Learning Pandas: Creating New DataFrames by Subsetting Existing Data

Authored by

Mohammed looti

October 29, 2025

The Fundamentals of DataFrame Subsetting in Pandas

The Pandas library, an essential component of the Python data science ecosystem, provides robust tools for data manipulation and analysis. At its core lies the DataFrame, a two-dimensional, labeled data structure that is ubiquitous in modern data processing workflows. During typical data analysis projects, it is frequently necessary to extract specific subsets of data from a large, existing DataFrame to form a new, more focused structure. This critical process, known as subsetting, is fundamental for tasks such as feature engineering for machine learning models, isolating relevant variables for statistical testing, or simply enhancing readability by reducing dataset complexity.

Creating a new DataFrame based on selections from its predecessor can be accomplished through several high-performance techniques offered by Pandas. These methods cater to various analytical needs, whether the goal is to precisely select a collection of columns, extract a single column for specialized processing, or streamline the data by explicitly removing unwanted variables. Understanding these distinct approaches ensures that data preparation is both efficient and structurally sound, paving the way for reliable analysis downstream.

This comprehensive guide is dedicated to exploring the three most effective and commonly employed methods for generating new DataFrames via column selection or exclusion. We will meticulously review the required syntax for each technique, provide clear, practical code examples, and--most importantly--highlight the critical considerations surrounding data integrity. Specifically, we will emphasize the proper application of the `.copy()` method, which is paramount for preventing unintended modifications to the source data structure.

Preparing the Environment: Defining Our Sample Data

To ensure clarity and consistency throughout the subsequent technical demonstrations, we must first establish a standardized dataset. This common reference point will allow us to observe the effects of each subsetting technique in a tangible and repeatable manner. We will initialize a simple DataFrame, which we will name `old_df`, designed to simulate typical structured data--in this case, hypothetical team performance statistics.

The structure of `old_df` will include four distinct columns: `team` (categorical identifier), `points`, `assists`, and `rebounds` (numerical metrics). These variables represent key performance indicators in a sports context, providing a meaningful context for our column manipulation and subsetting operations. The following Python code snippet illustrates the necessary steps to import the library and construct this base DataFrame using the `pandas.DataFrame` constructor.

It is essential to familiarize yourself with the column names and overall structure of this initial dataset. All subsequent methods will operate on `old_df`, and the resulting new DataFrames will be derived directly from its contents. Pay close attention to the column labels, as these strings form

the basis of all indexing and selection operations moving forward.

import pandas as pd

# Create the sample DataFrame
old_df = pd.DataFrame({'team': ,
'points': ,
'assists': ,
'rebounds': })

# View the initial DataFrame
print(old_df)

Method 1: Creating a New DataFrame by Selecting Specific Columns

One of the most frequently utilized operations in data preparation involves isolating a select group of features from a larger dataset. This technique is indispensable when preparing input variables for a statistical model, focusing on key metrics for reporting, or simply reducing the dimensionality of the working data structure. The power of Pandas indexing allows for a straightforward solution to this requirement.

To perform this multi-column extraction, the user supplies a Python list of desired column names to the indexing operator of the original DataFrame. This is syntactically represented using double square brackets: `old_df]`. The outer brackets signify the indexing operation on the DataFrame, while the inner brackets define the list object containing the column identifiers. Crucially, when using this selection method, the operation may sometimes return a "view" of the original data rather than an independent "copy," leading to potential issues related to the SettingWithCopyWarning if subsequent modifications are attempted.

To guarantee the absolute independence of the newly created DataFrame, it is considered best practice to append the `.copy()` method immediately after the column selection. This command forces a deep copy of the selected data, ensuring that `new_df` exists in its own memory space, completely isolated from `old_df`. For instance, if our analysis requires only the `points` and `rebounds` statistics from our sample data, the following code demonstrates the correct, robust implementation of this selection technique.

 # Create new DataFrame containing only 'points' and 'rebounds'
new_df = old_df].copy()

# Display the resulting DataFrame
print(new_df)

points rebounds
0 18 11
1 22 8
2 19 10
3 14 6
4 14 6
5 11 7
6 20 9
7 28 12

# Confirm the data type is indeed a DataFrame
type(new_df)

pandas.core.frame.DataFrame

The resulting structure, `new_df`, is a clean, independent subset that contains only the **points** and **rebounds** columns, ready for focused analysis without risk of side effects on the original dataset.

Method 2: Extracting a Single Column as a DataFrame

Although many scenarios demand the extraction of multiple columns, there are specific instances where a data scientist requires a new DataFrame consisting of just one variable from the source. When selecting a single column using standard single-bracket indexing (e.g., `old_df`), Pandas defaults to returning a Series object. While a Series is highly efficient, certain downstream operations, functions, or model input requirements necessitate the structure of a two-dimensional DataFrame, even if it only possesses one column.

To ensure the output remains a DataFrame structure, the exact same double square bracket notation employed for multi-column selection must be used, even when naming only one column. For instance, selecting the `points` column as a DataFrame is achieved via `old_df]`. This syntax signals to the Pandas indexer that the desired output structure is a DataFrame. This consistency in syntax is key to maintaining uniform data structures across different stages of a pipeline.

As before, to protect the integrity of the original data, the utilization of the `.copy()` method remains mandatory. This ensures that the newly isolated column, now residing within its dedicated DataFrame, is a true deep copy. Let us apply this technique to our sample data, creating a new DataFrame that focuses exclusively on the `points` column, demonstrating how to retain DataFrame dimensionality even for single-feature extraction.

 # Create new DataFrame containing only the 'points' column
new_df = old_df].copy()

```
# Display the resulting DataFrame
print(new_df)

points
0 18
1 22
2 19
3 14
4 14
5 11
6 20
7 28

# Confirm the data type
type(new_df)

pandas.core.frame.DataFrame
```

The output clearly shows that `new_df` is now a DataFrame containing solely the **points** column, confirming its successful extraction as a dedicated DataFrame.

Method 3: Excluding Unwanted Columns Using the `.drop()` Method

In contrast to selection, which involves explicitly listing the desired columns, the exclusion method is often superior when the dataset is wide (i.e., contains many columns) and only a small handful of variables need to be discarded. This approach is highly efficient for feature reduction, particularly when specific columns have been identified as irrelevant, redundant, or containing too many missing values.

The primary tool for this operation in Pandas is the highly versatile `.drop()` method. This method accepts a single column name or a list of column names to be removed. Critically, to instruct Pandas to perform the removal across the columns (the vertical dimension) rather than the default row removal (the horizontal dimension), the parameter `axis=1` (or its equivalent, `axis='columns'`) must be explicitly passed. Failure to specify `axis=1` will result in a key error, as the method will attempt to find the specified names in the row index.

A notable difference between `.drop()` and the indexing methods discussed previously is that `.drop()` inherently returns a new DataFrame, effectively making a copy of the remaining data structure by default. Therefore, explicitly calling `.copy()` is generally unnecessary immediately following a basic drop operation, simplifying the syntax while maintaining data safety. Let's demonstrate this by creating a new DataFrame from `old_df` where the `points` column is

intentionally excluded.

 # Create new DataFrame by dropping the 'points' column
new_df = old_df.drop('points', axis=1)

# Display the resulting DataFrame
print(new_df)

team assists rebounds
0 A 5 11
1 A 7 8
2 A 7 10
3 A 9 6
4 B 12 6
5 B 9 7
6 B 9 9
7 B 4 12

# Confirm the data type
type(new_df)

pandas.core.frame.DataFrame

As clearly demonstrated by the output, `new_df` now includes all columns from the original dataset *except* for the designated **points** column, confirming the successful application of the `.drop()` method for column exclusion.

Ensuring Data Integrity: Understanding Views, Copies, and `SettingWithCopyWarning`

A core concept that must be grasped when manipulating data structures in Pandas is the subtle but significant difference between a **"view"** and a **"copy."** When you subset or index a DataFrame, Pandas sometimes returns a view--a new object that merely references the data stored in the original DataFrame's memory location. If you attempt to modify data within this "view," those changes propagate back and inadvertently overwrite the data in the source DataFrame, leading to unpredictable results and frustrating debugging sessions.

This ambiguity in memory management is precisely what triggers the infamous SettingWithCopyWarning. This warning is Pandas' mechanism for alerting the user that they might be attempting to assign a value to a derived object that could potentially be a view, thus risking modification of the original data. While the warning does not always indicate an error, it serves as a

strong signal for the need for defensive programming practices, especially when chaining multiple operations together.

To eliminate this risk entirely and ensure that all new DataFrames are completely decoupled from their source, the explicit use of the `.copy()` method is essential for selection-based subsetting. When you execute `new_df = old_df].copy()`, you are forcing a deep copy, meaning a completely new block of memory is allocated for `new_df`. Any subsequent modifications made to `new_df`, such as updating cell values or adding new columns, will only affect the new structure, leaving the original `old_df` pristine and unaltered. This practice guarantees predictable behavior and robust data pipelines in Python.

Conclusion: Mastering DataFrame Manipulation

The ability to confidently and accurately create independent DataFrames from existing data structures is a cornerstone of effective data analysis using Pandas. We have thoroughly examined three highly practical methods: using double-bracket indexing for selecting multiple columns, employing the same method to isolate a single column while preserving DataFrame structure, and utilizing the powerful `.drop()` function for efficient column exclusion. Each technique provides targeted control over the data structure, allowing analysts to tailor datasets precisely to their immediate analytical needs.

The single most important takeaway from this guide is the non-negotiable requirement of using the `.copy()` method when performing selection-based subsetting. This defensive coding measure ensures absolute data independence, shielding your original DataFrame from any unintended side effects or modifications that arise from working with a view. By embedding this best practice into your routine, you ensure the reliability and reproducibility of your data manipulation scripts.

As you grow more comfortable with these foundational techniques, we encourage you to explore more advanced methods for subsetting and filtering. Pandas offers sophisticated tools such as boolean indexing, which filters rows based on conditional criteria, and the expressive `.query()` method, which allows for SQL-like conditional selections. Mastering these capabilities will further enhance your ability to efficiently manage and prepare complex datasets for advanced statistical modeling or visualization in Python.

Further Learning Resources

To continue developing your skills in data science and manipulation, consider exploring the following related tutorials:

Tutorial on how to perform row-wise filtering in Pandas using boolean masks.
A guide to effectively merging and joining multiple DataFrames.

Understanding and handling missing data (NaN values) in Pandas.