Week 1: Pandas Tutorial for Data Wrangling

Documentation & Notes

Kevin Miao, Youngli Hong, Sandeep Sainath, Amanda Ma (Team DSS)
EECS189 - Introduction to Machine Learning
Project T (Final)
University of California, Berkeley

1 Introduction

The Python ecosystem is universally popular in practical machine learning applications. Among the various libraries in this ecosystem, Pandas is the most popular choice for data wrangling for all machine learning practitioners - industry professionals, research scientists, and students like yourself alike. In this documentation, we cover the most essential concepts and syntax relevant to using Pandas in your own machine learning applications.

We begin with introducing the primary data structures in Pandas and how to use basic functionality to import, view, and manipulate data in theses structures. Then, we introduce more advanced functionality through key data aggregation concepts such as functions, grouping, and merging data. We also briefly cover common trends and practices in data wrangling, and end with a brief case study/example to reinforce the concepts covered in this documentation.

To begin, the conventional Pandas import statement in Python is "import pandas as pd". This will be assumed in conjunction with the conventional NumPy import statement "import numpy as np".

2 Data Structures

The primary data structures in Pandas consist of DataFrames and Series.

2.1 DataFrames

DataFrames are the analog of tabular datasets in Pandas. They are a 2-dimensional data structure with rows and columns. Objects in DataFrames can be comprised of most conventional data types - strings, numbers (integers or floats), lists, etc. DataFrames are indexed by Index objects, which can be either one-dimensional (example: 0, 1, 2, 3...) or multi-dimensional, also known as MultiIndex objects. Indexing will be covered later under Viewing DataFrames. Throughout this tutorial, we will follow convention and refer to 'df' as our loaded DataFrame.

To create your own DataFrame, use the pd.DataFrame() method to pass in dict-like data, with key-value pairs corresponding to column-row mappings. To view the columns in your DataFrame, check the df.columns attribute.

2.2 Series

Series are the analog of columns of data in Pandas. Series generally take the form of arrays and lists, with a common data type (strings, integers, etc.) in the entire Series. Series are similar to Numpy arrays in that most common array-like operations are element-wise.

DataFrames and Series are fundamentally different objects. A Series may be part of a DataFrame as a column, but a DataFrame cannot be part of a Series. Thus, there is no other relation between them.

To create your own Series, use the pd.Series() method to pass in a dictionary, array, or scalar value.

2.3 Data Types

Most data types in Pandas are native to Python. These include integers, floats, strings, booleans, and datetime objects. These are all classified as **dtypes** in Pandas. The .dtypes attribute returns all available data types in a particular DataFrame or Series, making it an excellent way to check the types of data once data has been imported.

3 Basic Functionality

3.1 Importing Data

Pandas has functionality to import several file types into DataFrames.

To import CSV files, use the 'pd.read_csv()'method. While most Excel files can also be formatted as CSV files, the 'pd.read_excel()' method works just as well for Excel files. If not well-formatted, arguments such as 'sep' to specify delimiters and 'header' to specify the row to use as column names for your DataFrame are useful to read in a clean DataFrame.

To import JSON files, use the 'pd.read_json()' method. As JSON files come in various formats, use the 'orient' keyword in the method to indicate the format that the input JSON file is in.

Primary arguments to these methods are strings of exact file names if in the same directory, or file paths to the same files.

3.2 Viewing DataFrames

Once a DataFrame is ready for use, the .head() and .tail() methods can be used to view the first n and the last n rows of a DataFrame. The .head() is especially useful for a quick scan of the structure of a DataFrame either after being imported or manipulated in some way.

3.3 Indexing

As aforementioned, indexing in Pandas is structured through the Index class. This class can be used to create Index and MultiIndex objects, depending on the granularity of the data's index. Index objects are immutable and thus cannot be edited.

3.3.1 .loc() vs. iloc()

There are two ways to access data via indexing in Pandas: .loc() and .iloc(). The key difference between these functions is that .loc() function is used to access a group of rows and columns by **label**, whereas the .iloc function is used to access the same by **position**.

Note that these functions are available to both DataFrame and Series objects. It is also possible to use slicing in place of a label or position for .loc() and .iloc() respectively.

3.3.2 Transforming the Index

The two primary ways to transform a DataFrame index are to **set an index** and to **reset an index**. These are done using .set_index() and .reset_index() respectively.

Setting an index is useful when either the raw dataset comes with no meaningful index and/or if there is another column in the same DataFrame that is more appropriate to be an index. Resetting an index is useful when either the raw dataset comes with no meaningful index and/or the index is required for manipulation as a Series rather than an Index object, which is immutable.

3.4 Data Selection

3.4.1 Subsetting Data

Aside from indexing, bracket notation can also be used to subset certain data. The df['column name'] syntax can be used to select a single column, and df[[list_of_columns]] can be used to select multiple columns.

Boolean subsetting is another common way to filter and select data. For example, if we wanted all rows in a DataFrame with a numerical value of a particular column greater than 10, we can simply write:

df[df['column name'] > 10].

It is also possible to specify multiple conditions to filter data. These can be used with the symbols & (and) and | (or). For example, if we wanted the condition of a string value of a particular column equal to "Hello" in addition to the aforementioned condition, we can write:

df[(df['column_1'] > 10) & (df['column_2'] == "Hello")]

Note that in this case, each condition is surrounded by parentheses.

3.4.2 Creating & Dropping Columns

To create a column in a DataFrame using any Series or list-like object, we can write:

df['new_column_name'] = Series_or_list_of_data

Note that the Series or list passed in on the right side of the expression must have length equivalent to the number of rows in the DataFrame.

To drop a column, use df.drop('column_name', axis=1). The axis=1 parameter indicates that we want to drop a column, not a row (to drop rows, use axis=0). Similarly to creating columns, the 'column_name' argument can be replaced with a list of columns to be dropped.

Note that this method has the "inplace" default parameter, which indicates whether this method will mutate an existing DataFrame or return a new one with the specified column(s) dropped. This parameter is False by default.

3.4.3 Renaming Columns

To rename a column in a DataFrame, use the df.rename() method. The method takes in a dict-like object for the columns parameter with key-value mappings corresponding to old column names as keys and their respective new names as values. Here is an example:

df.rename(columns='old1':'new1', 'old2':'new2')

Similar to df.drop(), this method also contains an inplace=False default parameter that can be modified.

3.4.4 Sorting Values

Values in a DataFrame and Series can be sorted in any order using the sort_values() method. For DataFrames, a column to sort by must be specified under the "by" parameter. For both DataFrames and Series, the method takes in an "axis" parameter (0 to sort by index, 1 to sort by columns) and an "ascending" parameter (False by default). Here are some examples:

Sort by the "date" column in ascending order: df.sort_values(by='date', axis=1, ascending=True)

Sort the "order_number" column in descending order: df["number"].sort_values(axis=0, ascending=False)

By default, these methods return mutated DataFrames and Series. Note that the method for DataFrames also has the "inplace" argument to mutate the DataFrame in place.

3.4.5 Dropping Duplicates

The df.drop_duplicates() method returns a DataFrame with duplicate rows removed. This method with no additional parameter specified drops rows that are entirely duplicates, in the sense that they are the same across all columns in the DataFrame. To specify only certain columns to be considered, use the "subset" parameter to pass in the column name or a list-like object containing names.

In addition, the "keep" parameter determines which duplicates, if any, to keep. By default this parameter is "first", indicating that the first occurrence of a series of duplicates will be kept. This can also be modified to "last". Finally, the "inplace" parameter indicates whether to mutate the DataFrame in place or to return it.

3.5 Descriptive Statistics

Pandas has very easy ways to gain descriptive statistics about DataFrames and Series. These from basic functions like .mean(), which retrieves the arithmetic mean, to .describe(), which returns the mean, standard deviation, count, and quartiles. The .corr() function is also particularly useful with its calculation of correlation. Finally, the .value_counts() method for Series yields the frequency distribution of each unique value of a particular Series, in descending order.

4 Advanced Functionality

This section goes over syntax for the most common data aggregation methods in Pandas.

4.1 Applying Functions

Applying functions is quite intuitive in Pandas due to the capability to perform elementwise operations, similar to NumPy operations. The .apply() function allows you to pass in any function, built-in or custom, to apply along any axis of a DataFrame; the "axis" parameter in df.apply() with value 0 corresponds to applying a function over its rows, while value 1 corresponds to columns. For Series, the only axis is along the entire column.

The final return type is inferred from the function passed in. For example:

- using the NumPy function "np.sum" aggregation function to calculate the arithmetic sum in .apply() will return one value, aka the sum of the specified Series.

- using a custom function such as "lambda x: x + 2", which adds 2 to every row, will return a DataFrame/Series with the column modified appropriately.

Note that custom functions can be either passed in as lambdas or as pre-defined functions. This gives great flexibility to the .apply() function in data aggregation and wrangling.

4.2 Concatenating Data

It is sometimes useful to combine two separate DataFrames that share similar schema, but contain different data. The .concat() method allows you to concatenate multiple DataFrames and Series together along any axis. To combine two DataFrames, df1 and df2, here is the syntax:

- Concatenate vertically (along rows): pd.concat([df1, df2], axis=0)

- Concatenate horizontally (along columns): pd.concat([df1, df2], axis=1)

Note that this syntax is exactly the same for two Series objects instead of DataFrames.

4.3 Grouping Data

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups (Pandas documentation).

The df.groupby() function enables this functionality. It is possible to group by any number of columns by passing in an ordered list of columns into the "by" parameter. The function returns a

DataFrameGroupBy object, a special class in Pandas. While this object by itself is not useful, it is possible to perform an aggregation function of any kind on this object. The result is a regular DataFrame that is indexed by an Index or MultiIndex object depending on the number of groups.

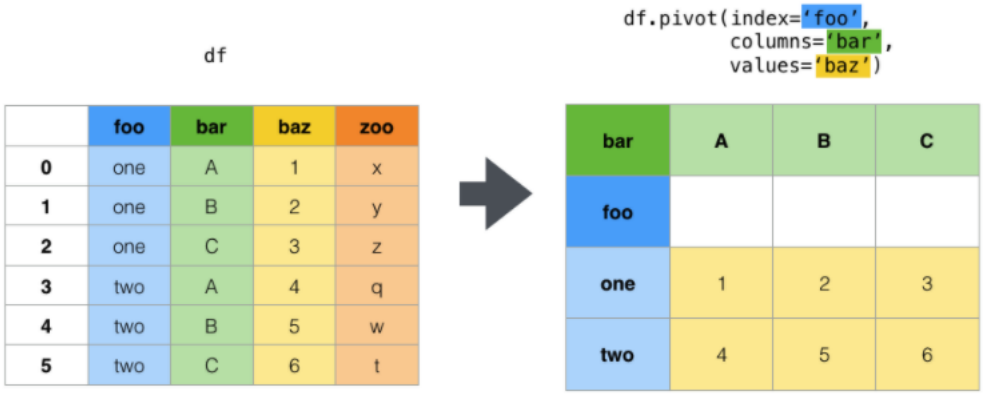
Aggregation functions can take the form of both built-in functions and custom functions. Common built-in functions include .mean(), .max(), .min(), .sum(), and .count(). These can be directly used on DataFrameGroupBy objects to gain the appropriate aggregated data. For custom functions, the .agg() function takes in any lambda or pre-defined function and applies to a DataFrameGroupBy object.

4.4 Reshaping Data

4.4.1 Pivots

A pivot, as the name suggests, is a rotational transformation of a DataFrame. At a basic level, it reorganizes a DataFrame such that the table is indexed by one or more of its columns. In addition, it is also possible to replace the remaining columns with either the original index or another column, with the values in the table themselves referring to another variable. As you can imagine, pivots provide a lot of flexibility in representing data and are one of the most functional forms of data wrangling.

Figure 1: Example of using df.pivot()



	foo	bar	baz	zoo
0	one	A	1	x
1	one	B	2	y
2	one	C	3	z
3	two	A	4	q
4	two	B	5	w
5	two	C	6	t

bar	A	B	C
foo			
one	1	2	3
two	4	5	6

The df.pivot() method enables this functionality. It takes in three main parameters: "index", "columns", and "values". Above is an example of this functionality on a DataFrame with 4 columns. The syntax above the **pivot table** on the right shows how the unique values of the column "foo" becomes the index, with the unique values of the column "bar" becoming the columns, and the corresponding values of the column "baz" in the original table simply becoming the values in the new DataFrame according to their "foo" and "bar" values.

An alternative to the df.pivot() method is the pd.pivot_table() method, which gives additional functionality to use aggregation functions for computing values. Like df.pivot(), this function also takes "values" and "index" parameters, along with an "aggfunc" parameter that enables aggregating columns in the "values" parameter to be aggregated in a specific way. This parameter takes either a single built-in or custom function, or a dict-like object for multiple "values" columns with key-value pairings corresponding to columns as keys and functions as values.

4.4.2 Stacking and Unstacking

Stacking are based around the idea of MultiIndex objects in DataFrames. Stacking involves adding a column as the next additional level to a DataFrame, whereas unstacking is the opposite - removing a column from the index of a DataFrame and reverting it to a column. Like pivots, these operations are useful to wrangle data and represent data in the tabular format best suited to your project.

Figure 2: Example of using df.stack()

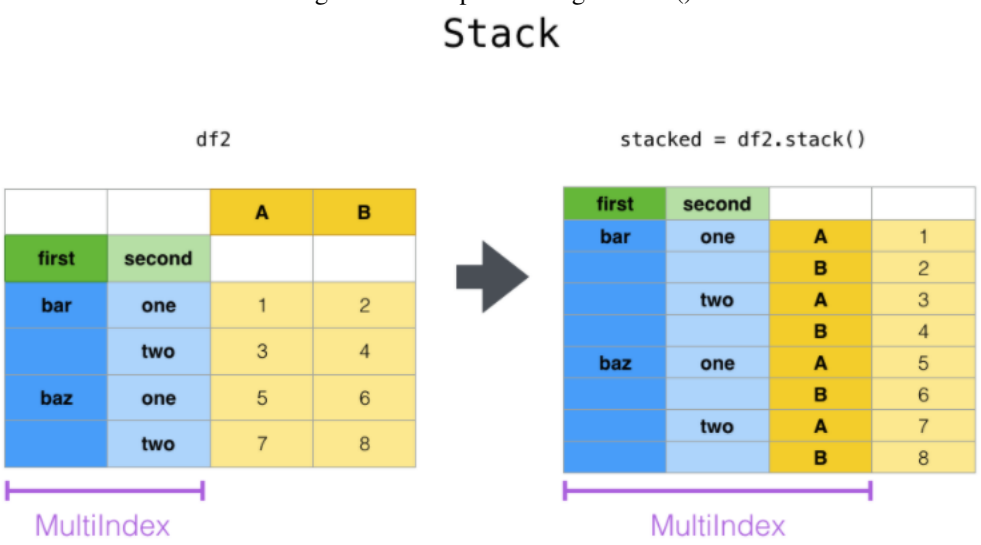


Figure 2 is an example of stacking using the .stack() method. By simply using the function, we convert all remaining columns to an index, thereby extending the existing MultiIndex.

Figure 3: Example of using df.unstack()

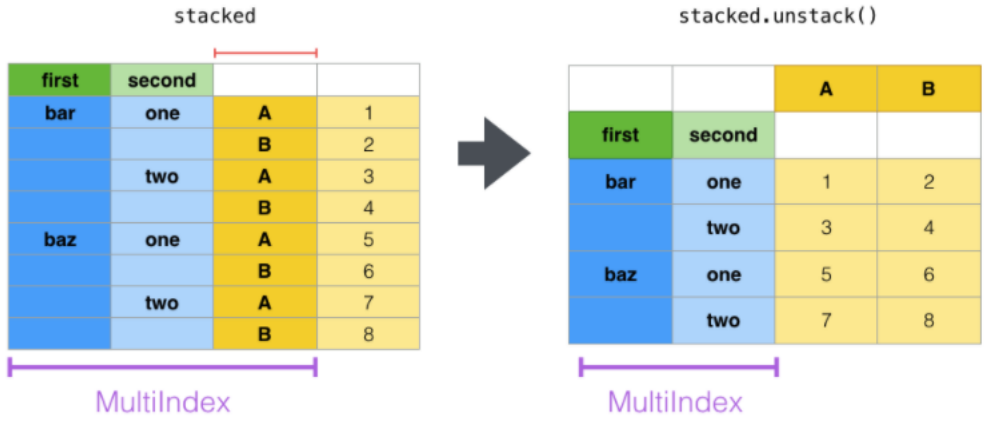
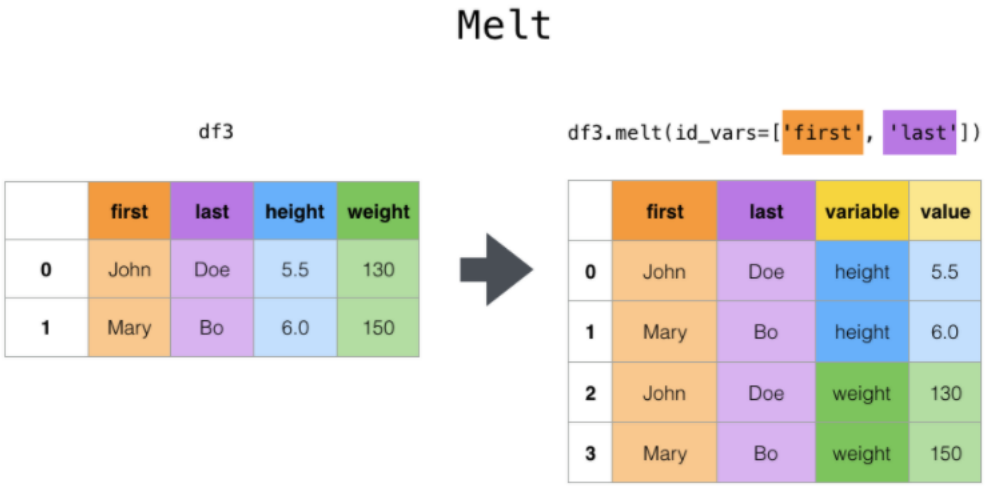


Figure 3 is an example of unstacking using the .unstack() method. By applying this method, we see that this is simply a reversal of our transformation using the .stack() method. The latest level in the MultiIndex, or the rightmost column, is "unstacked" and reverts to being a column, with the same values in the DataFrame persisting.

4.4.3 Melts

Melting is yet another way to reshape a DataFrame. A melt involves "unpivoting" columns over columns that are identifier variables.

Figure 4: Example of using .melt()



The df.melt() function enables this functionality. In the example above, the identifier variables are columns "first" and "last. Through specifying this with the "id_vars" parameter through a list, the columns together become a kind of indexer (note: **not** a MultiIndex).

By default, the function unpivots all columns that are not identifier variables. This can be modified by specifying columns not to unpivot with the "value_vars" parameter.

4.5 Merging Data

Pandas has extensive functionality to perform traditional database-style joins. These include inner joins, left joins, right joins, and outer joins.

The pd.merge() function enables this functionality. The function takes several key parameters:

- **left**: the DataFrame on the left.
- **right**: the DataFrame on the right.
- **how**: the type of join ("inner" by default, "left", "right", "outer")
- **on**: column name to join on; use this if this column name is present in both DataFrames
- **left_on**: joiner column from **left**; unnecessary if **on** is correctly specified.
- **right_on**: joiner column from **right**; unnecessary if **on** is correctly specified.
- **suffixes**: takes in a tuple of two suffixes corresponding to the left and right DataFrames; these suffixes are added to column names if they exist in both DataFrames (which results in duplicate names for varying data).

5 Data Wrangling in Practice

5.1 Missing Value Imputation

It is not always the case that, like in most coursework, data is perfectly clean. Missing values are a very common culprit in unclean, messy data. This poses a threat to machine learning applications.

Sometimes, large datasets provide the luxury of simply dropping these rows or columns with largely missing data. However, smaller datasets require some kind of imputation to maximize the amount of training data available for model performance.

In this section, we go over a fairly simplistic but useful way to go about missing value imputation for most cases. In the case of relatively small datasets, the method of imputation largely depends on the rest of the data. For our purposes, we will assume that "missing values" refer to values that do not exist, or "NaNs" in Pandas.

5.1.1 Detecting Missing Values

The easiest way to detect missing values, aside from observation, is the .isna() function. This function, when applied on a Series or DataFrames, returns elementwise booleans corresponding to whether that value is a NaN or None value, or not. For datetimes, this is a NaT value.

5.1.2 Deciding Method of Imputation

This section will assume that only a minority of values are missing along any given set of rows or columns. If a predominant amount of values are missing, any effort to impute values might be in vain.

As aforementioned, the method of imputation largely depends on the context of the data. There are situations where filling in a standard value, such as 0, is appropriately meaningful for a particular column. This might be due to the type of data represented itself, or something more trivial like the knowledge that missing values in imported data were automatically read in by Pandas as NaNs.

If a constant, fixed value is not meaningful, numerical data has the option of using a descriptive statistic such as the mean, mode, or median. Considerations for this kind of imputation are the distribution of the data, both univariate and joint with other columns, as well as the context of the data as always.

5.1.3 Filling in Missing Values

Regardless of method of imputation, the fillna() method is the fastest way to fill in missing values along any axis in a DataFrame and Series. This method takes in any kind of object valid to insert in a DataFrame or Series under the "value" parameter and fills it in along the axis specified by the "axis" parameter (0 for along rows, 1 for columns).

5.2 Iteration

Pandas has several small and large optimizations alike to ensure fast operations over large datasets. The most prominent of these that we have covered in this documentation is the apply() convenience function.

The most direct analog of the use cases of this function are regular Python for loops; both are used for elementwise operations. To iterate through rows, the iterrows() function in Pandas enables the traditional form of iteration. The analog for this for columns would be a simple "for col in df.columns" statement, which accesses the .columns attribute of DataFrames.

The advantage of the apply() function is most obvious when compared to iteration with iterrows(). It is well-documented that the apply() function is significantly faster than iterrows() to perform rowwise operations. Since virtually any elementwise row operation performed with iterrows() can be performed by apply(), and also considering the advantage of significantly less code written while using apply(), we strongly recommend its usage over iterrows() for this purpose.

5.3 Time Series Data

Pandas has functionality drawing from NumPy's datetime and timedelta objects, which can be used in turn to create time series data in DataFrames and Series. Below is an overview of the most common time object representations in Pandas.

Figure 5: Overview of time related data representations in Pandas

Concept	Scalar Class	Array Class	pandas Data Type	Primary Creation Method
Date times	`Timestamp`	`DatetimeIndex`	`datetime64[ns]` or `datetime64[ns, tz]`	`to_datetime` or `date_range`
Time deltas	`Timedelta`	`TimedeltaIndex`	`timedelta64[ns]`	`to_timedelta` or `timedelta_range`

Timestamp objects are ideal to represent full dates and times (for example: 05/24/2020, 2:54PM), whereas Timedelta objects are more suited toward measurements of change in time (for example: 1 day, 6 hours, 2 minutes, 53 seconds).

5.3.1 Converting Time Series Data to Pandas Representation

The to_datetime and to_timedelta methods are particularly useful to convert standard time series data into the proper Pandas Timestamp and Timedelta objects respectively for ease of manipulation and calculation.

For both methods, it is usually enough to simply pass in a Series or list-like object of time series data for automatic conversion to the appropriate objects. However, Pandas also offers ways to specify custom date and time formats through parameters like "dayfirst", which uses dates that start with the day first, and the "format" parameter, which allows specific parsing of dates to fit a particular format (for example: "%Y/%m/%d" for "2010/11/12").