✦ Member-only story

11-Hidden Pandas Functions You Didn't Know You Could Do with NumPy

Discover Little-Known Pandas Functions and Their NumPy Equivalents

Ajay Parmar · Follow

Published in Top Python Libraries · 10 min read · Sep 26, 2024

Image Created By Author

The whole idea behind building so many libraries for Python is to save developers from reinventing the wheel. Instead of coding the same small tasks repeatedly, developers created sets of functions specific to a task and placed them into libraries.

Now, we have tons of libraries designed to work efficiently with Python, reducing complexity and saving time. This helps **extend Python's capabilities**, encourages **standardization**, and **improves productivity**.

But here's the thing: while each library, like **Pandas** for data manipulation and analysis or **NumPy** for numerical computing, is specialized, not many people realize that these libraries can work together.

This blog is about exploring 5 Pandas functions that you can also use with NumPy, unlocking new ways to streamline your work. Stay tuned till the end for an insightful learning experience!

1. `pandas.Series.shift()`

The `pandas.Series.shift()` function shifts the values of a particular position in a Series up or down by a specified number of periods. For example, given the Series `[10, 20, 30, 40, 50]`, using `shift(1)` would result in `[NaN, 10, 20, 30, 40]`, making it ideal for time series analysis where you need to create lagged variables.

In NumPy, you can achieve a similar effect using `np.roll()`, which shifts the values in a column up or down by a specified number of periods.

Pandas: Shifts the values of a series by a certain number of periods.

NumPy Equivalent: You can use `np.roll()` to achieve a similar effect.

Example:

```
# Pandas
df['shifted'] = df['column'].shift(1)

# NumPy
df['shifted'] = np.roll(df['column'].values, 1)
```

2. `pandas.Series.diff()`

Alright, when we talk about `pandas.Series.diff()`, this particular function is used to calculate the difference between consecutive elements in a given series, like `[30, 32, 35, 31, 29]`. Applying `diff()` would yield `[NaN, 2, 3, -4, -2]`. This can be really useful for analyzing daily stock prices, temperature changes, and similar data.

Interestingly, you can achieve the same result with NumPy using `np.diff()`, which calculates the n-th discrete difference along the specified axis. So, while `diff()` is specifically designed for use with Pandas, NumPy offers a similar functionality that can also be very effective!

- **Pandas:** Computes the difference between subsequent elements.
- **NumPy Equivalent:** Use `np.diff()` for a similar result.
- **Example:**

```
# Pandas
df['difference'] = df['column'].diff()

# NumPy
df['difference'] = np.diff(df['column'], prepend=np.nan)
```

output

```
   column  difference  difference_np
0      30         NaN            NaN
1      32         2.0            2.0
2      35         3.0            3.0
3      31        -4.0           -4.0
4      29        -2.0           -2.0
```

3. `pandas.DataFrame.apply()`

See, by using `pandas.DataFrame.apply()`, you can add two rows and form a new one as an answer. This function is indeed a powerful tool that allows you to apply a function along a specified axis (rows or columns) of a

DataFrame. It's particularly useful for performing complex calculations or transformations on your data, like aggregating values or applying custom logic.

Now, in NumPy, you can achieve a similar functionality using `np.apply_along_axis()` , which applies a function to 1-D slices along the specified axis of an array. So, while both libraries have their own methods, they can tackle similar tasks depending on the structure of your data!

- **Pandas**: Apply a function to each row or column.
- **NumPy Equivalent**: Use `np.apply_along_axis()` for similar functionality on NumPy arrays.
- **Example**:

```
import pandas as pd
import numpy as np

# Sample DataFrame
data = {'a': [1, 2, 3], 'b': [4, 5, 6]}
df = pd.DataFrame(data)

# Using Pandas
df['result'] = df.apply(lambda row: row['a'] + row['b'], axis=1)

# Using NumPy
result = np.apply_along_axis(lambda row: row[0] + row[1], 1, df[['a', 'b']].valu

print(df)
print("NumPy Result:", result)
```

Output is:

```
   a  b  result
0  1  4       5
1  2  5       7
2  3  6       9
NumPy Result: [5 7 9]
```

4. `pandas.Series.rank()`

Oh, you might have seen the ranking of shares based on their value, right? This is often done using `pandas.Series.rank()`. The `pandas.Series.rank()` function computes the rank of each element in a Series, assigning ranks based on their position relative to other values. It's great for sorting data or assigning ranks in competitions, where the highest value is ranked first, the second-highest comes next, and so on.

Now, in NumPy, you can achieve a similar functionality using `np.argsort()`. The only catch is that you need to apply it twice. By applying `argsort()` twice, you can derive ranks for each element, although it doesn't handle ties or offer different ranking methods like Pandas does (e.g., average, min, max ranking methods). But it's still a clever way to emulate ranking in NumPy!

- **Pandas**: Computes the rank of each element in a Series.
- **NumPy Equivalent**: You can achieve similar functionality using `np.argsort()`.
- **Example**:

```
# Pandas
df['rank'] = df['column'].rank()
```

```
# NumPy
ranks = np.argsort(np.argsort(df['column'].values))
```

```
import pandas as pd
import numpy as np

# Sample DataFrame
data = {'column': [50, 20, 80, 60]}
df = pd.DataFrame(data)

# Using Pandas
df['rank'] = df['column'].rank()

# Using NumPy
ranks = np.argsort(np.argsort(df['column'].values))

df['numpy_rank'] = ranks + 1  # Adding 1 because ranks start from 1 in Pandas

print(df)
```

Output

```
   column  rank  numpy_rank
0      50   2.0           2
1      20   1.0           1
2      80   4.0           4
3      60   3.0           3
```

5. `pandas.Series.isin()`

You might have seen how companies, especially in e-commerce or AI detecting vehicle numbers, often need to check lists to identify things like product categories or vehicle addresses. In such cases, `pandas.Series.isin()`

comes in handy. It allows you to sort or filter out products or data based on specific categories, making the task much easier and faster.

As per the title, our goal is to achieve this using `np.in1d()` . The equivalent functionality in NumPy is provided by `np.in1d()` , which returns a boolean array indicating whether each element of the first array exists in the second array.

- **Pandas:** Check if elements in a Series are in a given list or array.
- **NumPy Equivalent:** Use `np.in1d()` to perform this task.
- **Example:**

```
import pandas as pd
import numpy as np

# Sample DataFrame
data = {
    'product': ['TV', 'Sofa', 'Laptop', 'Table', 'Shirt', 'Headphones', 'Shoes']
    'category': ['Electronics', 'Furniture', 'Electronics', 'Furniture', 'Clothi
}
df = pd.DataFrame(data)

# Categories to check
target_categories = ['Electronics', 'Furniture', 'Clothing']

# Using Pandas
df['is_in'] = df['category'].isin(target_categories)

# Using NumPy
df['is_in_np'] = np.in1d(df['category'], target_categories)

# Display the DataFrame
print(df)
```

Output:

```
      product    category  is_in  is_in_np
0          TV  Electronics   True      True
1        Sofa    Furniture   True      True
2      Laptop  Electronics   True      True
3       Table    Furniture   True      True
4       Shirt     Clothing   True      True
5  Headphones  Electronics   True      True
6       Shoes      Apparel  False     False
```

6. `pandas.Series.cumsum()`

The **pandas.Series.cumsum()** function you can use to calculate the cumulative sum. **Now, what is cumulative sum?** In simple words, it's the running total of a sequence of numbers, where each value in the result is the sum of itself and all previous values. Now, see the importance of this panda function; this calculation is super important in banking for calculating deposits based on cumulative sums.

Now, interestingly, you can achieve this by using NumPy too. In NumPy, you can achieve the same functionality with **np.cumsum()**, which also calculates the cumulative sum of the values in an array.

- **Pandas:** Calculates the cumulative sum of the values in a series.
- **NumPy Equivalent:** Use `np.cumsum()` for cumulative summation.
- **Example:**

```
import pandas as pd
import numpy as np

# Sample DataFrame
data = {'column': [1, 2, 3, 4, 5]}
df = pd.DataFrame(data)

# Using Pandas to calculate cumulative sum
df['cumsum_pandas'] = df['column'].cumsum()

# Using NumPy to calculate cumulative sum
df['cumsum_numpy'] = np.cumsum(df['column'].values)

print(df)
```

Output:

```
   column  cumsum_pandas  cumsum_numpy
0       1              1             1
1       2              3             3
2       3              6             6
3       4             10            10
4       5             15            15
```

7. `pandas.Series.expanding()`

Let's first understand the **expanding mean**: it is the running average that takes into account all the previous data points up to the current point. For example, the average of the first value [1] is 1.0, the average of the first two values [1, 2] is 1.5, and the average of the first three values [1, 2, 3] is 2.0, and so on for a sequence like [1, 2, 3, 4, 5].

Now, we can achieve this using the **pandas.Series.expanding()** function, which expands a window over the data to compute cumulative statistics like the running mean.

Interestingly, we can achieve this in **NumPy** as well, but with different steps. In NumPy, there isn't a single function like in pandas, so we have to break it down:

1. **np.cumsum(df['column'].values)** calculates the cumulative sum.
2. **np.arange(1, len(df) + 1)** generates an array representing the count of elements from 1 to the length of the DataFrame, allowing you to compute the expanding mean.

This combination gives the same effect as `expanding()` in pandas.

- **Pandas**: Expands a window over the data to compute cumulative statistics (like cumulative sum, mean, etc.).
- **NumPy Equivalent**: You can achieve this manually with `np.cumsum()` and computing the desired statistic over expanding windows.
- **Example**:

```
import pandas as pd
import numpy as np

# Sample DataFrame
data = {'column': [1, 2, 3, 4, 5]}
df = pd.DataFrame(data)

# Using Pandas to calculate the expanding mean
df['expanding_mean'] = df['column'].expanding().mean()

# Using NumPy to calculate the manual expanding mean
```

```python
expanding_mean = np.cumsum(df['column'].values) / np.arange(1, len(df) + 1)

# Adding the NumPy result to the DataFrame
df['expanding_mean_numpy'] = expanding_mean

# Display the DataFrame
print(df)
```

Output:

```
   column  expanding_mean  expanding_mean_numpy
0       1             1.0                   1.0
1       2             1.5                   1.5
2       3             2.0                   2.0
3       4             2.5                   2.5
4       5             3.0                   3.0
```

8. `pandas.Series.pct_change()`

Now see what kind of operation we can perform using **pandas.Series.pct_change()** in Pandas. This function calculates the percentage change between consecutive values. It is often useful when working with time series data, like stock prices or sales, to see how they have increased or decreased compared to the previous value.

In **NumPy**, we can achieve this by manually computing the difference between consecutive elements and dividing by the previous element to get the percentage change.

- **Pandas:** Computes the percentage change between the current and prior element.

- **NumPy Equivalent:** Use a combination of NumPy array operations to calculate the percentage change.
- **Example:**

```
import pandas as pd
import numpy as np

# Sample sales data
data = {'day': ['Day 1', 'Day 2', 'Day 3', 'Day 4', 'Day 5'],
        'sales': [100, 110, 150, 120, 130]}

# Create DataFrame
df = pd.DataFrame(data)

# Using Pandas to compute percentage change
df['pct_change_pandas'] = df['sales'].pct_change()

# Using NumPy to compute percentage change manually
pct_change_numpy = np.diff(df['sales'].values) / df['sales'].values[:-1]
pct_change_numpy = np.insert(pct_change_numpy, 0, np.nan)  # Insert NaN at the s

# Add the NumPy result to the DataFrame
df['pct_change_numpy'] = pct_change_numpy

# Display the DataFrame
print(df)
```

Output

```
     day  sales  pct_change_pandas  pct_change_numpy
0  Day 1    100                NaN               NaN
1  Day 2    110           0.100000          0.100000
2  Day 3    150           0.363636          0.363636
3  Day 4    120          -0.200000         -0.200000
4  Day 5    130           0.083333          0.083333
```

9. `pandas.Series.fillna()`

The `pandas.Series.fillna()` function is used to replace `NaN` (Not a Number) values in a Pandas Series with a specified value or method (e.g., filling forward, backward, or with a constant).

In **NumPy**, there isn't a direct equivalent, but you can achieve similar functionality using `np.where()` combined with `np.isnan()` to detect `NaN` values.

- **Pandas:** Fills `NaN` values with a specified value or method.
- **NumPy Equivalent:** Use `np.where()` to replace `NaN` values in a NumPy array.
- **Example:**

```
[100, 200, NaN, 300, NaN]
```

Using Pandas and NumPy

```
import pandas as pd
import numpy as np

# Sample data with NaN values
data = {'scores': [100, 200, np.nan, 300, np.nan]}

# Create DataFrame
df = pd.DataFrame(data)

# Using Pandas to fill NaN values
df['filled_pandas'] = df['scores'].fillna(0)
```

```
# Using NumPy to fill NaN values manually
df['filled_numpy'] = np.where(np.isnan(df['scores'].values), 0, df['scores'].val

# Display the DataFrame
print(df)
```

Output:

```
   scores  filled_pandas  filled_numpy
0   100.0          100.0         100.0
1   200.0          200.0         200.0
2     NaN            0.0           0.0
3   300.0          300.0         300.0
4     NaN            0.0           0.0
```

10. `pandas.DataFrame.dropna()`

The `pandas.DataFrame.dropna()` function removes rows (or columns) from a DataFrame that contain missing values (`NaN`). It's useful when you need to clean your data by removing incomplete rows or columns.

In **NumPy**, you can filter out rows containing `NaN` using `np.isnan()` combined with a logical NOT operation (`~`), which allows you to keep only the rows without `NaN` values.

- **Pandas:** Drops rows or columns with `NaN` values.
- **NumPy Equivalent:** Use `~np.isnan()` to filter out rows containing `NaN` .
- **Example:**

```
Scores:
  [100, 200, NaN, 300, 150]
```

Using Pandas and NumPy

```
import pandas as pd
import numpy as np

# Sample data with NaN values
data = {'scores': [100, 200, np.nan, 300, 150]}

# Create DataFrame
df = pd.DataFrame(data)

# Using Pandas to drop rows with NaN values
df_cleaned_pandas = df.dropna()

# Using NumPy to manually filter out rows with NaN values
df_cleaned_numpy = df[~np.isnan(df['scores'].values)]

# Display the results
print("Original DataFrame:\n", df)
print("\nPandas dropna():\n", df_cleaned_pandas)
print("\nNumPy equivalent:\n", df_cleaned_numpy)
```

Output

```
Original DataFrame:
    scores
0   100.0
1   200.0
2     NaN
3   300.0
4   150.0
```

```
Pandas dropna():
    scores
0   100.0
1   200.0
3   300.0
4   150.0

NumPy equivalent:
    scores
0   100.0
1   200.0
3   300.0
4   150.0
```

11. `pandas.Series.value_counts()`

The **pandas.Series.value_counts()** function is designed to quickly show out similar class values, like how many "a"s are in "abaacdad." It's popularly called a quick frequency analyzer.

In **NumPy**, you can achieve the same result using **np.unique()** with the **return_counts=True** argument, which returns the unique values along with their respective counts.

- **Pandas:** Counts the unique values in a series.
- **NumPy Equivalent:** Use `np.unique()` with `return_counts=True` to get unique values and their counts.
- **Example:**

```
Responses: ['A', 'B', 'A', 'C', 'B', 'A', 'B']
```

Using Pandas and NumPy

```python
import pandas as pd
import numpy as np

# Sample data
data = {'responses': ['A', 'B', 'A', 'C', 'B', 'A', 'B']}

# Create DataFrame
df = pd.DataFrame(data)

# Using Pandas to count unique values
value_counts_pandas = df['responses'].value_counts()

# Using NumPy to count unique values manually
unique, counts = np.unique(df['responses'].values, return_counts=True)
value_counts_numpy = dict(zip(unique, counts))  # Combine unique values and coun

# Display the results
print("Pandas value_counts():\n", value_counts_pandas)
print("\nNumPy equivalent:\n", value_counts_numpy)
```

Output:

```
Pandas value_counts():
 A    3
 B    3
 C    1
Name: responses, dtype: int64

NumPy equivalent:
 {'A': 3, 'B': 3, 'C': 1}
```

Summary of Additional Functions:

- **`pandas.Series.shift()`** → Use `np.roll()` for shifting values in a Series.
- **`pandas.Series.diff()`** → Use `np.diff()` for calculating differences between elements.
- **`pandas.DataFrame.apply()`** → Use `np.apply_along_axis()` for applying a function along an axis of a NumPy array.
- **`pandas.Series.rank()`** → Use `np.argsort()` to compute ranks based on the sorted order of elements.
- **`pandas.Series.isin()`** → Use `np.in1d()` for checking if elements are in a given list.
- **`pandas.Series.cumsum()`** → Use `np.cumsum()` for cumulative summation.
- **`pandas.Series.expanding()`** → Manual expansion with `np.cumsum()` and computation of statistics.
- **`pandas.Series.pct_change()`** → Manual calculation of percentage change with `np.diff()`.
- **`pandas.Series.fillna()`** → Use `np.where()` for filling NaN values.
- **`pandas.DataFrame.dropna()`** → Use `~np.isnan()` for dropping rows with NaN values.
- **`pandas.Series.value_counts()`** → Use `np.unique()` for counting unique values in a Series.

These NumPy equivalents allow you to handle similar tasks as in Pandas but with more control and speed in some cases, especially for numerical and array-based operations.

Hey there, I'm Ajay — a passionate engineer, writer on Medium, and a huge Python enthusiast. Thanks for sticking with me till the end! If you enjoyed this content and want to support my work, the best ways are:

- Drop a Clap 👏 and share your thoughts 💬 below!
- **Follow me here on Medium.**
- **Connect with me on LinkedIn.**
- Join my email list so you never miss another article.
- And don't forget to follow **Top Python Libraries Publication** for more stories like this.

Thanks again for reading! 😊

Python | Python Libraries | Numpy | Pandas | Python Programming

Published in Top Python Libraries

Follow

3.1K Followers · Last published 16 hours ago

Python is widely used in fields such as data analysis, machine learning, and web development. Sharing these skills will help you advance further in your career.https://join.slack.com/t/aidisruptiont-9307882/shared_invite/zt-2vb2pzkqq-oTJmcTR_v0AWgJ31ZPfOGA

Written by Ajay Parmar

Follow

505 Followers · 207 Following

Coding | Data Science | Tech Enthusiast | Python | SQL | ML | Founder of NewZolokiya | Youtube | Life | Learn & Teach 😊

Responses (1)

What are your thoughts?

Respond

Kyle Gilde, ML Data Scientist
Oct 27, 2024

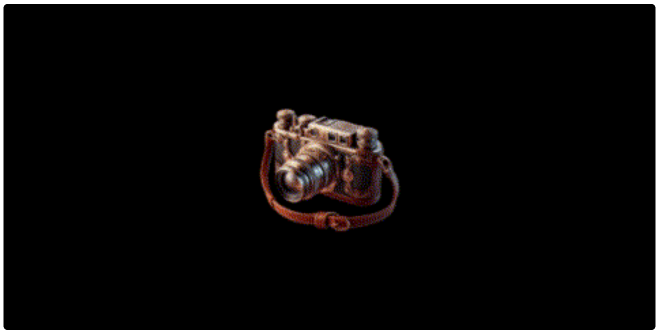
> NumPy Equivalent: Use np.where() to replace NaN values in a NumPy array.

Nan_to_num is faster and more elegant.

https://numpy.org/doc/stable/reference/generated/numpy.nan_to_num.html

7 Reply

More from Ajay Parmar and Top Python Libraries



In The Pythoneers by Ajay Parmar

I Found a Solution to Get Your Friend’s Wedding Photos Using...

Getting photos from a photographer can sometimes feel like a nightmare, especially...

Dec 28, 2024 52 1

In Top Python Libraries by Jenny Ouyang

What Is Hugging Face? A Complete Guide for Beginners

Discover how to leverage Hugging Face’s powerful AI platform for your projects, from...

Dec 30, 2024 319 2

In Top Python Libraries by Meng Li

Pythoncode-tutorials: All-in-One Python Tutorial Collection!

Explore Pythoncode-tutorials: Hundreds of Python guides from cybersecurity to AI, web...

Dec 25, 2024 44

In Top Python Libraries by Ajay Parmar

9-Python Library Projects to Automate Social Media Like a Pro

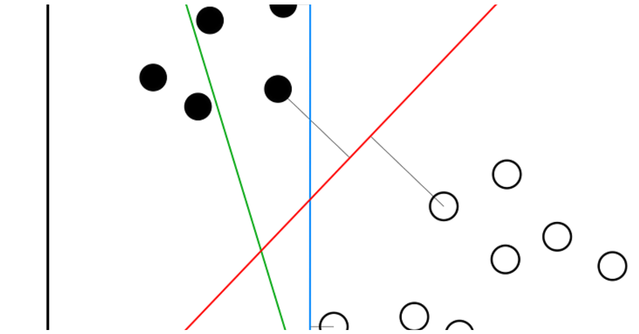
In this article, we’re addressing 9 social media problems and how Python can help solve...

Dec 24, 2024 7

See all from Ajay Parmar

See all from Top Python Libraries

Recommended from Medium



AI In Artificial Intelligence in Plain En... by Ritesh Gu...

Data Science All Algorithm Cheatsheet 2025

Stories, strategies, and secrets to choosing the perfect algorithm.

Jan 5 · 502 · 9

Yash

Think Python Is Slow? Try These Hacks for 3x Faster Scripts Today

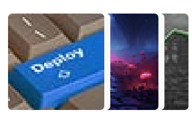
Why investing time in optimization pays off big

Jan 16 · 347 · 5

Lists

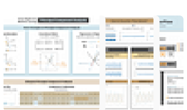
Coding & Development

11 stories · 983 saves



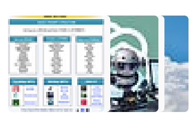
Predictive Modeling w/ Python

20 stories · 1792 saves



Practical Guides to Machine Learning

10 stories · 2171 saves



ChatGPT

21 stories · 952 saves

Tomer Gabay

How to Setup Your Macbook for Data Science in 2025

Easy Steps to Get the Best Experience From Your MacBook as a Data Scientist

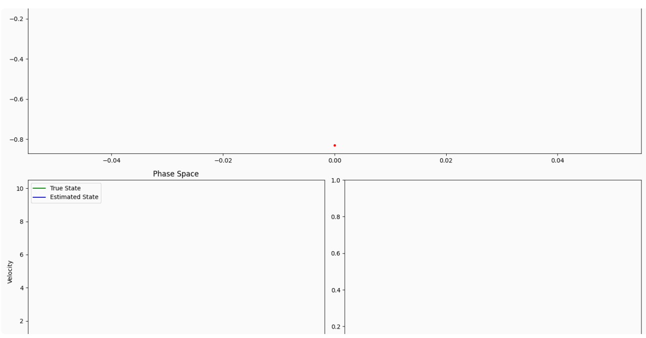
Dec 28, 2024 · 275 · 2

Chris Ferrie

If You Still Don’t Understand the Monty Hall Paradox, You Never Will

You are presented with three doors and know that behind one of the doors is a new car,...

6d ago · 1.4K · 45



Kyle Jones

State Space Models and Kalman Filtering for Time Series Analysis

Techniques for understanding the hidden states of time series data

6d ago · 164 · 3

$$\binom{n}{r} + \binom{n}{r+1} = \binom{n+1}{r+1}$$

Keith McNulty

An Introduction to one of the Toughest Disciplines in...

Combinatorics—the discipline of counting—is one of the hardest tests of mathematical...

5d ago · 962 · 21

See more recommendations