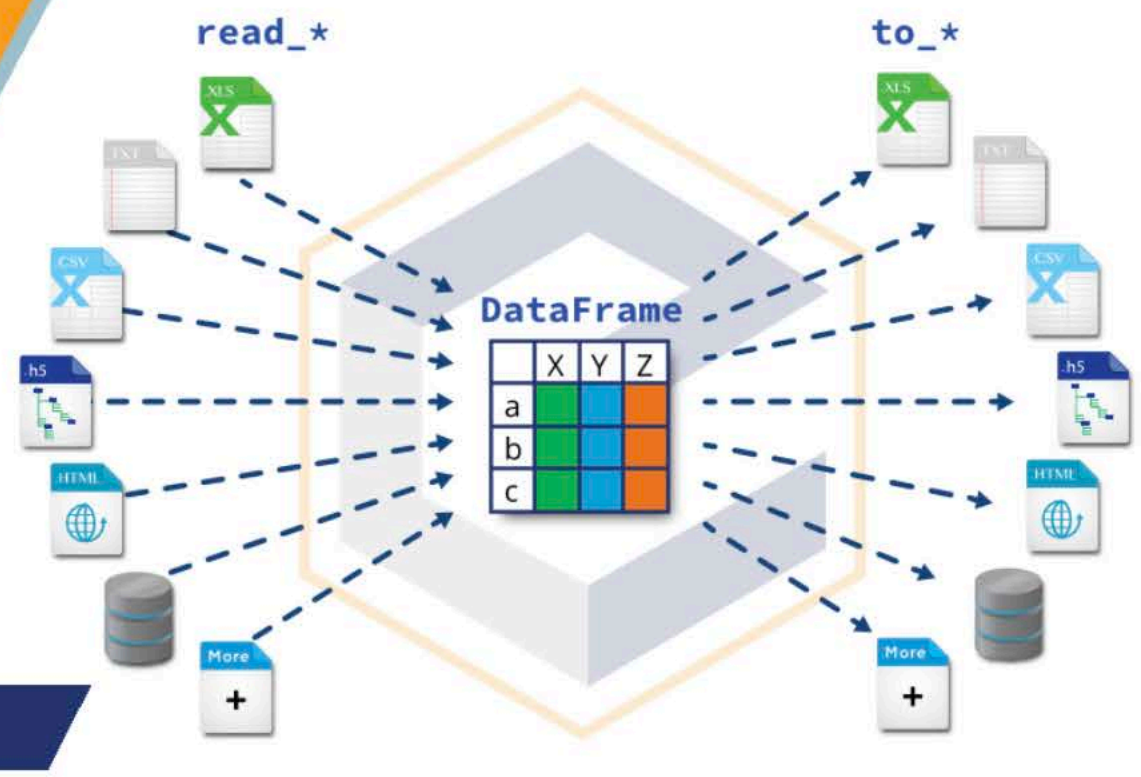
Reading and Writing Data with Pandas

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=['-'])
```

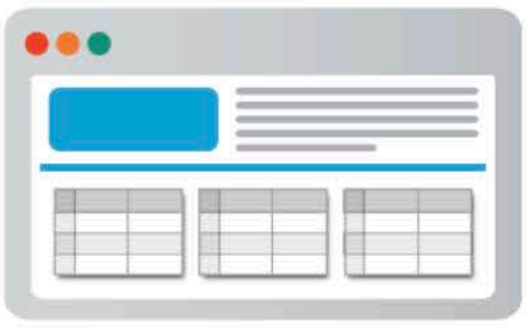
Date	Cs	Rd

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

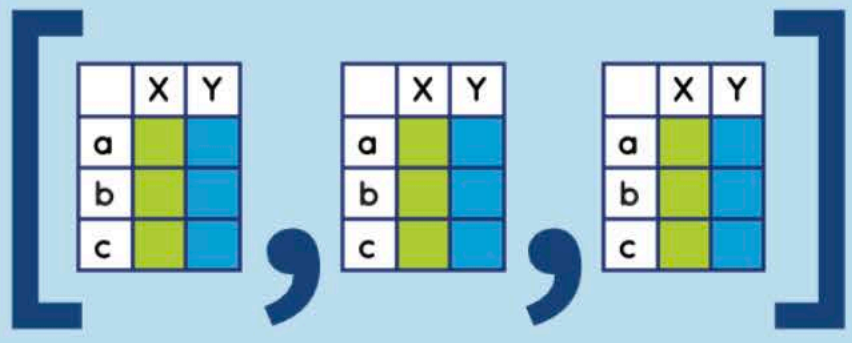
Possible values of **`parse_dates`**:
- **`[0, 2]`**: Parse columns 0 and 2 as separate dates
- **`[[0, 2]]`**: Group columns 0 and 2 and parse as single date
- **`{'Date': [0, 2]}`**: Group columns 0 and 2, parse as single date in a column named Date.

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)
```

ENTHOUGHT

Pandas Data Structures: Series and DataFrames

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

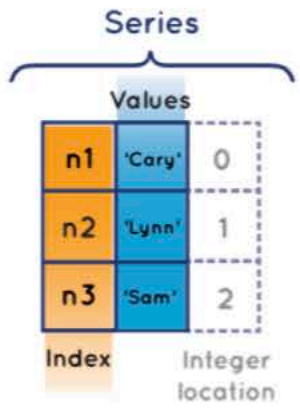
s_df applies to both Series and DataFrames.
Assume that manipulations of Pandas object return copies.

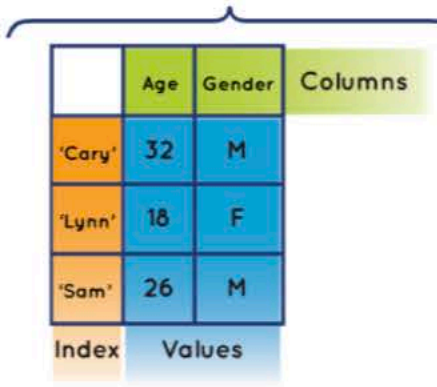
Creating Series and DataFrames

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.sort_index()`	Sorts index labels
`df.set_index(column_name_or_names)`	
`s_df.reset_index()`	Inserts index into columns, resets index to default integer index.

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
```

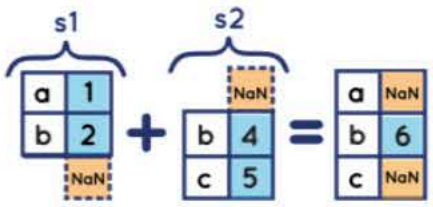
ENTHOUGHT

Computation with Series and DataFrames

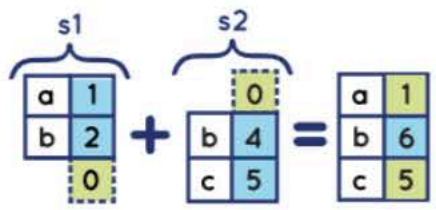
Pandas objects do not behave exactly like Numpy arrays. They follow three main rules (see on the right). Aligning objects on the index (or columns) before calculations might be the most important difference. There are built-in methods for most common statistical operations, such as **mean** or **sum**, 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.

Rule 1: Alignment First

```
> s1 + s2
```



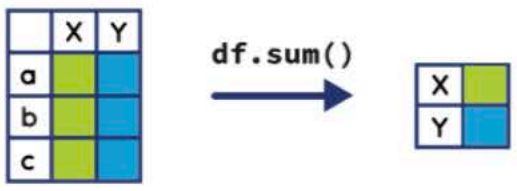

```
> s1.add(s2, fill_value=0)
```



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'**.

- **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

The **3 Rules** of Binary Operations

Rule 1:

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

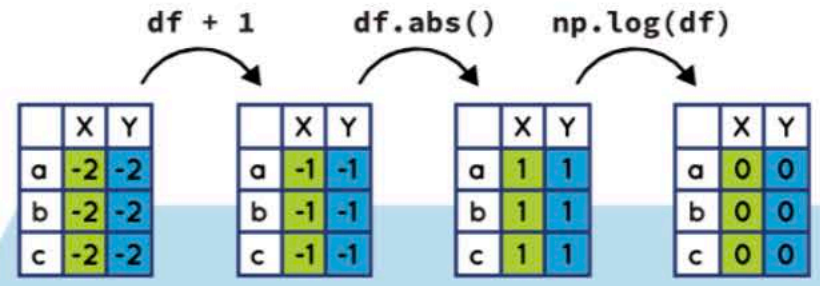
Rule 2:

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



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).

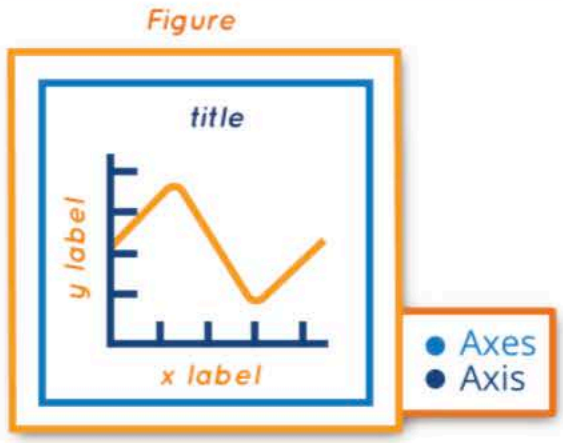
- They propagate in operations across Pandas objects (**1 + NaN → NaN**).
- They are ignored in a "sensible" way in computations, they equal 0 in **sum**, they're ignored in **mean**, etc.
- They stay **NaN** with mathematical operations (**np.log(NaN) → NaN**).

Plotting with Pandas Series and DataFrames

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.

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.



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
```

Plotting with Pandas Objects

Series

With a Series, Pandas plots values against the index:

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

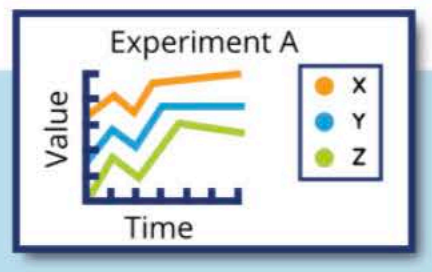
Dataframe

With a DataFrame, Pandas creates one line per column:

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

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).

Labels



Use Matplotlib to override or add annotations:

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

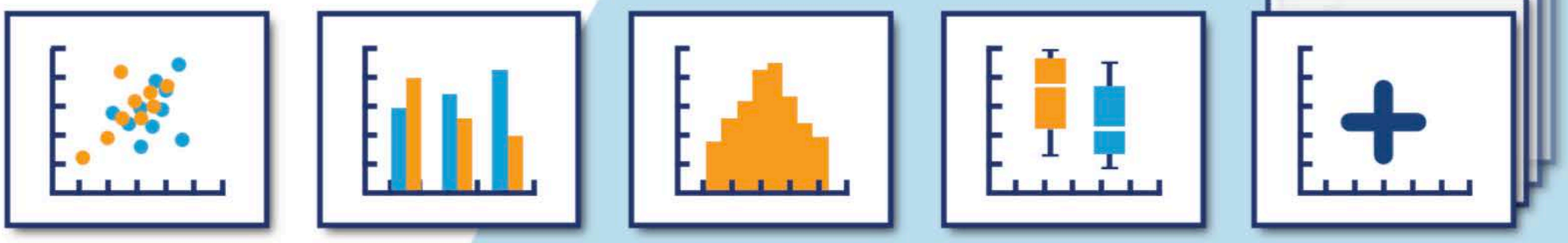
Pass labels if you want to override the column names and set the legend location:

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

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.scatter(x, y)` `df.plot.bar()` `df.plot.hist()` `df.plot.box()`

ENTHOUGHT

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