# Workshop Python 3.9, 3.10, 3.11, 3.12

Oxiane INSTITUT

make it clever

Oxiane Institut

# 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: Leveraging Python 3.11 Features for Resilient and Typing-Aware Data Processing

**Scenario**

Participants will process a local sales dataset (sales_data.csv) to calculate metrics like total revenue, monthly trends, and handle various errors. The dataset will simulate common issues, such as malformed rows, missing data, or invalid values, which participants must identify and handle effectively using Python 3.11 features.

**Step 1: Data Loading and Validation with add_note()**

Create a function to read the CSV file line by line, validate each row, and add notes to exceptions for better debugging.

- Function: load_and_validate_data(file_path)
- Reads the CSV file.
- Validates:
- The presence of required fields (OrderID, OrderDate, Quantity, Price).
- Positive values for Quantity and Price.
- Proper date formatting.
- Uses add_note() to attach additional context to validation errors.

# Workshop 4: Leveraging Python 3.11 Features for Resilient and Typing-Aware Data Processing

**Step 2: Asynchronous Revenue Calculation with ExceptionGroup**

Split the validated data into chunks and calculate revenue asynchronously, handling multiple errors using ExceptionGroup and except*.

- Function: calculate_chunk_revenue(chunk)
- Calculates total revenue for a chunk of data.
- Function: calculate_total_revenue(data)
- Divides the data into chunks and processes them asynchronously.
- Raises an ExceptionGroup if multiple chunks fail.

# Workshop 4: Leveraging Python 3.11 Features for Resilient and Typing-Aware Data Processing

**Step 4: Monthly Revenue Aggregation**

Group the processed data by month and calculate total revenue for each month.

- Function: calculate_monthly_revenue(data)
- Iterates through the data, extracts the month from OrderDate, and sums up the revenue.

**Step 5: Directory Handling with pathlib and StrEnum**

- List all .csv files in a directory.
- Use StrEnum to define data categories.

# 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**

Participants will process a local sales dataset (sales_data.csv) to calculate business metrics while exploring advanced features from Python 3.11 and 3.12. Key areas of focus include error handling, type annotations, advanced string formatting, and method overrides.

**Step 1: Advanced String Formatting (Python 3.12)**

Use Python 3.12's new string formatting capabilities, such as variable braces {} for dynamic templates, to generate reports and summaries.

- Function: generate_report(monthly_revenue)
- Produces a formatted string summary of monthly revenues.

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

**Step 2: Method Overrides and Type Consistency (Python 3.12)**

Use method override annotations to enforce type consistency in subclass implementations.

- Class: CustomProcessor
- Extends DataProcessor and overrides filter_by_date.

**Step 3: Flexible Configurations with TypedDict (Python 3.12)**

Use TypedDict to define and enforce structured configurations for dynamic sales report generation.

- Class: ReportConfig
- Specifies the structure of configurations, including fields such as Customer, Shipping, and Discount.
- Uses total=False to allow optional fields.
- Class: Order
- Defines the structure of order data with fields like OrderID, ProductID, Quantity, and Price.
- Function: generate_sales_report
- Accepts an Order and dynamically merges configurations from multiple sources using Python 3.12's enhanced keyword argument unpacking.
- Leverages TypedDict to ensure type safety for both orders and configurations.