How to Read Specific Columns from Excel File using Pandas?

Authored by

stats writer

November 21, 2025

The efficient handling of external data sources is fundamental in modern data science and analysis workflows. When working with large datasets often stored in proprietary formats like Excel file spreadsheets, minimizing memory usage and processing time is paramount. Fortunately, the widely adopted Pandas library--a cornerstone for data manipulation in Python--provides robust tools for selective data ingestion, ensuring you only load the necessary components of a spreadsheet.

The primary method for accessing Excel data within Pandas is the powerful `read_excel()` function. While this function, by default, reads the entire contents of a specified sheet into a DataFrame object, it offers specialized parameters that allow analysts to cherry-pick data columns. This capability is essential for managing memory constraints, especially when dealing with Excel files that contain dozens of irrelevant columns alongside the few critical variables needed for analysis.

Leveraging the `usecols` parameter within the read_excel() method is the key to this selective loading process. The parameter accepts several formats, including a list of strings containing the exact column names or column index integers, or, as demonstrated below, specific Excel-style letter notation (e.g., 'A', 'C', 'A:C'). Understanding the nuances of these formats allows data engineers and analysts to craft precise loading scripts that enhance performance and streamline data preparation tasks.

Three Effective Techniques for Selective Column Reading

When you need to import data from an Excel spreadsheet into a Pandas DataFrame, you typically do not require every column present in the source file. Focusing the import process on essential variables drastically improves efficiency. The read_excel() function accommodates this need through three highly versatile implementations of the `usecols` parameter, detailed below. These methods provide flexibility whether you need specific, non-contiguous columns or large, sequential blocks of data.

These techniques are particularly useful in environments where source data is constantly evolving or where performance optimization is critical. By explicitly defining the columns to import, you create a robust script less susceptible to errors caused by changes in the column order or the addition of new, unnecessary columns in the source spreadsheet. Furthermore, specifying column ranges using Excel's native letter notation often saves time compared to manually listing every required column name.

Below, we outline the fundamental syntax for employing the `usecols` parameter. Note that in these generic examples, `my_data.xlsx` represents your input file, and the letters ('A', 'C', etc.) refer to the standard Excel column index headers. These concise examples serve as the foundational blueprints for the practical demonstrations that follow, illustrating how to manage various column

selection scenarios.

Method 1: Reading Specific, Non-Contiguous Columns by Letter

df = pd.read_excel('my_data.xlsx', usecols='A, C')

Method 2: Reading a Contiguous Range of Columns

df = pd.read_excel('my_data.xlsx', usecols='A:C')

Method 3: Reading Multiple Column Ranges and Specific Individuals

df = pd.read_excel('my_data.xlsx', usecols='A:C, F, G:J')

Understanding the Example Dataset Structure

To fully illustrate these column selection methods, we will apply them to a practical scenario using a sample Excel file named **player_data.xlsx**. This dataset is typical of sports statistics, containing multiple fields, only some of which may be relevant for a particular analysis task. The file contains essential metrics for several players, including their team identification, offensive output (points), defensive contribution (rebounds), and playmaking ability (assists).

The ability to visualize the source data is crucial before writing the import script. Observing the structure helps determine whether you need to use individual column identifiers (like 'A' and 'C') or range notation (like 'A:C'). For instance, if the file has 10 columns, but columns D through J are just metadata timestamps, selectively loading A, B, and C saves seven columns worth of unnecessary data from being processed and stored in memory, providing a significant efficiency boost.

The following image represents the structure and content of our sample file, **player_data.xlsx**, which will be the basis for all subsequent examples. We will use Pandas to selectively extract subsets of this data based on the requirements of each demonstration, showcasing how the `usecols` parameter provides granular control over the data ingestion pipeline.

	A	B	C	D	E	F
1	team	points	rebounds	assists		
2	A	24	8	5		
3	B	20	12	3		
4	C	15	4	7		
5	D	19	4	8		
6	E	32	6	8		
7	F	13	7	9		
8						
9						
10						
11						
12						
13						
14						
15						
16						
17						
18						
19						
20						

Example 1: Selecting Specific, Non-Contiguous Columns

The most common requirement in data filtering is importing only a handful of specific columns that are not adjacent to each other. For our **player_data.xlsx** file, let us assume we are only interested in the **team** column (Column A) and the **rebounds** column (Column C). We are deliberately excluding the **points** column (Column B) and any other subsequent columns, thereby focusing our analysis solely on a subset of the available variables for a defensive study.

To achieve this specific selection, we pass a comma-separated string of the required column letters to the `usecols` parameter. This approach signals to the read_excel() function exactly which columns to parse and retain, discarding all others during the I/O process. This method is highly recommended when dealing with sparse data requirements across a wide sheet, as it ensures maximal data pruning at the loading stage, optimizing subsequent processing steps.

The code snippet below demonstrates the required syntax. By setting `usecols='A, C'`, we instruct Pandas to read only the data from the first and third columns of the spreadsheet. The resulting DataFrame, `df`, will contain two columns, corresponding to the original data found in columns A and C of the Excel source, maintaining the original column headers if they were present

in the source file.

We can use the following code to import the data in columns **A** and **C** from the Excel file:

```
import pandas as pd

#import columns A and C from Excel file
df = pd.read_excel('player_data.xlsx', usecols='A, C')

#view DataFrame
print(df)
```

```
team rebounds
0 A 8
1 B 12
2 C 4
3 D 4
4 E 6
5 F 7
```

As the output confirms, the resulting DataFrame only contains the **team** and **rebounds** data, successfully excluding the intermediate **points** column name (B). This illustrates the precision and efficiency gained by leveraging the comma-separated letter notation within the `usecols` parameter for targeted data extraction.

Example 2: Reading a Contiguous Range of Columns

When the required columns form an unbroken sequence within the Excel sheet, using range notation simplifies the syntax significantly. Instead of listing every individual column (e.g., A, B, C, D...), we can define the start and end points of the desired block. This method is highly efficient for data sources where critical information is grouped together at the beginning or middle of the spreadsheet, such as sequential measurements or key identifiers.

For this example, let's assume we need all core statistics: **team** (A), **points** (B), and **rebounds** (C). Since these three columns are contiguous, starting at A and ending at C, we can define the requirement concisely. The syntax for defining a range is straightforward: use a colon (`:`) between the starting and ending column letters, mirroring the standard cell referencing used within Excel itself.

The key benefit of using range notation, such as `usecols='A:C'`, is clarity and conciseness, especially when dealing with wide datasets. It reduces the chance of manual transcription errors

that might occur when typing out a long list of individual column identifiers. Furthermore, it explicitly communicates the intent to load a block of data, improving code readability for other developers reviewing the script.

We can use the following code to import the data in columns **A** through **C** from the Excel file:

import pandas as pd

#import columns A through C from Excel file
df = pd.read_excel('player_data.xlsx', **usecols='A:C')**

#view DataFrame
print(df)

team points rebounds
0 A 24 8
1 B 20 12
2 C 15 4
3 D 19 4
4 E 32 6
5 F 13 7

The resulting DataFrame successfully captured all columns from **A** through **C**, demonstrating how the range syntax provides a clean and effective way to handle contiguous data blocks. If there were other columns present (e.g., D, E, F), they would have been automatically skipped during the import process, ensuring optimal utilization of computational resources by only loading necessary data.

Example 3: Combining Ranges and Individual Columns

The versatility of the `usecols` parameter truly shines when a dataset requires the import of both contiguous ranges and specific, isolated columns simultaneously. This hybrid approach is necessary when the core data resides in a block (e.g., A:C), but a critical metadata field or identifier is located far away (e.g., column F or G), separated by irrelevant fields.

In this final illustration, we aim to import the core statistics (Columns A through C) along with the **assists** data, which is located immediately following in column D. While this could be achieved simply by extending the range to `A:D`, we use the combined syntax `'A:C, D'` to demonstrate how Pandas seamlessly merges selection criteria defined by range notation and individual column identifiers within a single string input.

This technique, capable of handling highly complex selection patterns (such as `'A:C, F, G:J, M'`), is the most powerful method for large-scale data ingestion. It allows the user to define a highly precise data schema at the point of import, filtering out all noise and focusing only on the variables essential for immediate analysis, without the need for post-load DataFrame dropping operations which are less efficient.

We can use the following code to import the data in columns **A** through **C** and column **D** from the Excel file:

import pandas as pd

#import columns A through C and column D from Excel file
df = pd.read_excel('player_data.xlsx', **usecols='A:C, D')**

#view DataFrame
print(df)

team points rebounds assists
0 A 24 8 5
1 B 20 12 3
2 C 15 4 7
3 D 19 4 8
4 E 32 6 8
5 F 13 7 9

The resulting DataFrame now correctly includes all data points from columns **A** through **D**. This example reinforces the principle that the `usecols` parameter accepts a concatenated string of both range definitions (A:C) and individual column identifiers (D), separated by commas, providing comprehensive control over the input data schema.

Advanced Considerations: Using Column Names and Indices

While using Excel column letters (A, B, C) is convenient and often quick for small tasks, it is generally less robust than referencing columns by their actual header names, especially in professional environments where the column order might shift or new columns are inserted unexpectedly. If your Excel file has a header row (which Pandas assumes by default unless specified otherwise), you can pass a list of strings containing the exact column names you wish to load instead of the letter notation.

This name-based method offers significant advantages in terms of code maintainability and readability. For example, instead of writing `usecols='A, C'`, one could write `usecols=`. This

makes the code self-documenting, as the reader immediately understands the purpose of the imported data without needing to cross-reference the original spreadsheet layout. This approach is widely considered best practice when the header row is guaranteed to be present and accurate.

Alternatively, when dealing with very large spreadsheets where performance is critical and column names are long, Pandas also supports providing the indices of the columns (0-indexed integers) instead of the letter or name. For example, to read the first and third columns, you could use `usecols=`. While slightly less intuitive than using names, this index-based method is computationally faster during the parsing process, as Pandas bypasses the overhead of mapping Excel letters or header strings to internal indices.

Conclusion: Mastering Selective Data Ingestion with Pandas

The Pandas library, via its versatile read_excel() function, provides robust and highly flexible mechanisms for importing data from spreadsheets. By mastering the `usecols` parameter, analysts gain powerful control over the data ingestion process, allowing them to optimize performance, manage memory effectively, and ensure that only the most relevant variables are brought into the computational environment. This capability is essential for handling large, complex, and often messy real-world data sources.

Whether you require specific, non-contiguous columns (e.g., `'A, C'`), continuous ranges (e.g., `'A:C'`), or complex combinations of both, Pandas supports all necessary syntax through simple string inputs. This focused approach is foundational to building scalable and efficient data analysis scripts in Python, minimizing overhead and accelerating the time-to-insight by reducing the volume of data handled at every stage of processing.

For those seeking to explore all possible configurations and advanced features of this vital function, the complete official documentation for the Pandas **read_excel()** function is an essential reference point.