经济大数据与 Python 应用

Introduction to Pandas

陈洲
湘潭大学商学院
2025

Press Space for next page →

Pandas

- Pandas is a newer package built on top of NumPy
- provides an efficient implementation of a `DataFrame`

Installing and Using Pandas

Install Pandas with Anaconda

```
conda install pandas
```

Once Pandas is installed, check the version:

```
import pandas
pandas.__version__
```

import Pandas under the alias `pd` :

```
import pandas as pd
```

Introducing Pandas Objects

Start our code sessions with the standard NumPy and Pandas imports:

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

The Pandas Series Object

A Pandas `Series`

- a one-dimensional array of indexed data.
- created from a list or array as follows:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

Series as Generalized NumPy Array

The essential difference

- the NumPy array has an *implicitly defined* integer index
- the Pandas `Series` has an *explicitly defined* index associated with the values.

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

And the item access works as expected:

```
data['b']
```

Series as Specialized Dictionary

In this way, Pandas `Series`

- a bit like a specialization of a Python dictionary.
- a `Series` is a structure that maps typed keys to a set of typed values.
- the type information of a Pandas `Series` makes it more efficient than Python dictionaries for certain operations.

Constructing a `Series` object directly from a Python dictionary

- here the five most populous US states according to the 2020 census:

```
population_dict = {'California': 39538223, 'Texas': 29145505,
                   'Florida': 21538187, 'New York': 20201249,
                   'Pennsylvania': 13002700}
population = pd.Series(population_dict)
population
```

Dictionary-style item access can be performed:

```
population['California']
```

Slicing:

```
population['California':'Florida']
```

Constructing Series Objects

From list

```
pd.Series([2, 4, 6])
```

From scalar

```
pd.Series(5, index=[100, 200, 300])
```

From dictionary

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

The Pandas DataFrame Object

The next fundamental structure in Pandas is the `DataFrame`

DataFrame as Generalized NumPy Array

A `DataFrame` as a sequence of aligned `Series` objects.

- "aligned" we mean that they share the same index.

Construct a new `Series` listing the area of each of the five states discussed in the previous section (in square kilometers):

```
area_dict = {'California': 423967, 'Texas': 695662, 'Florida': 170312,
             'New York': 141297, 'Pennsylvania': 119280}
area = pd.Series(area_dict)
area
```

Use a dictionary to construct a single two-dimensional object containing this information:

```
states = pd.DataFrame({'population': population,
                       'area': area})
states
```

Like the `Series` object, the `DataFrame` has an `index` attribute that gives access to the index labels:

```
states.index
```

The `DataFrame` has a `columns` attribute

```
states.columns
```

DataFrame as Specialized Dictionary

A `DataFrame` maps a column name to a `Series` of column data.

```
states['area']
```

Constructing DataFrame Objects

From a single Series object

```
pd.DataFrame(population, columns=['population'])
```

From a list of dicts

```
data = [{'a': i, 'b': 2 * i}
        for i in range(3)]
pd.DataFrame(data)
```

From a dictionary of Series objects

```
pd.DataFrame({'population': population,
              'area': area})
```

From a two-dimensional NumPy array

```
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
```

Data Indexing and Selection

Data Selection in Series

```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

Series as Dictionary

```
data['b']
```

```
'a' in data
```

```
data.keys()
```

```
list(data.items())
```

```
data['e'] = 1.25
data
```

Series as One-Dimensional Array

```
# slicing by explicit index
data['a':'c']
```

```
# slicing by implicit integer index
data[0:2]
```

```
# masking
data[(data > 0.3) & (data < 0.8)]
```

```
# fancy indexing
data[['a', 'e']]
```

Indexers: loc and iloc

- an indexing operation such as `data[1]` will use the explicit indices
- a slicing operation like `data[1:3]` will use the implicit Python-style indices:

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
# explicit index when indexing
data[1]
# implicit index when slicing
data[1:3]
```

First, the `loc` attribute allows indexing and slicing that always references the explicit index:

```
data.loc[1]
```

```
data.loc[1:3]
```

The `iloc` attribute allows indexing and slicing that always references the implicit Python-style index:

```
data.iloc[1]
```

```
data.iloc[1:3]
```

Data Selection in DataFrames

Example of areas and populations of states:

```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'Florida': 170312, 'New York': 141297,
                  'Pennsylvania': 119280})
pop = pd.Series({'California': 39538223, 'Texas': 29145505,
                 'Florida': 21538187, 'New York': 20201249,
                 'Pennsylvania': 13002700})
data = pd.DataFrame({'area':area, 'pop':pop})
data
```

Accessed via dictionary-style indexing of the **column** name:

```
data['area']
```

Use attribute-style access with column names that are strings:

```
data.area
```

Adding a new column:

```
data['density'] = data['pop'] / data['area']
data
```

DataFrame as Two-Dimensional Array

Raw underlying data array

```
data.values
```

Transpose the full `DataFrame` to swap rows and columns:

```
data.T
```

Passing a single index to an array accesses a row:

```
data.values[0]
```

Access a column:

```
data['area']
```

Using the `iloc` indexer

```
data.iloc[:3, :2]
```

Using the `loc` indexer

```
data.loc[:'Florida', :'pop']
```

Task

Get rows with `density` greater than 120 and columns with `pop` and `density`.

In the `loc` indexer we can combine masking and fancy indexing as follows:

```
data.loc[data.density > 120, ['pop', 'density']]
```

Additional Indexing Conventions

First, while *indexing* refers to columns, *slicing* refers to rows:

```
data['Florida':'New York']
```

```
data[1:3]
```

Masking operations are interpreted row-wise rather than column-wise:

```
data[data.density > 120]
```

Operating on Data in Pandas

NumPy ufunc will work on Pandas `Series` and `DataFrame` objects.

Defining a simple `Series` and `DataFrame`

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

```
rng = np.random.default_rng(42)
ser = pd.Series(rng.integers(0, 10, 4))
ser
```

```
df = pd.DataFrame(rng.integers(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
df
```

NumPy ufunc on either of these objects

- with the indices preserved

```
np.exp(ser)
```

```
np.sin(df * np.pi / 4)
```

Ufuncs: Index Alignment

Index Alignment in Series

```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967}, name='area')
population = pd.Series({'California': 39538223, 'Texas': 29145505,
                        'Florida': 21538187}, name='population')
```

Divide these to compute the population density:

```
population / area
```

Any missing values are marked by `NaN`

```
A = pd.Series([2, 4, 6], index=[0, 1, 2])
B = pd.Series([1, 3, 5], index=[1, 2, 3])
A + B
```

Optional explicit specification of the fill value for any elements in `A` or `B` that might be missing:

```
A.add(B, fill_value=0)
```

Index Alignment in DataFrames

```
A = pd.DataFrame(rng.integers(0, 20, (2, 2)),
                 columns=['a', 'b'])
A
```

```
B = pd.DataFrame(rng.integers(0, 10, (3, 3)),
                 columns=['b', 'a', 'c'])
B
```

```
A + B
```

Task

Fill with the mean of all values in A

```
A.add(B, fill_value=A.values.mean())
```

Ufuncs: Operations Between DataFrames and Series

Operations between a `DataFrame` and a `Series`

- the index and column alignment is similarly maintained

```
A = rng.integers(10, size=(3, 4))
A
```

```
df = pd.DataFrame(A, columns=['Q', 'R', 'S', 'T'])
df - df.iloc[0]
```

To operate column-wise

- use the object methods mentioned earlier, while specifying the `axis` keyword:

```
df.sub(df['R'], axis=0)
```

Handling Missing Data

NaN: Missing Numerical Data

Special floating-point value

```
vals2 = np.array([1, np.nan, 3, 4])
vals2
```

- supports fast operations pushed into compiled code
- the result of arithmetic with `NaN` will be another `NaN`

```
1 + np.nan
```

```
vals2.sum(), vals2.min(), vals2.max()
```

`NaN`-aware versions of aggregations

```
np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

Pandas Nullable Dtypes

A `Series` of integers with missing data, created from a list containing all three available markers of missing data

```
pd.Series([1, np.nan, 2, None, pd.NA], dtype='Int32')
```

Operating on Null Values

Pandas treats `None`, `NaN`, and `NA` as essentially interchangeable for indicating missing or null values.

Detecting Null Values

```
data = pd.Series([1, np.nan, 'hello', None])
```

```
data.isnull()
```

Boolean masks can be used directly as a `Series` or `DataFrame` index:

```
data[data.notnull()]
```

Dropping Null Values

```
data.dropna()
```

For a `DataFrame`, there are more options. Consider the following `DataFrame`:

```
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])
df
```

```
df.dropna()
```

```
df.dropna(axis='columns')
```

The `how` or `thresh` parameters

```
df[3] = np.nan
df
```

```
df.dropna(axis='columns', how='all')
```

```
df.dropna(axis='rows', thresh=3)
```

Filling Null Values

```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'), dtype='Int32')
data
```

```
data.fillna(0)
```

forward fill

```
data.fillna(method='ffill')
```

back fill

```
data.fillna(method='bfill')
```

Task

```
data = {
    'Country': ['CountryA', 'CountryA', 'CountryB', 'CountryB', 'CountryC', 'CountryC'],
    'Year': [2018, 2019, 2018, 2019, 2018, 2019],
    'GDP': [500e9, 520e9, 650e9, 670e9, 480e9, 490e9],
    'Population': [50e6, 50.5e6, 30e6, 30.5e6, 40e6, 40.5e6]
}
```

- Calculate the GDP per capita for each country for each year and add this as a new column `GDP_per_capita`
- Calculate the mean GDP per capita for the year 2019

Hierarchical Indexing

Also known as **multi-indexing**

A Multiply Indexed Series

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

The Bad Way

```
index = [('California', 2010), ('California', 2020),
         ('New York', 2010), ('New York', 2020),
         ('Texas', 2010), ('Texas', 2020)]
populations = [37253956, 39538223,
               19378102, 20201249,
               25145561, 29145505]
pop = pd.Series(populations, index=index)
pop
```

Task

Select all values from 2010

- messy (and potentially slow) munging to make it happen:

```
pop[[i for i in pop.index if i[1] == 2010]]
```

The Better Way: The Pandas MultiIndex

A multi-index from the tuples as follows:

```
index = pd.MultiIndex.from_tuples(index)
```

Reindex our series

```
pop = pop.reindex(index)
pop
```

Access all data for which the second index is 2020

```
pop[:, 2020]
```

MultiIndex as Extra Dimension

`unstack` method will quickly convert a multiply indexed `Series` into a conventionally indexed `DataFrame`:

```
pop_df = pop.unstack()
pop_df
```

the `stack` method provides the opposite operation:

```
pop_df.stack()
```

To add another column of demographic data for each state at each year

```
pop_df = pd.DataFrame({'total': pop,
                       'under18': [9284094, 8898092,
                                   4318033, 4181528,
                                   6879014, 7432474]})
pop_df
```

ufuncs

```
f_u18 = pop_df['under18'] / pop_df['total']
f_u18.unstack()
```

MultiIndex Level Names

```
pop.index.names = ['state', 'year']
pop
```

MultiIndex for Columns

Hierarchical indices and columns

```
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]],
                                   names=['year', 'visit'])
columns = pd.MultiIndex.from_product([['Bob', 'Guido', 'Sue'], ['HR', 'Temp']],
                                     names=['subject', 'type'])

# mock some data
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37

# create the DataFrame
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
```

Get a full `DataFrame` containing just that person's information:

```
health_data['Guido']
```

Indexing and Slicing a MultiIndex

Multiply Indexed Series

```
pop
```

Access single elements by indexing with multiple terms:

```
pop['California', 2010]
```

partial indexing

```
pop['California']
```

```
pop[:, 2010]
```

Partial slicing

```
pop.loc['California':'New York']
```

Multiply Indexed DataFrames

```
health_data
```

Recover Guido's heart rate data with a simple operation:

```
health_data['Guido', 'HR']
```

Use the `loc`, `iloc`

```
health_data.iloc[:2, :2]
```

```
health_data.loc[:, ('Bob', 'HR')]
```

Slice over multi-index

```
idx = pd.IndexSlice
health_data.loc[idx[:, 1], idx[:, 'HR']]
```

Rearranging Multi-Indexes

Sorted and Unsorted Indices

```
index = pd.MultiIndex.from_product([['a', 'c', 'b'], [1, 2]])
data = pd.Series(np.random.rand(6), index=index)
data.index.names = ['char', 'int']
data
```

```
try:
    data['a':'b']
except KeyError as e:
    print("KeyError", e)
```

```
data = data.sort_index()
data
```

```
data['a':'b']
```

Stacking and Unstacking Indices

```
pop.unstack(level=0)
```

```
pop.unstack(level=1)
```

```
pop.unstack().stack()
```

Index Setting and Resetting

```
pop_flat = pop.reset_index(name='population')
pop_flat
```

Build a `MultiIndex` from the column values

```
pop_flat.set_index(['state', 'year'])
```

Combining Datasets: concat and append

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

For convenience, define this function - creates a `DataFrame` of a particular form

```
def make_df(cols, ind):
    """Quickly make a DataFrame"""
    data = {c: [str(c) + str(i) for i in ind]
            for c in cols}
    return pd.DataFrame(data, ind)

# example DataFrame
make_df('ABC', range(3))
```

Helper class

```
class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

Simple Concatenation with pd.concat

Concatenation of `Series`

```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

Concatenate higher-dimensional objects, such as `DataFrame` s:

```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
display('df1', 'df2', 'pd.concat([df1, df2])')
```

`pd.concat` allows specification of an axis

```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
display('df3', 'df4', "pd.concat([df3, df4], axis='columns')")
```

Duplicate Indices

```
x = make_df('AB', [0, 1])
y = make_df('AB', [2, 3])
y.index = x.index  # make indices match
display('x', 'y', 'pd.concat([x, y])')
```

Notice the repeated indices in the result

Treating repeated indices as an error

```
try:
    pd.concat([x, y], verify_integrity=True)
except ValueError as e:
    print("ValueError:", e)
```

Ignoring the index

```
display('x', 'y', 'pd.concat([x, y], ignore_index=True)')
```

Adding MultiIndex keys

Use the `keys` option to specify a label for the **data sources**

```
display('x', 'y', "pd.concat([x, y], keys=['x', 'y'])")
```

Concatenation with Joins

```
df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
display('df5', 'df6', 'pd.concat([df5, df6])')
```

Change this to an intersection of the columns using `join='inner'`:

```
display('df5', 'df6',
        "pd.concat([df5, df6], join='inner')")
```

The append Method

```
display('df1', 'df2', 'df1.append(df2)')
```

Keep in mind

- the `append` method in Pandas does not modify the original object
- it creates a new object with the combined data.

It also is not a very efficient method

- it involves creation of a new index *and* data buffer.
- to do multiple `append` operations
  - better to build a list of `DataFrame` objects and pass them all at once to the `concat` function.

Combining Datasets: merge and join

For convenience, define the `display` function

```
import pandas as pd
import numpy as np

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

Relational Algebra
Categories of Joins

- *one-to-one*
- *many-to-one*
- *many-to-many*

One-to-One Joins

```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering',
                              'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
display('df1', 'df2')
```

Merge

```
df3 = pd.merge(df1, df2)
df3
```

Many-to-One Joins

```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
display('df3', 'df4', 'pd.merge(df3, df4)')
```

The information is repeated in one or more locations as required by the inputs

Many-to-Many Joins

```
df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                              'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'software', 'math',
                               'spreadsheets', 'organization']})
display('df1', 'df5', "pd.merge(df1, df5)")
```

Specification of the Merge Key

The on Keyword

```
display('df1', 'df2', "pd.merge(df1, df2, on='employee')")
```

The left_on and right_on Keywords

```
df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'salary': [70000, 80000, 120000, 90000]})
display('df1', 'df3', 'pd.merge(df1, df3, left_on="employee", right_on="name")')
```

The left_index and right_index Keywords

Merge on an index.

```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
display('df1a', 'df2a')
```

```
display('df1a', 'df2a',
        "pd.merge(df1a, df2a, left_index=True, right_index=True)")
```

An index-based merge without extra keywords:

```
df1a.join(df2a)
```

Specifying Set Arithmetic for Joins

```
df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                   columns=['name', 'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['wine', 'beer']},
                   columns=['name', 'drink'])
display('df6', 'df7', 'pd.merge(df6, df7)')
```

- the *intersection* of the two sets of inputs
- known as an *inner join*

```
pd.merge(df6, df7, how='inner')
```

An *outer join* returns a join over the union of the input columns, and fills in all missing values with NAs:

```
display('df6', 'df7', "pd.merge(df6, df7, how='outer')")
```

The *left join* and *right join* return joins over the left entries and right entries, respectively. For example:

```
display('df6', 'df7', "pd.merge(df6, df7, how='left')")
```

Overlapping Column Names: The suffixes Keyword

```
df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [1, 2, 3, 4]})
df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [3, 1, 4, 2]})
display('df8', 'df9', 'pd.merge(df8, df9, on="name")')
```

Specify a custom suffix using the `suffixes` keyword:

```
pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])
```

Example: US States Data (Optional)

Aggregation and Grouping

For convenience, we'll use the same `display` magic function that we used in the previous chapters:

```
import numpy as np
import pandas as pd

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

Planets Data

csv file

```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape
```

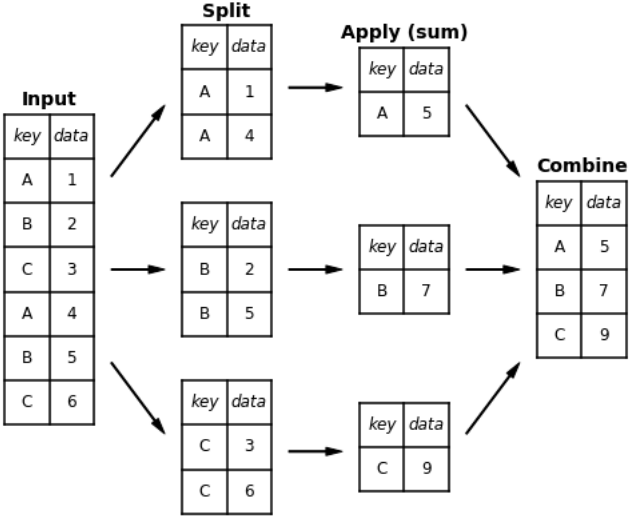
```
planets.head()
```

Computes several common aggregates for each column and returns the result

```
planets.dropna().describe()
```

groupby: Split, Apply, Combine

Split, Apply, Combine



The GroupBy Object

Column indexing

```
planets.groupby('method')
```

```
planets.groupby('method')['orbital_period']
```

```
planets.groupby('method')['orbital_period'].median()
```

Dispatch methods

For example, using the `describe` method

- equivalent to calling `describe` on the `DataFrame` representing each group:

```
planets.groupby('method')['year'].describe().unstack()
```

Aggregate, Filter

```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6),
                   'data2': rng.randint(0, 10, 6)},
                   columns = ['key', 'data1', 'data2'])
df
```

Aggregation

```
df.groupby('key').aggregate(['min', np.median, max])
```

Pass a dictionary mapping column names

```
df.groupby('key').aggregate({'data1': 'min',
                             'data2': 'max'})
```

Filtering

Drop data based on the group properties

```
def filter_func(x):
    return x['data2'].std() > 4

display('df', "df.groupby('key').std()",
        "df.groupby('key').filter(filter_func)")
```

Pivot Tables

Motivating Pivot Tables

Database of passengers on the *Titanic*, available through the Seaborn library

titanic.csv

```
import numpy as np
import pandas as pd
import seaborn as sns
titanic = sns.load_dataset('titanic')
```

```
titanic.head()
```

Pivot Tables by Hand

```
titanic.groupby('sex')[['survived']].mean()
```

Survival rates by both sex and, say, class

```
titanic.groupby(['sex', 'class'])['survived'].aggregate('mean').unstack()
```

Pivot Table Syntax

```
titanic.pivot_table('survived', index='sex', columns='class', aggfunc='mean')
```

Multilevel Pivot Tables

Bin the age using the `pd.cut` function:

```
age = pd.cut(titanic['age'], [0, 18, 80])
titanic.pivot_table('survived', ['sex', age], 'class')
```

Add info on the fare paid

```
fare = pd.qcut(titanic['fare'], 2)
titanic.pivot_table('survived', ['sex', age], [fare, 'class'])
```

Vectorized String Operations

Introducing Pandas String Operations

```
data = ['peter', 'Paul', 'MARY', 'gUIDO']
```

Capitalize

```
s if s is None else s.capitalize() for s in data
```

In Pandas

```
import pandas as pd
names = pd.Series(data)
names.str.capitalize()
```

Tables of Pandas String Methods (Optional)

```
monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                   'Eric Idle', 'Terry Jones', 'Michael Palin'])
```

```
monte.str.lower()
monte.str.len()
monte.str.startswith('T')
monte.str.split()
```

Methods Using Regular Expressions

Method	Description
`match`	Calls `re.match` on each element, returning a Boolean.
`extract`	Calls `re.match` on each element, returning matched groups as strings.
`findall`	Calls `re.findall` on each element
`replace`	Replaces occurrences of pattern with some other string
`contains`	Calls `re.search` on each element, returning a boolean
`count`	Counts occurrences of pattern
`split`	Equivalent to `str.split`, but accepts regexps
`rsplit`	Equivalent to `str.rsplit`, but accepts regexps

Vectorized item access and slicing

```
monte.str[0:3]
```

For example, to extract the last name of each entry

```
monte.str.split().str[-1]
```

Task

Match gdp data to population data

```
gdp = pd.DataFrame([{'省市': '湖南', 'gdp': 100},
                    {'省市': '上海', 'gdp': 200},
                    {'省市': '内蒙古', 'gdp': 300}])
```

```
population = pd.DataFrame([{'省市': '湖南省', 'population': 10},
                           {'省市': '上海市', 'population': 20},
                           {'省市': '内蒙古自治区', 'population': 30}])
```

Working with Time Series

Working with dates, times, and time-indexed data.

Dates and Times in Pandas: The Best of Both Worlds

```
import pandas as pd
date = pd.to_datetime("4th of July, 2021")
date
```

Vectorized operations

```
date + pd.to_timedelta(np.arange(12), 'D')
```

Pandas Time Series: Indexing by Time

```
index = pd.DatetimeIndex(['2020-07-04', '2020-08-04',
                          '2021-07-04', '2021-08-04'])
data = pd.Series([0, 1, 2, 3], index=index)
data
```

Passing values that can be coerced into dates:

```
data['2020-07-04':'2021-07-04']
```

Passing a year

```
data['2021']
```

Shifting, and Windowing

Using some stock price data as an example.

- `pandas-datareader` package (installable via `pip install pandas-datareader`) knows how to import data from various online sources.

Here we will load part of the S&P 500 price history:

```
from pandas_datareader import data

sp500 = data.get_data_fred('GS10') # use GS10 as example
sp500.head()
```

sp500.csv

Time Shifts

Resample the data to daily values

- shift by 364 to compute the 1-year return on investment for the S&P 500 over time

```
sp500 = sp500.asfreq('D', method='pad')

ROI = 100 * (sp500.shift(-365) - sp500) / sp500
# ROI.plot()
# plt.ylabel('% Return on Investment after 1 year');
```

Rolling Windows

The centered rolling mean and standard deviation of the stock prices

```
rolling = sp500.rolling(10, center=True)
# Rolling sum with the result assigned to the center of the window index.

rolling.mean()
```

High-Performance Pandas: eval and query

pandas.eval for Efficient Operations

```
import pandas as pd
nrows, ncols = 100000, 100
df1, df2, df3, df4 = (pd.DataFrame(rng.random((nrows, ncols)))
                      for i in range(4))
```

To compute the sum of all four `DataFrame`s using the typical Pandas approach

```
%timeit df1 + df2 + df3 + df4
```

The same result can be computed via `pd.eval` by constructing the expression as a string:

```
%timeit pd.eval('df1 + df2 + df3 + df4')
```

```
np.allclose(df1 + df2 + df3 + df4,
            pd.eval('df1 + df2 + df3 + df4'))
```

DataFrame.eval for Column-Wise Operations

```
df = pd.DataFrame(rng.random((1000, 3)), columns=['A', 'B', 'C'])
df.head()
```

The `DataFrame.eval` method allows much more succinct evaluation of expressions with the columns:

```
result3 = df.eval('(A + B) / (C - 1)')
np.allclose(result1, result3)
```

Assignment in DataFrame.eval

```
df.head()
```

```
df.eval('D = (A + B) / C', inplace=True)
df.head()
```

In the same way, any existing column can be modified:

```
df.eval('D = (A - B) / C', inplace=True)
df.head()
```

Local Variables in DataFrame.eval

```
column_mean = df.mean(1)
result1 = df['A'] + column_mean
result2 = df.eval('A + @column_mean')
np.allclose(result1, result2)
```

The DataFrame.query Method

```
result1 = df[(df.A < 0.5) & (df.B < 0.5)]
result2 = df.query('A < 0.5 and B < 0.5')
np.allclose(result1, result2)
```

END