PANDAS 基础

金　林
中南财经政法大学统计系
jinlin82@qq.com

2020 年 2 月 4 日

Facts

1. Original author(s): Wes McKinney
2. Initial release: 2008
3. Stable release: 0.19.1 / November 3, 2016;
4. Website: `http://pandas.pydata.org`

What is Pandas?

1. pandas is a software library written for the Python programming language for data manipulation and analysis.
2. In particular, it offers data structures and operations for manipulating numerical tables and time series.
3. It aims to be the fundamental high-level building block for doing practical, real world data analysis in Python.
4. Additionally, it has the broader goal of becoming the most powerful and flexible open source data analysis / manipulation tool available in any language.

Library features

1. The two primary data structures of pandas, Series (1-dimensional) and DataFrame (2-dimensional), handle the vast majority of typical use cases in finance, statistics, social science, and many areas of engineering.
2. For R users, DataFrame provides everything that R′ s data.frame provides and much more.
3. pandas is fast. Many of the low-level algorithmic bits have been extensively tweaked in Cython code. However, as with anything else generalization usually sacrifices performance.
4. pandas is a dependency of statsmodels, making it an important part of the statistical computing ecosystem in Python.
5. pandas has been used extensively in production in financial applications.

things that pandas does well

1. Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data
2. Size mutability: columns can be inserted and deleted from DataFrame and higher dimensional objects
3. Automatic and explicit data alignment: objects can be explicitly aligned to a set of labels, or the user can simply ignore the labels and let Series, DataFrame, etc. automatically align the data for you in computations
4. Powerful, flexible group by functionality to perform split-apply-combine operations on data sets, for both aggregating and transforming data
5. Make it easy to convert ragged, differently-indexed data in other Python and NumPy data structures into DataFrame objects

things that pandas does well

6. Intuitive merging and joining data sets
7. Intelligent label-based slicing, fancy indexing, and subsetting of large data sets
8. Flexible reshaping and pivoting of data sets
9. Hierarchical labeling of axes (possible to have multiple labels per tick)
10. Robust IO tools for loading data from flat files (CSV and delimited), Excel files, databases, and saving / loading data from the ultrafast HDF5 format
11. Time series-specific functionality: date range generation and frequency conversion, moving window statistics, moving window linear regressions, date shifting and lagging, etc.

pandas consists of the following elements

1. A set of labeled array data structures, the primary of which are Series and DataFrame
2. Index objects enabling both simple axis indexing and multi-level / hierarchical axis indexing
3. An integrated group by engine for aggregating and transforming data sets
4. Date range generation (date_range) and custom date offsets enabling the implementation of customized frequen-
5. cies
6. Input/Output tools: loading tabular data from flat files (CSV, delimited, Excel 2003), and saving and loading
7. pandas objects from the fast and efficient PyTables/HDF5 format.
8. Memory-efficient "sparse" versions of the standard data structures for storing data that is mostly missing or
9. mostly constant (some fixed value)
10. Moving window statistics (rolling mean, rolling standard deviation, etc.)

CSV

1. Writing to a csv file: `df.to_csv('foo.csv')`
2. Reading from a csv file: `pd.read_csv('foo.csv')`

Excel

1. Writing to an excel file:
   `df.to_excel('foo.xlsx', sheet_name='Sheet1')`
2. Reading from an excel file:
   `pd.read_excel('foo.xlsx', 'Sheet1', index_col=None, na_`

Object Creation

1. Creating a Series by passing a list of values, letting pandas create a default integer index:

```
import pandas as pd
import numpy as np
pd.Series([1, 2, 5, np.nan, 8])
```

1. Creating a DataFrame by passing a numpy array, with a datetime index and labeled columns:

```
dates = pd.date_range('20130101', periods=6)
df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=
    list('ABCD'))
```

1. Creating a DataFrame by passing a dict of objects that can be converted to series-like.

```

df2 = pd.DataFrame({ 'A':1.,
^^I^^I      'B':pd.Timestamp('20130102'),
^^I^^I      'C' : pd.Series(1,index=list(range(4)),dtype='
    float32'),
^^I^^I      'D' : np.array([3]*4,dtype='int32'),
```

Viewing Data

1. See the top & bottom rows of the frame
    1. df.head()
    2. df.tail(3)
2. Display the index, columns, and the underlying numpy data
    1. df.index
    2. df.columns
    3. df.values
3. Describe shows a quick statistic summary of your data
    1. df.describe()
4. Transposing your data
    1. df.T

排序

1. Sorting by an axis `df.sort_index(axis=1, ascending=False)`
2. Sorting by values `df.sort_values(by='B')`

使用中括号 [ ] 选取

1. Selecting a single column, which yields a Series, equivalent to df.A

```
df['A']
```

1. 选择连续行

```
df[2:4]
```

使用标签（label）选取：.loc[ ]

1. 必须都是使用标签，不是数字
2. 标签可以使用冒号
3. DataFrame 使用两个下标，中间用逗号分开，全选的要使用冒号
4. 可以用于非连续行列选取

```
df2 = pd.DataFrame(np.random.randn(5,4), columns=list('ABCD'),
^^I^^I   index=pd.date_range('20130101',periods=5))

df2.loc[2:3] ### 错误
df2.loc['20130102':'20130104']
df2.loc[:,['A','C']]
```

使用位置选取：.iloc[ ]

The `.iloc` attribute is the primary access method. The following are valid inputs:

1. An integer e.g. 5
2. A list or array of integers [4, 3, 0]
3. A slice object with ints 1:7
4. A boolean array
5. When slicing, the start bounds is included, while the upper bound is excluded.

```

df2 = pd.DataFrame(np.random.randn(5,4), columns=list('ABCD'),
^^I^^I   index=pd.date_range('20130101',periods=5))

df2.iloc[2]
df2.iloc[2,3]
df2.iloc[[0,2,4]]
df2.iloc[[0,2,4], :3]
```

DataFrame.at 和 DataFrame.iat

1. `DataFrame.at` :Access a single value for a row/column label pair.
2. `DataFrame.iat` :Access a single value for a row/column pair by integer position.

Boolean indexing(逻辑值下标)

1. The operators are: $|$ for or, & for and, and $\sim$ for not.
2. These must be grouped by using parentheses.
3. Using a boolean vector to index a Series works exactly as in a numpy ndarray

```

###Using a single column′ s values to select data.
df[df.A > 0]
df[df > 0]

df2.loc[df2['A'] > 0, 'A':'C']

df2.iloc[list(df2['A'] > 0), 0:3]
```

Indexing with `isin`

1. Consider the isin method of Series, which returns a boolean vector that is true wherever the Series elements exist in the passed list.

```
s = pd.Series(np.arange(5), index=np.arange(5)[::-1], dtype='
    int64')
s[s.isin([2,4,6])]
```

1. DataFrame also has an `isin` method. When calling isin, pass a set of values as either an array or dict.
2. If values is an array, isin returns a DataFrame of booleans that is the same shape as the original DataFrame.

```
df = pd.DataFrame({'vals': [1, 2, 3, 4], 'ids': ['a', 'b', 'f',
     'n'],
^^I^^I    'ids2': ['a', 'n', 'c', 'n']})
values = ['a', 'b', 1, 3]
df.isin(values)
```

Indexing with `isin`

1. Oftentimes you' ll want to match certain values with certain columns. Just make values a dict where the key is the column, and the value is a list of items you want to check for.

```python
values = {'ids': ['a', 'b'], 'vals': [1, 3]}
df.isin(values)
```

1. Combine DataFrame' s isin with the `any()` and `all()` methods to quickly select subsets of your data that meet a given criteria. To select a row where each column meets its own criterion:

```python
values = {'ids': ['a', 'b'], 'ids2': ['a', 'c'], 'vals': [1,
    3]}
row_mask = df.isin(values).all(1)
df[row_mask]
```

The `where()` Method

1. Selecting values from a Series with a boolean vector generally returns a subset of the data.
2. To guarantee that selection output has the same shape as the original data, you can use the where method in Series and DataFrame.
3. In addition, where takes an optional other argument for replacement of values where the condition is False, in the returned copy.
4. Note: The signature for DataFrame.where() differs from numpy.where(). Roughly df1.where(m, df2) is equivalent to np.where(m, df1, df2).

```
s[s > 0]
s.where[s>0]
df.where(df < 0, -df)
```

Duplicate Data

1. If you want to identify and remove duplicate rows in a DataFrame, there are two methods that will help: `duplicated` and `drop_duplicates`.
2. Each takes as an argument the columns to use to identify duplicated rows.
3. duplicated returns a boolean vector whose length is the number of rows, and which indicates whether a row is duplicated.
4. drop_duplicates removes duplicate rows.
5. By default, the first observed row of a duplicate set is considered unique, but each method has a keep parameter to specify targets to be kept.
   - keep='first' (default): mark / drop duplicates except for the first occurrence.
   - keep='last': mark / drop duplicates except for the last occurrence.
   - keep=False: mark / drop all duplicates.

Duplicate Data

```
df2 = pd.DataFrame({'a': ['one', 'one', 'two', 'two', 'two', '
    three', 'four'],
^^I^^I    'b': ['x', 'y', 'x', 'y', 'x', 'x', 'x'],
^^I^^I    'c': np.random.randn(7)})

df2.duplicated('a')
df2.duplicated('a', keep='last')
df2.duplicated('a', keep=False)
df2.drop_duplicates('a')
df2.drop_duplicates('a', keep='last')
df2.drop_duplicates('a', keep=False)
### pass a list of columns to identify duplications.
df2.duplicated(['a', 'b'])
df2.drop_duplicates(['a', 'b'])
```

Missing Data

1. pandas primarily uses the value np.nan to represent missing data. It is by default not included in computations.
2. To drop any rows that have missing data. `df1.dropna(how='any')`
3. Filling missing data: `df1.fillna(value=5)`
4. To get the boolean mask where values are nan: `pd.isna(df1)`

统计

1. Performing a descriptive statistic:
   - `df.mean()`
   - `df.mean(1)`
   - `np.mean(np.array(df))`

apply 函数

1. Applying functions to the data:
   - `df.apply(np.cumsum)`
2. `df.apply(lambda x: x.max() - x.min())`

字符处理

1. Series is equipped with a set of string processing methods in the str attribute that make it easy to operate on each element of the array.

```
s = pd.Series(['A', 'B', 'C', 'Aaba', 'Baca', np.nan, 'CABA', '
    dog', 'cat'])
s.str.lower()
```

concat

1. Concatenate pandas objects along a particular axis with optional set logic along the other axes.
2. The concat() function (in the main pandas namespace) does all of the heavy lifting of performing concatenation operations along an axis while performing optional set logic (union or intersection) of the indexes (if any) on the other axes.
3. Like its sibling function on ndarrays, numpy.concatenate, pandas.concat takes a list or dict of homogeneously-typed objects and concatenates them with some configurable handling of “what to do with the other axes”

```
df = pd.DataFrame(np.random.randn(10, 4))
pieces = [df[:3], df[3:7], df[7:]]
pd.concat(pieces)
```

appdend

1. A useful shortcut to concat() are the append() instance methods on Series and DataFrame.
2. These methods actually predated concat. They concatenate along axis=0, namely the index.

```
df = pd.DataFrame(np.random.randn(8, 4), columns=['A','B','C','D'])
s = df.iloc[3]
df.append(s)
df.append(s, ignore_index=True)
```

join

1. pandas has full-featured, high performance in-memory join operations idiomatically very similar to relational databases like SQL.
2. These methods perform significantly better (in some cases well over an order of magnitude better) than other open source implementations (like `base::merge.data.frame` in R).
3. The reason for this is careful algorithmic design and the internal layout of the data in DataFrame.
4. pandas provides a single function, `merge()`, as the entry point for all standard database join operations between DataFrame or named Series objects
5. the related `join()` method, uses merge internally for the index-on-index (by default) and column(s)-on-index join.

Group By: split-apply-combine

- By “group by” we are referring to a process involving one or more of the following steps:
  1. Splitting the data into groups based on some criteria.
  2. Applying a function to each group independently.
  3. Combining the results into a data structure.
- Out of these, the split step is the most straightforward. In fact, in many situations we may wish to split the data set into groups and do something with those groups.

apply step

- Aggregation: compute a summary statistic (or statistics) for each group. Some examples:
  1. Compute group sums or means.
  2. Compute group sizes / counts.
- Transformation: perform some group-specific computations and return a like-indexed object. Some examples:
  1. Standardize data (zscore) within a group.
  2. Filling NAs within groups with a value derived from each group.
- Filtration: discard some groups, according to a group-wise computation that evaluates True or False. Some examples:
  1. Discard data that belongs to groups with only a few members.
  2. Filter out data based on the group sum or mean.
- Some combination of the above: GroupBy will examine the results of the apply step and try to return a sensibly combined result if it doesn' t fit into either of the above two categories.

例子

```

df = pd.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
^^I^^I^^I   'foo', 'bar', 'foo', 'foo'],
^^I^^I      'B': ['one', 'one', 'two', 'three',
^^I^^I^^I   'two', 'two', 'one', 'three'],
^^I^^I      'C': np.random.randn(8),
^^I^^I      'D': np.random.randn(8)})

df.groupby('A').sum()
df.groupby(['A','B']).sum()
```

- Aggregation

Applying multiple functions at once

1. With grouped Series you can also pass a list or dict of functions to do aggregation with, outputting a DataFrame.
2. On a grouped DataFrame, you can pass a list of functions to apply to each column, which produces an aggregated result with a hierarchical index.

```
df.groupby(['A','B']).agg([np.sum, np.mean, np.std])
```

Applying different functions to DataFrame columns

1. By passing a dict to aggregate you can apply a different aggregation to the columns of a DataFrame:

```
grouped.agg({'C' : np.sum,
^^I        'D' : lambda x: np.std(x, ddof=1)})
```

Group Plotting

1. Groupby also works with some plotting methods.
2. For example, suppose we suspect that some features in a DataFrame may differ by group, in this case, the values in column 1 where the group is “B” are 3 higher on average.

```
import matplotlib.pyplot as plt
np.random.seed(1234)
df = pd.DataFrame(np.random.randn(50, 2))
df['g'] = np.random.choice(['A', 'B'], size=50)
df.loc[df['g'] == 'B', 1] += 3
df.groupby('g').boxplot()
```

使用 pivot() 方法

```
import pandas.util.testing as tm; tm.N = 3
def unpivot(frame):
    N, K = frame.shape
    data = {'value' : frame.values.ravel('F'),
^^I    'variable' : np.asarray(frame.columns).repeat(N),
^^I    'date' : np.tile(np.asarray(frame.index), K)}
    return pd.DataFrame(data, columns=['date', 'variable', '
     value'])

df = unpivot(tm.makeTimeDataFrame())

df[df['variable'] == 'A']
df.pivot(index='date', columns='variable', values='value')
df['value2'] = df['value']*2
df.pivot('date', 'variable')

pivoted = df.pivot('date', 'variable')
pivoted['value2']
```

Pivot tables

1. While pivot provides general purpose pivoting of DataFrames with various data types (strings, numerics, etc.),
2. the pivot_table function for pivoting with aggregation of numeric data.
3. The function pandas.pivot_table can be used to create spreadsheet-style pivot tables.

Pivot tables

1. pandas.pivot_table takes a number of arguments
   - data: A DataFrame object
   - values: a column or a list of columns to aggregate
   - index: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.
   - columns: a column, Grouper, array which has the same length as data, or list of them. Keys to group by on the pivot table column. If an array is passed, it is being used as the same manner as column values.
   - aggfunc: function to use for aggregation, defaulting to numpy.mean
2. pass margins=True to pivot_table, special All columns and rows will be added with partial group aggregates across the categories on the rows and columns

Pivot tables

```
df = pd.DataFrame({"A": ["foo", "foo", "foo", "foo", "foo",
^^I^^I^^I "bar", "bar", "bar", "bar"],
^^I^^I   "B": ["one", "one", "one", "two", "two",
^^I^^I^^I "one", "one", "two", "two"],
^^I^^I   "C": ["small", "large", "large", "small",
^^I^^I^^I "small", "large", "small", "small",
^^I^^I^^I "large"],
^^I^^I   "D": [1, 2, 2, 3, 3, 4, 5, 6, 7],
^^I^^I   "E": [2, 4, 5, 5, 6, 6, 8, 9, 9]})

table = pd.pivot_table(df, values='D', index=['A', 'B'],
^^I^^I    columns=['C'], aggfunc=np.sum)

table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
^^I^^I    aggfunc={'D': np.mean,
^^I^^I^^I     'E': np.mean})

table = pd.pivot_table(df, values=['D', 'E'], index=['A', 'C'],
^^I^^I    aggfunc={'D': np.mean,
^^I^^I^^I     'E': [min, max, np.mean]})
```

Cross tabulations

1. Use the crosstab function to compute a cross-tabulation of two (or more) factors.
2. By default crosstab computes a frequency table of the factors unless an array of values and an aggregation function are passed.
3. arguments:
    - index: array-like, values to group by in the rows
    - columns: array-like, values to group by in the columns
    - values: array-like, optional, array of values to aggregate according to the factors
    - aggfunc: function, optional, If no values array is passed, computes a frequency table
    - rownames: sequence, default None, must match number of row arrays passed
    - colnames: sequence, default None, if passed, must match number of column arrays passed
    - margins: boolean, default False, Add row/column margins (subtotals)
    - normalize: boolean, { 'all' , 'index' , 'columns' }, or {0,1}, default False. Normalize by dividing all values by the sum of values.

Cross tabulations

```
foo, bar, dull, shiny, one, two = 'foo', 'bar', 'dull', 'shiny'
    , 'one', 'two'
a = np.array([foo, foo, bar, bar, foo, foo], dtype=object)
b = np.array([one, one, two, one, two, one], dtype=object)
c = np.array([dull, dull, shiny, dull, dull, shiny], dtype=
    object)
pd.crosstab(a, [b, c], rownames=['a'], colnames=['b', 'c'])

df = pd.DataFrame({'A': [1, 2, 2, 2, 2], 'B': [3, 3, 4, 4, 4],
^^I^^I   'C': [1, 1, np.nan, 1, 1]})
pd.crosstab(df.A, df.B)
pd.crosstab(df['A'], df['B'], normalize=True)
pd.crosstab(df['A'], df['B'], normalize='columns')
pd.crosstab(df.A, df.B, values=df.C, aggfunc=np.sum, normalize=
    True,
margins=True)
```

`cut` function

1. The cut function computes groupings for the values of the input array and is often used to transform continuous variables to discrete or categorical variables.
2. If the bins keyword is an integer, then equal-width bins are formed. Alternatively we can specify custom bin-edges

cut function

```
ages = np.array([10, 15, 13, 12, 23, 25, 28, 59, 60])
pd.cut(ages, bins=3)
pd.cut(ages, bins=[0, 18, 35, 70])
```

dummy variables: get_dummies()

1. get_dummies() converts a categorical variable into a “dummy” or “indicator” DataFrame,
2. for example a column in a DataFrame (a Series) which has k distinct values, can derive a DataFrame containing k columns of 1s and 0s
3. get_dummies() also accepts a DataFrame. By default all categorical variables (categorical in the statistical sense, those with object or categorical dtype) are encoded as dummy variables.
4. control the columns that are encoded with the columns keyword.
5. drop_first keyword only keep k-1 levels of a categorical variable to avoid collinearity.

dummy variables: get_dummies()

```
df = pd.DataFrame({'key': list('bbacab'), 'data1': range(6)})
pd.get_dummies(df['key'])
dummies = pd.get_dummies(df['key'], prefix='key')

df = pd.DataFrame({'A': ['a', 'b', 'a'], 'B': ['c', 'c', 'b'],
^^I^^I    'C': [1, 2, 3]})

pd.get_dummies(df['key'])
pd.get_dummies(df, columns=['A'])
```

其他函数

1. stack unstack
2. melt
3. wide_to_long

创建分类

1. By specifying dtype=”category” when constructing a Series
2. pandas can include categorical data in a DataFrame by convert
3. Rename the categories to more meaningful names (assigning to Series.cat.categories
4. Reorder the categories and simultaneously add the missing categories
5. Sorting is per order in the categories, not lexical order.
6. Grouping by a categorical column shows also empty categories.

例子

```
s = pd.Series(["a","b","c","a"], dtype="category")
df = pd.DataFrame({"id":[1,2,3,4,5,6], "raw_grade":['a', 'b', '
    b', 'a', 'a', 'e']})
df["grade"] = df["raw_grade"].astype("category")
df["grade"].cat.categories = ["very good", "good", "very bad"]
df["grade"] = df["grade"].cat.set_categories(["very bad", "bad"
    , "medium", "good", "very good"])

df.sort_values(by="grade")
df.groupby("grade").size()
```

plot()

1. The plot method on Series and DataFrame is just a simple wrapper around plt.plot()
2. On DataFrame, plot() is a convenience to plot all of the columns with labels
3. plot one column versus another using the x and y keywords in plot()

```python
ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/
    2000',periods=1000))
ts = ts.cumsum()
ts.plot()

df3 = pd.DataFrame(np.random.randn(1000, 2), columns=['B', 'C'
    ]).cumsum()
df3['A'] = pd.Series(list(range(len(df))))

df3.plot(x='A', y='B')
```

Other Plots

1. Plotting methods allow for a handful of plot styles other than the default Line plot. These methods can be provided as the kind keyword argument to plot(). These include:
   - 'bar' or 'barh' for bar plots
   - 'hist' for histogram
   - 'box' for boxplot
   - 'kde' or 'density' for density plots
   - 'area' for area plots
   - 'scatter' for scatter plots
   - 'hexbin' for hexagonal bin plots
   - 'pie' for pie plots

其他画图函数

1. These functions can be imported from pandas.plotting and take a Series or DataFrame as an argument.
2. Scatter Matrix Plot：pandas.plotting.scatter_matrix
3. create density plots using the Series.plot.kde() and DataFrame.plot.kde() methods.
4. Andrews curves allow one to plot multivariate data as a large number of curves that are created using the attributes of samples as coefficients for Fourier series: andrews_curves

其他画图函数

5. Parallel coordinates is a plotting technique for plotting multivariate data. It allows one to see clusters in data and to estimate other statistics visually: parallel_coordinates
6. Lag plots are used to check if a data set or time series is random: lag_plot
7. Autocorrelation Plot: autocorrelation_plot
8. Bootstrap plots are used to visually assess the uncertainty of a statistic, such as mean, median, midrange, etc: bootstrap_plot
9. RadViz is a way of visualizing multi-variate data: radviz