Learning How to Select Numeric Columns in Pandas DataFrames

Authored by

Mohammed looti

October 28, 2025

Understanding the Need for Data Type Selection

When working with complex datasets, particularly within the pandas library, it is common to encounter a mixture of data types, including numerical values, categorical strings, dates, and boolean flags. Many critical data analysis tasks, such as statistical modeling, correlation analysis, or aggregation operations, require input data to be strictly numerical. Attempting to run these operations on columns containing non-numeric data, often represented as **object** types, will typically result in errors or inaccurate results. Therefore, the ability to efficiently and reliably filter a pandas DataFrame to isolate only the numerical columns is an essential skill for any data scientist or analyst. This segregation process ensures data integrity and operational efficiency, allowing complex calculations to proceed smoothly without manual column selection, which can be tedious and error-prone in wide datasets.

The challenge often lies in maintaining dynamic code that adapts to changes in the underlying dataset schema. If column names shift or new non-numeric features are added, hard-coded column lists will break the data pipeline. This necessity drove the development of specialized methods within pandas designed specifically for automatic introspection and filtering based on the column's data type, or **dtype**. The method we will explore, select_dtypes, provides a robust, programmatic solution to this problem, leveraging the powerful type-handling capabilities of both pandas and NumPy. By using this function, analysts can create resilient workflows that automatically identify and utilize all numerical features, regardless of the specific column naming conventions employed in the source data, thereby significantly increasing the efficiency of data preparation.

The Fundamental Syntax of `select_dtypes`

The primary mechanism for filtering columns based on their type in pandas is the `select_dtypes` method. This function is extremely versatile, allowing users to include or exclude columns based on their designated data types. To specifically target numeric columns, we must utilize the `include` parameter, passing it the appropriate definition of what constitutes a 'number' within the Python/NumPy ecosystem. This definition is typically accessed through the NumPy library, which provides the highly convenient classification `np.number`. This classification encompasses all standard numeric types recognized by NumPy, such as integers (e.g., **int64**, **int32**) and floating-point numbers (e.g., **float64**).

The following basic syntax outlines the required setup and function call necessary to execute this selection. It begins by ensuring the necessary libraries--pandas for data structure management and NumPy for type definitions--are imported into the environment, establishing the foundation for data manipulation. Following the imports, the method is applied directly to the existing DataFrame, instructing pandas to return a new DataFrame composed exclusively of columns that adhere to the

specified numeric criteria defined by `np.number`.

import pandas as pd
import numpy as np

df.select_dtypes(include=np.number)

It is crucial to understand that `select_dtypes` returns a copy of the subsetted data, not a view, ensuring that subsequent operations performed on the result DataFrame do not inadvertently modify the original data structure. Furthermore, the use of `np.number` is preferred over listing individual numeric types (like 'int' or 'float') because it provides a comprehensive and future-proof way to capture all numeric variations, including complex numbers or custom numeric extensions that might be introduced in later versions of the libraries, thereby maximizing the robustness and longevity of the selection process within large-scale analytical frameworks.

Practical Example: Isolating Numeric Data in a DataFrame

To illustrate the power and simplicity of this technique, let us construct a sample pandas DataFrame representing statistics for several basketball players. This dataset deliberately includes a mix of data types: numerical columns (points, assists, rebounds) suitable for mathematical analysis, and a categorical column (team) represented by strings, which is not. This mixed structure is typical of real-world data and provides an ideal scenario for demonstrating the efficacy of `select_dtypes` in isolating the quantitative features necessary for performance analysis.

We initialize the DataFrame below, ensuring clarity in how the data structure is created using dictionaries and the `pd.DataFrame` constructor. The output clearly shows the initial structure, where the 'team' column is non-numeric, while the performance metrics ('points', 'assists', 'rebounds') are numeric integers. This structure simulates the raw data often encountered after reading a CSV file, where column types are automatically inferred by pandas.

import pandas as pd

#create DataFrame
df = pd.DataFrame({'team': ,
'points': ,
'assists': ,
'rebounds': })

#view DataFrame
print(df)

team points assists rebounds
0 A 18 5 11
1 B 22 7 8
2 C 19 7 10
3 D 14 9 6
4 E 14 12 6
5 F 11 9 5
6 G 20 9 9
7 H 28 4 12

Having established our sample data, we can now apply the select_dtypes method. As demonstrated in the syntax section, we include the necessary NumPy import and then execute the function call, specifying `include=np.number`. This single line of code instructs pandas to scan the entire DataFrame, identify columns whose underlying data storage mechanism is numeric, and return a filtered result, excluding all others. The output immediately validates the successful isolation of the quantitative attributes, leaving behind the categorical 'team' column, which is essential for focused numerical analysis.

import numpy as np

#select only the numeric columns in the DataFrame
df.select_dtypes(include=np.number)

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

The result clearly shows that only the three numeric columns--**points**, **assists**, and **rebounds**--have been successfully selected and retained in the resulting DataFrame subset. This operation is fundamentally important for subsequent analytic steps, as it provides a clean, numerically homogeneous structure ready for statistical processing. For instance, if the goal was to compute the covariance matrix or perform principal component analysis (PCA), this subsetted DataFrame is now correctly formatted for those mathematically intensive operations without requiring manual

verification of column types.

Verifying Data Types Using the `dtypes` Attribute

While the visual output confirms that `select_dtypes` performed as expected, it is often necessary, especially when debugging complex data pipelines or ensuring strict adherence to schema requirements, to programmatically verify the data type (dtype) assigned to each column by pandas. Understanding these dtypes is paramount, as pandas uses specific NumPy designations to manage memory and performance efficiency. The `.dtypes` attribute provides a comprehensive Series detailing the inferred data type of every column in the original DataFrame, acting as a crucial diagnostic tool.

By calling this attribute on our original DataFrame `df`, we gain insight into why the filtering worked successfully. The non-numeric column, 'team', is identified as an object (dtype), which in pandas typically signifies string data or mixed types that cannot be easily cast into a single numerical format. Conversely, the performance columns are identified as int64, confirming their status as standard 64-bit integers, which falls squarely under the broad definition of `np.number` used in our selection criteria.

#display data type of each variable in DataFrame
df.dtypes

team object
points int64
assists int64
rebounds int64
dtype: object

The output confirms that **team** is indeed an object (representing a string), while **points**, **assists**, and **rebounds** are all numeric, specifically int64. This explicit type declaration is the reason our filtering method was successful; `select_dtypes(include=np.number)` accurately interpreted the int64 dtypes as numeric and excluded the non-numeric object type. Understanding the underlying dtype structure is fundamental to mastering data manipulation in pandas, particularly for operations reliant on type specificity, and serves as the backbone for efficient data preprocessing.

Advanced Application: Retrieving a List of Numeric Column Names

In certain scripting scenarios, particularly when automating tasks or dynamically generating feature lists for machine learning models, the goal is often not to retrieve the entire subset of data values, but rather to obtain a concise list of the column names that are numerical. This list can then be

utilized for further processing, such as iterating through columns for normalization or standardization, or for input into specific library functions that require an explicit list of feature names as strings. Fortunately, pandas is designed to facilitate method chaining, enabling us to combine `select_dtypes` with other DataFrame attributes to achieve this specialized output efficiently and in a single, readable line of code.

By chaining the `.columns` attribute and the `.tolist()` method directly onto the result of the select_dtypes operation, we effectively bypass the need to display the actual data values. First, `select_dtypes` returns the subsetted DataFrame containing only numeric columns. Second, the `.columns` attribute extracts the Index object containing the names of the columns in that numeric subset. Finally, `.tolist()` converts this Index object into a standard Python list containing only the string names of the identified features.

#display list of numeric variables in DataFrame
df.select_dtypes(include=np.number).columns.tolist()

This resulting list, , is highly valuable for programmatic tasks. It allows developers to quickly ascertain the names of the numeric variables in the pandas DataFrame without the overhead of rendering the entire data subset. This is particularly efficient when dealing with extremely large datasets where displaying the data itself would be time-consuming and unnecessary. This technique provides a powerful method for dataset introspection and preparation, adhering to best practices in automated data workflow design and ensuring that only relevant feature names are passed to downstream models.

Conclusion and Further Resources

The ability to accurately and dynamically select columns based on their data type is a cornerstone of effective data manipulation in pandas. The `select_dtypes` method, when paired with the robust type definitions provided by NumPy (specifically `np.number`), offers a superior solution compared to manual or hard-coded column selection. This approach ensures that data preparation pipelines are resilient, scalable, and adaptable to inevitable changes in the source data schema, thereby maximizing efficiency in complex analytical processes and reducing the likelihood of runtime errors.

By mastering the concepts demonstrated here--from basic syntax to programmatic verification using `.dtypes` and advanced list extraction--you are well-equipped to handle complex, mixed-type datasets with confidence. Integrating these methods into your standard workflow will streamline data cleaning and feature engineering, setting a solid foundation for advanced statistical analysis and subsequent machine learning applications.

For those interested in expanding their proficiency with pandas and data handling in Python, the following list of tutorials explains how to perform other common tasks crucial for comprehensive data preparation and analysis:

Tutorial on using the `exclude` parameter within `select_dtypes` to filter out specific non-numeric data types.

Guide to converting column data types (e.g., converting 'object' strings to numeric types after cleaning).

Explanation of handling missing values specific to numeric columns using imputation techniques.

Detailed analysis of various numeric dtypes available in NumPy and pandas and their memory implications.