Overview

> *A simple analysis of clean data can be more productive than a complex analysis of noisy and irregular data.*

In chapter 1, we came across the **Data Science Process** as a list of tasks that Data Scientists perform in their day-to-day work. This can be considered to be an extension of **the Scientific Method** (which is a commonly taught framework for conducting scientific investigations) and can be applied to solving almost any kind of problem.

Attempts have been made by several practitioners to give structure to this process that begins with raw data and ends in a data product (typically a machine learning model or an interactive visualization.)

The CRISP-DM methodology and Hilary Mason's OSEMN things are examples of this process.

Fig. 3-1 CRISP-DM

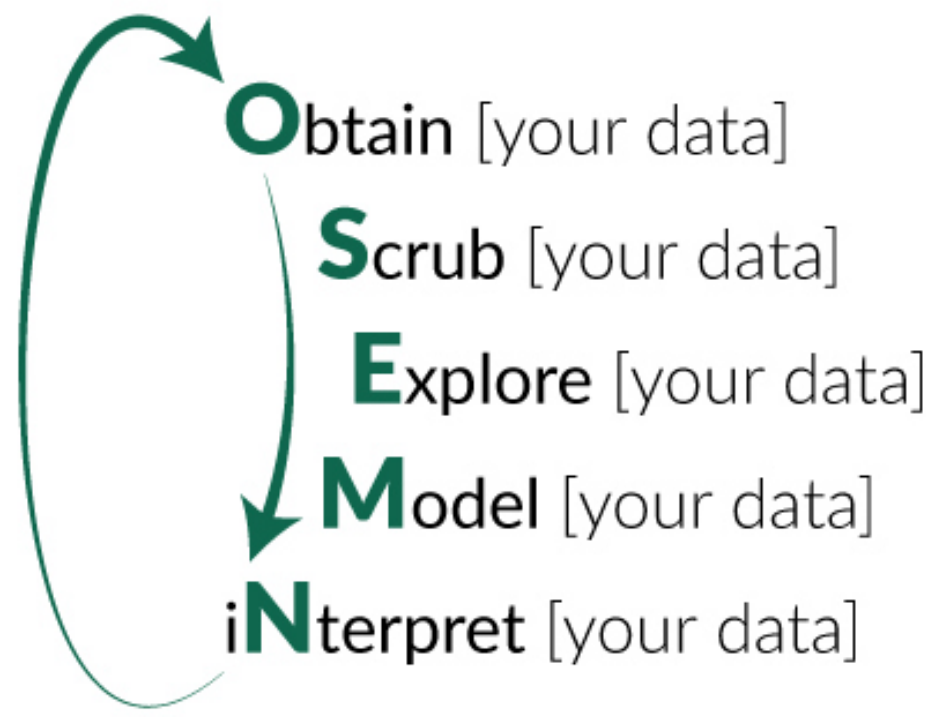


Fig 3-2 OSEMN Things

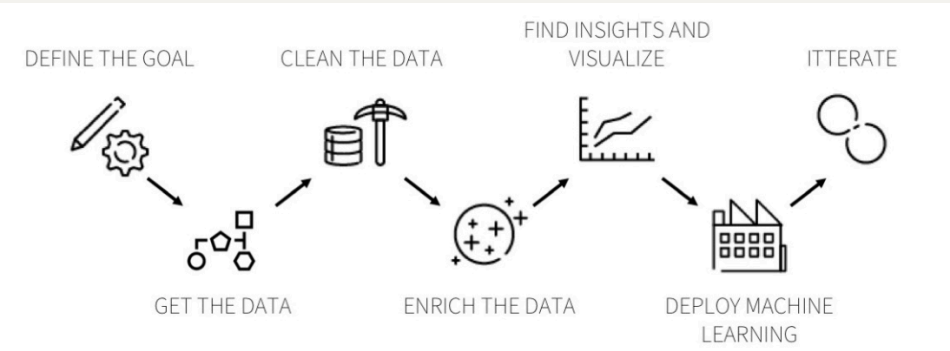


Fig 3-2-1 The Data Science Process

Broadly, it consists of the following steps

STEP	TASK
1. Question	Clearly define the business problem
2. Get Data	Obtain data from internal/external sources, APIs
3. Wrangle	Clean messy data. Engineer features. Roll data up (or down.)
4. Explore	Visualize distributions. Investigate relationships. Build intuition for subsequent steps.

5. Model	Build and Tune models. Select the best from competing statistical models.
6. Interpret	Assess model performance on out-of-sample data. Understand results. Draw Insights.
7. Deploy	Productionalize your analysis. Build a data product.

In this chapter, we will focus on #2 (Get Data) and #3 (Wrangle) above.

But before we begin, a word of caution.
Raw data in the real-world is often *messy* (or *untidy*) and before we conduct any analysis on the data, we must first make it *tidy*. Cleaning data can be a tedious and repetitive task, and handling this problem inefficiently may lead to great annoyance and/or frustration.

The path from messy to clean data is fraught with sinister dangers like strange delimiters, inconsistent character encodings, spelling mistakes, missing values, dates stored as numbers ... the list goes on. It is for these reasons that data preparation has often been labeled by various practitioners as the *least sexy, most time & labour intensive* task in data science.

Fig 3-3 Data Preparation will take upto 80% of your time

Here are some of the more commonly observed patterns in unclean data.

- Table headers are numbers instead of text
- Data from multiple variables are concatenated into a single column
- **Missing data** are encoded in different ways
- Features are stored using varying units of measurement (discrepancy of **scales**)
- **Outliers**

All of the tasks listed here (and more) are performed in Python using the `pandas` library.

Once the data is clean (or *tidy*) it may still require more work before it can be used to create meaningful visualizations or for building machine learning models. This is where we perform advanced data wrangling tasks such as

- **Reshaping** long-data to wide-data
- **Subsetting** data to retain relevant rows and/or columns
- **Aggregating** data using the `split-apply-combine` strategy
- **Sorting , Merging** to combine data from different sources

If you persevere through these steps, you should have a `tidy` dataset that satisfies the following criterion and makes it easy to carry out data analysis.

- Observations are in rows
- Variables are in columns
- Entities of one kind to be contained in a single dataset (ex. customers, transactions)

Pandas

Pandas (an acronym, stands for **PAN**el **D**ata **A**nalysi**S**) is a free, open-source data science library aimed at quick and simplified data munging and exploratory analysis in Python. Specifically, it provides high-level data structures like the `DataFrame` (similar to the R data.frame) and `Series` (one-dimensional structure with an index) which have rich methods for tackling the entire spectrum of data munging tasks. Additionally, `pandas` has specialized methods for manipulating and visualizing numerical variables and time series data.

Pandas creator Wes McKinney started building the library in 2008 during his time at an investment management firm. He was motivated by a need to address a distinct set of data analysis requirements that were not fully satisfiedby any one tool at his disposal at the time.

> *Python had long been great for data munging and preparation, but less so for data analysis and modeling.*
> *Pandas helps fill this gap, enabling you to carry out your entire data analysis workflow in Python without having to switch to a more domain specific language.*

Features

- Easy handling of ***missing data.*** (`dropna, fillna, ffill, isnull, notnull`)
- Simple ***mutations*** of tables to add/remove columns on the fly (`assign, drop`)
- Easy ***slicing*** of data (fancy indexing and subsetting)

- Automatic ***data alignment*** (by `index`)
- Powerful ***split-apply-combine*** (`groupby`) for aggregations and transformations
- Intuitive ***merge/join*** (`concat, join`)
- Flexible **Reshaping** and ***Pivoting*** (`stack, pivot`)
- ***Hierarchical Labeling*** of axes indices for working with higher-dimensional data
- Robust ***I/O tools*** to work with csv, Excel, flat files, ***databases and HDFS***
- Integrated ***Time Series*** Functionality
- Easy visualization (`plot`)

Pandas is built on a solid foundation of NumPy arrays, and is optimized for performance (pandas is about 15x faster), with essential code pieces written in Cython or C.
NumPy's `ndarray` and its broadcasting capabilities are leveraged extensively.

The documentation is available at http://pandas.pydata.org

We will now take a quick detour to highlight a few important concepts of NumPy that will simplify our understanding of a few concepts in pandas.

NumPy Reference

Numpy is the fundamental library for all scientific computing in Python. It provides a *high-performance, homogenous, multidimensional* array and matrix objects called the `ndarray`, along with an assortment of high-level functions for fast operations on these arrays, such as:

- mathematical, logical routines
- shape manipulations
- sorting, selecting data
- basic linear algebra and statistical operations,
- random number generation capabilities
- sophisticated (broadcasting) functions

Almost every data analysis or machine learning package in the PyData ecosystem (most notably `pandas` and `scikit-learn`) uses NumPy ndarrays under the hood.

Alright, let's jump into some code. Conventionally, we start with importing NumPy

```
import numpy as np
```

NumPy Arrays

It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In Numpy dimensions are called *axes*. The number of axes is *rank*. For example, a 2D array has a rank of 2.

- Array can be of different **types**
    - including `int, float, string, object, bool`
    - type conversion can be achieved using `.astype()`

- can be **created** by
    - calling `np.array()` or `reshape` on Python lists (gives 1D arrays) or nested lists (gives 2D arrays)
    - using special functions like `np.zeros(), np.ones(), np.eye()`
    - sequence generators like `np.linspace(), np.arange()`
    - random number generators like `np.random.randint(), np.random.randn()`
    - You may read about other methods of array creation in the documentation

```
# Create a 1D array from a Python list
 In [9]: ndarr_1 = np.array([100, 3, 19, 75, 43])
In [10]: ndarr_1
Out[10]: array([100,   3,  19,  75,  43])

# Create a 2D array from a Python list-of-lists
In [11]: ndarr_2 = np.array([[2, 4, 6], [3, 5, 7]])
In [12]: ndarr_2
Out[12]:
array([[2, 4, 6],
       [3, 5, 7]])

# Create a 2D array by reshaping a 1D array
In [14]: ndarr_3 = np.arange(20).reshape(4, 5)
In [15]: ndarr_3
Out[15]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])

# Create a 3x3 array filled with random integers
In [16]: arr_4 = np.random.randint(0, 100,
9).reshape(3, 3)
In [17]: arr_4
Out[17]:
array([[66, 87, 41],
       [49, 41, 33],
       [ 5, 57, 78]])

# Create a 4x4 matrix filled with random numbers
In [26]: arr_5 = np.random.rand(4, 4)
In [27]: arr_5
Out[27]:
array([[ 0.35827069,  0.0687782 ,  0.16042925,
0.40275736],
       [ 0.8666519 ,  0.66905765,  0.05632903,
0.41828043],
       [ 0.12067091,  0.34682221,  0.68698091,
0.87333071],
       [ 0.72627949,  0.56991059,  0.80397042,
0.50586414]])
```

- can be **indexed** by
    - using **integer slices** like Python lists, but here we provide one slice each for rows and columns as ndarrays

may be multidimensional

- Syntax `ndarr_1[<for rows> start: stop: step, <for columns> start: stop: step]`

```
# Using one of the arrays we created above
In [42]: ndarr_3
Out[42]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])

# Subset one element
In [43]: ndarr_3[0, 0]
Out[43]: 0

# Subset the first row
In [44]: ndarr_3[0, :]
Out[44]: array([0, 1, 2, 3, 4])

# Subset the first column
In [45]: ndarr_3[:, 0]
Out[45]: array([ 0,  5, 10, 15])

# Subset the 2nd and 3rd rows
In [46]: ndarr_3[1:3, :]
Out[46]:
array([[ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

# Subset the 3rd and 4th columns
In [47]: ndarr_3[:, 2:4]
Out[47]:
array([[ 2,  3],
       [ 7,  8],
       [12, 13],
       [17, 18]])

# Subset both rows and columns (2nd and 3rd rows and
columns)
In [48]: ndarr_3[1:3, 1:3]
Out[48]:
array([[ 6,  7],
       [11, 12]])
```

* using **boolean subsetting** to select the elements of an array that satisfy some condition

```
# Create an array of numbers from 0 to 15
In [57]: arr_6 = np.arange(15)
In [58]: arr_6
Out[58]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
10, 11, 12, 13, 14])

# Logical operations on arrays produce Boolean arrays
In [59]: arr_6 % 2 == 0
Out[59]:
array([ True, False,  True, False,  True, False,  True,
False,  True,
       False,  True, False,  True, False,  True],
dtype=bool)

# Boolean array to subset even numbers
In [60]: even_bool = (arr_6 % 2 == 0)

# Boolean subsetting
In [61]: arr_6[even_bool]
Out[61]: array([ 0,  2,  4,  6,  8, 10, 12, 14])
```

- Mathematical operations involving arrays are carried out **element-wise** and can be specified
    - using mathematical operators like `+, -, *, /`
    - using numpy functions like `np.add(), np.subtract(), np.multiply(), np.divide()`

```
# Create two new arrays
In [73]: x = np.array([[2, 5],[9, 6]],
dtype=np.float64)
In [74]: y = np.array([[8, 4],[3, 7]],
dtype=np.float64)
In [75]: x
Out[75]:
array([[ 2.,  5.],
       [ 9.,  6.]])
In [76]: y
Out[76]:
array([[ 8.,  4.],
       [ 3.,  7.]])

# Add
In [77]: x + y
Out[77]:
array([[ 10.,   9.],
       [ 12.,  13.]])

# Subtract
In [78]: x - y
Out[78]:
array([[-6.,  1.],
       [ 6., -1.]])

# Multiply
In [79]: x * y
Out[79]:
array([[ 16.,  20.],
       [ 27.,  42.]])

# Divide
In [81]: x / y
Out[81]:
array([[ 0.25      ,  1.25      ],
       [ 3.        ,  0.85714286]])
```

- NumPy has universal functions or `ufuncs` that operate elementwise on an array, producing an array as output.
    - Ex. `np.sqrt()`
- Array **Attributes** and **Methods** include the following (assume we have an array object called *ndarr*.)

```
ndarr.all          ndarr.data         ndarr.nbytes       ndarr.size
ndarr.any          ndarr.diagonal     ndarr.ndim         ndarr.sort
ndarr.argmax       ndarr.dot          ndarr.newbyteorder ndarr.squeeze
ndarr.argmin       ndarr.dtype        ndarr.nonzero      ndarr.std
ndarr.argpartition ndarr.dump         ndarr.partition    ndarr.strides
ndarr.argsort      ndarr.dumps        ndarr.prod         ndarr.sum
ndarr.astype       ndarr.fill         ndarr.ptp          ndarr.swapaxes
ndarr.base         ndarr.flags        ndarr.put          ndarr.T
ndarr.byteswap     ndarr.flat         ndarr.ravel        ndarr.take
ndarr.choose       ndarr.flatten      ndarr.real         ndarr.tobytes
ndarr.clip         ndarr.getfield     ndarr.repeat       ndarr.tofile
ndarr.compress     ndarr.imag         ndarr.reshape      ndarr.tolist
ndarr.conj         ndarr.item         ndarr.resize       ndarr.tostring
ndarr.conjugate    ndarr.itemset      ndarr.round        ndarr.trace
ndarr.copy         ndarr.itemsize     ndarr.searchsorted ndarr.transpose
ndarr.ctypes       ndarr.max          ndarr.setfield     ndarr.var
ndarr.cumprod      ndarr.mean         ndarr.setflags     ndarr.view
ndarr.cumsum       ndarr.min          ndarr.shape
```

As we saw for methods and attributes of other Python objects, these too can be explored using the in-built help. We will be seeing some of these attributes and methods again with pandas' Series and DataFrame objects.

Getting Data into Pandas

We start, as always, by importing modules and frequently used classes from the module. As per convention, we write

```
import pandas as pd
from pandas import Series, DataFrame
```

Reading Flat Files

pandas has functions such as `read_csv, read_table, read_fwf` and `read_clipboard` for reading tabular data into a DataFrame object. These functions take as arguments the following options:

- Which columns to consider?
    - Import the header (`header=None`) or provide column names (`names=`)
- Type inference and conversion
    - Processing dates, combining date and time
- Which column serves as the index? (`index_col=`)
    - For a hierarchical index, pass a list of column names
- Which values to interpret as missing data (`na_values=`)
    - If there are multiple sentinels for missing data, pass a dictionary

- If the file is too large, read chunks iteratively (`nrows=` and `chunksize=`)
- Skipping over rows/footer (`skiprows=`)
- Interpreting decimal numbers (points or commas to mark thousands)

Reading a CSV is as simple as calling the read_csv function. By default, the read_csv function expects the column separator to be a comma, but you can change that using the sep parameter.

Syntax:

```
 pd.read_csv(filepath, sep=, header=, names=,
skiprows=, na_values= ... )
```

These functions are also able to pull data from a URL.

Support for MS Excel

Pandas allows you to read and write Excel files, so you can easily read from Excel, write your code in Python, and then write back out to Excel - no need for VBA. Reading Excel files requires the xlrd library.

You can install it via pip (`pip install xlrd`).

```
football = pd.read_excel('football.xlsx', 'sheet1')
football.to_excel('football.xlsx', index=False)
```

Support for SQL Databases

pandas also has some support for reading/writing DataFrames directly from/to a database. The workflow consists of setting up a connection to the database (add-on libraries like `sql-alchemy` and `psycopg2` help in establishing connections to databases like RDBMS and Redshift) and then passing a SQL query over that connection to the database. The query results (usually a table) are returned to pandas as a DataFrame.

```
import sqlite3

conn = sqlite3.connect('my-sqlite-database.db')
query = "SELECT * FROM my-sqlite-database LIMIT 10;"

df = pd.read_sql(query, con=conn)
```

Reading from the Clipboard

Sometimes we need to quickly import data from an HTML table or an Excel sheet without needing to import the entire page or workbook. We could then just highlight the required data, hit `CTRL + C`, move over to pandas and run `pd.read_clipboard()`. It may be prudent, after the data is imported into your workspace, to back it up using DataFrame methods like `to_csv`

Example:

```
df = pd.read_clipboard()
```

Pandas Data Structures

Pandas has two **workhorse** data structures – *Series* and *DataFrame* - that provide robust and easy-to-implement solutions to many data analysis tasks. These are built on top of (1D and 2D) NumPy arrays and inherit many of their

Operations using these structures include

CREATE	SUBSET	INSERT	UPDATE	VIEW
FILTER	APPEND	SORT	JOIN	MERGE
GROUP	SUMMARIZE	RESHAPE	MAP	APPLY

1. Series

A `Series` is a one-dimensional, homogeneous array-like data structure containing a vector of data (of any valid NumPy type) and an associated array of data labels, called its *index*.

1.1 Creation

A Series can be created by calling the `pd.Series()` function on a NumPy 1D array or a Python object (like list, tuple, dictionary.) The index of the Series could be constructed in the same way from one of these objects. If the index isn't explicitly specified, a numeric (from 0 to length-1) index is automatically generated. Additionally, the user may specify the *type* and a *name* while declaring a Series.

Syntax:

```
import pandas as pd
from pandas import Series

pd.Series(data=, index=, dtype=, name=)
```

Examples:

```
# Basic Series with an array, no index
In []: series_1 =
pd.Series(np.random.random(5).round(2))
In []: series_1
Out[]:
0    0.03
1    0.94
2    0.69
3    0.16
4    0.61
dtype: float64
```

Here, we created a series called *series_1* using an array of 5 elements filled with random numbers. The type was induced from the data to be *float64* and a numeric index was automatically generated. Note that this series has no *name*.

```
In []: series_2 = pd.Series(np.random.randint(0, 100,
5),
    ...: index=list('abcde'),
    ...: dtype=float,
    ...: name='S2')

In []: series_2
Out[]:
a    60.0
b    13.0
c    54.0
d    13.0
e    65.0
Name: S2, dtype: float64
```

We create a series of random integers, but convert the type to *float* explicitly. The automatic index is replaced by the one we provide. The name *S2* is important as it will later become the column name if this Series is imported into a DataFrame.

```
# From a numeric list
In []: pd.Series([-55, 4, 79, 101])
Out[]:
0    -55
1      4
2     79
3    101
dtype: int64

# From a tuple
In []: pd.Series((1, 2, 3, 4))
Out[]:
0    1
1    2
2    3
3    4
dtype: int64

# From a dictionary
# Note that when we use a Python dict to create a
Series, the keys become the index.
In []: pd.Series({'a': 101, 'b': -55, 'c': 79, 'd': 4})
Out[]:
a    101
b    -55
c     79
d      4
dtype: int64
```

A series can be converted into a list or a dictionary using methods like `tolist()` and `to_dict()`

1.2 Series Attributes

Just like attributes for primitive Python data structures like Lists or Dictionaries provide useful metadata about the contents of the structure, we can use Series attributes like `values, index, shape`

```
In [ ]: series_2
Out[ ]:
a    34.0
b    60.0
c    21.0
d    22.0
e     7.0
Name: S2, dtype: float64
# Get the underlying NumPy array
In [ ]: series_2.values
Out[ ]: array([ 34.,  60.,  21.,  22.,   7.])

# Get the index
In [ ]: series_2.index
Out[ ]: Index([u'a', u'b', u'c', u'd', u'e'],
dtype='object')

# Get the size on disk
In [ ]: series_2.nbytes
Out[ ]: 40

# Get the number of elements
In [ ]: series_2.shape
Out[ ]: (5,)
```

1.3 Subsetting a Series

There are many ways to extract subsets of a Series in Pandas. In addition to allowing NumPy-like subsetting using integer lists and slices, it is possible to subset a Series using

- *label-based indexing* by passing index labels associated with the values
- *fancy indexing* using methods like `loc, iloc, ix, at, iat`
- *boolean indexing* for subsetting with logical arrays

Example: Label and integer based indexing on a *Breakfast Menu*

```
In [16]: menu = Series({'ham': 1, 'eggs': 3, 'bacon':
2, 'coffee': 1, 'toast': 0.5, 'jam': 0.2})
In [17]: menu
Out[17]:
bacon     2.0
coffee    1.0
eggs      3.0
ham       1.0
jam       0.2
```

```
toast     0.5
dtype: float64

# A single label
In [18]: menu['bacon']
Out[18]: 2.0

# A list of labels
In [19]: menu[['eggs', 'bacon']]
Out[19]:
eggs     3.0
bacon    2.0
dtype: float64

# A slice of labels
In [20]: menu['bacon':'eggs']
Out[20]:
bacon     2.0
coffee    1.0
eggs      3.0
dtype: float64

# Another label slice
In [21]: menu['ham':]
Out[21]:
ham      1.0
jam      0.2
toast    0.5
dtype: float64

# Positional Slicing
In [22]: menu[0:3]
Out[22]:
bacon     2.0
coffee    1.0
eggs      3.0
dtype: float64

# More positional slicing
In [24]: menu[0::2]
Out[24]:
bacon    2.0
eggs     3.0
jam      0.2
dtype: float64

# Reversing the Series
```

```
In [23]: menu[::-1]
Out[23]:
toast     0.5
jam       0.2
ham       1.0
eggs      3.0
coffee    1.0
bacon     2.0
dtype: float64
```

- Subsetting with *fancy indexing* methods
    - `.loc()` for label based subsetting
    - `.iloc()` for integer based subsetting

  Note that other methods such as `.ix()` and `.at(), .iat()` exist, but they serve the same purpose. So we will leave it to the reader to explore these functions in the documentation.

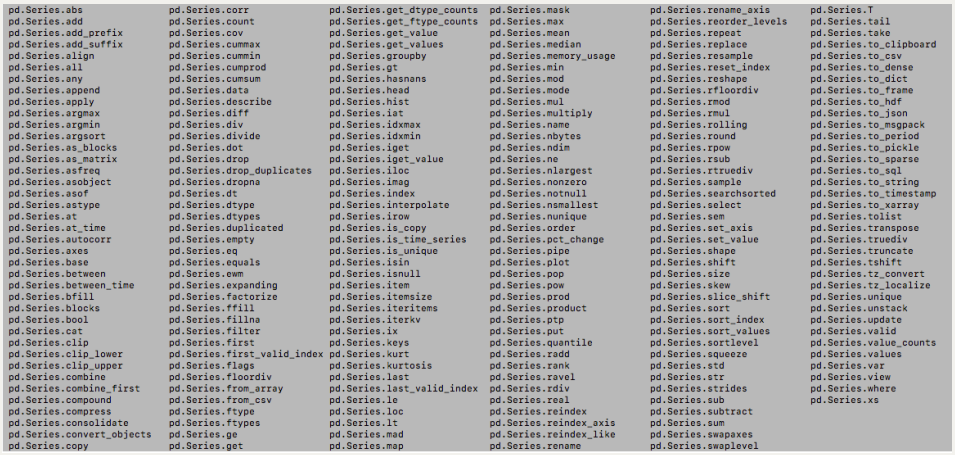
```
# Using loc
In [30]: menu.loc['coffee']
Out[30]: 1.0

In [31]: menu.loc['eggs':'jam']
Out[31]:
eggs    3.0
ham     1.0
jam     0.2
dtype: float64

# Using iloc
In [32]: menu.iloc[3]
Out[32]: 1.0

In [33]: menu.iloc[2:4]
Out[33]:
eggs    3.0
ham     1.0
dtype: float64
```

- *boolean indexing* works in the same way as it does for subsetting NumPy arrays. We create a boolean of the same length as the Series, (using the same Series), and then pass the boolean to the squre bracket subsetter.

```
# Create a boolean series using a logical comparison
In [35]: menu > 1
Out[35]:
bacon      True
coffee    False
eggs       True
ham       False
jam       False
toast     False
dtype: bool

# Subset the series using the boolean to retain values
where True
In [36]: menu[menu > 1]
Out[36]:
bacon    2.0
eggs     3.0
dtype: float64

# We can also pass any boolean (needs to be of the same
length)
In [37]: menu[[True, True, False, False, True, False]]
Out[37]:
bacon     2.0
coffee    1.0
jam       0.2
dtype: float64
```

1.4 Important Series Methods

We've just seen a few Series methods that allow us to subset data. There's a variety of other methods that are useful across the entire spectrum of data wrangling tasks. The figure below shows an exhaustive list of all methods. We will discuss some the most important ones of these here, and encourage the readers to peruse the latest pandas documentation to explore other methods.



```
pd.Series.abs                pd.Series.corr                pd.Series.get_dtype_counts    pd.Series.mask                pd.Series.rename_axis         pd.Series.T
pd.Series.add                pd.Series.count               pd.Series.get_ftype_counts    pd.Series.max                 pd.Series.reorder_levels      pd.Series.tail
pd.Series.add_prefix         pd.Series.cov                 pd.Series.get_value           pd.Series.mean                pd.Series.repeat              pd.Series.take
pd.Series.add_suffix         pd.Series.cummax              pd.Series.get_values          pd.Series.median              pd.Series.replace             pd.Series.to_clipboard
pd.Series.align              pd.Series.cummin              pd.Series.groupby             pd.Series.memory_usage        pd.Series.resample            pd.Series.to_csv
pd.Series.all                pd.Series.cumprod             pd.Series.gt                  pd.Series.min                 pd.Series.reset_index         pd.Series.to_dense
pd.Series.any                pd.Series.cumsum              pd.Series.hasnans             pd.Series.mod                 pd.Series.reshape             pd.Series.to_dict
pd.Series.append             pd.Series.data                pd.Series.head                pd.Series.mode                pd.Series.rfloordiv           pd.Series.to_frame
pd.Series.apply              pd.Series.describe            pd.Series.hist                pd.Series.mul                 pd.Series.rmod                pd.Series.to_hdf
pd.Series.argmax             pd.Series.diff                pd.Series.iat                 pd.Series.multiply            pd.Series.rmul                pd.Series.to_json
pd.Series.argmin             pd.Series.div                 pd.Series.idxmax              pd.Series.name                pd.Series.rolling             pd.Series.to_msgpack
pd.Series.argsort            pd.Series.divide              pd.Series.idxmin              pd.Series.nbytes              pd.Series.round               pd.Series.to_period
pd.Series.as_blocks          pd.Series.dot                 pd.Series.iget                pd.Series.ndim                pd.Series.rpow                pd.Series.to_pickle
pd.Series.as_matrix          pd.Series.drop                pd.Series.iget_value          pd.Series.ne                  pd.Series.rsub                pd.Series.to_sparse
pd.Series.asfreq             pd.Series.drop_duplicates     pd.Series.iloc                pd.Series.nlargest            pd.Series.rtruediv            pd.Series.to_sql
pd.Series.asobject           pd.Series.dropna              pd.Series.imag                pd.Series.nonzero             pd.Series.sample              pd.Series.to_string
pd.Series.asof               pd.Series.dt                  pd.Series.index               pd.Series.notnull             pd.Series.searchsorted        pd.Series.to_timestamp
pd.Series.astype             pd.Series.dtype               pd.Series.interpolate         pd.Series.nsmallest           pd.Series.select              pd.Series.to_xarray
pd.Series.at                 pd.Series.dtypes              pd.Series.irow                pd.Series.nunique             pd.Series.sem                 pd.Series.tolist
pd.Series.at_time            pd.Series.duplicated          pd.Series.is_copy             pd.Series.order               pd.Series.set_axis            pd.Series.transpose
pd.Series.autocorr           pd.Series.empty               pd.Series.is_time_series      pd.Series.pct_change          pd.Series.set_value           pd.Series.truediv
pd.Series.axes               pd.Series.eq                  pd.Series.is_unique           pd.Series.pipe                pd.Series.shape               pd.Series.truncate
pd.Series.base               pd.Series.equals              pd.Series.isin                pd.Series.plot                pd.Series.shift               pd.Series.tshift
pd.Series.between            pd.Series.ewm                 pd.Series.isnull              pd.Series.pop                 pd.Series.size                pd.Series.tz_convert
pd.Series.between_time       pd.Series.expanding           pd.Series.item                pd.Series.pow                 pd.Series.skew                pd.Series.tz_localize
pd.Series.bfill              pd.Series.factorize           pd.Series.itemsize            pd.Series.prod                pd.Series.slice_shift         pd.Series.unique
pd.Series.blocks             pd.Series.ffill               pd.Series.iteritems           pd.Series.product             pd.Series.sort                pd.Series.unstack
pd.Series.bool               pd.Series.fillna              pd.Series.iterkv              pd.Series.ptp                 pd.Series.sort_index          pd.Series.update
pd.Series.cat                pd.Series.filter              pd.Series.ix                  pd.Series.put                 pd.Series.sort_values         pd.Series.valid
pd.Series.clip               pd.Series.first               pd.Series.keys                pd.Series.quantile            pd.Series.sortlevel           pd.Series.value_counts
pd.Series.clip_lower         pd.Series.first_valid_index   pd.Series.kurt                pd.Series.radd                pd.Series.squeeze             pd.Series.values
pd.Series.clip_upper         pd.Series.flags               pd.Series.kurtosis            pd.Series.rank                pd.Series.std                 pd.Series.var
pd.Series.combine            pd.Series.floordiv            pd.Series.last                pd.Series.ravel               pd.Series.str                 pd.Series.view
pd.Series.combine_first      pd.Series.from_array          pd.Series.last_valid_index    pd.Series.rdiv                pd.Series.strides             pd.Series.where
pd.Series.compound           pd.Series.from_csv            pd.Series.le                  pd.Series.real                pd.Series.sub                 pd.Series.xs
pd.Series.compress           pd.Series.ftype               pd.Series.loc                 pd.Series.reindex             pd.Series.subtract
pd.Series.consolidate        pd.Series.ftypes              pd.Series.lt                  pd.Series.reindex_axis        pd.Series.sum
pd.Series.convert_objects    pd.Series.ge                  pd.Series.mad                 pd.Series.reindex_like        pd.Series.swapaxes
pd.Series.copy               pd.Series.get                 pd.Series.map                 pd.Series.rename              pd.Series.swaplevel
```

Peeking at the data

- `head and tail` are used to view a small sample of a Series or DataFrame object, use the `head()` and `tail()` methods. The default number of elements to display is five, but you may pass a custom number.

```
In [44]: series_8 =
Series(np.random.randn(1000).round(2))

In [45]: series_8.head()
Out[45]:
0   -0.23
1    0.55
2    0.77
3    0.18
4    0.76
dtype: float64

In [46]: series_8.tail()
Out[46]:
995    0.66
996   -0.48
997    1.13
998    1.05
999    1.61
dtype: float64
```

Type Conversion

- `astype` explicitly convert dtypes from one to another

```
In [51]: series_8.head()
Out[51]:
0   -0.23
1    0.55
2    0.77
3    0.18
4    0.76
dtype: float

In [52]: series_8.astype(str).head()
Out[52]:
0    -0.23
1     0.55
2     0.77
3     0.18
4     0.76
dtype: object
```

Treating Outliers

- `clip_upper, clip_lower` can be used to clip outliers at a threshold value. All values lower than the one supplied to `clip_lower`, or higher than the one supplied to `clip_upper` will be replaced by that value. Note how the values in *series_8* below are clipped at the supplied threshold of `0.50`

  This function is especially useful in **treating outliers** when used in conjunction with `.quantile()`
  [PB] In data wrangling, we generally clip values at the 1st-99th Percentile (or the 5th-95th percentile)

```
In [53]: series_8.head()
Out[53]:
0   -0.23
1    0.55
2    0.77
3    0.18
4    0.76
dtype: float64

In [54]: series_8.head().clip_upper(.50)
Out[54]:
0   -0.23
1    0.50
2    0.50
3    0.18
4    0.50
dtype: float64

In [55]: series_8.head().clip_lower(.50)
Out[55]:
0    0.50
1    0.55
2    0.77
3    0.50
4    0.76
dtype: float64
```

Replacing Values

- `replace` is an effective way to replace source values with target values by suppling a dictionary with the required substitutions.

```
In [60]: fruits = Series(['apples', 'oranges',
'peaches', 'mangoes'])
In [61]: fruits
Out[61]:
0     apples
1    oranges
2    peaches
3    mangoes
dtype: object

In [62]: fruits.replace({'apples':'grapes',
'peaches':'bananas'})
Out[62]:
0     grapes
1    oranges
2    bananas
3    mangoes
dtype: object
```

Checking if values belong to a list

- `isin` produces a boolean by comparing each element of the Series against the provided list. It takes `True` if the element belongs to the list. This boolean may then be used for subsetting the Series.

```
In [64]: fruits.isin(['mangoes', 'oranges'])
Out[64]:
0    False
1     True
2    False
3     True
dtype: bool
```

Finding uniques and their frequency

- `unique, nunique, value_counts` These methods are used to find the array of distinct values in a categorical Series, to count the number of distinct items, and to create a frequency table respectively.

```
In [68]: series_9 = Series(list('abcd' * 3))
In [69]: series_9
Out[69]:
0     a
1     b
2     c
3     d
4     a
5     b
6     c
7     d
8     a
9     b
10    c
11    d
dtype: object

In [70]: series_9.unique()
Out[70]: array(['a', 'b', 'c', 'd'], dtype=object)

In [71]: series_9.nunique()
Out[71]: 4

In [72]: series_9.value_counts()
Out[72]:
d    3
b    3
c    3
a    3
dtype: int64
```

Dealing with Duplicates

- `duplicated` produces a boolean that marks every instance of a value after its first occurrence as True. `drop_duplicates` returns the Series with the duplicates removed. If you want to drop duplicated permanently, pass the `inplace=True` argument.

```
In [88]: series_9.duplicated()
Out[88]:
0     False
1     False
2     False
3     False
4      True
5      True
6      True
7      True
8      True
9      True
10     True
11     True
dtype: bool

In [89]: series_9.drop_duplicates()
Out[89]:
0    a
1    b
2    c
3    d
dtype: object
```

Finding the largest/smallest values

- `idxmax, idxmin, nlargest, nsmallest` As their names imply, these methods help in finding the largest, smallest, n-largest and n-smallest respectively. Note that the index label is returned with these values, and this can be especially helpful in many cases.

```
In [76]: series_10 = Series(np.random.randint(0, 50,
6), index=list('xyzabc'))

In [77]: series_10
Out[77]:
x    21
y     5
z    28
a    12
b    45
c     5
dtype: int64

In [78]: series_10.idxmax()
Out[78]: 'b'

In [79]: series_10.idxmin()
Out[79]: 'y'

In [80]: series_10.nlargest(2)
Out[80]:
b    45
z    28
dtype: int64

In [81]: series_10.nsmallest(2)
Out[81]:
y    5
c    5
dtype: int64
```

Sorting the data

- `sort_values` , `sort_index` help in sorting a Series by values or by index.
  Note: that in order to make the sorting permanent, we need to pass an `inplace=True` argument.

```
In [82]: series_10.sort_values()
Out[82]:
y     5
c     5
a    12
x    21
z    28
b    45
dtype: int64

In [83]: series_10.sort_index()
Out[83]:
a    12
b    45
c     5
x    21
y     5
z    28
dtype: int64
```

Mathematical Summaries

- `mean, median, std, quantile, describe` are mathematical methods employed to find the measures of central tendency for a given set of data points. `quantile` finds the requested percentiles, whereas `describe` produces the summary statistics for the data.
  These functions come in handy when we're exploring data for patterns.

```
# Create a long series with numeric values
In [108]: series_11 = Series(np.random.randn(1000))

In [109]: series_11.head()
Out[109]:
0   -0.808280
1   -0.361064
2    1.098265
3   -0.400104
4   -0.401763
dtype: float64

In [110]: series_11.mean()
Out[110]: 0.010034515870708844

In [111]: series_11.std()
Out[111]: 0.9999153362726881

In [112]: series_11.median()
Out[112]: 0.008293242730166963

In [113]: series_11.quantile([0.10, 0.50, 0.80])
Out[113]:
0.1   -1.290976
0.5    0.008293
0.8    0.854793
dtype: float64

In [114]: series_11.describe()
Out[114]:
count    1000.000000
mean        0.010035
std         0.999915
min        -3.095835
25%        -0.665170
50%         0.008293
75%         0.693991
max         3.762116
dtype: float64
```

Dealing with missing data

- `isnull, notnull` are complementary methods that work on a Series with missing data to produce boolean Series to identify missing or non-missing values respectively.
  Note that both the NumPy `np.nan` and the base Python `None`

type are identified as missing values.

```
In [93]: series_12 = Series([1.12, 3.14, np.nan, 6.02,
2.73, None])

In [94]: series_12
Out[94]:
0    1.12
1    3.14
2     NaN
3    6.02
4    2.73
5     NaN
dtype: float64

In [95]: series_12.isnull()
Out[95]:
0    False
1    False
2     True
3    False
4    False
5     True
dtype: bool

In [96]: series_12.notnull()
Out[96]:
0     True
1     True
2    False
3     True
4     True
5    False
dtype: bool
```

Missing Values Imputation

- `fillna, ffill and bfill, dropna` This set of Series methods allow us to deal with missing data by choosing to either impute them with a particular value, or by copying the last known value over the missing ones (typically used in time-series analysis.) We may sometimes want to drop the missing data altogether and `dropna` helps us in doing that.

  Note: It is a common practice in data science to replace missing values in a numeric variable by its mean (or median if the data is

skewed) and in categorical variables with its mode.

```
In [118]: series_12 = Series([1.12, 3.14, np.nan, 6.02,
2.73, None])

In [119]: series_12.fillna(0)
Out[119]:
0    1.12
1    3.14
2    0.00
3    6.02
4    2.73
5    0.00
dtype: float64

In [120]: series_12.fillna(series_12.mean())
Out[120]:
0    1.1200
1    3.1400
2    3.2525
3    6.0200
4    2.7300
5    3.2525
dtype: float64

In [121]: series_12.ffill()
Out[121]:
0    1.12
1    3.14
2    3.14
3    6.02
4    2.73
5    2.73
dtype: float64

In [122]: series_12.dropna()
Out[122]:
0    1.12
1    3.14
3    6.02
4    2.73
dtype: float64
```

Apply a function to each element

- `map` is perhaps the **most important** of all Series methods. It

takes a general-purpose or user-defined function and applies it to each value in the Series. Combined with base Python's lambda functions, it can be an incredibly powerful tool in transforming a given Series.

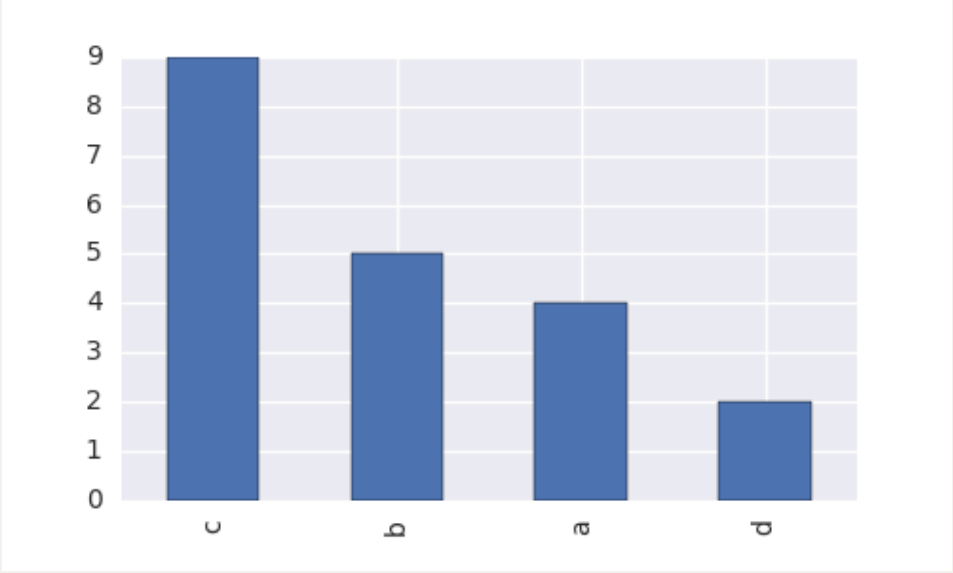
Some of you should've recalled by now that this sounds like the `map` function for List objects in Base Python. The `.map()` method can be understood as a wrapper around that function.

```
# Let's say we have a list of names stored in a Series
In [125]: series_13 = Series(['Dave Smith', 'Jane Doe',
'Carl James', 'Jim Hunt'])

In [126]: series_13
Out[126]:
0    Dave Smith
1      Jane Doe
2    Carl James
3      Jim Hunt
dtype: object

# Find the length of each name
In [126]: series_13.map(lambda x: len(x))
Out[126]:
0    10
1     8
2    10
3     8
dtype: int64

# Find the initials
In [127]: series_13.map(lambda x: '.'.join([i[0] for i
in x.split(' ')]))
Out[127]:
0    D.S
1    J.D
2    C.J
3    J.H
dtype: object
```

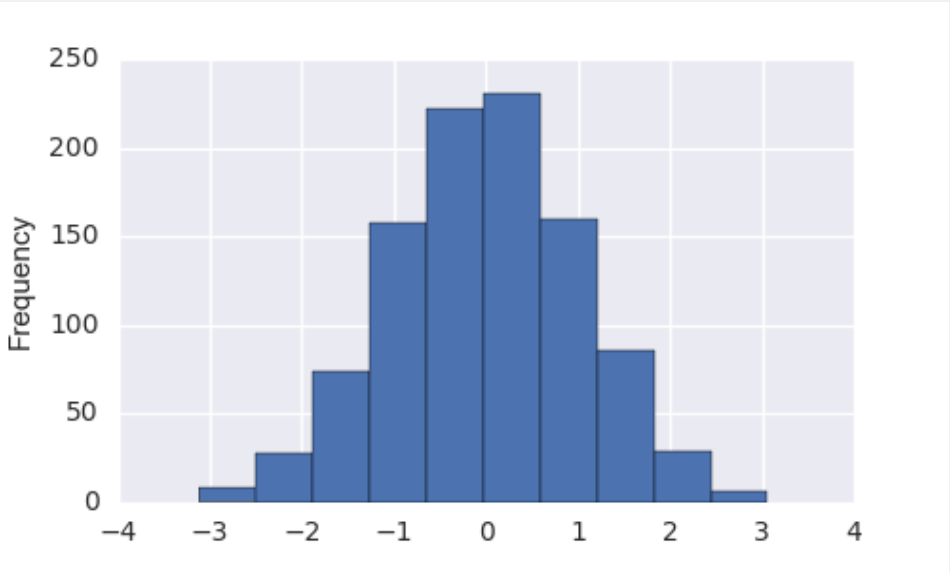
Visualize the Data

- The `plot` method is a gateway to a treasure trove of potential visualizations like histograms, bar charts, scatterplots, boxplots and more. As examples, we will visualize the histogram of a numeric variable, and the bar chart for a categorical series.

```
# Create a categorical series
In [148]: series_14 = Series(list('a' * 3) + list('b' * 5) + list('c' * 9) + list('d' * 2))

In [149]: series_14.head()
Out[149]:
0    a
1    a
2    a
3    b
4    b
dtype: object

In [150]: series_14.value_counts().plot.bar()
```




```
In [151]: series_15 = Series(np.random.randn(1000))

In [151]: series_15.head()
Out[151]:
0    1.796526
1    0.323100
2   -1.747839
3   -0.435137
4    0.182139
dtype: float64

In [152]: series_15.plot.hist()
```



There are extensive customiztions that we can make to the aesthetics of these plots, but we will explore those features of `plot` in more detail in the next chapter on Visualizing data.

---

2. DataFrame

It is 2-dimensional table-like mutable data structure. It is fundamentally different from NumPy 2D arrays in that here each column can be a different `dtype`.

- Has both a row and column index for
    - Fast lookups
    - Data alignment and joins
- Is operationally identical tothe R *data.frame*
- Can contain columns of different data types
- Can be thought of a dict of Series objects.
- Has a number of associated methods that make common data wrangling tasks very straightforward

2.1 Creating a DataFrame

One of the most common ways of creating a dataframe is from a dictionary of arrays or lists or from NumPy 2D arrays. Here's the syntax,

```
DataFrame(data=, index=, columns=)
```

As was the case with Series, if the `index` and the `columns` parameters are not specified, default numeric sequences running from 0 to N-1 will be used.

From a 2D Array

First we create a DataFrame using all defaults, and then another where we pass index and column labels.

```
In [152]: df_1 = DataFrame(np.arange(20, 32).reshape(3,
4))

In [153]: df_1
Out[153]:
    0   1   2   3
0  20  21  22  23
1  24  25  26  27
2  28  29  30  31

In [155]: df_2 = DataFrame(data = np.arange(20,
32).reshape(3, 4),
                           columns = list('WXYZ'),
                           index = list('ABC'))
In [156]: df_2
Out[156]:
    W   X   Y   Z
A  20  21  22  23
B  24  25  26  27
C  28  29  30  31
```

Using a Dict of equal length Lists

The keys of the dictionary will be used as column names, the values will form the data in the table. We can optionally provide a list of strings to be used as the index (or *row labels*.)

```
In [157]: dict_1 = {'ints': np.arange(5),
     ...:            'floats': np.arange(0.1, 0.6,
0.1),
     ...:           'strings': list('abcde')}

In [158]: df3 = DataFrame(dict_1, index=list('vwxyz'))

In [159]: df3
Out[159]:
   floats  ints strings
v     0.1     0       a
w     0.2     1       b
x     0.3     2       c
y     0.4     3       d
z     0.5     4       e
```

DataFrame Attributes

Some of the most commonly used ones are – `index, columns, dtypes, shape, info`

```
# Get row labels
In [161]: df3.index
Out[161]: Index([u'v', u'w', u'x', u'y', u'z'],
dtype='object')

# Get column labels
In [162]: df3.columns
Out[162]: Index([u'floats', u'ints', u'strings'],
dtype='object')

# Get data types for each column
In [163]: df3.dtypes
Out[163]:
floats     float64
ints         int64
strings     object
dtype: object

# Get number of rows, columns
In [164]: df3.shape
Out[164]: (5, 3)

# Get overview of the dataset
In [165]: df3.info()
<class 'pandas.core.frame.DataFrame'>
Index: 5 entries, v to z
Data columns (total 3 columns):
floats     5 non-null float64
ints       5 non-null int64
strings    5 non-null object
dtypes: float64(1), int64(1), object(1)
memory usage: 160.0+ bytes
```

Subsetting DataFrames

Pandas allows us to subset DataFrames in a variety of ways to extract from a given dataset

- a value
- a row or a column (returns a Series)
- multiple rows/columns (returns a subset of the dataframe)

Let's look at a few examples of these

Selecting a single Row/Column

For a DataFrame, basic indexing selects the columns. An individual column can be retrieved as a Series using (a) the square bracket accessor (b) the dot accessor, or (c) one of the accessors - `loc, iloc, ix` etc.

Syntax

```
df['column_name']
df.column_name

df.loc[:, 'column_name']
df.iloc[:, column_number]
```

Examples

```
# The DataFrame
In [169]: df3
Out[169]:
   floats  ints strings
v     0.1     0       a
w     0.2     1       b
x     0.3     2       c
y     0.4     3       d
z     0.5     4       e

# Using a column label with the square bracket accessor
In [170]: df3['floats']
Out[170]:
v    0.1
w    0.2
x    0.3
y    0.4
z    0.5
Name: floats, dtype: float64

# Using a column label with the dot accessor
In [171]: df3.ints
Out[171]:
v    0
w    1
x    2
y    3
z    4
Name: ints, dtype: int64
```

```
# Using .loc and a column label
In [172]: df3.loc[:, 'strings']
Out[172]:
v    a
w    b
x    c
y    d
z    e
Name: strings, dtype: object

# Using .iloc and a column position
In [173]: df3.iloc[:, 0]
Out[173]:
v    0.1
w    0.2
x    0.3
y    0.4
z    0.5
Name: floats, dtype: float64

# Using .loc and a row label
In [174]: df3.loc['v']
Out[174]:
floats     0.1
ints         0
strings      a
Name: v, dtype: object

# Using .iloc and a row position
In [175]: df3.iloc[3]
Out[175]:
floats     0.4
ints         3
strings      d
Name: y, dtype: object
```

Selecting 2 or more rows/columns

This can be accomplished by

- passing a list of column labels to the double square bracket accessor like `[[list-of-columns]]`
- passing a list or slice of row/column labels/positions to `loc, iloc, ix`
- passing a boolean series to `loc, iloc, ix` for selecting particular rows/columns

Each time we subset 2 or more rows/columns from a DataFrame, the result will be a DataFrame.

Syntax for `loc, iloc, ix`

```
# loc is label based
df.loc[list/slice_of_row_labels,
list/slice_of_column_labels]
df.loc[boolean_for_rows, boolean_for_columns]

# iloc is integer based
df.iloc[row-positions, column-positions]

# ix is a mix of the two (takes both labels and/or
positions)
df.ix[row_positions_or_labels,
column_position_or_labels]
```

Examples

```
# Create a new DataFrame
In [183]: df4 = DataFrame(np.random.randint(0, 100,
25).reshape(5, 5),
                          index=list('ABCDE'),
columns=list('PQRST'))
In [184]: df4
Out[184]:
    P   Q   R   S   T
A  42  22   5  10  28
B  17  85  17  26  20
C  48  53   3  35  79
D  80  29   0  24  85
E  71  66  30  31  32

# Subset multiple columns using [[]]
In [185]: df4[['R', 'S', 'T']]
Out[185]:
    R   S   T
A   5  10  28
B  17  26  20
C   3  35  79
D   0  24  85
E  30  31  32

# Subset multiple columns using .loc
In [187]: df4.loc[:, ['R', 'S', 'T']]
```

```
# or df4.loc[:, ['R':'T']]
Out[187]:
    R   S   T
A   5  10  28
B  17  26  20
C   3  35  79
D   0  24  85
E  30  31  32

# Subset multiple columns using .iloc
In [188]: df4.iloc[:, 2:]
Out[188]:
    R   S   T
A   5  10  28
B  17  26  20
C   3  35  79
D   0  24  85
E  30  31  32

# Select multiple rows using []
In [186]: df4['C':'E']
Out[186]:
    P   Q   R   S   T
C  48  53   3  35  79
D  80  29   0  24  85
E  71  66  30  31  32

# Subset multiple rows using .loc
In [189]: df4.loc['B':'D', :]
# or df4.loc[['B', 'C', 'D'], :]
Out[189]:
    P   Q   R   S   T
B  17  85  17  26  20
C  48  53   3  35  79
D  80  29   0  24  85

# Subset multiple rows using .iloc
In [191]: df4.iloc[2:4, :]
Out[191]:
    P   Q  R   S   T
C  48  53  3  35  79
D  80  29  0  24  85

# Mixed subsetting with ix
In [194]: df4.ix[2:4, 'P':'R']
Out[194]:
    P   Q  R
```

```
C  48  53  3
D  80  29  0

In [195]: df4.ix[3:, ['S', 'T']]
Out[195]:
    S   T
D  24  85
E  31  32
```

For a beginner, so many ways of subsetting data may seem intimidating at first and confusing at worst. The prudent thing to do here would be to pick your favorite method of subsetting data from a DataFrame, and stick to it. If you find your more comfortable using `ix` over `loc, iloc`, by all means, use that method. Remember that these are just *tools* and your focus should be more on the analysis and less on the selection of the right tool for the job.

> *Thus, any in-place modifications to the Series will be reflected in the original DataFrame.*
> *The column can be explicitly copied using the Series `copy` method*
> ***[NOTE]** The columns returned when indexing a DataFrame is a **view** on the underlying data, **not a copy***

Adding/Removing/Renaming Columns or Rows

The DataFrame methods `assign, drop` and `rename` come in handy for these tasks.

- New variables may be created by simply using a column label that doesn't exist in our data with the square bracket(or the `loc`) accessor. This method permanently adds a new column to the data.
  `assign` is for creating new variables on the fly, or for deriving new columns from existing ones. This method returns a copy of the DataFrame, so you should overwrite the original if you want to retain the created column in subsequent operations. Note the peculiar syntax in the code box below (lambda functions are required.)

```
# method for creating permanent new columns
In [205]: df4['U'] = df4['P'] + df4['Q']
In [206]: df4
Out[206]:
    P   Q   R   S   T    U
A  42  22   5  10  28   64
B  17  85  17  26  20  102
C  48  53   3  35  79  101
D  80  29   0  24  85  109
E  71  66  30  31  32  137

# assign for creating columns on the fly
In [207]: df4.assign(U = lambda x: x['P'] + x['Q'])
Out[207]:
    P   Q   R   S   T    U
A  42  22   5  10  28   64
B  17  85  17  26  20  102
C  48  53   3  35  79  101
D  80  29   0  24  85  109
E  71  66  30  31  32  137
```

- `drop` returns a copy of the DataFrame after deleting the rows and columns specified (as a list of index or column labels). The `axis=` parameter controls which axis (row or column) we want to drop the Series from. The `inplace=` parameter decides whether the change must be made permanent.
  Also see `dropna`, which helps us get rid of rows with missing data.

```
# The DataFrame
In [196]: df4
Out[196]:
    P   Q   R   S   T
A  42  22   5  10  28
B  17  85  17  26  20
C  48  53   3  35  79
D  80  29   0  24  85
E  71  66  30  31  32

# Dropping a single row (axis=0 is set by default)
In [197]: df4.drop('A')
Out[197]:
    P   Q   R   S   T
B  17  85  17  26  20
C  48  53   3  35  79
D  80  29   0  24  85
E  71  66  30  31  32

# We can drop multiple columns by providing a list of
labels
In [199]: df4.drop(['A', 'E'])
Out[199]:
    P   Q   R   S   T
B  17  85  17  26  20
C  48  53   3  35  79
D  80  29   0  24  85

# This has the same effect; axis=0 is default (can be
omitted)
In [200]: df4.drop(['A', 'E'], axis=0)
Out[200]:
    P   Q   R   S   T
B  17  85  17  26  20
C  48  53   3  35  79
D  80  29   0  24  85

# Pass axis=1 to drop columns
In [201]: df4.drop(['P', 'R', 'T'], axis=1)
Out[201]:
    Q   S
A  22  10
B  85  26
C  53  35
D  29  24
E  66  31
```

- `rename` takes a DataFrame as input and a dictionary that maps old names to new names for columns. This method is particularly useful right in the beginning of data analysis, as sometimes when we get data we find that the column names are all messed up (have spaces or unwanted characters in them.)

  [Pro Tip] Use dictionary comprehensions to create the substitution dict.

```
In [203]: df4.rename(columns={'P':'P_new',
'R':'R_new'})
Out[203]:
   P_new   Q  R_new   S   T
A     42  22      5  10  28
B     17  85     17  26  20
C     48  53      3  35  79
D     80  29      0  24  85
E     71  66     30  31  32
```

Math/Stats Operations

These sets of methods is one of the major reasons why so many people love pandas for data wrangling. By default, when you call a mathematical operation (like `sum`) or a statistical operation (like `std`) on a DataFrame the results are produced for *all numeric columns*. Other languages like R required you to either use an *apply* function or to use one of their specialized functions like `colMeans, rowMeans`. Pandas, however, requires that you only pass the `axis=` parameter to control whether math/stat summaries should be produced for rows or columns. These methods also take a `skipna=` parameter that signals whether to exclude missing data (`True` by default.)

```
# Column Sums
In [208]: df4.sum(axis=0)
Out[208]:
P    258
Q    255
R     55
S    126
T    244
U    513
dtype: int64

# Row Sums
In [209]: df4.sum(axis=1)
Out[209]:
A    171
B    267
C    319
D    327
E    367
dtype: int64
```

Try the above with other methods like `mean, std, var` to produce statistical summaries of your data. Below is a list of all math/stat methods available to objects of the DataFrame class.

Function	Description
count	Number of non-null observations
sum	Sum of values
mean	Mean of values
mad	Mean absolute deviation
median	Arithmetic median of values
min	Minimum
max	Maximum
mode	Mode
abs	Absolute Value
prod	Product of values
std	Bessel-corrected sample standard deviation
var	Unbiased variance
sem	Standard error of the mean
skew	Sample skewness (3rd moment)
kurt	Sample kurtosis (4th moment)
quantile	Sample quantile (value at %)
cumsum	Cumulative sum
cumprod	Cumulative product
cummax	Cumulative maximum
cummin	Cumulative minimum

The `describe()` method

One method, however, stands apart from the rest in its usefulness. The `.describe()` method applied to a DataFrame wil produce summary statistics for **all numeric variables** in the data and return the result in a neat DataFrame. Note that here too, NAs are excluded by default.

```
In [213]: df4.describe()
Out[213]:
              P       Q       R       S       T       U
count    5.00    5.00    5.00    5.00    5.00    5.00
mean    51.60   51.00   11.00   25.20   48.80  102.60
std     24.93   26.03   12.43    9.52   30.69   26.06
min     17.00   22.00    0.00   10.00   20.00   64.00
25%     42.00   29.00    3.00   24.00   28.00  101.00
50%     48.00   53.00    5.00   26.00   32.00  102.00
75%     71.00   66.00   17.00   31.00   79.00  109.00
max     80.00   85.00   30.00   35.00   85.00  137.00
```

We can specift exactly which percentiles to evaluate (but the median will always be printed by default.)

```
In [215]: df4.describe(percentiles=[.01, .05, .95,
.99]).round(2)
Out[215]:
              P      Q      R      S      T       U
count      5.00   5.00   5.00   5.00   5.00    5.00
mean      51.60  51.00  11.00  25.20  48.80  102.60
std       24.93  26.03  12.43   9.52  30.69   26.06
min       17.00  22.00   0.00  10.00  20.00   64.00
1%        18.00  22.28   0.12  10.56  20.32   65.48
5%        22.00  23.40   0.60  12.80  21.60   71.40
50%       48.00  53.00   5.00  26.00  32.00  102.00
95%       78.20  81.20  27.40  34.20  83.80  131.40
99%       79.64  84.24  29.48  34.84  84.76  135.88
max       80.00  85.00  30.00  35.00  85.00  137.00
```

For categorical data, `describe()` will give a simple summary of the number of unique values and most frequently occurring values

```
In [217]: s = pd.Series(['a', 'a', 'b', 'b', 'a', 'a',
np.nan, 'c', 'd', 'a'])

In [218]: s.describe()
Out[218]:
count     9
unique    4
top       a
freq      5
dtype: object
```

We can pass the `include=` parameter to `describe` to control whether the summaries are printed for numeric or categorical variables by default.

```
# Our DataFrame with mixed types
In [223]: df3
Out[223]:
   floats  ints strings
v     0.1     0       a
w     0.2     1       b
x     0.3     2       c
y     0.4     3       d
z     0.5     4       e

# Default behavior
# same as: df3.describe(include=['number'])
In [224]: df3.describe()
```

```
Out[224]:
         floats      ints
count  5.000000  5.000000
mean   0.300000  2.000000
std    0.158114  1.581139
min    0.100000  0.000000
25%    0.200000  1.000000
50%    0.300000  2.000000
75%    0.400000  3.000000
max    0.500000  4.000000

# To get summaries for categorical variables only
In [225]: df3.describe(include=['object'])
Out[225]:
       strings
count        5
unique       5
top          d
freq         1

# Get summaries for all variables
In [228]: df3.describe(include='all')
Out[228]:
          floats      ints strings
count   5.000000  5.000000       5
unique       NaN       NaN       5
top          NaN       NaN       d
freq         NaN       NaN       1
mean    0.300000  2.000000     NaN
std     0.158114  1.581139     NaN
min     0.100000  0.000000     NaN
25%     0.200000  1.000000     NaN
50%     0.300000  2.000000     NaN
75%     0.400000  3.000000     NaN
max     0.500000  4.000000      Na
```

Handling Missing Values

By *missing* data we simply mean **NULL** or *not present for whatever reason*. Many phenomena could give rise to missing data but mostly it is just a matter of either the data existed and was not collected or it never existed.

Pandas treats the NumPy `np.nan` and the Python `None` as missing values. The approach for handling missing values in DataFrames is the same as that for Series (a Dataframe is afterall just a list of Series objects.)

- These can be detected in a Series or DataFrame using `isnull, notnull` which return booleans.
- To filter out missing data from a Series/DataFrame, or to remove rows (default action) or columns with missing data in a DataFrame, we use `dropna` with the `axis=` and `inplace=` parameters.
- Missing Value imputation is done using the `fillna` method (along with options like `ffill, bfill`)

> *[Note] Prior to version v0.10.0* `inf` *and* `-inf` *were also considered to be "null" in computations. This is no longer the case by default; use the* `mode.use_inf_as_null` *option to recover it.*

```
# Create some mising data
In [231]: df4.ix[::2] = np.nan
In [232]: df4
Out[232]:
      P     Q     R     S     T      U
A   NaN   NaN   NaN   NaN   NaN    NaN
B  17.0  85.0  17.0  26.0  20.0  102.0
C   NaN   NaN   NaN   NaN   NaN    NaN
D  80.0  29.0   0.0  24.0  85.0  109.0
E   NaN   NaN   NaN   NaN   NaN    NaN

# Detect Missing data
In [233]: df4.isnull()
Out[233]:
       P      Q      R      S      T      U
A   True   True   True   True   True   True
B  False  False  False  False  False  False
C   True   True   True   True   True   True
D  False  False  False  False  False  False
E   True   True   True   True   True   True

# Replace missings with 0 (manually)
In [241]: df4[df4.isnull()] = 0
In [242]: df4
Out[242]:
    P   Q   R   S   T    U
A   0   0   0   0   0    0
B  17  85  17  26  20  102
C   0   0   0   0   0    0
D  80  29   0  24  85  109
E   0   0   0   0   0    0

# Replace missings with 0 (using fillna)
In [243]: df4.fillna(0)
```

```
Out[243]:
      P     Q     R     S     T      U
A   0.0   0.0   0.0   0.0   0.0    0.0
B  17.0  85.0  17.0  26.0  20.0  102.0
C   0.0   0.0   0.0   0.0   0.0    0.0
D  80.0  29.0   0.0  24.0  85.0  109.0
E   0.0   0.0   0.0   0.0   0.0    0.0

# Drop rows with missing data
In [247]: df4.dropna()
Out[247]:
    P   Q   R   S   T    U
B  17  85  17  26  20  102
D  80  29   0  24  85  109
```

Sorting Data

Sorting data is a basic task that allows us to figure out if a given variable has outliers by looking at the values at its extremes. Both sort functions in Pandas take an `ascending=` parameter to control the nature of the sort. By default, it takes `True` so if you want to get the Series/DataFrame sorted in descending order, pass `ascending=False`

- For Reordering rows or columns we use `sort_index()`
- For Sorting on column values use `sort_values()` which takes a `by=` parameter through which we may specify the column(s) on which we want to sort the data.

```
In [249]: df5 =
DataFrame(np.random.randn(9).reshape(3,3),
                          index=list('cba'),
                          columns=list('prq'))
In [249]: df5
Out[249]:
          p         r         q
c -0.091751 -0.617904  0.267026
b -0.156745 -0.182804 -0.255996
a  1.864256  0.944659  2.858450

# without arguments, sort_index() will sort the index
(rows) of the DataFrame
In [250]: df5.sort_index()
Out[250]:
          p         r         q
a  1.864256  0.944659  2.858450
b -0.156745 -0.182804 -0.255996
c -0.091751 -0.617904  0.267026

# To sort column names
In [251]: df5.sort_index(axis=1)
Out[251]:
          p         q         r
c -0.091751  0.267026 -0.617904
b -0.156745 -0.255996 -0.182804
a  1.864256  2.858450  0.944659

# Sort the data by the values of a column
In [252]: df5.sort_values(by=['p'])
Out[252]:
          p         r         q
b -0.156745 -0.182804 -0.255996
c -0.091751 -0.617904  0.267026
a  1.864256  0.944659  2.858450

# Sort the data by the values of 2 columns
In [253]: df5.sort_values(by=['p', 'r'],
ascending=False)
Out[253]:
          p         r         q
a  1.864256  0.944659  2.858450
c -0.091751 -0.617904  0.267026
b -0.156745 -0.182804 -0.255996
```

Handling Duplicates

The methods `duplicated, drop_duplicates` help us in identifying rows that are duplicates of other rows, and to ignore those rows from the data.

```
# Create a duplicate row in the data
In [260]: df5.loc['x', :] = df5.loc['c', :]
In [261]: df5
Out[261]:
          p         r         q
c -0.091751 -0.617904  0.267026
b -0.156745 -0.182804 -0.255996
a  1.864256  0.944659  2.858450
x -0.091751 -0.617904  0.267026

# Detect duplicates
In [262]: df5.duplicated()
Out[262]:
c    False
b    False
a    False
x     True
dtype: bool

# Ignore duplicates
In [263]: df5.drop_duplicates()
Out[263]:
          p         r         q
c -0.091751 -0.617904  0.267026
b -0.156745 -0.182804 -0.255996
a  1.864256  0.944659  2.858450
```

Binning Data with `cut` and `qcut`

Binning is the process of ***converting numeric variables to categoricals***. This technique finds widespread use in data science when we have numeric variables (such as Age, Salary) and we want to convert them into ranges (such a Age Groups, Salary Groups.) We might also want to convert a variable into quantile groups (deciling is a popular example.)

The `pd.cut()` and `pd.qcut()` functions are used for this purpose.

- `cut` discretizes variables by following the underlying distribution. For example, discretizing a normally-distributed variable would have more cases in the middle bins than the bins at the extremes.
    - With `cut`, each bin will have the same width, but they may not have the same number of records.

- `qcut` discretizes variables into equal-sized buckets by choosing bin thresholds accordingly. So if your data has 100 values, and you specify bins = 5, there will be 20 values in each bin.
    - With `qcut`, each bin will have the same number of records but they may not have the same width.

They take as arguments the following;

- `var`, the continuous variable to discretize
- `bins`, specified as a number (equal sized bins will be computed based on min/max) or a list of bin edges
- `right=True`, a boolean to include the edge or not
- `labels=`, for naming the bins
- `precision=`

Example

```
# Create an array (normally distributed data)
In [ ]: num_var = (np.random.randn(1000) *
100).astype(int)
In [ ]: num_var[:5]
Out[ ]: array([-114,   58,    0, -122,  138])

# Bins produced by pd.cut
In [ ]: pd.cut(num_var, 10)[:5]
Out[ ]:
[(-142.2, -73.6], (-5, 63.6], (-5, 63.6], (-142.2,
-73.6], (132.2, 200.8]]

# Bins produced by pd.qcut

In [ ]: pd.qcut(num_var, 10)[:5]
Out[ ]:
[(-127, -83.2], (49, 86.2], (-2, 23], (-127, -83.2],
(132.2, 338]]
```

Observe how bins created by `cut` are roughly equi-spaced. Running `value_counts()` on these binned variables reveals an interesting phenomenon.

```
# No. of values in cut bins follow the underlying
distribution
In [ ]: pd.cut(num_var, 10).value_counts()
Out[ ]:
(-348.686, -279.4]      3
(-279.4, -210.8]       13
(-210.8, -142.2]       63
(-142.2, -73.6]       142
(-73.6, -5]           265
(-5, 63.6]            260
(63.6, 132.2]         154
(132.2, 200.8]         78
(200.8, 269.4]         20
(269.4, 338]            2
dtype: int64

# No. of values in qcut bins follow the uniform
distribution
In [ ]: pd.qcut(num_var, 10).value_counts()
Out[ ]:
[-348, -127]      101
(-127, -83.2]      99
(-83.2, -49.3]    100
(-49.3, -25]      101
(-25, -2]         101
(-2, 23]          103
(23, 49]           96
(49, 86.2]         99
(86.2, 132.2]     100
(132.2, 338]      100
dtype: int64
```

If you wish to manually create the bins, you may pass the bin boundaries as a list to the `bins=` argument of `cut`

```
In []: pd.cut(num_var, bins=range(-350, 350,
50)).value_counts()
Out[]:
(-350, -300]      2
(-300, -250]      4
(-250, -200]     19
(-200, -150]     44
(-150, -100]     94
(-100, -50]     137
(-50, 0]        218
(0, 50]         188
(50, 100]       129
(100, 150]       97
(150, 200]       46
(200, 250]       17
(250, 300]        4
dtype: int64
```

Note: If your data has missing values (or a lot of duplicate values - typically the case with skewed discrete distributions), it will cause `qcut` to fail as it would not be able to detect unqiue bin edges.

Creating Dummies from Categorical data

A dummy variables is one that takes the value 0 or 1 to indicate the absence or presence of a particular level of a categorical variable. Typically, for a categorical variable of length N and k levels, we can derive k dummy variables, each of length N (as a DataFrame.) These find extensive applications in statistical learning, most notably in regression analysis and text mining. Many machine learning models require the user to code all categorical data into dummy variables as a preprocessing step.

As a simple example, suppose `Gender` is one of the qualitative explanatory variables relevant to a data problem. Then, `female` and `male` would be the levels of this variable. We can create two dummy variables from this variable (let's call them `dummy_female` and `dummy_male`) where the former would take the value 1 whenever Gender takes the value female and vice-versa.

Creating Dummies Manually

```
# Create a toy dataset with one numeric and one
categorical variable
```

```
In []:
df_G = DataFrame({'key': list('bbacccb'),
                  'val': np.random.randn(7) }).round(2)
df_G

Out[]:
  key   val
0   b -1.63
1   b -0.10
2   a  1.30
3   c  0.09
4   c  0.37
5   c  1.86
6   b  0.93

# Dummy for Level 'a'
In []: (df_G['key'] == 'a').astype(int)
Out[]:
0    0
1    0
2    1
3    0
4    0
5    0
6    0
Name: key, dtype: int64

# Dummy for level 'b'
In []: (df_G['key'] == 'b').astype(int)
Out[]:
0    1
1    1
2    0
3    0
4    0
5    0
6    1
Name: key, dtype: int64

# Dummies for all 3 levels
In []:
DataFrame({'key': df_G['key']}).assign(dummy_a = lambda
x: (x['key'] == 'a').astype(int),
                                       dummy_b = lambda
x: (x['key'] == 'b').astype(int),
                                       dummy_c = lambda
x: (x['key'] == 'c').astype(int))
```

```
Out[]:
  key  dummy_a  dummy_b  dummy_c
0   b        0        1        0
1   b        0        1        0
2   a        1        0        0
3   c        0        0        1
4   c        0        0        1
5   c        0        0        1
6   b        0        1        0
```

Note how the dummy corresponding each level of `key` takes 1 whenever the corresponding level in `key` is 'a', 'b' or 'c'. Now that you've seen how dummies work, let's look at how we can do this automatically.

Creating Dummies with `pd.get_dummies`

This function takes as input a categorical variable, and a `prefix=` argument for supplying names to created variables.

```
# Creating Dummies
In [345]: pd.get_dummies(df_G['key'])
Out[345]:
     a    b    c
0  0.0  1.0  0.0
1  0.0  1.0  0.0
2  1.0  0.0  0.0
3  0.0  0.0  1.0
4  0.0  0.0  1.0
5  0.0  0.0  1.0
6  0.0  1.0  0.0

# Joining the dummies back to the original data
In [346]: df_G.join(pd.get_dummies(df_G['key']))
Out[346]:
  key   val    a    b    c
0   b -1.63  0.0  1.0  0.0
1   b -0.10  0.0  1.0  0.0
2   a  1.30  1.0  0.0  0.0
3   c  0.09  0.0  0.0  1.0
4   c  0.37  0.0  0.0  1.0
5   c  1.86  0.0  0.0  1.0
6   b  0.93  0.0  1.0  0.0
```

The Dummy Variable Trap

If we build dummy variables for **every** level of a categorical variable, it leads to a situation where one or more of these dummies would be **highly correlated** leading to problems of multicollinearity. It is thus advised to create (k-1) dummies for a categorical variable with k levels.

Keeping this in mind, we can summarize the steps of dummification as

- Identify the categorical variables you want to create dummies from
- Create the dummies for n-1 categories for each
- Join the dummies in to the original table
- Drop the categorical variables in step 1.

```
# Create dummies, drop the categorical variable, and
one of the dummies to avoid the trap
In [ ]: (df_G.join(pd.get_dummies(df_G['key'],
prefix='dummy'))
            .drop('key', axis=1)
            .drop('dummy_c', axis=1))

Out[ ]:
     val  dummy_a  dummy_b
0 -1.63       0.0       1.0
1 -0.10       0.0       1.0
2  1.30       1.0       0.0
3  0.09       0.0       0.0
4  0.37       0.0       0.0
5  1.86       0.0       0.0
6  0.93       0.0       1.0
```

Reshaping Data

Usually, for convenience and for storage efficiencies, data in relational databases is stored in the so called `long` or `stacked` format. This means that there are fewer columns and more rows, with label duplication in keys.

Example of `long` data

```
          date variable     value
0   2000-01-03        A  0.469112
1   2000-01-04        A -0.282863
2   2000-01-05        A -1.509059
3   2000-01-03        B -1.135632
4   2000-01-04        B  1.212112
5   2000-01-05        B -0.173215
6   2000-01-03        C  0.119209
7   2000-01-04        C -1.044236
8   2000-01-05        C -0.861849
9   2000-01-03        D -2.104569
10  2000-01-04        D -0.494929
11  2000-01-05        D  1.071804
```

But for certain kinds of analysis (especially time series analysis), we might prefer to have the data in the `wide` format (more columns, unique labels in keys). Other applications would include cases where we have data at the transaction level for every customer (so multiple rows per customer) and we want to reduce it to one row per customer (typical input to most machine learning algoritms.) Reshaping functions also come in handy when creating visualizations.

Example of `wide` data (same DataFrame as above)

```
variable           A         B         C         D
date
2000-01-03  0.469112 -1.135632  0.119209 -2.104569
2000-01-04 -0.282863  1.212112 -1.044236 -0.494929
2000-01-05 -1.509059 -0.173215 -0.861849  1.071804
```

As should be evident from the examples above, both `long` and `wide` are just different representations of the same data!

`stack()` and `unstack()`

For pandas dataframes with hierarchical indices, `stack` and `unstack` provide a convenient way to reshape the data from wide-to-long or long-to-wide formats.

- `stack` pivots the columns into rows (wide to long)
- `unstack` pivots rows into columns (long to wide.) It works with DataFrames that have a `hierarchical index`

Example `stack`

```
# Create a toy dataset in the LONG format
# Here, we use set_index to create a hierarchical index
(combination of two columns)
In []: df_long = (DataFrame({'group': list('PQR' * 4),
                             'item': list('ABCD' *
3),
                             'status':
(np.random.randn(12).round(2))})
                    .set_index(['group', 'item']))
In []: df_long
Out[]:
            status
group item
P     A       0.81
Q     B       0.23
R     C      -0.75
P     D      -0.39
Q     A      -0.05
R     B       0.15
P     C       0.96
Q     D      -0.59
R     A      -0.81
P     B      -3.04
Q     C      -0.89
R     D      -0.68

In []: df_long.unstack()
Out[]:
      status
item       A     B     C     D
group
P       0.81 -3.04  0.96 -0.39
Q      -0.05  0.23 -0.89 -0.59
R      -0.81  0.15 -0.75 -0.68
```

Example `unstack`

```
# Create a toy dataset in the WIDE format
In [ ]: df_wide =
(DataFrame(np.random.randn(12).reshape(3, 4).round(2),
                              index=list('ABC'),
                              columns=list('PQRS')))
In [ ]: df_wide
Out[ ]:
      P     Q     R     S
A  1.43  1.11 -1.80  0.24
B  0.19  1.15  1.46  0.83
C -0.75 -0.48 -0.64  1.84

# Use unstack to convert it to LONG
In [363]: df_wide.stack()
Out[363]:
A  P    1.43
   Q    1.11
   R   -1.80
   S    0.24
B  P    0.19
   Q    1.15
   R    1.46
   S    0.83
C  P   -0.75
   Q   -0.48
   R   -0.64
   S    1.84
dtype: float64
```

`pivot()` and `pivot_table()`

The `df.pivot()` method takes the names of columns to be used as row (the `index=` parameter) and column indexes (the `columns=` parameter) and a column to fill in the data as (the `values=` parameter) and converts *(or pivots)* wide data to long data.

Simply put, `pivot` is just a convenient wrapper function that replaces the need to create a hierarchical index using `set_index` and reshaping with `stack`.

Example `pivot`

```
In []: df_long.reset_index(inplace=True)
In []: df_long.pivot(index='group', columns='item',
values='status')

Out[]:
item       A     B     C     D
group
P       0.81 -3.04  0.96 -0.39
Q      -0.05  0.23 -0.89 -0.59
R      -0.81  0.15 -0.75 -0.68
```

The `pivot_table()` function is similar to `pivot`, but

- can work with duplicate indices and
- lets you specify an aggregation function

For those with an understanding how pivot tables work in Excel, the `pivot_table` function in pandas is a very natural way of specifying the same things you would using Excel, ie, a variable for rows, a variable for columns, a variable for the data and an aggregation function like sum, mean or count.

Example -

```
In [ ]: df = (pd.DataFrame({'C1':list(('x' * 4 + 'y'*
4)*2),

'C2':list('abbbaabaabbbaaba'),

'N1':np.random.randn(16)})); df
Out[ ]:
   C1 C2        N1
0   x  a -0.720376
1   x  b -0.717720
2   x  b -0.411926
3   x  b -0.478841
4   y  a  0.338465
5   y  a  1.309893
6   y  b  0.050849
7   y  a  1.739078
8   x  a -1.201153
9   x  b -1.178396
10  x  b -0.583237
11  x  b -0.785945
12  y  a -0.858983
13  y  a  1.641496
14  y  b  0.795528
15  y  a  0.444192

In [ ]: (df.pivot_table(index='C1',
                        columns='C2',
                        values='N1',
                        aggfunc='mean'))
Out[ ]:
C2         a         b
C1
x  -0.960765 -0.692677
y   0.769024  0.423189
```

`apply` and `applymap`

There are often situations in data analysis where you would want to apply an existing (in-built) or a user-defined funtion to values in your dataset. The `apply` and `applymap` methods provide a powerful, yet simple interface for doing this while eliminating the need to write for-loops to iterate over data. Lambda functions are leveraged extensively here.

- `applymap` allows you to apply a function to each element of a DataFrame.

- `apply` allows you to apply a function to rows/columns (controlled via the `axis=` parameter) of a DataFrame

Example: `applymap`

```
In [ ]: df_3 = DataFrame(np.random.randn(20).reshape(4,
5),
                         index=list('abcd'),
                         columns=list('pqrst'))
In [ ]: df_3
Out[ ]:
          p         q         r         s         t
a  1.400444  0.915017 -0.993813  0.153223  1.927246
b -1.099701 -0.401442 -1.357320  1.873588  1.006781
c -1.733929 -0.410928  0.640509  1.219760 -0.008854
d -0.727955 -1.565173  1.084859  2.574694 -0.449346

# Find the square root of each number
In [ ]: df_3.applymap(lambda x: np.sqrt(np.abs(x)))
Out[ ]:
          p         q         r         s         t
a  1.183403  0.956565  0.996902  0.391437  1.388253
b  1.048666  0.633595  1.165041  1.368791  1.003385
c  1.316787  0.641037  0.800318  1.104428  0.094098
d  0.853203  1.251069  1.041566  1.604585  0.67033
```

Example: Using `apply` to standardize variables in a dataset.

> *Standardization is the process of reducing each numeric variable's mean to 0 and standard deviation to 1 by subtracting each value from its column mean and dividing it by the column standard deviation.*

```
# Create a 200 x 5 toy dataframe with all numeric
columns
In [ ]: df_4 =
DataFrame(np.random.randn(1000).reshape(200, 5),
                        columns=['Col_' + str(x) for x
in range(5)]); df_4.head()
Out[ ]:
      Col_0     Col_1     Col_2     Col_3     Col_4
0 -1.157990  0.681349  1.030490  0.014480 -1.114365
1  0.173016  1.080648 -0.265723 -0.219716 -0.555620
2  0.885419 -1.179766 -0.491304 -0.883111 -1.635053
3 -2.011335 -0.115281  0.385405 -1.302427 -0.051955
4  0.984635  0.063711 -1.048197 -0.401136 -1.221760

# Apply the standardization function to each column
In [ ]: df_4_standardized = df_4.apply(lambda x: (x -
x.mean())/x.std())

In [ ]: df_4_standardized.head()
Out[ ]:
      Col_0     Col_1     Col_2     Col_3     Col_4
0 -1.242254  0.716079  0.952361  0.027532 -1.017306
1  0.247038  1.107296 -0.376966 -0.202084 -0.464047
2  1.044162 -1.107369 -0.608310 -0.852506 -1.532882
3 -2.197081 -0.064428  0.290796 -1.263623  0.034673
4  1.155177  0.110941 -1.179430 -0.379956 -1.123646

# Check if standardization was successful
In [ ]: df_4_standardized.mean().round(2)
Out[ ]:
Col_0    0.0
Col_1    0.0
Col_2    0.0
Col_3    0.0
Col_4    0.0
dtype: float64

In [ ]: df_4_standardized.std()
Out[ ]:
Col_0    1.0
Col_1    1.0
Col_2    1.0
Col_3    1.0
Col_4    1.0
dtype: float64
```

Split - Apply - Combine using `groupby`

In Data Analysis workflows, operations like data loading, cleaning and merging are usually followed by **summarizations** using some grouping (categorical) variable(s). This includes evaluating summary statistics over variables, (or groups within variables), within-group transformations, computing pivot-tables and by-group analyses.

Pandas DataFrames have a `.groupby()` method that worksin the same way as the SQL `GROUP BY` aggregations.

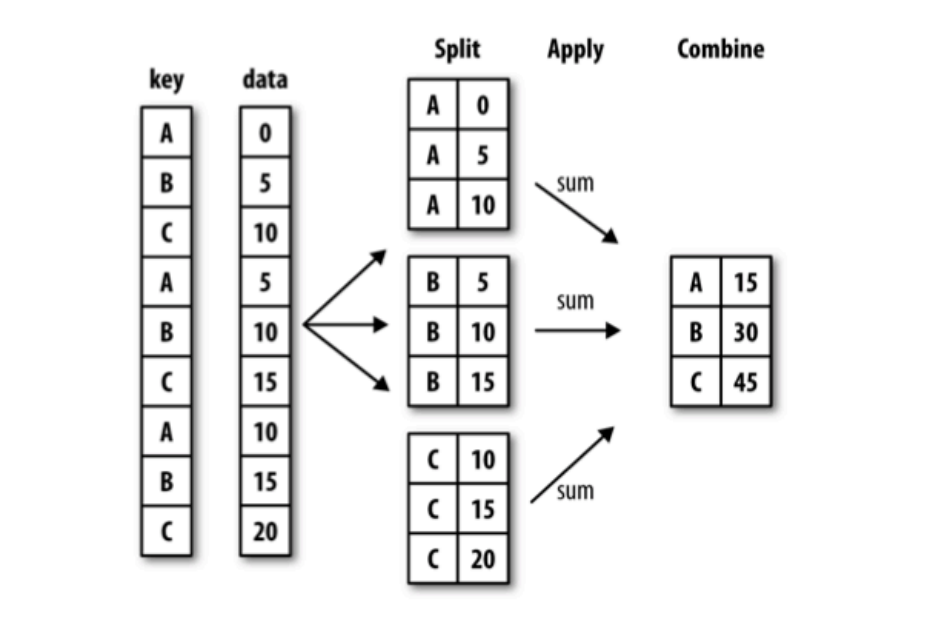
This process involves three steps

- **Splitting** the data into groups based on the levels of a categorical variable. This is generally the simplest step.
- **Applying** a function to each group individually. There are 3 classes of functions we might consider:
    - *Aggregate* – estimate summary statistics (like counts, means) for each group.
      This will reduce the size of the data.
    - *Transform* – within group standardization, imputation using group values.
      The size of the data will not change.
    - *Filter* – ignore rows that belong to a particular group
    - A combination of these three

- **Combining** the results into a Series orDataframe

The image below shows a graphical explanation of this process:
We split by `key`, apply the function `sum` to each group formed and then append the results.

Note that as a result of the aggregation, there will exist one row per level of the categorical variable.



With `pandas`, we can implement this strategy as

- **Split**
    - A DataFrame can be split up byrows(axis=0) or columns(axis=1) into groups.
    - We use `pd.groupby()` to create a `groupby` object
- **Apply**
    - A function is applied to eachgroup using `.agg()` or `.apply()`
- **Combine**
    - The results of applying functionsto groups are put together into an object

[Note] Data types of returnedobjects are handled gracefully by pandas

The `DataFrameGroupBy` object

This object is created by calling the `groupby()` function on a dataframe, passing a list of column names that we wish to use for grouping. These objects,

- have a .size() method, which returns the count of elements in each group.
- can be subsetted using columnnames (or arrays of column names) to select variables for aggregation
- have optimized methods for general aggregation operations like
    - `count, sum`
    - `mean, median, std, var`

    - `first, last`
    - `min, max`

- specialized methods like `.describe()` also apply to these objects

By far, the most important `DataFrameGroupBy` object methods are `.agg()`, `.transform()`, and `.apply()`

Syntax:

```
# Create a groupBy object
gb_obj = my_df.groupby(['col_x'])

# Summarize each group
gb_obj.my_func()
```

Example:

```
# Create a toy dataframe with 2 numeric and 2
categorical columns
In []: df = DataFrame({'k1': list('abcd' * 25),
                        'k2': list('xy' * 25 + 'yx' *
25),
                        'v1': np.random.rand(100),
                        'v2': np.random.rand(100)})

In []: df.head()
Out[]:
  k1 k2        v1        v2
0  a  x  0.037904  0.042694
1  b  y  0.638175  0.277161
2  c  x  0.872149  0.084649
3  d  y  0.317732  0.044503
4  a  x  0.972954  0.890182
```

Grouping by One key

This results in a summarized data frame indexed by levels of the key.

```
# Since k1 has 4 categories, this will return 4 rows
In [ ]: df.groupby('k1').mean()
Out[ ]:
          v1        v2
k1
a   0.521484  0.571119
b   0.521301  0.388422
c   0.492358  0.555235
d   0.508871  0.419638

# Since k2 has 2 categories, this will return 2 rows
In [ ]: df.groupby('k2').sum()
Out[ ]:
           v1         v2
k2
x   24.547462  22.477416
y   26.552891  25.882961
```

Grouping by Two keys

This will result in a summarized dataframe with a hierarchical index.

```
# A dataframe with a hierarchical index formed by a
combination of the levels
In [ ]: df.groupby(['k1', 'k2']).sum()
Out[ ]:
             v1        v2
k1 k2
a  x   6.069234  6.044262
   y   6.967875  8.233720
b  x   6.270785  4.383762
   y   6.761731  5.326794
c  x   5.665012  6.593410
   y   6.643937  7.287471
d  x   6.542430  5.455982
   y   6.179347  5.034976
```

Column-wise aggregations (using optimized statistical methods)

For simple statistical aggregations (of numeric columns of a DataFrame) we can call methods like `mean` and `sum`

```
# Finding the sums of a Series by group
In [ ]: df.groupby('k1')['v1'].sum()
Out[ ]:
k1
a    13.037110
b    13.032516
c    12.308950
d    12.721777
Name: v1, dtype: float64

# Summing all Series of a DataFrame
In [ ]: df.groupby('k2').sum()
Out[ ]:
          v1         v2
k2
x   24.547462  22.477416
y   26.552891  25.882961
```

The `.agg( )` method

When we have a `DataFrameGroupBy` object, we may choose to apply

- one or more functions to one or more columns,
- different functions to individual columns.

The `.agg()` method allows us to easily and flexibly specify these details. It takes as arguments the following –

- *list of function names* to be applied to all selected columns
- *tuples of (colname, function)* to be applied to all selected columns
- *dict of (colname, function)* to be applied to each column

`.agg()` Examples

1. Apply more than 1 function(s) to selected column(s) by passing names of functions as a list

```
# Apply min, mean, max and max to v1 grouped by k1
In []: df.groupby('k1')['v1'].agg(['min', 'mean',
'max'])
Out[]:
          min       mean       max
k1
a    0.037904  0.521484  0.972954
b    0.167095  0.521301  0.902595
c    0.115346  0.492358  0.872149
d    0.127018  0.508871  0.904145

# Apply min and max to all numeric columns of df
grouped by k2
In []: df.groupby('k2')[['v1', 'v2']].agg(['min',
'max'])
Out[]:
           v1                   v2
          min       max       min       max
k2
x    0.037904  0.972954  0.026337  0.986863
y    0.118852  0.897646  0.008764  0.997575
```

2. We can supply names for the columns in the (new) aggregated DataFrame to the agg() method, **in a list of tuples**
   The first element of the 2-tuple should be the **name** and the second element should be the **function**

```
# Provide names for the aggregated columns
In []: df.groupby('k1')[['v1',
'v2']].agg([('smallest','min'), ('largest', 'max')])
Out[]:
           v1                   v2
    smallest   largest  smallest   largest
k1
a    0.037904  0.972954  0.042694  0.986863
b    0.167095  0.902595  0.008764  0.997575
c    0.115346  0.872149  0.016577  0.982617
d    0.127018  0.904145  0.044503  0.949769
```

3. We can supply DataFrame column names and which functions to apply to each, **in a dictionary**

```
# Apply max and min to v1; and mean and sum to v2; all
grouped by k1
In []: df.groupby('k1')[['v1', 'v2']].agg({'v1':
['max', 'min'], 'v2': ['mean', 'sum']})
Out[]:
          v1                   v2
         max       min       mean        sum
k1
a   0.972954  0.037904  0.571119  14.277982
b   0.902595  0.167095  0.388422   9.710556
c   0.872149  0.115346  0.555235  13.880881
d   0.904145  0.127018  0.419638  10.490958
```

The `.apply( )` method

This method takes as arguments the following:

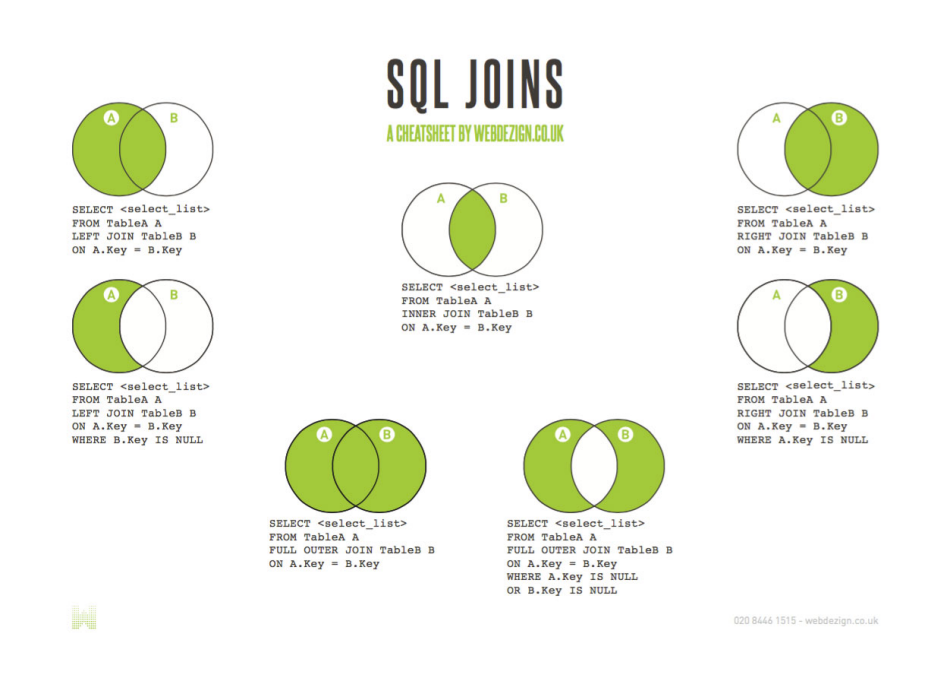
- a general or **user defined function**
- any other **parameters** that the function would take

Remember that any function passed through `apply` should be designed to work on a subset of the DataFrame (smaller tables with the same columns, and all rows corresponding to a particular level of the `groupby` variable)

```
#  Function to Retrieve the top N cases from each group
def topN(data, col, N):
        """
        This function takes as input a DataFrame, a
column name, and a numeric value
        It sorts the DataFrame in descending order by
the values of the column supplied
        Then it returns the Top N records from that
column
        """
        return data.sort_values(by=col,
ascending=False).loc[:, col].head(N)

# Apply the function to groups formed by levels of k2,
return the top 5
In []: df.groupby('k2').apply(topN, col='v1', N=5)
Out[]:
k2
x   4     0.972954
    71    0.904145
    57    0.902595
    79    0.885875
    2     0.872149
y   68    0.897646
    21    0.896949
    72    0.884951
    70    0.859120
    82    0.833257
Name: v1, dtype: float64
```

Combine 2 DataFrames with `pd.merge`

`pandas.merge` is similar to the *SQL join* operations; it links rows of tables using one or more *keys*. The following graphic outlines the specifics of common SQL join operations.



Syntax

```
pd.merge(df1, df2, how='left', on='key',
         left_on=None, right_on=None,
         left_index=False, right_index=False,
         sort=True, copy=True,
         suffixes=('x', 'y'))
```

The syntax includes specifications of the following arguments

- **Which column to merge on;**
    - the `on='key'` if the same key is present in the two DFs,
    - or `left_on='lkey', right_on='rkey'` if the keys have different names in the DFs
    - [*Note*] To merge on multiple keys, pass a *list of column names*

- **The nature of the join;**
    - the `how=` option, with `left`, `right`, `outer`
    - By default, the merge is an `inner` join

- Tuple of string values to append to **overlapping column names** to identify them in the merged dataset
    - the `suffixes=` option
    - defaults to `('_x', '_y')`

- If you wish **to merge on the index**, pass `left_index=True` or `right_index=True` or both.

- Sort the result DataFrame by the join keys in lexicographical order or not;
    - `sort=` option;
    - Defaults to `True`, setting to `False` will improve performance substantially in many cases

Example

```
# Let's define a few toy datasets to use as examples

df0 = DataFrame({'key': ['a', 'b', 'c', 'd', 'e'],
                 'data0': np.random.randint(0, 100,
5)})
df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a',
'b'],
                 'data1': np.random.randint(0, 100,
7)})
df2 = DataFrame({'key': ['a', 'b', 'd', 'f', 'g'],
                 'data2': np.random.randint(0, 100,
5)})
df3 = DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a',
'b'],
                 'data3': np.random.randint(0, 100,
7)})
df4 = DataFrame({'rkey': ['a', 'b', 'd'],
                 'data4': np.random.randint(0, 100,
3)})

print df0, '\n\n', df1, '\n\n', df2, '\n\n', df3,
'\n\n', df4
```

```
   data0 key
0      1   a
1     55   b
2      5   c
3     59   d
4     67   e

   data1 key
0     29   b
1     39   b
2     26   a
3     39   c
4     91   a
5     39   a
6     43   b
```

```
   data2 key
0     86   a
1     85   b
2     25   d
3     98   f
4      8   g

   data3 lkey
0     29    b
1     16    b
2     60    a
3     22    c
4     81    a
5     28    a
6     95    b

   data4 rkey
0     23    a
1     48    b
2     84    d
```

1. Default merge (*inner join*) when both DataFrames have a common key(s)

This leads to an *inner join* by default (output key is the **intersection** of input keys)
Merge happens on the column '*key*' which is common to both datasets.

```
# inner join
pd.merge(df0, df2)

   data0 key  data2
0      1   a     86
1     55   b     85
2     59   d     25
```

Note that we could've written

```
pd.merge(df0, df2, on='key', how='inner')
```

to achieve the same effect.

If there are no matching column names in the two DataFrames to merge on, the default merge will throw an error.

```
pd.merge(df0, df4)


---------------------------------------------------------
--------------------
MergeError: No common columns to perform merge on
```

Outer Join

This leads to a **union** of keys, missing values are imputed in the resulting dataset wherever a match isn't found in either table.

```
pd.merge(df0, df2, how='outer')
```

	data0	key	data2
0	1.0	a	86.0
1	55.0	b	85.0
2	5.0	c	NaN
3	59.0	d	25.0
4	67.0	e	NaN
5	NaN	f	98.0
6	NaN	g	8.0

Left Join

The value for '**c**' is absent in df2, so there will be a `NaN` in column data2 in the resulting dataset.

```
pd.merge(df1, df2, how='left')
```

	data1	key	data2
0	57	b	18.0
1	70	b	18.0
2	86	a	60.0
3	34	c	NaN
4	4	a	60.0
5	3	a	60.0
6	67	b	18.0

2. Merge with keys having different names in the two dataframes

```
pd.merge(df1, df4, left_on='key', right_on='rkey')

   data1 key  data4 rkey
0     57   b     41    b
1     70   b     41    b
2     67   b     41    b
3     86   a     71    a
4      4   a     71    a
5      3   a     71    a
```

3. Merge with DataFrame containing overlapping column names

By default, pandas will append suffixes (`_x and _y`) to the column names in the merged dataset to identify where the column came from.

```
# Add a column with the same name to df1 and df2
In []:
df1['newcol'] = np.random.randn(7)
df2['newcol'] = np.random.randn(5)

pd.merge(df1, df2, on='key')

Out[]:
   data1 key  newcol_x  data2  newcol_y
0     33   b  2.178623     72 -0.379224
1      3   b  0.288692     72 -0.379224
2     91   b -0.176319     72 -0.379224
3      3   a -0.765213     10 -0.915644
4      2   a -0.371988     10 -0.915644
5      2   a  0.926292     10 -0.915644
```

We could specify other suffixes explicitly with the `suffixes=` option

```
pd.merge(df1, df2, on='key', suffixes=['_df1', '_df2'])

    data1 key  newcol_df1  data2  newcol_df2
0     33   b    2.178623     72   -0.379224
1      3   b    0.288692     72   -0.379224
2     91   b   -0.176319     72   -0.379224
3      3   a   -0.765213     10   -0.915644
4      2   a   -0.371988     10   -0.915644
5      2   a    0.926292     10   -0.915644
```

4. Merge when the key is the `index` in one (or both) of the DataFrames

```
# Set lkey to be the index of df3
df3.set_index('lkey', inplace=True)


# We specify that for the left df we will use the
column called 'key' and for the right df, we will use
its index for the merge
pd.merge(df2, df3, how='left', left_on='key',
right_index=True)
```

Combine multiple DataFrames with `DataFrame.join()`

`.join()` is a convenient **DataFrame method** for combining many DataFrames objects with the same or similar indexes but (non-overlapping) columns into a single result DataFrame.

- By default, the `join` method performs a *left join* on the join keys.
- For simple **index-on-index merges** we can pass a list of DataFrames to `join.`

The `join` operation is equivalent to performing a `reduce` on a list of DataFrames that have the same index.

Syntax

```
df1.join(df2)

# which is equivalent to
pd.merge(df1, df2, how='left', right_index=True,
left_index=True)

# we can alter the nature of the join
df1.join(df2, how='outer')
```

Example

```
In []:
# Create dataFrames with partially overlapping index
objects
```

```
df = DataFrame(np.random.randint(0, 50, 32).reshape(8,
4),
               columns=list('WXYZ'),
               index=list('abcdefgh'))
df1 = df.ix[2:, ['W', 'X']]
df2 = df.ix[:5, ['Y', 'Z']]
print df, '\n\n', df1, '\n\n', df2

Out[]:
    W   X   Y   Z
a   9  11  19  33
b  12  48   6  33
c  18   7  18  39
d  11  12  15  23
e  36   5  32  38
f  35  22  40  36
g  40  29  11  10
h  37  14  32  39

    W   X
c  18   7
d  11  12
e  36   5
f  35  22
g  40  29
h  37  14

    Y   Z
a  19  33
b   6  33
c  18  39
d  15  23
e  32  38

# Perform the join
In []: df1.join(df2)
Out[]:
    W   X     Y     Z
c  18   7  18.0  39.0
d  11  12  15.0  23.0
e  36   5  32.0  38.0
f  35  22   NaN   NaN
g  40  29   NaN   NaN
h  37  14   NaN   NaN
```

The real strength of the join method is apparent when we have multiple dataFrames. Then, instead of having to write a number of `merge` statements, we can get away with writing a single `join`.

```
# Create a couple more DFs with the same index
In []:
df3 = df.ix[0:3, ['X', 'Z']]
df3.columns = ['P', 'Q']
df4 = df.ix[4:6, ['W']]
df4.columns = ['R']
print df3, "\n\n", df4

Out[]:
    P   Q
a  11  33
b  48  33
c   7  39

    R
e  36
f  35

# Merging multiple DFs with the same index by passing a
list of names to .join
In []: df1.join([df2, df3, df4])

Out[]:
    W   X     Y     Z    P     Q     R
c  18   7  18.0  39.0  7.0  39.0   NaN
d  11  12  15.0  23.0  NaN   NaN   NaN
e  36   5  32.0  38.0  NaN   NaN  36.0
f  35  22   NaN   NaN  NaN   NaN  35.0
g  40  29   NaN   NaN  NaN   NaN   NaN
h  37  14   NaN   NaN  NaN   NaN   NaN
```

Since the default action is a `left join` we observe that all values of the index in `df1` are retained in the output, and missing values are imputed whereever no matching values are found. This action can be altered by passing the `how=` parameter.

Combine multiple DataFrames/Series with `pd.concat()`

The concat() function in pandas is used to concatenate pandas objects along a particular axis with optional set logic along the other axes. This operation is synonymous with `binding, stacking, union all` functions in other languages.

Concatentation can happen along either axis, the action being governed by the `axis=` parameter.

- With `axis=0` the objects will be *appended vertically*, i.e. the resulting object will have more **rows**. This is similar to the `rbind` operation in R and the `UNION` operation in SQL.
- With `axis=1` the objects will be *concatenated horizontally*, leading to an object with more **columns**. This is analogous to an `outer join`

Depending on whether the objects have overlapping index (or column) labels, the concat will also include merging of the data where an overlap is found.

Syntax

```
pd.concat([list-of-series/dataframe-objects],
axis=0_or_1)
```

Examples

```
# Create toy Series with non-overlapping indices
In []:
s1 = Series(np.random.randn(3), index=list('abc'))
s2 = Series(np.random.randn(4), index=list('defg'))
s3 = Series(np.random.randn(2), index=list('hi'))
print '\n\n', s1, '\n\n', s2, '\n\n', s3

Out[]:
a    0.766440
b   -1.022994
c    1.801808
dtype: float64

d   -0.410916
e    2.007578
f   -1.303415
g   -0.194217
dtype: float64

h   -1.401666
i    0.156245
dtype: float64
```

1. For `Series` objects with **no index overlap**

```
# with axis=0 (default) will append the Series (~rbind)
In []:
pd.concat([s1, s2, s3])

Out []:
a    0.766440
b   -1.022994
c    1.801808
d   -0.410916
e    2.007578
f   -1.303415
g   -0.194217
h   -1.401666
i    0.156245
dtype: float64

# with axis=1 will *merge* the Series to produce a DF
(~full outer join)
In []:
pd.concat([s1, s2, s3], axis=1)

Out[]:
          0         1         2
a  0.766440       NaN       NaN
b -1.022994       NaN       NaN
c  1.801808       NaN       NaN
d       NaN -0.410916       NaN
e       NaN  2.007578       NaN
f       NaN -1.303415       NaN
g       NaN -0.194217       NaN
h       NaN       NaN -1.401666
i       NaN       NaN  0.156245
```

2. For `Series` objects **with an overlap on indexes**, we can specify the `join=` parameter to intersect the data. Note that the `join=` option takes only `'inner'` and `'outer'` as valid arguments.

```
# Create a new Series
In []:
s4 = Series(np.random.randn(5), index=list('abcde'))
s4

Out[]:
a   -0.758740
b   -1.557941
c    0.673917
d   -1.605039
e    0.382931
dtype: float64

# concat with overlapping index (default join type is
outer)
In []:
pd.concat([s1, s4], axis=1)

Out[]:
          0         1
a  0.766440 -0.758740
b -1.022994 -1.557941
c  1.801808  0.673917
d       NaN -1.605039
e       NaN  0.382931

# if we specify a join type, this will be equivalent to
a merge
In []:
pd.concat([s1, s4], axis=1, join='inner')

Out[]:
          0         1
a  0.766440 -0.758740
b -1.022994 -1.557941
c  1.801808  0.673917
```

3. For `DataFrame` objects with ***no overlapping index,***
    - `axis = 0` will produce a concatenation
    - `axis = 1` will produce as merge, imputing NaN values where necessary.

```
# Create toy dataframes with non-overlapping indexes
In []:
df1 = DataFrame(np.random.randn(9).reshape(3, 3),
                index=list('abc'), columns=list('XYZ'))
df2 = DataFrame(np.random.randn(4).reshape(2, 2),
                index=list('pq'), columns=list('XZ'))
print df1, '\n\n', df2

Out[]:
          X         Y         Z
a -0.35526  0.629197  0.605691
b -0.44103 -0.416271  0.212713
c -0.37739 -2.500837  0.497077

          X         Z
p  1.146606  0.010390
q  1.738752 -0.727999

# concat with axis=0
In []: pd.concat([df1, df2], axis=0)

Out[]:
          X         Y         Z
a -0.355260  0.629197  0.605691
b -0.441030 -0.416271  0.212713
c -0.377390 -2.500837  0.497077
p  1.146606       NaN  0.010390
q  1.738752       NaN -0.727999

# concat with axis=1
In []: pd.concat([df1, df2], axis=1)

Out[]:
          X         Y         Z         X         Z
a -0.35526  0.629197  0.605691       NaN       NaN
b -0.44103 -0.416271  0.212713       NaN       NaN
c -0.37739 -2.500837  0.497077       NaN       NaN
p       NaN       NaN       NaN  1.146606  0.010390
q       NaN       NaN       NaN  1.738752 -0.727999
```

4. For `DataFrame` objects with an ***overlapping index***,

```
# Create toy dataframes with overlapping indexes
In []:
df1 = DataFrame(np.random.randn(9).reshape(3, 3),
                index=list('abc'), columns=list('XYZ'))
df2 = DataFrame(np.random.randn(4).reshape(2, 2),
                index=list('ac'), columns=list('XZ'))
print df1, '\n\n', df2

Out[]:
          X         Y         Z
a  1.490298 -1.424274  0.498258
b -0.375134 -0.175445  0.533636
c  2.198040 -0.375883 -0.312773

          X         Z
a  0.240080  1.137680
c  0.121164 -0.076435

# with axis=0 (default)
In []: pd.concat([df1, df2])

Out[]:
          X         Y         Z
a  1.490298 -1.424274  0.498258
b -0.375134 -0.175445  0.533636
c  2.198040 -0.375883 -0.312773
a  0.240080       NaN  1.137680
c  0.121164       NaN -0.076435

# with axis=1
In []: pd.concat([df1, df2], axis=1)
Out[]:
          X         Y         Z         X         Z
a  1.490298 -1.424274  0.498258  0.240080  1.137680
b -0.375134 -0.175445  0.533636       NaN       NaN
c  2.198040 -0.375883 -0.312773  0.121164 -0.076435
```