Chapter 4: The pandas Library - An Introduction

Saul SL

June 2023

1 Introduction

In Pandas two data structures are defined, `Series` and `Data Frame`. The former is a uni-dimensional object similar to an array whereas, the later is designed to hold multi-dimensional data.

```
import pandas as pd
import numpy as np
```

2 Working with Series

- Series consist of two associated arrays, one holding the *values* and another holding labels or *indices*
- Values and indices are accesed as follows.

```
# Define a series
series = pd.Series([12, -4, 7, 9], index=["a", "b", "c", "d"])

# Return the values
series.values

# Return the indices
series.index

# Return the second value
series[1]

# Return the first 2 values
series[0:3]

# Return the value associated with label 'c'
series["c"]

# Return the values associated with labels 'a' and 'b''
series[["a", "b"]]
```

```
>>> series.values
array([12, -4,  7,  9])

>>> series.index
Index(['a', 'b', 'c', 'd'], dtype='object')

>>> series[["a", "b"]]
a    12
b    -4
dtype: int64
```

2.1 Convert from a numpy array.

```
arr = np.array([1, 2, 3, 4])
s3 = pd.Series(arr)
arr[0] = 0
```

```
>>> s3
0    0
1    2
2    3
3    4
dtype: int64
```

Note that the values of `arr` are passed to s3 by reference; changes in `arr` causes changes in `s3`.

2.2 Functions on series

```
s = pd.Series([12, -4, 7, 9], index=["a", "b", "c", "d"])

# Filtering. Returns a series
s[s > 8]

# Arithmetic operations
s / 2
s * 3
s + 3
s - 5

# numpy math function
np.log(s)

# Unique values not indices.
s.unique()

# count number of occurrences
s.value_counts()

# are any of the values of s in [7,9]. Returns bool.
s.isin([7, 9])

# checks for missing values NaN. Returns bool
s.isnull()

# removes NaN
s.notnull()
s[s.notnull()]
```

```
>>> s[s > 8]
a    12
d     9
dtype: int64

>>> s.isin([7, 9])
a    False
b    False
c     True
d     True
```

2.3 Dictionary as Series

```
my_dict = {"red": 2000, "blue": 1000, "yellow": 100}
my_series = pd.Series(my_dict)
```

2.4 Operations between series

They consider the index name.

```
my_dict2 = {'red': 400, 'green': 100}
my_series2 = pd.Series(my_dict2)
```

```
>>> my_series + my_series2
blue         NaN
green        NaN
red       2400.0
yellow       NaN
dtype: float64
```

The addition is performed only on elements common to both the rest are filled with `NaN`

3 Working with data frames

- Can be defined as an ordered collection of columns
- It has two index arrays, `column` and `index` (row)

3.1 Creating a data frame.

```
# From a dictionary
data = {
    "color": ["blue", "green", "yellow", "red", "white"],
    "object": ["ball", "pen", "pencil", "peper", "mug"],
    "price": [4.2, 1.0, 0.6, 0.9, 1.7],
}

frame1 = pd.DataFrame(data)

# Only selected data/columns
frame2 = pd.DataFrame(data, columns=["object", "price"])

# Specify index
frame3 = pd.DataFrame(data, index=["one", "two", "three", "tour", "five"])

# From a matix with specific index and columns.
frame4 = pd.DataFrame(
    np.arange(16).reshape((4, 4)),  # 4 by 4 matrix
    index=["red", "blue", "yellow", "white"],
    columns=["ball", "pen", "pencil", "peper"],
)
```

```
>>> frame1
    color  object  price
0    blue    ball    4.2
```

```
1   green     pen    1.0
2  yellow  pencil    0.6
3     red   peper    0.9
4   white     mug    1.7

>>> frame2
   object  price
0    ball    4.2
1     pen    1.0
2  pencil    0.6
3   peper    0.9
4     mug    1.7

>>> frame3
        color  object  price
one      blue    ball    4.2
two     green     pen    1.0
three  yellow  pencil    0.6
tour      red   peper    0.9
five    white     mug    1.7

>>> frame4
        ball  pen  pencil  peper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

3.2 Retrieving values

```
1  # returns Index object
2  frame1.columns
3  frame1.index
4
5  # returns an array
6  frame1.values
7
8  # returns a single column (series abject)
9  frame1["price"]
10 frame1.price
11
12 # To select a row, returns a series object with columns as index.
13 # Second example is for multiple rows.
14 frame1.loc[2]
15 frame1.loc[[2, 4]]
16
17 # Returns the first row as a date frame
18 frame1[0:1]
19
20 # Single cell. For column 'object' and index 3
21 frame1["object"][3]
```

```
>>> frame1.values
array([['blue', 'ball', 4.2],
       ['green', 'pen', 1.0],
       ['yellow', 'pencil', 0.6],
       ['red', 'peper', 0.9],
       ['white', 'mug', 1.7]], dtype=object)

>>> frame1["price"]
0    4.2
1    1.0
2    0.6
3    0.9
```

```
4    1.7
Name: price, dtype: float64

>>> frame1.loc[2]
color     yellow
object    pencil
price        0.6
Name: 2, dtype: object

>>> frame1[0:1]
  color object  price
0  blue   ball    4.2
```

3.3 Assigning values

```
data = {
    "color": ["blue", "green", "yellow", "red", "white"],
    "object": ["ball", "pen", "pencil", "peper", "mug"],
    "price": [4.2, 1.0, 0.6, 0.9, 1.7],
}

frame0 = pd.DataFrame(data)  # backup for comparison
frame1 = pd.DataFrame(data)

# change the mame of index array
frame1.index.name = "id"

# Change name of colum array
frame1.columns.name = "item"

# Add a new column full of 10's
frame1["new"] = 10

# Add a new column
frame1["new"] = [3, 2, 1, 4, 5]

# Add a new column
frame1["new"] = pd.Series(np.arange(5))
```

```
>>> frame0
    color  object  price
0    blue    ball    4.2
1   green     pen    1.0
2  yellow  pencil    0.6
3     red   peper    0.9
4   white     mug    1.7

>>> frame1
item   color  object  price  new
id
0       blue    ball    4.2    0
1      green     pen    1.0    1
2     yellow  pencil    0.6    2
3        red   peper    0.9    3
4      white     mug    1.7    4
```

3.4 Check membership

```
# check if cell is either 1.0 or 'pen' returns bool ditatame
frame1.isin([1.0, 'pen'])
```

```
>>> frame1
item   color  object  price  new
id
0       blue    ball    4.2    0
1      green     pen    1.0    1
2     yellow  pencil    0.6    2
3        red   peper    0.9    3
4      white     mug    1.7    4

>>> frame1.isin([1.0, 'pen'])
item  color  object  price    new
id
0     False   False  False  False
1     False    True   True   True
2     False   False  False  False
3     False   False  False  False
4     False   False  False  False
```

Note that the integer `1` in the column **`new`** also matches the expression.

3.5 Delete a column

```
1 del frame1['new']
```

```
>>> frame1
item   color  object  price
id
0       blue    ball    4.2
1      green     pen    1.0
2     yellow  pencil    0.6
3        red   peper    0.9
4      white     mug    1.7
```

3.6 data frame from a nested dict.

```
1 nest_dict = {
2     "red": {2012: 22, 2013: 33},
3     "white": {2011: 13, 2012: 22, 2013: 16},
4     "blue": {2011: 17, 2012: 27, 2013: 18},
5 }
6 frame2 = pd.DataFrame(nest_dict)
```

```
>>> frame2
       red  white  blue
2012  22.0     22    27
2013  33.0     16    18
2011   NaN     13    17
```

External keys are assigned to columns whereas, internal keys to index. missing valves are `NaN`.

4 Indexes

- Are immutable (but see `reindex()`)
- Can be repeated

4.1 Methods on indices

```
# Find minimum value
frame2.idxmin()  # 2011

# Find maximum value
frame2.idxmax()  # 2013

# Test if all indices are unique
frame2.index.isunique  # bool

# 'Change' the indices
frame1.reindex(['a', 'b', 'c', 'd', 'e'])
# may not be useful for large data frames
```

The `idxmin` and `idxmax` methods are also applicable to strings using alphabetical ranking.

For numerical indices, where they don´t follow a perfect sequence, it is possible to interpolate the missing values. Two different methods are available, `ffill` and `bfill`. In the former, values are assigned copying the lower value of the available index. For example in the original series (`ser`), indices 1 and 2 are missing. Using the `ffill` method their value was taken from the value at index for index 0 (1), similarly for the missing index 4, its value is taking from index 3 (*i.e.* 5). For the latter, the upper available value is used to fill the missing ones.

```
# Fill missing indices over a range
ser = pd.Series([1, 5, 6, 3], index=[0, 3, 5, 6])
ser1 = ser.reindex(range(6), method='ffill')
ser2 = ser.reindex(range(6), method='bfill')
```

```
>>> ser
0    1
3    5
5    6
6    3
dtype: int64

>>> ser1
0    1
1    1
2    1
3    5
4    5
5    6
dtype: int64

>>> ser2
0    1
1    5
2    5
3    5
4    6
5    6
dtype: int64
```

4.2 Drop values using indices

```
frame = pd.DataFrame(np.arange(16).reshape((4,4)),
                     index=['red','blue','yellow','white'],
                     columns=['ball','pen','pencil','paper'])

```

```
# Delete 1 or multiple rows
frame_drop_row = frame.drop('yellow')
frame_drop_rows = frame.drop(['white', 'blue'])

# Delete columns (axis=1)
frame_drop_cols = frame.drop(['pen', 'pencil'], axis=1)
```

```
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
>>> frame_drop_rows
        ball  pen  pencil  paper
red        0    1       2      3
yellow     8    9      10     11
>>> frame_drop_cols
        ball  paper
red        0      3
blue       4      7
yellow     8     11
white     12      1
```

The same can be done to a series object.

5 Arithmetic between series and data frames

- It uses the index and, in the case of data frames, columns as well.
- Only common cells one operated on the rest one filled with `NaN`.
- The operation can be stated using math symbols `+`,`=-`, `=*`, `/` or with the method notation `add()`, `sub()`, `mul()`, `div()`. Note that the latter should be written as follows, `frame1.add(frame2)`
- If a series is added to a dataframe, the index of the former will be matched to the columns of the latter then, the operation will be carried across rows. Missing indices will be filled with `NaN`

```
frame = pd.DataFrame(
  np.arange(16).reshape((4, 4)),
  index=["red", "blue", "yellow", "white"],
  columns=["ball", "pen", "pencil", "paper"],
)
ser = pd.Series([0, 1, 2, 3],
                index=["ball", "pen", "pencil", "paper"])
```

```
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15

>>> frame - ser
        ball  pen  pencil  paper
red        0    0       0      0
blue       4    4       4      4
yellow     8    8       8      8
white     12   12      12     12
```

5.1 Other functions

- There are functions that operate on every cell such as universal functions `ufunc` for example `np.sqrt(frame1)`
- Functions can also be applied to entire rows or columns

```
# Function definition, returns a series
def f(x):
    return x.max() - x.min()


frame1 = pd.DataFrame(
  np.arange(16).reshape((4, 4)),
  index=["red", "blue", "yellow", "white"],
  columns=["ball", "pen", "pencil", "paper"],
)

# Function applied to rows
row_range = frame1.apply(f)

# Function applied to columns
col_range = frame1.apply(f, axis=1)


# Re-define function, returns a data frame
def f(x):
    return pd.Series([x.min(), x.max()], index=["min", "max"])


row_range2 = frame1.apply(f)
```

```
>>> row_range
ball      12
pen       12
pencil    12
paper     12
dtype: int64

>>> row_range2
     ball  pen  pencil  paper
min     0    1       2      3
max    12   13      14     15
```

- Statistical functions do not need to be nested inside `apply()`

```
frame = pd.DataFrame(
    np.arange(16).reshape((4, 4)),
    index=["red", "blue", "yellow", "white"],
    columns=["ball", "pen", "pencil", "paper"],
)

frame.sum()
frame.mean()
frame.describe()  # summary statistics

# Correlation and covariance
frame.corr() #Returns a data trame with correlation matrix.
frame.cov()
frame.corrwith(ser) # pairwise correlation with matching objects.
```

```
>>> frame.describe()

            ball        pen     pencil      paper
count   4.000000   4.000000   4.000000   4.000000
mean    6.000000   7.000000   8.000000   9.000000
std     5.163978   5.163978   5.163978   5.163978
min     0.000000   1.000000   2.000000   3.000000
25%     3.000000   4.000000   5.000000   6.000000
50%     6.000000   7.000000   8.000000   9.000000
75%     9.000000  10.000000  11.000000  12.000000
max    12.000000  13.000000  14.000000  15.000000
```

6 Sorting and ranking

6.1 Sorting

```
# Sorting indices (don´t change object)
# Sort from low to high (default)
frame_sort_rows = frame.sort_index()  # operates on rows
frame_sort_cols = frame.sort_index(axis=1)  # operates on columns

# Inverse sorting
frame_sort_rows_rev = frame.sort_index(ascending=False)

# Sorting values. It needs one or more ref. columns
frame_sort_rows_val1 = frame.sort_values(by='pen')
frame_sort_rows_val2 = frame.sort_values(by=['pen', 'pencil'])
```

```
>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15

>>> frame_sort_rows
        ball  pen  pencil  paper
blue       4    5       6      7
red        0    1       2      3
white     12   13      14     15
yellow     8    9      10     11

>>> frame_sort_cols
        ball  paper  pen  pencil
red        0      3    1       2
blue       4      7    5       6
yellow     8     11    9      10
white     12     15   13      14

>>> frame_sort_rows_val1
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15
```

6.2 Ranking

```
# Ranking
ser = pd.Series([5, 0, 3, 8, 4],
                index=['red', 'blue', 'yellow', 'white', 'green'])

# Ascending (default)
ser.rank()

# As they occurr (Doesn´t work)
# ser.rank(method='first')

# Inverse ranking
ser.rank(ascending=False)
```

```
>>> ser
red       5
blue      0
yellow    3
white     8
green     4
dtype: int64

>>> ser.rank()
red       4.0
blue      1.0
yellow    2.0
white     5.0
green     3.0
dtype: float64

>>> ser.rank(ascending=False)
red       2.0
blue      5.0
yellow    4.0
white     1.0
green     3.0
dtype: float64
```

7 `NaN` values (Not a number)

- On a date frame, how `NaN` are treated relative to the row should be specified or else if there is a single `NaN` the whole row will be removed.

```
# Assign NaN value
ser = pd.series([0, 1, 2, np.NaN, 4])
ser['0'] = None

# Removing NaN values.
ser.dropna()
ser[ser.notnull()]

# Remove row only if all the values are NaN
frame_nan = pd.DataFrame(
  np.arange(16).reshape((4, 4)),
  index=["red", "blue", "yellow", "white"],
  columns=["ball", "pen", "pencil", "paper"],
)
frame_nan.loc['blue'] = None  # Convert whole row to NaN
frame_nan['pen'][0] = np.NaN  # Convert a single cell to NaN

frame_noNaN = frame_nan.dropna()
```

```
frame_noNaN_row = frame_nan.dropna(how='all')

# Replace all NaN with 0
frame_noNaN_zero = frame_nan.fillna(0)

# Specific replacement of NaN
frame_noNaN_custom = frame_nan.fillna({'ball': 1,
                                       'pencil': -1,
                                       'paper': 0,
                                       'pen': 99})
```

```
>>> frame_nan
        ball   pen  pencil  paper
red      0.0   NaN     2.0    3.0
blue     NaN   NaN     NaN    NaN
yellow   8.0   9.0    10.0   11.0
white   12.0  13.0    14.0   15.0

>>> frame_noNaN
        ball   pen  pencil  paper
yellow   8.0   9.0    10.0   11.0
white   12.0  13.0    14.0   15.0

>>> frame_noNaN_row
        ball   pen  pencil  paper
red      0.0   NaN     2.0    3.0
yellow   8.0   9.0    10.0   11.0
white   12.0  13.0    14.0   15.0

>>> frame_noNaN_custom
        ball   pen  pencil  paper
red      0.0  99.0     2.0    3.0
blue     1.0  99.0    -1.0    0.0
yellow   8.0   9.0    10.0   11.0
white   12.0  13.0    14.0   15.0
```

8 Hierarchical indices

- Allows multiple levels of indexing on a single axis which in turn permits to work with multidimensional data on a two dimensional structure.

```
# Colors
w = 'white'; b = 'blue'; r = 'red'
# Directions
u = 'up'; d = 'down'; rg = 'right'; l = 'left'
# Values
ivalues = np.random.rand(8)

# Series
mser = pd.Series(ivalues,
                 index=[[w, w, w, b, b, r, r, r],
                        [u, d, rg, u, d, u, d, l]])
```

```
>>> mser
white  up       0.724634
       down     0.072744
       right    0.488042
blue   up       0.353832
       down     0.139895
```

```
red    up       0.411952
       down     0.571109
       left     0.582208
dtype: float64

>>> mser['white']
up       0.724634
down     0.072744
right    0.488042
dtype: float64

>>> mser[:, 'up']
white    0.724634
blue     0.353832
red      0.411952
dtype: float64
```

Note that in the second example the first index is used, whereas in the last example all the indices from the first index are selected (`:`) followed by an element from the second index (`up`).

8.1 Stacking and un-stacking

It is possible to convert the data frame with hierarchical index into one with column indices (*wide table*) using the `unstack()` method

```
mser.unstack() # Returns a regular df. 2nd index -> column
```

```
>>> mser.unstack()
          down      left     right        up
blue   0.139895       NaN       NaN  0.353832
red    0.571109  0.582208       NaN  0.411952
white  0.072744       NaN  0.488042  0.724634
```

Note that during un-stacking, missing values are filled with `NaN`.

To convert a wide table into a long one use the `stack()` method.

```
frame = pd.DataFrame(
    np.arange(16).reshape((4, 4)),
    index=["red", "blue", "yellow", "white"],
    columns=["ball", "pen", "pencil", "paper"],
)

frame_stacked = frame.stack()
```

```
>>>> frame
        ball  pen  pencil  paper
red        0    1       2      3
blue       4    5       6      7
yellow     8    9      10     11
white     12   13      14     15

>>> frame_stacked
red     ball       0
        pen        1
        pencil     2
        paper      3
blue    ball       4
        pen        5
        pencil     6
        paper      7
yellow  ball       8
        pen        9
```

```
        pencil    10
        paper     11
white   ball      12
        pen       13
        pencil    14
        paper     15
dtype: int64
```

8.2 Changing order of indices

```
w = 'white'; r = 'red'
u = 'up'; d = 'down'
pn = 'pen'; pp = 'paper'
ivalues = np.random.randn(16).reshape(4, 4)
mframe = pd.DataFrame(ivalues,
                      index=[[w, w, r, r], [u, d, u, d]],
                      columns=[[pn, pn, pp, pp], [1, 2, 1, 2]])

# set the names of indices. Second value is the nested index
mframe.columns.names = ['object', 'id']
mframe.index.names = ['colors', 'status']

# swap row indices
mframe_swaped = mframe.swaplevel('colors', 'status')

# sort the indices (level should be specified)
# axis=1 is used to sort columns
mframe_sorted_cols = mframe.sort_index(level='object', axis=1)
```

```
>>> mframe
object              pen               paper
id                    1         2         1         2
colors status
white  up     -1.337392 -0.794368  0.156086  0.458869
       down    0.504518 -1.168813  0.249712  1.008960
red    up      0.027992  1.175541 -2.158374  0.080673
       down   -0.694113 -1.253198 -0.243587  0.836986

>>> mframe_swaped
object              pen               paper
id                    1         2         1         2
status colors
up     white  -1.337392 -0.794368  0.156086  0.458869
down   white   0.504518 -1.168813  0.249712  1.008960
up     red     0.027992  1.175541 -2.158374  0.080673
down   red    -0.694113 -1.253198 -0.243587  0.836986

>>> mframe_sorted_cols
object            paper                 pen
id                    1         2         1         2
colors status
white  up      0.156086  0.458869 -1.337392 -0.794368
       down    0.249712  1.008960  0.504518 -1.168813
red    up     -2.158374  0.080673  0.027992  1.175541
       down   -0.243587  0.836986 -0.694113 -1.253198
```

8.3 Summary statistics

```
# mframe.sum(level='colors') # Deprecated
mframe.groupby(level='colors').sum()
```

```
>>> mframe.groupby(level='colors').sum()
object       pen               paper
id             1         2         1         2
colors
red    -0.666121 -0.077656 -2.401961  0.917659
white  -0.832874 -1.963181  0.405797  1.467828
```

```
# mframe.sum(level='id', axis=1) # Deprecated
mframe.groupby(level='id', axis=1).sum()
```

```
>>> mframe.groupby(level='id', axis=1).sum()
id                   1         2
colors status
white  up     -1.181306 -0.335499
       down    0.754230 -0.159853
red    up     -2.130382  1.256214
       down   -0.937700 -0.416212
```