

Pandas

Aprendizaje Automático para la Robótica
Máster Universitario en Ingeniería Industrial

Departamento de Automática

Objectives

1. Introduce Series and DataFrame data structures
2. Understand Pandas features
3. Fluent data manipulation with Pandas
4. Data exploration

Bibliography

Jake VanderPlas. *Python Data Science Handbook*. Chapter 3. O'Reilly. (Link).

Table of Contents

1. Introduction
2. The Pandas Series object
3. The Pandas DataFrame object
 - DataFrame concept
 - Constructing DataFrame objects
4. Data indexing and selection
 - Series
 - Loc, iloc and ix
5. Operating on data
 - Overview
 - Missing data
6. Combining datasets
 - `pd.concat()`
 - `pd.merge()`
7. Aggregation in Pandas
 - Grouping in Pandas

Introduction

A DS/ML workflow needs more features

- Missing data
- Data input
- Operations on groups
- Label columns and rows

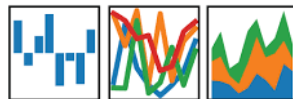
Pandas provides all those features, and more

- Pandas = **PAN**el **DA**ta **S**ystem
- Built on NumPy's ndarray
- Provides **dataframes**

Pandas provides two main objects

- **Series** and **DataFrame**

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$


Convention

```
^^I^^I^^I import numpy as
      np
^^I^^I^^I import pandas as
      pd
^^I^^I^^I
```

The Pandas Series object (I)

A **Series** is a one-dimensional array of indexed data

- NumPy arrays indices are implicit (i.e. its position)
- Series indices are explicit, and can be any type

INDEX	VALUES
'a'	0.25
'b'	0.5
'c'	0.75
'd'	0.99

Two attributes

- **values:** ndarray
- **index:** pd.Index object

Two indices

- Implicit: Regular index
- Explicit: Custom index

```

^^I^^I^^Idata = pd.Series
    ([0.25, 0.5, 0.75,
      1.0])
^^I^^I^^Idata.values
^^I^^I^^Idata.index
^^I^^I^^Idata[1:3]
^^I^^I^^I
    
```

The Pandas Series object (II)

Custom indices

```

^^I^^I^^IIn [1] : data = pd.Series([0.25, 0.5, 0.75,
                                     1.0],
                                     index=['a', 'b', 'c', 'd'])
^^I^^I^^IOut [1]:
^^I^^I^^Ia      0.25
^^I^^I^^Ib      0.50
^^I^^I^^Ic      0.75
^^I^^I^^Id      1.00
^^I^^I^^Idtype: float64

^^I^^I^^IIn [3]: data['a']
^^I^^I^^IOut [2]: 0.25
^^I^^I^^IIn [4]: data[0]
^^I^^I^^IOut [3]: 0.25
^^I^^I^^I

```

The Pandas DataFrame object

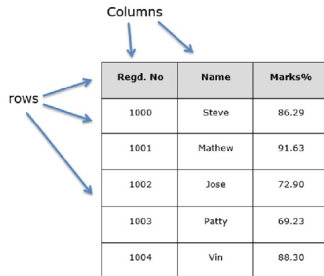
Dataframe concept (I)

A DataFrame is a 2-D tabular data structure

- Similar to a spreadsheet
- Homogeneous columns
- Heterogeneous rows

Two read-only attributes, both `pd.Index`

- `index`: Rows
- `columns`: Columns



The diagram shows a table with 3 columns and 6 rows. The first row is a header with columns 'Regd. No', 'Name', and 'Marks%'. The subsequent rows contain data for students with registration numbers 1000 through 1004. Annotations include 'Columns' with arrows pointing to the header row, and 'rows' with arrows pointing to the first, second, and last rows of the data.

Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

(Source)

The Pandas DataFrame object

Dataframe concept (II)

DataFrame example

```
In [1]: import seaborn as sns
```

```
In [2]: iris = sns.load_dataset('iris')
```

```
In [3]: iris.head()
```

```
Out [1]:
```

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

```
In [246]: iris.columns
```

```
Out [246]:
```

```
Index(['sepal_length', 'sepal_width', 'petal_length',  
      'petal_width', 'species'], dtype='object')
```

```
^^ I ^^ I
```


The Pandas DataFrame object

Constructing DataFrame objects (I)

Manual initialization

- From a single Series object
`pd.DataFrame(population, columns=['population'])`
- From several Series objects
`pd.DataFrame('population': population, 'area': area)`
- From a dictionary
`pd.DataFrame([{'a': 0, 'b': 0}, {'a': 1, 'b': 2}])`
- From a NumPy 2-D array
`pd.DataFrame(np.random.rand(3, 2),
 columns=['foo', 'bar'], index=['a', 'b', 'c'])`

The Pandas DataFrame object

Constructing DataFrame objects (II)

Read from a file

- CSV (very common!!!): `pd.read_csv('filename.csv')`
- Excel:
`pd.read_excel('filename.xlsx', sheetname='mysheet')`

CSV example

```
# This CSV file contains data about weights and heights
"id", "weight", "height", "sex", "race"
1, 143.5, 81.6, "Female", "White"
2, 109.1, 83.7, "Female", "Black"
4, 104.8, 54.6, "Female", "Hisp"
7, 130.2, 81.7, "Male", "White"
```

CSV can be exported from MS Excel or programmatically

Data indexing and selection

Series

Dictionary-like syntax

```
^^I^^I^^I>>> data = pd.Series
    ([0.25, 0.5, 0.75, 1.0],
     index=['a', 'b', 'c', 'd'])
```

```
^^I^^I^^I>>> 'a' in data
^^I^^I^^ITrue
```

```
^^I^^I^^I>>> data.keys()
^^I^^I^^IIndex(['a', 'b', 'c'],
    dtype='object')
```

```
^^I^^I^^I>>> list(data.items())
^^I^^I^^I[( 'a', 0.25), ('b',
    0.5), ('c', 0.75)]
```

Array-like syntax

```
^^I^^I^^I>> data['a':'c'] #
    Explicit index
```

```
^^I^^I^^IIa      0.25
^^I^^I^^IIb      0.50
^^I^^I^^IIc      0.75
```

```
^^I^^I^^Idtype: float64
```

```
^^I^^I^^I>> data[0:2] # Implicit
    index
```

```
^^I^^I^^IIa      0.25
^^I^^I^^IIb      0.50
^^I^^I^^Idtype: float64
```

```
^^I^^I^^I>> data[data > 0.5] #
    Masking
```

```
^^I^^I^^IIc      0.75
^^I^^I^^IID      1.00
^^I^^I^^Idtype: float64
```

```
^^I^^I^^I>> data[['b', 'c']] #
    Fancy index
```

Data indexing and selection

DataFrame

Dictionary-like syntax

```
^^I^^I^^I>>> data['area']
^^I^^I^^I>>> data.area
^^I^^I^^I>>> data.area is data['
    area']
^^I^^I^^ITrue
^^I^^I^^I>>> data['density'] =
    data['pop']/data['area']
^^I^^I^^I
```

Array-like syntax

```
^^I^^I^^I>>> data.values # Get
    values array
^^I^^I^^I>>> data.T # Transpose
^^I^^I^^I>>> data[0] # First row
^^I^^I^^I>>> data['area'] # Area
    column
^^I^^I^^I
```

Remember indexing conventions

- Indexing refers to columns (`data['area']`)
- Slicing refers to rows (`data['Florida':'Illinois']`)
- Masking refers to rows (`data[data.density > 100]`)

Data indexing and selection

loc, iloc and ix

Two types of indices in Pandas

- Explicit and implicit
- Indexing (`data[0]`) is explicit
- Slicing (`data[:2]`) is implicit (Python-like)
- Source of troubles!

Pandas makes explicit the used scheme

- `loc`: Explicit index
- `iloc`: Implicit index
- `ix`: Hybrid

```
^^I^^I^^I# Series
^^I^^I^^I>>> serie.loc[1]
^^I^^I^^I>>> serie.loc[1:3]
^^I^^I^^I>>> serie.iloc[1]
^^I^^I^^I>>> serie.iloc[1:3]

^^I^^I^^I# Dataframes
^^I^^I^^I>>> df.iloc[:3, :2]
^^I^^I^^I>>> df.loc[: 'illinois',
    : 'pop']
^^I^^I^^I>>> df.ix[:3, : 'pop']
^^I^^I^^I>>> df.loc[df.data>100,
    [ 'pop', 'density']]
^^I^^I^^I>>> df.iloc[0, 2] = 90
^^I^^I^^I
```

Operating on data

Overview (I)

Pandas fully supports NumPy's ufuncs

- Efficient computations

Additional Pandas features

- Index and column name preservation
- Index aligning
- Easy data combination

```

^^I^^I^^I>>> rng = np.random.
                RandomState(42)
^^I^^I^^I>>> df = pd.DataFrame(rng.
                randint(0, 10, (3,4)))
^^I^^I^^I>>> df = pd.DataFrame(rng.
                randint(0, 10, (3,4)), columns=['A
                ', 'B', 'C', 'D'])
^^I^^I^^I>>> print(df)
^^I^^I^^I    A    B    C    D
^^I^^I^^I0    7    2    5    4
^^I^^I^^I1    1    7    5    1
^^I^^I^^I2    4    0    9    5
^^I^^I^^I>>> np.sin(df * np.pi / 4)
^^I^^I    A          B          C          D
^^I^^I^^I 0   -7.07e-01    1.0   -0.7    1.22e
                -16
^^I^^I^^I 1    7.07e-01   -0.7   -0.7    7.07e
                -01
^^I^^I^^I 2    1.22e-16    0.0    0.7   -7.07e
                Pandas
  
```

Operating on data

Overview (II)

Index preservation

```

^^I^^I^^I>>> A = pd.Series([2, 4, 6], index=[0, 1,
2])
^^I^^I^^I>>> B = pd.Series([1, 3, 5], index=[1, 2,
3])
^^I^^I^^I>>> A + B
^^I^^I^^Io      NaN
^^I^^I^^I1      5.0
^^I^^I^^I2      9.0
^^I^^I^^I3      NaN
^^I^^I^^Idtype: float64
^^I^^I^^I>>> A.add(B, fill_value=0)
^^I^^I^^Io      2.0
^^I^^I^^I1      5.0
^^I^^I^^I2      9.0
^^I^^I^^I3      5.0
^^I^^I^^Idtype: float64
^^I^^I^^I

```

Operating on data

Missing data (I)

NumPy supports missing data in floating-point data

- Specific value defined by IEEE
- Available as `np.nan`

Pandas supports missing data through two mechanisms

- None object, interpreted as NaN (Not a Number)
- `np.nan`: for floating-point data
- Almost automatic NaN handling (types upcast)

```

^^I^^I^^I>>> pd.Series([1, np.nan,
                        2, None])
^^I^^I^^Io      1.0
^^I^^I^^I1      NaN
^^I^^I^^I2      2.0
^^I^^I^^I3      NaN
^^I^^I^^Idtype: float64
^^I^^I^^I
    
```


Pandas

Missing data (II)

Useful functions for missing data

- `isnull()`: Boolean mask with missing data
- `notnull()`: Opposite of `isnull()`
- `dropna()`: Filtered data
- `fillna()`: NaNs filled

```

^^I^^I^^I>>> data = pd.Series
                ([1, np.nan, 'hello', None
                ])
^^I^^I^^I>>> data[data.notnull()
                ]
^^I^^I^^I I_0      1
^^I^^I^^I I_2    hello
^^I^^I^^I Idtype: object

^^I^^I^^I>>> data.dropna()
^^I^^I^^I I_0      1
^^I^^I^^I I_2    hello
^^I^^I^^I Idtype: object

^^I^^I^^I>>> data.fillna(0)
^^I^^I^^I I_0      1
^^I^^I^^I I_1      0
^^I^^I^^I I_2    hello
^^I^^I^^I I_3      0

```

Combining datasets

pd.concat() (I)

Many times we need to combine two or more datasets

- Pandas provides `pd.concat()`, `append()` and `pd.merge()`

pd.concat() signature

```
^^ I ^^ I
pd.concat( objs , axis=0 , join='outer' ,
           join_axes=None , ignore_index=False , keys
           =None , levels=None , names=None ,
           verify_integrity=False , copy=True )
^^ I ^^ I
```

By default, `pd.concat()` joins rows preserving index

- `axis`: Join columns (`axis=1`)
- `verify_integrity`: Raise error if duplicates (`verify_integrity=True`)
- `ignore_index`: Create new index (`ignore_index=True`)
- `join`: Can be 'outer' (union) or 'inner' (intersection)

Combining datasets

pd.concat() (II)

```
>> df1 = pd.DataFrame([{'A': 'A0', 'B': 'B0'}, {'A': 'A1', 'B': 'B1'}])
>> df2 = pd.DataFrame([{'A': 'A2', 'B': 'B2'}, {'A': 'A3', 'B': 'B3'}])

>> print(df1), print(df2); print(pd.concat([df1, df2]))
  A  B      A  B      A  B
0 A0 B0  0 A2 B2  0 A0 B0
1 A1 B1  1 A3 B3  1 A1 B1
                  0 A2 B2
                  1 A3 B3

>> pd.concat([df1, df2], axis=1)
  A  B  A  B
0 A0 B0 A2 B2
1 A1 B1 A3 B3
>> df1.append(df2)
^^ I ^^ I ^^ I
```

Combining datasets

`pd.merge()` (I)

Merging based on relational algebra

- Similar to databases tables joins
- Pretty intelligent figuring out the desired output
- By default, join dataframes using shared columns names

Combining datasets

pd.merge() (II)

One-to-one

```
>> print(df1); print(df2)
employee      group
0      Bob  Accounting
1      Jake  Engineering
2      Lisa  Engineering
3      Sue      HR

employee  hire_date
0      Lisa      2004
1      Bob      2008
2      Jake      2012
3      Sue      2014

>> print(pd.merge(df1, df2))
employee  group  hire_date
0      Bob  Accounting      2008
1      Jake  Engineering      2012
2      Lisa  Engineering      2004
3      Sue  HR      2014
^^ I ^^ I
```

Many-to-one

```
>>> print(df3); print(df4)
employee  group  hire_date
0      Bob  Accounting      2008
1      Jake  Engineering      2012
2      Lisa  Engineering      2004
3      Sue      HR      2014

group  supervisor
0  Accounting  Carly
1  Engineering  Guido
2      HR      Steve

>> print(pd.merge(df3, df4))
employee  group  hire_date  supervisor
0      Bob  Accounting      2008  Carly
1      Jake  Engineering      2012  Guido
2      Lisa  Engineering      2004  Guido
3      Sue      HR      2014  Steve
^^ I ^^ I
```

Combining datasets

pd.merge() (III)

Many-to-many

```
>>> print(df1); print(df5)
```

	employee	group		group	skills
0	Bob	Accounting	0	Accounting	math
1	Jake	Engineering	1	Accounting	spreadsheets
2	Lisa	Engineering	2	Engineering	coding
3	Sue	HR	3	Engineering	linux
			4	HR	spreadsheets
			5	HR	organization

```
>>> pd.merge(df1, df5)
```

	employee	group	skills
0	Bob	Accounting	math
1	Bob	Accounting	spreadsheets
2	Jake	Engineering	coding
3	Jake	Engineering	linux
4	Lisa	Engineering	coding
5	Lisa	Engineering	linux
6	Sue	HR	spreadsheets
7	Sue	HR	organization

```
AA I AA I
```

Pandas

Combining datasets

pd.merge() (IV)

pd.merge() signature

```

^^ I ^^ I
pd.merge( left , right , how= 'inner' , on=
        None , left_on= None , right_on= None ,
        left_index= False , right_index= False ,
        sort= False , suffixes= ( '_x' , '_y' ) , copy=
        True , indicator= False , validate= None )
^^ I ^^ I
    
```

Arguments:

- **on**: Key column name
- **left_on**: Left table key column name
- **right_on**: Right table key column name
- **how**: Set arithmetic, 'inner' (default, intersection), 'outer' (union, fills missings with NaNs), 'left' (left entries), 'right' (right entries)

Combining datasets

pd.merge() (V)

```
>>> A
   lkey  value
0   foo    1
1   bar    2
2   baz    3
3   foo    4

>>> B
   rkey  value
0   foo    5
1   bar    6
2   qux    7
3   bar    8

>>> A.merge(B, left_on='lkey', right_on='rkey', how='outer')
   lkey  value_x  rkey  value_y
0   foo    1     foo    5
1   foo    4     foo    5
2   bar    2     bar    6
3   bar    2     bar    8
4   baz    3     NaN    NaN
5   NaN    NaN    qux    7
^^ I ^^ I
```


Aggregation in Pandas (I)

The first step in data analysis is summarization

- First contact with data
- Insight to the dataset

Aggregation methods

- Applied to columns

AGGREGATION	DESCRIPTION
<code>count()</code>	Total number of items
<code>first(), last()</code>	First and last item
<code>mean(), median()</code>	Mean and median
<code>min(), max()</code>	Minimum and maximum
<code>std(), var()</code>	Standard dev. and variance
<code>mad()</code>	Mean absolute deviation
<code>prod()</code>	Product of all items
<code>sum()</code>	Sum of all items
<code>describe()</code>	Data summary

```
>>> import seaborn as sns
>>> planets = sns.load_dataset('planets')
>>> planets.head()
   method  number  orbital_period  mass  distance  year
0  Radial  Velocity  1         269.300      7.10      77.40  2006
1  Radial  Velocity  1         874.774      2.21      56.95  2008
2  Radial  Velocity  1         763.000      2.60      19.84  2011
3  Radial  Velocity  1         326.030     19.40     110.62  2007
4  Radial  Velocity  1         516.220     10.50     119.47  2009
>>> planets.dropna().describe()
      number  orbital_period      mass  distance      year
count    498.00      498.000000    498.00    498.0000    498.000
mean       1.73      835.778671      2.50     52.0682    2007.377
std       1.17     1469.128259      3.63     46.5960      4.167
min       1.00      1328300      0.00      1.3500    1989.000
25%       1.00      38.272250      0.21     24.4975    2005.000
50%       1.00     357.000000      1.24     39.9400    2009.000
75%       2.00     999.600000      2.86     59.3325    2011.000
max       6.00    17337.500000     25.00    354.0000    2014.000
>>> planets.mean()
number           1.785507
orbital_period   2002.917596
mass             2.638161
distance         264.069282
year            2009.070531
dtype: float64
^^I^^I^^I
```

Grouping in Pandas (I)

Aggregation is generally used ...

- ... good to operate with the whole dataset ...
- ... but also is usually insufficient

We need conditional aggregations

- Aggregate conditionally on some label

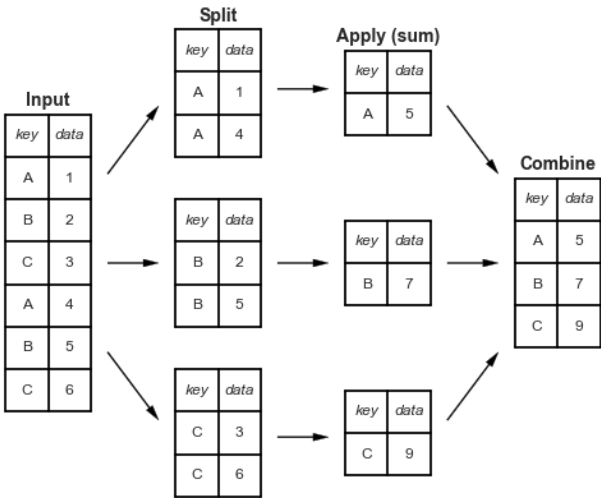
This is done with the operation `groupby` (yes, that name comes from SQL)

- Example: `df.groupby("key")`

Three tasks in one step

1. Split: Break up dependening on a key
2. Apply: Compute some function
3. Combine: Merge results into an output

Grouping in Pandas (II)



Grouping in Pandas (III)

```

>>> df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                        'data': range(6)})

>>> print(df)
   key  data
0    A     0
1    B     1
2    C     2
3    A     3
4    B     4
5    C     5
>>> df.groupby('key')
<pandas.core.groupby.groupby.DataFrameGroupBy object at 0
x102685438 >
>>> df.groupby('key').sum()
   data
key
A      3
B      5
C      7
^^ I ^^ I ^^ I
    
```

Grouping in Pandas (IV)

Several mapping methods available

- List
`df.groupby([2,3,4,1]).sum()`
- Dictionary
`df.groupby('A': 'vowel', 'B': 'consonant', 'C': 'vowel')`
- Python function
`df.groupby(str.lower)`
- Multiple keys
`planets.groupby(['method', 'year'])`
- Mixed keys
`df.groupby(['key1', 'key2', str.lower])`

Grouping in Pandas (V)

The method `groupby()` returns an object `groupby`

- Basically, it is a collection of dataframes
`planets.groupby('method').get_group('Transit')`
- Column selection as dataframe
`planets.groupby('method')['year']`

Interesting `groupby` attribute, `groups`

- Dictionary with groups
`planets.groupby('method').groups`
- Compatible with the `len()` method
`len(planets.groupby('method'))`

Grouping in Pandas (VI)

Usual operations with groupings

- Aggregation:

```
df.groupby('key').aggregate(['min', np.median, max])
df.groupby('key').aggregate('data1': 'min', 'data2': 'max')
```
- Filtering:

```
planets.groupby('method').filter(lambda x:
x['distance'].mean() > 50.)
```
- Transformation:

```
df.groupby('key').transform(lambda x: x - x.mean())
```

`Apply()`: Apply arbitrary function and combine results

- Takes a function as argument that takes a DataFrame

```
planets.groupby("method").apply(lambda x: x / x.sum())
```


Grouping in Pandas (VII)

Grouping by decade

```

^^I^^ Idecade = 10 * (planets['year'] // 10)
^^I^^ Idecade = decade.astype(str) + 's'
^^I^^ Idecade.name = 'decade'
^^I^^ Iplanets.groupby(['method', decade])['number'
    ].sum().unstack().fillna(0)
^^I^^ I
  
```