Tutorial IV.III - DataFrames

Programming: Everyday Decision-Making Algorithms

Introduction

Imagine you're planning a Mars mission. Every task, from equipment checks to crew training, needs careful scheduling. In this tutorial, we'll learn how to use Python's Pandas library to implement scheduling algorithms like Earliest Due Date (EDD) and Shortest Processing Time (SPT).

> **Note**
>
> Before starting, ensure you have the `pandas` library installed. Depending on your operating system and installation method, you might need to use `pip`, `pip3` or `conda` to install it via the command line.
>
> ```{bash}
> pip install pandas
> ```
>
> ```{bash}
> pip3 install pandas
> ```
>
> ```{bash}
> conda install pandas
> ```

Let's begin by importing pandas:

```
import pandas as pd
```

Section 1 - Creating Task DataFrames

In mission planning, we need to track various tasks with their durations and deadlines. Let's create a DataFrame to manage these tasks from a dictionary and a list of index labels:

```
tasks = pd.DataFrame({
    'task': ['Equipment Check', 'Crew Training', 'Supply Loading'],
    'duration': [6, 3, 7],  # hours
    'deadline': [10, 15, 8]  # hours from now
},
index=['Task 1', 'Task 2', 'Task 3'])
print(tasks)
```

```
                  task  duration  deadline
Task 1  Equipment Check         6        10
Task 2    Crew Training         3        15
Task 3   Supply Loading         7         8
```

> **Note**
>
> The `pd.DataFrame()` function is used to create a DataFrame. A DataFrame is a two-dimensional, size-mutable, and heterogeneous tabular data structure with labeled axes (rows and columns). Here, we've created a DataFrame with three rows and four columns: `task`, `duration`, and `deadline`.

Exercise 1.1 - Create Mission Tasks DataFrame

Create a DataFrame named `mission_tasks` with critical Mars mission preparation tasks. Include columns for `task` (a string), `duration` (in hours), and `deadline` (hours from now) with the following data:

- Life Support Check: 4 hours, deadline 12
- Navigation Systems: 7 hours, deadline 16
- Fuel Loading: 6 hours, deadline 8
- Communication Setup: 3 hours, deadline 6

Make sure, that the hours and deadlines are integers. And create the DataFrame with the index labels `Task 1`, `Task 2`, `Task 3`, `Task 4`.

```
# YOUR CODE BELOW
```

```
# Test your answer
def test_mission_tasks(df):

    # Check if all required columns exist
    required_cols = ['task', 'duration', 'deadline']
    assert all(col in df.columns for col in required_cols), "Missing
required columns"

    # Check if we have the correct number of tasks
    assert len(df) == 4, "Should have exactly 4 tasks"

    # Check if all values are of correct type
    assert df['duration'].dtype in ['int32', 'int64'], "Duration should be
integers"
    assert df['deadline'].dtype in ['int32', 'int64'], "Deadline should be
integers"

    # Check if all required tasks are present
    required_tasks = {'Life Support Check', 'Navigation Systems', 'Fuel
Loading', 'Communication Setup'}
    assert set(df['task']) == required_tasks, "Missing or incorrect task
```

```
names"

test_mission_tasks(mission_tasks)

print("Mission tasks created successfully!")
```

Section 2 - Accessing Rows and Columns

When working with DataFrames, we sometimes need to access specific rows or columns. Let's explore different ways to do this using our previously created `tasks` DataFrame.

Accessing Columns

There are two main ways to access columns:

1. Using square bracket notation `[]`
2. Using dot notation `.`

```
# Access a single column (returns a Series)
durations = tasks['duration']
print("Durations:\n", durations)

# Access multiple columns (returns a DataFrame)
selected_cols = tasks[['task', 'duration']]
print("\nSelected columns:\n", selected_cols)

# Using dot notation (only works for simple column names)
priorities = tasks.deadline
print("\nPriorities:\n", priorities)
```

```
Durations:
 Task 1    6
Task 2    3
Task 3    7
Name: duration, dtype: int64

Selected columns:
                   task  duration
Task 1  Equipment Check         6
Task 2    Crew Training         3
Task 3   Supply Loading         7

Priorities:
 Task 1    10
Task 2    15
Task 3     8
Name: deadline, dtype: int64
```

Accessing Rows

We can access rows using:

1. `iloc[]` for integer-based indexing
2. `loc[]` for label-based indexing

```
# Get first row using iloc (integer location)
first_row = tasks.iloc[0]
print("First row:\n", first_row)

# Get rows 0 and 2
selected_rows = tasks.iloc[[0, 2]]
print("\nSelected rows:\n", selected_rows)

# Get specific rows and columns
subset = tasks.iloc[0, 0]  # First row, first column
print("\nSubset of data based on integer index:\n", subset)

# Get specific rows and columns using loc
subset = tasks.loc['Task 2', 'task']
print("\nSubset of data based on labels:\n", subset)
```

```
First row:
 task        Equipment Check
duration                  6
deadline                 10
Name: Task 1, dtype: object

Selected rows:
                    task  duration  deadline
Task 1  Equipment Check         6        10
Task 3   Supply Loading         7         8

Subset of data based on integer index:
 Equipment Check

Subset of data based on labels:
 Crew Training
```

Exercise 4.1 - Data Access Practice

Using the `mission_tasks` DataFrame from earlier, complete these tasks:

1. Extract only the 'duration' column as `duration_col`
2. Get the first two tasks with all columns as `first_two`
3. Create a subset containing only the 'task' and 'deadline' columns as `task_deadline_subset`

```
# YOUR CODE BELOW
```

```
# Test your answer
assert len(duration_col) == 4, "Duration column should have 4 entries"
assert len(first_two) == 2, "Should have first two rows"
assert list(task_deadline_subset.columns) == ['task', 'deadline'], "Should
```

```
only have task and deadline columns"
print("Data access exercises completed successfully!")
```

> Tip
>
> Remember:
>
> - Use `[]` for single or multiple columns
> - Use `iloc[]` when you want to access by position
> - Use `loc[]` when you want to access by label (using the index labels)
> - Slicing works similar to Python lists: `0:2` means "from 0 up to (but not including) 2"

Section 3 - Sorting DataFrames for EDD and SPT

The EDD algorithm minimizes maximum lateness by scheduling tasks in order of their deadlines. This is crucial for mission-critical tasks where delays could be catastrophic.

```
sorted_tasks = tasks.sort_values('deadline')
print("Tasks sorted by EDD:")
print(sorted_tasks)
```

```
Tasks sorted by EDD:
                   task  duration  deadline
Task 3   Supply Loading         7         8
Task 1  Equipment Check         6        10
Task 2    Crew Training         3        15
```

> Note
>
> The `sort_values()` method is used to sort the DataFrame by the `deadline` column. The `tasks` in this example is the DataFrame we created in the previous section. If you want to sort a different dataframe, just replace `tasks` with the name of the dataframe you want to sort. The method returns a new DataFrame with the rows sorted by the `deadline` column. If you want to sort the DataFrame by a different column, you can pass the name of the column as a string to the `by` argument. If you want to sort the DataFrame in descending order, you can pass `ascending=False` to the function.

If you want to add a new column, you can do so by assigning a new column to the DataFrame. For example, if you want to add a new column called `completion_time` to the `tasks` DataFrame, you can do so by assigning a new column to the DataFrame:

```
tasks['delayed_deadline'] = tasks['deadline'] + 1
print(tasks)
```

```
                  task  duration  deadline  delayed_deadline
Task 1  Equipment Check         6        10                11
Task 2    Crew Training         3        15                16
Task 3   Supply Loading         7         8                 9
```

Here, we've added a new column called `delayed_deadline` to the `tasks` DataFrame. The new column is calculated by adding 1 to the `deadline` column.

Exercise 2.1 - Apply EDD to Mission Tasks

Sort your `mission_tasks` using the EDD algorithm (sort by deadline).

```
# YOUR CODE BELOW
```

```
# Test your answer
assert edd_schedule.iloc[0]['task'] == 'Communication Setup', "First task
should be Communication Setup"
print("EDD Schedule created successfully!")
print(edd_schedule)
```

Exercise 2.2 - Calculate New Deadline

Unfortuantely, the mission has been delayed as some tasks are rather complex. Add a new column to the `mission_tasks` DataFrame called `new_deadline`. The new deadline is the old deadline plus the duration of the task.

```
# YOUR CODE BELOW
```

```
# Test your answer
assert 'new_deadline' in mission_tasks.columns, "Missing new_deadline
column"
assert all(mission_tasks['new_deadline'] > mission_tasks['deadline']), "New
deadlines should be later than original deadlines"
print("New deadlines calculated successfully!")
print(mission_tasks)
```

Section 3 - Looping over DataFrames

Now, we want to apply the EDD algorithm to the `mission_tasks` DataFrame. We can do this sorting the rows by deadline and then looping over the rows, computing the completion time and the maximum delay.

To do this, you can use the `iterrows()` method. The `iterrows()` method returns an iterator yielding index and row as a Series. For each row, you can access the values using the column names as keys. Let's do this step by step on the `tasks` DataFrame for the original deadline.

```
# Sort by deadline
sorted_tasks = tasks.sort_values('deadline')
```

```python
# Initialize completion time and max delay
completion_time = 0
max_delay = 0

# Loop over sorted tasks
for index, row in sorted_tasks.iterrows():
    print(f"Processing task: {row['task']} with index: {index}")
    completion_time += row['duration']

    if completion_time > row['deadline']:
        delay = completion_time - row['deadline']

        if delay > max_delay:
            max_delay = delay

print(f"Maximum delay: {max_delay} hours")
print(f"Completion time: {completion_time} hours")
print(sorted_tasks)
```

```
Processing task: Supply Loading with index: Task 3
Processing task: Equipment Check with index: Task 1
Processing task: Crew Training with index: Task 2
Maximum delay: 3 hours
Completion time: 16 hours
                   task  duration  deadline  delayed_deadline
Task 3   Supply Loading         7         8                 9
Task 1  Equipment Check         6        10                11
Task 2    Crew Training         3        15                16
```

> **Note**
>
> Note, how the index shows the index of the row in the original DataFrame!

If you want to access the index of the row, you can use the `name` attribute to access it. For example, if you want to access the index of the last row, you can use `sorted_tasks.iloc[-1].name`.

```python
print(sorted_tasks.iloc[-1].name)
```

```
Task 2
```

Exercise 3.1 - Apply Earliest Due Date to Mission Tasks

Now it is your turn: Apply the EDD algorithm to the `mission_tasks` DataFrame. Remember, that the EDD algorithm sorts the tasks by the original deadline and then loops over the rows, computing the completion time and maximum delay for each task (just like in the example above).

```
# YOUR CODE BELOW
```

```
# Test your answer
assert edd_schedule.iloc[0]['task'] == 'Communication Setup', "First task
should be Communication Setup"
assert max_delay == 4, "Maximum delay should be 4 hours"
assert completion_time == 20, "Completion time should be 20 hours"
print("EDD Schedule created successfully!")
print(edd_schedule)
```

Exercise 3.2 - Apply Shortest Processing Time to Mission Tasks

Now, apply the SPT algorithm to the `mission_tasks` DataFrame. Remember, that the SPT algorithm sorts the tasks by duration and then loops over the rows, computing the completion time and total waiting time for all tasks.

Compute the sum of the waiting times for all tasks in a variabel called `total_waiting_time` and compute the completion time of the last task in a variable called `completion_time`.

```
# YOUR CODE BELOW
```

```
# Test your answer
assert spt_schedule.iloc[0]['task'] == 'Communication Setup', "First task
should be Communication Setup"
assert total_waiting_time == 23, "Total waiting time should be 23"
assert completion_time == 20, "Completion time should be 20"
print("SPT Schedule created successfully!")
print(spt_schedule)
print(f"Total waiting time: {total_waiting_time}")
```

Conclusion

You've learned how to implement and compare different scheduling algorithms using Pandas DataFrames. These concepts aren't just theoretical - they're actively used in mission planning and can be applied to personal task management as well.

Remember: - EDD minimizes maximum lateness - SPT minimizes average completion time - Real-world scheduling often needs hybrid approaches considering priorities and dependencies

Solutions

You will likely find solutions to most exercises online. However, we strongly encourage you to work on these exercises independently without searching explicitly for the exact answers to the exercises. Understanding someone else's solution is very different from developing your own. Use the lecture notes and try to solve the exercises on your own. This approach will significantly enhance your learning and problem-solving skills.

Remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities. If you encounter difficulties, review the lecture materials, experiment with different approaches, and don't hesitate to ask for clarification during class discussions.

Later, you will find the solutions to these exercises online in the associated GitHub repository, but we will also quickly go over them next week. To access the solutions, click on the Github button on the lower right and search for the folder with today's lecture and tutorial. Alternatively, you can ask ChatGPT or Claude to explain them to you. But please remember, the goal is not just to complete the exercises, but to understand the concepts and improve your programming abilities.

That's it for this weeks tutorials. Next week we are going to cover the topic of randomness in everyday life.