Filtering Data by Time of Day: A Pandas Tutorial

Authored by

Mohammed looti

November 13, 2025

When conducting sophisticated analysis of time-series data, a frequent and essential requirement is the ability to filter specific records based solely on the time of day, completely ignoring the calendar date. For example, a business analyst might need to isolate all server activity logs or sales transactions that occurred strictly between 9:00 AM and 5:00 PM, spanning several months or even years of collected information. Executing this specific type of time-based query efficiently within a Pandas DataFrame necessitates a dedicated, high-performance function.

The most intuitive and robust function designed specifically for this temporal slicing challenge is **`between_time()`**. This powerful method is engineered to select rows where the primary index--which must be a DatetimeIndex--falls within a user-defined time range. By leveraging this built-in Pandas functionality, analysts can achieve significantly cleaner and more performant code compared to complicated manual operations involving string parsing or iterative Boolean masking across the index.

Understanding the Power of between_time() for Temporal Analysis

In the realm of data science, especially when processing high-frequency datasets such as sensor readings, financial trades, or transactional logs, selecting data based purely on consistent time intervals is fundamental. This process allows analysts to identify key behavioral patterns, such as peak usage hours, off-peak activity, or critical operational windows. Traditional indexing methods in Pandas often default to requiring the specification of both the full date and time, which creates unnecessary complexity when the goal is to analyze recurring daily cycles, independent of which day they occurred.

The **`between_time()`** function effectively resolves this analytical hurdle. Its core strength lies in its ability to operate exclusively on the time component (HH:MM:SS) of the index values. This isolation means you can effortlessly filter data that spans multiple days, weeks, or even years, using nothing more than simple start and end time strings. For any professional working with temporal data in Pandas, this function is an indispensable utility for streamlining time-based data retrieval and analysis.

It is important to emphasize the prerequisite: the DataFrame must possess a robust, ordered DatetimeIndex. If your time data resides in a standard column, you must first convert it to datetime objects and set it as the index using the `set_index()` method before applying **`between_time()`**.

Syntax and Essential Parameters of between_time()

To harness this method effectively, a clear understanding of its formal syntax and the controlling arguments is crucial. The function is always invoked directly on the Pandas DataFrame object, assuming the necessary DatetimeIndex is already in place for row selection. The simplicity of its implementation belies the complexity of the filtering it performs.

The formal syntax for calling the `between_time()` function is structured as follows:

```
pandas.DataFrame.between_time(start_time, end_time, inclusive='both', axis=None)
```

We provide a detailed breakdown of both the required positional arguments and the optional keyword parameters that define the filtering behavior:

`start_time`: This required argument sets the initial lower time limit for the filter. It offers flexibility in input formats, accepting a simple string (e.g., '4:30', '09:00:00'), a Python `time` object for greater precision, or even simple integer representations of hours or minutes.

`end_time`: This required argument defines the final upper time limit for the filter. It must adhere to the same flexible input formats accepted by `start_time`. Note that the function handles time intervals that cross midnight (e.g., 10 PM to 4 AM) gracefully.

`inclusive`: This optional parameter dictates how the boundaries (`start_time` and `end_time`) are handled in the resulting dataset. The default setting is `'both'`, which includes data points exactly matching the start and end times. Other precise options include `'neither'` (strict exclusion of both boundaries), `'left'` (includes `start_time` only), and `'right'` (includes `end_time` only).

`axis`: This parameter specifies which DataFrame axis the function should inspect for time values. The default value, `None`, correctly instructs the function to operate on the index (Axis 0). Although it is technically possible to filter along columns (Axis 1), for time-based slicing, the index is almost universally the correct and intended target.

Example: Setting Up DataFrames for Time Slicing

To clearly demonstrate the practical utility of `between_time()`, we will first construct a representative sample Pandas DataFrame. This DataFrame simulates a simple dataset, such as sales figures collected at high temporal granularity over a short period. The index, which contains the timestamps, is the perfect target for our time-based filtering exercise.

Assume we are monitoring the total sales revenue generated by a retail employee. The data points are recorded every 30 minutes, beginning precisely at 4:00 AM on a fixed date, January 1, 2018. We utilize the highly efficient Pandas utility `pd.date_range()` to generate a comprehensive and accurate DatetimeIndex for the experiment.

import pandas as pd

#create DataFrame
df = pd.DataFrame({'sales': },
index=pd.date_range('1/1/2018 4:00:00', periods=9, freq='30min'))

#view DataFrame
print(df)

sales
2018-01-01 04:00:00 2
2018-01-01 04:30:00 5
2018-01-01 05:00:00 5
2018-01-01 05:30:00 4
2018-01-01 06:00:00 7
2018-01-01 06:30:00 8
2018-01-01 07:00:00 9
2018-01-01 07:30:00 12
2018-01-01 08:00:00 10

As the output clearly illustrates, the DatetimeIndex is correctly generated and spaced at 30-minute increments. This structured index now allows us to proceed with the core operation: using `between_time()` to extract a highly specific temporal subset of this DataFrame.

Practical Application: Executing Basic Time Slicing

For our initial practical application, let us assume our analytical focus is strictly on the sales data recorded during the early morning operational window, specifically between 4:30 AM and 6:30 AM. Utilizing `between_time()` simplifies this extraction tremendously. We simply pass the start and end times as concise strings. Since we intentionally omit the optional `inclusive` argument, it automatically utilizes the default value of `'both'`, ensuring that if timestamps exactly matching 4:30 AM and 6:30 AM exist in the index, they will be included in our resulting subset.

The following concise syntax executes this powerful time-based filter:

#extract all rows with time between 4:30 and 6:30
df.between_time('4:30', '6:30')

sales
2018-01-01 04:30:00 5
2018-01-01 05:00:00 5
2018-01-01 05:30:00 4
2018-01-01 06:00:00 7
2018-01-01 06:30:00 8

The resulting DataFrame flawlessly returns all rows whose time component falls precisely within

the specified two-hour window. The key takeaway here is that **`between_time()`** is laser-focused on the time stamp (HH:MM:SS), entirely disregarding the date (YYYY-MM-DD) portion of the index value. If this original DataFrame had spanned an entire year, the function would have returned every single row from all 365 days that matched the 4:30 AM to 6:30 AM interval, making it invaluable for cyclical analysis of time-series data.

Fine-Tuning Boundaries with the inclusive Parameter

In statistical modeling, financial reporting, or database querying, the precise handling of boundary conditions is often non-negotiable. Analysts frequently require the ability to strictly exclude the start time, the end time, or both, particularly when defining non-overlapping time bins or intervals. This is exactly why the **`inclusive`** parameter provides such critical, granular control over the filtering process.

We can easily modify our previous example to demonstrate this control. Suppose we need to analyze the data but must exclude the very last measurement at 6:30 AM while retaining the starting point at 4:30 AM. We achieve this by setting the `inclusive` argument to **`'left'`**, indicating that only the left (start) boundary should be included in the results.

#extract all rows with time between 4:30 and 6:30 (only including start time)
df.between_time('4:30', '6:30', inclusive='left')

sales
2018-01-01 04:30:00 5
2018-01-01 05:00:00 5
2018-01-01 05:30:00 4
2018-01-01 06:00:00 7

As expected, the row corresponding to 6:30 AM is now successfully excluded from the result set, as the filter only includes the left boundary (the start time). This level of control is essential for ensuring data integrity in subsequent calculations.

Conversely, if the requirement was to exclude the start time (4:30 AM) but ensure the inclusion of the end time (6:30 AM), we would set `inclusive='right'`. This scenario is frequently encountered when preparing data for aggregation or charting where the intervals must be closed on the right side.

#extract all rows with time between 4:30 and 6:30 (only including end time)
df.between_time('4:30', '6:30', inclusive='right')

sales

2018-01-01 05:00:00 5
2018-01-01 05:30:00 4
2018-01-01 06:00:00 7
2018-01-01 06:30:00 8

In this final example, the row captured at 4:30 AM is excluded, but the row at 6:30 AM is retained. By selecting `'left'`, `'right'`, `'both'` (the default), or `'neither'`, you gain absolute control over the boundary conditions necessary for precise and reliable time-series data analysis in Pandas. This functional specificity eliminates the need for complex indexing operations and greatly improves code clarity.

Note: For advanced usage scenarios, such as filtering across midnight or handling specific edge cases related to time zones, it is highly recommended to consult the complete documentation for the `between_time()` function in Pandas.

Additional Resources

The following tutorials explain how to perform other common tasks in pandas:

Featured Posts

5 Statistical Biases to Avoid

April 25, 2024

5 Free Statistics Courses for Beginners

April 19, 2024

5 MIT Statistics Courses That Are Free

April 18, 2024

5 Free Books to Learn Statistics

April 18, 2024

How to Use the info() Method in Pandas

April 12, 2024

How to Use pct_change() in Pandas

April 12, 2024