Python: Assignment 1- Task Management System
Work Done by : Ilyes REZGUI
Neptun ID: EIJV51

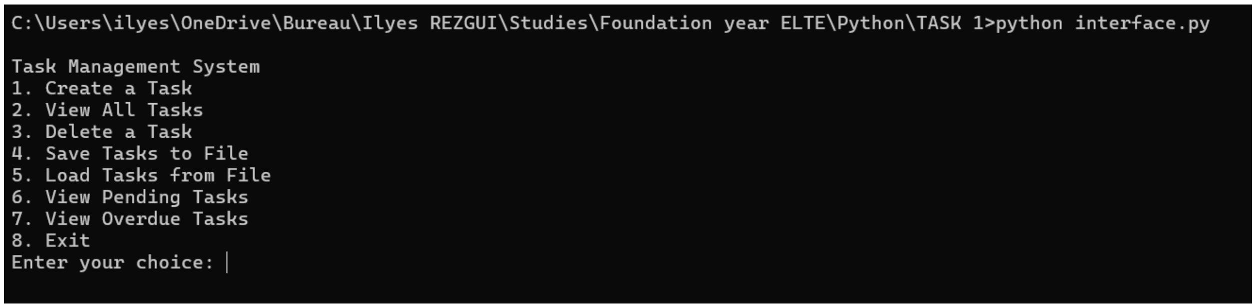
DOCUMENTATION

System descriptions :

The Task Management System is a command-line application designed to help users manage personal and work-related tasks efficiently. Users can create, view, filter, delete, save, and load tasks, providing a good approach to task tracking and organization. This system uses object-oriented design principles to separate different task types and management functionalities, enabling a modular, easy-to-maintain structure.

Setup instructions :

- ☐ 1) go to the directory where the source code is located
- ☐ 2) Open a new command line in there by typing “CMD” in the path bar
- ☐ 3) launch the program by typing “python interface.py”



```
C:\Users\ilyes\OneDrive\Bureau\Ilyes REZGUI\Studies\Foundation year ELTE\Python\TASK 1>python interface.py

Task Management System
1. Create a Task
2. View All Tasks
3. Delete a Task
4. Save Tasks to File
5. Load Tasks from File
6. View Pending Tasks
7. View Overdue Tasks
8. Exit
Enter your choice:
```

Class and method descriptions :

I. TASK.py

This file contains 3 main classes being:

- Task: A general task with attributes like title, due_date, status, and description.
- PersonalTask: A specialized Task with a priority level, specifically for personal tasks.
- WorkTask: A specialized Task intended for work tasks, with support for team members.

Imports

from datetime import datetime, date : For handling date related attributes and objects.

1) Task

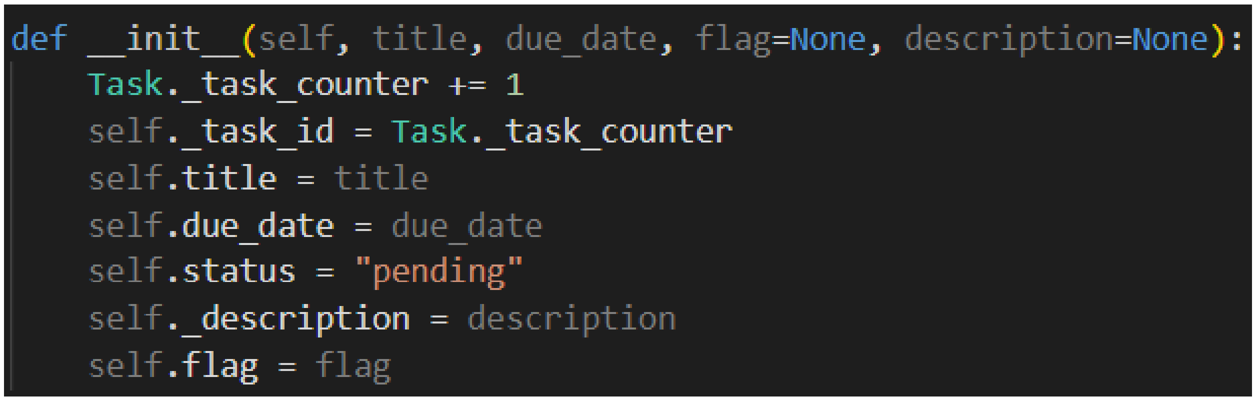
Represents a general task with attributes for the title, due date, status, description, and a flag indicating the task type. Each task has a unique ID generated automatically and that is auto-incremented.

Attributes

- _task_counter (int): Class attribute for auto-incrementing the task ID.
- title (str): Title of the task.
- due_date (date): The due date for the task; must be in YYYY-MM-DD format.
- status (str): Task status, either "pending" (default) or "completed".
- _description (str): Short description of the task, limited to 15 characters and is a protected attribute.
- flag (str): Type of task, and can be"personal" or "work"

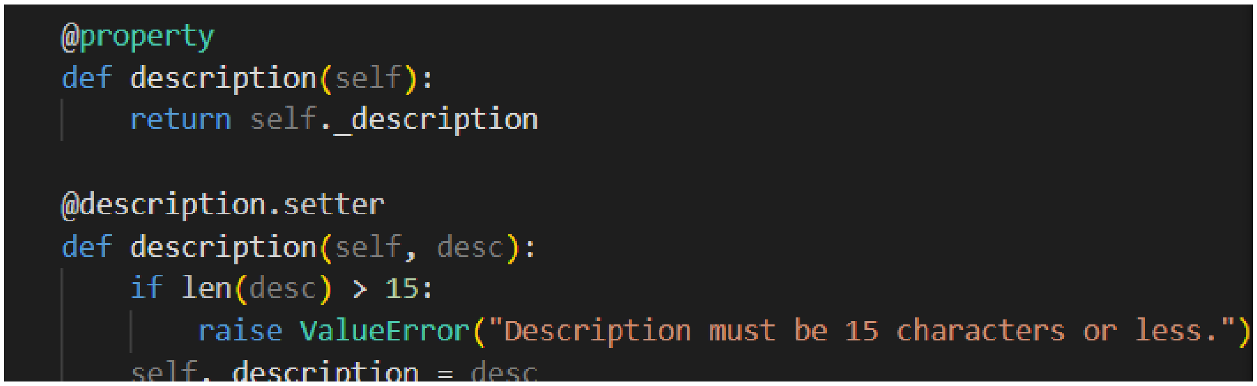
Methods

-
  ```
  def __init__(self, title, due_date, flag, description=None):
  ```
- Constructor. Initializes the task's attributes.



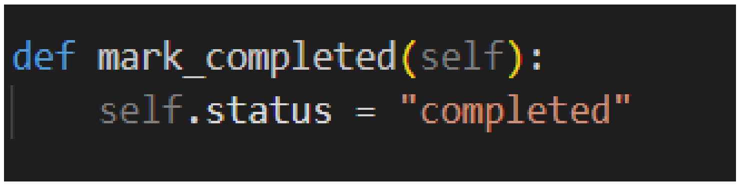
```python
def __init__(self, title, due_date, flag=None, description=None):
    Task._task_counter += 1
    self._task_id = Task._task_counter
    self.title = title
    self.due_date = due_date
    self.status = "pending"
    self._description = description
    self.flag = flag
```

For the property we used two different decorators, The property and description setter. This will allow us to set the value of the description private attribute as if it is an attribute through the use of two methods.



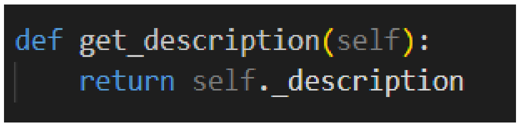
```python
@property
def description(self):
    return self._description

@description.setter
def description(self, desc):
    if len(desc) > 15:
        raise ValueError("Description must be 15 characters or less.")
    self._description = desc
```

- mark_completed(self): Marks the task status as "completed".



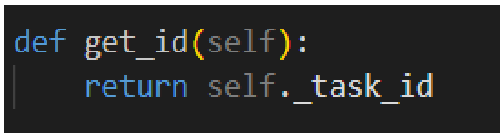
```python
def mark_completed(self):
    self.status = "completed"
```

- get_description(self): Returns the task's description.



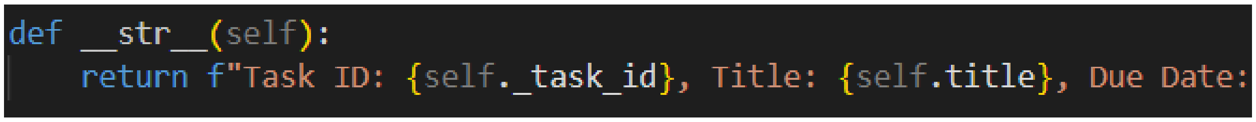
```python
def get_description(self):
    return self._description
```

- get_id(self): Returns the task's unique ID.



```python
def get_id(self):
    return self._task_id
```

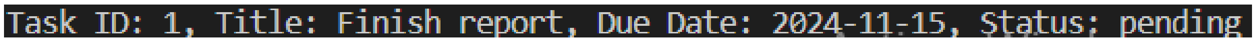
- __str__(self): Returns a string representation of the task.



```python
def __str__(self):
    return f"Task ID: {self._task_id}, Title: {self.title}, Due Date:
```

Example Usage

task = Task(title="Finish report", due_date="2024-11-15", description="Quarterly update")
print(task)
task.mark_completed()
print(task.status)



```
Task ID: 1, Title: Finish report, Due Date: 2024-11-15, Status: pending
```

2) PersonalTask:

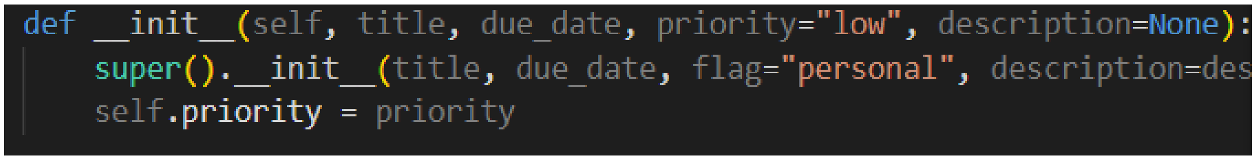
Inherits from Task and represents a personal task with an additional priority attribute (low, medium, or high).

Additional Attribute

- **priority** (str): The priority level for the task (default: "low").

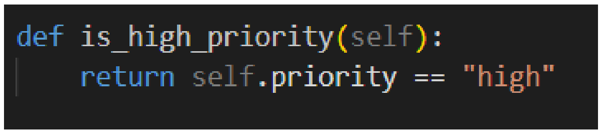
Additional Methods

- __init__(self, title=None, due_date=None, status="pending", description=None, flag="personal", priority="low"): Constructor that takes into consideration the priority attribute as a new attribute.



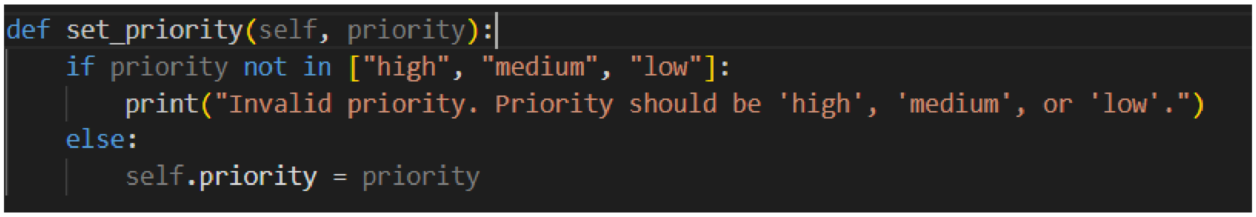
```
def __init__(self, title, due_date, priority="low", description=None):
    super().__init__(title, due_date, flag="personal", description=des
    self.priority = priority
```

- is_high_priority(self): Checks if the task's priority is "high".



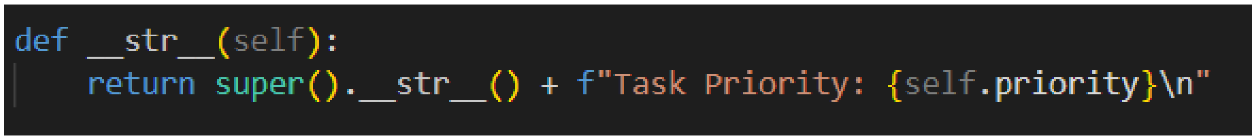
```
def is_high_priority(self):
    return self.priority == "high"
```

- set_priority(self, priority): Sets the task's priority to "high", "medium", or "low". Prints an error if the priority is invalid.



```
def set_priority(self, priority):
    if priority not in ["high", "medium", "low"]:
        print("Invalid priority. Priority should be 'high', 'medium', or 'low'.")
    else:
        self.priority = priority
```

- __str__(self): Returns a string representation of the personal task, including priority.

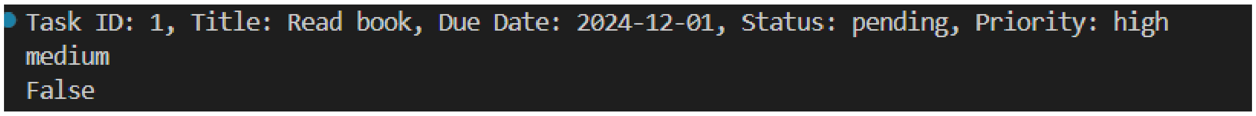


```
def __str__(self):
    return super().__str__() + f"Task Priority: {self.priority}\n"
```

Example Usage

```
# Create a PersonalTask instance
personal_task = PersonalTask(title="Read book", due_date="2024-12-01",
priority="high")
print(personal_task)  # Print personal task details

# Set priority to "medium"
personal_task.set_priority("medium")
print(personal_task.priority)  # Output: "medium"
```

```
# Check if high priority
print(personal_task.is_high_priority())  # Output: False
```



```
Task ID: 1, Title: Read book, Due Date: 2024-12-01, Status: pending, Priority: high
medium
False
```

3) WorkTask:

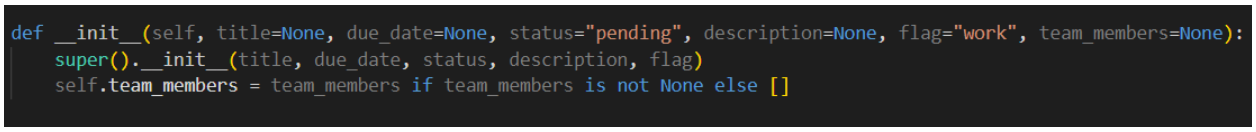
Inherits from Task and represents a work-related task. Includes a list of team members involved in the task.

Additional Attribute

- **team_members** (list of str): Names of team members associated with the task.

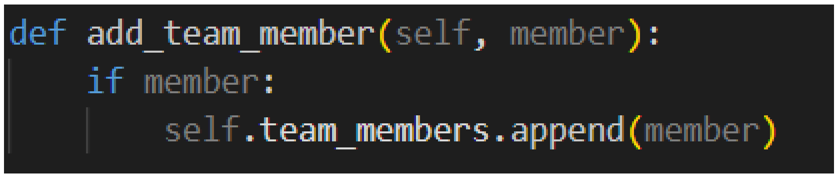
Additional Methods

- __init__(self, title=None, due_date=None, status="pending", description=None, flag="personal", team_members=[]): Constructor that takes into consideration the priority attribute as a new attribute.



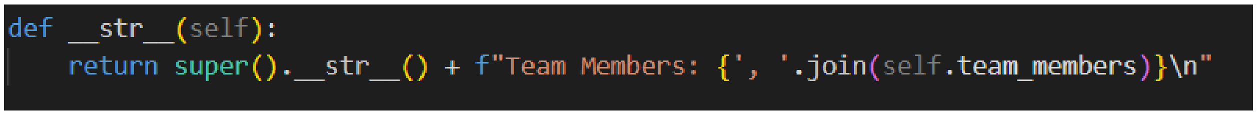
```python
def __init__(self, title=None, due_date=None, status="pending", description=None, flag="work", team_members=None):
    super().__init__(title, due_date, status, description, flag)
    self.team_members = team_members if team_members is not None else []
```

- add_member(self, member): Adds a team member to the task.



```python
def add_team_member(self, member):
    if member:
        self.team_members.append(member)
```

- __str__(self): Returns a string representation of the work task, including team members.

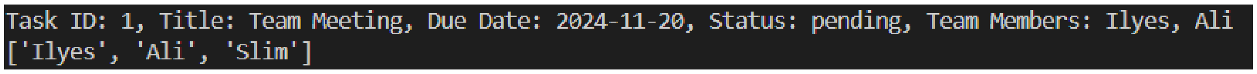


```python
def __str__(self):
    return super().__str__() + f"Team Members: {', '.join(self.team_members)}\n"
```

Example Usage

# Create a WorkTask instance
work_task = WorkTask(title="Team Meeting", due_date="2024-11-20",
team_members=["Ilyes", "Ali"])
print(work_task) # Print work task details

# Add a new team member
work_task.add_team_member("Slim")
print(work_task.team_members) # Output: ['Ilyes', 'Ali', 'Slim']



```
Task ID: 1, Title: Team Meeting, Due Date: 2024-11-20, Status: pending, Team Members: Ilyes, Ali
['Ilyes', 'Ali', 'Slim']
```

II. TASK_manager.py

❖ The task_manager.py file contains the TaskManager class, which provides functionalities for managing Task objects, including creating, listing, deleting, saving, loading, and filtering tasks based on status and due dates.

Imports

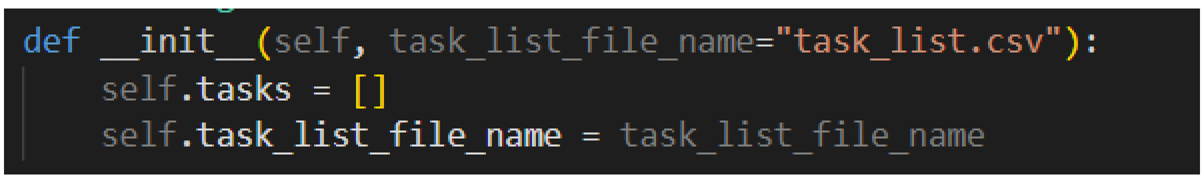
import csv : to process csv files
from datetime import datetime : for the management of dates
from task import Task, PersonalTask, WorkTask : the classes created in the TASK.py file

Attributes

- **tasks** (list of Task): A list containing all task objects.
- **task_list_file_name** (str): The name of the CSV file where tasks are saved and loaded from (task_list.csv by default).

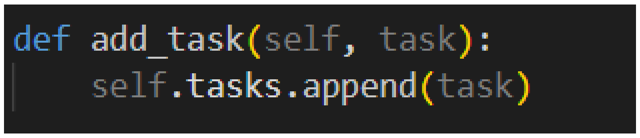
Methods

- **__init__(self) :**Initializes a new TaskManager instance with an empty task list and sets the default file name for saving/loading tasks.



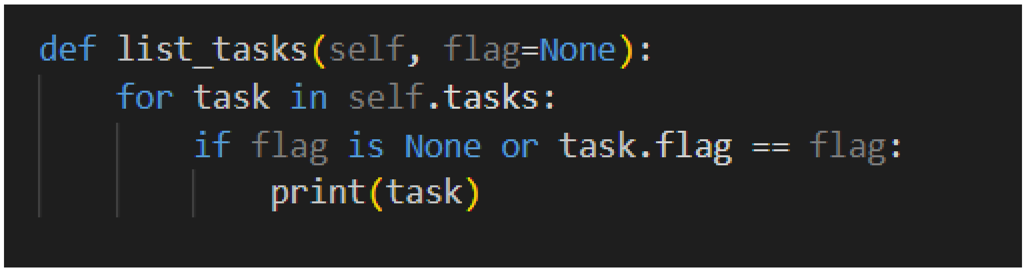
```python
def __init__(self, task_list_file_name="task_list.csv"):
    self.tasks = []
    self.task_list_file_name = task_list_file_name
```

- **add_task(self, task):** Adds a task to the tasks list.
    - Task could be a PersonalTask, or WorkTask instance to add.



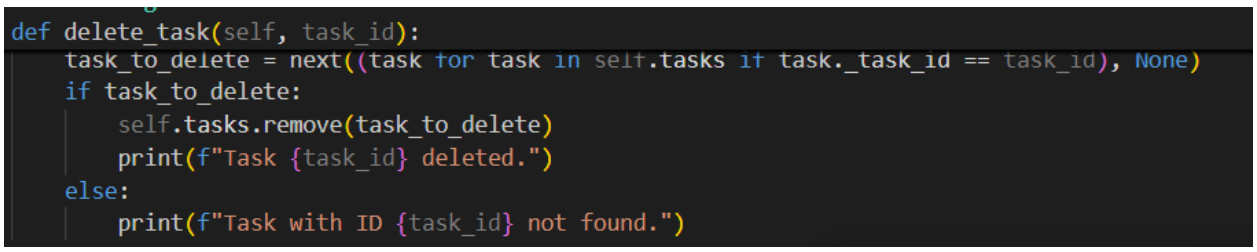
```python
def add_task(self, task):
    self.tasks.append(task)
```

- **list_tasks(self, flag=None) :**Lists tasks. If a flag (e.g., "personal" or "work") is specified, only tasks matching that flag are returned, else returns the tasks.
    - flag is the type of task to filter by (either "personal" or "work").



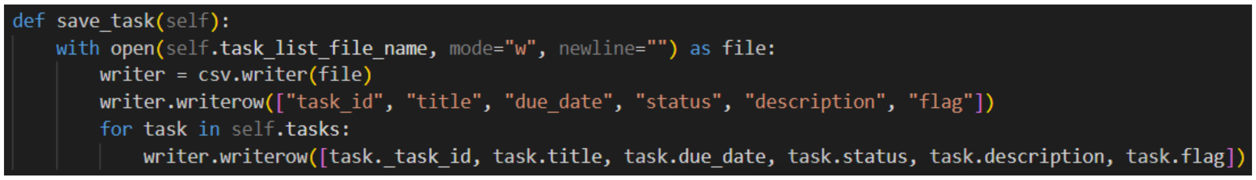
```python
def list_tasks(self, flag=None):
    for task in self.tasks:
        if flag is None or task.flag == flag:
            print(task)
```

- **delete_task(self, task_id) :**Deletes a task with the specified task_id from the tasks list.
    - task_id (int): The ID of the task to delete.



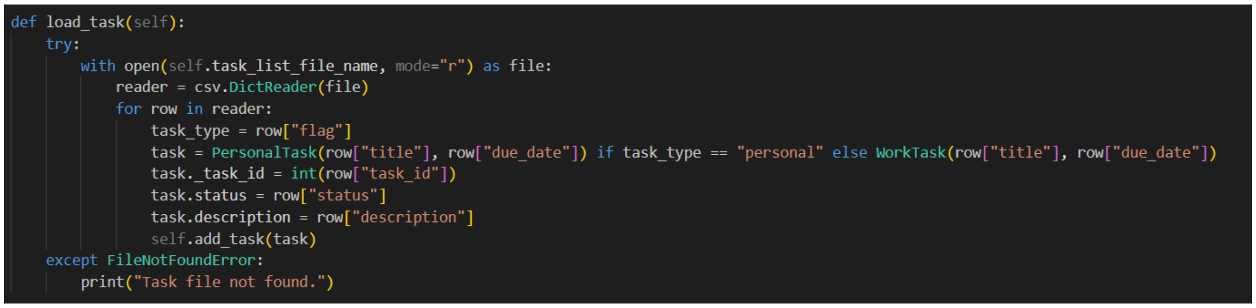
```python
def delete_task(self, task_id):
    task_to_delete = next((task for task in self.tasks if task._task_id == task_id), None)
    if task_to_delete:
        self.tasks.remove(task_to_delete)
        print(f"Task {task_id} deleted.")
    else:
        print(f"Task with ID {task_id} not found.")
```

- **save_task(self) :** Saves all tasks to a CSV file (task_list.csv by default). Each task is saved with its attributes in a comma-separated format.. Overwrites task_list.csv each time it's called.



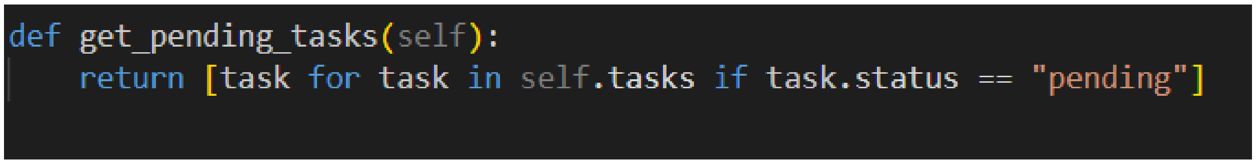
```python
def save_task(self):
    with open(self.task_list_file_name, mode="w", newline="") as file:
        writer = csv.writer(file)
        writer.writerow(["task_id", "title", "due_date", "status", "description", "flag"])
        for task in self.tasks:
            writer.writerow([task._task_id, task.title, task.due_date, task.status, task.description, task.flag])
```

- **load_task(self) :** Loads tasks from the CSV file into the tasks list. If the file does not exist, prints a message indicating that no saved tasks were found.



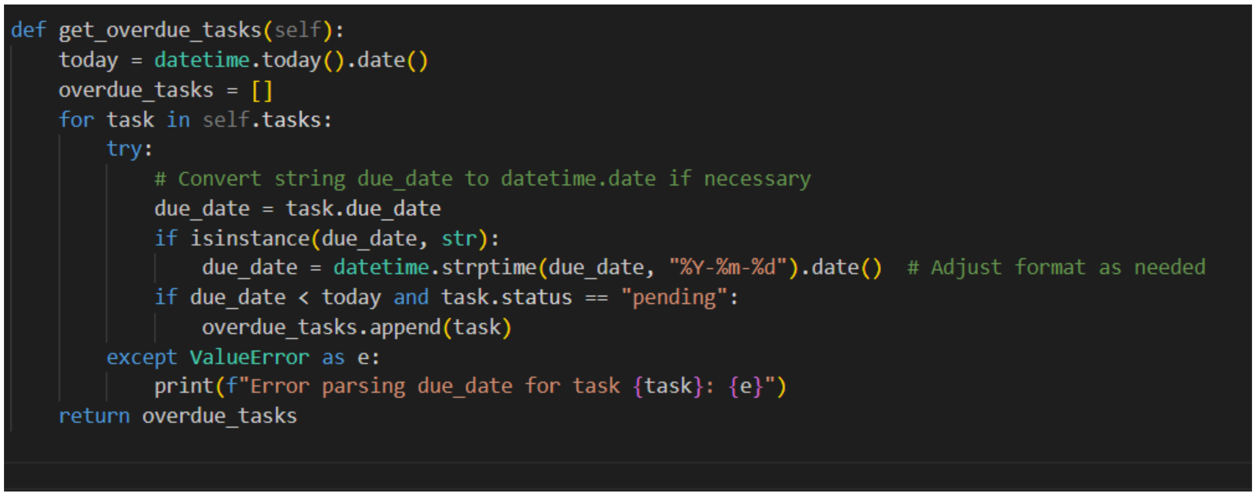
```python
def load_task(self):
    try:
        with open(self.task_list_file_name, mode="r") as file:
            reader = csv.DictReader(file)
            for row in reader:
                task_type = row["flag"]
                task = PersonalTask(row["title"], row["due_date"]) if task_type == "personal" else WorkTask(row["title"], row["due_date"])
                task._task_id = int(row["task_id"])
                task.status = row["status"]
                task.description = row["description"]
                self.add_task(task)
    except FileNotFoundError:
        print("Task file not found.")
```

- **get_pending_tasks(self) :** Returns a list of tasks that have a status of "pending"



```python
def get_pending_tasks(self):
    return [task for task in self.tasks if task.status == "pending"]
```

- **get_overdue_tasks(self):**Returns a list of tasks that have a due date earlier than today's date (overdue tasks).



```python
def get_overdue_tasks(self):
    today = datetime.today().date()
    overdue_tasks = []
    for task in self.tasks:
        try:
            # Convert string due_date to datetime.date if necessary
            due_date = task.due_date
            if isinstance(due_date, str):
                due_date = datetime.strptime(due_date, "%Y-%m-%d").date()  # Adjust format as needed
            if due_date < today and task.status == "pending":
                overdue_tasks.append(task)
        except ValueError as e:
            print(f"Error parsing due_date for task {task}: {e}")
    return overdue_tasks
```

Example Usage

```python
# Instantiate the TaskManager
manager = TaskManager()
# Add a Task
```

```
task1 = Task(title="Complete project", due_date="2024-11-20",
description="Work on project")
# Add a PersonalTask
personal_task = PersonalTask(title="Buy groceries", due_date="2024-11-19",
priority="high")
manager.add_task(personal_task)
# Add a WorkTask
work_task = WorkTask(title="Team meeting", due_date="2024-11-21",
team_members=["Alice", "Bob"])
manager.add_task(work_task)
# List all tasks
manager.list_tasks()
# List only personal/work tasks
manager.list_tasks("personal")
manager.list_tasks("work")
```

III. Interface.py

❖ interface.py provides a simple command-line interface (CLI) for interacting with the TaskManager class and managing tasks. Here's a breakdown of the key features and what each section does:

Key Features:

1. **`display_menu()`**: Shows a list of options for the user to choose from, including creating, viewing, deleting, saving, and loading tasks.
2. **`create_task()`**: Guides the user through creating a new task, allowing them to specify whether it's a personal or work task and prompting for the necessary details accordingly.
3. **`main()`**: The core loop where the program runs. It repeatedly displays the menu, processes the user’s selection, and executes the corresponding actions (such as creating tasks, viewing tasks, deleting tasks, saving/loading tasks, etc.).

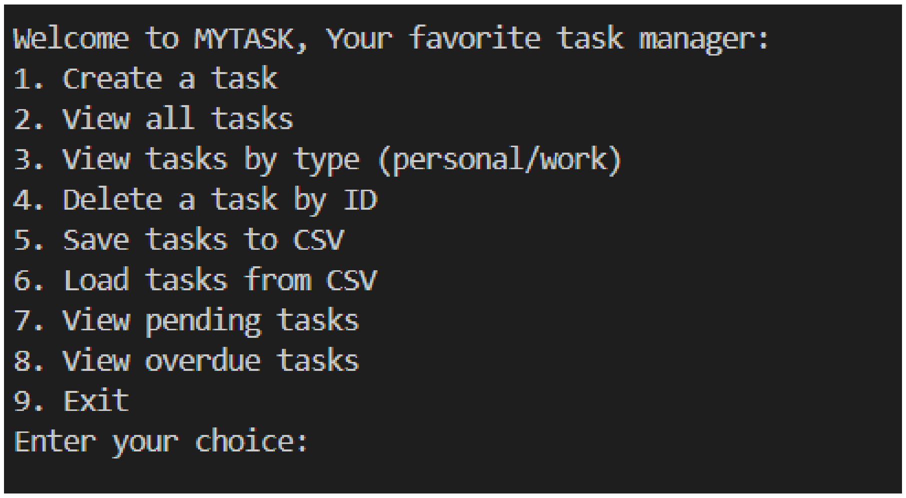
How It Works:

At program start, the `main()` function initializes the `TaskManager` and continuously displays the menu in a loop. Based on the user's selection, it performs one of the following actions:

- Creates a new task by calling `create_task()` and adds it to the `TaskManager`.
- Views all tasks, filters tasks by type, or displays pending/overdue tasks by calling the appropriate `TaskManager` methods.
- Deletes a task by its ID.
- Saves tasks to a CSV file or loads tasks from a CSV file.
- Exits the program if the user selects the "Exit" option.

Example Interaction:

Here's an example of what the interaction might look like when running the program:



```
Welcome to MYTASK, Your favorite task manager:
1. Create a task
2. View all tasks
3. View tasks by type (personal/work)
4. Delete a task by ID
5. Save tasks to CSV
6. Load tasks from CSV
7. View pending tasks
8. View overdue tasks
9. Exit
Enter your choice:
```

#Two possible executions are :

Enter your choice: 1
Select task type:
1. Personal Task
2. Work Task
Enter choice (1 or 2): 1
Enter task title: Buy groceries
Enter due date (YYYY-MM-DD): 2024-11-19
Enter description (max 15 characters): Buy food

```
Enter priority (low, medium, high): high
Task added successfully.

Enter your choice: 2
Task Id: 1
Task Name: Buy groceries
Task Due Date: 2024-11-19
Task Status: pending
Task Description: Buy food
Task Flag: personal
Task Priority: high

Enter your choice: 9
Exiting Task Manager. Good bye!
```

Explanation of Code:

1. **create_task()**:
    - The function prompts the user to enter task details, including the title, due date, and description. It also requests additional information specific to the type of task: priority for personal tasks or team members for work tasks. Based on the user's input, the function then creates and returns the corresponding task.
2. **main()**:
    - **Option 1:** Use `create_task()` to create and add a new task to the task manager.
    - **Option 2:** Use `list_tasks()` to display all tasks.
    - **Option 3:** Use `list_tasks(flag)` to filter and display tasks by type (either personal or work).
    - **Option 4:** Delete a task by its ID with `delete_task(task_id)`.
    - **Option 5:** Save tasks to a CSV file using `save_task()`.
    - **Option 6:** Load tasks from a CSV file with `load_task()`.
    - **Option 7:** View pending tasks using `get_pending_tasks()`.
    - **Option 8:** View overdue tasks with `get_overdue_tasks()`.
    - **Option 9:** Exit the program.

Error handling :

The program includes several mechanisms to handle invalid data inputs and provides user feedback to guide them.

1. Date Input Validation for Tasks

- When creating a task, the user is prompted to enter a due date in the format YYYY-MM-DD. The program checks that this input is valid by attempting to convert it to a datetime.date object using datetime.strptime().
- Invalid Date Format: If the user enters an incorrect date format, the program catches this error, prints the message "Invalid date format. Please enter the date as YYYY-MM-DD.", and sets the due_date attribute to None.

2. Title and Description Validation

- When creating a task, if no title is provided, the program prints the message "Please consider entering a title". This encourages the user to input a title, although it allows the creation of tasks without one.
- Description Length Check: The description attribute is limited to a maximum of 15 characters. If the user tries to input a description longer than this, the program sets the description to None, indicating that the input did not meet the requirements.
- Manual Description Update: When setting a description manually using set_description(), if the description is too long, the program raises a ValueError with the message "The description cannot be longer than 15 characters". This informs the user of the exact problem and enforces the description limit.

3. Priority Validation for Personal Tasks

- For PersonalTask objects, the priority attribute must be one of "high", "medium", or "low". If the user inputs an invalid priority, the program displays the message "Invalid priority. Priority should be 'high', 'medium', or 'low'." and keeps the current priority unchanged. This prevents the user from setting an unsupported priority level.

4. Deleting Tasks by ID

- When the user tries to delete a task by ID, the program searches the task list for a task matching that ID. If no task with the specified ID is found, the program prints "Task not found." to let the user know that the deletion was unsuccessful. This prevents silent failures and ensures the user is aware of the task list status.

5. File Operations (Save and Load Tasks)

- Loading Tasks: During task loading, if the CSV file (task_list.csv) is missing, a FileNotFoundError is caught, and the program displays the message "No saved tasks found.". This alerts the user that no data was loaded and prevents the program from crashing due to a missing file.

6. User Menu Choices

- In interface.py, the program prompts the user to make selections from a menu of options. If the user enters a choice that is not recognized, the program prints "Invalid choice. Please try again.". This ensures that the user is aware of their mistake and can reattempt without disrupting the program’s flow.