

**The Superior University**

**📌 Project Title**

FCFS Printing Task Scheduler using Python.

**👥 Group Members**

Team members with roll numbers:

* Muhammad Usama (SU92-BSSEM-F23-049)
* Muhammad Saad Zahid (SU92-BSSEM-F23-030)
* Muhammad Noman (SU92-BSSEM-F23-016)

**📂 GitHub Repository**

**GitHub Repository Link of All Group Members with their names:**

Links:

<https://github.com/noman763/Printing-Task-Scheduler-Python> Muhammad Noman  
<https://github.com/saadzahid34/saad.git> Muhammad Saad Zahid

<https://github.com/muhammadusama792/Printing-Task-Scheduler-Python-.git>

Muhammad Usama

**🔧 Scheduling Algorithm Implemented**

✅ Tick the scheduling algorithm your group implemented:

* FCFS (First Come First Serve)



* SJF (Shortest Job First – Non-Preemptive)



* SJF (Preemptive)



* Round Robin



**📄 Project Description**

**What Problem Does This Project Solve?**

This project simulates a **Printing Task Scheduler**, allowing users to manage print jobs using a **simple queue-based system**. It mimics a real-world printer queue, where documents are added, processed, and tracked based on their **order of arrival**.

**In simple words**, it works like a print shop or office printer: documents are printed in the order they are submitted. Users can add tasks, edit them, track progress, and clear completed jobs — offering a clear view of the entire print process.

This project helps users understand how basic **task queues** and **status tracking** work in scheduling systems.

**What Inputs Are Required?**

* **Document Name**: A label or name for the document (e.g., Report1, Invoice.pdf)
* **Pages**: The number of pages the document has (used to simulate print time)

**Note**: Each document is automatically assigned a "Pending" status upon creation.

**What Outputs Are Generated?**

The program provides:

* A list of **Pending Tasks** with document names, page counts, and current status
* A list of **Completed Tasks** after processing
* **Simulated printing** delay based on the number of pages (e.g., longer documents take more time)
* **Status updates**: from "Pending" to "Completed" after a task is printed
* **Task count summary**: showing how many tasks are pending and completed
* **Task search results**: find tasks by document name
* **Editable tasks**: change the number of pages before printing
* **Menu-based user interface** for easy navigation

**How the System Is Implemented**

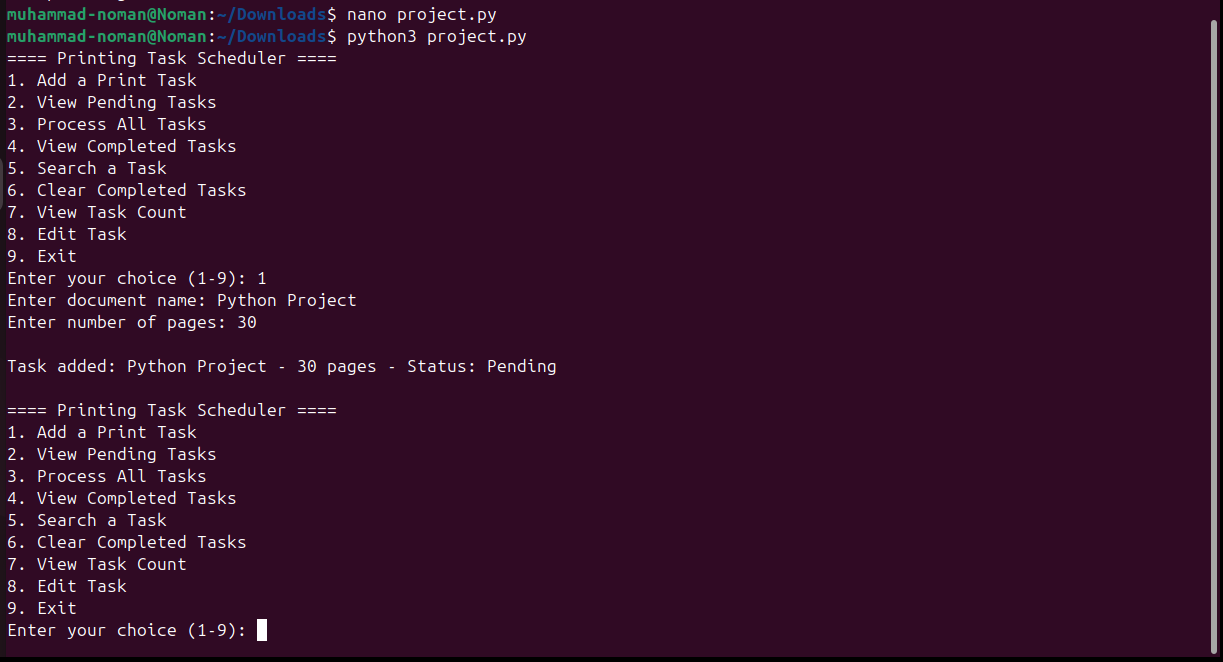
When users interact with the system through a menu, they can perform a range of actions to manage print jobs.

Here’s how the main operations work:

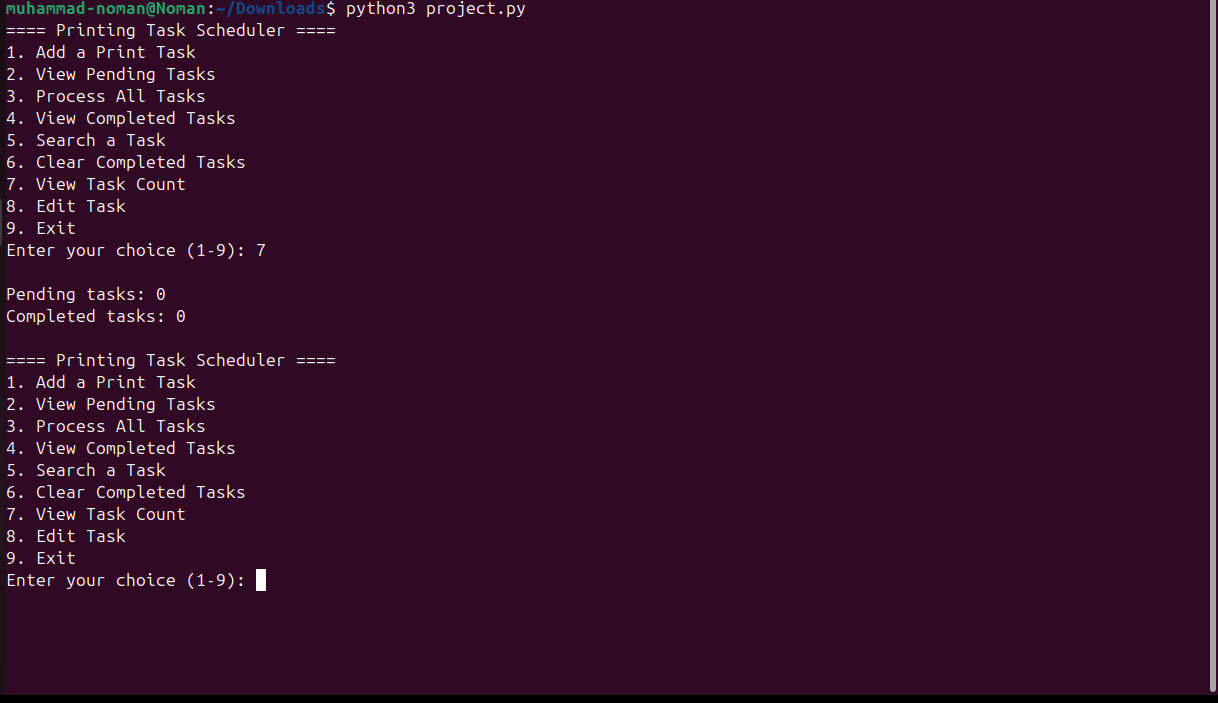
1. **Adding Tasks**:  
   When a task is added, it's stored in the task queue with its name, page count, and a "Pending" status.
2. **Viewing Tasks**:  
   Users can view all pending or completed tasks with their details.
3. **Processing Tasks**:  
   The program processes tasks in **First-In-First-Out (FIFO)** order — the oldest task is printed first. It simulates delay using time.sleep() based on the number of pages (e.g., pages × 0.2 seconds per page).
4. **Completing Tasks**:  
   Once a task is processed, its status is changed to "Completed" and moved to the completed\_tasks list.
5. **Searching for a Task**:  
   Users can search for a document by name across both pending and completed tasks.
6. **Editing a Task**:  
   Before printing, users can update the number of pages in any pending task.
7. **Clearing Completed Tasks**:  
   Users can clear all finished jobs from memory.
8. **Viewing Task Counts**:  
   A quick summary of pending and completed task totals is available.
9. **Menu Navigation**:  
   A menu-driven interface allows users to easily select and perform desired actions.

**Output Screenshots**

**1st sample Output**



**2nd sample Output**



**Code Structure & Explanation**

**a. Functions or Logic Used**

**1. Task Representation (Dictionary-Based)**  
Each task in the printing queue is stored as a dictionary containing the document name, number of pages, and its status (either "Pending" or "Completed"). Two lists are used to manage the state of tasks — one for pending tasks and another for completed ones.

**2. Task Processing Logic**  
Tasks are processed in the order they are added, following a First-In, First-Out (FIFO) pattern. When a task is processed, it is removed from the pending queue, printed (simulated using time.sleep() based on page count), marked as completed, and added to the completed task list. The logic also supports additional features such as editing the page count of pending tasks, searching tasks by name, clearing completed jobs, and showing a count of all tasks.

**3. Output and Task Management**  
The program provides a simple menu-driven interface. Users can view pending and completed tasks, search for a task by name, or see task counts. Outputs are printed in a clean and readable format, showing task details and system status updates after each action. Input validation is included to ensure the application does not crash on incorrect entries.

**4. User Interface**  
The interface is entirely text-based and operated through the terminal. Users interact via numbered options, making the system easy to navigate and understand without needing a graphical interface.

**b. Core Logic of the Task Scheduler**

The project simulates how a real-world print queue might operate. Users are prompted to enter a document name and the number of pages when adding a new task. These tasks are stored in a queue and processed in the exact order of entry.

When the user selects the option to process tasks, the program simulates printing each document one by one. The delay during printing is proportional to the number of pages (0.2 seconds per page). Once a task is completed, it is updated with a status of "Completed" and transferred to the appropriate list.

In addition to processing, users can edit any task that hasn’t yet been printed, search for a task by name in either list, and clear all completed tasks. Task statistics such as the number of pending and completed jobs are also available.

**c. External Libraries Used**

This project uses the time module from Python's standard library to simulate printing delays. The sleep() function is applied to create a realistic pause during task processing, based on the size of the task (number of pages). No additional or third-party libraries are required, making the system lightweight and portable.

**Performance Metrics**

| **Metric** |  |  |  |  |  | **Description** |
| --- | --- | --- | --- | --- | --- | --- |
| Task Execution Order |  |  |  |  |  | First-In, First-Out (FIFO) |
| Average Processing Time |  |  |  |  |  | Depends on total pages and number of tasks |
| Time Quantum |  |  |  |  |  | Not used — FCFS logic, not time-sliced |

**Challenges Faced**

**Incorrect Delay Handling**  
One early issue was balancing the simulated delay. Without proper scaling, longer documents could slow down the program too much. The issue was resolved by adjusting the sleep time to a reasonable 0.2 seconds per page, providing both realism and responsiveness.

**User Input Errors**  
Initially, entering invalid input (such as non-integer values for pages) would crash the program. This was fixed by implementing input validation using try-except blocks, ensuring the system prompts users to re-enter correct data without breaking.

**Task Editing Management**  
Another challenge was allowing users to edit tasks without disrupting the queue. The solution involved displaying all pending tasks with assigned numbers, so users could select and edit them easily while preserving order.

**Summary**

This printing task scheduler is a practical simulation of how print jobs are managed in a queue. It uses clear logic, user input handling, and terminal-based interaction to replicate the experience. By focusing on queue processing, task management, and real-time simulation through delay, the project reinforces key programming concepts such as data structures, condition handling, and user experience. The end result is a functional and user-friendly Python program ideal for understanding scheduling workflows.