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'])` `series1 = pd.Series(dict1)` *
Get Series Values	`series1.values`
Get Values by Index	`series1['a']` `series1[['b','a']]`
Get Series Index	`series1.index`
Get Name Attribute (None is default)	`series1.name` `series1.index.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 substitued into many functions that expect a dict.
- ** Auto-align differently-indexed data in arithmetic operations

DATAFRAME (2D)

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

Create DF
(from a dict of equal-length lists or NumPy arrays)
	`dict1 = {'state': ['Ohio', 'CA'], 'year': [2000, 2010]}`
	`df1 = pd.DataFrame(dict1)`
	# columns are placed in sorted order
	`df1 = pd.DataFrame(dict1, index = ['row1', 'row2']))`
	# specifying index
	`df1 = pd.DataFrame(dict1, columns = ['year', 'state'])`
	# columns are placed in your given order

* Create DF (from nested dict of dicts) The inner keys as row indices	`dict1 = {'col1': {'row1': 1, 'row2': 2}, 'col2': {'row1': 3, 'row2': 4}}` `df1 = pd.DataFrame(dict1)`

Get Columns and Row Names	`df1.columns` `df1.index`
Get Name Attribute (None is default)	`df1.columns.name` `df1.index.name`
Get Values	`df1.values` # returns the data as a 2D ndarray, the dtype will be chosen to accomandate all of the columns
** Get Column as Series	`df1['state']` or `df1.state`
** Get Row as Series	`df1.ix['row2']` or `df1.ix[1]`
Assign a column that doesn't exist will create a new column	`df1['eastern'] = df1.state == '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 modificatons to the data will be reflected in df1.

PANEL DATA (3D)

Create Panel Data : (Each item in the Panel is a DF)

```
import pandas_datareader.data as web
panel1 = pd.Panel({stk : web.get_data_
yahoo(stk, '1/1/2000', '1/1/2010')
for stk in ['AAPL', 'IBM']})
```

panel1 Dimensions : 2 (item) * 861 (major) * 6 (minor)

"Stacked" DF form : (Useful way to represent panel data)

```
panel1 = panel1.swapaxes('item', 'minor')
panel1.ix[:, '6/1/2003', :].to_frame() *
```

=> Stacked DF (with **hierarchical indexing** **) :

```
#                  Open High Low Close Volume Adj-Close
# major      minor
# 2003-06-01 AAPL
#            IBM
# 2003-06-02 AAPL
#            IBM
```

Data Structures Continued

- * DF has a "to_panel()" method which is the inverse of "to_frame()".
- ** Hierarchical indexing makes N-dimensional arrays unnecessary in a lot of cases. Aka prefer to use Stacked DF, not Panel data.

INDEX OBJECTS

Immutable objects that hold the axis labels and other metadata (i.e. axis name)

- i.e. Index, MultiIndex, DatetimeIndex, PeriodIndex
- Any sequence of labels used when constructing Series or DF internally converted to an Index.
- Can functions as fixed-size set in additional to being array-like.

HIERARCHICAL INDEXING

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

MultiIndex :

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

Series Partial Indexing	`series1['b']` # Outer Level `series1[:, 2]` # Inner Level

DF Partial Indexing	`df1['outerCol3','InnerCol2']`
	Or
	`df1['outerCol3']['InnerCol2']`

Swaping and Sorting Levels

Swap Level (level interchanged) *	`swapSeries1 = series1.swaplevel('key1', 'key2')`
Sort Level	`series1.sortlevel(1)` # sorts according to first inner level

Common Ops : Swap and Sort **	`series1.swaplevel(0, 1).sortlevel(0)` # the order of rows also change

- * 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 `sortlevel(0)` or `sort_index()`.

Summary Statistics by Level

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

Sum rows (that have same 'key2' value)	`df1.sum(level = 'key2')`
Sum columns ..	`df1.sum(level = 'col3', axis = 1)`

- 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.

New DF using columns as index	`df2 = df1.set_index(['col3', 'col4'])` * ‡ # col3 becomes the outermost index, col4 becomes inner index. Values of col3, col4 become the index values.

- * "reset_index" does the opposite of "set_index", the hierarchical index are moved into columns.
- ‡ By default, 'col3' and 'col4' will be removed from the DF, though you can leave them by option : `'drop = False'`.

Missing Data

Python	`NaN - np.nan` (not a number)
Pandas *	`NaN` or python built-in `None` mean missing/NA values

* Use `pd.isnull()`, `pd.notnull()` or `series1/df1.isnull()` to detect missing data.

FILTERING OUT MISSING DATA

`dropna()` returns with ONLY non-null data, source data NOT modified.

`df1.dropna()` # drop any row containing missing value
`df1.dropna(axis = 1)` # drop any column containing missing values
`df1.dropna(how = 'all')` # drop row that are all missing
`df1.dropna(thresh = 3)` # drop any row containing < 3 number of observations

FILLING IN MISSING DATA

`df2 = df1.fillna(0)` # fill all missing data with 0 `df1.fillna('inplace = True')` # modify in-place
Use a different fill value for each column : `df1.fillna({'col1' : 0, 'col2' : -1})`
Only forward fill the 2 missing values in front : `df1.fillna(method = 'ffill', limit = 2)` i.e. for column1, if row 3-6 are missing. so 3 and 4 get filled with the value from 2, NOT 5 and 6.

Essential Functionality

INDEXING (SLICING/SUBSETTING) †

† Same as 'NdArray' : In INDEXING : **'view'** of the source array is returned.

† Endpoint is inclusive in pandas slicing with labels : `series1['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'], 'col3']`

Boolean Indexing	`df1[ [True, False] ]`
	`df1[df1['col2'] > 6] *`
	# returns df that has col2 value > 6

Note
- Note that `df1['col2'] > 6` returns a boolean Series, with each True/False value determine whether the respective row in the result.
- Avoid integer indexing since it might introduce subtle bugs (e.g. `series1[-1]`).
- If have to use position-based indexing, use "`iget_value()`" from Series and "`irow/icol()`" from DF instead of integer indexing.

DROPPING ROWS/COLUMNS

Drop operation returns a new object (i.e. DF) :

Remove Row(s) (axis = 0 is default)	`df1.drop('row1')` `df1.drop(['row1', 'row3'])`
Remove Column(s)	`df1.drop('col2', axis = 1)`

REINDEXING

Create a new object with rearraging data conformed 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
 	
(ReIndex default is row index)
	`date_index = pd.date_range('01/23/2010', periods = 10, freq = 'D')`
	
	`df1.reindex(date_index)`

Replace Missing Values (NaN) wth 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[[..], [..]]`

ARITHMETIC AND DATA ALIGNMENT

- `df1 + df2` : For indices that don't overlap, internal data alignment introduces `NaN`.

1, Instead of `NaN`, replace with 0 for the indice that is not found in th df : `df1.add(df2, fill_value = 0)`
2, Useful Operations : `df1 - df1.ix[0]` # subtract every row in df1 by first row

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.

Sort Data

Missing values (`np.nan`) are sorted to the end of the Series by default

Series Sorting	`sortedS1 = series1.order()` `series1.sort()` # In-place sort
DF Sorting	`df1.sort_index(by = ['col2', 'col1'])` # sort by col2 first then col1

Ranking

Break rank ties by assigning each tie-group the mean rank. (e.g. 3, 3 are tie as the 5th place; thus, the result is 5.5 for each)

Output Rank of Each Element
(Rank start from 1)
	`series1.rank()`
	`df1.rank(axis = 1)`
	# rank each row's value

FUNCTION APPLICATIONS

NumPy works fine with pandas objects : np.abs(df1)

Applying a Function to Each Column or Row
(Default is to apply to each column : axis = 0)
	`f = lambda x: x.max() - x.min()` # return a scalar value
	`def f(x): return Series([x.max(), x.min()])`
	# return multiple values
	`df1.apply(f)`

Applying a Function Element-Wise	`f = lambda x: '%.2f' %x`
	`df1.applymap(f)`
	# format each entry to 2-decimals

UNIQUE, COUNTS

- It's **NOT** mandatory for index labels to be unique although many functions require it. Check via : `series1/df1.index.is_unique`
- `pd.value_counts()` returns value frequency.

Data Aggregation and Group Operations

Categorizing a data set and applying a function to each group, whether an aggregation or transformation.

Note: Aggregation of "Time Series" data - please see Time Series section. Special use case of "groupby" is used - called "resampling".

GROUPBY (SPLIT-APPLY-COMBINE)

- Similar to **SQL groupby**

Compute Group Mean	`df1.groupby('col2').mean()`
GroupBy More Than One Key	`df1.groupby([df1['col2'], df1['col3']]).mean()` # result in hierarchical index consisting of unique pairs of keys
"GroupBy" Object : (ONLY computed intermediate data about the group key - `df1['col2']`)	`grouped = df1['col1'].groupby(df1['col2'])` `grouped.mean()` # gets the mean of each group formed by 'col2'

Indexing "GroupBy" Object	# select 'col1' for aggregation :
	`df1.groupby('col2')['col1']`
	or
	`df1['col1'].groupby(df1['col2'])`

Note: Any missing values in the group are excluded from the result.

1. Iterating over GroupBy object

"GroupBy" object supports iteration : generating a sequence of 2-tuples containing the group name along with the chunk of data.

`for name, groupdata in df1.groupby('col2'):` # name is single value, groupdata is filtered DF contains data only match that single value.
`for (k1, k2), groupdata in df1.groupby(['col2', 'col3']):` # If groupby multiple keys : first element in the tuple is a tuple of key values.

Convert Groups to Dict	`dict(list(df1.groupby('col2')))` # col2 unique values will be keys of dict

Group Columns by "dtype"	`grouped = df1.groupby([df1.dtypes, axis = 1)`
	`dict(list(grouped))`
	# separates data Into different types

2. Grouping with functions

Any function passed as a group key will be called once per (default is row index) value, with the return values being used as the group names. (This assumes row index are named)

`df1.groupby(len).sum()` # returns a DF with row index that are length of the names. Thus, names of same length will sum their values. Column names retain.

DATA AGGREGATION

Data aggregation means any data transformation that produces **scalar** values from arrays, such as "mean", "max", etc.

Use Self-Defined Function	`def func1(array): ...` `grouped.agg(func1)`
Get DF with Column Names as Fuction Names	`grouped.agg([mean, std])`
Get DF with Self-Defined Column Names	`grouped.agg([('col1', mean), ('col2', std)])`
Use Different Fuction Depending on the Column	`grouped.agg({'col1' : [min, max], 'col3' : sum})`

GROUP-WISE OPERATIONS AND TRANSFORMATIONS

Agg() is a special case of data transformation, aka reduce a **one-dimensional array to scalar**.

Transform() is a specialized data transformation :
- It applies a function to each group, if it produces a scalar value, the value will be placed in **every row** of the group. Thus, if DF has 10 rows, after "transform()", there will be still 10 rows, each one with the scalar value from its respective group's value from the function.
- The passed function must either produce a scalar value or a transformed array of same size.

General purpose transformation : apply()

```
df1.groupby('col2').apply(your_func1)
# your func ONLY need to return a pandas object or a scalar.
```

```
# Example 1 : Yearly Correlations with SPX
# "close_price" is DF with stocks and SPX closed price columns and dates index
returns = close_price.pct_change().dropna()
by_year = returns.groupby(lambda x : x.year)
spx_corr = lambda x : x.corrwith(x['SPX'])
by_year.apply(spx_corr)
```

```
# Example 2 : Exploratory Regression
import statsmodels.api as sm
def regress(data, y, x):
    Y = data[y]; X = data[x]
    X['intercept'] = 1
    result = sm.OLS(Y, X).fit()
    return result.params
by_year.apply(regress, 'AAPL', ['SPX'])
```

Created by Arianne Colton and Sean Chen
www.datasciencefree.com
Based on content from "Python for Data Analysis" by Wes McKinney
Updated: August 22, 2016

Data Wrangling : Merge, Reshape, Clean, Transform

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`

- Use ALL overlapping column 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 : `left_on='lkey', right_on='rkey'`

- 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").

```
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 value.

```
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 row1 of `df1` and row3 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.

- Default is to stack/unstack innermost level. If want a different level, i.e. `stack(level = 0)` - the outermost level.

Note : Unstacking might introduce missing data if not all of the values in the level aren't found in each of the subgroups. Stacking filters out missing data by default, i.e. `data.unstack().stack()`

2. **Pivoting**

- Common format of storing multiple "time series" in databases and CSV is :

Long/Stacked 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 prefered : 'date' as row index, 'stock_name' as columns, 'price' as DF data values.

```
pivotedDf2 = df1.pivot('date', 'stock_name', 'price')
# Example pivotedDf2 :
#              AAPL    IBM     JD
# 2003-06-01   120.2   100.1   20.8
```

COMMON OPERATIONS

1. **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`.

2. **Add New Column Based On Value of Column(s)**

```
df1['newCol'] = df1['col2'].map(dict1)
```
Maps col2 value as `dict1`'s key, gets `dict1`'s value

```
df1['newCol'] = df1['col2'].map(func1)
```
Apply a function to each col2 value

3. **Replacing Values**

Replace is NOT In-Place

```
df2 = df1.replace(np.nan, 100)
```
Replace Multiple Values At Once

```
df2 = df1.replace([-1, np.nan], 100)
df2 = df1.replace([-1, np.nan], [1, 2])
```
Argument Can Be a Dict As Well

```
df2 = df1.replace({-1: 1, np.nan : 2})
```

4. **Renaming Axis Indexes**

Convert Index to Upper Case	`df1.index = df1.index.map(str.upper)`
Rename 'row1' to 'newRow1'	`df2 = df1.rename(index = {'row1' : 'newRow1'}, columns = str.upper)` # Optionally `inplace = True`

5. **Discretization and Binning**

- Continuous data is often discretized into "bins" for analysis.

Divide Data Into 2 Bins of Number (18 - 26], (26 - 35]
']' means inclusive, ')' is NOT inclusive

```
bins = [18, 26, 35]
cat = pd.cut(array1, bins, labels=[..])
```
`cat` is "Categorical" object.

```
pd.value_counts(cat)
cat = pd.cut(array1, numofBins)
```
Compute equal-length bins based on min and max values in array1

```
cat = pd.qcut(array1, numofBins)
```
Bins the data based on sample quantiles - 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 1 or -1.

7. **Permutation and Random Sampling**

```
randomOrder = np.random.permutation(df1.shape[0])
df2 = df1.take(randomOrder)
```

8. **Computing Indicator/Dummy Variables**

- 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.

```
dummyDf = pd.get_dummies(df1['col2'], prefix = 'col-')
```
Add prefix to the K column names

Getting Data

TEXT FORMAT (CSV)

```
df1 = pd.read_csv(file/URL/file-like-object, sep = ',', header = None)
```
Type-Inference : do NOT have to specify which columns are numeric, integer, boolean or string.

In Pandas, missing data in the source data is usually empty string, NA, -1, #IND or NULL. You can specify missing values via option i.e. : `na_values = ['NULL']`.

Default delimiter is comma.

Default is first row is the column header.

```
df1 = pd.read_csv(.., names = [..])
```
Explicitly specify column header, also imply first row is data

```
df1 = pd.read_csv(.., names = [.., 'date'], index_col = 'date')
```
Want 'date' column to be row index of the returned DF

```
df1.to_csv(filepath/sys.stdout, sep = ',')
```
Missing values appear as empty strings in the output. Thus, It is better to add option i.e. : `na_rep = 'NULL'`

Default is row and column labels are written. Disabled by options : `index = False, header = False`

JSON (JAVASCRIPT 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.

Convert JSON string to Python form	`data = json.load(jsonObj)`
Convert Python object to JSON	`asJson = json.dumps(data)`
Create DF from JSON	`df1 = pd.DataFrame(data['name'], columns = ['field1'])`

XML AND HTML DATA

HTML :

```
doc = lxml.html.parse(urlopen('http://..')).getroot()
tables = doc.findall('.//table')
rows = tables[1].findall('.//tr')
```

XML :

```
lxml.objectify.parse(open(filepath)).getroot()
```

Descriptive Statistics Methods †

† Compared with equivalent methods of ndArray, descriptive statistics methods in Pandas are built from the ground up to exclude missing data.

† NA (i.e. `NaN`) values are excluded. This can be disabled using the "`skipna = False`" option.

Column Sums (Use axis = 1 to sum over rows)

```
series1 = df1.sum()
```
Returns Index Labels Where Min/Max Values are Attained

```
df1.idxmin()   or   df1.idxmax()
```
Mutiple Summary Statistics (i.e. count, mean, std)

On Non-Numeric Data, Alternate Statistics (i.e. count, unique)

```
df1.describe()
```

CORRELATION AND COVARIANCE

- `cov()`, `corr()`
- `corrwith()` - pairwise correlations : aka compute a DF with a Series. If input is not Series, but another DF, it will compute the correlations of matching column names. i.e. `returns.corrwith(volumes)`

```
# Example : Correlation
import pandas_datareader.data as web
data = {}
for ticker in ['AAPL', 'JD']:
    data[ticker] = web.get_data_yahoo(ticker, '1/1/2000', '1/1/2010')
    prices = pd.DataFrame({ticker : d['Adj Close'] for ticker, d in data.iteritems()})
    volumes = ...
returns = prices.pct_change()
returns.AAPL.corr(returns.JD)
```
Series `corr()` computes correlation of overlapping, non-NA, aligned-by-index values in two Series.

Created by Arianne Colton and Sean Chen
www.datasciencefree.com

Based on content from "Python for Data Analysis" by Wes McKinney
Updated: August 22, 2016

Time Series

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

Convert String to DateTime	`from datetime import datetime` `datetime.strptime('8/8/2008', '%m/%d/%Y')`
Get Time Now	`now = datetime.now()`

DateTime Arithmetic	`from datetime import timedelta`
	`datetime(2011, 1, 8) + timedelta(12) => 2011-01-20`
	# Timedelta represents temporal difference between two datetime objects.

Convert String to Pandas Timestamp Type	`timestamps = pd.to_datetime(['8/8/2008', ..])`
	# NaT (Not a Time) is Pandas NA Value for Timestamp Data
	`pd.to_datetime('') => NaT`
	`pd.isnull(NaT) => True`
	# Missing value (i.e. empty string)

† "datetime" is widely used, it stores both the date and time down to microsecond.

* "Timestamp" object can be substituted anywhere you would use "datetime" object.

PANDA TIME SERIES

Create Time Series

```
ts1 = pd.Series(np.random.randn(8), index = [ datetime(2011, 1, 2), .. ])
ts1 = pd.Series(..., index = pd.date_range('1/1/2000', periods = 1000))
```

`ts1.index` is "DatetimeIndex" Panda class

† Index value `ts1.index[0]` is Panda "Timestamp" object which stores timestamp using NumPy's "datetime64" type at the nanoseond resolution. Further, Timestamp class stores the frequency information as well as timezone. `ts1.index.dtype => datetime64[ns]`

Indexing (Slicing/Subsetting)

Argument can be a string date, datetime or Timestamp.

Select Year of 2001	`ts1['2001']` `df1.ix['2001']`
Select June 2001	`ts1['2001-06']`
Select From 2001-01-01 to 2001-08-01	`ts1['1/1/2001':'8/1/2001']`
Select From 2001-01-08 On	`ts1[datetime(2001, 1, 8):]`

Common Operations

Get Time Series Data Before 2011-01-09	`ts1.truncate(after = '1/8/2011')`

DATE RANGES, FRQUENCIES 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.

Take a Look at "Resampling" Section	# Convert to Fixed Daily Frequency. # Introduce Missing Value (NaN) If Needed `ts1.resample('D', how = ..)`

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.

```
freq = '4H'
freq = '1h30min'
```

Standard US equity option monthly expirataion, every third Friday of the month : `freq = 'WOM-3FRI'`

2. Generating Date Ranges

Default Frequency is Daily	`pd.date_range(begin, end)` Or `pd.date_range(begin or end, periods = n)` # Option `freq = 'BM'` means last business day at end of the month

3. Shifting (Leading and Lagging) Data

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

`ts1` is Daily Data `ts1.shift(1)` # move yesterday's value to today, today value to tomorrow, etc.
`ts1` is Any Time Series Data. Shift Data By 3 Days `ts1.shift(3, freq = 'D')` Or `ts1.shift(1, freq = '3D')`
Common Use of Shift : To Computer % Change `ts1 / ts.shift(1) - 1`

* In the return result from shift(), some data value might be NaN.

- Other ways to shift data :

```
from pandas.tseries.offsets import Day, MonthEnd
datetime(2008, 8, 8) + 3*Day() => 2008-08-11
datetime(2008, 8, 8) + MonthEnd(2) => 2008-09-30
MonthEnd().rollforward(datetime(2008, 8, 8)) => 2008-08-31
```

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. *

* NY is 4 hours behind UTC during daylight saving time and 5 hours the rest of the year.

1. Python Time Zone (From 3rd-party pytz library)

Get List of Timezone Names	`pytz.common_timezones`
Get a Timezone Object	`pytz.timezone('US/Eastern')`

2. Localization and Conversion

Time Series By Default is Time Zone Naive	`ts1.index.tz` => None
Specify Time Zone When Create Time Series	Use option : `tz = 'UTC'` in `pd.date_range()`
Localization From Naive	`ts1_utc = ts1.tz_localize('UTC')`
Convert to Another Time Zone Once Time Series Been Localized	`ts1_eastern = ts1_utc.tz_convert('US/Eastern')`

3. ** Time Zone-aware Timestamp Objects

```
stamp_utc = pd.Timestamp('2008-08-08 03:00', tz = 'UTC')
stamp_eastern = stamp_utc.tz_convert(...)
```

Panda's Time Arithmetic - Daylight Savings Time Transitions Are Respected :

`stamp = pd.Timestamp('2012-11-04 00:30', tz = 'US/Eastern')` => 2012-11-04-00:30:00 -400 EDT

`stamp + 2 * Hour()` => 2012-11-04-01:30:00 -500 EST

** If two time series with different time zones are combined, i.e. ts1 + ts2, the timestamps will auto-align with respect to time zone. The result will be in UTC.

RESAMPLING

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

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

* `ts1.resample('M', how = 'mean')` => Index : 2000-01-31, 2000-02-29, ...
`ts1.resample('M', ..., kind ='period')` # 'period' - Use time-span representation => Index : 2000-01, 2000-02, ...
`ts1` is one minute data of value 1 to 100 of time : 00:00:00, 00:01:00, ... `ts1.resample('5min', how = 'sum') =>` 00:00:00 15 (aka : 1 + 2 + 3 + 4 + 5) 00:05:00 40 # Default is left bin edge is inclusive, thus 00:00:00 value in included in the 00:00:00 to 00:05:00 interval. # Option : `closed = 'right'` change interval to right inclusive. Also include option `label = 'right'` as well : 00:00:00 1 00:05:00 20 (aka : 2 + 3 + 4 + 5 + 6)

`ts1.resample('5min', how = 'ohlc')` # returns a DF with 4 columns - open, high, low , close

* Alternate way to downsample : `ts1.groupby(lamba x : x.month).mean()`

2. Upsampling and Interpolation * - Interpolate low frequency to higher frequency. By default missing values (NaN) are introduced.

`df1.resample('D', fill_method = 'ffill')` # Forward fills values : i.e. missing value index such as index 3 will copy value from index 2.

* Interpoation will ONLY apply row-wise.

TIME SERIES PLOTTING

Example : Source Data Format - First Column is Date.

Use first column as the Index, then parse the index values as Date.

```
prices = pd.read_csv(.., parse_date = True, index_col = 0)
px = prices[['AAPL', 'IBM']]
px = px.resample('B', fill_method = 'ffill')
px['AAPL'].plot()
px['AAPL'].ix['01-2008':'03-2012'].plot()
px.ix['2008'].plot()
```

MOVING WINDOW FUNCTIONS

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

```
pd.rolling_mean(px.AAPL, 200).plot()
pd.rolling_std(px.AAPL.pct_change(), 22, min_periods = 20).plot()
pd.rolling_corr(px.AAPL.pct_change(), px.IBM.pct_change(), 22).plot()
```

PERFORMANCE

- Since "Timestamps" is 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 more 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 data alignment operations (i.e. `ts1 + ts2`) and resampling.

Created by Arianne Colton and Sean Chen
www.datasciencefree.com
Based on content from "Python for Data Analysis" by Wes McKinney
Updated: August 22, 2016