Seating Allocation Optimizer: Code Documentation

1. Overview

This document provides a detailed explanation of the Python code within the Mathwizzathon_Final.ipynb Jupyter Notebook. The notebook's primary function is to solve a complex seating allocation problem for a large organization.

The Goal: To create a weekly seating schedule for members across subteams, using available seats. The schedule must adhere to several rules:

- Each subteam must be in the office for at least required number of (currently = 3) days a week.
- Seating bay capacity limits cannot be exceeded.
- Splitting teams across multiple seating areas should be minimized.
- The overall use of office space should be efficient.

To achieve this, the code employs a primary optimization method called **Mixed-Integer Linear Programming (MILP)** and a secondary **Greedy Heuristic** algorithm as a fallback.

2. Step-by-Step Code Explanation

Cell 1: Importing Libraries

```
import pulp
import pandas as pd
from collections import defaultdict
import math
```

This first cell imports the necessary Python libraries:

- pulp: The core library used for building and solving the MILP optimization model. It provides the tools to define variables, constraints, and objectives.
- pandas: Used for data manipulation. The subteam and seating bay information is stored and managed in pandas DataFrames, which are like powerful spreadsheets.
- collections.defaultdict: A specialized dictionary that simplifies the process of grouping and counting things, which is useful when building the final schedule.
- math: A standard Python library for mathematical functions.

Cell 2 & 3: Data Setup and Initial Analysis

```
# 1) DATA (your dataset)
```

```
DAYS = ["Mon", "Tue", "Wed", "Thu", "Fri"]

# (subteams_data and bays_data are defined here)

subteams_df = pd.DataFrame(...)
bays_df = pd.DataFrame(...)

# ... feasibility check ...
```

This section sets up the foundational data for the problem.

- **DAYS**: A simple list defining the working days for the schedule.
- **subteams_data**: A list of tuples, where each tuple contains the details for a subteam: parent team, subteam name, shift time, and the number of members.
- **bays_data**: A similar list detailing each seating bay: the associated team, the bay's name, and its seating capacity.
- **Creating DataFrames**: The raw data lists are converted into pandas DataFrames (subteams_df, bays_df). This format makes it much easier to access and work with the data. For example, subteams_df.loc['C4', 'Members'] would easily retrieve the number of members in subteam 'C4'.
- **Feasibility Check**: The code calculates the total "seat-days" available per week (Total Seats × 5 Days) and compares it to the "seat-days" required (Total Members × 3 Days). This is a quick check to see if a solution is mathematically possible on an aggregate level.

The third cell simply uses display() to print these DataFrames, allowing the user to verify that the data has been loaded correctly.

Cell 4: Building the MILP Model - Variables & Constraints

This is the most critical part of the notebook, where the real-world seating problem is translated into a formal mathematical model that pulp can understand.

Part A: Model Initialization and Variables

```
# 2) MILP MODEL
model = pulp.LpProblem("Seat_Allocation_SoftAttendance", pulp.LpMinimize)

# Variables
x = pulp.LpVariable.dicts(...) # Is a team present? (Yes/No)
assign = pulp.LpVariable.dicts(...) # How many people are assigned? (Integer)
y = pulp.LpVariable.dicts(...) # Is a bay used by a team? (Yes/No)
splits = pulp.LpVariable.dicts(...) # How many times is a team split? (Integer)
z = pulp.LpVariable.dicts(...) # Does a team fit entirely in one bay? (Yes/No)
attend_short = pulp.LpVariable.dicts(...) # By how many days is a team short of the 3-day
```

target? (Integer)

- **model = pulp.LpProblem(...)**: This line initializes the model. We name it "Seat_Allocation_SoftAttendance" and tell pulp that our goal is to **minimize** a certain value (which we'll define later in the objective function).
- **Decision Variables**: These are the "levers" that the solver can adjust to find the best solution.
  - x: A **binary** variable (0 or 1). x[s][d] = 1 means subteam s is scheduled to be in the office on day d. 0 means they are not.
  - assign: An **integer** variable. assign[s][b][d] holds the number of members from subteam s sitting in bay b on day d.
  - y: A **binary** variable. It acts as a switch. y[s][b][d] = 1 if subteam s uses bay b on day d at all (even for one person).
  - splits: An **integer** variable that counts how many *extra* bays a team needs on a given day. If a team is split across 3 bays, this value will be 2.
  - z: A **binary** variable used to check if an entire subteam fits into a single bay.
  - attend_short: A crucial **integer** "slack" variable. It represents the number of days a subteam falls short of its mandatory day attendance goal. In a perfect solution, this is always 0.

Part B: Constraints (The Rules)

These are the rules that any valid solution must follow.

1. **Attendance (Constraint 2.1)**: sum(x) + attend_short >= days_required
   - For every subteam, the total number of days they are present (sum of x) plus any shortfall must be at least minimum days required. This is a "soft" constraint because the attend_short variable allows it to be violated, but doing so will incur a heavy penalty.
2. **Full Team Assignment (Constraint 2.2)**: sum(assign) == M_s * x
   - If a subteam s is scheduled to be present on a day d (x[s][d] = 1), then the total number of its members assigned across all bays must equal its total membership (M_s). If they are not present (x[s][d] = 0), then zero members are assigned.
3. **Bay Capacity (Constraint 2.3)**: sum(assign) <= cap_b
   - For any given bay b on any given day d, the sum of all people assigned to it from *all* subteams cannot exceed its capacity.
4. **Linking Variables (Constraint 2.4)**: These constraints ensure the variables work together logically. For example, assign <= cap_b * y means you can only assign people (assign > 0) to a bay if its "in-use" switch is on (y = 1).
5. **Split Calculation (Constraint 2.5)**: splits >= sum(y) - 1
   - This defines how a "split" is calculated. It's the total number of bays a team uses on a day (sum(y)) minus one.
6. **Other Constraints (2.6, 2.7, 2.8)**: These add further logical rules, such as ensuring every

team attends at least **one** day and bounding the slack variable.

Cell 5: The Objective Function

```
# 3) OBJECTIVE
model += (
   pulp.lpSum(y)
   + split_penalty * pulp.lpSum(splits)
   + attend_penalty * pulp.lpSum(attend_short)
)
```

This is the goal we told pulp to **minimize**. It's a weighted sum of three components:

1. **Minimize Bay Usages (pulp.lpSum(y))**: The primary goal is to use as few bay-days as possible. This encourages the model to consolidate teams and keep bays empty when possible, leading to an efficient use of space.
2. **Penalize Splits (split_penalty * ...)**: The model adds a small penalty (1.0) for every split. The model will try to avoid splits, but it will prefer splitting a team over a more costly action, like failing to meet the attendance target.
3. **Heavily Penalize Attendance Shortfall (attend_penalty * ...)**: The model adds a **very large penalty (1000.0)** for every day a team is short of its 3-day attendance target. This high weight tells the solver that meeting the 3-day rule is the most important priority, and it should only be violated as an absolute last resort.

Cell 6: Solving the Model

```
# 4) SOLVE MILP
solver = pulp.PULP_CBC_CMD(msg=False, timeLimit=120)
status = model.solve(solver)
```

This cell executes the solver.

- **pulp.PULP_CBC_CMD(...)**: This line selects the solver to be used. CBC (COIN-OR Branch and Cut) is a powerful, open-source solver that comes with pulp.
- **timeLimit=120**: A timeout of 120 seconds is set. For complex problems like this, a solver could potentially run for hours. This limit ensures we get a result in a reasonable amount of time.
- **model.solve(solver)**: This command runs the optimization process. The solver explores thousands of possible combinations of the decision variables to find the one combination that satisfies all the constraints and results in the lowest possible value for the objective function.
- **status**: After the solver finishes, this variable will hold the result, such as "Optimal", "Infeasible" (no solution exists), or "Not Solved" (e.g., timed out).

Cell 7: Processing the Solution or Using the Fallback

This block of code handles the results from the solver.

If the Solution is "Optimal"

```
if status_str == "Optimal":
    schedule_df, expanded_df = build_solution_from_model()
    # ... display results ...
```

If the solver successfully found the best possible solution, the build_solution_from_model function is called. This function iterates through the optimal values of the x and assign variables and transforms them into two user-friendly pandas DataFrames:

- schedule_df: A summary view showing which days each subteam is present.
- expanded_df: A detailed, "long-format" view showing the exact assignment of every subteam to every bay on each day.

If the Solution is Not Optimal

```
else:
    print("... Falling back to Greedy constructive scheduler...")
    # ... call greedy_scheduler_force3() ...
```

If the MILP solver fails (e.g., it times out), the code executes a backup plan: a **greedy heuristic algorithm**.

- **How it Works**: The greedy algorithm is much simpler. It first sorts all subteams from largest to smallest. Then, it iterates through them, making the locally best decision at each step without considering the global picture. It places the largest team on the day with the most available capacity, then the next largest, and so on, until every team is scheduled for minimum required days.
- **Trade-off**: This approach is extremely fast and guarantees a valid solution. However, because it doesn't look at the whole problem at once, the solution it produces is usually not as efficient or "optimal" as the one from the MILP solver. It's a robust backup plan.

Cell 8: Saving the Results

```
try:
    schedule_df.to_csv("seat_schedule_summary.csv")
    expanded_df.to_csv("seat_schedule_expanded.csv")
except Exception:
    pass
```

The final step is to save the generated schedules into CSV files. This allows the results to be easily shared, analyzed in other tools like Excel, or used in other systems. The try...except block ensures that the code doesn't crash if the DataFrames haven't been created for some reason (e.g., if the problem was infeasible from the start).

SUBMITTED BY:

TEAM : **“ READ THE SINES! ”**

RITIKA KUMARI
PRIYANSHU MANDIL