Handout Week 3: Pandas for Unstructured Data

Dr Ian (Yinglong) He

Click to download the Jupyter Notebook files (.ipynb)

based on the notes by Jake VanderPlas

Contents

1 Introduction

As we saw, NumPy's `ndarray` data structure provides essential features for the type of **clean, well-organized data** typically seen in numerical computing tasks. While it serves this purpose very well, its limitations become clear when we need more flexibility (e.g., attaching labels to data, working with missing data, etc.) and when attempting operations that do not map well to element-wise broadcasting (e.g., groupings, pivots, etc.), each of which is an important piece of analyzing the less structured data available in many forms in the world around us.

Pandas, and in particular its `Series` and `DataFrame` objects, builds on the NumPy array structure. If a `Series` is an analog of a one-dimensional array with flexible indices, a `DataFrame` is an analog of a two-dimensional array with both flexible row indices and flexible column names. `DataFrame`s are essentially multidimensional arrays with attached row and column labels, and often with **heterogeneous types and/or missing data**. As well as offering a convenient storage interface for labeled data, Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

In this handout, we will focus on the mechanics of using `DataFrame`, and related structures effectively. We will use examples drawn from real datasets where appropriate, but these examples are not necessarily the focus. Details on this installation can be found in the Pandas documentation. If you used the Anaconda stack, you already have Pandas installed.

Once Pandas is installed, you can import it and check the version:

[1]:
```python
import pandas
pandas.__version__
```

[1]:
```
'1.5.1'
```

Just as we generally import NumPy under the alias `np`, we will import Pandas under the alias `pd`:

[2]:
```python
import pandas as pd
```

This import convention will be used throughout the remainder of this book.

For convenience, we'll use the same `display` magic function that we've seen in previous sections:

[3]:
```python
import numpy as np
import pandas as pd

class display(object):
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

2 Constructing DataFrame objects

A Pandas DataFrame can be constructed in a variety of ways. Here we'll give several examples.

2.1 From a two-dimensional NumPy array

Given a two-dimensional array of data, we can create a DataFrame with any specified column and index names. If omitted, an integer index will be used for each:

```
[4]: pd.DataFrame(np.random.rand(3, 2),
                  columns=['foo', 'bar'],
                  index=['a', 'b', 'c'])
```

```
[4]:         foo       bar
     a  0.245258  0.496904
     b  0.316460  0.871386
     c  0.918690  0.193112
```

2.2 From a dictionary

Any dictionary can be made into a DataFrame:

```
[5]: data_dict = {
         'state': ['California', 'Texas', 'New York', 'Florida', 'Illinois'],
         'area': [423967, 695662, 141297, 170312, 149995],
         'population': [38332521, 26448193, 19651127, 19552860, 12882135]}

     df = pd.DataFrame(data_dict)
     df
```

```
[5]:         state    area  population
     0  California  423967    38332521
     1       Texas  695662    26448193
     2    New York  141297    19651127
     3     Florida  170312    19552860
     4    Illinois  149995    12882135
```

We can set the index to become one of the columns:

```
[6]: df = df.set_index('state')
     df
```

```
[6]:               area  population
     state
     California  423967    38332521
     Texas       695662    26448193
     New York    141297    19651127
     Florida     170312    19552860
     Illinois    149995    12882135
```

The DataFrame have an index attribute that gives access to the index labels:

```
[7]: df.index
```

```
[7]: Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'],
     dtype='object', name='state')
```

Additionally, the DataFrame and Series have a columns attribute, which is an Index object holding the column labels:

```
[8]: df.columns
```

```
[8]: Index(['area', 'population'], dtype='object')
```

3 Writing DataFrame to a CSV File

We can store a DataFrame object into a csv file using the to_csv method:

```
[9]: df.to_csv('my_dataframe.csv')
```

4 Data Selection in DataFrame

In the previous handout, we looked in detail at methods and tools to access, set, and modify values in NumPy arrays. These included indexing (e.g., arr[2, 1]), slicing (e.g., arr[:, 1:5]), masking (e.g., arr[arr > 0]), fancy indexing (e.g., arr[0, [1, 5]]), and combinations thereof (e.g., arr[:, [1, 5]]). Here we'll look at similar means of accessing and modifying values in Pandas DataFrame objects. If you have used the NumPy patterns, the corresponding patterns in Pandas will feel very familiar, though there are a few quirks to be aware of.

4.1 DataFrame as a dictionary

The first analogy we will consider is the DataFrame as a dictionary of related Series objects. Let's return to our example of areas and populations of states:

```
[10]: df['area']
```

```
[10]: state
      California    423967
      Texas         695662
      New York      141297
      Florida       170312
      Illinois      149995
      Name: area, dtype: int64
```

Equivalently, we can use attribute-style access with column names that are strings:

```
[11]: df.area
```

```
[11]: state
      California    423967
      Texas         695662
      New York      141297
      Florida       170312
```

```
Illinois      149995
Name: area, dtype: int64
```

This dictionary-style syntax can also be used to modify the object, in this case adding a new column:

[12]:
```
df['density'] = df['population'] / df['area']
df
```

[12]:
```
              area  population     density
state
California  423967    38332521   90.413926
Texas       695662    26448193   38.018740
New York    141297    19651127  139.076746
Florida     170312    19552860  114.806121
Illinois    149995    12882135   85.883763
```

This shows a preview of the straightforward syntax of element-by-element arithmetic between `Series` objects.

4.2 DataFrame as two-dimensional array

As mentioned previously, we can also view the `DataFrame` as an enhanced two-dimensional array. We can examine the raw underlying data array using the `values` attribute:

[13]:
```
df.values
```

[13]:
```
array([[4.23967000e+05, 3.83325210e+07, 9.04139261e+01],
       [6.95662000e+05, 2.64481930e+07, 3.80187404e+01],
       [1.41297000e+05, 1.96511270e+07, 1.39076746e+02],
       [1.70312000e+05, 1.95528600e+07, 1.14806121e+02],
       [1.49995000e+05, 1.28821350e+07, 8.58837628e+01]])
```

With this picture in mind, many familiar array-like observations can be done on the `DataFrame` itself. For example, we can transpose the full `DataFrame` to swap rows and columns:

[14]:
```
df.T
```

[14]:
```
state         California         Texas      New York       Florida  \
area        4.239670e+05  6.956620e+05  1.412970e+05  1.703120e+05
population  3.833252e+07  2.644819e+07  1.965113e+07  1.955286e+07
density     9.041393e+01  3.801874e+01  1.390767e+02  1.148061e+02

state           Illinois
area        1.499950e+05
population  1.288214e+07
density     8.588376e+01
```

4.2.1 Indexers: loc and iloc

For array-style indexing, Pandas uses the `loc` and `iloc` indexers:

- `loc` attribute allows indexing and slicing that always references the explicit (label based) index.

- `iloc` attribute allows indexing and slicing that always references the implicit (integer-location based) index.

Using the `iloc` indexer, we can index the underlying array as if it is a simple NumPy array, but the `DataFrame` index and column labels are maintained in the result:

[15]:
```
df.iloc[:3, :2]
```

[15]:
```
              area  population
state
California  423967    38332521
Texas       695662    26448193
New York    141297    19651127
```

Similarly, using the `loc` indexer we can index the underlying data using the explicit index and column names:

[16]:
```
df.loc[:'Illinois', :'population']
```

[16]:
```
              area  population
state
California  423967    38332521
Texas       695662    26448193
New York    141297    19651127
Florida     170312    19552860
Illinois    149995    12882135
```

Any of the familiar NumPy-style data access patterns can be used within these indexers. For example, in the `loc` indexer we can combine masking and fancy indexing as in the following:

[17]:
```
df.loc[df['density'] > 100, ['population', 'density']]
```

[17]:
```
          population     density
state
New York    19651127  139.076746
Florida     19552860  114.806121
```

Any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be accustomed to from working with NumPy:

[18]:
```
df.iloc[0, 2] = 90
df
```

[18]:
```
              area  population     density
state
California  423967    38332521   90.000000
Texas       695662    26448193   38.018740
New York    141297    19651127  139.076746
Florida     170312    19552860  114.806121
Illinois    149995    12882135   85.883763
```

To build up your fluency in Pandas data manipulation, I suggest spending some time with a simple `DataFrame` and exploring the types of indexing, slicing, masking, and fancy indexing that are allowed by these various indexing approaches.

4.3 Additional indexing conventions

There are a couple extra indexing conventions that might seem at odds with the preceding discussion, but nevertheless can be very useful in practice. First, while *indexing* refers to columns, *slicing* refers to rows:

```
[19]: df['Florida':'Illinois']
```

```
[19]:           area  population     density
      state
      Florida   170312    19552860  114.806121
      Illinois  149995    12882135   85.883763
```

Such slices can also refer to rows by number rather than by index:

```
[20]: df[1:3]
```

```
[20]:           area  population     density
      state
      Texas     695662    26448193   38.018740
      New York  141297    19651127  139.076746
```

Similarly, direct masking operations are also interpreted row-wise rather than column-wise:

```
[21]: df[df['density'] > 100]
```

```
[21]:           area  population     density
      state
      New York  141297    19651127  139.076746
      Florida   170312    19552860  114.806121
```

These two conventions are syntactically similar to those on a NumPy array, and while these may not precisely fit the mold of the Pandas conventions, they are nevertheless quite useful in practice.

5 Operating on Null Values

The difference between data found in many tutorials and data in the real world is that real-world data is rarely clean and homogeneous. In particular, many interesting datasets will have some amount of data missing. To make matters even more complicated, different data sources may indicate missing data in different ways.

Pandas treats `None` and `NaN` as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:

- `isnull()`: Generate a boolean mask indicating missing values
- `notnull()`: Opposite of `isnull()`
- `dropna()`: Return a filtered version of the data
- `fillna()`: Return a copy of the data with missing values filled or imputed

We will conclude this section with a brief exploration and demonstration of these routines.

5.1 Detecting null values

Pandas data structures have two useful methods for detecting null data: `isnull()` and `notnull()`. Either one will return a Boolean mask over the data. For example:

```
[22]: df = pd.DataFrame(np.array([[1.5, np.nan, 2.2],
                                  [2.0, 3.5, 5.3],
                                  [np.nan, 4.1, 6.2]]),
                        columns=['col1', 'col2', 'col3'],
                        index=['a', 'b', 'c'])
      df
```

```
[22]:    col1  col2  col3
      a   1.5   NaN   2.2
      b   2.0   3.5   5.3
      c   NaN   4.1   6.2
```

```
[23]: df.isnull()
```

```
[23]:     col1   col2   col3
      a  False   True  False
      b  False  False  False
      c   True  False  False
```

The `isnull()` and `notnull()` methods produce similar Boolean results for `DataFrame`s.

5.2 Dropping null values

In addition to the masking used before, there are the convenience methods, `dropna()` (which removes NA values) and `fillna()` (which fills in NA values). We cannot drop single values from a `DataFrame`; we can only drop full rows or full columns. Depending on the application, you might want one or the other, so `dropna()` gives a number of options for a `DataFrame`.

By default, `dropna()` will drop all rows in which *any* null value is present:

```
[24]: df.dropna()
```

```
[24]:    col1  col2  col3
      b   2.0   3.5   5.3
```

Alternatively, you can drop NA values along a different axis; `axis=1` drops all columns containing a null value:

```
[25]: df.dropna(axis='columns')
```

```
[25]:    col3
      a   2.2
      b   5.3
      c   6.2
```

5.3 Filling null values

Sometimes rather than dropping NA values, you'd rather replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. You could do this in-place using the `isnull()` method as a mask, but because it is such a common operation Pandas provides the `fillna()` method, which returns a copy of the array with the null values replaced.

Consider the following `DataFrame`:

```
[26]: df
```

```
[26]:    col1  col2  col3
      a   1.5   NaN   2.2
      b   2.0   3.5   5.3
      c   NaN   4.1   6.2
```

For `DataFrame`s, we can specify an `axis` along which the fills take place:

```
[27]: df.fillna(method='ffill', axis=1)
```

```
[27]:    col1  col2  col3
      a   1.5   1.5   2.2
      b   2.0   3.5   5.3
      c   NaN   4.1   6.2
```

6 Combining Datasets: Concat

Some of the most interesting studies of data come from combining different data sources. These operations can involve anything from very straightforward concatenation of two different datasets, to more complicated database-style joins and merges that correctly handle any overlaps between the datasets. `Series` and `DataFrame`s are built with this type of operation in mind, and Pandas includes functions and methods that make this sort of data wrangling fast and straightforward.

Pandas has a function, `pd.concat()`, which has a similar syntax to `np.concatenate` but contains a number of options that we'll discuss momentarily:

```
# Signature in Pandas v0.18
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False,
          copy=True)
```

`pd.concat()` can be used for a simple concatenation of `Series` or `DataFrame` objects, just as `np.concatenate()` can be used for simple concatenations of arrays:

```
[28]: df1 = pd.DataFrame([['a', 1], ['b', 2]],
                         columns=['col1', 'col2'])
      df2 = pd.DataFrame([['c', 3], ['d', 4]],
                         columns=['col1', 'col2'])
      display('df1', 'df2', 'pd.concat([df1, df2])')
```

```
[28]: df1
        col1  col2
      0    a     1
      1    b     2

      df2
        col1  col2
```

```
0    c     3
1    d     4

pd.concat([df1, df2])
  col1  col2
0    a     1
1    b     2
0    c     3
1    d     4
```

By default, the concatenation takes place row-wise within the `DataFrame` (i.e., `axis=0`). Like `np.concatenate`, `pd.concat` allows specification of an axis along which concatenation will take place. Consider the following example:

[29]:
```
df3 = pd.DataFrame([['a', 1], ['b', 2]],
                   columns=['col1', 'col2'])
df4 = pd.DataFrame([['c', 3], ['d', 4]],
                   columns=['col3', 'col4'])
display('df3', 'df4', "pd.concat([df3, df4], axis=1)")
```

[29]:
```
df3
  col1  col2
0    a     1
1    b     2

df4
  col3  col4
0    c     3
1    d     4

pd.concat([df3, df4], axis=1)
  col1  col2 col3  col4
0    a     1    c     3
1    b     2    d     4
```

We combine DataFrame objects horizontally (or vetically) along the x axis by passing in `axis=1` (or `axis=0`)

6.1 Ignoring the index

Sometimes the index itself does not matter, and you would prefer it to simply be ignored. This option can be specified using the `ignore_index` flag. With this set to true, the concatenation will create a new integer index for the resulting `DataFrame`:

[30]:
```
display('df1', 'df2', 'pd.concat([df1, df2])', 'pd.concat([df1, df2],␣
 ↪ignore_index=True)')
```

[30]:
```
df1
  col1  col2
0    a     1
1    b     2
```

```
df2
  col1  col2
0    c     3
1    d     4

pd.concat([df1, df2])
  col1  col2
0    a     1
1    b     2
0    c     3
1    d     4

pd.concat([df1, df2], ignore_index=True)
  col1  col2
0    a     1
1    b     2
2    c     3
3    d     4
```

6.2 Concatenation with joins

In the simple examples we just looked at, we were mainly concatenating DataFrames with shared column names. In practice, data from different sources might have different sets of column names, and pd.concat offers several options in this case. Consider the concatenation of the following two DataFrames, which have some (but not all!) columns in common:

[31]:

```
df5 = pd.DataFrame([['a', 1], ['b', 2]],
                   columns=['col1', 'col2'])
df6 = pd.DataFrame([['c', 3], ['d', 4]],
                   columns=['col2', 'col3'])
display('df5', 'df6', 'pd.concat([df5, df6])')
```

[31]:

```
df5
  col1  col2
0    a     1
1    b     2

df6
  col2  col3
0    c     3
1    d     4

pd.concat([df5, df6])
  col1 col2  col3
0    a    1   NaN
1    b    2   NaN
0  NaN    c   3.0
1  NaN    d   4.0
```

By default, the entries for which no data is available are filled with NA values. To change this, we can specify one of several options for the join and join_axes parameters of the concatenate function. By default, the join is a union of the input columns (join='outer'), but we can change this to an intersection of the columns using `join='inner'`:

```
[32]: display('df5', 'df6',
              "pd.concat([df5, df6], join='inner')")
```

```
[32]: df5
        col1  col2
      0    a     1
      1    b     2

      df6
        col2  col3
      0    c     3
      1    d     4

      pd.concat([df5, df6], join='inner')
        col2
      0    1
      1    2
      0    c
      1    d
```

The combination of options of the `pd.concat` function allows a wide range of possible behaviors when joining two datasets; keep these in mind as you use these tools for your own data.

7 Combining Datasets: Merge

One essential feature offered by Pandas is its high-performance, in-memory join and merge operations. If you have ever worked with databases, you should be familiar with this type of data interaction. The main interface for this is the `pd.merge` function, and we'll see few examples of how this can work in practice. The main difference between `pd.merge()` and `pd.concat()` is:

- `pd.merge()` is used to combine two (or more) dataframes on the basis of values of common columns.
- `pd.concat()` is used to append one (or more) dataframes one below the other (or sideways)

7.1 Categories of Joins

The `pd.merge()` function implements a number of types of joins: the *one-to-one*, *many-to-one*, and *many-to-many* joins. All three types of joins are accessed via an identical call to the `pd.merge()` interface; the type of join performed depends on the form of the input data. Here we will show simple examples of the three types of merges, and discuss detailed options further below.

7.1.1 One-to-one joins

Perhaps the simplest type of merge expresion is the one-to-one join, which is in many ways very similar to the column-wise concatenation. As a concrete example, consider the following two `DataFrames` which contain information on several employees in a company:

```
[33]: df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                          'group': ['Accounting', 'Engineering', 'Engineering',␣
        ↪'HR']})
```

```
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
display('df1', 'df2')
```

[33]:

```
df1
  employee        group
0      Bob   Accounting
1     Jake  Engineering
2     Lisa  Engineering
3      Sue           HR

df2
  employee  hire_date
0     Lisa       2004
1      Bob       2008
2     Jake       2012
3      Sue       2014
```

To combine this information into a single `DataFrame`, we can use the `pd.merge()` function:

[34]:

```
df3 = pd.merge(df1, df2)
df3
```

[34]:

```
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
3      Sue           HR       2014
```

The `pd.merge()` function recognizes that each `DataFrame` has an “employee” column, and automatically joins using this column as a key. The result of the merge is a new `DataFrame` that combines the information from the two inputs. Notice that the order of entries in each column is not necessarily maintained: in this case, the order of the “employee” column differs between `df1` and `df2`, and the `pd.merge()` function correctly accounts for this. Additionally, keep in mind that the merge in general discards the index, except in the special case of merges by index (see the `left_index` and `right_index` keywords, discussed momentarily).

7.1.2 Many-to-one joins

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting `DataFrame` will preserve those duplicate entries as appropriate. Consider the following example of a many-to-one join:

[35]:

```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
display('df3', 'df4', 'pd.merge(df3, df4)')
```

[35]:

```
df3
  employee        group  hire_date
0      Bob   Accounting       2008
1     Jake  Engineering       2012
2     Lisa  Engineering       2004
```

```
3      Sue           HR       2014

df4
         group supervisor
0   Accounting      Carly
1  Engineering      Guido
2           HR      Steve

pd.merge(df3, df4)
  employee        group  hire_date supervisor
0      Bob   Accounting       2008      Carly
1     Jake  Engineering       2012      Guido
2     Lisa  Engineering       2004      Guido
3      Sue           HR       2014      Steve
```

The resulting DataFrame has an aditional column with the “supervisor” information, where the information is repeated in one or more locations as required by the inputs.

7.1.3 Many-to-many joins

Many-to-many joins are a bit confusing conceptually, but are nevertheless well defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete example. Consider the following, where we have a DataFrame showing one or more skills associated with a particular group. By performing a many-to-many join, we can recover the skills associated with any individual person:

```
[36]: df5 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                                    'Engineering', 'Engineering', 'HR', 'HR'],
                          'skills': ['math', 'spreadsheets', 'coding', 'linux',
                                     'spreadsheets', 'organization']})
      display('df1', 'df5', "pd.merge(df1, df5)")
```

```
[36]: df1
        employee        group
      0      Bob   Accounting
      1     Jake  Engineering
      2     Lisa  Engineering
      3      Sue           HR

      df5
               group        skills
      0   Accounting          math
      1   Accounting  spreadsheets
      2  Engineering        coding
      3  Engineering         linux
      4           HR  spreadsheets
      5           HR  organization

      pd.merge(df1, df5)
        employee        group        skills
      0      Bob   Accounting          math
      1      Bob   Accounting  spreadsheets
```

```
2      Jake  Engineering        coding
3      Jake  Engineering         linux
4      Lisa  Engineering        coding
5      Lisa  Engineering         linux
6       Sue           HR  spreadsheets
7       Sue           HR  organization
```

These three types of joins can be used with other Pandas tools to implement a wide array of functionality. But in practice, datasets are rarely as clean as the one we're working with here. In the following section we'll consider some of the options provided by `pd.merge()` that enable you to tune how the join operations work.

7.2 Specification of the Merge Key

We've already seen the default behavior of `pd.merge()`: it looks for one or more matching column names between the two inputs, and uses this as the key. However, often the column names will not match so nicely, and `pd.merge()` provides a variety of options for handling this.

7.2.1 The `on` keyword

Most simply, you can explicitly specify the name of the key column using the `on` keyword, which takes a column name or a list of column names:

```
[37]: display('df1', 'df2', "pd.merge(df1, df2, on='employee')")
```

```
[37]: df1
        employee        group
      0      Bob   Accounting
      1     Jake  Engineering
      2     Lisa  Engineering
      3      Sue           HR

      df2
        employee  hire_date
      0     Lisa       2004
      1      Bob       2008
      2     Jake       2012
      3      Sue       2014

      pd.merge(df1, df2, on='employee')
        employee        group  hire_date
      0      Bob   Accounting       2008
      1     Jake  Engineering       2012
      2     Lisa  Engineering       2004
      3      Sue           HR       2014
```

This option works only if both the left and right `DataFrames` have the specified column name.

7.2.2 The `left_on` and `right_on` keywords

At times you may wish to merge two datasets with different column names; for example, we may have a dataset in which the employee name is labeled as "name" rather than "employee". In this case, we can use the `left_on` and `right_on` keywords to specify the two column names:

```
[38]: df3 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                          'salary': [70000, 80000, 120000, 90000]})
      display('df1', 'df3', 'pd.merge(df1, df3, left_on="employee",␣
       ↪right_on="name")')
```

```
[38]: df1
        employee        group
      0      Bob   Accounting
      1     Jake  Engineering
      2     Lisa  Engineering
      3      Sue           HR

      df3
         name  salary
      0   Bob   70000
      1  Jake   80000
      2  Lisa  120000
      3   Sue   90000

      pd.merge(df1, df3, left_on="employee", right_on="name")
        employee        group  name  salary
      0      Bob   Accounting   Bob   70000
      1     Jake  Engineering  Jake   80000
      2     Lisa  Engineering  Lisa  120000
      3      Sue           HR   Sue   90000
```

The result has a redundant column that we can drop if desired–for example, by using the `drop()` method of `DataFrames`:

```
[39]: pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis=1)
```

```
[39]:   employee        group  salary
      0      Bob   Accounting   70000
      1     Jake  Engineering   80000
      2     Lisa  Engineering  120000
      3      Sue           HR   90000
```

All of these options also work with multiple indices and/or multiple columns; the interface for this behavior is very intuitive. For more information on this, see the "Merge, Join, and Concatenate" section of the Pandas documentation.

8 Aggregation and Grouping

An essential piece of analysis of large data is efficient summarization: computing aggregations like `sum()`, `mean()`, `median()`, `min()`, and `max()`, in which a single number gives insight into the nature of a potentially large dataset. In this section, we'll explore aggregations in Pandas, from simple operations akin to what we've seen on NumPy arrays, to more sophisticated operations based on the concept of a `groupby`.

8.1 Simple Aggregation in Pandas

Here we will use the Planets dataset, available via the Seaborn package. It gives information on planets that astronomers have discovered around other stars (known as *extrasolar planets* or *exoplanets* for short). It can be downloaded with a simple Seaborn command:

[40]:
```
import seaborn as sns
planets = sns.load_dataset('planets')
planets.shape
```

[40]:
```
(1035, 6)
```

[41]:
```
planets.head()
```

[41]:
```
            method  number  orbital_period   mass  distance  year
0  Radial Velocity       1         269.300   7.10     77.40  2006
1  Radial Velocity       1         874.774   2.21     56.95  2008
2  Radial Velocity       1         763.000   2.60     19.84  2011
3  Radial Velocity       1         326.030  19.40    110.62  2007
4  Radial Velocity       1         516.220  10.50    119.47  2009
```

This has some details on the 1,000+ extrasolar planets discovered up to 2014.

Pandas `DataFrame` includes all of the common aggregates; in addition, there is a convenience method `describe()` that computes several common aggregates for each column and returns the result. Let's use this on the Planets data, for now dropping rows with missing values:

[42]:
```
planets.dropna().describe()
```

[42]:
```
          number  orbital_period        mass    distance         year
count  498.00000      498.000000  498.000000  498.000000   498.000000
mean     1.73494      835.778671    2.509320   52.068213  2007.377510
std      1.17572     1469.128259    3.636274   46.596041     4.167284
min      1.00000        1.328300    0.003600    1.350000  1989.000000
25%      1.00000       38.272250    0.212500   24.497500  2005.000000
50%      1.00000      357.000000    1.245000   39.940000  2009.000000
75%      2.00000      999.600000    2.867500   59.332500  2011.000000
max      6.00000    17337.500000   25.000000  354.000000  2014.000000
```

This can be a useful way to begin understanding the overall properties of a dataset. For example, we see in the `year` column that although exoplanets were discovered as far back as 1989, half of all known expolanets were not discovered until 2010 or after. This is largely thanks to the *Kepler* mission, which is a space-based telescope specifically designed for finding eclipsing planets around other stars.

The following table summarizes some other built-in Pandas aggregations:

Aggregation	Description
`count()`	Total number of items
`first()`, `last()`	First and last item
`mean()`, `median()`	Mean and median
`min()`, `max()`	Minimum and maximum
`std()`, `var()`	Standard deviation and variance
`mad()`	Mean absolute deviation

Aggregation	Description
`prod()`	Product of all items
`sum()`	Sum of all items

These are all methods of `DataFrame` and `Series` objects.

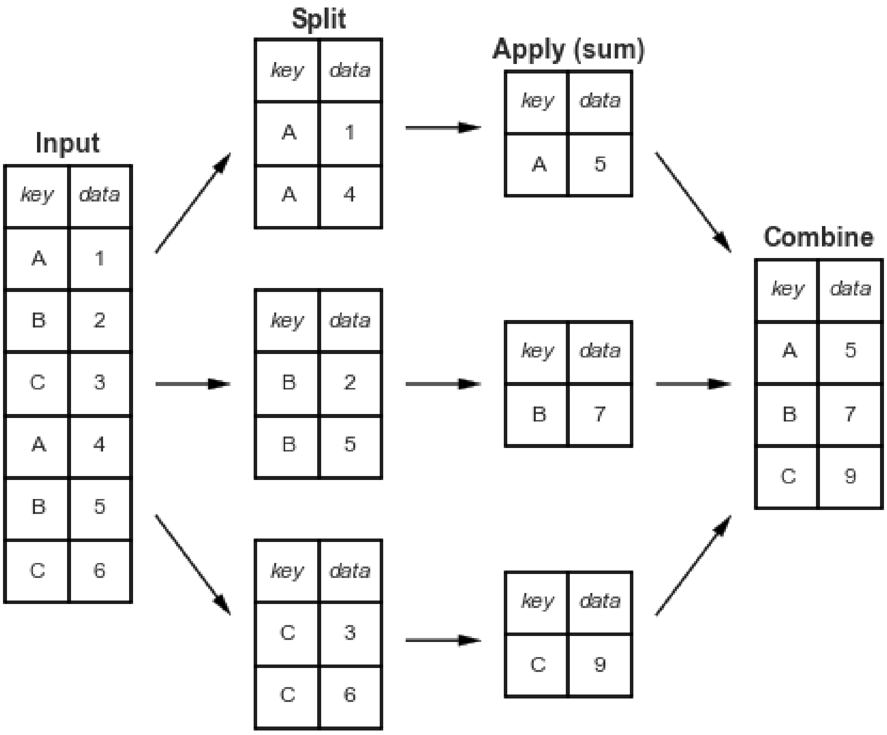
To go deeper into the data, however, simple aggregates are often not enough. The next level of data summarization is the `groupby` operation, which allows you to quickly and efficiently compute aggregates on subsets of data.

8.2 GroupBy: Split, Apply, Combine

Simple aggregations can give you a flavor of your dataset, but often we would prefer to aggregate conditionally on some label or index: this is implemented in the so-called `groupby` operation. The name "group by" comes from a command in the SQL database language, but it is perhaps more illuminative to think of it in the terms first coined by Hadley Wickham of Rstats fame: *split, apply, combine.*

8.2.1 Split, apply, combine

A canonical example of this split-apply-combine operation, where the "apply" is a summation aggregation, is illustrated in this figure:



This makes clear what the `groupby` accomplishes:

- The *split* step involves breaking up and grouping a `DataFrame` depending on the value of the specified key.
- The *apply* step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.

- The *combine* step merges the results of these operations into an output array.

While this could certainly be done manually using some combination of the masking, aggregation, and merging commands covered earlier, an important realization is that *the intermediate splits do not need to be explicitly instantiated.* Rather, the GroupBy can (often) do this in a single pass over the data, updating the sum, mean, count, min, or other aggregate for each group along the way. The power of the GroupBy is that it abstracts away these steps: the user need not think about *how* the computation is done under the hood, but rather thinks about the *operation as a whole.*

As a concrete example, let's take a look at using Pandas for the computation shown in this diagram. We'll start by creating the input DataFrame:

```
[43]: df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                         'data1': range(6),
                         'data2': range(5,11)}, columns=['key', 'data1', 'data2'])
      df
```

```
[43]:   key  data1  data2
      0   A      0      5
      1   B      1      6
      2   C      2      7
      3   A      3      8
      4   B      4      9
      5   C      5     10
```

The most basic split-apply-combine operation can be computed with the groupby() method of DataFrames, passing the name of the desired key column:

```
[44]: df.groupby('key')
```

```
[44]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x00000175A9DCB730>
```

Notice that what is returned is not a set of DataFrames, but a DataFrameGroupBy object. This object is where the magic is: you can think of it as a special view of the DataFrame, which is poised to dig into the groups but does no actual computation until the aggregation is applied. This "lazy evaluation" approach means that common aggregates can be implemented very efficiently in a way that is almost transparent to the user.

To produce a result, we can apply an aggregate to this DataFrameGroupBy object, which will perform the appropriate apply/combine steps to produce the desired result:

```
[45]: df.groupby('key').sum()
```

```
[45]:      data1  data2
      key
      A        3     13
      B        5     15
      C        7     17
```

```
[46]: df.groupby('key')['data1'].sum()
```

```
[46]: key
      A    3
```

```
B    5
C    7
Name: data1, dtype: int64
```

The `sum()` method is just one possibility here; you can apply virtually any common Pandas or NumPy aggregation function, as well as virtually any valid `DataFrame` operation, as we will see in the following discussion.

9 Reading and Writing Files

We can load a csv file into a DataFrame object (using `read_csv`). Here we will consider an example of some data about US states and their populations. The data are available in the files state_population.csv and state_abbrevs.csv.

[47]:
```
pop = pd.read_csv('data/state_population.csv')
abbrevs = pd.read_csv('data/state_abbrevs.csv')
display('pop.head()', 'abbrevs.head()')
```

[47]:
```
pop.head()
  state/region     ages  year  population
0           AL  under18  2012   1117489.0
1           AL    total  2012   4817528.0
2           AL  under18  2010   1130966.0
3           AL    total  2010   4785570.0
4           AL  under18  2011   1125763.0

abbrevs.head()
        state abbreviation
0     Alabama           AL
1      Alaska           AK
2     Arizona           AZ
3    Arkansas           AR
4  California           CA
```

[48]:
```
merged = pd.merge(pop, abbrevs, left_on='state/region',␣
 ↪right_on='abbreviation')
merged = merged.drop(columns=['abbreviation']) # drop duplicate info
merged.head()
```

[48]:
```
  state/region     ages  year  population    state
0           AL  under18  2012   1117489.0  Alabama
1           AL    total  2012   4817528.0  Alabama
2           AL  under18  2010   1130966.0  Alabama
3           AL    total  2010   4785570.0  Alabama
4           AL  under18  2011   1125763.0  Alabama
```

We can store a DataFrame object into a csv file (using `to_csv`)

[49]:
```
merged.to_csv('my_dataframe.csv')
```