Data Analysis Packages

NYC Data Science Academy

OVERVIEW

- **NumPy**
  - Ndarray
  - Subscripting and slicing
  - Operations
- Pandas
  - Data Structure
  - Data Manipulation
  - Grouping and aggregation

NumPy: Overview

- NumPy is the fundamental package for scientific computing with Python.
  - Primitive data types are often collected and arranged in certain ways to facilitate analysis. A common example are vectors and matrices in linear algebra. Numpy provide easy tools to achieve it.
  - Numpy provides many functions and modules for scientific uses, such as linear algebra operations, random number generation, Fourier transform, etc.
- For full documentation, go to: http://docs.scipy.org/doc/.

NumPy: Overview

- As a module, you need to import it to make the functions it has accessible. You can do it by run the following code:

```
import numpy as np # later you can refer the package as np
```

- Whenever you need to call a function from Numpy, say `matrix()`, you could run:

```
np.arange( arguments )     # The function arange will be
                           # discussed later
```

OVERVIEW

- NumPy
  - **Ndarray**
    - Subscripting and slicing
    - Operations
- Pandas
  - Data Structure
  - Data Manipulation
  - Grouping and aggregation

Data type: ndarray

- NumPy provides an N-dimensional array type, the *ndarray*. It can be described as a collection of elements of the same data type. We will discuss more about the creation of ndarray, but for the illustrating purpose, let's construct a simple example with the function `array()`

```
my_ary=np.array([1,2,3])
```

Data type: ndarray

- We see that `array()` takes a list as an argument. Then `my_ary` is a ndarray with three elements ordered exactly the same as the list inputted.

```
type(my_ary)
numpy.ndarray
my_ary
array([1,2,3])
```

- As you might suspect, each element in `my_ary` can be accessed by integer index. Details of indexing will be provided later.

```
print my_ary[0]  # In Python, first index is 0
print my_ary[1]
1
2
```

Data type: homogeneity and casting

- The type ndarray is flexible because it supports great variety of primitive data types (list of data types) subject to the homogeneity condition, which means every element in ndarray have the same type. The type of the elements can be accessed as an attribute of an ndarray.

```
my_ary.dtype
dtype('int64')
```

- When the elements are of string types, it print the dtype automatically. The letter 'S' stands for string and the number followed represents the largest number of letters among all the elements.

```
np.array(['a', 'ab'])
array(['a', 'ab'], dtype='|S2')
```

Data type: homogeneity and casting

- If the list inputted to the function `array()` is of mixed types, Numpy (upcast) cast all elements to a common (superclass) class:

```
np.array([1.0, 2, 3])
array([1.0, 2.0, 3.0])
# cast to float; The two integers included becomes
# float to meet the homogeneous condition.

np.array(['a', 2, 3])
array(['a', '2', '3'], dtype='|S1')
# Again for the homogeneity, all are casted to string.
```

Creating data array: 2 dimensional

- We have already seen a way to generate ndarray by plugging a list into the function `array()`. This method can be actually generalized to create multidimensional array by plugging nested list instead of simple list into the `array()` function.

```
simple_lst = [1,2]
nested_lst = [[1,2], [3,4]]
multi_ary = np.array(nested_lst)
array([[1, 2],
       [3, 4]])
```

- Recall that nested list is a list having lists as elements. In this particular case, multidimensional ndarray is designed to mimic mathematical matrices, so each inner list in `nested_lst` is like a row in a matrix.

Creating data array: Regular arrays creation

- So far we have been creating arrays mainly by specifying every element. When creating larger arrays this is not practical. Numpy provides functions which create functions by certain rules. We discuss some of them below.
- The simplest way to create a Numpy array is by the function `arange()`.

```
# np.arange(n) return an array from 0 to n-1
np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- Plugging float into `arange()` result in array upto the greatest number smaller than the float.

```
np.arange(9.5)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Creating data array: Regular arrays creation

- We don't have to always start from 0. The first argument below indicates the array starts from 2 and the second argument indicates it stops at 9 (10-1).

```
np.arange(2, 10)
array([2, 3, 4, 5, 6, 7, 8, 9])
```

- And step between terms doesn't need to be 1. Below we specify the step to be 3 as the third argument, therefore 2 + 3 = 5, then 5 + 3 = 8. Since the next one 8 + 3 = 11, which is greater than the upper bound 9, so it stops at 8.

```
np.arange(2,10,3)
array([2, 5, 8])
```

Creating data array: Regular arrays creation

- A very similar method is `linspace()`. It takes the start (first argument), the end (second argument) and the desired length (last argument).

```
np.linspace(0,10, 51)
array([  0. ,   0.2,   0.4,   0.6,   0.8,   1. ,   1.2,
         1.4,   1.6,   1.8,   2. ,   2.2,   2.4,   2.6,
         2.8,   3. ,   3.2,   3.4,   3.6,   3.8,   4. ,
         4.2,   4.4,   4.6,   4.8,   5. ,   5.2,   5.4,
         5.6,   5.8,   6. ,   6.2,   6.4,   6.6,   6.8,
         7. ,   7.2,   7.4,   7.6,   7.8,   8. ,   8.2,
         8.4,   8.6,   8.8,   9. ,   9.2,   9.4,   9.6,
         9.8,  10. ])
```

- Without the last argument, the length defaults 50.

Creating data array: Regular arrays creation

- We might want to construct a constant array. The function ones() returns an array with as many 1 as the number plugged in.

```
np.ones(3)
array([1.0, 1.0, 1.0])
```

- We can construct all the constant array by multiply the desired constant to a array produced as above. The details of array operation will be discussed, but multiplying a number to an array results in every entry in the array multiplied by the constant.

```
3.1415 *np.ones(5)
array([ 3.1415,  3.1415,  3.1415,  3.1415,  3.1415])
```

Creating data array: Regular arrays creation

- We have seen from above that we can easily generate a zero array by times 0 to a array generated by the function `ones()`. However, Numpy also provides a function `zeros()`, which works in the same way as the function `ones()`:

```
np.zeros(5)
array([ 0.,  0.,  0.,  0.,  0.])
```

OVERVIEW

- NumPy
  - Ndarray
  - **Subscripting and slicing**
  - Operations
- Pandas
  - Data Structure
  - Data Manipulation
  - Grouping and aggregation

Index

- We often need to access a particular element (subscripting) in our array. We have already seen that ndarray is treated as an ordered sequence, whose entries are indexed by integers.

```
x = np.arange(1, 11)
```

- We may print out an arbitrary entry. For example, the 3rd element in x:

```
x[2]  # Python index from 0
3
```

- We may plug it into a function:

```
3**(x[2])
27
```

Index

- We may update an element with indexes:

```
x[2] = 100
x
array([  1,   2, 100,   4,   5,   6,   7,   8,   9,  10])
```

- Negative indexes select the element by the opposite order.

```
print x[-1]; print x[-2]; print x[-10]
10.0
9.0
1.0
```

Exercise 1

- Initialize x again with:

```
x = np.arange(1, 11)
```

- Run `x[2]=3.0`. Which entry of the array x will be updated?
- Run `x[2]=3.1`. Which entry of the array x will be updated?
- Run `x.astype(float)` to change the type to float, then update the third element to 3.1

Index

- We often need to subscript in a higher dimensional array. We start by constructing one:

```
high_x = np.array([[1,2,3],[4,5,6]])
```

- As we mentioned before, 2-dimensional ndarray can be thought as a nested list. Therefore, to access the number 2, it is the element 1 in the list 0:

```
high_x[0][1]
2
```

Index

- In this way, we may slice a particular row by selecting an inner list:

```
high_x[1]
array([4, 5, 6])
```

- A 2-dimensional array can be also taken for a matrix.

```
array([[1, 2, 3],
       [4, 5, 6]])
```

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Index

- Therefore as matrices, a 2-dimensional array can be indexed a pair of integers -- row indexes and columns indexes. Conventionally, the first index is for row, the second for column.

```
print high_x[0,0]; print high_x[0,1]; print high_x[0,2]
print high_x[1,0]; print high_x[1,1]; print high_x[1,2]
1
2
3
4
5
6
```

Index

- We may then slice part of the second row by specifying the row index and the range of the columns.

```
high_x[1,0:2]
array([4, 5])
```

- When specifying index to be from the first entry, we can leave it blank.

```
high_x[1,:2]
array([4, 5])
```

Shape

- Again as a matrix, the shape of an array can be denoted by the number of rows and columns it has. And again, first rows and then columns. For example, consider the following is a 2 by 3 matrix.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

- For an array, the shape (or dimension in more matrix convention) can be seen by calling the `.shape` attribute.

```
high_x.shape
(2, 3)
```

Shape

- Now we understand the notation of the shape. We may discuss the regular creation of 2-dimensional arrays. We start with generating a constant matrix by the functions `ones()`.

```
np.ones([2,3])
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

- The function `zeros()` works in the same way:

```
np.zeros([3,2])
array([[ 0.,  0.],
       [ 0.,  0.],
       [ 0.,  0.]])
```

Shape

- We may also rearrange the shape of a 1-dimensional array to obtain a higher dimensional one. This can be done by a method `reshape()`

```
x = np.arange(8)
x.reshape([2,4])
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

- This can also be achieved by assigning the desired value to the attribute `.shape`

```
x = np.arange(8)
x.shape = (2,4)
```

Exercise 2

- Create a 4 by 5 five matrix with all the entries 8.
- Create an array corresponding to the matrix below:

$$\begin{pmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \end{pmatrix}$$

OVERVIEW

- NumPy
  - Ndarray
  - Subscripting and slicing
  - **Operations**
- Pandas
  - Data Structure
  - Data Manipulation
  - Grouping and aggregation

Arithmetic operations

- We often want to do math on an array. Addition and scalar multiplication are widely used. They work as expected:

```
x = np.array([1,2])
y = np.array([3,4])
z = np.array([5,6])
x+y+z
array([ 9, 12])

5*x
array([5, 10])
```

Arithmetic operations

- Addition is actually an example of pointwise operations. Here is another example:

```
y*z
array([ 15, 24])
```

- All the arithmetic operators can work pointwisely.

```
x**y
array([1, 16])
y/x   # (Integer) Division
array([3, 2])
y-x
array([2, 2])
z%y
array([2, 2])
```

Arithmetic operations

- On the other hand, scalar product is an special case of broadcasting.

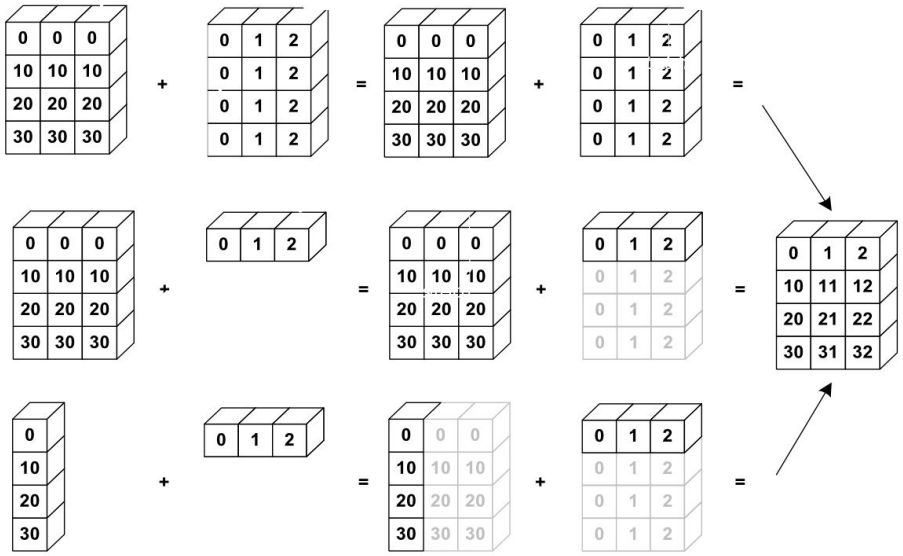
```
x+3         # Addition can be broadcasted
array([4, 5])
# Add 3 to each position of x
```

- All the arithmetic operators can be broadcasted similarly

```
x**2
array([1, 4])
x/2   # (Integer) Division
array([0, 1])
y-3
array([0, 1])
z%3
array([2, 0])
```

Broadcast operations

- More general broadcasting rule can be applied when the arrays have different shape. We leave the details to the audiences.



- The general broadcasting rules can be found via:
  - http://docs.scipy.org/doc/numpy/user/basics.broadcasting.html

Comparison operations

- Comparison operators generates Boolean arrays. As arithmetic operators, they also follow the broadcasting rule.

```
x>1
array([False,  True], dtype=bool)
```

- All the comparison operators can be broadcasted similarly.

```
x < 1
array([False, False], dtype=bool)
y<=3
array([ True, False], dtype=bool)
z>=6
array([False,  True], dtype=bool)
z==5
array([ True, False], dtype=bool)
```

Comparison operations

- Comparison operators can be applied pointwisely to multiple arrays.

```
x>y
array([False,  False], dtype=bool)
```

- All the operators can be broadcasted similarly.

```
x < y
array([True, True], dtype=bool)
y<=z
array([True, True], dtype=bool)
z>=x
array([True,  True], dtype=bool)
z==x
array([False, False], dtype=bool)
```

Fancy indexing

- We often need to filter an array according to some condition. This can be done by two steps.
  - Generate Boolean array

```
x==1
array([ True, False], dtype=bool)
```

  - Slice the array with the Boolean array

```
x[np.array([ True, False])]
array([1])
```

  - The two steps can be actually combined

```
x[x==1]
array([1])
```

Fancy indexing

- Fancy indexing can be applied to (higher) 2 dimensional arrays. However, slicing with fancy index drop the structure to the 1 dimensional array.

```
high_x = np.array([[1,2,3],[4,5,6]])
high_x[high_x>=3]
array([3, 4, 5, 6])
```

- However, assignment with fancy indexing doesn't change the shape.

```
high_x[high_x>=3]=10
high_x
array([[ 1,  2, 10],
       [10, 10, 10]])
```

Exercise 3

- Run the code below to create arrays.

```
ary_1= np.ones([3,2])
ary_2=np.arange(1,7).reshape(3,2)
```

  - Sum up the arrays.
  - Add 1 to the first column of ary_1, and add 2 to the second column of ary_1.
  - Update ary_2: change any number greater than 4 to 2.5.

OVERVIEW

- NumPy
  - Ndarray
  - Subscripting and slicing
  - Operations
- Pandas
  - **Data Structure**
  - Data Manipulation
  - Grouping and aggregation

Pandas

- Pandas is a large package defining several new data types, plus a variety of convenient functions for data manipulation, plotting, and web scraping.
- The *DataFrame* structure is inspired by the type of the same name in R, a programming language popular among statisticians and data scientists.
- Pandas is particularly strong in the area of handling missing data and, relatedly, handling time series data.
- There are four new data structures in pandas: `Series`, `DataFrame`, `time series` and `panel`. We will mainly discuss the first two.

Pandas data types

- These are the new data types introduced by pandas:
  - **Series**: 1D labeled homogeneously-typed array.
  - **DataFrame**: General 2D labeled, size-mutable tabular structure with potentially heterogeneously-typed columns.
  - **Time Series**: Series with index containing datetimes.
  - **Panel**: General 3D labeled, also size-mutable array.
- We first import the package:

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

Series

- A series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its *index*. By default, the index just consists of ordinary array indices, i.e. consecutive integers starting from zero.

```
obj = pd.Series(['a', 'b', 'c', 'd'])
obj
0   a
1   b
2   c
3   d
```

Series

- Often it will be more desirable to create a series with an index identifying each data point. Here we manually set the index from 1 to 4.

```
obj2 = pd.Series(['a', 'b', 'c', 'd'], index=[1, 2, 3, 4])
obj2
1   a
2   b
3   c
4   d
```

- The method `values` accesses all the values.

```
obj.values
array(['a', 'b', 'c', 'd'], dtype=object)
obj.values[1]
'b'
```

Series

- The Series object is similar to a dictionary, *Series.index* is like *dictionary.keys*, and *Series.values* is like *dictionary.values*. We can convert a dictionary to a Series directly:

```
dict_ = {1: 'a', 2: 'b', 3: 'c', 4: 'd'}
obj3 = pd.Series(dict_)
obj3
1   a
2   b
3   c
4   d
```

```
obj3.to_dict()        # convert Series to dict
{1: 'a', 2: 'b', 3: 'c', 4: 'd'}
```

DataFrame

- A data frame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns; each can be of a different value type (integers, strings, floating point numbers, Python objects, etc.), but all must be the same length.

```
# create a dictionary
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9]}
# convert to DataFrame
df = pd.DataFrame(data)
df
```

DataFrame

- A data frame can be created with a nested list as well.

```
df_2 = pd.DataFrame([[1.5, 'Ohio', 2000],
                     [1.7, 'Ohio', 2001],
                     [3.6, 'Ohio', 2002],
                     [2.4, 'Nevada', 2001],
                     [2.9, 'Nevada', 2002]],
                      columns=['pop','state','year'])
```

- The two ways are equivalent.

DataFrame

- A DataFrame has an attribute **values**, which is of the multidimensional array type.

```
df.values
array([[1.5, 'Ohio', 2000],
       [1.7, 'Ohio', 2001],
       [3.6, 'Ohio', 2002],
       [2.4, 'Nevada', 2001],
       [2.9, 'Nevada', 2002]], dtype=object)
```

- `df_2.values` gives the same result.

DataFrame

- DataFrame v.s. Series is similar to 2D array v.s. 1D array. A data frame has column names.

```
df.columns    # column name
# here u'pop' means the string 'pop' is encoded in unicode
Index([u'pop', u'state', u'year'], dtype='object')
```

DataFrame

- Each column in a data frame can be retrieved as a Series. We have two ways to get the column: to retrieve by attribute and to retrieve by dictionary-like notation. They will give the same result.

```
df.year         # retrieve by attribute
df['year']      # retrieve by dictionary-like notation

0    2000
1    2001
2    2002
3    2001
4    2002
Name: year, dtype: int64
```

Exercise 4

- Create a Pandas DataFrame, 'Employee', whose columns are 'Name', 'Year' and 'Department'. The rows are supposed to be:
  - Bob has been working for IT department for a year.
  - Sam has been working for Trade department for 3 years.
  - Peter has been working for HR department for 8 years.
  - Jake has been working for IT department for 2 years.
- Now set the index of Employee to be their names using `set_index` function of a data frame. Make sure you update the DataFrame.

DataFrame

- Pandas has a number of functions for reading tabular data as a DataFrame object.

```
pd.read_csv('foo.csv')    # use comma as the default delimiter
```

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

- Note that the function consider the first row as a header giving the column names, and both add incremental numbers as indices.

DataFrame

- Parsing can't be done properly with a bad delimiter.

```
# read_csv reads a \t separated file
pd.read_csv('foo.txt')
```

	a b c d message
0	1\t2\t3\t4\thello
1	5\t6\t7\t8\tworld
2	9\t10\t11\t12\tfoo

- We see the DataFrame becomes messy with a bad delimiter.

DataFrame

- The problem will be fixed by passing `sep = '\t'` to read_csv.

```
# read_csv reads a \t separated file
pd.read_csv('foo.txt', sep='\t')
```

.

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

DataFrame

- In some cases, there is no header in the file. With argument `header = None`, the column names will be filled with incremental numbers.

```
pd.read_csv('foo_noheader.csv', header = None)
```

.

	0	1	2	3	4
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

DataFrame

- But we can manually set the names of the columns by passing the list of column names.

```
# Set the names manually
pd.read_csv('foo_noheader.csv',
            names=['a', 'b', 'c', 'd', 'message'])
```

.

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

Exercise 5

So far we covered only importing a file. With this exercise we first demonstrate how exporting is done.

- Write the data frame, `Employee`, to a file, Employee.csv. Use function `to_csv`.

OVERVIEW

- NumPy
  - Ndarray
  - Subscripting and slicing
  - Operations
- Pandas
  - Data Structure
  - **Data Manipulation**
  - Grouping and aggregation

Data manipulation in pandas: concat

- Pandas DataFrames can be expanded in both directions. Let's create two data frames first.

```
df1 = pd.DataFrame(np.arange(9).reshape((3, 3)),
                   columns=['a', 'b', 'c'],
                   index=['one', 'two', 'three'])
df2 = pd.DataFrame(np.arange(6).reshape((3, 2)),
                   columns=['d','e'],
                   index=['one', 'two', 'three'])
```

	a	b	c
one	0	1	2
two	3	4	5
three	6	7	8

	d	e
one	0	1
two	2	3
three	4	5

Data manipulation in pandas: concat

- Since the two data frames have exactly the same rows, it is natural that we can combine them "horizontally".

```
pd.concat([df1, df2], axis = 1)
```

	a	b	c	d	e
one	0	1	2	0	1
two	3	4	5	2	3
three	6	7	8	4	5

- The argument "axis = 1" means expanding along the column indices. Setting "axis = 0" will combine two data frames with same number of columns vertically.

Exercise 6

- In the iPython notebook, we created the data frame below. How should we combine it with the old `Employee`? Observe that this is a data frame with new features.

	Education	Sex	Title
Bob	Bachelor	M	analyst
Sam	PHD	M	associate
Peter	Master	M	VP
Jake	Master	M	analyst

Data manipulation in pandas: merge

- Merging is the most common way to combine multiple data frames. Let's create two data frames first.

```
df1 = pd.DataFrame(np.array([0,0,0,2,2,2,8,8,8]).\
                    reshape((3, 3)),columns=['a','b','c'],
                    index=['one', 'two', 'three'])
df2 = pd.DataFrame(np.arange(6).reshape((3, 2)),
                    columns=['b', 'd'],
                    index=['one', 'two', 'four'])
```

	a	b	c
one	0	0	0
two	2	2	2
three	8	8	8

	b	d
one	0	1
two	2	3
four	4	5

Data manipulation in pandas: merge

- The code identifies the column 'b' from both data frames. The argument 'inner' means it only keeps rows occur in both data frames.

```
pd.merge(df1, df2, how='inner', on ='b')
```

	a	b	c	d
0	0	0	0	1
1	2	2	2	3

- The 'how' argument defaults to 'inner'. So the following code performs the same task as above.

```
pd.merge(df1, df2, on ='b')
```

Data manipulation in pandas: merge

- If we want to keep every row in df1, then we can specify how = "left".

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

	a	b	c	d
0	0	0	0	1
1	2	2	2	3
2	8	8	8	NaN

- Since df2 does not have a row with b=8, pandas leaves NaN for column d.

Data manipulation in pandas: merge

- If we want to keep all rows from both df1 and df2, then we can specify how = "outer".

```
pd.merge(df1, df2, how='outer', on ='b')
```

	a	b	c	d
0	0	0	0	1
1	2	2	2	3
2	8	8	8	NaN
3	NaN	4	NaN	5

- All the rows are kept this way.

Data manipulation in pandas: merge

- We can also merge on columns with different names.

```
pd.merge(df1, df2, right_on='b', left_on='a')
```

	a	b_x	c	b_y	d
0	0	0	0	0	1
1	2	2	2	2	3

- Since we have a row with a=0 in df1 and a row with b=0 in df2, they are identified. Similarly the row with a=2 in df1 and the row with b=2 in df2 are identified. Since the inner merge is default, and there is no row with a=b=4 nor a=b=8, so those two rows are discarded. Since this time the column b from the two data frames are not identified, there are still two after merging, namely, **b_x** and **b_y**.

Exercise 7

- Run the code provided in iPython notebook to create a data frame, 'Salary'. How is this data frame related to `Employee`?
- How should we combine the two data frames in a meaningful way?
  - **Caution:** Pay attention to the indices after merging.

	Title	Salary
0	VP	250
1	associate	120
2	analyst	90

Data manipulation in pandas: selection

- The *loc* method provides purely label (index/columns)-based indexing. This method only allows you do selection from a data frame by its index and columns. For example:

```
df1.loc['two'] # the row that has index two
```

```
a    3
b    4
c    5
Name: two, dtype: int64
```

Data manipulation in pandas: selection

- You can also pass a second parameter to *loc* to specify which column you want to choose. For example:

```
df1.loc['two', 'b'] # the row with index two and column b
4
```

Data manipulation in pandas: filter

- Fancy indexing as in Numpy can be done with *loc* in pandas as well. We may select a row with a condition:

```
df1.loc[df1.a==0,:]
```

	a	b	c
one	0	1	2

- We may select columns in a similar way:

```
df1.loc[:,df1.loc['one']==0]
```

	a
one	0
two	3
three	6

Data manipulation in pandas: selection

- Note: loc only accepts labels as input. If you try to use numbers, it will give you an error. For example:

```
df1.loc[1, 2]
KeyError: 'the label [1] is not in the [index]'
```

Data manipulation in pandas: selection

- If you want to select data by number, you need the help of *iloc*. The *iloc* method provides a purely position based indexing.

```
df1.iloc[1, 2]
# select as a matrix
# row 2, col 3
5
```

```
# first row, first two columns
# return a Series
row1 = df1.iloc[0, :2]
row1
```

```
a    0
b    1
Name: one, dtype: int64
```

OVERVIEW

- NumPy
  - Ndarray
  - Subscripting and slicing
  - Operations
- Pandas
  - Data Structure
  - Data Manipulation
  - **Grouping and aggregation**

Grouping and aggregation

- Aggregation is often a critical component of a data analysis workflow. It involves one or more of the following steps:
  - **Splitting** the data into groups based on some features.
  - **Applying** a function to each group independently.
  - **Combining** the result into a data structure.

Grouping and aggregation

- Let's create a data frame.

```
np.random.seed(10)
df = pd.DataFrame({'key1':['a','a','b','b','a'],
                   'key2':['one','two','one','two','one'],
                   'data1': np.random.randn(5),
                   'data2': np.random.randn(5)})
df
```

	data1	data2	key1	key2
0	1.331587	-0.720086	a	one
1	0.715279	0.265512	a	two
2	-1.545400	0.108549	b	one
3	-0.008384	0.004291	b	two
4	0.621336	-0.174600	a	one

Grouping and aggregation: groupby

- A natural question is: How many 'a's do we have in key1? One way to answer this is to group the data frame by the value in key1. That is:

```
group = df.groupby('key1')
```

- group is assigned the value returned by the groupby function, whose type is:

```
print type(group)
<class 'pandas.core.groupby.DataFrameGroupBy'>
```

Grouping and aggregation: groupby

- Here we introduce an important feature of the object. A *DataFrameGroupBy* object is an iterable. That says we can iterate over the object:

```
for item in group:
    print item
```

```
('a',       data1     data2 key1 key2
0  1.331587 -0.720086    a  one
1  0.715279  0.265512    a  two
4  0.621336 -0.174600    a  one)
('b',       data1     data2 key1 key2
2 -1.545400  0.108549    b  one
3 -0.008384  0.004291    b  two)
```

Grouping and aggregation: groupby

- With a careful inspection we see that each item we printed is a tuple with two components. In Python, there is an alternative way of iteration:

```
for key, values in group:
        print key
        print '-'*55
        print values
        print '\n'
```

Grouping and aggregation: groupby

- The result:

```
a
---------------------------------------------
      data1     data2 key1 key2
0  1.331587 -0.720086    a  one
1  0.715279  0.265512    a  two
4  0.621336 -0.174600    a  one

b
---------------------------------------------
      data1     data2 key1 key2
2 -1.545400  0.108549    b  one
3 -0.008384  0.004291    b  two
```

- In this way we can print and inspect a DataFrameGroupBy object. We also see how **splitting** is done.

Grouping and aggregation: groupby

- **Applying** and **combining** are often done together with a single function. For example:

```
group.size()
key1
a    3
b    2
dtype: int64
```

- The function `size` counts the number of observations in each group and then combines the result into a pandas series. This answers our question in the beginning of this section.

Grouping and aggregation: agg

- We may group the data by multiple keys:

```
group2 = df.groupby(['key1', 'key2'])
group2.mean()
```

		data1	data2
key1	key2		
a	one	0.976461	-0.447343
	two	0.715279	0.265512
b	one	-1.545400	0.108549
	two	-0.008384	0.004291

Grouping and aggregation: agg

- We may apply multiple functions to each group with the method agg:

```
group2.agg(['count', 'sum', 'min', 'max', 'mean', 'std'])
```

- Part of the result look like:

		data1					
		count	sum	min	max	mean	std
key1	key2						
a	one	2	1.952922	0.621336	1.331587	0.976461	0.502223
	two	1	0.715279	0.715279	0.715279	0.715279	NaN
b	one	1	-1.545400	-1.545400	-1.545400	-1.545400	NaN
	two	1	-0.008384	-0.008384	-0.008384	-0.008384	NaN

Grouping and aggregation: agg

- The other part of the result looks like:

		data2					
		count	sum	min	max	mean	std
key1	key2						
a	one	2	-0.894686	-0.720086	-0.174600	-0.447343	0.385716
	two	1	0.265512	0.265512	0.265512	0.265512	NaN
b	one	1	0.108549	0.108549	0.108549	0.108549	NaN
	two	1	0.004291	0.004291	0.004291	0.004291	NaN

- The column std misses three values because each of `['a', 'one']`, `['b', 'one']` and `['b', 'two']` includes only one row in `df`.

Grouping and aggregation: agg

- We may apply different aggregating functions to different columns. This can be done with a dictionary.

```
colFun = {'data1': ['min', 'max'],
          'data2': ['mean','std']}
group.agg(colFun)
```

	data1		data2	
	min	max	mean	std
key1				
a	0.621336	1.331587	-0.209725	0.493737
b	-1.545400	-0.008384	0.056420	0.073721