Plotting with Pandas Series and DataFrames

pandas

Pandas uses Matplotlib to generate figures. Once a figure is generated with Pandas, all of Matplotlib's functions can be used to modify the title, labels, legend, etc. In a Jupyter notebook, all plotting calls for a given plot should be in the same cell.

Setup

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:

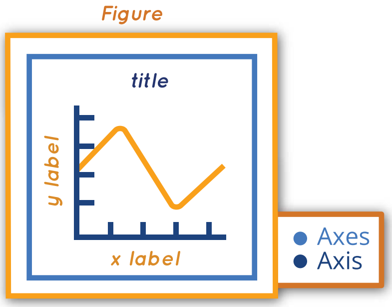
```
> %matplotlib inline
```

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

```
> %matplotlib notebook
```

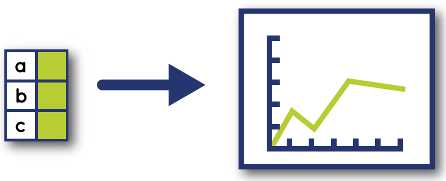
Parts of a Figure

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 Figure.



Plotting with Pandas Objects

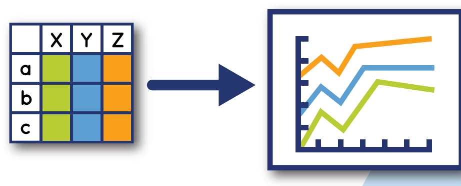
Series



With a Series, Pandas plots values against the index:

```
> ax = s.plot()
```

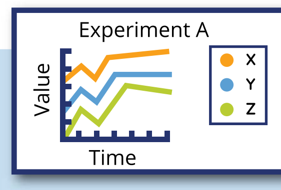
Dataframe



With a DataFrame, Pandas creates one line per column:

```
> ax = df.plot()
```

Labels



Use Matplotlib to override or add annotations:

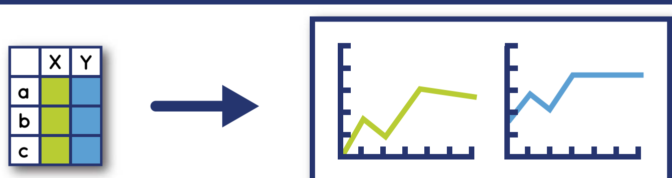
```
> ax.set_xlabel('Time')
> ax.set_ylabel('Value')
> ax.set_title('Experiment A')
```

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

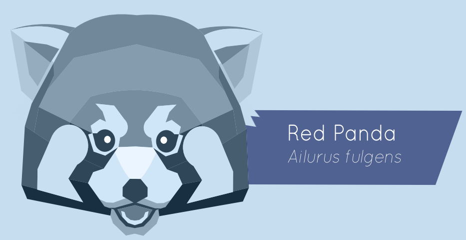
```
> ax.legend(labels, loc='best')
```

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).

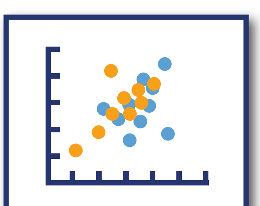
Useful Arguments to **plot**



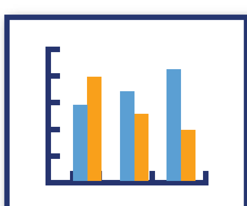
- **subplots=True**: one subplot per column, instead of one line
- **figsize**: set figure size, in inches
- **x** and **y**: plot one column against another



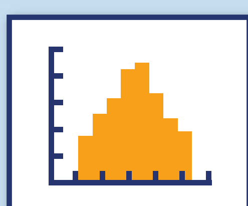
Kinds of Plots



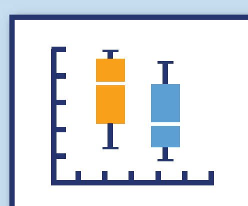
`df.plot(kind='scatter')`



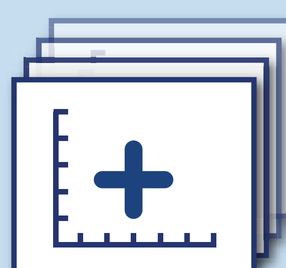
`df.plot(kind='bar')`



`df.plot(kind='hist')`



`df.boxplot()`

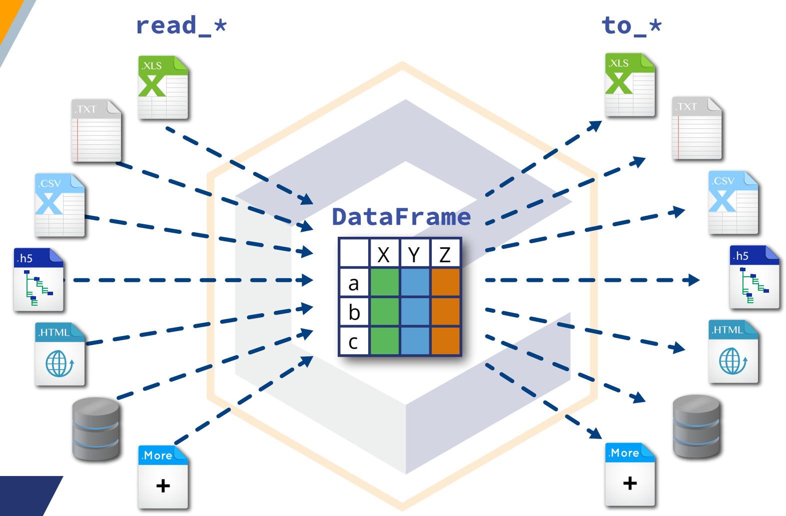


ENTHOUGHT

Take your Pandas skills to the next level! Register at www.enthought.com/pandas-master-class

Reading and Writing Data with 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

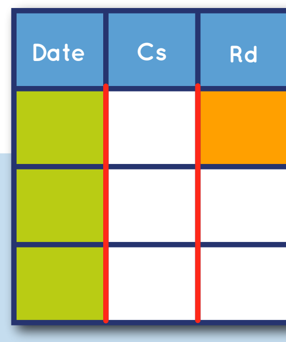
- Use `pd.read_clipboard()` for one-off data extractions.
- Use the other `pd.read_*` methods in scripts for repeatable analyses.

Reading Text Files into a DataFrame

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

```
# Historical_data.csv
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:

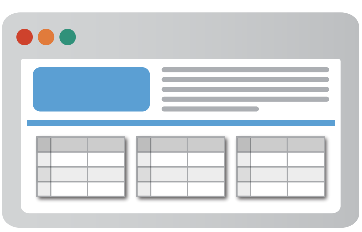
- **`names`**: set or override column names
- **`parse_dates`**: accepts multiple argument types, see on the right
- **`converters`**: manually process each element in a column
- **`comment`**: character indicating commented line
- **`chunksize`**: read only a certain number of rows each time

Possible values of **`parse_dates`**:

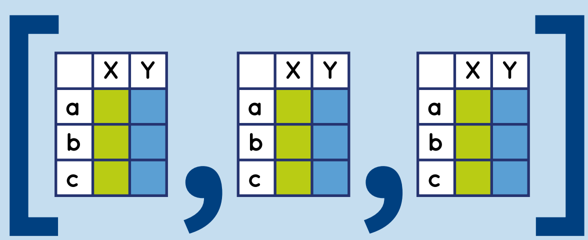
- **`[0, 2]`**: Parse columns 0 and 2 as separate dates
- **`[[0, 2]]`**: Group columns 0 and 2 and parse as single date
- **`{'Date': [0, 2]}`**: Group columns 0 and 2, parse as single date in a column named Date.

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

Parsing Tables from the Web



```
>>> df_list = read_html(url)
```



Writing Data Structures to Disk

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()
```

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)
```

Take your Pandas skills to the next level! Register at www.enthought.com/pandas-master-class

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) 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.

The **3 Rules** of Binary Operations

Rule 1:

Operations between multiple Pandas objects implement auto-alignment based on index first.

Rule 2:

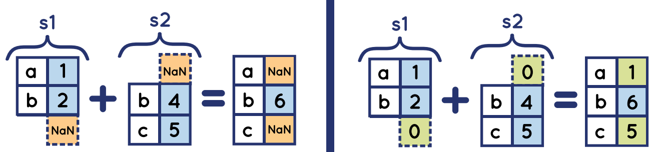
Mathematical operators (+ - * / exp, log, ...) apply element by element, on the values.

Rule 3:

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

Rule 1: Alignment First

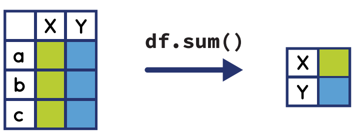
`> s1 + s2` | `> s1.add(s2, fill_value=0)`



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

Rule 3: Reduction Operations

`>>> df.sum()` → Series

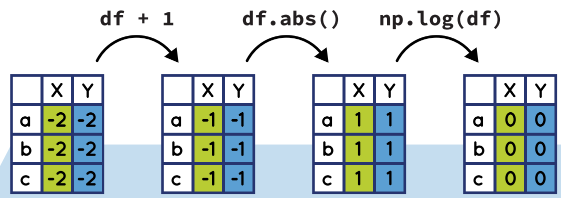


Operates across rows by default (**axis=0**, or **axis='rows'**).
Operate across columns with **axis=1** or **axis='columns'**.

Reduction functions

count:	Number of non-null observations
sum:	Sum of values
mean:	Mean of values
mad:	Mean absolute deviation
median:	Arithmetic median of values
min:	Minimum
max:	Maximum
mode:	Mode
prod:	Product of values
std:	Bessel-corrected sample standard deviation
var:	Unbiased variance
sem:	Standard error of the mean
skew:	Sample skewness (3rd moment)
kurt:	Sample kurtosis (4th moment)
quantile:	Sample quantile (Value at %)
value_counts:	Count of unique values

Rule 2: Element-By-Element Mathematical Operations



Apply a Function to Each Value

Apply a function to each value in a Series or DataFrame

s.apply(value_to_value) → Series
df.applymap(value_to_value) → DataFrame

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, or a single value:

df.pipe(df_to_df) → DataFrame
df.pipe(df_to_series) → Series
df.pipe(df_to_value) → Value

What Happens with Missing Values?

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

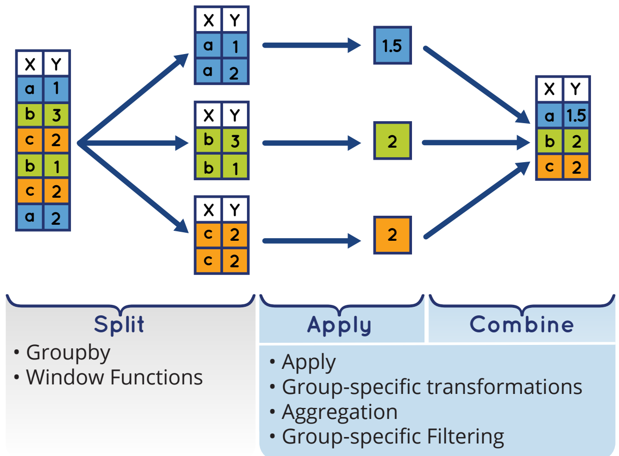
- They propagate in operations across Pandas objects (**1 + NaN** → **NaN**).
- They are ignored in a "sensible" way in computations, they equal 0 in **sum**, they're ignored in **mean**, etc.
- They stay **NaN** with mathematical operations (**np.log(NaN)** → **NaN**).

Split / Apply / Combine with DataFrames

1. Split the data based on some criteria.
2. Apply a function to each group to aggregate, transform, or filter.
3. Combine the results.

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

Split/Apply/Combine



Split: Group By

Group by a single column:

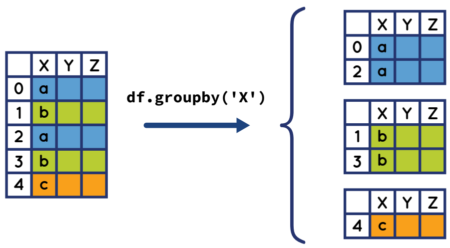
```
> g = df.groupby(col_name)
```

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)
```



Split: What's a GroupBy Object?

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

`> 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: 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: 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:

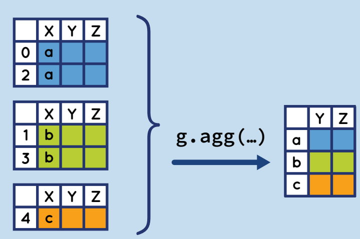
```
> g.agg(series_to_value)
```

... 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})
```



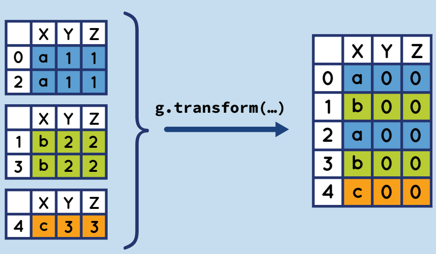
Apply/Combine: Transformation

The shape and the index do not change.

```
> g.transform(df_to_df)
```

Example, normalization:

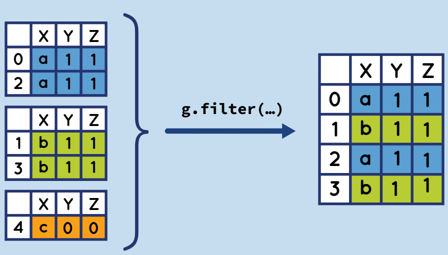
```
> def normalize(grp):
.       return (grp - grp.mean()) / grp.var()
> g.transform(normalize)
```



Apply/Combine: Filtering

Returns a group only if condition is true.

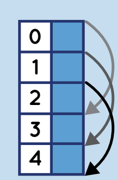
```
> g.filter(lambda x: len(x)>1)
```



Other Groupby-Like Operations: Window Functions

• **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).

• **resample** is often used before **rolling**, **expanding**, and **ewm** when using a DateTime index.



Take your Pandas skills to the next level! Register at www.enthought.com/pandas-master-class

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.

Converting Objects to Time Objects

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 "Frequency offsets" for acceptable values. Specify time zones with **`tz`**.

Frequency Offsets

Used by **`date_range`**, **`period_range`** and **`resample`**:

- B: Business day
- D: Calendar day
- W: Weekly
- M: Month end
- MS: Month start
- BM: Business month end
- Q: Quarter end
- A: Year end
- AS: Year start
- H: Hourly
- T, min: Minutely
- S: Secondly
- L, ms: Milliseconds
- U, us: Microseconds
- N: Nanoseconds

For more:
Lookup "Pandas Offset Aliases" or check out **`pandas.tseries.offsets`**, and **`pandas.tseries.holiday`** modules.

Timestamps vs Periods

Timestamps

Periods

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')
```

Creating Ranges or Periods

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

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

```
> s.str.lower()        > s.str.strip()
> s.str.isupper()      > s.str.normalize()
> s.str.len()
```

and more...

Index by character position:

```
> s.str[0]
```

`True` if regular expression pattern or string in Series:

```
> s.str.contains(str_or_pattern)
```

Splitting and Replacing

`split` returns a Series of lists:

```
> s.str.split()
```

Access an element of each list with **`get`**:

```
> s.str.split(char).str.get(1)
```

Return a DataFrame instead of a list:

```
> s.str.split(expand=True)
```

Find and replace with string or regular expressions:

```
> s.str.replace(str_or_regex, new)
> s.str.extract(regex)
> s.str.findall(regex)
```

ENTHOUGHT

Take your Pandas skills to the next level! Register at www.enthought.com/pandas-master-class

Pandas Data Structures: Series and DataFrames

pandas

A Series, **s**, maps an index to values. It is:
- Like an ordered dictionary
- 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

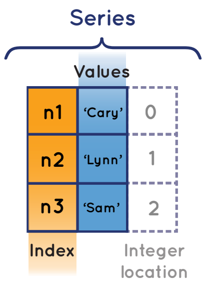
s_df applies to both Series and DataFrames.
Assume that manipulations of Pandas object return copies.

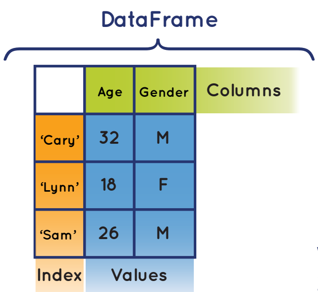
Creating Series and DataFrames

Series

```
> pd.Series(values, index=index,
            name=name)
> pd.Series({'idx1': val1, 'idx2': val2}
```

Where **values**, **index**, and **name** are sequences or arrays.





DataFrame

```
> 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

Manipulating **Columns**

`df.rename(columns={old_name: new_name})`	Renames column
`df.drop(name_or_names, axis='columns')`	Drops column name

Manipulating **Index**

`s_df.reindex(new_index)`	Conform to new index
`s_df.drop(labels_to_drop)`	Drops index labels
`s_df.rename(index={old_label: new_label})`	Renames index labels
`s_df.reset_index()`	Drops index, replaces with **Range** index
`s_df.sort_index()`	Sorts index labels
`df.set_index(column_name_or_names)`	

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

`s_df.index`	Array-like row labels
`df.columns`	Array-like column labels
`s_df.values`	Numpy array, data
`s_df.shape`	(n_rows, m_cols)
`s.dtype, df.dtypes`	Type of **Series**, of each column
`len(s_df)`	Number of rows
`s_df.head()` and `s_df.tail()`	First/last rows
`s.unique()`	Series of unique values
`s_df.describe()`	Summary stats
`df.info()`	Memory usage

Indexing and Slicing

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

`s_df.loc[]`	Refers only to the index labels
`s_df.iloc[]`	Refers only to the integer location, similar to lists or Numpy arrays
`s_df.xs(key, level)`	Select rows with label **key** in level **level** of an object with MultiIndex.

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
```

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.

`s_df.loc[rows]`	Some rows (all columns in a DataFrame)
`df.loc[:, cols_list]`	All rows, some columns
`df.loc[rows, cols]`	Subset of rows and columns
`s_df.loc[mask]`	Boolean mask of rows (all columns)
`df.loc[mask, cols]`	Boolean mask of rows, some columns

Using [ ] on Series and DataFrames

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

`s['a']`	Value
`s[:2]`	Series, first 2 rows

On DataFrames, [ ] refers to columns labels:

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

```
df['new_or_old_col'] = series_or_array
```

EXCEPT! with a slice or mask.

`df[:2]`	DataFrame, first 2 rows
`df[mask]`	DataFrame, rows where mask is True

NEVER CHAIN BRACKETS!

```
> df[mask]['X'] = 1
SettingWithCopyWarning
```

```
> df.loc[mask , 'X'] = 1
```

ENTHOUGHT

Take your Pandas skills to the next level! Register at www.enthought.com/pandas-master-class

Combining DataFrames

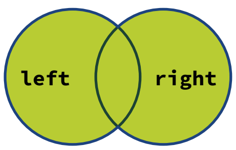
Tools for combining Series and DataFrames together, with SQL-type joins and concatenation. Use join if merging on indices, otherwise use merge.

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



how="outer"

left

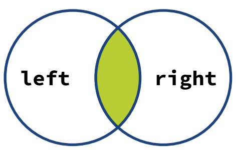
	long	X
0	aaaa	a
1	bbbb	b

left_on='X' right_on='Y'

	long	X	Y	short
0	aaaa	a	—	—
1	bbbb	b	b	bb
2	—	—	c	cc

right

	Y	short
0	b	bb
1	c	cc

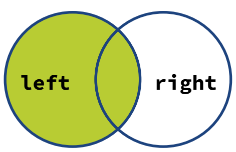


how="inner"

	long	X
0	aaaa	a
1	bbbb	b

	long	X	Y	short
0	bbbb	b	b	bb

	Y	short
0	b	bb
1	c	cc

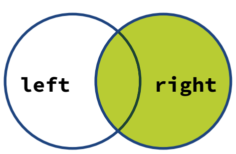


how="left"

	long	X
0	aaaa	a
1	bbbb	b

	long	X	Y	short
0	aaaa	a	—	—
1	bbbb	b	b	bb

	Y	short
0	b	bb
1	c	cc



how="right"

	long	X
0	aaaa	a
1	bbbb	b

	long	X	Y	short
0	bbbb	b	b	bb
1	—	—	c	cc

	Y	short
0	b	bb
1	c	ctc

Concatenating DataFrames

```
> pd.concat(df_list)
```

"Stacks" DataFrames on top of each other.
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 index. Set **on=keys** to join on index of **df** and on **keys** of **other**. **Join** uses **pd.merge** under the covers.

Cleaning Data with Missing Values

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.

Find Missing Values

```
> s_df.isnull()    or    > pd.isnull(obj)
> s_df.notnull()   or    > pd.notnull(obj)
```

Replacing Missing Values

`s_df.loc[s_df.isnull()] = 0`	Use mask to replace **NaN**
`s_df.interpolate(method='linear')`	Interpolate using different methods
`s_df.fillna(method='ffill')`	Fill forward (last valid value)
`s_df.fillna(method='bfill')`	Or backward (next valid value)
`s_df.dropna(how='any')`	Drop rows if any value is **NaN**
`s_df.dropna(how='all')`	Drop rows if all values are **NaN**
`s_df.dropna(how='all', axis=1)`	Drop across columns instead of rows

ENTHOUGHT

Take your Pandas skills to the next level! Register at www.enthought.com/pandas-master-class

Reshaping Dataframes and Pivot Tables

pandas

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, 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

MultiIndex: A Multi-Level Hierarchical Index

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:

`long.loc[1900]`	All 1900 rows
`long.loc[(1900, 'March')]`	value **2**
`long.xs('March', level='Month')`	All March rows

Simpler than using boolean indexing, for example:

```
> long[long.Month == 'March']
```

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

Pivot **column** level **to index**, i.e. "stacking the columns" (wide to long):

```
> df.stack()
```

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

```
> df.unstack()
```

If multiple indices or column levels, use level number or name to **stack**/**unstack**:

`> df.unstack(0)` or `> df.unstack('Year')`

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

```
> (df.groupby(['A', 'B'])['relevant'].mean()
     .unstack().plot())
```

Wide

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

Stack

Unstack

Long

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

Pivot Tables

```
> pd.pivot_table(df,
      index=cols,
      columns=cols2,
      values=cols3,
      aggfunc='mean')
```

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

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

Continent 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)
```

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')
```

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	5

df.pivot() vs **pd.pivot_table**

`df.pivot()`	Does not deal with repeated values in index. It's a declarative form of **stack** and **unstack**.
`pd.pivot_table()`	Use if you have repeated values in index (specify **aggfunc** argument).

Red Panda
Ailurus fulgens

Take your Pandas skills to the next level! Register at www.enthought.com/pandas-master-class