024-11-01 14:30:00" and the user's timezone is "America/New_York", the function should return "2024-11-01 10:30:00".

Use str to annotate the return type.

Workshop 1: Developing Functions for User Session Management and Activity Reporting

Step 4: Validating and Archiving the Session Report

- Create a function named validate_and_archive_session_report to validate the session report and archive it. This function should check the validity of the session time before archiving the report.

Function Details:

- Inputs:
  - An integer representing the session time in minutes.
  - The generated session report file name.
    Output: Confirmation of the session report's archiving if validation is successful; otherwise, an error is raised.

Example of Validation:

If the session time is 30, the function should confirm the report's archiving. If the session time is 0 or negative, the function should raise an error indicating that the session time is invalid.
Use int to annotate the session time parameter and str for the file name return type.

Workshop 2: Advanced Enhancements for the User Session Management Application

Objectives

In this second part of the workshop, you will continue to enhance the user session management application by:

1. **Adding precise adjustments to session durations** to personalize the user experience.
2. **Identifying the most common session durations** to analyze user preferences.
3. **Archiving session reports asynchronously** to optimize performance.
4. **Simplifying the management and analysis of multiple archived report files.**

Workshop 2: Advanced Enhancements for the User Session Management Application

Step 1: Precise Adjustment of Session Duration

In some cases, administrators may want to finely adjust session durations to test specific configurations and optimize the user experience. This step will allow you to make precise adjustments to the session duration using advanced Python 3.9 floating-point functions.

Instructions

1. **Create a function `adjust_session_timeout()`:** This function will take as input:
   - The session configuration (`session_config`), a dictionary containing the user's settings.
   - A target value for the session duration.

   Use `math.nextafter()` to incrementally adjust the session duration stored in `session_config["session_timeout"]` toward this target value.

2. **Create a function `get_session_timeout_ulp()`:** This function will take as input:
   - The session duration (a floating-point number).

   It should return the ULP (Unit in the Last Place), representing the smallest possible adjustment for this duration. Use `math.ulp()` to retrieve this value.

Workshop 2: Advanced Enhancements for the User Session Management Application

Step 2: Analyzing the Most Common Session Durations

To better understand user preferences, it is helpful to identify the most commonly chosen session durations. This information will help customize the app's default settings to meet user expectations.

Instructions

1. **Simulate a day's worth of user sessions:** Create a list containing various session durations (in minutes) to simulate a day's sessions. For example, durations could include 15, 30, 45, 60, etc., with repetitions to reflect common choices.
2. **Create a function `find_common_session_durations()`:** This function will take as input:
   - The list of session durations for the day.

   Use `statistics.multimode()` to identify the most frequent values in the list. The function should return a list of the most common session durations.

Workshop 2: Advanced Enhancements for the User Session Management Application

Step 3: Asynchronous Archiving of Session Reports

In a production application, each session generates a report stored in a file. To avoid slowing down the application, you will set up asynchronous archiving using Python's new asynchronous features.

Instructions

1. **Create a synchronous function `archive_report()`:** This function will take as input:
   - The session report filename (e.g., `"session_report_2024-11-01.json"`).
   - The session report content as a string (e.g., a JSON string with session details).

   This function will write the report content to the specified file.
2. **Create an asynchronous function `async_archive_reports()`:** This function will take as input:
   - A dictionary with filenames as keys and report content as values.

   For each report in the dictionary, use `asyncio.to_thread` to execute `archive_report()` asynchronously, allowing each file to be archived without blocking the main application.
3. **Test asynchronous archiving:** Generate a few sample session reports and archive them using `async_archive_reports()`.

Workshop 2: Advanced Enhancements for the User Session Management Application

Step 4: Reading and Analyzing Archived Reports

Regular session archiving results in multiple report files over time. To verify or analyze this data, you will read multiple reports at once, using a single `with` statement to open several files.

Instructions

1. **Create multiple sample report files:** Make sure to have several simulated JSON report files, each containing session information to represent archived reports.
2. **Create a function `read_multiple_reports()`:** This function will take as input:
   - A list of report filenames.

   Use a single `with` statement to open all files simultaneously, then read and display their content to verify or analyze the data.

Workshop 3: Managing and Validating User Sessions with Python 3.10 Features

Objective

Enhance the user session management application by implementing a **role-based access control system** that handles different types of user profiles (e.g., **admins**, **members**, **guests**) and provides role-specific access validation. Use Python 3.10's structured pattern matching, union type annotations, asynchronous socket handling, and improved error messages to build a robust system that can dynamically verify and process user permissions.

Scenario

Your application is now designed to support multiple types of users, each with distinct roles and permissions. The system must:

1. **Validate user roles** and **grant or deny access** based on these roles.
2. **Handle incoming network requests asynchronously** to scale with multiple connections.
3. **Provide clear error feedback** when incorrect or incomplete profile data is received.

Workshop 3: Managing and Validating User Sessions with Python 3.10 Features

Step 1: Define User Roles with Structured Pattern Matching

Implement a function, `validate_user_access`, to verify user access based on role-specific requirements. For example:

- **Admins** have access to all resources.
- **Members** have limited access, depending on their subscription status.
- **Guests** have view-only access and cannot modify data.

Instructions:

Workshop 3: Managing and Validating User Sessions with Python 3.10 Features

1. **Create a function `validate_user_access(user_profile)`**:
   - The `user_profile` parameter is a dictionary with keys like `role`, `name`, and `subscription_status` (for members).
   - Use pattern matching to handle different roles:
     - **Case 1**: If the user is an admin (`role="admin"`), return "Access granted to all resources."
     - **Case 2**: If the user is a member (`role="member"`) and has an active subscription, return "Access granted to member resources."
     - **Case 3**: If the user is a member but has an inactive subscription, return "Limited access: please renew your subscription."
     - **Case 4**: If the user is a guest (`role="guest"`), return "View-only access granted."
     - **Wildcard Case (_)**: Return "Access denied: unknown role" for profiles that don't match any known structure.

Workshop : Managing and Validating User Sessions with Python 3.10 Features

Step 2: Asynchronous Handling of Access Requests

To manage multiple user access requests efficiently, implement asynchronous handling for each request.

Instructions:

1. **Create an asynchronous function** `process_access_request(reader, writer)`:
   - This function will simulate receiving a user profile request over a network connection.
   - Call `validate_user_access` to determine access permissions based on the user profile received, and send the response back to the client.
2. **Set up an asynchronous server function** `start_access_server()`:
   - Use a socket to listen for incoming connections.
   - Accept connections asynchronously with `asyncio.connect_accepted_socket()`.
   - For each connection, create a task to handle it with `process_access_request`.

Workshop 3: Managing and Validating User Sessions with Python 3.10 Features

Step 3: Union Type Annotations for Flexibility in Profile Data

To handle different types of input data for validation, update the type annotations using union types.

Instructions:

1. **Update the function** `validate_user_access(user_profile: dict | None) -> str`:
    - This allows `user_profile` to be `None`, which could occur if a request is missing data.
    - If `user_profile` is `None`, return "Error: Missing profile data."

Workshop 3: Managing and Validating User Sessions with Python 3.10 Features

Step 4: Enhanced Error Handling for Missing or Invalid Data

To provide clearer feedback, implement validation and error handling that informs the user if their profile data is incomplete or incorrect.

Instructions:

1. **Introduce validation errors**:
   - Modify `validate_user_access` to check for required fields like `role`.
   - If `role` or any other critical field is missing, raise a `ValueError` with a message specifying the missing field.
2. **Catch validation errors** in `process_access_request`:
   - If an error occurs, return a message like "Validation error: missing 'role' in profile data."

Workshop 4: Advanced Exception Handling and Async Task Management in Python 3.11

Objective: Enhance the user session management application by implementing advanced exception handling, type annotations, and asynchronous task management using Python 3.11 features:

1. **Exception Notes** to add detailed context when handling errors.
2. **Exception Groups** to handle multiple exceptions in a structured way.
3. **Self Typing** for cleaner type annotations in class methods.
4. **Enhanced Error Messages** for more precise debugging.
5. **Async Task Groups** for efficient management of concurrent tasks.

Workshop 4: Advanced Exception Handling and Async Task Management in Python 3.11

Scenario: In this extended version of the session management application, you will:

1. **Handle errors more effectively** by adding context and grouping exceptions to improve debugging.
2. **Annotate methods with the `Self` type** to simplify type hints within the class.
3. **Manage concurrent user requests** using `TaskGroup` for efficient asynchronous task grouping.

Imagine the application now handles multiple user requests simultaneously. If errors occur (e.g., missing data, invalid role, or connection issues), they should be clearly grouped and logged. Additionally, each request handler will be part of an async task group to ensure smooth handling of concurrent connections.

Workshop 4: Advanced Exception Handling and Async Task Management in Python 3.11

Step 1: Using Exception Notes for Detailed Error Context

Python 3.11 allows you to add custom notes to exceptions to provide additional context. This feature will help you describe issues more precisely, such as when specific fields are missing in the user profile.

Instructions:

1. **Modify the `validate_user_access` function** to check for the required fields (`role`, `name`, etc.) in `user_profile`.
2. **Raise a `ValueError` if a required field is missing**, and use `e.add_note()` to add context to the exception.
   - For example, add a note specifying which field is missing, such as `"The field 'role' is missing from the user profile."`
3. **Handle the exception** in `process_access_request` and print the full error message, including the note, to see how it enhances debugging.

Workshop 4: Advanced Exception Handling and Async Task Management in Python 3.11

Step 2: Grouping Exceptions for Multiple Errors

Python 3.11's `ExceptionGroup` and `except*` notation allow you to handle multiple exceptions that occur within the same block. This will be useful if multiple validation errors can occur simultaneously (e.g., missing `role` and `name`).

Instructions:

1. **Update the `validate_user_access` function** to check for multiple fields.
   - Use an `ExceptionGroup` to raise grouped exceptions if multiple fields are missing.
2. **Catch grouped exceptions** in `process_access_request` using `except*` notation.
   - Print out each exception and its note for a clear summary of all issues found in the user profile.

Workshop 4: Advanced Exception Handling and Async Task Management in Python 3.11

Step 3: Self Typing for Class Methods

Python 3.11 adds the `Self` type in the `typing` module, which allows you to annotate that a method returns an instance of its own class. This will improve readability and type hinting in classes that manage user sessions.

Instructions:

1. **Create a `UserSession` class** to manage user sessions. Include methods such as `start_session` and `end_session`.
   - Use `Self` as the return type in methods that return an instance of `UserSession`.
2. **Update type annotations** for any methods that return `self` or an instance of `UserSession`.

Workshop 4: Advanced Exception Handling and Async Task Management in Python 3.11

Step 4: Improved Error Messages for Debugging

Python 3.11's error messages are more precise, showing exactly where an issue occurs. Use these enhanced error messages to help identify and correct any issues with the code.

Instructions:

1. **Intentionally introduce a syntax error** in the code (such as a missing parenthesis) to see the new error message format.
2. **Correct the error** based on the feedback from the error message.

Workshop 4: Advanced Exception Handling and Async Task Management in Python 3.11

Step 5: Asynchronous Task Groups for Concurrent User Connections

The new `TaskGroup` class in `asyncio` allows you to manage groups of tasks more effectively than `asyncio.gather()`, making it ideal for handling multiple user connections concurrently.

Instructions:

1. **Update** `start_access_server` to use a `TaskGroup` for handling concurrent requests.
    - Each user connection should be processed as part of the `TaskGroup`.
2. **Add exception handling within the TaskGroup** to capture and log any issues that arise during individual tasks.

Workshop 5: Advanced String Formatting, Type Annotations, and Method Overrides in Python 3.12

Objective: In this workshop, you will enhance the user session management application by:

1. **Utilizing advanced f-string syntax** to simplify string formatting with complex expressions.
2. **Leveraging improved error messages** to quickly debug issues.
3. **Using enhanced type annotations for** `**kwargs` to clearly define expected keyword arguments.
4. **Applying the** `@override` **decorator** to ensure that methods in derived classes correctly override those in base classes.

Workshop 5: Advanced String Formatting, Type Annotations, and Method Overrides in Python 3.12

Scenario: The user session management application has grown more complex, and there is a need to:

1. **Dynamically format log messages** with detailed expressions.
2. **Utilize improved error messages** for easier debugging.
3. **Define flexible but specific keyword arguments** for advanced functions.
4. **Ensure accurate method overrides** to maintain consistency in derived classes.

This workshop will guide you through using these new Python 3.12 features to make the application more maintainable and debug-friendly.

Workshop 5: Advanced String Formatting, Type Annotations, and Method Overrides in Python 3.12

Step 1: Advanced f-String Formatting for Log Messages

With Python 3.12, you can now use complex expressions within f-strings, even if they involve the same quotes, multiline expressions, comments, and Unicode escape sequences. This makes log messages more expressive and reduces the need for complex formatting workarounds.

Instructions:

1. **Update log messages** in the `validate_user_access` and `process_access_request` functions using f-strings to:
   - Include dynamic information like user roles, names, and access status.
   - Use multiline f-strings to include explanations or comments within the f-string itself.
2. **Test complex expressions** within f-strings by including expressions like `role.upper()`, string concatenations, and nested f-strings.

Workshop 5: Advanced String Formatting, Type Annotations, and Method Overrides in Python 3.12

Step 2: Improved Error Messages for Debugging

Python 3.12 introduces even more precise error messages that indicate exactly where issues occur, including missing `self` references and module import errors.

Instructions:

1. **Intentionally introduce errors** by:
   - Omitting `self` in a method within a class and observing the error message.
   - Referencing an undeclared module (e.g., `json.loads(...)` without importing `json`).
2. **Observe the error messages** to see how Python 3.12 identifies specific issues, then **correct the errors**.

Workshop 5: Advanced String Formatting, Type Annotations, and Method Overrides in Python 3.12

Step 3: Enhanced Type Annotations for `**kwargs`

Python 3.12 improves the ability to annotate `**kwargs` with specific types, allowing for more precise typing. This is particularly useful in functions that take variable keyword arguments to handle user data or configurations.

Instructions:

1. **Define a function** `log_access_event(**kwargs)` that logs different types of access events, where each keyword argument has a specific type (e.g., `role: str`, `timestamp: str`).
2. **Annotate** `**kwargs` **with** `TypedDict` to specify the types of each keyword argument explicitly.
3. **Test the function** by passing different keyword arguments and verify that each argument matches the specified type.

Workshop 5: Advanced String Formatting, Type Annotations, and Method Overrides in Python 3.12

Step 4: Using the `@override` Decorator

Python 3.12 introduces the `@override` decorator, which helps ensure that methods in derived classes correctly override those in base classes. This is useful for avoiding subtle errors when subclassing and customizing behavior.

Instructions:

1. **Create a `BaseUserSession` class** with a `log_session` method.
2. **Create a derived class `AdminSession`** that overrides `log_session`.
   - Use the `@override` decorator on the overriding method in `AdminSession`.
3. **Test the decorator** by attempting to override a non-existent method, and observe how the `@override` decorator catches this error.