# How to use pandas case\_when()

And other new features in pandas 2.2.0

### Authors:

Michael Walshe, Senior Consultant

### Tags:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Python | Pandas | Case\_when | DataFrame | Arrow |
| ADBC | How to | SQL | Data Analysis | NumPy |
|  |  |  |  |  |
|  |  |  |  |  |

### Description:

How to use the pandas case\_when function, along with a summary of other new features in pandas 2.2.0.

Pandas is the most popular Python package for data manipulation and analysis, providing a high-level tool for flexible manipulation of data in a tabular format. It just released version 2.2.0, the second minor release in the 2.0 series, and likely the last minor release before pandas 3.0 is scheduled to arrive in April!

Included are several new features, bug fixes, deprecations, and performance enhancements. I’m going to demonstrate a few of the most notable changes, but please go read the [release notes](https://pandas.pydata.org/docs/whatsnew/v2.2.0.html) for a full overview.

# A new Series method: Series.case\_when

## Overview

The first new feature is a new method for a Series, that allows you to replace values based on a set of conditions. This is the most exciting change in 2.2.0 for me, as it’s something that I and the community have been requesting for a long time (see various StackOverflow posts with over 2m collective views at [1]((https://stackoverflow.com/questions/49228596/pandas-case-when-default-in-pandas)), [2](https://stackoverflow.com/questions/26886653/create-new-column-based-on-values-from-other-columns-apply-a-function-of-multi), and [3](https://stackoverflow.com/questions/19913659/how-do-i-create-a-new-column-where-the-values-are-selected-based-on-existing-col)).

Previously, there were several different ways to conditionally create or alter columns in pandas: [np.select](https://numpy.org/doc/stable/reference/generated/numpy.select.html), [np.where](https://numpy.org/doc/stable/reference/generated/numpy.where.html), [Series.where](https://pandas.pydata.org/docs/reference/api/pandas.Series.where.html), [Series.mask](https://pandas.pydata.org/docs/reference/api/pandas.Series.mask.html), and of course [<Series/DataFrame>.loc](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.loc.html). All of these have their place, but for the use-case of checking a *series* of conditions and returning values they all had problems, from readability to requiring you to use functionality outside of pandas. A new pandas function that cleanly solves this common problem in data wrangling has been long overdue!

## Example

Let’s create an example dataframe:

import numpy as np  
import pandas as pd  
  
np.random.seed(42)  
  
df = pd.DataFrame(  
 {  
 "A": np.random.choice(list("ABC"), 50),  
 "B": np.random.choice(list("XYZ"), 50),  
 "C": np.random.random(50)\*2 - 1,  
 "D": pd.date\_range("2020-01-01", periods=50, freq="D"),  
 }  
)  
  
df.head()

|  | A | B | C | D |
| --- | --- | --- | --- | --- |
| 0 | C | Y | -0.602569 | 2020-01-01 |
| 1 | A | X | -0.988956 | 2020-01-02 |
| 2 | C | Y | 0.630923 | 2020-01-03 |
| 3 | C | X | 0.413715 | 2020-01-04 |
| 4 | A | Y | 0.458014 | 2020-01-05 |

Now we can create a new column in that dataframe based on an existing column using case\_when:

# The only argument to case\_when is a list of tuples  
# of the form: (condition, value)  
df["E"] = df["A"].case\_when(  
 [  
 (df["A"] == "A", "A is A"),  
 (df["B"] == "X", "B is X"),  
 ]  
)  
  
df[["A", "B", "E"]].head()

|  | A | B | E |
| --- | --- | --- | --- |
| 0 | C | Y | C |
| 1 | A | X | A is A |
| 2 | C | Y | C |
| 3 | C | X | B is X |
| 4 | A | Y | A is A |

Note two important things:

* The conditions are applied such that if a condition is True, then the remaining conditions are **ignored** (*Note: this isn’t how it’s implemented, so there are no performance improvements from “short-circuiting”*)
* The default values used (for when no conditions match) are the values from the original Series

We can also use Callables (a function) for the conditions or replacements, these will be passed the Series that case\_when is called on as an argument, for example:

df["E"] = df["A"].case\_when(  
 [  
 (df["D"].dt.is\_month\_start, lambda s: s + ": At month start"),  
 (lambda s: s == "A", df["A"] + "A")  
 ]  
)  
  
df[["A", "D", "E"]].head()

|  | A | D | E |
| --- | --- | --- | --- |
| 0 | C | 2020-01-01 | C: At month start |
| 1 | A | 2020-01-02 | AA |
| 2 | C | 2020-01-03 | C |
| 3 | C | 2020-01-04 | C |
| 4 | A | 2020-01-05 | AA |

As a final handy trick, if you want to use your own default value, you can either create a new constant Series to use as the input, or have a final condition in your caselist argument that is always true:

* Method 1:

df["E"] = pd.Series(np.nan, index=df.index).case\_when(  
 [  
 (df["A"] == "A", 0),  
 (df["C"] < 0.25, -df["C"]),  
 ]  
)  
  
df[["E"]].head()

|  | E |
| --- | --- |
| 0 | 0.602569 |
| 1 | 0.000000 |
| 2 | NaN |
| 3 | NaN |
| 4 | 0.000000 |

* Method 2:

df["E"] = df["C"].case\_when(  
 [  
 (df["A"] == "A", 0),  
 (df["C"] < 0.25, -df["C"]),  
 (pd.Series(True), np.nan)  
 ]  
)  
  
df[["E"]].head()

|  | E |
| --- | --- |
| 0 | 0.602569 |
| 1 | 0.000000 |
| 2 | NaN |
| 3 | NaN |
| 4 | 0.000000 |

# More performant database drivers with ADBC

## Overview

Pandas now supports [Arrow ADBC drivers](https://arrow.apache.org/adbc/current/index.html) when reading from or writing to a database. This leads to much better performance, better type handling, and is part of the general move to a pandas backed by Arrow as well as NumPy.

## Example

Here I’ll connect to a local SQLite database with SQLAlchemy, and using the ADBC drivers. As we’ll see, using these (not very scientific) benchmarks we get a huge performance improvement!

import timeit  
  
import adbc\_driver\_sqlite.dbapi  
import sqlalchemy as sa  
  
engine = sa.create\_engine("sqlite:///temp.db")  
  
with (  
 engine.connect() as conn1,  
 adbc\_driver\_sqlite.dbapi.connect("temp.db") as conn2,  
):  
 df = pd.DataFrame(  
 np.random.randint(10\_000, size=(100\_000, 10)), columns=list("abcdefghij")  
 )  
  
 print(  
 "Writing using the default driver: ",  
 timeit.timeit(lambda: df.to\_sql("TEST", conn1, if\_exists="replace"), number=10),  
 )  
 print(  
 "Writing using ADBC: ",  
 timeit.timeit(lambda: df.to\_sql("TEST", conn2, if\_exists="replace"), number=10),  
 )  
 print(  
 "Reading using the default driver: ",  
 timeit.timeit(lambda: pd.read\_sql("TEST", conn1), number=10),  
 )  
 print(  
 "Reading using ADBC: ",  
 timeit.timeit(lambda: pd.read\_sql("TEST", conn2), number=10),  
 )

Writing using the default driver: 8.174787900003139  
Writing using ADBC: 1.3331496000173502  
Reading using the default driver: 4.264690399984829  
Reading using ADBC: 0.5991979999816976

# Improved Functionality for Processing Structured Columns

## Overview

The final new feature I’ll mention is again part of the move to an Arrow backed pandas. There is now more functionality and support for nested PyArrow data, with the struct and list Series accessors. This makes working with columns that contain structured data (such as list columns or custom structures) much easier. There are only a few methods at the moment, but this could be the start of array & struct columns as first class citizens in pandas.

## Example

Often, you may receive structured data via JSON, such as the below:

from io import StringIO  
import pyarrow as pa  
  
raw\_data = StringIO(  
 """  
 [  
 {"A": [1, 2, 3],},  
 {"A": [4, 5]},  
 {"A": [6]}  
 ]  
 """  
)  
  
df = pd.read\_json(  
 raw\_data,  
 dtype=pd.ArrowDtype(pa.list\_(pa.int64())),  
)  
  
df.head()

|  | A |
| --- | --- |
| 0 | [1 2 3] |
| 1 | [4 5] |
| 2 | [6] |

We can now use built-in pandas methods to interact with these PyArrow structures:

* List Accessors:

df["A"].list[0]

0 1  
1 4  
2 6  
dtype: int64[pyarrow]

* List Functions:

df["A"].list.flatten()

0 1  
1 2  
2 3  
3 4  
4 5  
5 6  
dtype: int64[pyarrow]

That’s all for today, but check back in for our blog going over the key features and what to watch out for in pandas 3.0 when it lands!