```
In [1]: import pandas as pd
```

Data used for this tutorial:

Titanic data

```
In [2]: titanic = pd.read_csv("data/titanic.csv")

In [3]: titanic.head()
Out[3]:
   PassengerId  Survived  Pclass  ...     Fare Cabin  Embarked
0            1         0       3  ...   7.2500   NaN         S
1            2         1       1  ...  71.2833   C85         C
2            3         1       3  ...   7.9250   NaN         S
3            4         1       1  ...  53.1000  C123         S
4            5         0       3  ...   8.0500   NaN         S

[5 rows x 12 columns]
```

How to manipulate textual data

Make all name characters lowercase.

```
In [4]: titanic["Name"].str.lower()
Out[4]:
0                                braund, mr. owen harris
1      cumings, mrs. john bradley (florence briggs th...
2                                 heikkinen, miss. laina
3           futrelle, mrs. jacques heath (lily may peel)
4                               allen, mr. william henry
                             ...
886                               montvila, rev. juozas
887                        graham, miss. margaret edith
888            johnston, miss. catherine helen "carrie"
889                               behr, mr. karl howell
890                                 dooley, mr. patrick
Name: Name, Length: 891, dtype: object
```

To make each of the strings in the `Name` column lowercase, select the `Name` column (see the tutorial on selection of data), add the `str` accessor and apply the `lower` method. As such, each of the strings is converted element-wise.

Similar to datetime objects in the time series tutorial having a `dt` accessor, a number of specialized string methods are available when using the `str` accessor. These methods have in general matching names with the equivalent built-in string methods for single elements, but are applied element-wise (remember element-wise calculations?) on each of the values of the columns.

Create a new column `Surname` that contains the surname of the passengers by extracting the part before the comma.

```
In [5]: titanic["Name"].str.split(",")
Out[5]:
```

```
2                              [Heikkinen,  Miss. Laina]
3        [Futrelle,  Mrs. Jacques Heath (Lily May Peel)]
4                            [Allen,  Mr. William Henry]
                             ...
886                             [Montvila,  Rev. Juozas]
887                      [Graham,  Miss. Margaret Edith]
888          [Johnston,  Miss. Catherine Helen "Carrie"]
889                              [Behr,  Mr. Karl Howell]
890                                [Dooley,  Mr. Patrick]
Name: Name, Length: 891, dtype: object
```

Using the `Series.str.split()` method, each of the values is returned as a list of 2 elements. The first element is the part before the comma and the second element is the part after the comma.

```
In [6]: titanic["Surname"] = titanic["Name"].str.split(",").str.get(0)

In [7]: titanic["Surname"]
Out[7]:
0         Braund
1        Cumings
2      Heikkinen
3       Futrelle
4          Allen
         ...
886     Montvila
887       Graham
888     Johnston
889         Behr
890       Dooley
Name: Surname, Length: 891, dtype: object
```

As we are only interested in the first part representing the surname (element 0), we can again use the `str` accessor and apply `Series.str.get()` to extract the relevant part. Indeed, these string functions can be concatenated to combine multiple functions at once!

To user guide More information on extracting parts of strings is available in the user guide section on splitting and replacing strings.

Extract the passenger data about the countesses on board of the Titanic.

```
In [8]: titanic["Name"].str.contains("Countess")
Out[8]:
0      False
1      False
2      False
3      False
4      False
       ...
886    False
887    False
888    False
889    False
890    False
Name: Name, Length: 891, dtype: bool
```

```
In [9]: titanic[titanic["Name"].str.contains("Countess")]
Out[9]:
     PassengerId  Survived  Pclass  ... Cabin Embarked  Surname
759          760         1       1  ...   B77        S   Rothes

[1 rows x 13 columns]
```

(*Interested in her story? See Wikipedia!*)

The string method `Series.str.contains()` checks for each of the values in the column `Name` if the string contains the word `Countess` and returns for each of the values `True` (`Countess` is part of the name) or `False` (`Countess` is not part of the name). This output can be used to subselect the data using conditional (boolean) indexing introduced in the subsetting of data tutorial. As there was only one countess on the Titanic, we get one row as a result.

Note

More powerful extractions on strings are supported, as the `Series.str.contains()` and `Series.str.extract()` methods accept regular expressions, but out of scope of this tutorial.

To user guide More information on extracting parts of strings is available in the user guide section on string matching and extracting.

Which passenger of the Titanic has the longest name?

```
In [10]: titanic["Name"].str.len()
Out[10]:
0      23
1      51
2      22
3      44
4      24
       ..
886    21
887    28
888    40
889    21
890    19
Name: Name, Length: 891, dtype: int64
```

To get the longest name we first have to get the lengths of each of the names in the `Name` column. By using pandas string methods, the `Series.str.len()` function is applied to each of the names individually (element-wise).

```
In [11]: titanic["Name"].str.len().idxmax()
Out[11]: 307
```

Next, we need to get the corresponding location, preferably the index label, in the table for which the name length is the largest. The `idxmax()` method does exactly that. It is not a string method and is applied to integers, so no `str` is used.

```
In [12]: titanic.loc[titanic["Name"].str.len().idxmax(), "Name"]
Out[12]: 'Penasco y Castellana, Mrs. Victor de Satode (Maria Josefa Perez de Soto y Vallejo)'
```

Based on the index name of the row (`307`) and the column (`Name`), we can do a selection using the `loc` operator, introduced in the tutorial on subsetting.

In the “Sex” column, replace values of “male” by “M” and values of “female” by “F”.

```
In [13]: titanic["Sex_short"] = titanic["Sex"].replace({"male": "M", "female": "F"})

In [14]: titanic["Sex_short"]
```

```
1      F
2      F
3      F
4      M
      ..
886    M
887    F
888    F
889    M
890    M
Name: Sex_short, Length: 891, dtype: object
```

Whereas `replace()` is not a string method, it provides a convenient way to use mappings or vocabularies to translate certain values. It requires a `dictionary` to define the mapping `{from : to}`.

Warning

There is also a `replace()` method available to replace a specific set of characters. However, when having a mapping of multiple values, this would become:

```
titanic["Sex_short"] = titanic["Sex"].str.replace("female", "F")
titanic["Sex_short"] = titanic["Sex_short"].str.replace("male", "M")
```

This would become cumbersome and easily lead to mistakes. Just think (or try out yourself) what would happen if those two statements are applied in the opposite order...

REMEMBER

- String methods are available using the `str` accessor.
- String methods work element-wise and can be used for conditional indexing.
- The `replace` method is a convenient method to convert values according to a given dictionary.

To user guide A full overview is provided in the user guide pages on working with text data.

Previous Next