Learning How to Create Categorical Variables in Pandas with Examples

Authored by

Mohammed looti

October 27, 2025

Working within the **Pandas** ecosystem, the creation and management of **categorical variables** are essential steps in effective data preparation and feature engineering. These specialized variables are crucial because they enable data practitioners to organize raw observations into distinct, manageable groups, which significantly simplifies **data analysis**, often boosts the performance of statistical models, and clarifies visualization efforts. This comprehensive guide details the two most common and effective methodologies for establishing categorical data structures within your **DataFrames**, providing foundational knowledge for data manipulation in **Python**.

The Importance of Categorical Data Types

Before diving into implementation, it is vital to understand why explicit categorical typing is beneficial. Unlike simple string or **object** **data types**, a Pandas categorical type stores data much more efficiently by mapping unique values to underlying integer codes, leading to substantial gains in **memory efficiency**, particularly in datasets with low cardinality (few unique categories). Furthermore, using the dedicated categorical type ensures that sorting and comparison operations are semantically correct, especially when dealing with ordered categories (e.g., 'Low', 'Medium', 'High').

Method 1: Creating a Categorical Variable from Scratch

This methodology is utilized when the requirement is to introduce a completely new column to a dataset where the category labels are predefined or explicitly assigned based on external knowledge. This is the most straightforward approach, involving the direct assignment of a list or array of category labels to a newly designated column within your **DataFrame**. It is often employed when initializing mock data or when importing data where the category status is already known and manually provided.

While the initial assignment might default the column to a string or **object** **data type**, the intent here is clear: the values represent discrete groups. The following code snippet demonstrates the basic syntax for this direct assignment method, where a list of category labels is used to populate a new column named `cat_variable`.

df =

Method 2: Deriving a Categorical Variable from an Existing Numerical Variable

A far more frequent data manipulation task involves converting continuous or discrete **numerical data** into distinct, ordered categories. This process, often referred to as **binning** or quantization,

transforms quantitative metrics into qualitative groups based on defined ranges. This conversion is highly effective for simplifying complex distributions, handling outliers, and preparing data for specific machine learning algorithms that perform better with discrete inputs.

The `pd.cut()` function in **Pandas** is the canonical tool for executing this transformation. It requires the user to specify explicit boundaries (`bins`) and corresponding names (`labels`) for the resultant categories. This allows for precise control over how the original numerical distribution is segmented into logical groups, thereby deriving a meaningful **categorical variable**.

df = pd.cut(df,
bins=,
labels=)

The subsequent examples will provide practical, executable demonstrations of these two powerful methods, illustrating how to seamlessly integrate them into typical **Pandas** data processing workflows.

Example 1: Implementing a Categorical Variable from Scratch

We begin by constructing a simple **DataFrame** in **Python** and directly incorporating a new categorical column. In this foundational scenario, we model data related to sports teams, where the `team` column naturally serves as our initial **categorical variable** and the `points` column represents a quantitative metric.

The following code initializes the DataFrame, ensuring that the `team` column is populated with textual labels and the `points` column contains **numerical data**. After initialization, we use the `df.dtypes` attribute, a standard diagnostic tool in **Pandas**, to confirm the automatically inferred **data types** assigned to each column, verifying that the structure aligns with our expectations.

import pandas as pd

# Create DataFrame with one categorical variable and one numeric variable
df = pd.DataFrame({'team': ,
'points': })

# View DataFrame
print(df)

team points
0 A 12
1 B 15

2 C 19
3 D 22
4 E 24
5 F 25
6 G 26
7 H 30

# View data type of each column in DataFrame
print(df.dtypes)

team object
points int64
dtype: object

The output of `df.dtypes` confirms the inherent structure: the `points` variable is correctly recognized as an integer (specifically `int64`), appropriate for quantitative metrics. Conversely, the `team` variable is identified as an **object** **data type**. In the context of Pandas, `object` is the default representation for textual data, often used to implicitly handle categorical or string columns. While this works for simple grouping, we will later discuss the best practice of explicitly converting this to the specialized `'category'` type for enhanced performance.

Example 2: Transforming Numerical Data into Categories using pd.cut()

This second example demonstrates the powerful technique of **binning**, converting the existing **numerical variable**, `points`, into a new, discrete **categorical variable** called `status`. This transformation allows us to categorize team performance into performance tiers (Bad, OK, Good) based on their score ranges, adding rich, interpretable context to the quantitative data.

We utilize the versatile **`pd.cut()`** function, specifying the `bins` (the numerical boundaries) and the `labels` (the names for the resultant categories). This function iterates through the `points` column of the **DataFrame** and assigns the appropriate category label based on which interval the point value falls into.

import pandas as pd

# Create DataFrame with one categorical variable and one numeric variable
df = pd.DataFrame({'team': ,
'points': })

# Create categorical variable 'status' based on existing numerical 'points' variable
df = pd.cut(df,

```
bins=,
labels=)

# View updated DataFrame
print(df)
```

```
team points status
0 A 12 Bad
1 B 15 Bad
2 C 19 OK
3 D 22 OK
4 E 24 OK
5 F 25 OK
6 G 26 Good
7 H 30 Good
```

The successful application of **`pd.cut()`** results in the `status` column, where categories are assigned according to the defined ranges: the `'Bad'` status is assigned to scores up to 15; `'OK'` covers scores between 15 (exclusive) and 25 (inclusive); and `'Good'` encompasses scores greater than 25, extending theoretically up to positive **infinity** (achieved using `float('Inf')`). This method of boundary definition ensures that every numerical point falls into exactly one category.

It is essential to strictly adhere to the structural requirement of the **`pd.cut()`** function: the number of `labels` must be exactly one less than the number of `bins`. For instance, providing four bin values (0, 15, 25, Inf) defines three discrete intervals. Consequently, we must supply three corresponding labels ('Bad', 'OK', 'Good') to name these resultant categories, ensuring a correct mapping for the new **categorical variable**.

Best Practices: Explicitly Setting the Category Data Type

While Pandas often infers categorical intent (or uses the `object` type), the best practice for robust data engineering is to explicitly convert the column's **data type** to `'category'`. This conversion is performed easily using the method `df.astype('category')`. This practice is strongly recommended for columns containing non-numerical data with a limited set of unique values.

The primary advantage of explicit categorization is enhanced **memory efficiency**. By storing categories as integer codes rather than repeating strings, Pandas significantly reduces the memory footprint, making operations faster, especially on very large datasets. Beyond resource optimization, the dedicated categorical type facilitates specialized statistical operations and ensures that downstream processes, such as advanced **data analysis** libraries or machine

learning pipelines, interpret the variable correctly as nominal or ordinal data, rather than treating it as free-form text.

Further Learning and Resources

To further advance your expertise in data manipulation and management, particularly within the **Pandas** library, we highly recommend consulting the official documentation. The official resources provide detailed explanations of all functions discussed here, including advanced usage of `pd.cut()` and other binning methods like `pd.qcut()`.

Additionally, exploring tutorials related to indexing, grouping, and aggregation techniques will solidify your ability to leverage the power of explicitly defined categorical variables for efficient and insightful **data analysis**.

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