Introduction to the pandas library

Janusz Jabłonowski, MIMUW

Purpose

- Designed for
  - cleaning data
  - analyzing data
  - work with tabular data
  - work with heterogeneous data
- Provides
  - data structures
  - tools for data manipulation
- Uses NumPy

History

- Started in 2008
- Open source project
- Over 800 contributors

Name

- from the term "**pan**el **da**ta" (term used in econometrics)
- also a play on the phrase "**P**ython **d**ata **a**nalysi**s**"

But in a library created for Python there must have been an influence of them:

(picture taken from www.onthegotours.com)

Conventions

```python
import pandas as pd

from pandas import Series, DataFrame
```

Series

- One dimensional sequence of data with labels
- Contains
    - series of values
    - index - an array of data labels
- The default index is a sequence of integer numbers starting from 0
- Attributes
    - values
    - index
- A Series object can be created from a sequence (like a list)
- The index might be provided, it is not obligatory to do so, by default these are consecutive integers starting from 0
- The values in index do not [sic] have to be different

Series

```
pd.Series(range(-5, 6, 2))
```

```
0   -5
1   -3
2   -1
3    1
4    3
5    5
dtype: int64
```

- Left column is the index, the right - values

Series index

- The index can be specified explicitly

```
pd.Series([3,1,-2], ['a', 'b', 'c'])
a     3
b     1
c    -2
dtype: int64
```

Series index

- The index can be also retrieved

```
p1 = pd.Series([3,1,-2], ['a','b','c'])
p1.index
Index(['a', 'b', 'c'], dtype='object')
```

Series index

- Compare the following

```
p1, p2, p3, p4 = pd.Series([3,-7]), pd.Series([3,-7], ['a','b']),
             pd.Series([3,-7], [0,1]), pd.Series([3,-7], range(2))

p1.index
RangeIndex(start=0, stop=2, step=1)
p2.index
Index(['a', 'b'], dtype='object')
p3.index
Int64Index([0, 1], dtype='int64')
p4.index
RangeIndex(start=0, stop=2, step=1)
```

Series - indexing

- Elements of a series can be retrieved by indices (as lists or NumPy's arrays) or by the index
- Accessing by index is fast but (sometimes) slower than by indices
    - it seems that specifying non-integer index and then not using it in favor of intg indices works fast
    - it requires further investigation (see also slide about index/indices)

Series - indexing

```python
import timeit
n = 1000000
ser = pd.Series([i for i in range(n)], [str(i) for i in range(n)])

for ind in [0, n//2, n-1, 0]:
    print(f"{ind=}")
    print(f'{timeit.timeit("ser[ind]",globals={"ser": ser, "ind": ind})=}')
    ind = str(ind)
    print(f"{ind=}")
    print(f'{timeit.timeit("ser[ind]",globals={"ser": ser, "ind": ind})=}')
```

```
ind=0
timeit.timeit("ser[ind]",globals={"ser": ser, "ind": ind})=1.5375394
ind='0'
timeit.timeit("ser[ind]",globals={"ser": ser, "ind": ind})=2.654256799999999
```

Series - indexing

- Be careful with integer index!
- Which indexing works then (general or from index)?
- pandas decides based on type of index expression used, in case the index is made of integers index takes precedence

```
ser = pd.Series(['a', 'b', 'c'], [1, 3, -2])
for i in [1, 3, -2]:
    print(f"{i=} {ser[i]=}")
# 0 and 2 indices would cause an KeyError
```

```
i=1 ser[i]='a'
i=3 ser[i]='b'
i=-2 ser[i]='c
```

Series - indexing by lists

```
ser = pd.Series(range(5), ['a', 'b', 'c', 'd', 'e'])

ser['b']
ser['b']=1

ser[['b','e','b','a']]
b     1
e     4
b     1
a     0
dtype: int64
```

Series - indexing by lists

- Both index and indices can be used within a list
- But cannot be mixed

```
ser = pd.Series([i*10 for i in range(5)], ['a', 'b', 'c', 'd', 'e'])
ser[['b','e','b','a']]
    # using as index a list of series index elements
ser[[1,4,1,0]]
    # using as index a list of indices
b    10
e    40
b    10
a     0
dtype: int64
```

Series - indexing by ranges

- Ranges work for series in indexing
- Both integer indices and labels can be used BUT
- They have different meaning (label ranges are right-inclusive)

```
ser = pd.Series([i * 10 for i in range(5)], ['a', 'b', 'c', 'd', 'e'])

ser[:2]
a     0
b    10
dtype: int64

ser[1:3]
b    10
c    20
dtype: int64
```

Series - indexing by ranges

```
ser = pd.Series([i * 10 for i in range(5)], ['a', 'b', 'c', 'd', 'e'])

ser[:2]
a     0
b    10
dtype: int64

ser[1:3]
b    10
c    20
dtype: int64

ser['b':'d']  # different (other users, there is no successor for a string)
b    10
c    20
d    30
dtype: int64
```

Series - indexing by ranges

```
ser.index = ['b', 'b', 'b', 'd', 'd']
ser['b':'d']
b     0
b    10
b    20
d    30
d    40
dtype: int6464
```

- Same for `ser.index = ['b', 'b', 'c', 'd', 'd']` (only c 20)
- But with this index `['b', 'c', 'b', 'd', 'd']` it wouldn’t work

Series and scalar operations

- Similarly like with NumPy

```
import numpy as np
ser = pd.Series([i*10 for i in range(5)], ['a', 'b', 'c', 'd', 'e'])
ser*2+7
a     7
b    27
c    47
d    67
e    87
# math.sqrt(ser)  # TypeError: cannot convert the series to <class 'float'>
np.sqrt(ser)

a    0.000000
b    3.162278
c    4.472136
d    5.477226
```

(slightly formatted results)

Series - selecting with boolean vectors

- Similarly like with NumPy (again)

```
ser = pd.Series([i*10 for i in range(5)], ['a', 'b', 'c', 'd', 'e'])

ser<30
a      True
b      True
c      True
d     False
e     False
dtype: bool

ser[ser<30]
a      0
b     10
c     20
dtype: int64
```

(slightly formatted results)

Series as dictionaries

- Labels may be searched for with the in operator
- Series may be initialized from a dictionary (labels are sorted keys then)

```
ser = pd.Series([i*10 for i in range(5)], ['a', 'b', 'c', 'd', 'e'])

'a' in ser  # verifying if the key is present
'a' in ser=True

ser = pd.Series({'f': 60, 'g': 70, 'h': 80})
ser
f     60
g     70
h     80
dtype: int64
```

(slightly formatted results)

Series as dictionaries

- Index may be specified when creating a series from dictionary
- Then labels (and their order) are taken from given index, but values are searched for in the dictionary
- Those not found are replaced with the NaN (Not a Number) value

```
ser = pd.Series({'f': 60, 'g': 70, 'h': 80}, ['g', 'h', 'i'])
ser
```

```
g    70.0
h    80.0
i     NaN
```

(slightly formatted results)

Series and missing data

- NaN represents missing data
- pandas provides tools for distinguishing missing data

```
pd.notnull(ser)     # or print(f"{ser.notnull()=}")
g       True
h       True
i      False

pd.isnull(ser)      # or print(f"{ser.isnull()=}")
g      False
h      False
i       True
```

(slightly formatted results)

Series and missing data

- Notion of missing data, indexing by lists/arrays and scalar operations enable for quite sophisticated expressions

```
ser = pd.Series({'a': 1, 'd':2, 'e':3, 'z':4},
                [chr(i) for i in range(ord('a'),ord('z')+1)])

ser[ser.notnull() & (ser>2)]
e    3.0
z    4.0
dtype: float64
```

(slightly formatted results)

Series - arithmetical operations

- Arithmetical operations on series are done scalarly
- Elements are selected based on their labels
- If label is missing in one of the series the result is NaN

```
ser1 = pd.Series({'a':1, 'b':2, 'c':3})
ser2 = pd.Series({'a':4, 'c':5, 'e':6})
ser1+ser2
a    5.0
b    NaN
c    8.0
e    NaN
dtype: float64

(slightly formatted results)
```

- Converted to float64 since integer type does not have the value NaN, without NaNs the result type is of course int64

Series - assigning

- Series elements can be accessed and assigned
  - by index (dictionary style)
  - by name (!) if it is a valid Python name
  - by (integer) indices

```
ser = pd.Series({'a': 1, 'b': 2, 'c': 3})
ser['a'] = -1
ser.b = -2
ser[2] = -3
ser
a    -1
b    -2
c    -3
```

(slightly formatted results)

Series - (some) attributes

- The series and the index can be assigned a name

```
ser = pd.Series({'a': 1, 'b': 2, 'c': 3})
ser.name = "My data"
ser.index.name = "Labels"  # no name attribute for values
ser
Labels
a    1
b    2
c    3
Name: My data, dtype: int64
```

(slightly formatted results)

Series - replacing index

- The index may be replaced (the number of labels must be the same, else ValueError)

```
ser = pd.Series({'a': 1, 'b': 2, 'c': 3})
a    1
b    2
c    3

ser.index = ['d', 'e', 'f']
ser
d    1
e    2
f    3
```

(slightly formatted results)

DataFrame

- Rectangular collection of data
- Ordered collection of columns
- Each column can contain different type of data
- Can be indexed on rows and columns
- Idea: a dictionary of series with the same index
- Internal implementation is different

Creating

- Many possibilities
- Most common - from a dictionary of lists or arrays
- Those arrays have to have the same length (else ValueError)

DataFrame example

```
df = pd.DataFrame(
        {'A': ['a', 'b', 'c', 'd'],
         'B': range(4),
         'C': (1.5, 3.14, -1.0, 4)
        })
```

```
   A  B     C
0  a  0  1.50
1  b  1  3.14
2  c  2 -1.00
3  d  3  4.00
```

DataFrame example

- The order of columns from the data source can be changed

```
df = pd.DataFrame(
        {'A': ['a', 'b', 'c', 'd'],
         'B': range(4),
         'C': (1.5, 3.14, -1.0, 4)
        }, columns=["C", "A", "B"])
```

```
      C  A  B
0  1.50  a  0
1  3.14  b  1
2 -1.00  c  2
3  4.00  d  3
```

DataFrame example

- Columns can be skipped

```
df = pd.DataFrame(
        {'A': ['a', 'b', 'c', 'd'],
         'B': range(4),
         'C': (1.5, 3.14, -1.0, 4)
        }, columns=["C", "A"])
```

```
      C  A
0  1.50  a
1  3.14  b
2 -1.00  c
3  4.00  d
```

DataFrame example

- Columns can be duplicated

```
df = pd.DataFrame(
        {'A': ['a', 'b', 'c', 'd'],
         'B': range(4),
         'C': (1.5, 3.14, -1.0, 4)
        }, columns=["C", "A", "C"])

      C  A     C
0  1.50  a  1.50
1  3.14  b  3.14
2 -1.00  c -1.00
3  4.00  d  4.00
```

DataFrame example

- And even: new columns can be added

```
df = pd.DataFrame(
        {'A': ['a', 'b', 'c', 'd'],
         'B': range(4),
         'C': (1.5, 3.14, -1.0, 4)
        }, columns=["C", "A", "D"])
```

```
      C  A    D
0  1.50  a  NaN
1  3.14  b  NaN
2 -1.00  c  NaN
3  4.00  d  NaN
```

DataFrame index

- Index can be added when creating a dataframe or later

```
df = pd.DataFrame({'A': ['a', 'b', 'c', 'd'], 'B': range(4), 'C': (1.5, 3.14, -1.0, 4)},
                  index = ["r_a","r_b","r_c","r_d"])
     A  B     C
r_a  a  0  1.50
r_b  b  1  3.14
r_c  c  2 -1.00
r_d  d  3  4.00
df.index = ["r"+str(i) for i  in range(len(df.A))]
df
    A  B     C
r0  a  0  1.50
r1  b  1  3.14
r2  c  2 -1.00
r3  d  3  4.00
```

(the results have been slightly edited)

DataFrame - accessing columns

- Columns of a DataFrame may be accessed
    - in dictionary style
    - by name (!) if it is a valid Python name
    - accessing columns by (integer) indices does not work
- The result is a series (object)
- With the same index as in the DataFrame
- And a name (column name from the DataFrame)
- (see the example on the next slide)

DataFrame - accessing columns

```
df = pd.DataFrame({'A': ['a', 'b', 'c', 'd'], 'B': range(4), 'C': (1.5, 3.14, -1.0, 4)})
df['B']=0    0
1    1
2    2
3    3
Name: B, dtype: int64
```

df.B

```
df.B=0    0
1    1
2    2
3    3
Name: B, dtype: int64

# print(f"{df[1]=}")  # KeyError: 1
```

DataFrame - accessing rows

- Rows of a DataFrame may be accessed using the special *loc* attribute
- This is the label based method of accessing rows
- Syntax: `<df>.loc[<label>]`
    - integer indices (e.g. loc[7]) cannot be used instead of labels
    - attribute syntax (e.g. loc.b) cannot be used
- The result is a series (object)
- With index being columns names (from the dataframe)
- And a name (row name from the DataFrame)
- (see the example on the next slide)

DataFrame - accessing rows

```
df = pd.DataFrame({'A': ['one', 'two', 'three', 'four'], 'B': range(4), 'C': (1.5, 3.14, -1.0,
4)},
                                 index=['a','b','c', 'd'])
       A  B     C
a    one  0  1.50
b    two  1  3.14
c  three  2 -1.00
d   four  3  4.00

r = df.loc['b']
A     two
B       1
C    3.14

r['B'], r.B, r[1]
1, 1, 1
```

(the results have been slightly edited)

DataFrame - accessing rows

- There is another way of accessing rows - indices based
- Using the special *iloc* attribute
- Syntax: `<df>.iloc[<index>]`
    - Neither labels (e.g. iloc['a']) nor attribute syntax (e.g. iloc.b) can be used
- The result is a series (object)
- With index being columns names (from the dataframe)
- And a name (row name from the DataFrame, and not the index!)
- (see the example on the next slide)

DataFrame - accessing rows

```
df = pd.DataFrame({'A': ['one', 'two', 'three', 'four'],
                   'B': range(4),
                   'C': (1.5, 3.14, -1.0, 4)},
                  index = ['a', 'b', 'c', 'd'])
       A  B     C
a    one  0  1.50
b    two  1  3.14
c  three  2 -1.00
d   four  3  4.00

r = df.iloc[1]
A      two
B        1
C     3.14

r['B'], r.B, r[1]
1, 1, 1
```

(the results have been slightly edited)

DataFrame - accessing single cells

- One can access column (row) and then select elements
- There is a shorter way of writing it
- Syntax1: `<df>.at[<row_label>, <col_label>]`
- Syntax2: `<df>.iat[<row_index>, <col_index>]`
    - One cannot provide label instead of index or vice-versa
- The result is a single value from the dataframe (usually a NumPy number)
- (see the example on the next slide)

DataFrame - accessing rows

```
df = pd.DataFrame({'A': ['one', 'two', 'three', 'four'],
                   'B': range(4),
                   'C': (1.5, 3.14, -1.0, 4)},
                  index=['a', 'b', 'c', 'd'])
       A  B     C
a    one  0  1.50
b    two  1  3.14
c  three  2 -1.00
d   four  3  4.00

df.at['b', 'B'], type(df.at['b', 'B'])
1, <class 'numpy.int64'>

df.iat[1, 1], type(df.iat[1, 1])
1, <class 'numpy.int64'>
```

(the results have been slightly edited)

DataFrame - cell access timing

- Testing time of various methods of access
- Note: timings are heavily implementation/version/hardware type dependent!
- Pandas version used in this tests: 1.2.0
- Below: data for tests (test are on the next slides)

```
n = 1000
data = {}
for i in range(n):
    data[str(i)] = [i for i in range(n)]
df = pd.DataFrame(data, index = [str(i) for i in range(n)])

pd.__version__
np.__version__
df.size, df.shape
```

DataFrame - cell access timing

```
print(f"{timeit.timeit('df[i][j]', globals={'df': df, 'i':'500', 'j':'500'})=}")
print(f"{timeit.timeit('df.loc[i][j]', globals={'df': df, 'i':'500', 'j':'500'})}=")
print(f"{timeit.timeit('df[i][j]', globals={'df': df, 'i':'500', 'j':500})=}")
print(f"{timeit.timeit('df[i].values[j]', globals={'df': df, 'i':'500', 'j':500})=}")
print(f"{timeit.timeit('df.loc[i][j]', globals={'df': df, 'i':'500', 'j':500})=}")
print(f"{timeit.timeit('df.iloc[i][j]', globals={'df': df, 'i':500, 'j':500})=}")
print(f"{timeit.timeit('df.at[i,j]', globals={'df': df, 'i':'500', 'j':'500'})=}")
print(f"{timeit.timeit('df.iat[500, 500]', globals={'df': df, 'i':500, 'j':500})=}")

i, j = '500', 500
old = df[i][j]

print(f"{timeit.timeit('df[i][j]=-1', globals={'df': df, 'i':'500', 'j':'500'})=}")
assert df[i][j] == -1
df[i][j] = old

...
```

DataFrame - accessing rows

```
pd.__version__='1.2.0', np.__version__='1.19.3', df.size=1000000, df.shape=(1000, 1000)
timeit.timeit('df[i][j]', globals={'df': df, 'i':'500', 'j':'500'})                 =  5.38...
timeit.timeit('df.loc[i][j]', globals={'df': df, 'i':'500', 'j':'500'})             =110.47...
timeit.timeit('df[i][j]', globals={'df': df, 'i':'500', 'j':500})                   =  3.90...
timeit.timeit('df[i].values[j]', globals={'df': df, 'i':'500', 'j':500})            =  2.45...
timeit.timeit('df.loc[i][j]', globals={'df': df, 'i':'500', 'j':500})               =104.46...
timeit.timeit('df.iloc[i][j]', globals={'df': df, 'i':500, 'j':500})                = 95.57...
timeit.timeit('df.at[i,j]', globals={'df': df, 'i':'500', 'j':'500'})               =  4.15...
timeit.timeit('df.iat[500, 500]', globals={'df': df, 'i':500, 'j':500})             = 20.18...

timeit.timeit('df[i][j]=-1', globals={'df': df, 'i':'500', 'j':'500'})              = 47.57...
timeit.timeit('df.loc[i][j]=-1', globals={'df': df, 'i':'500', 'j':'500'})          =105.56...
timeit.timeit('df[i][j]=-1', globals={'df': df, 'i':'500', 'j':500})                = 47.30...
timeit.timeit('df[i].values[j]=-1', globals={'df': df, 'i':'500', 'j':500})         =  2.22...
timeit.timeit('df.loc[i][j]=-1', globals={'df': df, 'i':'500', 'j':500})            =106.35...
timeit.timeit('df.iloc[i][j]=-1', globals={'df': df, 'i':500, 'j':500})             = 97.32...
timeit.timeit('df.at[i,j]=-1', globals={'df': df, 'i':'500', 'j':'500'})            =  6.91...
timeit.timeit('df.iat[500, 500]=-1', globals={'df': df, 'i':500, 'j':500})          = 21.80...
```

(the results have been slightly edited)

DataFrame - assigning to columns

- One can assign to entire column
- A scalar gets copied over all values within the column
- A list or array has to be of the same size and is assigned element by element
- A series is assigned by indexes (that of the column at the series, elements with matching index values are assigned)

DataFrame - examples of column assignments

```
df = pd.DataFrame({'A': ['one', 'two', 'three', 'four'],
                   'B': range(4),
                   'C': (1.5, 3.14, -1.0, 4)},
                  index=['a', 'b', 'c', 'd'])
df['B'] = 13
```

```
       A   B     C
a    one  13  1.50
b    two  13  3.14
c  three  13 -1.00
d   four  13  4.00
```

DataFrame - examples of column assignments

```
df['B'] = [1, 2, 3, 4]
df['B'] = (1, 2, 3, 4)
df['B'] = {1, 2, 3, 4}
df['B'] = range(1, 4+1)
df['B'] = np.array([1, 2, 3, 4])
```

```
       A  B     C
a    one  1  1.50
b    two  2  3.14
c  three  3 -1.00
d   four  4  4.00
```

DataFrame - examples of column assignments

```
df['B'] = {'a':1, 'b': 2, 'd': 3, 'e':4}  # ignores values, copied keys!!
       A  B      C
a    one  a   1.50
b    two  b   3.14
c  three  d  -1.00
d   four  e   4.00

df['B'] = pd.Series({'a':1, 'b': 2, 'd': 3, 'e':4})
       A    B      C
a    one  1.0   1.50
b    two  2.0   3.14
c  three  NaN  -1.00
d   four  3.0   4.00
```

DataFrame - adding and deleting columns

- Assigning to non existing column creates one
- The *del* instruction removes the named column

DataFrame - adding and deleting columns

```
pd.DataFrame({'A': ['one', 'two', 'three', 'four'],
              'B': range(4),
              'C': (1.5, 3.14, -1.0, 4)},
             index=['a', 'b', 'c', 'd'])
df['D'] = np.arange(4.)
       A  B     C    D
a    one  0  1.50  0.0
b    two  1  3.14  1.0
c  three  2 -1.00  2.0
d   four  3  4.00  3.0
del df['C']
       A  B    D
a    one  0  0.0
b    two  1  1.0
c  three  2  2.0
d   four  3  3.0
```

DataFrames and nested dictionaries

- Nested dictionaries enable creation of dataframes with index
- Keys of nested dictionaries are combined and sorted to create the index

```
d = {'A': {'a':1, 'b':2, 'c': 3},
     'B': {'b':4},
     'C': {'b':1.5, 'd':2.5} }
df = pd.DataFrame(d)
```

```
    A    B    C
a  1.0  NaN  NaN
b  2.0  4.0  1.5
c  3.0  NaN  NaN
d  NaN  NaN  2.5
```

DataFrame - transposing

```
pd.DataFrame({'A': ['one', 'two', 'three', 'four'],
              'B': range(4),
              'C': (1.5, 3.14, -1.0, 4)},
             index=['a', 'b', 'c', 'd'])
       A  B     C
a    one  0  1.50
b    two  1  3.14
c  three  2 -1.00
d   four  3  4.00
df = df.T
     a     b      c     d
A  one   two  three  four
B    0     1      2     3
C  1.5  3.14   -1.0   4.0
```

DataFrame - getting all data

- Entire contents of a dataframe can be accessed as an 2d NumPy array
- It is a copy (assigning to its elements does not change the original dataframe)
- The type of data elements is selected as one matching with all column types (e.g. object)
- (see the next slides)

DataFrame - getting all data

```
df = pd.DataFrame({'A': ['one', 'two', 'three', 'four'],
                   'B': range(4),
                   'C': (1.5, 3.14, -1.0, 4)},
                   index=['a', 'b', 'c', 'd'])
```

```
       A  B     C
a    one  0  1.50
b    two  1  3.14
c  three  2 -1.00
d   four  3  4.00
```

(the results have been slightly edited)

DataFrame - getting all data

```
tab = df.values

array([['one', 0, 1.5],
       ['two', 1, 3.14],
       ['three', 2, -1.0],
       ['four', 3, 4.0]], dtype=object)

tab[1,1] = 13    # it is a copy of the original data

       A  B     C
a    one  0  1.50
b    two  1  3.14
c  three  2 -1.00
d   four  3  4.00
```

(the results have been slightly edited)

DataSets - small utilities

- The head method takes only the specified (default is 5) number of first rows

```
df = pd.DataFrame({'A': ['a', 'b', 'c', 'd'], 'B': range(4), 'C': (1.5, 3.14, -1.0, 4)})
   A  B     C
0  a  0  1.50
1  b  1  3.14
2  c  2 -1.00
3  d  3  4.00


df.head(1)
   A  B    C
0  a  0  1.5
```

(the results have been slightly edited)

DataSets - assigning names

- Both index as well as columns may have names
- These (when set) are displayed with the frame

```
df = pd.DataFrame({'A': ['one', 'two', 'three', 'four'], 'B': range(4), 'C': (1.5, 3.14, -1.0, 4)},
                  index=['a', 'b', 'c', 'd'])
df.index.name, df.columns.name = "index", "columns"
```

```
columns      A  B      C
index
a          one  0   1.50
b          two  1   3.14
c        three  2  -1.00
d         four  3   4.00
```

(the results have been slightly edited)

SettingWithCopyWarning

- Full discussion is outside of the scope of this lecture
- A Warning - not an error
- Often with hierarchical indexing
- On some operations pandas may return a view or a copy of data
- Assigning to result of such operation may or may not update the data frame - this is the message of that warning

There is much more

- It is only an introduction to pandas
- It provides for example many more operations on data
- Reading and writing data will be shown during tutorials

Thank you for your attention!