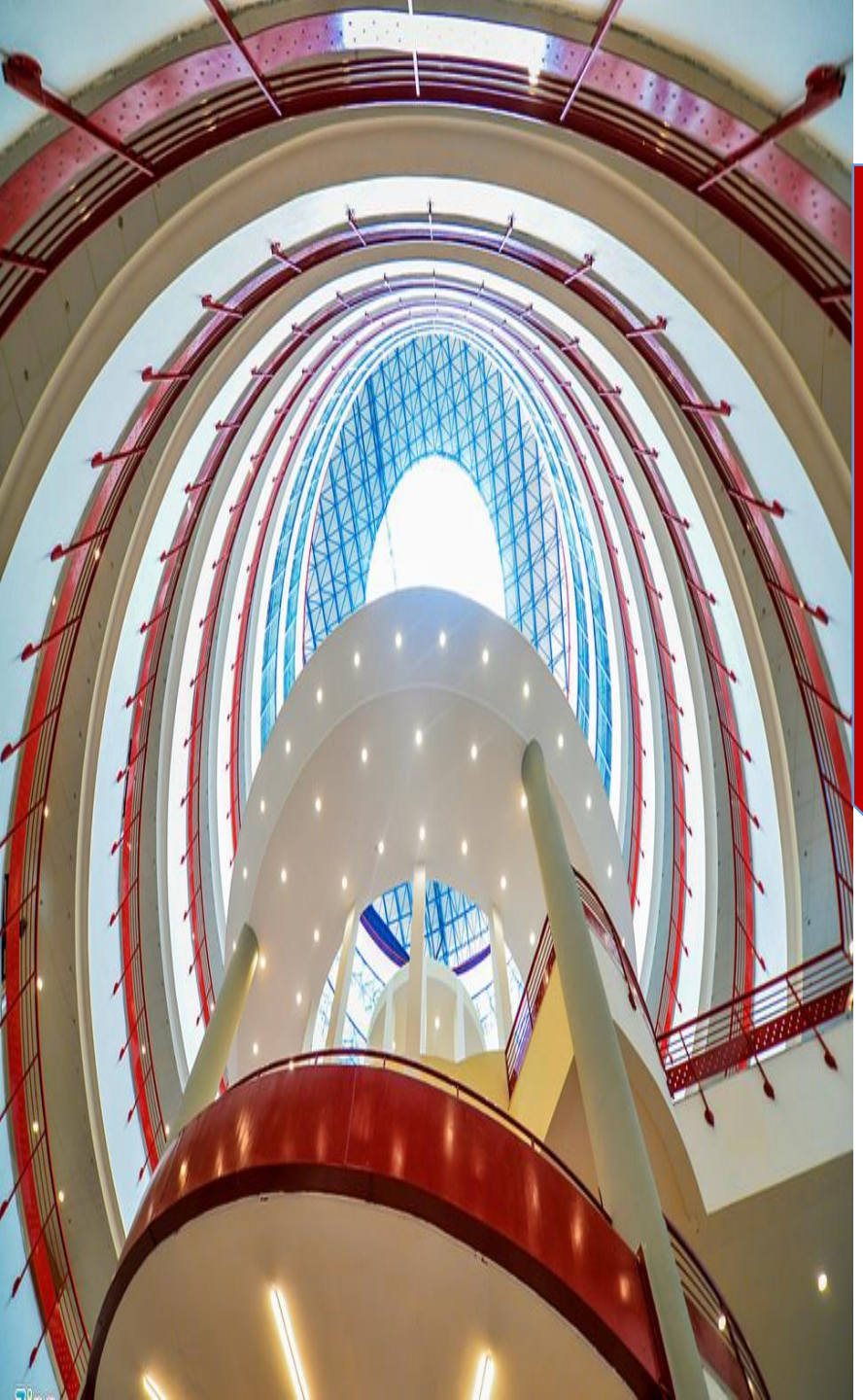
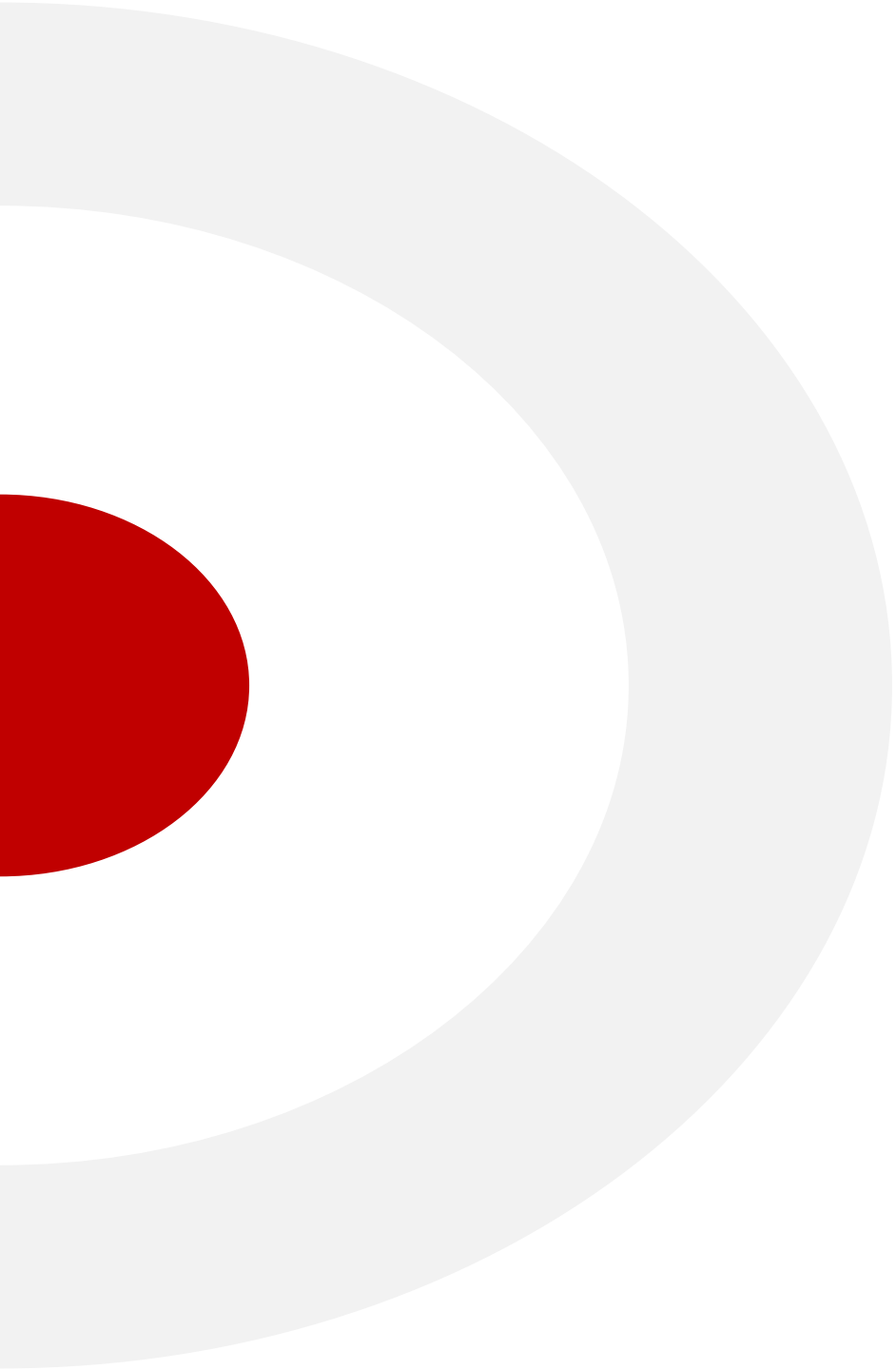


DATA PREPARATION AND VISUALIZATION

Department of Mathematical Economics

National Economics University
https://www.neu.edu.vn/



Chapter 5: Vectorized String Operations

String Manipulation

Introducing Pandas String Operations

- Numpy and Pandas generalize arithmetic operations so that we can easily and quickly perform the same operation on many array elements. Ex

```
In[1]: import numpy as np
       x = np.array([2, 3, 5, 7, 11, 13])
       x * 2
```

- For arrays of strings, NumPy does not provide such simple access, you're stuck using a more verbose loop syntax

```
In[2]: data = ['peter', 'Paul', 'MARY', 'gUIDO']
       [s.capitalize() for s in data]

Out[2]: ['Peter', 'Paul', 'Mary', 'Guido']
```

- It will break if there are any missing values

String Manipulation

Introducing Pandas String Operations

- Pandas includes features to address both this need for vectorized string operations and for correctly handling missing data via the ***str attribute*** of Pandas Series Index objects containing strings. Ex

```
In[4]: import pandas as pd
       names = pd.Series(data)
       names

Out[4]: 0    peter
        1     Paul
        2     None
        3     MARY
        4    gUIDO
        dtype: object
```

- We can now call a single method that will capitalize all the entries, while skipping over any missing values:

```
In[5]: names.str.capitalize()

Out[5]: 0    Peter
        1     Paul
        2     None
        3     Mary
        4    Guido
        dtype: object
```

String Manipulation

Common Vectorized string methods

Method	Description
Series.str.split()	Split strings on delimiter or regular expression
Series.str.strip()	Trim whitespace from both sides, including newlines
Series.str.lower()	
Series.str.upper()	
Series.str.get()	Index into each element(retrieve i-th element)
Series.str.replace()	Replace occurences of pattern/regex with some other string Luu y: Replacement string or a callable (TH replace boi nan thi str.replace() ko hoat dong)

Series + Str + Method_Name()

String Manipulation

series.str.split()

	[CurrencyUnit]
0	Swiss franc
1	Iceland krona
2	Danish krone
3	Norwegian krone
4	Canadian dollar

HOW?

	[CurrencyUnit2]
0	franc
1	krona
2	krone
3	krone
4	dollar

String Manipulation

series.str.split()

- First we use the ‘split’ method to break string into list of substrings

‘Swiss franc’ —split→ [‘Swiss’, ‘franc’]

- Use the ‘map’ method to apply the ‘split’ method to all element of series
- But we should use build-in vectorized methods (if exist) instead of the ‘Series.apply’ method for performance reasons

String Manipulation

series.str.split()

	[CurrencyUnit]
0	Swiss franc
1	Iceland krona
2	Danish krone
3	Norwegian krone
4	Canadian dollar

Str.split

	[Currency Split]
0	[Swiss, franc]
1	[Iceland, krona]
2	[Danish, krone]
3	[Norwegian, krone]
4	[Canadian, dollar]

- We can use the ‘Series.str.split()’ method to perform this task
- This kind of method called the vectorized string methods. This perform the same operation for string in series as Python string method

String Manipulation

series.str.split()

- By default, the series.str.split() method splits the string on whitespace. You can use the pat parameter to split the string by other characters or regular expression
  - Ex:

```
In [23]: s = pd.Series(
             ["this is a regular sentence",\
              "https://docs.python.org/3/tutorial/index.html",\
              np.nan])
         s.str.split('/')

Out[23]: 0                          [this is a regular sentence]
         1    [https:, , docs.python.org, 3, tutorial, index...
         2                                                   NaN
         dtype: object
```

```
In [28]: s.str.split(pat='\d+')

Out[28]: 0                          [this is a regular sentence]
         1    [https://docs.python.org/, /tutorial/index.html]
         2                                                   NaN
         dtype: object
```

String Manipulation

series.str.split()

- The n parameter can be used to limit the number of splits on the delimiter

```
In [9]: s.str.split(n=2)
```

```
Out[9]: 0                    [this, is, a regular sentence]
        1    [https://docs.python.org/3/tutorial/index.html]
        2                                                NaN
        dtype: object
```

- When using expand = True, the split elements will expand out into separate columns. If NaN is present, it is propagated throughout the columns during the split

```
In [16]: s.str.split(expand=True)
```

Out[16]:

	0	1	2	3	4
0	this	is	a	regular	sentence
1	https://docs.python.org/3/tutorial/index.html	None	None	None	None
2	NaN	NaN	NaN	NaN	NaN

String Manipulation

series.str.replace()

- The pandas series.str.replace() is used to replace text in a series
  - The *pat* parameter: str or compiled regex
  - The *repl* parameter: str or callable

```
In [59]: s2 = pd.Series(['foo','fuz',np.nan])
         s2.str.replace('f.','ba',regex=False)

Out[59]: 0    foo
         1    fuz
         2    NaN
         dtype: object
```

Pat repl

String Manipulation

series.str.replace()

- The *regex* parameter determines if the pass-in pattern is a regular expression

```
In [60]: s2.str.replace('f.','ba',regex=True)

Out[60]: 0    bao
         1    baz
         2    NaN
         dtype: object
```

- The *flags* parameter determines if replace is case sensitive

```
In [13]: s2 = pd.Series(['Foo','fuz',np.nan])
         s2.str.replace('f','ba',regex=True,flags = re.IGNORECASE)

Out[13]: 0    baoo
         1    bauz
         2     NaN
         dtype: object
```

String Manipulation

series.str.replace()

- The *flags* parameter determines if replace is case sensitive

```
In [13]: s2 = pd.Series(['Foo','fuz',np.nan])
         s2.str.replace('f','ba',regex=True,flags = re.IGNORECASE)

Out[13]: 0    baoo
         1    bauz
         2     NaN
         dtype: object
```

- Using a compiled regex
  - Regex_pat = re.compile('f.',flags = re.IGNORECASE)

    s2.str.replace(regex_pat,'ba',regex=True)

NOTE: When pat is a compiled regex, all flags should be included in the compiled regex. Use of case, flags, or regex = False with a compiled regex will raise an error

String Manipulation

Methods using regular expressions

- Regular expressions provide a flexible way to search or match(often more complex) string patterns in text
- A single expression, commonly called a regex, is a string formed according to the regular expression language
  - Ex: if we want to find the string “and” within another string, the regex pattern is simply and
- Python’s build-in ***re module*** is responsible for applying regular expressions to string

String Manipulation

Regular Expressions

Character	Pattern	Explanation
Set	[fud]	Either f, u, or d
Range	[a-e]	Any of the characters a, b, c ,d or e
Range	[0-3]	Any of the characters 0, 1, 2, or 3
Range	[A-Z]	Any uppercase letter
Set + Range	[A-Za-z]	Any uppercase or lowercase character
Digit	\d	Any digit character (equivalent to [0-9])
Word	\w	Any digit, uppercase, or lowercase character (equivalent to [A-Za-z0-9]
Whitespace	\s	Any space, tab or linebreak character
Dot	.	Any character except newline

String Manipulation

Regular Expressions

Character Class	Pattern	Explanation
Negative set	[^fud]	Any character except f, u or d
Negative set	[^1-3Z\s]	Any character except 1, 2, 3, Z or whitespace
Negative Digit	\D	Any character except digit characters
Negative Word	\W	Any character except word characters
Negative Whitespace	\s	Any character except whitespace characters

String Manipulation

Regular Expressions

Quantifier	Pattern	Explanation
Zero or more	a*	The character a zero or more times
One or more	a+	The character a one or more times
Optional	a?	The character a zero or one times
Numeric	a{3}	The character a three times
Numeric	a{3, 5}	The character a three, four, or five times
Numeric	a{,3}	The character a one, two or three times
Numeric	a{8,}	The character a eight or more times

String Manipulation

Regular Expressions

Anchor	Pattern	Explanation
Beginning	^abc	Matches abc only at the start of a string
End	abc$	Matches abc only at the end of a string
Word boundary	s\b	Matches s only when it’s followed by a word boundary
Word boundary	s\B	Matches s only when it’s not followed by a word boundary

Resources:
re — Regular expression operations — Python 3.10.5 documentation
https://regexr.com

String Manipulation

Vectorized string methods using regular expressions

The pandas regular expression methods

Method	Description
Series.str.match()	Use re.match with the passed regular expression on each element, returning matched groups as list
Series.str.contains()	Return Boolean array if each string contains pattern/regex
Series.str.extract()	Use a regular expression with groups to extract one or more strings from a Series of strings; the result will be a DataFrame with one column per group
Series.str.findall()	Compute list of all occurrences of pattern/regex for each string

Note:

- The pandas regular expression methods accept an optional flags argument. The most common and the most useful is the re.IGNORECASE(re.I) flag.
- All available flags are here
  re — Regular expression operations — Python 3.10.5 documentation

String Manipulation

series.str.match()

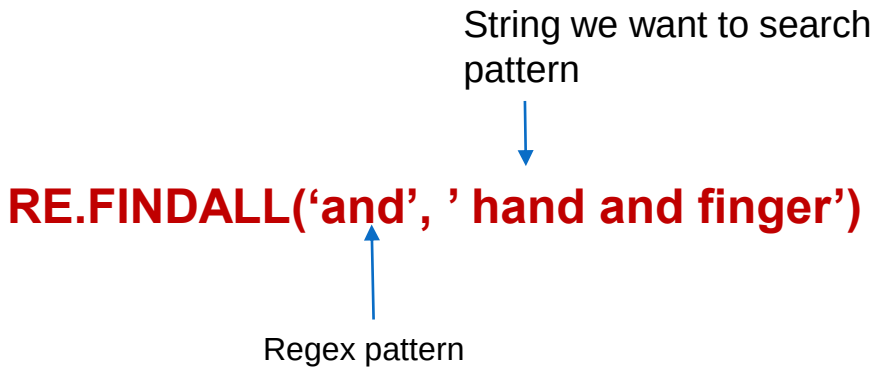
- pd.str.match(pat, flags=0)
  - Parameters:
    - Pat: character sequence or regular expression
    - Flags: int, default 0 (no flags)
  - Returns: Series/Index/array of Boolean values

```
In [14]: s = pd.Series(['Lion','Monkey','Rabbit'])
         s.str.match('\w+on')
```

```
Out[14]: 0     True
         1     True
         2    False
         dtype: bool
```

String Manipulation

series.str.findall()



- The re.findall() function finds all the matches of the pattern in the string
- Findall() matches all occurrences of a pattern, not just the first one as search() does
- Ex: Find all of adverbs in some text
  - Text = “He was carefully disguised but captured quickly by police”
  - Re.findall(r”\w+ly\b”,text)
  - Result: [‘carefully’, ‘quickly”]

String Manipulation

series.str.findall()

SERIES.STR.FINDALL(pattern, flags = 0)

Return all non-overlapping matches of pattern or regular expression in each string of this Series/Index

```
In [5]: s = pd.Series(['Lion','Monkey','Rabbit'])
        s.str.findall('Monkey')

Out[5]: 0         []
        1    [Monkey]
        2         []
        dtype: object
```

```
In [8]: s.str.findall('on$')

Out[8]: 0    [on]
        1      []
        2      []
        dtype: object
```

```
In [9]: s.str.findall('on')

Out[9]: 0    [on]
        1    [on]
        2      []
        dtype: object
```

String Manipulation

series.str.extractall

Syntax

Pd.series.str.extractall(pattern, flags=0)

Return a dataframe with one row for each match, and one column for each group

In [2]:
```
Series = pd.Series(['a1a2', 'b1', 'c1'], index = ['A', 'B', 'C'])
Series.str.extractall(r'[ab](\d)')
```

Out[2]:

		0
	match	
A	0	1
	1	2
B	0	1

String Manipulation

series.str.extractall

Syntax

Pd.series.str.extractall(pattern, flags=0)

Return a dataframe with one row for each match, and one column for each group

```
In [3]: Series = pd.Series(['a1a2', 'b1', 'c1'], index = ['A', 'B', 'C'])
        Series.str.extractall(r'[ab](?P<digit>\d)')
```

Out[3]:

		digit
	match	
A	0	1
	1	2
B	0	1

String Manipulation

series.str.extractall()

Syntax

Pd.series.str.extractall(pattern, flags=0)

Return a dataframe with one row for each match, and one column for each group

```
In [4]: Series.str.extractall(r'(?P<letter>[ab])(?P<digit>\d)')
```

Out[4]:

		letter	digit
match			
A	0	a	1
	1	a	2
B	0	b	1

String Manipulation

series.str.contains()

Syntax

Pd.series.str.contains(pattern, flags=0)

- Test if pattern or regex is contained within a string of a Series or Index
- Return Boolean Series or Index based on whether a given pattern or regex is contained within a string of Series or Index
- ex

```
In [4]: s1 = pd.Series(['Mouse', 'dog', 'house and parrot', '23', np.NaN])
        s1.str.contains('og', regex=False)
```

```
Out[4]: 0    False
        1     True
        2    False
        3    False
        4      NaN
        dtype: object
```