ATME College of Engineering

Exploratory Data Analysis- BDS613B

Prepared By,
Dr. Anitha DB
Associate Professor & Head
Department of CSE-Data Science
ATME College of Engineering, Mysuru

Module2 : Data Manipulation with Pandas

- Introducing Pandas Objects,
- Handling Missing Data,
- Hierarchical Indexing,
- Pivot Tables.

Introducing Pandas Objects

- At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices.
- Pandas provides a host of useful tools, methods, and functionality on top of the basic data structures, but nearly everything that follows will require an understanding of what these structures are.
- Three fundamental Pandas data structures: The **Series, DataFrame, and Index**.
- The standard NumPy and Pandas imports:

```
import numpy as np
import pandas as pd
```

The Pandas Series Object

A Pandas Series is a one-dimensional array of indexed data. It can be created from a list or array as follows:

```
In[2]: data = pd.Series([0.25, 0.5, 0.75, 1.0])
       data

Out[2]: 0    0.25
        1    0.50
        2    0.75
        3    1.00
        dtype: float64
```

The Series wraps both a sequence of values and a sequence of indices, which we can access with the values and index attributes.

```
In[3]: data.values

Out[3]: array([ 0.25,  0.5 ,  0.75,  1.  ])
```

```
In[4]: data.index

Out[4]: RangeIndex(start=0, stop=4, step=1)
```

Introducing Pandas Objects

Data can be accessed by the associated index via square-bracket notation:

```
In[5]: data[1]

Out[5]: 0.5

In[6]: data[1:3]

Out[6]: 1    0.50
        2    0.75
        dtype: float64
```

The Pandas Series is much more general and flexible than the one-dimensional NumPy array

Introducing Pandas Objects

Series as generalized NumPy array

The essential difference is the presence of the index: while the NumPy array has an implicitly defined integer index used to access the values, the Pandas Series has an explicitly defined index associated with the values.

This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index

```
In[7]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                        index=['a', 'b', 'c', 'd'])
       data

Out[7]: a    0.25
        b    0.50
        c    0.75
        d    1.00
        dtype: float64
```

And the item access works as expected:

```
In[8]: data['b']

Out[8]: 0.5
```

We can even use noncontiguous or nonsequential indices:

```
In[9]: data = pd.Series([0.25, 0.5, 0.75, 1.0],
                        index=[2, 5, 3, 7])
       data

Out[9]: 2    0.25
        5    0.50
        3    0.75
        7    1.00
        dtype: float64

In[10]: data[5]

Out[10]: 0.5
```

Series as specialized dictionary

- A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a **Series** is a structure that maps typed keys to a set of typed values. The type information of a Pandas Series makes it much more efficient than Python dictionaries for certain operations.

```
In[11]: population_dict = {'California': 38332521,
                           'Texas': 26448193,
                           'New York': 19651127,
                           'Florida': 19552860,
                           'Illinois': 12882135}
        population = pd.Series(population_dict)
        population

Out[11]: California    38332521
         Florida       19552860
         Illinois      12882135
         New York      19651127
         Texas         26448193
         dtype: int64
```

- By default, a Series will be created where the index is drawn from the sorted keys. From here, typical dictionary-style item access can be performed:

```
In[12]: population['California']

Out[12]: 38332521
```

- The Series also supports array-style operations such as slicing:

```
In[13]: population['California':'Illinois']

Out[13]: California    38332521
         Florida       19552860
         Illinois      12882135
         dtype: int64
```

Introducing Pandas Objects

Constructing Series objects

```
>>> pd.Series(data, index=index)
```

where index is an optional argument, and data can be one of many entities. For example, data can be a list or NumPy array, in which case index defaults to an integer sequence

```
In[14]: pd.Series([2, 4, 6])

Out[14]: 0    2
         1    4
         2    6
         dtype: int64
```

Constructing Series objects

`data` can be a scalar, which is repeated to fill the specified index:

```
In[15]: pd.Series(5, index=[100, 200, 300])

Out[15]: 100    5
         200    5
         300    5
         dtype: int64
```

`data` can be a dictionary, in which `index` defaults to the sorted dictionary keys:

```
In[16]: pd.Series({2:'a', 1:'b', 3:'c'})

Out[16]: 1    b
         2    a
         3    c
         dtype: object
```

In each case, the index can be explicitly set if a different result is preferred:

```
In[17]: pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])

Out[17]: 3    c
         2    a
         dtype: object
```

The Pandas DataFrame Object:

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.

```
In[18]:
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}

area = pd.Series(area_dict)
area

Out[18]: California    423967
         Florida       170312
         Illinois      149995
         New York      141297
         Texas         695662
         dtype: int64
```

The Pandas DataFrame Object: DataFrame as a generalized NumPy array

```
In[19]: states = pd.DataFrame({'population': population,
                               'area': area})
        states

Out[19]:              area      population
         California   423967    38332521
         Florida      170312    19552860
         Illinois     149995    12882135
         New York     141297    19651127
         Texas        695662    26448193
```

Like the `Series` object, the `DataFrame` has an `index` attribute that gives access to the index labels:

```
In[20]: states.index

Out[20]:
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'], dtype='object')
```

The Pandas DataFrame Object: DataFrame as a generalized NumPy array

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

```
In[21]: states.columns

Out[21]: Index(['area', 'population'], dtype='object')
```

Thus the DataFrame can be thought of as a generalization of a two-dimensional NumPy array, where both the rows and columns have a generalized index for accessing the data.

The Pandas DataFrame Object: DataFrame as a specialized dictionary

We can also think of a DataFrame as a specialization of a dictionary. Where a **dictionary maps** a key to a value, a **DataFrame maps** a column name to a Series of column data. For example, asking for the **'area'** attribute returns the Series object containing the areas

```
In[22]: states['area']

Out[22]: California    423967
         Florida       170312
         Illinois      149995
         New York      141297
         Texas         695662
         Name: area, dtype: int64
```

In a two-dimensional **NumPy array, data[0]** will return the first row. For a **DataFrame**, **data['col0']** will return the first column.

Constructing DataFrame objects

A Pandas DataFrame can be constructed in a variety of ways.

1.From a single Series object: A DataFrame is a collection of Series objects, and a single column DataFrame can be constructed from a single Series:

```
In[23]: pd.DataFrame(population, columns=['population'])

Out[23]:              population
         California    38332521
         Florida       19552860
         Illinois      12882135
         New York      19651127
         Texas         26448193
```

Constructing DataFrame objects

2.From a list of dicts: Any list of dictionaries can be made into a DataFrame. We'll use a simple list comprehension to create some data:

```
In[24]: data = [{'a': i, 'b': 2 * i}
                for i in range(3)]
        pd.DataFrame(data)

Out[24]:    a  b
         0  0  0
         1  1  2
         2  2  4
```

Even if some keys in the dictionary are missing, Pandas will fill them in with NaN (i.e., "not a number") values

```
In[25]: pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])

Out[25]:    a    b  c
         0  1.0  2  NaN
         1  NaN  3  4.0
```

Constructing DataFrame objects

3.From a dictionary of Series objects: A DataFrame can be constructed from a dictionary of Series objects as well:

```
In[26]: pd.DataFrame({'population': population,
                      'area': area})

Out[26]:              area      population
        California  423967    38332521
        Florida     170312    19552860
        Illinois    149995    12882135
        New York    141297    19651127
        Texas       695662    26448193
```

```
In[27]: pd.DataFrame(np.random.rand(3, 2),
                     columns=['foo', 'bar'],
                     index=['a', 'b', 'c'])

Out[27]:     foo       bar
         a  0.865257  0.213169
         b  0.442759  0.108267
         c  0.047110  0.905718
```

4.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:

Constructing DataFrame objects

5.From a NumPy structured array: A Pandas DataFrame operates much like a structured array, and can be created directly from one:

```
In[28]: A = np.zeros(3, dtype=[('A', 'i8'), ('B', 'f8')])
        A

Out[28]: array([(0, 0.0), (0, 0.0), (0, 0.0)],
               dtype=[('A', '<i8'), ('B', '<f8')])

In[29]: pd.DataFrame(A)

Out[29]:    A  B
         0  0  0.0
         1  0  0.0
         2  0  0.0
```

The Pandas Index Object

The index object is considered as an **immutable array** or as an ordered set (technically a multiset, as Index objects may contain repeated values).

As a simple example, let's construct an Index from a list of integers:

```
In[30]: ind = pd.Index([2, 3, 5, 7, 11])
        ind

Out[30]: Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Index as immutable array

The Index object in many ways operates like an array. For example, we can use standard Python indexing notation to retrieve values or slices:

```
In[31]: ind[1]

Out[31]: 3

In[32]: ind[::2]

Out[32]: Int64Index([2, 5, 11], dtype='int64')
```

Index objects also have many of the attributes familiar from NumPy arrays:

```
In[33]: print(ind.size, ind.shape, ind.ndim, ind.dtype)

5 (5,) 1 int64
```

One difference between Index objects and NumPy arrays is that indices are immutable—that is, they cannot be modified via the normal means:

```
In[34]: ind[1] = 0
---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call last)

<ipython-input-34-40e631c82e8a> in <module>()
----> 1 ind[1] = 0


/Users/jakevdp/anaconda/lib/python3.5/site-packages/pandas/indexes/base.py ...
   1243
   1244     def __setitem__(self, key, value):
-> 1245         raise TypeError("Index does not support mutable operations")
   1246
   1247     def __getitem__(self, key):
```

This **immutability** makes it **safer to share** indices between multiple DataFrames and arrays, without the potential for side effects from inadvertent index modification.

Index as ordered set

Pandas objects are designed to facilitate operations such as **joins** across datasets, which depend on many aspects of set arithmetic.

The Index object follows many of the conventions used by Python's built-in set data structure, so that unions, intersections, differences, and other combinations can be computed in a familiar way:

```
In[35]: indA = pd.Index([1, 3, 5, 7, 9])
        indB = pd.Index([2, 3, 5, 7, 11])

In[36]: indA & indB  # intersection

Out[36]: Int64Index([3, 5, 7], dtype='int64')

In[37]: indA | indB  # union

Out[37]: Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')

In[38]: indA ^ indB  # symmetric difference

Out[38]: Int64Index([1, 2, 9, 11], dtype='int64')
```

These operations may also be accessed via object methods—for example, `indA.intersection(indB)`.

Handling Missing Data

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. In general Missing data is referred as **null, NaN, or NA values**.

Trade-Offs in Missing Data Conventions

A number of schemes have been developed to indicate the presence of missing data in a table or DataFrame. Generally, they revolve around one of two strategies: using a mask that globally indicates missing values, or choosing a sentinel value that indicates a missing entry.

In the **masking approach**, the mask might be an entirely separate Boolean array, or it may involve appropriation of one bit in the data representation to locally indicate the null status of a value.

In the **sentinel approach**, the sentinel value could be some data-specific convention, such as indicating a missing integer value with –9999 or some rare bit pattern, or it could be a more global convention, such as indicating a missing floating-point value with NaN (Not a Number), a special value which is part of the IEEE floating-point specification.

Handling Missing Data

None of these approaches is without trade-offs: use of a separate mask array requires allocation of an additional Boolean array, which adds overhead in both storage and computation. A sentinel value reduces the range of valid values that can be represented, and may require extra (often non-optimized) logic in CPU and GPU arithmetic. Common special values like NaN are not available for all data types. As in most cases where no universally optimal choice exists, different languages and systems use different conventions. For example, the **R language** uses reserved bit patterns within each data type as sentinel values indicating missing data, while the SciDB system uses an extra byte attached to every cell to indicate a NA state.

Handling Missing Data

Missing Data in Pandas

Pandas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floating point NaN value, and the Python None object.

None: Pythonic missing data

The first sentinel value used by Pandas is None, a Python singleton object that is often used for missing data in Python code. Because None is a Python object, it cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type 'object' (i.e., arrays of Python objects):

```
In[1]: import numpy as np
       import pandas as pd

In[2]: vals1 = np.array([1, None, 3, 4])
       vals1

Out[2]: array([1, None, 3, 4], dtype=object)
```

This dtype = object means that the best common type representation NumPy could infer for the contents of the array is that they are Python objects.

Handling Missing Data

```
In[3]: for dtype in ['object', 'int']:
           print("dtype =", dtype)
           %timeit np.arange(1E6, dtype=dtype).sum()
           print()

dtype = object
10 loops, best of 3: 78.2 ms per loop

dtype = int
100 loops, best of 3: 3.06 ms per loop
```

The use of Python objects in an array also means that if you perform aggregations like sum() or min() across an array with a None value, you will generally get an error:

```
In[4]: vals1.sum()

TypeError                                 Traceback (most recent call last)

<ipython-input-4-749fd8ae6030> in <module>()
----> 1 vals1.sum()
```

This reflects the fact that addition between an integer and None is undefined.

Handling Missing Data

NaN: Missing numerical data

The other missing data representation, NaN (acronym for Not a Number), is different; it is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation:

```
In[5]: vals2 = np.array([1, np.nan, 3, 4])
       vals2.dtype

Out[5]: dtype('float64')
```

Notice that NumPy chose a native floating-point type for this array: this means that unlike the object array from before, this array supports fast operations pushed into compiled code. You should be aware that NaN is a bit like a data virus—it infects any other object it touches. Regardless of the operation, the result of arithmetic with NaN will be another NaN:

```
In[6]: 1 + np.nan

Out[6]: nan

In[7]: 0 *  np.nan

Out[7]: nan
```

Handling Missing Data

Note that this means that aggregates over the values are well defined (i.e., they don't result in an error) but not always useful:

```
In[8]: vals2.sum(), vals2.min(), vals2.max()
Out[8]: (nan, nan, nan)
```

NumPy does provide some special aggregations that will ignore these missing values:

```
In[9]: np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
Out[9]: (8.0, 1.0, 4.0)
```

Keep in mind that `NaN` is specifically a floating-point value; there is no equivalent NaN value for integers, strings, or other types.

Handling Missing Data

NaN and None in Pandas

NaN and None both have their place, and Pandas is built to handle the two of them nearly interchangeably, converting between them where appropriate:

```
In[10]: pd.Series([1, np.nan, 2, None])

Out[10]: 0    1.0
         1    NaN
         2    2.0
         3    NaN
         dtype: float64
```

For types that don't have an available sentinel value, Pandas automatically type-casts when NA values are present.

For example, if we set a value in an integer array to np.nan, it will automatically be upcast to a floating-point type to accommodate the NA:

Handling Missing Data

```
In[11]: x = pd.Series(range(2), dtype=int)
        x

Out[11]: 0    0
         1    1
         dtype: int64

In[12]: x[0] = None
        x

Out[12]: 0    NaN
         1    1.0
         dtype: float64
```

Notice that in addition to casting the integer array to floating point, Pandas automatically converts the None to a NaN value. Table 3-2 lists the upcasting conventions in Pandas when NA values are introduced.

Table 3-2. Pandas handling of NAs by type

Typeclass	Conversion when storing NAs	NA sentinel value
`floating`	No change	`np.nan`
`object`	No change	None or `np.nan`
`integer`	Cast to `float64`	`np.nan`
`boolean`	Cast to `object`	None or `np.nan`

Operating on Null Values

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

Detecting null values

Pandas data structures have two useful methods for detecting null data: isnull() and notnull().

```
In[13]: data = pd.Series([1, np.nan, 'hello', None])

In[14]: data.isnull()

Out[14]: 0    False
         1     True
         2    False
         3     True
         dtype: bool
```

Handling Missing Data

```
In[15]: data[data.notnull()]

Out[15]: 0         1
         2     hello
         dtype: object
```

The isnull() and notnull() methods produce similar Boolean results for Data Frames.

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). For a Series, the result is straightforward:

```
In[16]: data.dropna()

Out[16]: 0         1
         2     hello
         dtype: object
```

Handling Missing Data

For a DataFrame, there are more options. Consider the following DataFrame:

```
In[17]: df = pd.DataFrame([[1,      np.nan, 2],
                           [2,      3,      5],
                           [np.nan, 4,      6]])
        df

Out[17]:      0    1  2
          0  1.0  NaN  2
          1  2.0  3.0  5
          2  NaN  4.0  6
```

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:

```
In[18]: df.dropna()

Out[18]:      0    1  2
          1  2.0  3.0  5
```

Handling Missing Data

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

```
In[19]: df.dropna(axis='columns')

Out[19]:    2
         0  2
         1  5
         2  6
```

```
In[20]: df[3] = np.nan
        df

Out[20]:     0    1  2   3
         0  1.0  NaN  2 NaN
         1  2.0  3.0  5 NaN
         2  NaN  4.0  6 NaN

In[21]: df.dropna(axis='columns', how='all')

Out[21]:     0    1  2
         0  1.0  NaN  2
         1  2.0  3.0  5
         2  NaN  4.0  6
```

But this drops some good data as well; you might rather be interested in dropping rows or columns with all NA values, or a majority of NA values. This can be specified through the **how or thresh** parameters, which allow fine control of the number of nulls to allow through.

The default is how='any', such that any row or column (depending on the axis key word) containing a null value will be dropped. You can also specify how='all', which will only drop rows/columns that are all null values

Handling Missing Data

For finer-grained control, the thresh parameter lets you specify a minimum number of non-null values for the row/column to be kept:

```
In[22]: df.dropna(axis='rows', thresh=3)

Out[22]:      0    1  2   3
          1  2.0  3.0  5 NaN
```

Here the first and last row have been dropped, because they contain only two non null values.

Handling Missing Data

Filling null values

Instead of dropping NA values, replace them with a valid value. Pandas provides the fillna() method, which returns a copy of the array with the null values replaced.

Consider the following Series:

```
In[23]: data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
        data

Out[23]: a    1.0
         b    NaN
         c    2.0
         d    NaN
         e    3.0
         dtype: float64
```

We can fill NA entries with a single value, such as zero:

```
In[24]: data.fillna(0)

Out[24]: a    1.0
         b    0.0
         c    2.0
         d    0.0
         e    3.0
         dtype: float64
```

Handling Missing Data

We can specify a forward-fill to propagate the previous value forward:

```
In[25]: # forward-fill
        data.fillna(method='ffill')

Out[25]: a    1.0
         b    1.0
         c    2.0
         d    2.0
         e    3.0
         dtype: float64
```

Or we can specify a back-fill to propagate the next values backward:

```
In[26]: # back-fill
        data.fillna(method='bfill')

Out[26]: a    1.0
         b    2.0
         c    2.0
         d    3.0
         e    3.0
         dtype: float64
```

Handling Missing Data

For DataFrames, the options are similar, but we can also specify an axis along which the fills take place:

```
In[27]: df

Out[27]:      0    1  2   3
          0  1.0  NaN  2 NaN
          1  2.0  3.0  5 NaN
          2  NaN  4.0  6 NaN

In[28]: df.fillna(method='ffill', axis=1)

Out[28]:      0    1    2    3
          0  1.0  1.0  2.0  2.0
          1  2.0  3.0  5.0  5.0
          2  NaN  4.0  6.0  6.0
```

Notice that if a previous value is not available during a forward fill, the NA value remains.

Hierarchical Indexing

So far we have discussed about one-dimensional and two dimensional data, stored in Pandas Series and DataFrame objects, respectively.

Often it is useful to go beyond this and store higher-dimensional data—that is, data indexed by more than one or two keys. While Pandas does provide Panel and Panel4D objects that natively handle three-dimensional and four-dimensional data.

A far more common pattern in practice is to make use of hierarchical indexing (also known as multi-indexing) to incorporate multiple index levels within a single index.

In this way, higher-dimensional data can be compactly represented within the familiar one-dimensional Series and two-dimensional DataFrame objects.

Hierarchical Indexing

A Multiply Indexed Series

Represent two-dimensional data within a one-dimensional Series. Consider a series of data where each point has a character and numerical key.

The bad way

Suppose you would like to track data about states from two different years. Using the Pandas tools we've already covered, you might be tempted to simply use Python tuples as keys:

```
In[2]: index = [('California', 2000), ('California', 2010),
                ('New York', 2000), ('New York', 2010),
                ('Texas', 2000), ('Texas', 2010)]
       populations = [33871648, 37253956,
                      18976457, 19378102,
                      20851820, 25145561]
       pop = pd.Series(populations, index=index)
       pop
```

```
Out[2]: (California, 2000)    33871648
        (California, 2010)    37253956
        (New York, 2000)      18976457
        (New York, 2010)      19378102
        (Texas, 2000)         20851820

        (Texas, 2010)         25145561
        dtype: int64
```

Hierarchical Indexing

With this indexing scheme, you can straightforwardly index or slice the series based on this multiple index:

```
In[3]: pop[('California', 2010):('Texas', 2000)]

Out[3]: (California, 2010)    37253956
        (New York, 2000)      18976457
        (New York, 2010)      19378102
        (Texas, 2000)         20851820
        dtype: int64
```

But the convenience ends there. For example, if you need to select all values from 2010, you'll need to do some messy (and potentially slow) munging to make it happen:

```
In[4]: pop[[i for i in pop.index if i[1] == 2010]]

Out[4]: (California, 2010)    37253956
        (New York, 2010)      19378102
        (Texas, 2010)         25145561
        dtype: int64
```

This produces the desired result, but is not as clean (or as efficient for large datasets) as the slicing syntax we've grown to love in Pandas.

Hierarchical Indexing

The better way: Pandas MultiIndex

Fortunately, Pandas provides a better way. Our tuple-based indexing is essentially a rudimentary multi-index, and the Pandas MultiIndex type gives us the type of operations we wish to have. We can create a multi-index from the tuples as follows:

```
In[5]: index = pd.MultiIndex.from_tuples(index)
       index

Out[5]: MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],
                   labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

Notice that the MultiIndex contains multiple levels of indexing—in this case, the state names and the years, as well as multiple labels for each data point which encode these levels.

If we reindex our series with this MultiIndex, we see the hierarchical representation of the data:
Here the first two columns of the Series representation show the multiple index values, while the third column shows the data. Notice that some entries are missing in the first column: in this multi-index representation, any blank entry indicates the same value as the line above it.

```
In[6]: pop = pop.reindex(index)
       pop

Out[6]: California  2000    33871648
                    2010    37253956
        New York    2000    18976457
                    2010    19378102
        Texas       2000    20851820
                    2010    25145561
        dtype: int64
```

Hierarchical Indexing

Now to access all data for which the second index is 2010, we can simply use the Pandas slicing notation:

```
In[7]: pop[:, 2010]

Out[7]: California    37253956
        New York      19378102
        Texas         25145561
        dtype: int64
```

The result is a singly indexed array with just the keys we're interested in.

This syntax is much more convenient (and the operation is much more efficient!) than the home spun tuple-based multi-indexing solution that we started with.

Hierarchical Indexing

MultiIndex as extra dimension

The unstack() method will quickly convert a multiply indexed Series into a conventionally indexed DataFrame:

```
In[8]: pop_df = pop.unstack()
       pop_df

Out[8]:                2000      2010
       California  33871648  37253956
       New York    18976457  19378102
       Texas       20851820  25145561
```

Naturally, the stack() method provides the opposite operation:

```
In[9]: pop_df.stack()

Out[9]:  California  2000    33871648
                    2010    37253956
         New York   2000    18976457
                    2010    19378102
         Texas      2000    20851820
                    2010    25145561
         dtype: int64
```

Hierarchical Indexing

MultiIndex as extra dimension

we can also use multi-indexing to represent data of three or more dimensions in a Series or DataFrame. Each extra level in a multi-index represents an extra dimension of data; taking advantage of this property gives us much more flexibility in the types of data we can represent.

Concretely, we might want to add another column of demographic data for each state at each year (say, population under 18); with a MultiIndex this is as easy as adding another column to the DataFrame:

```
In[10]: pop_df = pd.DataFrame({'total': pop,
                               'under18': [9267089, 9284094,
                                           4687374, 4318033,
                                           5906301, 6879014]})
        pop_df
Out[10]:                    total  under18
        California 2000  33871648  9267089
                   2010  37253956  9284094
        New York   2000  18976457  4687374
                   2010  19378102  4318033
        Texas      2000  20851820  5906301
                   2010  25145561  6879014
```

Hierarchical Indexing

MultiIndex as extra dimension

Here we compute the fraction of people under 18 by year, given the above data:

```
In[11]: f_u18 = pop_df['under18'] / pop_df['total']
        f_u18.unstack()

Out[11]:                2000      2010
         California  0.273594  0.249211
         New York    0.247010  0.222831
         Texas       0.283251  0.273568
```

This allows us to easily and quickly manipulate and explore even high-dimensional data.

Hierarchical Indexing

Methods of MultiIndex Creation

The most straightforward way to construct a multiply indexed Series or DataFrame is to simply pass a list of two or more index arrays to the constructor. For example

```
In[12]: df = pd.DataFrame(np.random.rand(4, 2),
                          index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                          columns=['data1', 'data2'])
        df

Out[12]:          data1     data2
        a 1  0.554233  0.356072
          2  0.925244  0.219474
        b 1  0.441759  0.610054
          2  0.171495  0.886688
```

Hierarchical Indexing

Methods of MultiIndex Creation

The work of creating the MultiIndex is done in the background. Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a MultiIndex by default:

```
In[13]: data = {('California', 2000): 33871648,
                ('California', 2010): 37253956,
                ('Texas', 2000): 20851820,
                ('Texas', 2010): 25145561,
                ('New York', 2000): 18976457,
                ('New York', 2010): 19378102}
        pd.Series(data)

Out[13]: California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

Sometimes it is useful to explicitly create a MultiIndex

Hierarchical Indexing

Explicit MultiIndex constructors

pd.MultiIndex can be used to construct MultiIndex. For example, we can construct the MultiIndex from a simple list of arrays, giving the index values within each level:

```
In[14]: pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])

Out[14]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

We can construct it from a list of tuples, giving the multiple index values of each point:

```
In[15]: pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])

Out[15]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

We can even construct it from a Cartesian product of single indices:

```
In[16]: pd.MultiIndex.from_product([['a', 'b'], [1, 2]])

Out[16]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

Hierarchical Indexing

Explicit MultiIndex constructors

Similarly, we can construct the MultiIndex directly using its internal encoding by passing levels (a list of lists containing available index values for each level) and labels (a list of lists that reference these labels)

```
In[17]: pd.MultiIndex(levels=[['a', 'b'], [1, 2]],
                      labels=[[0, 0, 1, 1], [0, 1, 0, 1]])

Out[17]: MultiIndex(levels=[['a', 'b'], [1, 2]],
                    labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

We can pass any of these objects as the index argument when creating a Series or DataFrame, or to the reindex method of an existing Series or DataFrame.

Hierarchical Indexing

MultiIndex level names

Sometimes it is convenient to name the levels of the MultiIndex. We can accomplish this by passing the names argument to any of the above MultiIndex constructors, or by setting the names attribute of the index after the fact:

```
In[18]: pop.index.names = ['state', 'year']
        pop

Out[18]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

With more involved datasets, this can be a useful way to keep track of the meaning of various index values.

Hierarchical Indexing

MultiIndex for columns

In a DataFrame, the rows and columns are completely symmetric, and just as the rows can have multiple levels of indices, the columns can have multiple levels as well. Consider the following, which is a mock-up of some (somewhat realistic) medical data:

```
# hierarchical indices and columns
 index = pd.MultiIndex.from_product([[2013, 2014], [1, 2]], names=['year', 'visit'])
columns = pd.MultiIndex.from_product([[‘Jhon', ‘Peter', ‘Thomas'], ['HR', 'Temp']], names=['subject', 'type'])
# mock some data
data = np.round(np.random.randn(4, 6), 1)
data[:, ::2] *= 10
data += 37
# create the DataFrame
health_data = pd.DataFrame(data, index=index, columns=columns)
health_data
```

	subject	Jhon		Peter		Thomas	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	14.0	38.5	39.0	37.0	33.0	38.9
	2	19.0	36.7	40.0	35.3	52.0	35.7
2014	1	45.0	38.3	39.0	37.7	35.0	36.9
	2	20.0	39.2	55.0	37.2	49.0	35.5

Hierarchical Indexing

MultiIndex for columns

Here we see where the multi-indexing for both rows and columns can come in very handy. This is fundamentally four-dimensional data, where the dimensions are the subject, the measurement type, the year, and the visit number. With this in place we can, for example, index the top-level column by the person's name and get a full Data Frame containing just that person's information:

```
health_data['Peter']
```

		type	HR	Temp
year	visit			
2013	1		39.0	37.0
	2		40.0	35.3
2014	1		39.0	37.7
	2		55.0	37.2

Hierarchical Indexing

Indexing and Slicing a MultiIndex

Multiply indexed Series

Consider the multiply indexed Series of state populations

```
In[21]: pop
Out[21]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

We can access single elements by indexing with multiple terms:

```
In[22]: pop['California', 2000]
Out[22]: 33871648
```

Module 2 : Data Manipulation with Pandas

Hierarchical Indexing

Multiply indexed Series

The MultiIndex also supports partial indexing, or indexing just one of the levels in the index. The result is another Series, with the lower-level indices maintained

```
In[23]: pop['California']

Out[23]: year
         2000    33871648
         2010    37253956
         dtype: int64
```

Partial slicing is also possible.

```
In[24]: pop.loc['California':'New York']

Out[24]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         dtype: int64
```

With sorted indices, we can perform partial indexing on lower levels by passing an empty slice in the first index

```
In[25]: pop[:, 2000]

Out[25]: state
         California    33871648
         New York      18976457
         Texas         20851820
         dtype: int64
```

Hierarchical Indexing

Multiply indexed Series

Other types of indexing and selection is also possible; for example, selection based on Boolean masks:

```
In[26]: pop[pop > 22000000]

Out[26]: state       year
         California  2000    33871648
                     2010    37253956
         Texas       2010    25145561
         dtype: int64
```

Selection based on fancy indexing also works:

```
In[27]: pop[['California', 'Texas']]

Out[27]: state       year
         California  2000    33871648
                     2010    37253956
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

Hierarchical Indexing

Multiply indexed DataFrames

A multiply indexed DataFrame behaves in a similar manner. Consider our toy medical DataFrame

health_data

	subject		Jhon		Peter		Thomas	
	type		HR	Temp	HR	Temp	HR	Temp
year	visit							
2013	1		14.0	38.5	39.0	37.0	33.0	38.9
	2		19.0	36.7	40.0	35.3	52.0	35.7
2014	1		45.0	38.3	39.0	37.7	35.0	36.9
	2		20.0	39.2	55.0	37.2	49.0	35.5

```
health_data['Jhon', 'HR']
```

```
year  visit
2013  1        40.0
      2        30.0
2014  1        62.0
      2        27.0
Name: (Jhon, HR), dtype: float64
```

```
health_data.iloc[:2, :2]
```

	subject	Jhon	
	type	HR	Temp
year	visit		
2013	1	40.0	36.6
	2	30.0	37.5

Remember that columns are primary in a DataFrame, and the syntax used for multiply indexed Series applies to the columns. For example, we can recover Jhon's heart rate data with a simple operation:

Also, as with the single-index case, we can use the loc, iloc, and ix indexers. The loc function is lable –based. We can select rows columns by their labels(actual names of the rows and columns).

The iloc function is integrated-based. It allows you to select rows and columns by their integer positions(The numerical index of the rows and columns).

Hierarchical Indexing

Multiply indexed DataFrames

These indexers provide an array-like view of the underlying two-dimensional data, but each individual index in loc or iloc can be passed a tuple of multiple indices. For example:

```
health_data.loc[:, ('Jhon', 'HR')]
```

```
year  visit
2013  1        40.0
      2        30.0
2014  1        62.0
      2        27.0
Name: (Jhon, HR), dtype: float64
```

```
health_data.loc[:,'Jhon':'Peter']
```

	subject	Jhon		Peter	
	type	HR	Temp	HR	Temp
year	visit				
2013	1	40.0	36.6	36.0	38.9
	2	30.0	37.5	39.0	39.2
2014	1	62.0	36.1	20.0	38.0
	2	27.0	37.5	32.0	38.3

Hierarchical Indexing

Multiply indexed DataFrames

Working with slices within these index tuples is not especially convenient; trying to create a slice within a tuple will lead to a syntax error:

```
In[32]: health_data.loc[(:, 1), (:, 'HR')]

  File "<ipython-input-32-8e3cc151e316>", line 1
    health_data.loc[(:, 1), (:, 'HR')]
                    ^
SyntaxError: invalid syntax
```

```
idx = pd.IndexSlice
health_data.loc[idx[:, 1], idx[:, 'HR']]
```

	subject	Jhon	Peter	Thomas
	type	HR	HR	HR
year	visit			
2013	1	40.0	36.0	41.0
2014	1	62.0	20.0	36.0

You could get around this by building the desired slice explicitly using Python's built in slice() function, but a better way in this context is to use an IndexSlice object, which Pandas provides for precisely this situation. For example:

Hierarchical Indexing

Rearranging Multi-Indices

There are many ways to control the rearrangement of data between hierarchical indices and columns.

- Sorted and unsorted indices
- Stacking and unstacking indices
- Index setting and resetting

Sorted and unsorted indices

Many of the MultiIndex slicing operations will fail if the index is not sorted. Let's take a look at this here. We'll start by creating some simple multiply indexed data where the indices are not lexographically sorted

```
In[34]: index = pd.MultiIndex.from_product([['a', 'c', 'b'], [1, 2]])
        data = pd.Series(np.random.rand(6), index=index)
        data.index.names = ['char', 'int']
        data

Out[34]: char  int
         a     1      0.003001
               2      0.164974
         c     1      0.741650
               2      0.569264
         b     1      0.001693
               2      0.526226
         dtype: float64
```

Although it is not entirely clear from the error message, this is the result of the Multi Index not being sorted. For various reasons, partial slices and other similar operations require the levels in the MultiIndex to be in sorted (i.e., lexographical) order.

```
In[35]: try:
            data['a':'b']
        except KeyError as e:
            print(type(e))
            print(e)

<class 'KeyError'>
'Key length (1) was greater than MultiIndex lexsort depth (0)'
```

Hierarchical Indexing

Rearranging Multi-Indices

Pandas provides a number of convenience routines to perform this type of sorting; examples are the sort_index() and sortlevel() methods of the DataFrame. We'll use the simplest, sort_index(), here:

```
In[36]: data = data.sort_index()
        data

Out[36]: char  int
         a     1       0.003001
               2       0.164974
         b     1       0.001693
               2       0.526226
         c     1       0.741650
               2       0.569264
         dtype: float64
```

With the index sorted in this way, partial slicing will work as expected:

```
In[37]: data['a':'b']

Out[37]: char  int
         a     1       0.003001
               2       0.164974
         b     1       0.001693
               2       0.526226
         dtype: float64
```

Hierarchical Indexing

Rearranging Multi-Indices

Stacking and unstacking indices

it is possible to convert a dataset from a stacked multi-index to a simple two-dimensional representation, optionally specifying the level to use

```
In[38]: pop.unstack(level=0)

Out[38]:  state  California   New York     Texas
          year
          2000     33871648   18976457  20851820
          2010     37253956   19378102  25145561

In[39]: pop.unstack(level=1)

Out[39]: year            2000      2010
         state
         California  33871648  37253956
         New York    18976457  19378102
         Texas       20851820  25145561
```

The opposite of unstack() is stack(), which here can be used to recover the original series:

```
In[40]: pop.unstack().stack()

Out[40]: state       year
         California  2000    33871648
                     2010    37253956
         New York    2000    18976457
                     2010    19378102
         Texas       2000    20851820
                     2010    25145561
         dtype: int64
```

Hierarchical Indexing

Rearranging Multi-Indices

Index setting and resetting

Another way to rearrange hierarchical data is to turn the index labels into columns; this can be accomplished with the reset_index method. Calling this on the population dictionary will result in a DataFrame with a state and year column holding the information that was formerly in the index. For clarity, we can optionally specify the name of the data for the column representation:

```
In[41]: pop_flat = pop.reset_index(name='population')
        pop_flat

Out[41]:         state  year  population
         0  California  2000    33871648
         1  California  2010    37253956
         2    New York  2000    18976457
         3    New York  2010    19378102
         4       Texas  2000    20851820
         5       Texas  2010    25145561
```

Hierarchical Indexing

Index setting and resetting

Often when you are working with data in the real world, the raw input data looks like this and it's useful to build a MultiIndex from the column values. This can be done with the set_index method of the DataFrame, which returns a multiply indexed Data Frame

```
In[42]: pop_flat.set_index(['state', 'year'])

Out[42]:                    population
        state      year
        California 2000    33871648
                   2010    37253956
        New York   2000    18976457
                   2010    19378102
        Texas      2000    20851820
                   2010    25145561
```

Hierarchical Indexing

Data Aggregations on Multi-Indices

Pandas has built-in data aggregation methods, such as mean(), sum(), and max(). For hierarchically indexed data, these can be passed a level parameter that controls which subset of the data the aggregate is computed on.

For example, let's return to our health data:

```
health_data
```

	subject	Jhon		Peter		Thomas	
	type	HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	43.0	36.6	52.0	36.2	20.0	38.1
	2	23.0	36.5	31.0	38.2	32.0	36.7
2014	1	46.0	38.3	25.0	36.9	40.0	36.6
	2	49.0	36.7	58.0	36.4	36.0	35.9

Perhaps we'd like to average out the measurements in the two visits each year. We can do this by naming the index level we'd like to explore, in this case the year:

```
data_mean = health_data.mean(level='year')
data_mean
```

subject	Jhon		Peter		Thomas	
type	HR	Temp	HR	Temp	HR	Temp
year						
2013	33.0	36.55	41.5	37.20	26.0	37.40
2014	47.5	37.50	41.5	36.65	38.0	36.25

Hierarchical Indexing

Data Aggregations on Multi-Indices

By further making use of the axis keyword, we can take the mean among levels on the columns as well

```
data_mean.mean(axis=1, level='type')
```

type	HR	Temp
year		
2013	33.500000	37.05
2014	42.333333	36.80

Pivot Tables

The pivot table takes simple column wise data as input, and groups the entries into a two-dimensional table that provides a multidimensional summarization of the data.

Pivot Table allow you to perform common aggregate statistical calculations such as sums, counts, averages, and so on.

Motivating Pivot Tables

```
import numpy as np
import pandas as pd
```

```
titanic=pd.read_csv('tested.csv')
```

```
titanic.head()
```

This contains a wealth of information on each passenger of that ill-fated voyage, including gender, age, class, fare paid, and much more.

	PassengerId	Survived	Pclass	Name	Gender	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	892	0	3	Kelly, Mr. James	male	34.5	0	0	330911	7.8292	NaN	Q
1	893	1	3	Wilkes, Mrs. James (Ellen Needs)	female	47.0	1	0	363272	7.0000	NaN	S
2	894	0	2	Myles, Mr. Thomas Francis	male	62.0	0	0	240276	9.6875	NaN	Q
3	895	0	3	Wirz, Mr. Albert	male	27.0	0	0	315154	8.6625	NaN	S
4	896	1	3	Hirvonen, Mrs. Alexander (Helga E Lindqvist)	female	22.0	1	1	3101298	12.2875	NaN	S

Pivot Tables

Pivot Tables by Hand

To start learning more about this data, we might begin by grouping it according to gender, Age status, or some combination thereof.

Example

```
#Pivot Tables by Hand
titanic.groupby('Gender')[['Age']].mean()
```

Gender	Age
female	30.272362
male	30.272732

Pivot Tables

Pivot Tables by Hand

We group by class and gender, select Age, apply a mean aggregate, combine the resulting groups, and then unstack the hierarchical index to reveal the hidden multidimensionality.
Example:

```
titanic.groupby(['Gender', 'Pclass'])['Age'].aggregate('mean').unstack()
```

Pclass	1	2	3
Gender			
female	41.333333	24.376552	23.073400
male	40.520000	30.940678	24.525104

Pivot Tables

Pivot Table Syntax

Here is the equivalent to the preceding operation using the pivot_table method of DataFrames

```
titanic.pivot_table('Age', index='Gender', columns='Pclass')
```

Pclass	1	2	3
Gender			
female	41.333333	24.376552	23.073400
male	40.520000	30.940678	24.525104

This is eminently more readable than the GroupBy approach, and produces the same result.

Pivot Tables

Multilevel pivot tables

Just as in the GroupBy, the grouping in pivot tables can be specified with multiple levels, and via a number of options. For example, we might be interested in looking at age as a third dimension. We'll bin the age using the pd.cut function:

```
age = pd.cut(titanic['Age'], [0, 18, 80])
titanic.pivot_table('Survived', ['Gender', age], 'Pclass')
```

	Pclass	1	2	3
Gender	Age			
female	(0, 18]	1	1	1
	(18, 80]	1	1	1
male	(0, 18]	0	0	0
	(18, 80]	0	0	0

We can apply this same strategy when working with the columns as well; let's add info on the fare paid using pd.qcut to automatically compute quantiles:

```
fare = pd.qcut(titanic['Fare'], 2)
titanic.pivot_table('Survived', ['Gender', age], [fare, 'Pclass'])
```

	Fare	(-0.001, 14.454]			(14.454, 512.329]		
	Pclass	1	2	3	1	2	3
Gender	Age						
female	(0, 18]	NaN	1.0	1.0	1.0	1.0	1.0
	(18, 80]	NaN	1.0	1.0	1.0	1.0	1.0
male	(0, 18]	NaN	0.0	0.0	0.0	0.0	0.0
	(18, 80]	0.0	0.0	0.0	0.0	0.0	0.0

Pivot Tables

Additional pivot table options

The full call signature/syntax of the pivot_table method of DataFrames is as follows

DataFrame.pivot_table(data, values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All')

The aggfunc keyword controls what type of aggregation is applied, which is a mean by default. As in the GroupBy, the aggregation specification can be a string representing one of several common choices ('sum', 'mean', 'count', 'min', 'max', etc.) or a function that implements an aggregation (np.sum(), min(), sum(), etc.). Additionally, it can be specified as a dictionary mapping a column to any of the above desired options:

```
titanic.pivot_table(index='Gender', columns='Pclass',aggfunc={'Survived':sum, 'Fare':'mean'})
```

	Fare			Survived		
Pclass	1	2	3	1	2	3
Gender						
female	115.591168	26.438750	13.735129	50	30	72
male	75.586551	20.184654	11.826350	0	0	0

Pivot Tables

Additional pivot table options

At times it's useful to compute totals along each grouping. This can be done via the margins keyword:

```
titanic.pivot_table('Survived', index='Gender', columns='Pclass', margins=True)
```

Pclass	1	2	3	All
Gender				
female	1.00000	1.000000	1.000000	1.000000
male	0.00000	0.000000	0.000000	0.000000
All	0.46729	0.322581	0.330275	0.363636

Here this automatically gives us information about the class-agnostic survival rate by gender, the gender-agnostic survival rate by class, and the overall survival rate of 36%. The margin label can be specified with the margins_name keyword, which defaults to "All".

Pivot Tables

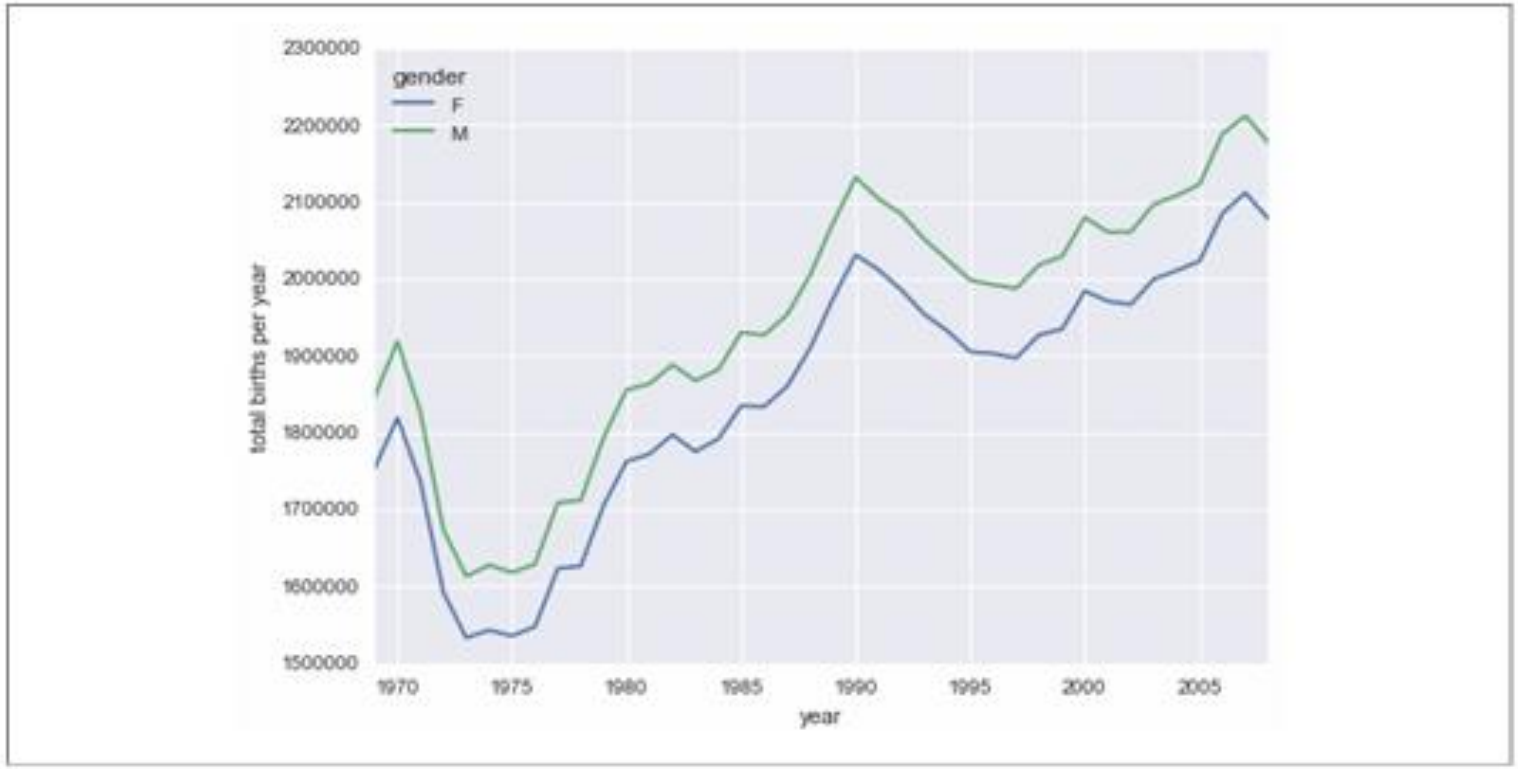
Example: Birthrate Data

data on births in the United States, provided by the Centers for Disease Control (CDC)

```
births = pd.read_csv('births.csv')
births.head()
births['decade'] = 10 * (births['year'] // 10)
births.pivot_table('births', index='decade', columns='gender', aggfunc='sum')
%matplotlib inline
import matplotlib.pyplot as plt
 sns.set() # use Seaborn styles
births.pivot_table('births', index='year', columns='gender', aggfunc='sum').plot()
plt.ylabel('total births per year');
```

Pivot Tables

Example: Birthrate Data



Total number of US births by year and gender

Pivot Tables

Further data exploration

There are a few more interesting features we can pull out of this dataset using the Pandas tools We must start by cleaning the data a bit, removing outliers caused by mistyped dates (e.g., June 31st) or missing values (e.g., June 99th). One easy way to remove these all at once is to cut outliers; we'll do this via a robust sigma-clipping operation:

THANK YOU