How to Use read_csv with usecols Argument

Authored by

stats writer

November 21, 2025

In modern pandas data processing workflows, efficiently loading data is paramount. The `read_csv()` function is the cornerstone for importing data stored in a CSV file format. A key feature enhancing this efficiency is the `usecols` argument, which allows users to selectively import only necessary columns into a DataFrame. Utilizing `usecols` is not just a stylistic preference; it is a critical optimization technique that significantly reduces both processing time and memory management overhead when dealing with large datasets containing hundreds of features.

The Necessity of Selective Data Loading

When working with enterprise-level datasets, it is common for source files to contain dozens or even hundreds of columns. Often, an analyst or data scientist only requires a small subset of these columns for a specific task, such as modeling or exploratory data analysis. Loading the entire dataset unnecessarily consumes valuable system resources. By specifying the exact columns needed upfront using the `usecols` parameter, we ensure that the pandas parser only processes and retains the relevant information, leading to much faster execution times, especially on systems with limited RAM.

Furthermore, excluding superfluous columns immediately improves code clarity and maintainability. When reviewing code that utilizes `read_csv()` with a defined `usecols` list, it is instantly clear what features are intended for use in the subsequent analytical steps. This practice avoids the unnecessary creation of a large, complex intermediate DataFrame that would require subsequent dropping operations, thereby streamlining the initial data ingestion phase of any project.

Understanding the `usecols` Parameter

The `usecols` parameter expects a list-like object as input. This list defines exactly which columns should be parsed from the source CSV file and subsequently loaded into the resulting DataFrame. The versatility of `usecols` lies in its ability to accept identifiers in two primary formats, catering to different scenarios and user preferences.

The two reliable methods for defining the required columns are:

Method 1: Column Names (Strings): Providing a list of strings corresponding exactly to the column headers present in the CSV file. This method is highly readable and robust, as column names generally remain consistent even if the order of the columns shifts slightly.
Method 2: Column Positions (Integers): Providing a list of integers representing the zero-based index positions of the desired columns. This is useful when the CSV file lacks headers or when the file structure is guaranteed to be stable, making positional indexing reliable for automation.

The following sections detail these two approaches, demonstrating the precise syntax required for

each method when executing the `read_csv()` function.

Method 1: Specifying Columns by Name

When importing columns using their explicit names, the list passed to `usecols` must consist of strings that exactly match the headers defined in the first row of the CSV file. This approach is generally recommended for production environments because it relies on semantic identifiers rather than fragile positional indices. If the order of non-required columns changes in the source file, this method remains completely unaffected, guaranteeing the correct data is always loaded.

The structure below illustrates the fundamental syntax for this technique:

df = pd.read_csv('my_data.csv', usecols=)

In this example, the resulting `df` DataFrame will only contain the data corresponding to 'this_column' and 'that_column', dramatically speeding up the loading process compared to importing all available columns in 'my_data.csv'.

Method 2: Specifying Columns by Position (Integer List)

Alternatively, developers can choose to identify the desired columns using their zero-based index positions. This method requires careful consideration of the file structure, as any change in the column order will break the script, potentially leading to incorrect data being loaded without raising a visible error. However, positional indexing can be beneficial when dealing with standardized, machine-generated files where column order is guaranteed, or when the CSV file deliberately omits header information.

The core requirement is that the list provided to `usecols` must contain integers representing the column indices, starting with 0 for the first column.

The standard implementation for positional selection is shown here:

df = pd.read_csv('my_data.csv', usecols=)

This command instructs pandas to skip all columns except the one at index 0 (the first column) and the one at index 2 (the third column). It is essential to remember this zero-indexing convention to accurately map the desired columns.

Illustrative Dataset for Examples

To demonstrate both methods effectively, we will utilize a sample CSV file named

`basketball_data.csv`. This file contains several metrics related to basketball team performance. We will specifically focus on importing only the 'team' and 'rebounds' columns using both string names and positional indices.

The structure of the source file is visually represented below, showing the available columns and their corresponding zero-based indices:

```
team,points,rebounds
A, 22, 10
B, 14, 9
C, 29, 6
D, 30, 2
```


Notice the headers and the content; specifically, 'team' is at index 0, 'points' is at index 1, 'rebounds' is at index 2, and 'assists' is at index 3. This visual reference will be crucial for understanding the positional examples that follow.

Practical Application: Selecting Columns by Name (Example 1)

In this first practical scenario, our goal is to isolate the team identification and the total rebounds recorded, discarding 'points' and 'assists'. We achieve this by providing `usecols` with a list containing the exact string names: . This approach is highly resilient and clear, making the intent of the data loading operation unmistakable.

We can use the following code to import the CSV file and only use the columns called 'team' and 'rebounds':

import pandas as pd

#import DataFrame and only use 'team' and 'rebounds' columns
df = pd.read_csv('basketball_data.csv', usecols=)

#view DataFrame
print(df)

team rebounds
0 A 10
1 B 9
2 C 6
3 D 2

Upon executing this code, observe the output DataFrame. Only the `team` and `rebounds` columns were successfully imported. This success confirms that the `usecols` argument effectively filtered the data during the initial parsing phase of the `read_csv()` function, preventing the unwanted columns from ever being loaded into memory. This demonstrates the efficiency and precision of specifying columns by name.

Practical Application: Selecting Columns by Index Position (Example 2)

In the second example, we achieve the exact same result--loading the 'team' and 'rebounds' columns--but this time, we rely on their positional indices. Referring back to the dataset image, 'team' is the first column (index 0) and 'rebounds' is the third column (index 2). Therefore, the list passed to `usecols` will be .

This method is faster in some low-level operations as it avoids string lookup, but as noted, it sacrifices flexibility. If the source file changes, column 2 might suddenly contain 'assists' instead of 'rebounds', leading to silent data corruption in the analysis pipeline. However, for fixed data formats, it is a perfectly valid and concise method.

We can use the following code to import the CSV file and only use the columns in index positions 0 and 2:

import pandas as pd

#import DataFrame and only use columns in index positions 0 and 2
df = pd.read_csv('basketball_data.csv', usecols=)

#view DataFrame
print(df)

team rebounds
0 A 10
1 B 9

2 C 6
3 D 2

The resulting DataFrame is identical to the output of Example 1. This reaffirms that **`usecols`** successfully identified and extracted only the columns corresponding to index positions 0 and 2. It is critical to grasp that column indexing in pandas, consistent with Python conventions, always begins with 0.

Key Considerations When Using `usecols`

While **`usecols`** is highly powerful, there are several advanced operational details and best practices that developers should internalize to avoid common pitfalls when performing read_csv() operations.

Zero-Indexing Rule: The first column in the CSV file has an index position of 0. This is a fundamental concept in Python and pandas that must be strictly followed when using integer lists for column selection. Misunderstanding this can lead to incorrect column selections. For instance, requesting index 1 will retrieve the second column, not the first.

Mixing Types is Not Allowed: Although the two methods (names and positions) are equally valid individually, the list provided to **`usecols`** cannot mix strings and integers. You must choose one method for a single **`read_csv()`** call. If you attempt to pass `usecols=`, pandas will raise a `ValueError` because it cannot interpret the heterogeneous list type consistently during parsing.

Handling Errors: If a string name provided in the **`usecols`** list does not exist in the CSV header, pandas will immediately raise a `ValueError`, indicating that the column name was not found. This fail-fast behavior is beneficial as it prevents unexpected data issues further down the processing pipeline. Always double-check column names, especially regarding case sensitivity.

Summary and Further Resources

The **`usecols`** argument is an indispensable tool for efficient data loading in pandas, offering flexibility through both name-based and positional selection methods. By mastering this argument, users can significantly optimize memory usage and processing speed, particularly when dealing with large, complex datasets.

To continue mastering data manipulation in Python, consider exploring these related topics:

How to filter rows using boolean indexing after data ingestion.
Methods for handling missing values (NaNs) in the loaded DataFrame.
Using the **`skiprows`** argument in **`read_csv()`** to ignore specific initial rows.

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