TIMETABLE
MONDAY | TUESDAY | WEDNESDAY | THURSDAY | FRIDAY | SATURDAY | SUNDAY

TIMETABLE AND ROOM ALLOTMENT MANAGEMENT SYSTEM

INTRODUCTION

The Timetable and Room Allotment Management System is a Python-based application designed to efficiently manage the scheduling of classes while avoiding conflicts such as overlapping classes, instructor availability, and room clashes. It ensures a well-organized schedule for institutions, helping in better resource management.

```
class Timetable:
    def _init_(self):
        self.schedule = []

    def add_class(self, course, instructor, batch, section, day,
start_time, end_time, room):
        def time_to_minutes(time_str):
            hours, minutes = map(int, time_str.split(":"))
            return hours * 60 + minutes

        start_time_mins = time_to_minutes(start_time)
        end_time_mins = time_to_minutes(end_time)

        day_start = 540 # 09:00 AM
        day_end = 1050  # 05:30 PM

         lunch_break_start = 720  # 12:00 PM
        lunch_break_end = 780    # 01:00 PM
```

The Timetable **class** is the main structure of the program.

The __init__ function is the **constructor** that runs when an object of Timetable is created.

self.schedule = [] initializes an **empty list** where all class schedules will be stored.

This function **accepts class details** from the user and tries to schedule it.**Helper function** to convert time from "HH:MM" format to **total minutes**.

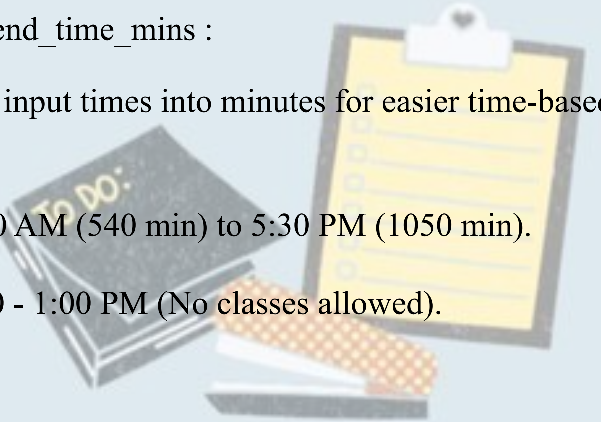
09:00" → 9 * 60 + 0 = 540 minutes.

start_time_mins or end_time_mins :

- Converts user input times into minutes for easier time-based calculations.

Working hours: 9:00 AM (540 min) to 5:30 PM (1050 min).

Lunch breaks: 12:00 - 1:00 PM (No classes allowed).



```
if start_time_mins < day_start or end_time_mins > day_end:
    print(f"Invalid time: {course} for {batch}-{section} must be scheduled between 09:00 and 17:30.")
    return
```

```
if not (end_time_mins <= lunch_break_start or start_time_mins >= lunch_break_end):
        print(f"Lunch break conflict: {course} for {batch}-{section} on {day} cannot be scheduled between 12:00-1:00.")
        return
```

if start_time_mins < day_start

- Checks if the class **starts before** 9:00 AM (540 minutes).
- If start_time_mins is **less than** day_start, it means the class is starting too early.

or end_time_mins > day_end

- Checks if the class **ends after** 5:30 PM (1050 minutes).
- If end_time_mins is **greater than** day_end, it means the class is ending too late.

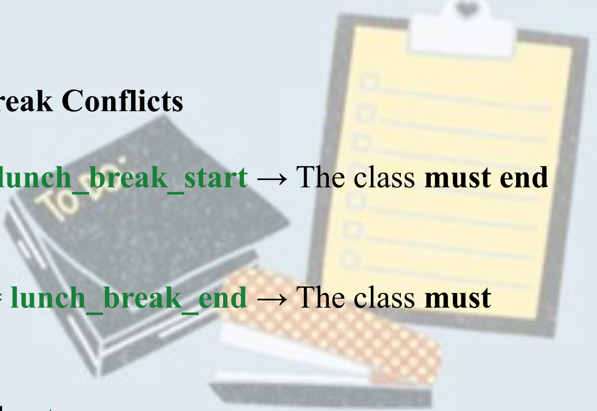
return stops the function, so the class **won't be added** to the timetable.

Checking Lunch Break Conflicts

end_time_mins <= lunch_break_start → The class **must end before 12:00 PM**

start_time_mins >= lunch_break_end → The class **must start after 1:00 PM**

Stops execution with return.



```
for entry in self.schedule:
    if entry['day'] == day:
        entry_start = time_to_minutes(entry['start_time'])
        entry_end = time_to_minutes(entry['end_time'])
```

self.schedule is a **list** that stores all scheduled classes.

entry represents one scheduled class at a time

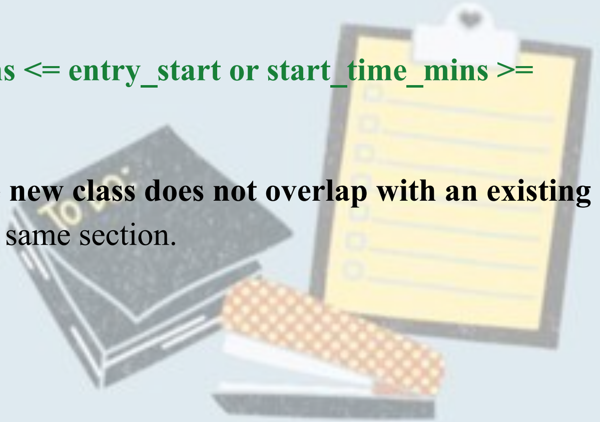
entry['day'] is the **day** of the already scheduled class.day is the **day** of the **new** class being added.

```
if entry['batch'] == batch and entry['section'] == section
and not (end_time_mins <= entry_start or
start_time_mins >= entry_end):
    print(f"Conflict: {batch}-{section} already has another
class scheduled during this time on {day}.")
    return
```

entry['batch'] == batch → Checks if the **existing** class and the **new** class are for the **same batch**.
entry['section'] == section → Checks if both classes are for the **same section** within the batch.

not (end_time_mins <= entry_start or start_time_mins >= entry_end)

- **Ensures the new class does not overlap with an existing class** for the same section.



```
if entry['room'] == room and not (end_time_mins <=
entry_start or start_time_mins >= entry_end):
    print(f"Sorry..! The room {room} is not available for
{course} on {day}. Please select another room.")
    return
```

```
if entry['instructor'] == instructor and not
(end_time_mins <= entry_start or start_time_mins >=
entry_end):
    print(f"Instructor {instructor} is not available for
{course} on {day}.")
    return
```

entry['room'] == room → Checks if the **existing class** and the **new class** are in the **same room**.

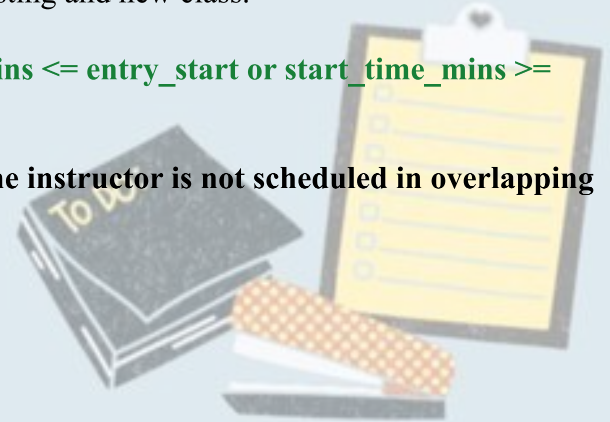
If **same room**, we check if the times overlap using:

- **end_time_mins <= entry_start** → The new class must **end before** the existing class starts.
- **start_time_mins >= entry_end** → The new class must **start after** the existing class ends.

entry['instructor'] == instructor → Checks if the **same instructor** is teaching the existing and new class.

not (end_time_mins <= entry_start or start_time_mins >= entry_end)

- **Ensures the instructor is not scheduled in overlapping times.**

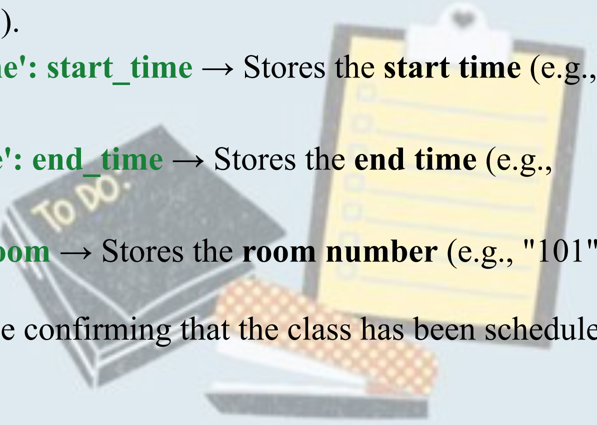


```
self.schedule.append({
    'course': course,
    'instructor': instructor,
    'batch': batch,
    'section': section,
    'day': day,
    'start_time': start_time,
    'end_time': end_time,
    'room': room
})
print(f"Class {course) scheduled for (batch}-{section) in
Room {room} on {day} from {start_time} to {end_time).")
```

self.schedule.append({...}) → This adds a new class to the **schedule list**.
Inside { ... }, we have **key-value pairs** representing the details of the class:

- **'course': course** → Stores the **course name** (e.g., "Math").
- **'instructor': instructor** → Stores the **instructor's name** (e.g., "Dr. Smith").
- **'batch': batch** → Stores the **batch name** (e.g., "B.Tech CSE").
- **'section': section** → Stores the **section** (e.g., "A").
- **'day': day** → Stores the **day of the class** (e.g., "Monday").
- **'start_time': start_time** → Stores the **start time** (e.g., "10:00").
- **'end_time': end_time** → Stores the **end time** (e.g., "11:00").
- **'room': room** → Stores the **room number** (e.g., "101").

It prints a message confirming that the class has been scheduled successfully.



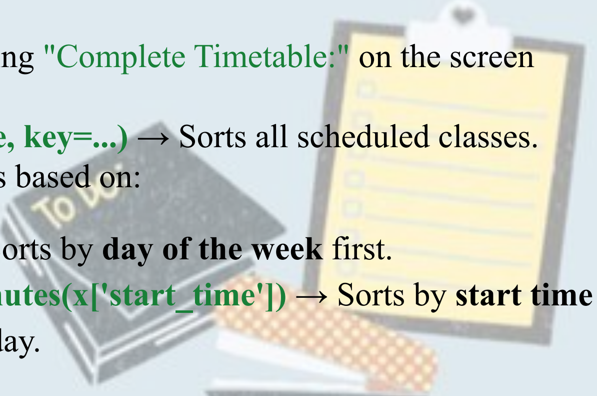
```
def display_schedule(self):

def time_to_minutes(time_str): hours, minutes =
map(int, time_str.split(":"))

return hours * 60 + minutes

print("\nComplete Timetable:")
for entry in sorted(self.schedule, key=lambda x: (x['day'],

time_to_minutes(x['start_time']))):

print(f"(entry['course']}| {entry['instructor']} |
{entry['batch']}-{entry['section']} | {entry['day']} |
{entry['start_time']} {entry['end_time']} | Room
{entry['room']}")
```

Displays all scheduled classes in a **sorted order** (by day and start time).

Converts a time string (e.g., "10:30") into minutes (from 00:00 midnight).
Breaking it down:

1. **time_str.split(":")** → Splits the input "10:30" into ["10", "30"].
2. **map(int, time_str.split(":"))** → Converts "10" to 10 and "30" to 30.
3. **hours * 60 + minutes** → Converts **hours into total minutes**:
   - 10 * 60 + 30 = 600 + 30 = 630 minutes.

This prints the heading "Complete Timetable:" on the screen



sorted(self.schedule, key=...) → Sorts all scheduled classes.
The **sorting order** is based on:

1. **x['day']** → Sorts by **day of the week** first.
2. **time_to_minutes(x['start_time'])** → Sorts by **start time** within each day.

Prints the **details of each class** in a structured format.

```
def get_user_input(self):
while True:
course = input("Enter course name (or 'exit' to stop): ")
if course.lower() == 'exit': break
instructor = input("Enter instructor name: ")
batch = input("Enter batch
name: ")
section = input("Enter section
name: ")
day = input("Enter day: ")
start_time = input("Enter start time (HH:MM): ")
end_time = input("Enter end time (HH:MM): ")
room = input("Enter room number: ")
self.add_class(course, instructor, batch, section, day, start_time,
end_time, room)

timetable = Timetable()
timetable.get_user_input()
timetable.display_schedule()
```

Defines a function that **asks the user for input** to add a new class.

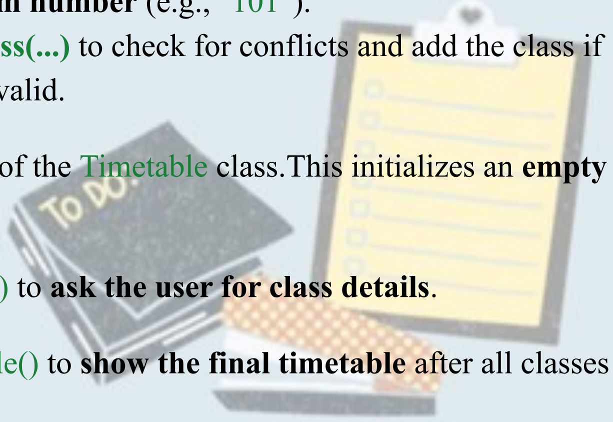
input("Enter course name...") → Asks the user to enter the course name.
if course.lower() == 'exit': break → If the user types "exit", the loop stops.

- Takes the **instructor's name** as input (e.g., "Dr. Smith").
- Takes the **batch name** as input (e.g., "B.Tech CSE").
- Takes the **section name** (e.g., "A")
- Takes the **day** when the class will be scheduled (e.g., "Monday").
- Takes the **start time** (e.g., "10:00") and **end time** (e.g., "11:00") of the class.
- Takes the **room number** (e.g., "101").
- Calls **add_class(...)** to check for conflicts and add the class if everything is valid.

Creates an instance of the Timetable class.This initializes an **empty** schedule.

Calls get_user_input() to **ask the user for class details**.

Calls display_schedule() to **show the final timetable** after all classes are added.



CONCLUSION:

The Timetable and Room Allotment Management System provides an efficient solution for scheduling classes in an educational institution. It ensures proper room allocation, avoids instructor overload, and prevents timetable conflicts. The system is easily extendable and can be further enhanced with additional features such as GUI-based user input, database integration, and report generation.

THANK YOU