Introduction

Similarly to the `dt` accessor that can handle dates, a `Series` containing strings can be managed using the `str` accessor.

In [1]:
```
import pandas as pd
df = pd.DataFrame({'text': ['aCag', '53Bc^', 'cc', '/c_8cd45', 'F98', '__m'],
                   'other column': range(6)})
df
```

Out[1]:

	text	other column
0	aCag	0
1	53Bc^	1
2	cc	2
3	/c_8cd45	3
4	F98	4
5	__m	5

Existing methods

Below are presented methods and attributes of the `str` accessor

In [2]:
```
# listing of attributes and methods of object dt
for attr in dir(df['text'].str):
    if not attr.startswith('_'):
        print(attr, end=' / ')
```

```
capitalize / casefold / cat / center / contains / count / decode / encode /
endswith / extract / extractall / find / findall / fullmatch / get / get_dum
mies / index / isalnum / isalpha / isdecimal / isdigit / islower / isnumeric
/ isspace / istitle / isupper / join / len / ljust / lower / lstrip / match
/ normalize / pad / partition / removeprefix / removesuffix / repeat / repla
ce / rfind / rindex / rjust / rpartition / rsplit / rstrip / slice / slice_r
eplace / split / startswith / strip / swapcase / title / translate / upper /
wrap / zfill /
```

Many of them also exist with the native Python `str` type. In other words, what you can do with a Python string can be done at large scale on a `pandas` Series containing strings:

In [3]:
```
print(*[attr for attr in dir(str) if not attr.startswith('_')], sep=' / ')
```

```
capitalize / casefold / center / count / encode / endswith / expandtabs / fi
nd / format / format_map / index / isalnum / isalpha / isascii / isdecimal /
isdigit / isidentifier / islower / isnumeric / isprintable / isspace / istit
le / isupper / join / ljust / lower / lstrip / maketrans / partition / remov
eprefix / removesuffix / replace / rfind / rindex / rjust / rpartition / rsp
lit / rstrip / split / splitlines / startswith / strip / swapcase / title /
translate / upper / zfill
```

Simple examples

In [4]:
```
df['text'].str.lower()   # lower case
```

Out[4]:
```
0        acag
1       53bc^
2          cc
3    /c_8cd45
4         f98
5          __m
Name: text, dtype: object
```

In [5]:
```
df['text'].str.len()   # length
```

Out[5]:
```
0    4
1    5
2    2
3    8
4    3
5    3
Name: text, dtype: int64
```

Indexing

In [6]:
```
df['text'].str[2:4]
```

Out[6]:
```
0    ag
1    Bc
2
3    _8
4     8
5     m
Name: text, dtype: object
```

Splitting

Splitting means building different strings by cutting the original one at the location of a special character. Below, the splitting operation results in the substrings being stored in a list, for each row of the Series.

In [7]:
```
df['text'].str.split('c')
```

Out[7]:
```
0          [aCag]
1        [53B, ^]
2          [, , ]
3    [/, _8, d45]
4           [F98]
5           [__m]
Name: text, dtype: object
```

The `expand` argument makes it possible to get distinct columns.

In [8]:
```
df['text'].str.split('c', expand=True)
```

Out[8]:

	0	1	2
0	aCag	None	None
1	53B	^	None
2			
3	/	_8	d45
4	F98	None	None
5	__m	None	None

Suffixes and prefixes

In [9]:
```
df['text'].str.startswith('53')
```

Out[9]:
```
0    False
1     True
2    False
3    False
4    False
5    False
Name: text, dtype: bool
```

In [10]:
```
df['text'].str.endswith('m')
```

Out[10]:
```
0    False
1    False
2    False
3    False
4    False
5     True
Name: text, dtype: bool
```

Advanced: *regex*

Introduction

The ***regex*** word means *regular expression*. A regex is a group of characters built in a very specific order in order to describe a generic type of strings.

Regex can be used on very large amount of data to detect some strings with a particular meaning.

Documentation of the Python version of regex is accessible here.

Regex is a difficult notion of computer engineering. A very simple case is presented here after.

Case study definition

Let's suppose one has some experimental values coming from different sensors.

In [11]:
```
import numpy as np
npr = np.random.default_rng(42)
# `npr.choice` randomly takes 10 values from a certain iterable
sensors = npr.choice(('AB-45-PL', 'AB-46-KL', 'AB-47-KL', 'AB-48-KL', 'ZB-76-PM', '87
values = range(len(sensors))
df = pd.DataFrame({'sensors': sensors, 'values': values})
df
```

Out[11]:

	sensors	values
0	AB-45-PL	0
1	ZB-76-PM	1
2	AB-48-KL	2
3	AB-47-KL	3
4	AB-47-KL	4
5	87-PA-98	5
6	AB-45-PL	6
7	ZB-76-PM	7
8	AB-46-KL	8
9	AB-45-PL	9

Question 1

How to access all sensors whose name contains both an 'A' and a '7'?

Solution 1

Let's use the `str.contains` method, 2 times. We use the `&` (and) operator to assemble the two conditions.

In [12]:
```
cond1 = df['sensors'].str.contains('A', regex=False)
cond2 = df['sensors'].str.contains('7', regex=False)
df[cond1 & cond2]
```

Out[12]:

	sensors	values
3	AB-47-KL	3
4	AB-47-KL	4
5	87-PA-98	5

Question 2

How to get all sensors whose name contains:

- either 'A' or 'Z'
- then '7'

Solution 2

Let's build a regex pattern:

In [13]:
```
pattern = '.*(A|Z).*7.*[a-zA-Z]'
df[df['sensors'].str.contains(pattern, regex=True)]
```

```
/tmp/ipykernel_26476/2582873680.py:2: UserWarning: This pattern is interpret
ed as a regular expression, and has match groups. To actually get the group
s, use str.extract.
  df[df['sensors'].str.contains(pattern, regex=True)]
```

Out[13]:

	sensors	values
1	ZB-76-PM	1
3	AB-47-KL	3
4	AB-47-KL	4
7	ZB-76-PM	7

Some explanations about the pattern:

- `.*` means: look for every possible characters
- `(A|Z)` means: look for either an 'A' or a 'Z'
- `7` means: look for a '7'

Note that order matters: the '7' must come after the 'A' or 'Z'