# Data Analysis with PANDAS

## CHEAT SHEET

* Created By: Arianne Colton and Sean Chen

## Data Structures

### Series (1D)

One-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its **index**. If index of data is not specified, then a default one consisting of the integers 0 through N-1 is created.

**Create Series**

- `series1 = pd.Series([(1, 2), index = [('a', 'b')])`
- `series2 = pd.Series(dict1)*`

**Get Columns and Row Names**

- `df1.columns`
- `df1.index`

**Get Name Attribute**

- `df1.column.name`
- `df1.index.name`

**Get Values**

- `df1.values`

**Get Columns as Series**

- `df1['state'] or df1.state`

**Get Row as Series**

- `df1.ix[('row2')] or df1[i1]`

**Assign a column that doesn’t exist will create a new column**

- `df1['eastern'] = 'Ohio'`

**Delete a column**

- `del df1['eastern']`

**Switch Columns and Rows**

- `df1.T`

**Dicts of Series are treated the same as Nested dict of dicts.**

**Data returned is a ‘view’ on the underlying data, NOT a copy. Thus, any in-place modifications to the data will be reflected in df1.**

### DataFrame (2D)

Tabular data structure with ordered collections of columns, each of which can be different value type. DataFrame (DF) can be thought of as a dict of Series.

**Create DF**

- `df1 = pd.DataFrame(dict1)`
- `df1 = pd.DataFrame({'item': 'value'} for idx in range(10))`

**Get Columns and Row Values**

- `df1.columns`
- `df1.values`

**Get Columns by Index**

- `df1['a']`
- `df1[['b','a']]]`

**Get Series Index**

- `df1.index`
- `df1['state'].name`
- `df1.index.name`

**Get Name Attribute**

- `df1.column.name`
- `df1['state'].name`

**Common Index Values are Added**

- `series1 + series2`

**Unique But Unsorted**

- `series2 = series1.unique()`

* Can think of Series as a fixed-length, ordered dict. Series can be substituted into many functions that expect a dict.

* Auto-align differently-indexed data in arithmetic operations.

## Dataframe (3D)

Tabular data structure with ordered collections of columns, each of which can be different value type. The inner keys as (from nested dict of dicts)

**Create Panel Data**

- `(Each item in the Panel is a DF)`

**PANEL DATA (3D)**

* Can be used to store hierarchical index (i.e. index, MultiIndex)

**Swap and Sort**

- `swaplevel('key1', 'key2')`
- `sort_level(0)`

**Summary Statistics by Axis**

- `df1.sum(axis = 0)`
- `df1.mean(axis = 0)`

**Filtering out Missing Data**

- `df1.isnull()`
- `df1.dropna(inplace = True)`

**Missing Data**

- `df1.dropna(thresh = 3)`
- `df1.fillna({'item': 'value'})`
- `df1.fillna(0)`

**Filling in Missing Data**

- `df1.interpolate()`
- `df1.fillna(method = 'ffill', limit = 2)`

* The order of the rows do not change. Only the two levels got swapped.

**Data selection performance is much better if the index is sorted starting with the outermost level, as a result of calling sort_index() or sort_level().**

**Swaping and Sorting Levels**

- `swaps = df1.swapaxes('item', 'minor') *`**
- `df1.sort_level()`

* "reset_index" does the opposite of "set_index", the hierarchical index are moved into columns.

**Under the hood, the functionality provided here utilizes panda’s “groupby”**

**DataFrame’s Columns as Indexes**

DF’s "set_index" will create a new DF using one or more of its columns as the index.

**Hierarchical Indexing**

Multiple index levels on an axis: A way to work with higher dimensional data in a lower dimensional form.

**MultiIndex: series1**

- `series1 = Series(rp.random.randn(6), index = [("a", 'a'), 'a', 'b', 'b', 'b', 'b'], [1, 2, 3, 1, 2, 3])`
- `series1.index.names = ["key1", "key2"]`

**Series Partial Indexing**

- `series1['b']` # Outer Level
- `series1[[1, 2]]` # Inner Level

**DF Partial Indexing**

- `df1["OutterCol3", 'InnerCol2']`
- `df1["OutterCol3", 'InnerCol2']`

* Most stats functions in DF or Series have a “level” option that you can specify the level you want on an axis.

* Under the hood, the functionality provided here utilizes panda’s “groupby”

**DF's ‘set_index’ will create a new DF using one or more of its columns as the index.**

**Summary Statistics by Level**

- `df1.sum(level = 'key2')`
- `df1.mean(level = 'col3', axis = 1)`
**INDEXING (SLICING/SUBSETTING)**

† Same as ‘NdArray’. In INDEXING ‘view’ of the source array is returned.

† Endpoint is inclusive in pandas slicing with labels: series[‘a’:’c’] where Python slicing is NOT. Note that pandas non-label (i.e. integer) slicing is still non-inclusive.

- **Index by Column(s)**
  - df1[‘col1’]
  - df1[‘col1’, ‘col3’]

- **Index by Row(s)**
  - df1.ix[‘row1’]
  - df1.ix[‘row1’, ‘row3’]

- **Index by Both Column(s) and Row(s)**
  - df1.ix[‘row2’, ‘row1’, ‘col1’]

- **Boolean Indexing**
  - df1[True, False]
  - df1[‘col2’ > 6] * # returns df of that has col2 value > 6

**SORTING AND RANKING**

- **Sort Index/Column**
  - sort_index() returns a new, sorted object. Default is “ascending = True”.
  - Row index are sorted by default, “axis = 1” is used for sorting column.

- **Sorting Index/Column means sort the row/ column labels, not sorting the data.**

- **DROP Operation returns a new object (i.e. DF):**
  - df1.drop(‘row1’)  
  - df1.drop(‘row1’, ‘row3’)  

- **REINDEXING**

  Create a new object with rearranging data conform to a new index, introducing missing values if any index values were not already present.

  - Change df1 Date Index Values to the New Index Values
    - date_index = pd.date_range(‘2010-12-31’, periods = 10, freq = ‘D’)
    - df1.reindex(date_index)
  
  - Replace Missing Values (NaN) with 0
    - df1.reindex(date_index, fill_value = 0)

  - ReIndex Columns
    - df1.reindex(columns = [‘a’, ‘b’])
  
  - ReIndex Both Rows and Columns
    - df1.reindex(index = […], columns = […])

  - Succinct ReIndex
    - df1.ix[[…], […]]
COMBINING AND MERGING DATA

1. pd.merge() aka database "join": connects rows in DF based on one or more keys.
   - Merge via Column (Common)
     df3 = pd.merge(df1, df2, on = ['col2'] *
     # INNER join is default or use option: how = 'outer'/left/right'
     # the indexes of df1 and df2 are discarded in df3
   - Merge via Row (Uncommon)
     df3 = pd.merge(df1, df2, left_index = True, right_index = True)
     # Use indexes as merge key : aka rows with same index value are joined together.

2. pd.concat() : glues or stacks objects along an axis (default is along 'rows : axis = 0').
   # Default delimiter is comma.
   df3 = pd.concat([df1, df2], ignore_index = True)
   # ignore_index = True : Discard indexes in df3
   # If df1 has 2 rows, df2 has 3 rows, then df3 has 5 rows

3. combine_first() : combine data with overlap, patching missing values.
   df3 = df1.combine_first(df2)

   - df1 and df2 indexes overlap in full or part. If a row NOT exist in df1 but in df2, it will be in df3. If row of df1 and row of df2 have the same index value, but row1's col3 value is NA, df3 get this row with the col3 data from df2

RESHAPING AND PIVOTING

1. Reshaping with Hierarchical Indexing
   series1 = df1.stack()
   # Rotates (innermost level) columns to rows as innermost index level, resulted in Series with hierarchical index.
   df1 = series1.unstack()  
   # Rotates (innermost level) rows to columns as innermost column level.
   df1 = series1.unstack(level = 0)

   - Use ALL overlapping columns names as the keys to merge. Good practice is to specify the keys: on = [{‘col2’, ‘col3’}]. If different key name in df1 and df2, use option: on = [{‘col2’, ‘col3’}]

   - Use ALL overlapping columns names as the keys to merge. Good practice is to specify the keys: on = [{‘col2’, ‘col3’}]. If different key name in df1 and df2, use option: on = [{‘col2’, ‘col3’}]

2. Pivoting
   - Common format of storing multiple “time series” in databases and CSV is :
     Long/Stacked Format : ['date', 'stock_name', 'price']
     Short/Pivoted Format : [date, stock_name, price]
   - However, a DF with these 3 columns data like above will be difficult to work with. Thus, “wide” format is preferred: 'date' as row index, 'stock_name' as columns, 'price' as DF data values.

   - Example pivotedDF2 :
     df2 = df3.replace([np.nan, -1], 100)

   - Another useful function : df2.replace(5, np.nan)
     Return this column as Nan

3. Removing Duplicate Rows
   series1 = df1.duplicated() # Boolean series1 indicating whether each row is a duplicate or not.
   df2 = df1.drop_duplicates() # Duplicates has been dropped in df2.

   - pd.merge()
   - pd.merge()
   - pd.merge()

4. Renaming Axis Indexes
   df1.index to Upper Case
   df1.rename(index = {str.upper : 'newRow'}, columns = str.upper)

5. Discretization and Binning
   - Continuous data can often be discretized into "bins" for analysis.
   - Divide Data Into 2 Bins of Number [18 - 26, 26 - 35]
   - cat = pd.cut(array, bins, labels=[...])

   - Compute equal-length bins based on min and max values in array!
   - cat = pd.cut(array, numofBins) # Bins the data based on sample quantities - roughly equal-size bins

6. Detecting and Filtering Outliers
   - any() test along an axis if any element is "True". Default is test along column (axis = 0).
   - df1[(np.abs(df1) > 3).any(axis = 1)]

   - Select all rows having a value > 3 or < -3.
   - Another useful function : np.sign() returns 0 or -1.

7. Permutation and Random Sampling
   randomorder = np.random.permutation(df1.shape[0])
   df2 = df1.take(randomorder)

8. Computing Indicator/Dummy Variables
   - Creating Indicator variables from categorical data used in regression analysis.
   - If a column in DF has “K” distinct values, derive a indicator DF containing K columns of 0s and 1s. 1 means exist, 0 means NOT exist.

得到数据

1. pd.read_csv(path) : read a file in CSV format.
2. pd.read_excel(path) : read Excel file.
3. pd.read_json(path) : read JSON object.

JSON (JAVA SCRIPT OBJECT NOTATION) DATA

One of the standard formats for sending data by HTTP request between web browsers and other applications.
It is much more flexible data format than tabular text from like CSV.

1. Convert JSON string to Python form
   data = json.loads(data)

2. Convert Python object to JSON
   jsonString = json.dumps(data)

3. Create DF from JSON
   df1 = pd.DataFrame(data['name'], columns = ['field1'])

XML AND HTML DATA

1. xml.to_csv(filepath/sys.stdout, sep = ','

   - The indexes of df1 and df2 are discarded in df3
   - pd.read_csv Hai/ URL/file-like-object, sep = ',', header = None)
   # Type-Inference: do NOT have to specify which columns are numeric, integer, boolean or string.
   # If Pandas, missing data in the source data is usually empty string.
   # Type-Inference: do NOT have to specify which columns are numeric, integer, boolean or string.
   # In order to handle the missing data, we need to specify missing values via option i.e.: na_values = ['NULL']
   # Default delimiter is comma.
   df1 = pd.read_csv Hai/ URL/file-like-object, sep = ',', names = ['..]

   # Explicitly specify column header, also imply first row is data
   df1 = pd.read_csv Hai/ URL/file-like-object, sep = ',', names = ['..]
   # Want 'date' columns to be row index of the returned DF

   df1.to_csv(filepath/sys.stdout, sep = ',')

   - Convert JSON to Python form
   data = json.load(data)

   - Returns index Labels Where Min/Max Values are Attained
   df1 = df1['col1'].value_counts()

   - Multiple Summary Statistics (i.e. count, mean, std)
   On Non-Numeric Data, Alternate Statistics (i.e. count, unique)
   df1.describe()

   - Correlation and Covariance
   df1.corr()
**TIME SERIES**

- **Python** standard library data types for date and time: "datetime", "time", "calendar.
- **Pandas** data type for date and time: "Timestamp".

**DATE RANGES, FREQUENCIES AND SHIFTING**

Generic time series in Pandas are assumed to be irregular, aka have no fixed frequency. However, we prefer to work with fixed frequency, i.e. daily, monthly, etc.

**1. Frequencies and Date Offsets**

- Frequencies in Pandas are composed of a base frequency and a multiplier. Base frequencies are typically referred to by a string alias, like 'M' for monthly or 'H' for hourly.

**2. Generating Date Ranges**

- Default Frequency is Daily

**3. Shifting (Leading and Lagging) Data**

- Shifting refers to moving data backward and forward through time.
- Series and DF if shift() does naive shift, aka index does not shift, only value.

**TIME ZONE HANDLING**

- Daylight saving time (DST) transitions are a common source of complication.
- **UTC** is the current international standard. Time zones are expressed as offsets from UTC.

**PANDA TIME SERIES**

Create Time Series

```python
tsl = pd.Series(np.random.randn(8), index = pd.date_range('2001-01', periods = 8))
# tsl.index is 'Datetimelike' Python class
```

Indexing (Slicing/Subsetting)

Argument can be a string date, datetime, or Timestamp.

Select Year of 2001

```python
tsl['2001']
```

Select June 2001

```python
tsl['2001-06']
```

Indexing From 2001-01 to 2001-08

```python
tsl['2001-01-01':'2001-08-01']
```

Select From 2001-01-01 to 2001-08-01

```python
tsl['2001-01-01':'2001-08-01']
```

Select From 2001-01-01 to 2001-08-01

```python
tsl['2001-01-01':'2001-08-01']
```

Common Operations

Get Time Series Data Before

```python
tsl.truncate(after = '2011-01')
```

**TIME SERIES PLOTTING**

Like other statistical functions, these functions automatically exclude missing data.

```python
df.plot()
```

**RESAMPLING**

Process of converting a time series from one frequency to another frequency.

**1. Downsampling - Aggregating higher frequency data to lower frequency.**

```python
tsl.resample('5min', how = 'ohlc')
```

**2. Upsampling and Interpolation - Interpolate low frequency to higher frequency.**

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

**PERFORMANCE**

- Since "Timestamps" are represented as 64-bit integers using NumPy's `datetime64` type, it means for each data point, there is an associated 8 bytes of memory per timestamp.
- "Creating views" on existing time series or DF do not cause any memory to be used.
- Indexes for lower frequencies (daily and up) are stored in a **central cache**, so any fixed-frequency index is a **view** on the date cache. Thus, low-frequency indexes memory footprint is not significant.
- Performance-wise, Pandas has been highly optimized for alignment operations (i.e. `ts1 + ts2`) and resampling.