## **Reading and Writing Data with Pandas**

#### pandas

**Methods to read data are all named** pd.read\_\* **where** \* **is the file type. Series and DataFrames can be saved to disk using their** to\_\* **method.**

Usage Patterns h5

###### Use pd.read\_clipboard() for one-oﬀ data extractions.

* Use the other pd.read\_\* methods in scripts for repeatable analyses.

read\_\* to\_\*

DataFrame

|  |  |  |  |
| --- | --- | --- | --- |
|  | X | Y | Z |
| a |  |  |  |
| b |  |  |  |
| c |  |  |  |

h5

### Reading Text Files into a DataFrame **+ +**

###### Colors highlight how diﬀerent arguments map from the data ﬁle to a DataFrame.

# Historical\_data.csv

**Rd**

**Cs**

**Date**

Date, Cs, Rd

2005-01-03, 64.78, -

2005-01-04, 63.79, 201.4

2005-01-05, 64.46, 193.45

...

Data from Lab Z.

Recorded by Agent E

>>> read\_table(

'historical\_data.csv', sep=',',

header=1, skiprows=1, skipfooter=2, index\_col=0, parse\_dates=True, na\_values=['-'])

Other arguments:

* names: set or override column names
* parse\_dates: accepts multiple argument types, see on the right
* converters: manually process each element in a column
* comment: character indicating commented line
* chunksize: read only a certain number of rows each time

### Parsing Tables from the Web

Writing Data Structures to Disk

>>> df\_list = read\_html(url)

Writing data structures to disk:

* s\_df.to\_csv(filename)
* s\_df.to\_excel(filename)

Write multiple DataFrames to single Excel ﬁle:

* writer = pd.ExcelWriter(filename)
* df1.to\_excel(writer, sheet\_name='First')
* df2.to\_excel(writer, sheet\_name='Second')
* writer.save()

Possible values of parse\_dates:

* + [0, 2]: Parse columns 0 and 2 as separate dates
  + [[0, 2]]: Group columns 0 and 2 and parse as single date
  + {'Date': [0, 2]}: Group columns 0 and 2, parse as single date in a column named Date.

Dates are parsed *after* the converters have been applied.

# , ,

|  |  |  |
| --- | --- | --- |
|  | **X** | **Y** |
| **a** |  |  |
| **b** |  |  |
| **c** |  |  |

|  |  |  |
| --- | --- | --- |
|  | **X** | **Y** |
| **a** |  |  |
| **b** |  |  |
| **c** |  |  |

|  |  |  |
| --- | --- | --- |
|  | **X** | **Y** |
| **a** |  |  |
| **b** |  |  |
| **c** |  |  |

### From and To a Database

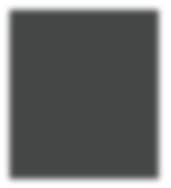
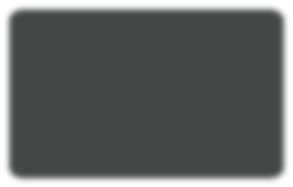
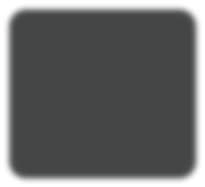
Read, using SQLAlchemy. Supports multiple databases:

* from sqlalchemy import create\_engine
* engine = create\_engine(database\_url)
* conn = engine.connect()
* df = pd.read\_sql(query\_str\_or\_table\_name, conn)

Write:

* df.to\_sql(table\_name, conn)

1. **Take your Pandas skills to the next level!** Register at [www.enthought.com/pandas-mastery-workshop](http://www.enthought.com/pandas-mastery-workshop)



© 2019 Enthought, Inc., licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>

**Pandas Data Structures: Series and DataFrames**

**A Series,** s**, maps an index to values. It is:**

* + **Like an ordered dictionary**

#### pandas

* + **A Numpy array with row labels and a name**

**A DataFrame,** df**, maps index and column labels to values. It is:**

* + **Like a dictionary of Series (columns) sharing the same index**
  + **A 2D Numpy array with row and column labels**

s\_df **applies to both Series and DataFrames.**

**Assume that manipulations of Pandas object return copies.**

Indexing and Slicing

Use these attributes on Series and DataFrames for indexing, slicing, and assignments:

### Creating Series and DataFrames

###### Series

* pd.Series(values, index=index,

**Series**

**Values**

s\_df.loc[] s\_df.iloc[]

s\_df.xs(key, level)

Refers only to the index labels Refers only to the integer location, similar to lists or Numpy arrays

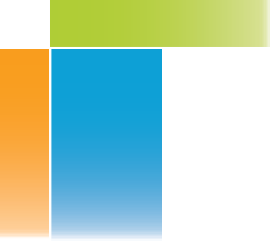
Select rows with label key in level

level of an object with MultiIndex.

name=name)

* pd.Series({'idx1': val1, 'idx2': val2} Where values, index, and name are sequences or arrays.

**DataFrame**

DataFrame

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Age** | **Gender** | **Columns** |
| **‘Cary’** | **32** | **M** |  |
| **‘Lynn’** | **18** | **F** |  |
| **‘Sam’** | **26** | **M** |  |
| **Index** | **Values** | |  |

**0**

**1**

**‘Sam’**

**‘Lynn’**

**n2**

**n1**

**‘Cary’**

**2**

**Integer location**

### Masking and Boolean Indexing

Create masks with, for example, comparisons

mask = df['X'] < 0

Or isin, for membership mask

mask = df['X'].isin(list\_valid\_values)

Use masks for indexing (must use loc) df.loc[mask] = 0

**Index**

**n3**

* pd.DataFrame(values, index=index,

columns=col\_names)

* pd.DataFrame({'col1': series1\_or\_seq,

'col2': series2\_or\_seq})

Where values is a sequence of sequences or a

2D array

### Manipulating Series and DataFrames

Combine multiple masks with bitwise operators (and (&), or (**|**), xor

(**^**), not (**~**)) and group them with parentheses:

mask = (df['X'] < 0) & (df['Y'] == 0)

### Common Indexing and Slicing Patterns

rows and cols can be values, lists, Series or masks.

Manipulating Columns df.rename(columns={old\_name: new\_name}) df.drop(name\_or\_names, axis='columns')

###### Manipulating Index

Renames column Drops column name

s\_df.loc[rows] df.loc[:, cols\_list] df.loc[rows, cols] s\_df.loc[mask] df.loc[mask, cols]

Some rows (all columns in a DataFrame) All rows, some columns

Subset of rows and columns Boolean mask of rows (all columns)

Boolean mask of rows, some columns

s\_df.reindex(new\_index) s\_df.drop(labels\_to\_drop) s\_df.rename(index={old\_label: new\_label}) s\_df.sort\_index()

Conform to new index Drops index labels Renames index labels Sorts index labels

### Using [ ] on Series and DataFrames

On Series, [ ] refers to the index labels, or to a slice

df.set\_index(column\_name\_or\_names)

s['a'] Value

s\_df.reset\_index()

Inserts index into columns, resets index to

default integer index.

s[:2] Series, ﬁrst 2 rows

On DataFrames, [ ] refers to columns labels:

###### Manipulating Values

All row values and the index will follow: df.sort\_values(col\_name, ascending=True) df.sort\_values(['X','Y'], ascending=[False, True])

### Important Attributes and Methods

df['X'] Series

df[['X', 'Y']] DataFrame

df['new\_or\_old\_col'] = series\_or\_array

EXCEPT! with a slice or mask.

s\_df.index df.columns s\_df.values

Array-like row labels Array-like column labels Numpy array, data

df[:2] df[mask]

DataFrame, ﬁrst 2 rows DataFrame, rows where mask is True

s\_df.shape s.dtype, df.dtypes

len(s\_df)

s\_df.head() and s\_df.tail()

s.unique() s\_df.describe()

df.info()

(n\_rows, m\_cols)

Type of Series, of each column Number of rows

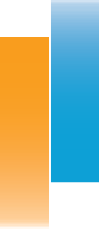
First/last rows

Series of unique values Summary stats Memory usage

NEVER CHAIN BRACKETS!

* df[mask]['X'] = 1 SettingWithCopyWarning
* df.loc[mask , 'X'] = 1

1. **Take your Pandas skills to the next level!** Register at [www.enthought.com/pandas-mastery-workshop](http://www.enthought.com/pandas-mastery-workshop)



© 2019 Enthought, Inc., licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>

## **Computation with Series and DataFrames**

**Pandas objects do not behave exactly like Numpy arrays. They follow three main rules (see on the right). Aligning objects on the index (or columns)**

#### pandas

**before calculations might be the most important difference. There are built-in methods for most common statistical operations, such as** mean **or** sum**, and they apply across one-dimension at a time. To apply custom functions, use one of three methods to do tablewise (**pipe**), row or column-wise (**apply**) or elementwise (**applymap**)**

**operations.**

**Rule 1:** Alignment First

* s1 + s2 > s1.add(s2, fill\_value=0)

**s1 s2 s1 s2**



**NaN**



**0**



**NaN**



|  |  |
| --- | --- |
| **a** | **1** |
| **b** | **2** |

|  |  |
| --- | --- |
| **b** | **4** |
| **c** | **5** |

|  |  |
| --- | --- |
| **a** | **NaN** |
| **b** | **6** |
| **c** | **NaN** |

|  |  |
| --- | --- |
| **a** | **1** |
| **b** | **2** |

|  |  |
| --- | --- |
| **b** | **4** |
| **c** | **5** |

|  |  |
| --- | --- |
| **a** | **1** |
| **b** | **6** |
| **c** | **5** |

Use add, sub, mul, div, to set ﬁll value.

**Rule 3:** Reduction Operations



**0**

The **3 Rules** of Binary Operations

##### Rule 1:

Operations between multiple Pandas objects implement

auto-alignment based on index ﬁrst.

##### Rule 2:

Mathematical operators (+ - \* / exp, log, ...) apply element by element, on the values.

##### Rule 3:

Reduction operations (mean, std, skew, kurt, sum, prod, ...) are applied column by column by default.

**Rule 2:** Element-By-Element Mathematical Operations

df + 1 df.abs() np.log(df)

|  |  |  |
| --- | --- | --- |
|  | **X** | **Y** |
| **a** | **-2** | **-2** |
| **b** | **-2** | **-2** |
| **c** | **-2** | **-2** |

|  |  |  |
| --- | --- | --- |
|  | **X** | **Y** |
| **a** | **-1** | **-1** |
| **b** | **-1** | **-1** |
| **c** | **-1** | **-1** |

|  |  |  |
| --- | --- | --- |
|  | **X** | **Y** |
| **a** | **1** | **1** |
| **b** | **1** | **1** |
| **c** | **1** | **1** |

|  |  |  |
| --- | --- | --- |
|  | **X** | **Y** |
| **a** | **0** | **0** |
| **b** | **0** | **0** |
| **c** | **0** | **0** |

###### 

>>> df.sum() Series

df.sum()

|  |  |  |
| --- | --- | --- |
|  | **X** | **Y** |
| **a** |  |  |
| **b** |  |  |
| **c** |  |  |

**X**

**Y**

### Apply a Function to Each Value

Operates across rows by default (axis=0, or axis='rows'). Operate across columns with axis=1 or axis='columns'.

Apply a function to each value in a Series or DataFrame s.apply(value\_to\_value) Series df.applymap(value\_to\_value) DataFrame

count sum: mean: mad: median:

min: max: mode: prod: std:

Number of non-null observations Sum of values

Mean of values

Mean absolute deviation Arithmetic median of values Minimum

Maximum Mode

Product of values

Bessel-corrected sample standard deviation

### Apply a Function to Each Series

Apply series\_to\_\* function to every column by default (across rows): df.apply(series\_to\_series) DataFrame df.apply(series\_to\_value) Series

To apply the function to every row (across columns), set axis=1: df.apply(series\_to\_series, axis=1)

### Apply a Function to a DataFrame

Apply a function that receives a DataFrame and returns a DataFrame, a Series,

var: sem: skew:

kurt:

Unbiased variance Standard error of the mean Sample skewness

(3rd moment) Sample kurtosis

or a single value:

df.pipe(df\_to\_df) DataFrame df.pipe(df\_to\_series) Series df.pipe(df\_to\_value) Value

quantile: value\_counts:

(4th moment) Sample quantile (Value at %) Count of unique values

### What Happens with Missing Values?

Missing values are represented by NaN (not a number) or NaT (not a time).

* + They propagate in operations across Pandas objects (1 + NaN NaN).
  + They are ignored in a "sensible" way in computations, they equal 0 in sum, they're ignored in mean, etc.
  + They stay NaN with mathematical operations (np.log(NaN) NaN).



**Take your Pandas skills to the next level!** Register at [www.enthought.com/pandas-mastery-workshop](http://www.enthought.com/pandas-mastery-workshop)

© 2019 Enthought, Inc., licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>

## **Plotting with Pandas Series and DataFrames**

###### **Pandas uses Matplotlib to generate figures. Once a figure is generated with Pandas, all of Matplotlib's functions**

Setup

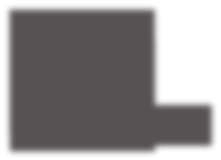
**can be used to modify the title, labels, legend, etc. In a**

pandas

**Jupyter notebook, all plotting calls for a given plot should be in the same cell.**

Parts of a Figure

***Figure***



An Axes object is what we think of as a “plot”. It has a title and two Axis objects that deﬁne data limits. Each Axis can have a label. There can be multiple Axes objects in a

***title***

***x label***

***y label***

Figure.

Import packages:

* import pandas as pd
* import matplotlib.pyplot as plt

Execute this at IPython prompt to display ﬁgures

in new windows:

* %matplotlib

Use this in Jupyter notebooks to display static images inline:

* %matplotlib inline

Use this in Jupyter notebooks to display zoom- able images inline:

* %matplotlib notebook

Plotting with Pandas Objects

Axes Axis

**Series Dataframe Labels**

Experiment A

Time

**X Y Z**

Value

|  |  |
| --- | --- |
| **a** |  |
| **b** |  |
| **c** |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **a** |  |  |  |
| **b** |  |  |  |
| **c** |  |  |  |

With a Series, Pandas plots values against the index:

* ax = s.plot()

With a DataFrame, Pandas creates one line per column:

* ax = df.plot()

Use Matplotlib to override or add annotations:

* ax.set\_xlabel('Time')
* ax.set\_ylabel('Value')

When plotting the results of complex manipulations with groupby, it's often useful to stack/unstack the resulting DataFrame to ﬁt the one-line-per-column assumption (see Data Structures cheatsheet).

Useful Arguments to plot

|  |  |  |
| --- | --- | --- |
|  | **X** | **Y** |
| **a** |  |  |
| **b** |  |  |
| **c** |  |  |

* subplots=True: one subplot per column, instead of one line
* figsize: set ﬁgure size, in inches
* x and y: plot one column against another

### Kinds of Plots

* ax.set\_title('Experiment A')

Pass labels if you want to override the column names and set the legend location:

* ax.legend(labels, loc='best')

Red Panda

*Ailurus fulgens*

**+**

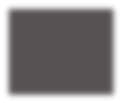
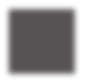
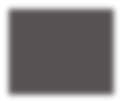
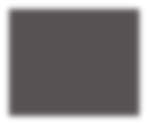
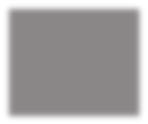
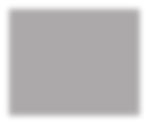
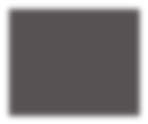
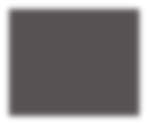
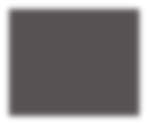
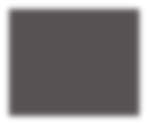
df.plot.scatter(x, y)

df.plot.bar()

df.plot.hist()

df.plot.box()

1. **Take your Pandas skills to the next level!** Register at [www.enthought.com/pandas-mastery-workshop](http://www.enthought.com/pandas-mastery-workshop)



© 2019 Enthought, Inc., licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>

## **Manipulating Dates and Times**

**Use a Datetime index for easy time-based indexing and slicing, as well as for powerful resampling and data alignment.**

**Pandas makes a distinction between timestamps, called**

Datetime **objects, and time spans, called** Period **objects.**

Timestamps vs Periods

Timestamps

pandas

**2016-01-01**

Converting Objects to Time Objects

**2016-01-02**

**2016-01-03 2016-01-04**

Convert diﬀerent types, for example strings, lists, or arrays to

Datetime with:

* pd.to\_datetime(value)

Convert timestamps to time spans: set period “duration” with

frequency oﬀset (see below).

* date\_obj.to\_period(freq=freq\_offset)

Creating Ranges of Timestamps

* + pd.date\_range(start=None, end=None,

periods=None, freq=offset, tz='Europe/London')

Specify either a start or end date, or both. Set number of

"steps" with periods. Set "step size" with freq; see "Frequen-

cy oﬀsets" for acceptable values. Specify time zones with tz.

Periods



... ...

**2016-01-01 2016-01-02 2016-01-03**

Save Yourself Some Pain: Use ISO 8601 Format

When entering dates, to be consistent and to lower the risk of error or confusion, use ISO format YYYY-MM-DD:

>>> pd.to\_datetime('12/01/2000') # 1st December Timestamp('2000-12-01 00:00:00')

>>> pd.to\_datetime('13/01/2000') # 13th January! Timestamp('2000-01-13 00:00:00')

>>> pd.to\_datetime('2000-01-13') # 13th January Timestamp('2000-01-13 00:00:00')

Frequency Offsets

Used by date\_range, period\_range and resample:

Creating Ranges or Periods

* + - B: Business day
    - D: Calendar day
    - W: Weekly
    - M: Month end
    - MS: Month start
    - BM: Business month end
    - Q: Quarter end

For more:

* A: Year end
* AS: Year start
* H: Hourly
* T, min: Minutely
* S: Secondly
* L, ms: Milliseconds
* U, us: Microseconds
* N: Nanoseconds
  + pd.period\_range(start=None, end=None,

periods=None, freq=offset)

Resampling

* + s\_df.resample(freq\_offset).mean() resample returns a groupby-like object that must be

Lookup "Pandas Oﬀset Aliases" or check out pandas.tseries.offsets,

and pandas.tseries.holiday modules.

## **Vectorized String Operations**

**Pandas implements vectorized string operations named after Python's string methods. Access them through the** str **attribute of string Series**

Some String Methods

aggregated with mean, sum, std, apply, etc. (See also the Split-Apply-Combine cheat sheet.)

Splitting and Replacing

split returns a Series of lists:

* + s.str.split()

Access an element of each list with get:

* s.str.lower()
* s.str.isupper()
* s.str.len()
* s.str.strip()
* s.str.normalize()

and more…

* + s.str.split(char).str.get(1)

Return a DataFrame instead of a list:

* + s.str.split(expand=True)

Index by character position:

* + s.str[0]

True if regular expression pattern or string in Series:

* + s.str.contains(str\_or\_pattern)

Find and replace with string or regular expressions:

* + s.str.replace(str\_or\_regex, new)
  + s.str.extract(regex)
  + s.str.findall(regex)

1. **Take your Pandas skills to the next level!** Register at [www.enthought.com/pandas-mastery-workshop](http://www.enthought.com/pandas-mastery-workshop)



© 2019 Enthought, Inc., licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>

## **Combining DataFrames**

###### **Tools for combining Series and DataFrames together, with SQL-type joins and concatenation. Use join if merging**

**on indices, otherwise use merge.**

Concatenating DataFrames

* pd.concat(df\_list)

“Stacks” DataFrames on top of each other.

pandas

Merge on Column Values

* + pd.merge(left, right, how='inner', on='id')

Ignores index, unless on=None. See value of how below.

Use on if merging on same column in both DataFrames, otherwise use left\_on**,** right\_on**.**

Merge Types: The how Keyword

Set ignore\_index=True, to replace index with RangeIndex**.**

Note: Faster than repeated df.append(other\_df)***.***

Join on Index

* df.join(other)

**Merge** DataFrames on indexes. Set on=columns to join on index of other and on columns of df. join uses pd.merge under the covers.

left right

left right

left right

###### how="outer"

how="inner"

left

right

|  |  |  |
| --- | --- | --- |
|  | **long** | **X** |
| **0** | **aaaa** | **a** |
| **1** | **bbbb** | **b** |

|  |  |  |
| --- | --- | --- |
|  | **long** | **X** |
| **0** | **aaaa** | **a** |
| **1** | **bbbb** | **b** |

how="left"

|  |  |  |
| --- | --- | --- |
|  | **long** | **X** |
| **0** | **aaaa** | **a** |
| **1** | **bbbb** | **b** |

how="right"

|  |  |  |
| --- | --- | --- |
|  | **long** | **X** |
| **0** | **aaaa** | **a** |
| **1** | **bbbb** | **b** |

left

left\_on='X' right\_on='Y'

###### 

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **long** | **X** | **Y** | **short** |
| **0** | **aaaa** | **a** |  |  |
| **1** | **bbbb** | **b** | **b** | **bb** |
| **2** |  |  | **c** | **cc** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **long** | **X** | **Y** | **short** |
| **0** | **bbbb** | **b** | **b** | **bb** |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **long** | **X** | **Y** | **short** |
| **0** | **aaaa** | **a** |  |  |
| **1** | **bbbb** | **b** | **b** | **bb** |

right

###### 

|  |  |  |
| --- | --- | --- |
|  | **Y** | **short** |
| **0** | **b** | **bb** |
| **1** | **c** | **cc** |

|  |  |  |
| --- | --- | --- |
|  | **Y** | **short** |
| **0** | **b** | **bb** |
| **1** | **c** | **cc** |

|  |  |  |
| --- | --- | --- |
|  | **Y** | **short** |
| **0** | **b** | **bb** |
| **1** | **c** | **cc** |

## **Cleaning Data with Missing Values**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **long** | **X** | **Y** | **short** |
| **0** | **bbbb** | **b** | **b** | **bb** |
| **1** |  |  | **c** | **cc** |

|  |  |  |
| --- | --- | --- |
|  | **Y** | **short** |
| **0** | **b** | **bb** |
| **1** | **c** | **ctc** |

###### **Pandas represents missing values as** NaN **(Not a Number). It comes from Numpy and is of type** float64**. Pandas has many methods to find and replace missing values.**

Replacing Missing Values

Find Missing Values

* s\_df.isnull() **or** > pd.isnull(obj)
* s\_df.notnull() **or** > pd.notnull(obj)

s\_df.loc[s\_df.isnull()] = 0 s\_df.interpolate(method='linear') s\_df.fillna(method='ffill') s\_df.fillna(method='bfill') s\_df.dropna(how='any') s\_df.dropna(how='all')

s\_df.dropna(how='all', axis=1)

Use mask to replace NaN Interpolate using diﬀerent methods Fill forward (last valid value)

Or backward (next valid value) Drop rows if any value is NaN Drop rows if all values are NaN

Drop across columns instead of rows

1. **Take your Pandas skills to the next level!** Register at [www.enthought.com/pandas-mastery-workshop](http://www.enthought.com/pandas-mastery-workshop)



© 2019 Enthought, Inc., licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>

## **Split / Apply / Combine with DataFrames**

1. ***Split* the data based on some criteria.**

|  |  |
| --- | --- |
| **X** | **Y** |
| **a** | **1** |
| **a** | **2** |

pandas

|  |  |
| --- | --- |
| **X** | **Y** |
| **a** | **1** |
| **b** | **3** |
| **c** | **2** |
| **b** | **1** |
| **c** | **2** |
| **a** | **2** |

1. ***Appl y* a function to each group to aggregate, transform, or filter.**
2. ***Combine* the results.**

**The apply and combine steps are typically done together in Pandas.**

**Split:** Group By

Group by a single column:

* g = df.groupby(col\_name)

Grouping with list of column names creates DataFrame with MultiIndex. (see “Reshaping DataFrames and Pivot Tables” cheatsheet):

* g = df.groupby(list\_col\_names)

Pass a function to group based on the index:

* g = df.groupby(function)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **0** | **a** |  |  |
| **2** | **a** |  |  |

df.groupby('X')

**Split:** What’s a GroupBy Object?

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **0** | **a** |  |  |
| **1** | **b** |  |  |
| **2** | **a** |  |  |
| **3** | **b** |  |  |
| **4** | **c** |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **1** | **b** |  |  |
| **3** | **b** |  |  |

**Split/Apply/Combine**

**1.5**

|  |  |
| --- | --- |
| **X** | **Y** |
| **b** | **3** |
| **b** | **1** |

|  |  |
| --- | --- |
| **X** | **Y** |
| **c** | **2** |
| **c** | **2** |

**2**

|  |  |
| --- | --- |
| **X** | **Y** |
| **a** | **1.5** |
| **b** | **2** |
| **c** | **2** |

**2**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **4** | **c** |  |  |

### **Apply/Combine:** General Tool: apply

More general than agg, transform, and filter. Can aggregate, transform or ﬁlter. The resulting dimensions can change, for example:

* g.apply(lambda x: x.describe())

### **Apply/Combine:** Transformation

The shape and the index do not change.

* g.transform(df\_to\_df)

Example, normalization:

* def normalize(grp):

. return (grp - grp.mean()) / grp.var()

* g.transform(normalize)

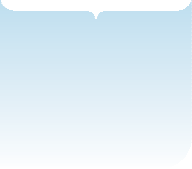
|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **0** | **a** | **0** | **0** |
| **1** | **b** | **0** | **0** |
| **2** | **a** | **0** | **0** |
| **3** | **b** | **0** | **0** |
| **4** | **c** | **0** | **0** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **0** | **a** | **1** | **1** |
| **2** | **a** | **1** | **1** |

g.transform(…)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **1** | **b** | **2** | **2** |
| **3** | **b** | **2** | **2** |

It keeps track of which rows are part of which group.



**Split**

* Groupby
* Window Functions

**Apply**

**Combine**

* Apply
* Group-specific transformations
* Aggregation
* Group-specific Filtering
  + g.groups Dictionary, where keys are group names, and values are indices of rows in a given group. It is iterable:
  + for group, sub\_df in g:

...

### **Apply/Combine:** Aggregation

Perform computations on each group. The shape changes; the categories in the grouping columns become the index. Can use built-in aggregation methods: mean, sum, size, count, std, var, sem, describe, first, last, nth, min, max, for example:

* + g.mean()

… or aggregate using custom function:

* + g.agg(series\_to\_value)

… or aggregate with multiple functions at once:

* + g.agg([s\_to\_v1, s\_to\_v2])

… or use diﬀerent functions on diﬀerent columns.

* + g.agg({'Y': s\_to\_v1, 'Z': s\_to\_v2})

###### 

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **4** | **c** | **3** | **3** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **0** | **a** |  |  |
| **2** | **a** |  |  |

**Apply/Combine:** Filtering

Returns a group only if condition is true.

* + g.filter(lambda x: len(x)>1)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **0** | **a** | **1** | **1** |
| **2** | **a** | **1** | **1** |

g.filter(…)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **0** | **a** | **1** | **1** |
| **1** | **b** | **1** | **1** |
| **2** | **a** | **1** | **1** |
| **3** | **b** | **1** | **1** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **1** | **b** | **1** | **1** |
| **3** | **b** | **1** | **1** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **4** | **c** | **0** | **0** |

g.agg(…)

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **4** | **c** |  |  |

### Other Groupby-Like Operations: Window Functions

|  |  |  |  |
| --- | --- | --- | --- |
|  | **X** | **Y** | **Z** |
| **1** | **b** |  |  |
| **3** | **b** |  |  |

|  |  |  |
| --- | --- | --- |
|  | **Y** | **Z** |
| **a** |  |  |
| **b** |  |  |
| **c** |  |  |

* + - resample, rolling, and ewm (exponential weighted function) methods behave like GroupBy objects. They keep track of which row is in which “group”. Results must be aggregated with sum, mean, count, etc. (see Aggregation).

|  |  |
| --- | --- |
| **0** |  |
| **1** |  |
| **2** |  |
| **3** |  |
| **4** |  |

* + - resample is often used before rolling, expanding, and

ewm when using a DateTime index.

1. **Take your Pandas skills to the next level!** Register at [www.enthought.com/pandas-mastery-workshop](http://www.enthought.com/pandas-mastery-workshop)



© 2019 Enthought, Inc., licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>

## **Reshaping DataFrames and Pivot Tables**

**Tools for reshaping** DataFrames **from the wide to the long format and back. The long format can be tidy, which means that "each variable is a column,**

#### pandas

**each observation is a row"1. Tidy data is easier to filter, aggregate, transform, sort, and pivot. Reshaping operations often produce multi-level indices or columns, which can be sliced and indexed.**

1 Hadley Wickham (2014) "Tidy Data", <http://dx.doi.org/10.18637/jss.v059.i10>

Long to Wide Format and Back with stack() and unstack()

Pivot **column** level **to index**,

MultiIndex: A Multi-Level Hierarchical Index

i.e. "stacking the columns" (wide to long):

* df.stack()

Pivot **index** level **to columns**, "unstack the columns" (long to wide):

* df.unstack()

Often created as a result of:

* df.groupby(list\_of\_columns)
* df.set\_index(list\_of\_columns)

Contiguous labels are *displayed* together but apply to each row. The concept is similar to multi-level columns.

A MultiIndex allows indexing and slicing one or multiple levels at once. Using the *Long* example from the right:

If multiple indices or column levels, use level number or name to

stack/unstack:

* df.unstack(1) or > df.unstack('Month')

A common use case for unstacking, plotting group data vs index after groupby:

* (df.groupby(['A', 'B])['relevant'].mean()

.unstack().plot())

**Long**

**Wide Stack**

|  |  |  |
| --- | --- | --- |
| **Year** | **Month** | **Value** |
| **1900** | **Jan.** | **1** |
| **Feb** | **7** |
| **Mar.** | **2** |
| **2000** | **Jan.** | **4** |
| **Feb** | **3** |
| **Mar.** | **9** |

|  |  |  |  |
| --- | --- | --- | --- |
| **Year** | **Jan.** | **Feb.** | **Mar.** |
| **1900** | **1** | **7** | **2** |
| **2000** | **4** | **3** | **9** |

long.loc[1900] long.loc[(1900, 'March')] long.xs('March', level='Month')

Simpler than using boolean indexing, for example:

* long[long.Month == 'March']

All 1900 rows

value **2**

All March rows

**Unstack**

### Pivot Tables

* pd.pivot\_table(df,

index=cols, (keys to group by for index) columns=cols2, (keys to group by for columns) values=cols3, (columns to aggregate) aggfunc='mean') (what to do with repeated values)

Omitting index, columns, or values will use all remaining columns of df. You can "pivot" a table manually using groupby, stack and unstack.

**Index Columns**

### From Wide to Long with melt

Specify which columns are identiﬁers (id\_vars, values will be repeated for each row) and which are "measured variables" (value\_vars, will become values in *variable* column.

All remaining columns by default).

pd.melt(df, id\_vars=id\_cols, value\_vars=value\_columns) pd.melt(team, id\_vars=['Color'],

value\_vars=['A', 'B', 'C'],

var\_name='Team', value\_name='Score')

**Continent**

|  |  |  |  |
| --- | --- | --- | --- |
| **0** | **Recently updated** | **Number of stations** | **Continent code** |
| **1** | **FALSE** | **1** | **EU** |
| **2** | **FALSE** | **1** | **EU** |
| **3** | **FALSE** | **1** | **EU** |
| **4** | **TRUE** | **1** | **EU** |
| **5** | **FALSE** | **1** | **AN** |
| **6** | **TRUE** | **1** | **AN** |
| **7** | **TRUE** | **1** | **AN** |

**code AN EU**

**Recently updated**

**FALSE 1 3**

**TRUE 2 1**

pd.pivot\_table(df,

index="Recently updated", columns="continent code", values="Number of Stations", aggfunc=np.sum)

### df.pivot() vs pd.pivot\_table

**Team**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Color** | **A** | **B** | **C** |
| **0** | **Red** | **1** | **3** | **4** |
| **1** | **Blue** | **2** | **-** | **6** |

**Melt**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Color** | **Team** | **Score** |
| **0** | **Red** | **A** | **1** |
| **1** | **Blue** | **A** | **2** |
| **2** | **Red** | **B** | **3** |
| **3** | **Blue** | **B** | **-** |
| **4** | **Red** | **C** | **4** |
| **5** | **Blue** | **C** | **6** |

df.pivot()

pd.pivot\_table()

Does not deal with repeated values in index. It's a declarative form of stack and unstack.

Use if you have repeated values in index (specify aggfunc argument).

Red Panda

*Ailurus fulgens*

1. **Take your Pandas skills to the next level!** Register at [www.enthought.com/pandas-mastery-workshop](http://www.enthought.com/pandas-mastery-workshop)



© 2019 Enthought, Inc., licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-nd/4.0/>