Pandas

What is Pandas?

- Opensource data analysis library
- Main objects:
  - 1-dimensional `Series`
  - 2-dimensional `DataFrame`
- Built on NumPy

What is Pandas?

- Optimized for wide variety of data analysis operations
  - I/O to/from formatted files and data bases
  - Missing data handling
  - Slicing, indexing, reshaping, adding columns
  - Powerful grouping for aggregating and transforming data sets
  - Merging and joining data sets
  - Time-series functionality
- Applied in finance, neuroscience, economics, statistics, advertising, web analytics, and more.
- Virtually no modeling capabilities (easy to integrate with scikit-learn)

Series

Series

- One-dimensional array
- Possibly heterogeneous type (although usually not)
- Each element has a label referred to as index
- Missing values are represented as NaN
- May be MultiIndexed hierarchically

Series constructors

- `pd.Series(ndarray, index=None)` — series from array-like in same order
  - `ndarray` must be 1-dimensional
  - If `index` is provided, must be same length as `ndarray`
  - If `index` is not provided, will be `0, ..., len(ndarray) – 1`
- `pd.Series(dict, index=None)` — series from dictionary
  - If `index` is provided, it gives the order over `dict`
  - If `index` contains keys not in `dict`, treated as missing value
  - If `index` does not contain some key in `dict`, it is discarded
  - If `index` is not provided, order will be insertion order into `dict` (for Python >= 3.6)
- `pd.Series(scalar, index)` — repeated scalar value
  - `index` is required

Indexing and slicing Series

- Indices must be hashable types
- Index labels may not be unique, although that will raise errors in certain functions that require uniqueness
- Series objects can be indexed by either their index labels or their underlying 0-based index
- Slicing can also be done by either index type
  - Slicing based on index labels is done based on the order of the Series

Series miscellaneous

- Series is array-like: valid argument for most NumPy functions
  - Array functions are modified to ignore missing values (NaN)
- Series is dict-like: get and set values by index label
- Series can be treated as arrays for vectorized operations
- Indices are automatically aligned: operating on two Series with different indices gives a Series with the *union* of the indices, where non-common indices are given NaN values

DataFrames

DataFrame

- 2-dimensional labeled structure
- Possibly heterogeneous type (common across columns)
- Intuition: spreadsheet or SQL table
  - Each column is an attribute
  - Each row is a record
- Also: like a dictionary of Series objects

DataFrame constructors

- `pd.DataFrame(dict, index=None, columns=None)` — dict of Series or dicts
  - Keys from outer dict are columns, keys from inner dict are indices
  - If the keys in the outer dict are tuples, columns are MultiIndexed
  - `index` and `columns` treated like `index` for creating a Series from a dict
    - Dict key missing from index/columns: discarded
    - Order from index/columns
    - Index/columns missing from dict: treated as empty

DataFrame constructors

- `pd.DataFrame(dict, index=None, columns=None)` — dict of array-like
  - All arrays in dict must be the same length
  - If `index` is present, must be the same length as arrays
  - `columns` is treated same as before
- `pd.DataFrame(list, index=None, columns=None)` — list of dicts
  - Each dict is treated as a row
  - Column names are the union of the keys in all the dicts

Accessing DataFrame columns

- DataFrames can be indexed like dicts for accessing, adding, and deleting columns
- Adding can be done with Series, array-like, or scalar
  - `df[col] = Series` — Series with indices not in the DataFrame get those indices removed
  - `df[col] = ndarray` — Array-like must have the same length as the indices in the DataFrame
  - `df[col] = scalar` — Scalars are propagated to fill all indices
- Columns are added at the end
  - Use `insert()` to specify a different location

Indexing DataFrame

- `df[col]` — return a Series/DataFrame corresponding to the column(s) with key `col`
- `df.loc[idx, col]` — return a Series/DataFrame corresponding to the row(s) with index label(s) `idx` and column label(s) `col`
- `df.iloc[n_idx, n_col]` — return a Series/DataFrame corresponding to the row(s) with 0-based index(es) `n_idx` and column 0-based index(es) `n_col`
- `df[slice]` — return a DataFrame with all columns and rows sliced by `slice`
  - Slicing is like with Series, can be 0-indexed or label-indexed
- `df.at[idx, col]`, `df.iat[idx, col]` — optimized versions of `loc`, `iloc` for accessing a scalar

DataFrame miscellaneous

- DataFrames are aligned automatically both on columns and rows
  - Operations on misaligned DataFrames result in the union of columns and rows
- Series are broadcasted row-wise when operating with DataFrames
  - Exception: if the index is a date stamp, broadcasting is column-wise
- Scalar operations are elementwise
- DataFrames can be transposed with `df.T`
- NumPy functions can operate on DataFrames with numeric types

Pandas I/O

Format Type	Data Description	Reader	Writer
text	CSV	read_csv	to_csv
text	JSON	read_json	to_json
text	HTML	read_html	to_html
text	Local clipboard	read_clipboard	to_clipboard
binary	MS Excel	read_excel	to_excel
binary	OpenDocument	read_excel	
binary	HDF5 Format	read_hdf	to_hdf
binary	Feather Format	read_feather	to_feather
binary	Parquet Format	read_parquet	to_parquet
binary	Msgpack	read_msgpack	to_msgpack
binary	Stata	read_stata	to_stata
binary	SAS	read_sas	
binary	Python Pickle Format	read_pickle	to_pickle
SQL	SQL	read_sql	to_sql
SQL	Google Big Query	read_gbq	to_gbq

Combining DataFrames

- `pd.concat(list)` — concatenate list (or iterable) of DataFrames/Series
  - `axis=0`: 0 concatenate rows, 1 columns
  - `join='outer'`: 'outer' union over index, 'inner' intersection
  - `ignore_index=False`: whether to drop the index of concatenation axis. Useful if indices aren't meaningful but may be repeated
  - `keys=None`: if present, create MultiIndex with keys at the outermost level (must be the length of list)

Combining DataFrames

- `pd.merge(left, right)` — implement SQL join operations on columns or indices
  - `how='inner'`
    - `'inner'`: SQL inner join, intersection of keys. Preserve order of `left` keys
    - `'outer'`: SQL outer join, union of keys. Sort keys lexicographically
    - `'left'`: SQL left outer join, only keys from `left`. Preserve order of `left` keys
    - `'right'`: SQL right outer join, only keys from `right`. Preserve order of `right` keys
  - `on=None`: which key to join on. If `None`, intersection of columns
  - `left_on/right_on`=None: which key from `left/right` to join on
  - `left_index/right_index`=False: use index from `left/right` as the join key