😊 Text Methods

Text Methods in Pandas (Python)

Text methods in Pandas refer to a set of built-in string functions that can be applied to columns containing text (string or object types). These methods are accessed using the .str accessor, which allows you to perform vectorized string operations efficiently across an entire column.

Why Use Text Methods?

In data analysis, it's common to work with columns that contain names, categories, addresses, or other textual information. Text methods in Pandas allow you to clean, transform, search, and manipulate such text data without using loops, making operations faster and more readable.

Categories of Text Methods

1. Case Conversion

- lower() – Converts all characters to lowercase.
- upper() – Converts all characters to uppercase.
- title() – Converts to title case (first letter capital).
- capitalize() – Capitalizes the first character.
- swapcase() – Swaps lowercase to uppercase and vice versa.

2. Whitespace Handling

- strip() – Removes leading and trailing whitespaces.
- lstrip() / rstrip() – Removes whitespaces from left or right only.

3. Searching and Matching

- contains() – Checks if each string contains a pattern or substring.
- startswith() / endswith() – Checks if strings start or end with a specific substring.
- find() / rfind() – Finds the position of the first or last occurrence of a substring.
- count() – Counts how many times a substring appears.

4. Replacing and Splitting

- replace() – Replaces substrings or patterns with another value.
- split() – Splits strings based on a delimiter.
- rsplit() – Splits from the right.
- partition() / rpartition() – Splits into three parts around a separator.

5. Combining and Padding

- cat() – Concatenates strings (usually used with multiple columns).

- pad() – Pads strings with a character.
- ljust() / rjust() / center() – Left, right, or center aligns the string within a given width.
- zfill() – Pads with zeros on the left.

6. Information and Checks

- len() – Gets length of each string.
- isnumeric(), isdigit(), isalpha(), isalnum(), isspace() – Checks for string properties.
- isupper(), islower(), istitle() – Checks for case style.

7. Regular Expressions

Most of the methods like contains(), replace(), and extract() support regular expressions for advanced pattern matching.

Advantages of Using .str Methods

- Vectorized: Operates on entire columns efficiently.
- Consistent: Syntax is similar to Python's native string methods.
- Flexible: Supports regular expressions and broadcasting.
- Robust: Handles missing (NaN) values gracefully in most cases.

Summary

Text methods in Pandas provide a robust set of tools for processing and analyzing text data. They are essential for tasks like data cleaning, formatting, feature extraction from strings, and preparing textual fields for analysis or machine learning models

😄 To begin with the Lab

1. A normal Python string has a variety of method calls available. The code below demonstrates basic string methods in Python.
2. 'hello'.capitalize() returns 'Hello', capitalizing the first letter.
3. 'hello'.isdigit() checks if the string contains only digits, returning False.
4. help(str) displays documentation for all string methods. These functions help manipulate and inspect string values easily.

```
[4]: mystring = 'hello'

[6]: mystring.capitalize()

[6]: 'Hello'

[8]: mystring.isdigit()

[8]: False

[10]: help(str)
```

```
|  __mod__(self, value, /)
|      Return self%value.
|
|  __mul__(self, value, /)
|      Return self*value.
|
|  __ne__(self, value, /)
|      Return self!=value.
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
```

5. The code creates a Pandas Series called names containing a list of strings.
6. Using names.str.capitalize(), it capitalizes the first letter of each string in the Series.
7. The .str accessor allows string functions to be applied element-wise.
8. Then, names.str.isdigit() checks if each string is made up entirely of digits, returning True or False for each item.
9. This is useful for transforming or validating text data in a DataFrame or Series.

```
[13]: import pandas as pd

[15]: names = pd.Series(['andrew','bobo','claire','david','4'])

[17]: names

[17]: 0    andrew
      1      bobo
      2    claire
      3     david
      4         4
      dtype: object

[19]: names.str.capitalize()

[19]: 0    Andrew
      1      Bobo
      2    Claire
      3     David
      4         4
      dtype: object

[21]: names.str.isdigit()

[21]: 0    False
      1    False
      2    False
      3    False
      4     True
      dtype: bool
```

10. The code defines a list `tech_finance` containing two comma-separated strings of stock tickers and converts it into a Pandas Series called `tickers`.
11. `tickers.str.split(',')` splits each string into a list using the comma as a delimiter.
12. `tickers.str.split(',').str[0]` retrieves the first ticker from each list.
13. `tickers.str.split(',', expand=True)` splits the strings and expands the results into separate columns.
14. The `.str.split()` method is useful for breaking apart delimited text, and `expand=True` turns the output into a DataFrame for easier access to individual values.

```
[24]: tech_finance = ['GOOG,APPL,AMZN','JPM,BAC,GS']

[26]: len(tech_finance)

[26]: 2

[28]: tickers = pd.Series(tech_finance)

[30]: tickers

[30]: 0    GOOG,APPL,AMZN
      1        JPM,BAC,GS
      dtype: object

[32]: tickers.str.split(',')

[32]: 0    [GOOG, APPL, AMZN]
      1        [JPM, BAC, GS]
      dtype: object

[34]: tickers.str.split(',').str[0]

[34]: 0    GOOG
      1     JPM
      dtype: object

[36]: tickers.str.split(',',expand=True)
```

[36]:

	0	1	2
0	GOOG	APPL	AMZN
1	JPM	BAC	GS

15. The code creates a Pandas Series `messy_names` with names that include extra spaces and a semicolon.
16. `str.replace(";", "")` removes semicolons, and `str.strip()` removes leading and trailing spaces.
17. Chaining these, `str.replace(";","").str.strip()` cleans the names.
18. Finally, adding `str.capitalize()` capitalizes the first letter of each cleaned name. These string methods, used with the `.str` accessor, allow element-wise text cleaning and formatting within a Series.

[39]:
```
messy_names = pd.Series(["andrew  ","bo;bo","  claire  "])
```

[41]:
```
# Notice the "mis-alignment" on the right hand side due to spacing in "andrew  " and "  claire  "
messy_names
```

[41]:
```
0      andrew
1         bo;bo
2      claire
dtype: object
```

[43]:
```
messy_names.str.replace(";","")
```

[43]:
```
0      andrew
1          bobo
2      claire
dtype: object
```

[45]:
```
messy_names.str.strip()
```

[45]:
```
0    andrew
1     bo;bo
2    claire
dtype: object
```

[47]:
```
messy_names.str.replace(";","").str.strip()
```

[47]:
```
0    andrew
1      bobo
2    claire
dtype: object
```

[49]:
```
messy_names.str.replace(";","").str.strip().str.capitalize()
```

[49]:
```
0    Andrew
1      Bobo
2    Claire
dtype: object
```

19. The code defines a function `cleanup` to remove semicolons, trim spaces, and capitalize names.
20. `messy_names.apply(cleanup)` applies this function to each element in the `messy_names` Series.
21. The `.apply()` method lets you use custom functions element-wise on a Pandas Series for flexible data cleaning or transformation.

[52]:
```
def cleanup(name):
    name = name.replace(";","")
    name = name.strip()
    name = name.capitalize()
    return name
```

[54]:
```
messy_names
```

[54]:
```
0      andrew
1         bo;bo
2      claire
dtype: object
```

[56]:
```
messy_names.apply(cleanup)
```

[56]:
```
0    Andrew
1      Bobo
2    Claire
dtype: object
```

22. This code compares the execution time of three different methods to clean up a Pandas Series of messy names using Python's timeit module. The setup block initializes the Series and defines the cleanup() function. The three test statements are:
    - **stmt_pandas_str**: Uses Pandas string methods (str.replace, str.strip, str.capitalize) chained together — this is vectorized and typically very fast.
    - **stmt_pandas_apply**: Uses .apply(cleanup) to apply a custom function element-wise — slower than vectorized methods.
    - **stmt_pandas_vectorize**: Uses np.vectorize(cleanup) to apply the function — technically wraps a loop and is usually similar in speed to .apply.
23. This benchmark shows which method is fastest for text cleaning tasks. Usually, vectorized Pandas string operations are the most efficient.

[59]:
```
import timeit

# code snippet to be executed only once
setup = '''
import pandas as pd
import numpy as np
messy_names = pd.Series(["andrew  ","bo;bo","  claire  "])
def cleanup(name):
    name = name.replace(";","")
    name = name.strip()
    name = name.capitalize()
    return name
'''

# code snippet whose execution time is to be measured
stmt_pandas_str = '''
messy_names.str.replace(";","").str.strip().str.capitalize()
'''

stmt_pandas_apply = '''
messy_names.apply(cleanup)
'''

stmt_pandas_vectorize='''
np.vectorize(cleanup)(messy_names)
'''
```

[61]:
```
timeit.timeit(setup = setup,
                    stmt = stmt_pandas_str,
                    number = 10000)
```

[61]:
```
2.5497709000001123
```

[62]:
```
timeit.timeit(setup = setup,
                    stmt = stmt_pandas_apply,
                    number = 10000)
```

[62]:
```
0.9469467999997505
```

[63]:
```
timeit.timeit(setup = setup,
                    stmt = stmt_pandas_vectorize,
                    number = 10000)
```

[63]:
```
0.2030239999994592
```