Pandas Cheat Sheet

by
Ted Petrou

Contents

Chapter 1

Pandas Cheat Sheet

This notebook provides a summary of the most important and useful pandas commands.

1.1 Resources

- Master Data Analysis with Python - Book by Ted Petrou - 850 Pages, 350 Exercises
- Official Pandas Documentation

1.2 Jupyter Notebook Overview

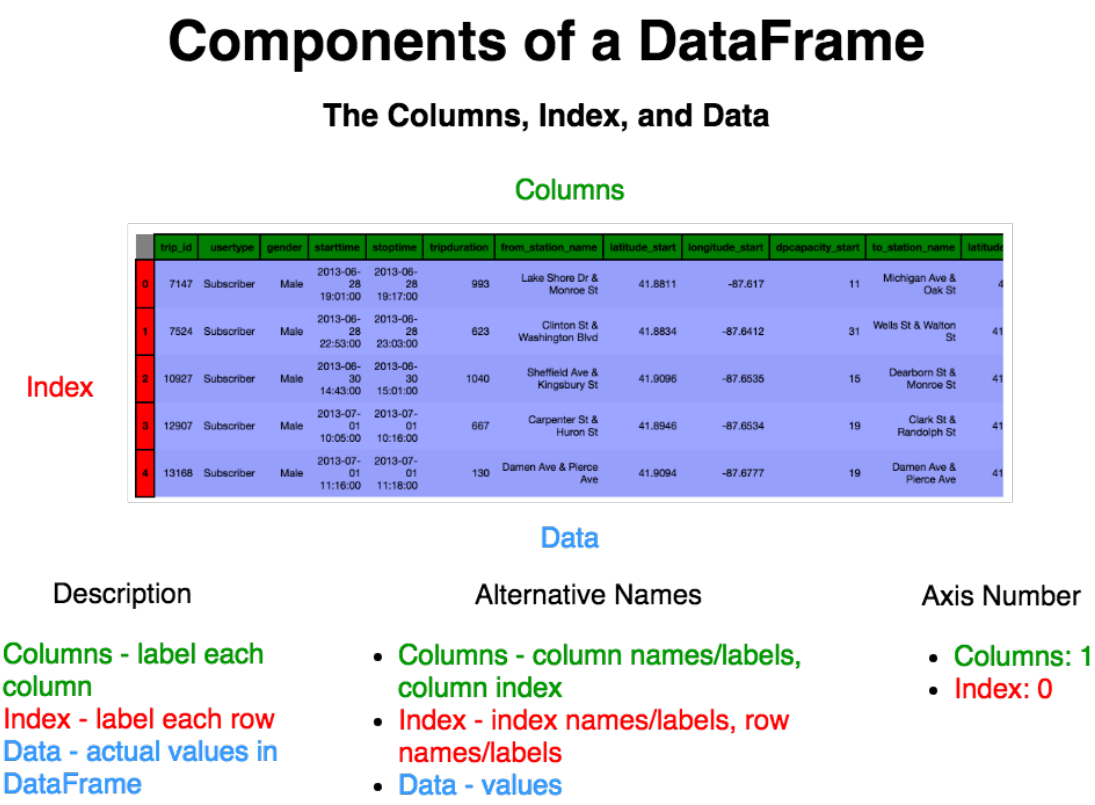
- Jupyter Notebooks are composed of **cells**
- There are two main **types** of cells
  - **Code** cells
    - Understand python code
    - Have `In [ ]` directly to the left of them
  - **Markdown** cells
    - Understand markdown (a simple plain text formatting language)
    - Nothing to the left of them
- Each cell has two separate **modes**
  - **Edit** mode - keys work like they do in a text editor
    - Cell is outlined in green
    - Flashing cursor inside the cell
  - **Command** mode - keys have special meaning and do not output to the screen
    - Cell is outlined in blue
    - No flashing cursor
- Keyboard shortcuts
  - Press **ESC** to switch to command mode
  - Press **ENTER** to switch to edit mode
- Keyboard shortcuts in command mode:
  - `A` - Add a new cell above
  - `B` - Add a new cell below
  - `X` - Delete a cell (and copy it)
  - `M` - Change cell type to markdown
  - `Y` - Change cell type to code
  - `S` - Save notebook
  - `H` - Get all keyboard shortcuts for both modes
- Executing code cells:

  - **Shift + enter** - execute and move to next cell in command cell OR if last cell create new code cell in edit mode
  - **Ctrl + enter** - execute and stay in current cell in command mode
- Help in the notebook
  - Press **Tab** to show list of options when typing
  - Press **Shift + Tab + Tab** at the end of a function/method to reveal the docstring as a popup window
- Keep your hands on the keyboard. Do not use your mouse to switch from edit to command mode

1.3 Pandas

- Name derived from panel + data
- Used for two-dimensional, 'tabular' data analysis
- The **DataFrame** and **Series** are the primary containers of data

1.3.1 DataFrame Definition



	trip_id	usertype	gender	starttime	stoptime	tripduration	from_station_name	latitude_start	longitude_start	dpcapacity_start	to_station_name	latitude
0	7147	Subscriber	Male	2013-06-28 19:01:00	2013-06-28 19:17:00	993	Lake Shore Dr & Monroe St	41.8811	-87.617	11	Michigan Ave & Oak St	4
1	7524	Subscriber	Male	2013-06-28 22:53:00	2013-06-28 23:03:00	623	Clinton St & Washington Blvd	41.8834	-87.6412	31	Wells St & Walton St	41
2	10927	Subscriber	Male	2013-06-30 14:43:00	2013-06-30 15:01:00	1040	Sheffield Ave & Kingsbury St	41.9096	-87.6535	15	Dearborn St & Monroe St	41
3	12907	Subscriber	Male	2013-07-01 10:05:00	2013-07-01 10:16:00	667	Carpenter St & Huron St	41.8946	-87.6534	19	Clark St & Randolph St	41
4	13168	Subscriber	Male	2013-07-01 11:16:00	2013-07-01 11:18:00	130	Damen Ave & Pierce Ave	41.9094	-87.6777	19	Damen Ave & Pierce Ave	41

- Two-dimensional - rows and columns
- Think of a DataFrame as a collection of columns. Pandas thinks column-wise
- Three components - **index**, **columns** and **data (values)**
- The index labels the rows and the column names label the columns
- Access each component with attributes:
  - `df.index` - returns a pandas Index object - default index is a `RangeIndex`
  - `df.columns` - returns a pandas Index object
  - `df.values` - returns numpy array
- Most common way to create a DataFrame is with `pd.read_csv('file_name.csv')`
- Use `head/tail` methods to shorten long output
  - `df.head()` - returns first 5 (by default) rows
  - `df.tail(10)` - returns last n (10 in this case) rows

1.3.2 Data Types

Every column of a Pandas DataFrame has a particular **data type**. The data type is very important and informs us that each value within a particular column is of that data type.

Most Common Data Types

There are many data types with the following being more common:

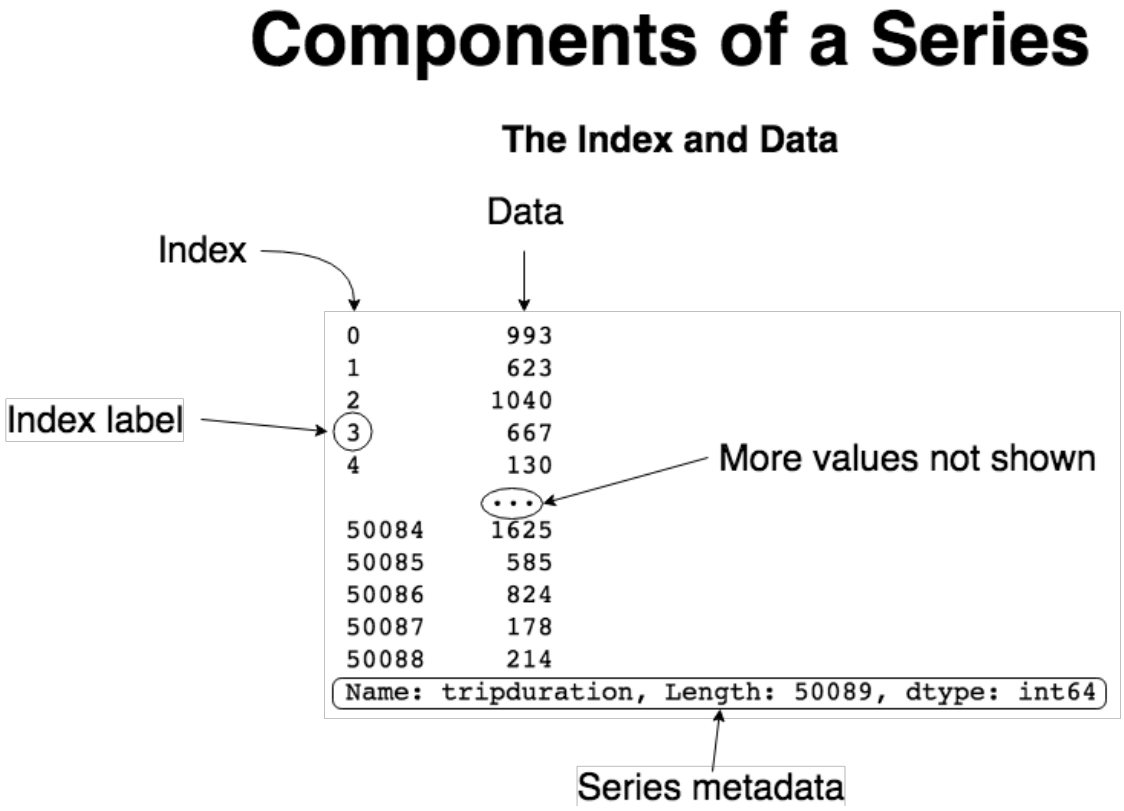
- Boolean
- Integer
- Float
- Object (can be any Python object but is usually strings)
- Datetime
- Timedelta
- Period
- Categorical

1.3.3 Missing Value Representation

Pandas uses two representations for missing values: `NaN`, `NaT`

- NaN - not a number, found only in float or object columns
- NaT - not a time, found only in Datetime, Timedelta, and Period columns
- No missing values for integer or booleans (an unfortunate Pandas limitation)
- In pandas 1.0+, there is a separate `pd.NA` and nullable integer/boolean data types

1.3.4 Series Definition



- One dimension of data
- Two components to a Series - the **index** and the **data (values)**

- Access each component
  - `s.index` - returns a pandas Index object - default index is a `RangeIndex`
  - `s.values` - returns a numpy array

1.3.5 Common DataFrame Attributes

- `df.index`
- `df.columns`
- `df.values`
- `df.shape` - tuple of `(rows, columns)`
- `df.size` - Total elements in DataFrame - rows x columns
- `df.dtypes` - Returns each data type of a column as a Series

1.3.6 Common Series attributes

- `s.index`
- `s.values`
- `s.size` - can also find number of Series elements with `len(s)`
- `s.dtype`

1.3.7 Built-in `len` function

Passing in the DataFrame or Series to the `len` function returns the number of rows (DataFrame) or number of values (Series) * `len(df)` * `len(s)`

1.3.8 Setting a Meaningful index on a DataFrame

- Default index is a `RangeIndex`
  - Consecutive integers 0 to n-1
- Not necessary to set a custom index with DataFrame
  - All data analysis is possible without setting an index
- If you do set an index, choose one that is both unique and descriptive. Uniqueness is not enforced
- Set the index on read with the `index_col` parameter. `pd.read_csv('file_name.csv', index_col='title')`
- Set the index after read with `df.set_index('title')`
- Turn the index into the first column of a DataFrame with `df.reset_index()`

1.3.9 Five step process for Doing Data Science a Jupyter Notebook

See the blog post for more

1. Write and execute a single line of code to explore your data
2. Verify that this line of code works by inspecting the output
3. Assign the result to a variable
4. Within the same cell, in a second line, output the head of the DataFrame or Series
5. Continue to the next cell. Do not add more lines of code to the cell

1.3.10 DataFrame Subset Selection

- Select rows, columns, or rows and columns simultaneously
- Three **indexers** each with their own rules - `[ ]` (just the brackets), `loc`, `iloc`

Just the brackets

- Primarily used to select one or more columns
- `df['col1']` - selects a single column as a Series. Column name is usually a string
- `df.col1` also selects a single column, but do NOT use. It doesn't work for columns with spaces
- Use a list of column names to select multiple columns as a DataFrame. `df[['col1', 'col2', 'col3']]`. Notice the inner list

The`loc` indexer

- Different rules than just the brackets
- Selects primarily by **label**
- Simultaneously selects rows and columns with `df.loc[rows, cols]`
- `rows` and `cols` can be:
  - A single label
  - A list of labels
  - A slice of labels
  - A boolean Series

`loc` examples

- `df.loc[['index1', 'index5'], ['col3', 'col1', 'col7']]` selects two rows and three columns with lists for each row and column selection
- `df.loc['index1':'index10', ['col3', 'col7']]` Uses slice notation for the rows and lists for the columns
- `df.loc['index1', 'col5]` selects a single scalar value

Column selection optional Column selection is optional and not present for the following examples. In these cases, all the columns are selected

- `df.loc['index1']` selects a single row as a Series
- `df.loc[['index1', 'index2']]` selects multiple rows with a list as a DataFrame
- `df.loc['index1':'index100':5]` - selects multiple rows with slice notation as a DataFrame

The`iloc` indexer

- Analogous to `loc` but selects by **integer location**
- Simultaneously selects rows and columns with `df.iloc[rows, cols]`
  - A single integer
  - A list of integers
  - A slice of integers

`iloc` examples

- `df.iloc[[2, 6], [0, 4, 2]]` selects two rows and three columns with lists for each row and column selection
- `df.iloc[5:10, [3, 7]]` Uses slice notation for the rows and lists for the columns
- `df.iloc[1, 5]` Uses a single integer for both row and column selection to select a single scalar value

Column selection optional Column selection is optional and not present for the following examples. In these cases, all the columns are selected

- `df.iloc[2]` selects a single row as a Series.
- `df.iloc[[2, 5]]` selects multiple rows as a DataFrame
- `df.iloc[10:200:5]` - selects multiple rows with slice notation as a DataFrame

1.3.11 Series Subset Selection

- Use `loc` and `iloc` to make your selection explicit
  - Just the brackets are possible but ambiguous
- Both indexers work analogously as the do with a DataFrame, but only take one selection as there are no columns.
- `s.loc['label1']` - select a single item in Series as a scalar
- `s.loc[['label1', 'label5']]` - use a list to select disjoint items
- `s.loc['start':'stop':step]` - use a slice to select from start to stop inclusive
- `s.iloc[integer1]` - select a single item with an integer in the Series as a scalar
- `s.iloc[[integer1, integer2]]` - use a list to select disjoint items
- `s.iloc[start:stop:step]` - use a slice to select from start to stop exclusive
- `s[label]` works for both integer and label location. It is ambiguous. Avoid if possible.

1.3.12 Boolean selection for DataFrames

- Boolean selection (a.k.a Boolean indexing) is the process of filtering data based on the actual values
- Examples of boolean selection
  - Find all the employees with salary over 100k
  - find all the flights leaving from Houston with a distance over 500 miles
- Two-step process
  1. Create a boolean Series, usually by using one of the comparison operators
     - All values compared to a single scalar - `filt = df['col1'] > 5`
     - Element-by-element comparison - `filt = df['col1'] > df['col2']`
  2. Pass boolean Series to just the brackets or `loc` indexers
     - `df[filt]`
- Boolean selection does not with `.iloc`

1.3.13 Multiple Condition Boolean Series

- Create multiple conditions by using `&`, `|` operators for the logical **and** and **or**.
- Example - select all rows where col1 is greater than 5 or col2 is less than -2
  - `filt1 = df['col1'] > 5`
  - `filt2 = df['col2'] < -2`
  - `filt = filt1 | filt2`
  - `df[filt]`
- Wrap each condition in parentheses when on the same line
  - `filt = (df['col1'] > 5) | (df['col2'] < -2)`
- Invert a Boolean Series with the ~ (not) operator

Simultaneous boolean selection and column selection

- Use `loc` to simultaneously boolean select the rows while also selecting particular columns
  - `df.loc[filt, [col1, col2]]`

Methods that create Boolean Series

- `isin` - multiple equality comparisons. `filt = df['col1'].isin([val1, val2, val3])`
- `between` - check whether between two values - `filt = df['col1'].between(50, 100)`
- `isna` - check whether each value is missing or not

Boolean Indexing on a Series

- Works similarly to boolean indexing for DataFrames

- Create the filter and pass it into just the brackets
  - `filt = ((s > 5) | (s < -2)) & (s % 2 == 0)`
  - `s[filt]`

1.3.14 `query` method

- More compact and readable way of filtering data by the values
- Pass a string to the **`query`** method
- The expression in the string must evaluate as True or False for every row

`query` method rules

- Column names may be accessed directly with their name
  - `df.query('col1 > 5')`
- Often you will use one of the comparison operators to create a condition
- Use chained comparison operators to shorten syntax
  - `df.query('20 > col1 > 5')`
- Use `and`, `or`, and `not` to create more complex conditions
  - `df.query('col1 > 5 or col2 < 10')`
- To use a literal string, surround it with quotes
  - `df.query('col3 == "Texas"')`
- Use `in` to test multiple equalities. Provide the test values in a list
  - `df.query('col3 in ["Texas", "Florida", "Alabama"]')`
- All arithmetic operators work just as they do outside of the string
  - `df.query('col1 ** 2 + col2 ** 2 == col3 ** 2')`
- Use the `@` character to reference a variable name
  - `df.query('col1 > @varname')`
- Reference the index with the string 'index' or the index's name
  - `df.query('index > 5')`
- Use backticks to reference a column name with spaces in it
  - `df.query('`column with spaces` > 5')`

1.3.15 Series operators

Series Arithmetic Operators

- `+, -, *, /, //, **, %` - all operate on every value of the Series
- `s + 5` adds 5 to every value in the Series
- Operations are **vectorized** - no explicit writing of a for-loop

Series Comparison Operators

- `<, >, <=, >=, ==, !=` - applies condition to each value in Series - returns boolean Series
- `s > 5` - compares every value in Series to 5 and returns a Series of booleans
- Comparison against missing values always return False
- Operations are vectorized

1.4 Essential Series Methods

Series Aggregation methods

Aggregation methods return a **single** value

- `s.sum`

- `s.min`
- `s.max`
- `s.median`
- `s.mean`
- `s.std`
- `s.var`
- `s.count` - counts non-missing values
- `s.quantile(q=.5)` - percentile of distribution
  - `s.quantile(q=[.1, .2, .5, .9, .99])` - Use a list to return multiple percentiles
- `s.describe` - returns most of the above aggregations in one Series

Non-Aggregation methods

Non-aggregating methods do not return a single value and return a Series the same length as the original.

- `s.abs` - takes absolute value
- `s.round` - rounds to the nearest given decimal
- `s.cummin` - cumulative minimum
- `s.cummax` - cumulative maximum
- `s.cumsum` - cumulative sum
- `s.diff` - difference between one element and another
- `s.pct_change` - percent change from one element to another

Missing Value methods

- `s.isna` - Returns a Series of booleans based on whether each value is missing or not. `isull` is an alias
- `s.notna` - Exact opposite of `isna`
- `s.fillna` - fills missing values in a variety of ways
  - `s.fillna(5)` - fills all missing values with 5
  - `s.fillna(method='bfill')` - fills missing values going backwards
  - `s.fillna(method='ffill')` - fills missing values going forwards
- `s.dropna` - Drops the missing values from the Series
- `s.interpolate` - fills in missing values with a function (linear, quadratic, etc…)

More Series methods

- `s.idxmin` - returns the index label of the minimum value
- `s.idxmax` - returns the index label of the maximum value
- `s.agg` - returns specific aggregations given as a list of strings as a Series
  - `s.agg(['min', 'max', 'idxmin', 'idxmax'])`
- `s.unique` - returns a numpy array of unique values in the order they appeared
- `s.nunique` - returns the number of unique values
- `s.drop_duplicates` - returns a Series of the first unique values by default
- `s.sort_values` - sorts from least to greatest by default. Use `ascending=False` to reverse
- `s.sort_index` - sorts Series by its index
- `s.rank` - return a ranking from 1 to n for each value in the Series
- `s.sample` - randomly samples Series
  - `s.sample(n=10)` - select 10 random items in the series without replacement
  - `s.sample(frac=.3, replace=True)` - select 30% of items in the series with replacement
- `s.replace` - replaces values in a Series `s.replace('DEPARTMENT', 'dept')` - replace entire value 'DEPARTMENT' with 'dept' `s.replace('DEPARTMENT', 'dept', regex=True)` - replace appearance of 'DEPARTMENT' anywhere in the string with 'dept'

Series `value_counts` - important method

- **`value_counts`** - returns the sorted frequency of each unique value in a Series. Use **`normalize=True`** to return proportions

String-only Series methods with the `str` accessor

- Only available to Series that have string data (object or string data type)
- Many methods overlap with the built-in str methods. Examples:
  - **`s.str.upper()`** - make all characters uppercase
  - **`s.str.lower()`** - make all characters lowercase
  - **`s.str.count('a')`** - count the occurrences of 'a' in the string. Return an integer Series
  - **`s.str.contains('dept')`** - test each string to see if it contains 'dept' - return a boolean Series
  - **`s.str.split('|')`** - split the string by given string. Return a Series containing lists
    - Use **`expand=True`** to return a DataFrame
  - **`s.str.extract('(pat)')`** - extract regex pattern from each string
- Learn regular expressions for more powerful searching and extracting

Datetime-only Series methods with the `dt` accessor

- Only available to Series with datetime data
- Mostly attributes. Examples:
  - Return datetime component as an integer Series - **`s.dt.year`**, **`s.dt.month`** (1-12), **`s.dt.day`** (1-31), **`s.dt.hour`**, **`s.dt.minute`**, etc..
  - **`s.dt.day_of_week`** - returns 0 (Monday) to 6 (Sunday)
  - **`s.dt.day_of_year`** - return 1 to 366
- Methods
  - Use offset aliases
  - **`s.dt.round('H')`** - round to nearest hour
  - **`s.dt.ceil('D')`** - return **up** to nearest day
  - **`s.dt.floor('S')`** - return **down** to nearest second
  - **`s.dt.to_period('M')`** - convert to period data type with frequency month

Period and Timedelta data types with the `dt` accessor

- Both the Period and Timedelta data types work with the `dt` accessor
- Very similar to the Datetime `dt` attributes and methods

1.4.1 DataFrame Operators

- Applies operation to every single value in every single column
- Operations are vectorized
- Error occurs if operation does not work with every single data type (i.e. adding 5 to a string column)

DataFrame Arithmetic Operators

- **`+, -, *, /, //, **, %`** - applies operation to all values of DataFrame
- **`df + 5`** adds 5 to every value in the DataFrame

DataFrame Comparison Operators

- **`<, >, <=, >=, ==, !=`** - applies comparison to each value in DataFrame - returns DataFrame of all booleans
- **`df > 5`** - compares every value in DataFrame to 5.

1.4.2 DataFrame vs Series Methods

- DataFrames and Series share the vast majority of their methods
- DataFrame methods usually have an `axis` parameter that controls the **direction of the operation**.
- The direction of the operation is either vertical or horizontal
- `axis` equal to `'index'` (or `0`) performs the operation vertically (i.e. summing all the values in each column).
- `axis` equal to `'columns'` (or `1`) performs the operation horizontally.
- The default is almost always `axis=0` - meaning Pandas **thinks column-wise**. By default, operations happen to each column independently.
- All the descriptive statistical methods are the same as they are with Series and by default operate down each column.

1.4.3 Specifics on certain DataFrame methods

- `sort_values` - must pass a string or list of strings to `by` parameter to sort columns. Pass `ascending` list of booleans that correspond with `by` columns to control direction of sort.
- `drop_duplicates` and `dropna` have a `subset` parameter. Pass columns to them to it to limit the functionality to just those columns.

1.4.4 Miscellaneous Methods

- `info` - prints out column names, data types, number of non-missing values
- `select_dtypes` selects all columns with the given type - Use strings 'int', 'float', 'bool', 'object', 'datetime', 'timedelta', 'category'. Use 'number' to select both int and float
  - `df.select_dtypes(['datetime', 'bool', 'int64'])`
  - `df.select_dtypes('number')`
- `copy` - copies the DataFrame
  - `df1 = df` does NOT copy, just creates a new variable name referencing the same DataFrame
  - `df1 == df.copy()` creates a completely new DataFrame copy in memory

1.4.5 DataFrame index/column handling methods

- `rename` - rename index/column names with a dictionary
  - `df.rename(columns={'col1': 'newcol1'}, index={'row1': 'newrow1})`
- `drop` - pass string or list of strings to `index/columns` parameter to drop index/columns
- `pop` - remove a column in-place and return it as a Series
  - **`some_col = df.pop('some_col')`

1.4.6 Adding new columns to a DataFrame

- `df['new_col'] = <some_expression>` - column gets added to the end
- `df.insert(3, 'new_col', values)` - inserts a column **in-place** at integer location 3 with name 'new_col' with given `values`

1.4.7 Unique to Series

- `str`, `dt`, `cat` accessors
- `unique`, `between`, `to_frame`

1.4.8 Unique to DataFrames

- `info`, `select_dtypes`, `set_index`, `query`, `stack`, `melt`
- `merge`, `join`, `insert`, `assign`, `pivot`, `pivot_table`

1.4.9 Convert Series

- `to_list` - values become new list
- `to_dict` - index used as dict keys, values used as dict values

1.5 Data Types

Numpy data types available to pandas

Generic Name	String Name of Default	Available Bit Sizes
Boolean	`bool`	8
Integer	`int64 or int32`	8, 16, 32, 64
Unsigned Integer	`uint64 or uint32`	8, 16, 32, 64
Float	`float64`	16, 32, 64

Missing values only available to floats as np.nan

Data types specific to pandas

Generic Name	String Name of Default	Pandas Object name of Default	Available Bit Sizes
Nullable Boolean	`boolean`	`pd.BooleanDtype()`	8
Nullable Integer	`Int64`	`pd.Int64Dtype()`	8, 16, 32, 64
Nullable Unsigned Integer	`UInt64`	`pd.UInt64Dtype()`	8, 16, 32, 64
Nullable Float	`Float64`	`pd.Float64Dtype()`	32, 64

Missing values available to all data types as pd.NA

Generic Name	String Name of Default	Available Bit Sizes	Notes
Object	`object`	any	May contain any Python object
String	`string`	any	Only contains strings
Category	`category`	Smallest integer capable of holding all categories	Use pd.CategoricalDtype() to create ordinal categories

Generic Name	String Name of Default	Available Bit Sizes	Notes
Datetime	`datetime64[u]`	64	A single moment in time
Timedelta	`timedelta64[u]`	64	An amount of time
Period	`period[u]`	64	A span of time

Note - Must specify **u**, the unit of date or time above. Datetime and timedelta columns will be in nanosecond precision.

1.5.1 Converting to different data types

Strings

- Use string name, such as `'int32'`, `'datetime64[ns]'`
- Pandas-specific data types use uppercase first letters such as `'Int32'`, `'Float64'`, except for `'boolean'`

Numpy or pandas objects directly

- `np.int32`
- `pd.Int64Dtype()` - for pandas specific data types, you must call them

1.5.2 Series

- Construct a Series with a specific data type
  - `pd.Series([4, 10, 8], dtype='uint8')`
  - `pd.Series([4, 10.3, 8], dtype=pd.Float64Dtype())`
- `astype` - convert a Series to specific data type
  - `s.astype('float64')`
- Force conversion with `pd.to_numeric(s, errors='coerce')`, where `s` is a Series of strings containing integers/floats. Any value unable to be converted will be replaced with `np.nan`

1.5.3 DataFrames

- Use a dictionary to map column name to data type
  - `pd.read_csv('data.csv', dtype={'col1': 'category', 'col2': 'Int64'})`
- Convert all columns with `astype`
  - `df.astype('int32')`
- Convert specific columns with a dictionary
  - `df.astype({'col1': 'category', 'col8': 'UInt32'})`

1.5.4 Object data type

- Only data type where you are NOT guaranteed to know the type of each value
- String data are read in as 'object' by default.
- Often good candidates to convert to categorical if the values are discreet, known, limited, and repeat
- Majority of the time columns with the 'object' data type are strings
- Can hold any python object
- Very flexible, but at the cost of not guaranteeing the data type

1.5.5 Categorical data type

- Nominal - no inherent ordering - color, gender, country, etc… - use category
  - string `'category'` or `pd.CategoricalDtype(list_of_categories)`
- Ordinal - inherent ordering - movie rating (very good, good, poor) - create ordered categorical
  - `pd.CategoricalDtype(list_of_categories, order=True)`
- Memory - each category is stored once in memory and mapped to an integer. This integer is what is stored for each value
- Performance - is often better using categorical data as opposed to object

1.5.6 `cat` accessor

- Available only to Series containing categorical data
  - `s.cat.categories` - list of unique categories

  - `s.cat.codes` - get array of underlying integer data
  - `s.cat.add_categories(list_of_new_categories)` - must do this first before adding new categories
  - `s.cat.remove_categories(list_of_categories_to_remove)`
  - `s.cat.remove_unused_categories()`

1.5.7 Date and time data types

- `datetime64[ns]` - a single moment in time, always with nanosecond precision
- `timedelta64[ns]` - an amount of time not connected to a specific date, nanosecond precision
- `Period[u]` - a period of time with a start and end, where `u` is the amount of time ('Y', 'M', 'D', 'S', etc..). pandas-only data type

1.5.8 Pandas-only data types

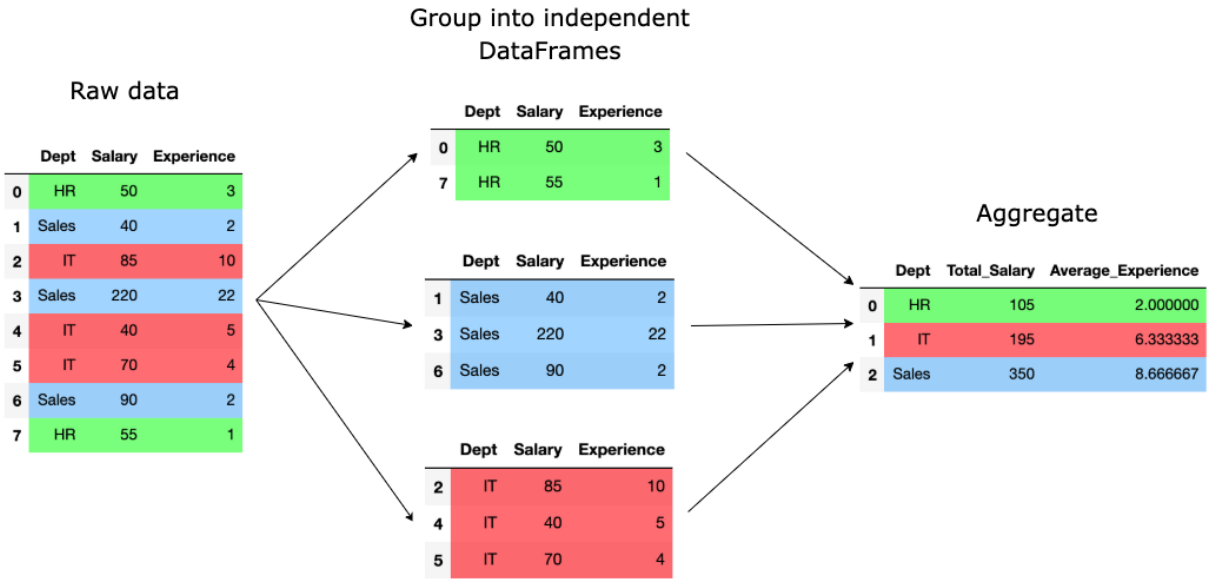
- Newer data types in pandas that are still labeled as 'experimental' - might want to hold off on using them in production
- Nullable integer, float, boolean
  - One missing value representation - `pd.NA`
- String - only contains strings. `str` accessor available

1.5.9 Less common data types

Only in pandas

- `SparseDtype` - for columns where vast majority of values are 0 and only a few non-zero
- `IntervalDtype` - each value is an interval, for example 3 to 6. Interval can be open or closed on either end.
- `DatetimeTZDtype` - timezone aware datetime

1.6 Grouping



- Apply operation to an independent grouping of the data
- Use `groupby` method
- Aggregation is the most common operation

1.6.1 Groupby Aggregation

Every `groupby` aggregation has three separate components:

- **Grouping column** - Every distinct value in this column forms its own group
- **Aggregating column** - Column being aggregated to a single value. Usually numeric.
- **Aggregating function** - Function applied to the aggregating column. ('mean', 'min', 'max', etc…)

Syntax - use method chaining

```
df.groupby('grouping column').agg(new_column=('aggregating column', 'aggregating function'))
```

- Provide **`agg`** method a keyword argument set to a tuple
  - keyword will be new column name
- Aggregating function can be a string
  - **`emp.groupby('dept').agg(avg_salary=('salary', 'mean'))`**
- Unique values of grouping column will be in **sorted** order in the index
  - Retain original ordering and get better performance with:
    - **`emp.groupby('dept', sort=False).agg(avg_salary=('salary', 'mean'))`**
  - Keep grouping columns as DataFrame columns and NOT in the index
    - **`emp.groupby('dept', as_index=False).agg(avg_salary=('salary', 'mean'))`**
    - **`emp.groupby('dept').agg(avg_salary=('salary', 'mean')).reset_index()`**
- Use a list to group multiple columns and multiple keyword arguments for multiple aggregations

```
sf_emp.groupby(['year', 'organization group']).agg(avg_salary=('salaries', 'mean'),
                  min_salary=('salaries', 'min'),
                  max_salary=('salaries', 'max'),
                  avg_overtime=('overtime', 'mean'))
```

Size of each group

Multiple ways to get size of each group * **`df.value_counts('col')`** * **`df.groupby('grouping column').size()`** * For the size of multi-column groups * **`df.value_counts(['col1', 'col2'])`**

1.6.2 Alternative `groupby` syntax

- When all aggregating columns use the same aggregating function(s)
  - **`df.groupby('grouping column')[list_of_aggregating_columns].agg(list_or_string_of_aggregat`**
- When all aggregating columns use exactly one aggregating function, can skip `agg` and use method directly
  - **`df.groupby('grouping column')[list_of_aggregating_columns].sum()`**
- Aggregate all the non-grouping columns with the same function
  - **`df.groupby('grouping column').sum()`**
  - non-numeric columns silently dropped
- Use a dictionary to map the grouping column to the aggregating function
  - **`df.groupby('grouping column').agg({'aggregating column': 'aggregating function'})`**

1.6.3 Pivot Table

- The **`pivot_table`** method is very similar to a **`groupby`** aggregation, but pivots one of the grouping columns. Uses the same terminology as groupby
- Pivot tables return **wide** data and are better for comparison and presentation
- **`groupby`** returns **long** data and is better for further analysis

Syntax

```
df.pivot_table(index='grouping column 1', columns='grouping column 2',
               values='aggregating column', aggfunc='sum')
```

1.6.4 Counting with the `pd.crosstab` function

- **`pd.crosstab`** counts occurrences of unique combinations of values in one (often two) or more columns
- Resulting DataFrame is known as a cross tabulation or contingency table
- Useful to normalize counts across more than one variable. Otherwise it is not needed and adds no value over `pivot_table`.
- Nearly identical to `df.pivot_table` but is a **function**, therefore you need to pass it Series and not strings.
- Set normalize to either **`'all'`**, **`'index'`**, or **`'columns'`**
- **`pd.crosstab(index=df['grouping column 1'], columns=df['grouping column 2'], normalize='index')`**

1.6.5 Custom aggregation function

- Needed when built-in pandas aggregation functions do not exist.
- Define a regular python function that accepts one argument
- Implicitly passed a Series of the aggregating column for that group
  - Function signature - **`def custom_agg(s)`**
    - Must return a single value
    - **`df.groupby('grouping column').agg(new_column=('aggregating column', custom_agg))`**
- Custom aggregation functions have poor performance. Avoid if possible.

1.6.6 Other groupby methods

Aggregation is the most common operation to perform on groups, but there are other possible methods that do not return a single value.

- **`filter`**
  - Syntax - `df.groupby('grouping col').filter(func)`
  - Use to filter out groups as a whole
  - Works similarly as boolean selection, but for the entire group
  - Must write a custom function that returns a single boolean for each group
  - Custom function is implicitly passed a DataFrame of the current group
  - A DataFrame with the same number of columns as the original is returned with rows that meet the filter condition
- **`transform`**
  - Syntax - `df.groupby('grouping col')['transformed col'].transform(func)`
  - Performs a calculation on each group
  - Can use aggregation function string names or custom functions
  - The function must return either a single value or a Series the same length as the group
  - If returning a single, that value is used for every row in the group
  - Returns a Series/DataFrame the same length of the group
  - The custom function accepts a pandas Series of all the values in the group

1.6.7 SeriesGroupBy vs DataFrameGroupBy methods

SeriesGroupBy and DataFrameGroupBy objects are very similar and are created after a call to the groupby method

- SeriesGroupBy - Perform an operation on exactly one column within each group
  - `g_s = df.groupby('grouping col')['other col']`
- DataFrameGroupBy - Perform an an operation on all columns within each group
  - `g_df = df.groupby('grouping col')`
- It's rare to assign these groupby objects to separate variable names. More often you'll chain one of their methods in the same line.
- `df.groupby('grouping col').head(n)` - Return the first `n` rows of each group
- `df.groupby('grouping col').tail(n)` - Return the last `n` rows of each group
- `df.groupby('grouping col').nth([i, j, k])` - Return rows with integer location `i`, `j`, and `k` of each group
- `df.groupby('grouping col').cumcount()` - returns the integer location for each row in the group
- `df.groupby('grouping col')['other col'].rank()` - ranks the values in `other col` beginning at 1
- `df.groupby('grouping col')['other col'].fillna(method='ffill')` - fills in missing values within each group using the last previous known value
- Many other normal DataFrame/Series methods are available and work similarly, but independently on the current group

1.6.8 Binning numeric columns

Use `pd.cut` and `pd.qcut` functions to bin numeric columns.

`pd.cut`

- Provide bins as either
  - a list of numeric boundaries
    - `pd.cut(df['col'], bins=[0, 5, 20, 100, 500])`
    - Creates n-1 bins with given boundaries
  - a single integer
    - `pd.cut(df['col'], bins=10)`
    - Creates 10 bins of equal width
- Bins are intervals in the form of `(left, right]` where the left boundary is exclusive and right is inclusive
- Set the number of decimal points with `precision`
  - `pd.cut(df['col'], bins=8, precision=0)`
- Label the bins with strings with `labels` and return the bins with `retbins`
  - `pd.cut(df['col'], bins=3, labels=['small', 'medium', 'large'], retbins=True)`

`pd.qcut`

- Provide bins as either
  - a list of quantiles (floats between 0 and 1)
    - `pd.qcut(df['col'], [0, .2, .7, 1])`
  - a single integer
    - `pd.qcut(df['col'], 8)`
    - Attempts to create bins with the same number of observation in each

1.7 Time Series

1.7.1 Definitions

- **Date** - only year, month, day - January 5, 2016
- **Time** - hour, minute, second, part of second - 5 hours, 45 minutes, and 6.74234 seconds

- **Datetime** - a date and a time combined. Has components year, month, day, hour, minute, second, part of second - January 5, 2016 at 5:45 p.m. and 6.74234 seconds

1.7.2 Datetimes vs Timestamps pandas

- The Python standard library has its own datetime object, but pandas has its own more powerful version
- The formal name of the object that pandas uses is `Timestamp`. This is confusing, but it is just a different name that means the same thing as a datetime. Technically, all of the datetimes created in pandas are `Timestamp` objects. The words datetime and timestamp are used interchangeably and mean the same thing.

Creating a single scalar datetime object

- `pd.Timestamp` - a constructor that always creates a single datetime (Timestamp) object
- `pd.to_datetime` - a function that can create a single datetime or a sequence of many datetimes.

Converting strings to datetimes

The most common way to create a datetime is with a string. Both `pd.Timestamp` and `pd.to_datetime` produce identical results and can be used interchangeably for single input strings. A wide variety of strings formats are understood:

- `pd.Timestamp('2020-12-3')`
- `pd.Timestamp('12/3/2020')`
- `pd.Timestamp('12-3-2020')`
- `pd.Timestamp('12-3-2020')`
- `pd.Timestamp('December 3, 2020')`
- `pd.Timestamp('Dec 3, 2020')`
- `pd.Timestamp('20201203')`
- `pd.Timestamp('December 3, 2020 12:45')`
- `pd.Timestamp('December 3, 2020 12:45:51.8534')`
- `pd.Timestamp('December 3, 2020 02:45:51.8534 PM')`

Every timestamp will have values for all components, regardless if they are given. The default value for all time components is 0. There will be a value for hours, minutes, seconds, and parts of a second. All timestamps have **nanosecond** precision. This level of precision is fixed and unable to be changed.

1.7.3 Day first

Only available to `pd.to_datetime` and not `pd.Timestamp`

- `pd.to_datetime("9/5/2020")` is September 9, 2020 by default
- `pd.to_datetime("9/5/2020", dayfirst=True)` is May 9, 2020

ISO-8601

International Organization for Standards has specific format for datetimes - `YYYY-MM-DDTHH:MM:SS`. Note the `T` separating the date and time. Parts of second are also allowed as decimals.

Unix epoch

- January 1, 1970 at midnight

Converting integers to datetimes

Pass an positive or negative integer to `pd.to_datetime` or `pd.Timestamp` along with the `unit` to get a timestamp relative to the Unix epoch..

- **Units** - **'d'** - days, **'h'** - hours, **'m'** - minutes, **'s'** - seconds, **'ms'** -milliseconds, **'us'** - microseconds, and **'ns'** - nanoseconds

Examples:
* `pd.Timestamp(100, unit='s')` - January 1, 1970 at 12:01 am and 40 seconds * `pd.Timestamp(10_000, unit='d')` - 10,00 days after epoch - May 19, 1995

1.7.4 Custom datetime strings

Convert a non-standard string to a datetime with these format codes. A partial list is given below. See the official Python documentation for full details.

Format code

Definition

Examples

%d

zero-padded day of month

- 01, 02, … 30,31
  <tr> <td>%b/%B</td> <td>abbreviated/full month name</td> <td>Jan/January, Feb/February</td>
  <tr> <td>%m</td> <td>zero-padded month number</td> <td>01, 02</td></tr>
  <tr> <td>%y/%Y</td> <td>two-digit/four-digit year</td> <td>05/2005, 10/2010</td></tr>
  <tr> <td>%H</td> <td>zero-padded 24 hour clock</td> <td>00, 01, 23</td></tr>
  <tr> <td>%I</td> <td>zero-padded 12 hour clock</td> <td>01, 02, 12</td></tr>
  <tr> <td>%M</td> <td>zero-padded minute</td> <td>00, 01, 59</td></tr>
  <tr> <td>%S</td> <td>zero-padded second</td> <td>00, 01, 59</td></tr>
  <tr> <td>%p</td> <td>AM or PM</td> <td>am, pm, AM, PM</td></tr>
- `pd.to_datetime('The 5th of January, 2020 at 5:45 pm', format='The %dth of %B, %Y at %I:%M %p')`

1.7.5 Timedeltas

An amount of time independent of a date. e.g. 5 hours 36 minutes and 10 seconds

Creating timedeltas from strings

Use the `pd.to_timedelta` function or the `pd.Timedelta` constructor

- `pd.Timedelta('5:36:22.5123')`
- `pd.Timedelta('8 days 5:36:22.5123')`

Converting integers to timedeltas

By default, numerical input to `pd.to_timedelta`/`pd.Timedelta` are treated as nanoseconds. Use the `unit` parameter to set the units.

- `pd.to_timedelta(500)` - 500 nanoseconds
- `pd.to_timedelta(510.34, unit='d')` - days

Timedelta math

```
ts1 = pd.to_datetime('2020-05-18 18:04:55')
ts2 = pd.to_datetime('2021-04-12 04:36:22')
td1 = pd.to_datetime('4 days 3:23:23')
td2 = pd.to_datetime('3 days 2:12:18')
```

- Create a timedelta by subtracting two datetimes `ts1 - ts2`
- Add two timedeltas `td1 + td2`
- Subtract two timedeltas `td1 - td2`
- Multiply a timedelta by a number `td1 * 5.7`
- Divide two timedeltas to get a number `td1 / td2`

1.7.6 Periods

A span of time with a start and end datetimes. Span can be any length.

1.7.7 Selecting rows at specific frequencies

DataFrame must have a DateTimeIndex. Pass the `asfreq` method an offset alias. Data must be ordered without duplicate dates.

Alias	Description	Alias	Description
`Y/A`	year end	`D`	day
`YS/AS`	year start	`H`	hourly
`Q`	quarter end	`T` or `min`	minutes
`QS`	quarter start	`S`	seconds
`M`	month end	`L` or `ms`	milliseconds
`MS`	month start	`U` or `us`	microseconds
`W`	weekly	`N`	nanoseconds

- `df.asfreq('M')` - select the last day of each month

Anchored offset

The year, quarter, and week offset aliases are all **anchored** to either a month or day of week. Year is anchored to Jan/Dec, quarter to Jan/Mar and week to Sunday. Use a hyphen and then the three character month or weekday name abbreviation to change the anchoring.

- `df.asfreq('A-Sep')` - Select the last day of September each year
- `df.asfreq('QS-Feb')` - Select the last day of September each year
- `df.asfreq('W-Tue')` - Select each Tuesday

Business offset aliases

Prepend the letter `B` the offset alias to select only business days (Mon-Fri)

- `df.asfreq('BMS')` - Select the first business day of each month
- `df.asfreq('BQ-Apr')` - Select the last business day of each quarter where the ending quarters are Jan, Apr, Jul, Oct

Integers in offset aliases

Prepend an integer to the offset alias to select multiples of that frequency

- `df.asfreq('3W-Fri')` - Select every third Friday
- `df.asfreq('2BMS')` - Select the first business day of every other month

Downsampling

Selecting less rows than there are originally. E.g. original data is daily, you select every Friday

Upsampling

Selecting more rows than there are originally. Useful when original data is not evenly spaced time intervals and you want to insert additional rows. These rows will be empty and need to be filled.

Selecting multiple rows at specific frequencies

Use the attributes of the DatetimeIndex to create a boolean filter

- `df[df.index.day == 5]` - Select the 5th day of each month for all years
- `df[df.index.month == 9]` - Select all rows in the month of September for each year
- `df[df.index.isocalendar()['week'] == 11]` - Select all rows in the 11th week for each year
- `df[df.index.month.isin([4, 6)]` - Select all rows in the months of April and June for each year

1.7.8 Grouping by time

Group together entire time periods and perform on operation on all independent groups.

`resample` method

Very similar to `groupby`. Pass it an offset alias. Must chain a method after it

- `df.resample('W-Fri').mean()` - group every week together using Friday as the end date. Take the mean of each column for every group
- `df.resample('M').agg({'column a': 'agg_func1', 'column b': 'agg_func2'})` - group every month applying different aggregations to different columns
- `df.resample('Y').agg(['min', 'mean', 'sum'])` - apply multiple aggregations to all columns
- `df.resample('2MS').mean()` - Group by consecutive time periods. More intuitive to use start time periods `MS` or `AS`.
- `df.resample('3AS-Oct').mean()` - Group by 3 years where year is Oct 1 to Sep 30.

1.7.9 Grouping by time with the `groupby` method

Do all the same grouping as `resample`, but set `freq` parameter of `pd.Grouper` to offset alias and pass this object to `groupby`. All normal `groupby` methods are available

- `df.groupby(pd.Grouper(freq='M')).agg(new_name=('column', 'agg func'))`

1.7.10 `resample` and `groupby` with a Series

- Both `resample` and `groupby` work the same for Series as they do for DataFrames

1.7.11 Rolling windows

Use a sliding window of rows as a group with the `rolling` method

Integer window sizes

- `df.rolling(5).mean()` - Find the mean of each column of the 5 last observations
- `df.rolling(5, min_periods=1).mean()` - Set `min_periods` to allow for calculations to be performed on incomplete window sizes (those at start or end of data). Otherwise they will be missing.
- `df.rolling(5).agg(['min', 'max', 'count'])` - return multiple aggregations by passing a list to the `agg` method

Offset alias window size

Offset alias window sizes will use all rows within the time period provided. Cannot use `Y`, `Q`, `M`, `W`

- **`df.rolling('5D').mean()`** - Groups together all rows within the last 5 days

Center the window

- **`df.rolling(5, center=True).mean()`** - center the window around current row. Equal number before and after.

1.7.12 Grouping by time and another column

Can group by both time and unique values of another column.

Group together

- **`tg = pd.Grouper(freq='Y')`** - create time group
- **`df.groupby(['non-datetime column', tg]).mean()`**
- **`df.pivot_table(index=tg, columns='non-datetime column', values='agg column', aggfunc='agg func')`**

Group independently

Group by the non-datetime column first. Resulting time periods might not align if first value in each group is from a different time period.

- **`df.groupby('non-datetime column').resample('Y').mean()`**

Rolling windows within a group

- **`df.groupby('non-datetime column').rolling('30D').mean()`**

1.7.13 Shifting the time series index

Pass the shift method an integer and an offset alias to shift the values in the index up/down. This does NOT shift the data, just changes the index.

- **`df.shift(3, 'D')`** - add three days to each index value
- **`df.shift(-2, 'D')`** - subtract two days to each index value
- **`df.shift(6, 'H')`** - add 6 hours to each index value

Caution when shifting by week, month, quarter, or year

When shifting by week, month, quarter, or year, index values are **rounded** up/down to the next start/end of the time period. The current start/last value of a time period may be moved to one more time period than the others.

- **`df.shift(1, 'M')`** - round each index up to the last day of the current month. The current last day of the month will be shifted up to the following month (confusing).
- **`df.shift(2, 'W-Fri')`** - round each index up to the second Friday from the current day.

1.7.14 PeriodIndex

Use **`to_period`** method to convert DatetimeIndex to a PeriodIndex. More intuitive than using **`shift`** method

- **`df.to_period('M')`** - rounds each index value to current month
- **`df.to_period('M').to_timestamp()`** - convert index back to datetime

1.7.15 Creating datetime, timedelta, and period ranges

Use `pd.date_range` function which always returns a DatetimeIndex. Must use exactly 2 of 3 parameters `start`, `end`, `periods`. Set `freq` to offset alias to choose interval. Default is 1 day.

- `pd.date_range('2021-6-10', '2021-6-16')` - all days between given dates
- `pd.date_range('2021-6-10', '2021-6-16', freq='4H')` - Every 4 hour interval between given dates
- `pd.date_range(end='2021-6-16', freq='4H', periods=100)` - 100 timestamps separated by 4 hour intervals before end date
- `pd.timedelta_range('8:00:00', '17:30:00', freq='30min')` - 30 minute intervals between 8 hours and 17.5 hours.
- `pd.period_range('2010-1-1', periods=20, freq='M')` - 20 months beginning at Jan, 2010.

1.8 Reading in Data

Read in data as a pandas DataFrame using a **function**. They all begin with `pd.read_`

1.8.1 CSVs

Comma separated values. Plain-text file with values separated by a **delimitter** (comma, tab, space, etc…). Common file type

- `pd.read_csv('file_location')`
  - File location can be remote - `pd.read_csv('http://github.com/ted/file.csv')`
- Dozens of parameters, some useful ones below
  - `sep` - string of delimiter - `,` (default), `\t`, `;`
  - `header` - int location of row where column names are
  - `index_col` - int location or label of column to use as index
  - `usecols` - list of int location or labels of subset of columns to read
  - `dtype` - dictionary mapping column name to data type
  - `skiprows` - int or list of ints of rows to skip
  - `nrows` - int of number of rows to read from the top of the file
  - `na_values` - string, list of strings, or dict (map column name to string) of non-default missing values
  - `parse_dates` - list of columns to convert to datetime columns

[ ]: