Functions 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.

Other arguments:
- `names`: Set or override column names
- `parse_dates`: Accepts multiple argument types
- `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.

Writing Data Structures to Disk
Write 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()
```

Writing Data Structures 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)
> df.to_sql(table_name, conn)
```
Within Pandas, there are two primary data structures: Series (s) and DataFrames (df).

- **s**: A Series, which maps an index to values. It can be thought of as an ordered dictionary or a Numpy array with row labels and a name.
- **df**: A DataFrame, which maps index and column labels to values. It is like a dictionary of Series (columns) sharing the same index, or like a 2D Numpy array with row and column labels.

**s_df**: Applies to both Series and DataFrames. Manipulations of Pandas objects usually return copies.

### Creating Series and DataFrames

**Values**

<table>
<thead>
<tr>
<th>Index (n)</th>
<th>Cols</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Cary</td>
</tr>
<tr>
<td>1</td>
<td>Lynn</td>
</tr>
<tr>
<td>2</td>
<td>Sam</td>
</tr>
</tbody>
</table>

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

#### Manipulating Index

- df.reindex(new_index) Conforms to new index
- df.drop(labels_to_drop) Drops index labels
- df.rename(index={old_label: new_label}) Renames index labels
- df.set_index(column_name_or_names) Sorts index labels
- 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) Sorts values
- df.sort_values(['X','Y'], ascending=[False, True]) Sorts values

### Important Attributes and Methods

- s.shape, df.shape (n_rows, n_cols) Shape of Series or DataFrame
- s.dtype, df.dtypes Type of Series or of each column
- len(s) Number of rows
- s.head() and s.tail() First/last rows
- s.unique() Series of unique values (s)
- s.describe() Summary stats
- df.info() Memory usage

### Indexing and Slicing

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

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

### Masking and Boolean Indexing

Create masks with comparisons:

- mask = df['X'] < 0

Or isin, for membership mask:

- mask = df['X'].isin(list_of_valid_values)

Use masks for indexing:

- df.loc[mask] = 0

Combine multiple masks with bitwise operators — and (&), or (|), or (^), not (~) — and group them with parentheses:

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

### Common Indexing and Slicing Patterns

- s.loc[row] Some rows (all columns in a DataFrame)
- df.loc[rows] All rows, some columns
- df.loc[rows, cols] Subset of rows and columns
- df.loc[mask, cols] Boolean mask of rows, some columns
- df.loc[mask] Boolean mask of rows (all columns)

#### Using [] on Series and DataFrames

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

- s['a'] Value
- s[12] Series, first two 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, as shown below:

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

**Never chain brackets**

- NO > df[mask]['X'] = 1
- SettingWithCopyWarning

- YES > df.loc[mask, 'X'] = 1
Pandas objects do not behave exactly like Numpy arrays. They follow three main rules of binary operations.

Rule 1: Operations between multiple Pandas objects implement auto-alignment based on index first.

\[
\begin{align*}
\text{s1} + \text{s2} &= \text{s1.add(s2, fill_value=0)} \\
\text{a} &\quad \text{b} \quad \text{c} \\
\text{1} &\quad \text{2} \quad \text{NaN} \\
\text{b} &\quad \text{4} \quad \text{c} \\
\text{5} &\quad \text{NaN} \\
\text{a} &\quad \text{1} \\
\text{b} &\quad \text{2} \\
\text{c} &\quad \text{5} \\
\end{align*}
\]

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

Rule 2: Mathematical operators (+ - * / exp, log, ...) apply element by element on the values.

\[
\begin{align*}
\text{df} + 1 \quad \text{df.abs()} \quad \text{np.log(df)} \\
\text{X} &\quad \text{Y} \\
\text{a} &\quad \text{-2} \\
\text{b} &\quad \text{-2} \\
\text{c} &\quad \text{-2} \\
\text{a} &\quad \text{-1} \\
\text{b} &\quad \text{-1} \\
\text{c} &\quad \text{-1} \\
\text{a} &\quad \text{1} \\
\text{b} &\quad \text{1} \\
\text{c} &\quad \text{1} \\
\text{a} &\quad \text{0} \\
\text{b} &\quad \text{0} \\
\text{c} &\quad \text{0} \\
\end{align*}
\]

Rule 3: Reduction operations (mean, std, skew, kurt, sum, prod, ...) are applied column by column by default.

\[
\begin{align*}
\text{df.sum()} \\
\text{X} &\quad \text{Y} \\
\text{a} &\quad \text{1} \\
\text{b} &\quad \text{2} \\
\text{c} &\quad \text{3} \\
\end{align*}
\]

Apply a Function to Each Series

Apply `series_to_*` function to every column by default (across rows):

\[
\begin{align*}
\text{df.apply(series_to_value)} &\rightarrow \text{Series} \\
\text{df.apply(series_to_series)} &\rightarrow \text{DataFrame}
\end{align*}
\]

To apply the function to every row (across columns), set `axis=1`:

\[
\begin{align*}
\text{df.apply(series_to_series, axis=1)}
\end{align*}
\]

Apply a Function to a DataFrame

Apply a function that receives a DataFrame and returns a Series, a DataFrame, or a single value:

\[
\begin{align*}
\text{df.pipe(df_to_series)} &\rightarrow \text{Series} \\
\text{df.pipe(df_to_df)} &\rightarrow \text{DataFrame} \\
\text{df.pipe(df_to_value)} &\rightarrow \text{Value}
\end{align*}
\]

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 (\(1 + \text{NaN} \rightarrow \text{NaN}\)).
- 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 such as `np.log(NaN) \rightarrow NaN`.

Differences Between Pandas Objects and Numpy Arrays

When it comes to Pandas objects and Numpy arrays, 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.

Apply a Function to Each Value

Apply a function to each value in a Series or DataFrame:

\[
\begin{align*}
\text{s.apply(value_to_value)} &\rightarrow \text{Series} \\
\text{df.applymap(value_to_value)} &\rightarrow \text{DataFrame}
\end{align*}
\]
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 zoomable 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()

Note: 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.

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()`
Converting Objects to Time Objects

Convert different types like strings, lists, or arrays to Datetime with:

\[
\text{pd.to_datetime(value)}
\]

Convert timestamps to time spans and set the period “duration” with frequency offset.

\[
\text{date_obj.to_period(freq=freq_offset)}
\]

Frequency Offsets

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

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

Save Yourself Some Pain: Use ISO 8601 Format

To be consistent and minimize the risk of error or confusion, use ISO format YYYY-MM-DD when entering dates:

**NO**

\[
\text{pd.to_datetime('12/01/2000')} # 1st December
\]

Timestamp('2000-12-01 00:00:00')

**NO**

\[
\text{pd.to_datetime('13/01/2000')} # 13th January!
\]

Timestamp('2000-01-13 00:00:00')

**YES**

\[
\text{pd.to_datetime('2000-01-13')} # 13th January
\]

Timestamp('2000-01-13 00:00:00')

Creating Ranges of Periods

\[
\text{pd.period_range(start=None, end=None, periods=None, freq=offset)}
\]

Resampling

\[
\text{s_df.resample(freq_offset).mean()}
\]

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.upper()`
- `s.str.strip()`
- `s.str.isupper()`
- `s.str.normalize()`
- `s.str.len()`
- `s.str[0]`
- `s.str.contains(str_or_pattern)`

Splitting and Replacing

Split returns a Series of lists:

\[
\text{s.str.split()}
\]

Access an element of each list with get:

\[
\text{s.str.split(char).str.get(1)}
\]

Return a DataFrame instead of a list:

\[
\text{s.str.split(expand=True)}
\]

Find and replace with string or regular expressions:

\[
\text{s.str.replace(str_or_regex, new)}
\]

\[
\text{s.str.extract(regex)}
\]

\[
\text{s.str.findall(regex)}
\]
Combining DataFrames

There are numerous 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 the section on the how keyword.

Use on if merging on same column in both DataFrames, otherwise use `left_on`, `right_on`.

**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.

**Concatenating DataFrames**

>`pd.concat(df_list)`

“Stacks” DataFrames on top of each other. Set `ignore_index=True` to replace index with `RangeIndex`. join uses `pd.merge` under the covers.

**CLEANING DATA WITH MISSING VALUES**

Pandas represents missing values as NaN (Not a Number), which comes from Numpy and is of type `float64`. To find and replace these missing values, you can use any number of methods.

### To find missing values, use:

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

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

### To replace missing values, use:

```python
s_df.loc[s_df.isnull()] = 0
s_df.interpolate(method='linear')
```

Interpolate using different methods

```python
s_df.fillna(method='ffill')
```

Fill forward (last valid value)

```python
s_df.fillna(method='bfill')
```

Or backward (next valid value)

```python
s_df.dropna(how='any')
```

Drop rows if any value is NaN

```python
s_df.dropna(how='all')
```

Drop rows if all values are NaN

```python
s_df.dropna(how='all', axis=1)
```

Drop across columns instead of rows
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: Group By**

Group by a single column:
> g = df.groupby(col_name)

Grouping with list of column names creates a DataFrame with a MultiIndex:
> g = df.groupby(list_col_names)

Pass a function to group based on the index:
> g = df.groupby(function)

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

apply is more general than agg, transform, and filter. It 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.apply(normalize())

**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})

**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.
- resample is often used before rolling, expanding, and ewm when using a DateTime index.
Let's explore some 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. It is easier to filter, aggregate, transform, sort, and pivot. Reshaping operations often produces multi-level indices or columns, which can be sliced and indexed.

**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:

```python
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:

```python
> long[(long.Month == 'March')]
```

**Pivot Tables**

```python
> 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`.

```python
df.pivot() vs pd.pivot_table()
```

- `df.pivot()` does not deal with repeated values in index.
- `pd.pivot_table()` is a declarative form of `stack` and `unstack`.
- Use if you have repeated values in index (specify `aggfunc` argument).

**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).

```python
> 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')
```

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

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

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

Pivot index level to columns, “unstack the columns” (long to wide):

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

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

```python
> df.unstack(1) or df.unstack('Month')
```

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

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

©2020 Enthought, Inc., licensed under the Creative Commons Attribution – Non-Commercial, No Derivatives 4.0 International License. To view a copy of this license, visit creativecommons.org/licenses/by-nc-nd/4.0/