

Engineering programming

Pandas

Thibault Thétier - thibault.thetier@vub.be

Tom Godden - tgodden@vub.be

1ste semester 2021

A major tool of interest throughout the rest of the course

- ▶ Data structures and data manipulation tools for fast and easy data cleaning and analysis
- ▶ Often used in tandem with numerical computing tools like NumPy and data visualization libraries like matplotlib
- ▶ Designed for working with tabular or heterogeneous data

Import convention :

- ▶ `import pandas as pd`

SERIES - DEFINITION

A Series is a one-dimensional array-like object containing:

- ▶ a sequence of values
- ▶ an associated array of data labels, called its index

```
1 obj = pd.Series([4, 7, -5, 3])
```

```
2 Out[12]:
```

```
3 0      4
```

```
4 1      7
```

```
5 2     -5
```

```
6 3      3
```

```
7
```

```
8 obj.values
```

```
9 array([ 4,  7, -5,  3])
```

```
1 obj2 = pd.Series([4, 7, -5, 3],
```

```
    ↪ index=['d', 'b', 'a', 'c'])
```

```
2 Out[16]:
```

```
3 d      4
```

```
4 b      7
```

```
5 a     -5
```

```
6 c      3
```

```
7
```

```
8 obj2.index
```

```
9 Index(['d', 'b', 'a', 'c'],
```

```
    ↪ dtype='object')
```

SERIES - INDEXING

You can use labels in the index when selecting single values or a set of values:

```
1 obj2['d'] = 6
```

```
2 obj2[['c', 'a', 'd']]
```

```
3 Out[20]:
```

```
4 c      3
```

```
5 a     -5
```

```
6 d      6
```

```
1 obj2[obj2 > 0]
```

```
2 Out[21]:
```

```
3 d      6
```

```
4 b      7
```

```
5 c      3
```

```
1 obj2 * 2
```

```
2 Out[22]:
```

```
3 d     12
```

```
4 b     14
```

```
5 a    -10
```

```
6 c      6
```

SERIES - DICT

You can create a Series from a dict:

```
1 sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah':  
    ↪ 5000}  
2 states = ['California', 'Ohio', 'Oregon', 'Texas']
```

```
1 obj3 = pd.Series(sdata)  
2 Out[28]:  
3 Ohio      35000  
4 Oregon    16000  
5 Texas     71000  
6 Utah      5000
```

```
1 obj4 = pd.Series(sdata,  
    ↪ index=states)  
2 Out[31]:  
3 California  NaN  
4 Ohio        35000.0  
5 Oregon      16000.0  
6 Texas       71000.0
```

SERIES - MISSING VALUES

You can easily detect the missing data:

```
1 pd.isnull(obj4)
2 #also obj4.isnull()
3 Out[32]:
4 California    True
5 Ohio          False
6 Oregon        False
7 Texas         False
8 dtype: bool
```

```
1 pd.notnull(obj4)
2 Out[33]:
3 California    False
4 Ohio          True
5 Oregon        True
6 Texas         True
7 dtype: bool
```

DATAFRAME

- ▶ Represents a rectangular table of data
- ▶ Contains an ordered collection of columns
- ▶ Each column can be of a different value type
- ▶ Has both a row and column index
 - ▶ Sort of a "dict of Series"

```
1 data =  
2 {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],  
3  'year': [2000, 2001, 2002, 2001, 2002, 2003],  
4  'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}  
5 frame = pd.DataFrame(data)
```

DATAFRAME

```
1 frame
2 Out[45]:
```

	pop	state	year
0	1.5	Ohio	2000
1	1.7	Ohio	2001
2	3.6	Ohio	2002
3	2.4	Nevada	2001
4	2.9	Nevada	2002
5	3.2	Nevada	2003

```
1 pd.DataFrame(data,
  ↪ columns=['year', 'state',
  ↪ 'pop'])
2 Out[47]:
```

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

DATAFRAME - MISSING VALUES

```
1 frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],  
2   index=['one', 'two', 'three', 'four', 'five', 'six'])  
3
```

```
4 Out[49]:
```

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN
six	2003	Nevada	3.2	NaN

DATAFRAME - ROWS AND COLUMNS

```
1 frame2['state']
2 frame2.state
3
4 one      Ohio
5 two      Ohio
6 three    Ohio
7 four     Nevada
8 five     Nevada
9 six      Nevada
```

```
1 frame2.loc['three']
2
3 year      2002
4
5 state     Ohio
6
7 pop       3.6
8
9 debt      NaN
```

DATAFRAME - MODIFICATIONS

Columns can be modified by assignment

```
1 frame2['debt'] = 16.5
```

```
2
```

	year	state	pop	debt
one	2000	Ohio	1.5	16.5
two	2001	Ohio	1.7	16.5
three	2002	Ohio	3.6	16.5
four	2001	Nevada	2.4	16.5
five	2002	Nevada	2.9	16.5
six	2003	Nevada	3.2	16.5

```
1 frame2['debt'] = np.arange(6.)
```

```
2
```

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0
six	2003	Nevada	3.2	5.0

DATAFRAME - MODIFICATIONS

If you assign a Series, its labels will be realigned exactly to the DataFrame's index

```
val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])  
frame2['debt'] = val
```

1		year	state	pop	debt
2	one	2000	Ohio	1.5	NaN
3	two	2001	Ohio	1.7	-1.2
4	three	2002	Ohio	3.6	NaN
5	four	2001	Nevada	2.4	-1.5
6	five	2002	Nevada	2.9	-1.7
7	six	2003	Nevada	3.2	NaN

DATAFRAME - MODIFICATIONS

Conditions can be used to create new columns

```
1 frame2['eastern'] = frame2.state == 'Ohio'
2
3   year  state  pop  debt  eastern
4 one   2000   Ohio   1.5   NaN   True
5 two   2001   Ohio   1.7  -1.2   True
6 three 2002   Ohio   3.6   NaN   True
7 four  2001  Nevada  2.4  -1.5  False
8 five  2002  Nevada  2.9  -1.7  False
9 six   2003  Nevada  3.2   NaN  False
```

New columns cannot be created with the `frame2.eastern` syntax

REINDEXING

An important method on pandas objects is `reindex`

```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])  
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
```

```
1  obj  
2  Out[92]:  
3  d      4.5  
4  b      7.2  
5  a     -5.3  
6  c      3.6
```

```
1  obj2  
2  Out[94]:  
3  a     -5.3  
4  b      7.2  
5  c      3.6  
6  d      4.5  
7  e      NaN
```

DROP

`drop` method will return a new object with the indicated value(s) deleted from an axis

```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
```

```
1 obj
2 Out[106]:
3 a      0.0
4 b      1.0
5 c      2.0
6 d      3.0
7 e      4.0
```

```
1 new_obj = obj.drop('c')
2 new_obj
3 Out[108]:
4 a      0.0
5 b      1.0
6 d      3.0
7 e      4.0
```

You can drop values from the columns by passing `axis=1` or `axis='columns'`

FUNCTIONS

NumPy ufuncs (element-wise array methods) also work with pandas objects

```
► frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),  
                        index=['Utah', 'Ohio', 'Texas', 'Oregon'])
```

```
1 frame
2      b      d      e
3 Utah -0.20  0.47 -0.51
4 Ohio -0.55  1.96  1.39
5 Texas  0.09  0.28  0.76
6 Oregon 1.24  1.00 -1.29
```

```
1 np.abs(frame)
2      b      d      e
3 Utah  0.20  0.47  0.51
4 Ohio  0.55  1.96  1.39
5 Texas  0.09  0.28  0.76
6 Oregon 1.24  1.00  1.29
```


SORTING

Sorting a dataset by some criterion is another important built-in operation

```
1 obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
2 frame = pd.DataFrame(np.arange(8).reshape((2, 4)), index=['three',
↪ 'one'], columns=['d', 'a', 'b', 'c'])
```


1	obj.sort_index()	1	frame.sort_index()	1	frame.sort_index(axis=1)
2	a 1	2	d a b c	2	a b c d
3	b 2	3	one 4 5 6 7	3	three 1 2 3 0
4	c 3	4	three 0 1 2 3	4	one 5 6 7 4
5	d 0				

To sort a Series by its values, use its `sort_values` method

SORTING

The data is sorted in ascending order by default, but can be sorted in descending order.

```
1 frame.sort_index(axis=1,  
  ↪ ascending=False)
```

	d	c	b	a
three	0	3	2	1
one	4	7	6	5

```
1 obj = pd.Series([4, 7, -3, 2])
```

```
2 obj.sort_values()
```

3	2	-3
4	3	2
5	0	4
6	1	7

SORTING

You can use the data in one or more columns as sort keys

```
1 frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
```

```
1 frame
2   a  b
3  0  4
4  1  7
5  2  0
6  3  1
```

```
.
1 frame.sort_values(by='b')
2   a  b
3  2  0
4  3  1
5  0  4
6  1  1
```

```
1 frame.sort_values(by=['a',
2   a  b
3  2  0
4  0  4
5  3  1
6  1  1
```