NUMPY

ARRAY AND VECTORIZED COMPUTATION

NOR HAMIZAH MISWAN

NUMPY

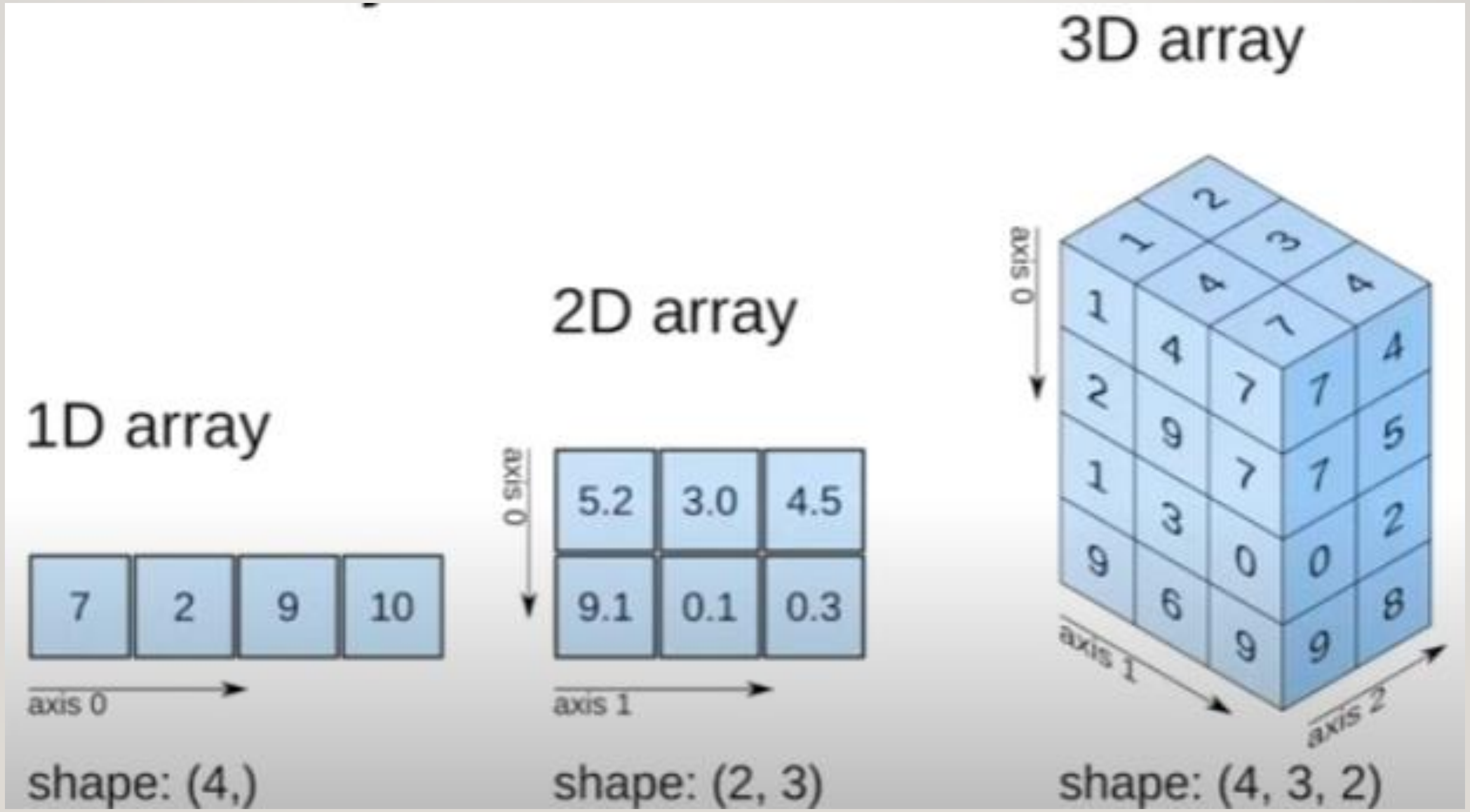
- NumPy, short for Numerical Python, has long been a cornerstone of numerical computing in Python.
- NumPy is a Python library used for working with arrays.
- NumPy contains, among other things:
  - A fast and efficient multidimensional array object ndarray
  - Functions for performing element-wise computations with arrays or mathematical operations between arrays

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- One of the key features of NumPy is its N-dimensional array object, or ndarray,
  - a fast, flexible container for large datasets in Python.
- Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.
- To import numpy: `import numpy as np`
- Random number generation: `np.random.randn(2, 3)` #indicate 2 rows & 3 columns
- Creating ndarray: use the array function. How?

  `np.array(list)`

ndarray: MULTIDIMENSIONAL ARRAY OBJECT



ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- Other function for creating new array,
  - zeros and ones create arrays of 0s or 1s.

  Try: np.zeros(10) and np.zeros((3, 6))
  - arange is an array-valued version of the built-in Python range function.

  Try: np.arange(15)
- The data type or dtype is a special object containing the information (or metadata, data about data).

Try: arr1.dtype and arr2.dtype

Note: refer Table 4-2 for NumPy data types

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- Any arithmetic operations between equal-size arrays applies the operation element-wise. Try:

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
arr * arr
arr – arr
1 / arr
```

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- For indexing and slicing of NumPy array,
  - One-dimensional arrays are simple; on the surface they act similarly to Python lists.

Try:

```
arr = np.arange(10)
arr[5:8] #index 5 to7
arr[5:8] = 12 # replace index 5 to 7 with 12
```

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- For indexing and slicing of NumPy array,
  - Two dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays.
  - Thus, individual elements can be accessed recursively, by a comma-separated list of indices to select individual elements.

Try:

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d[2]  #index two
arr2d[0][2] #index at zero, with element of index two
arr2d[0, 2] #equivalent to above
```

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- For indexing and slicing of NumPy array,
  - In multidimensional arrays, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions.

Try:

```
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
arr3d[0] #2x3 array
arr3d[1, 0] #index at one, whole element at index zero
arr3d[1, 0,1]
```

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- Ndarray can be sliced with familiar syntax [:].
  - Like one-dimensional objects such as Python lists, ndarrays can be sliced:

  Try: arr[1:6]
  - Slicing two-dimensional array is a bit different:

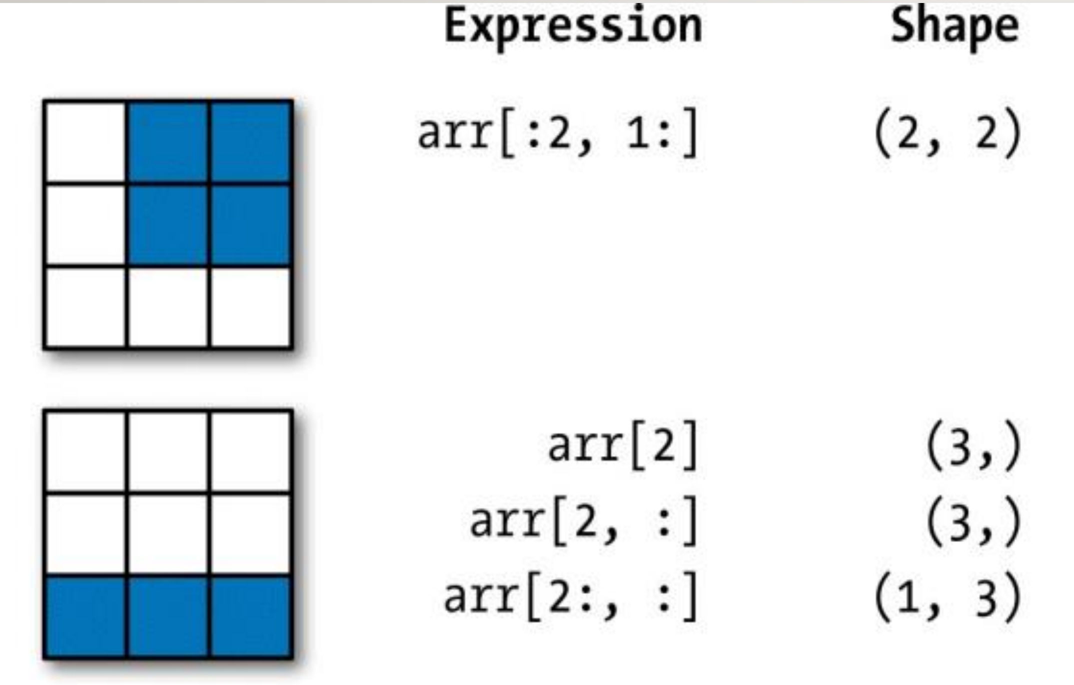
  Try: arr2d[:2]

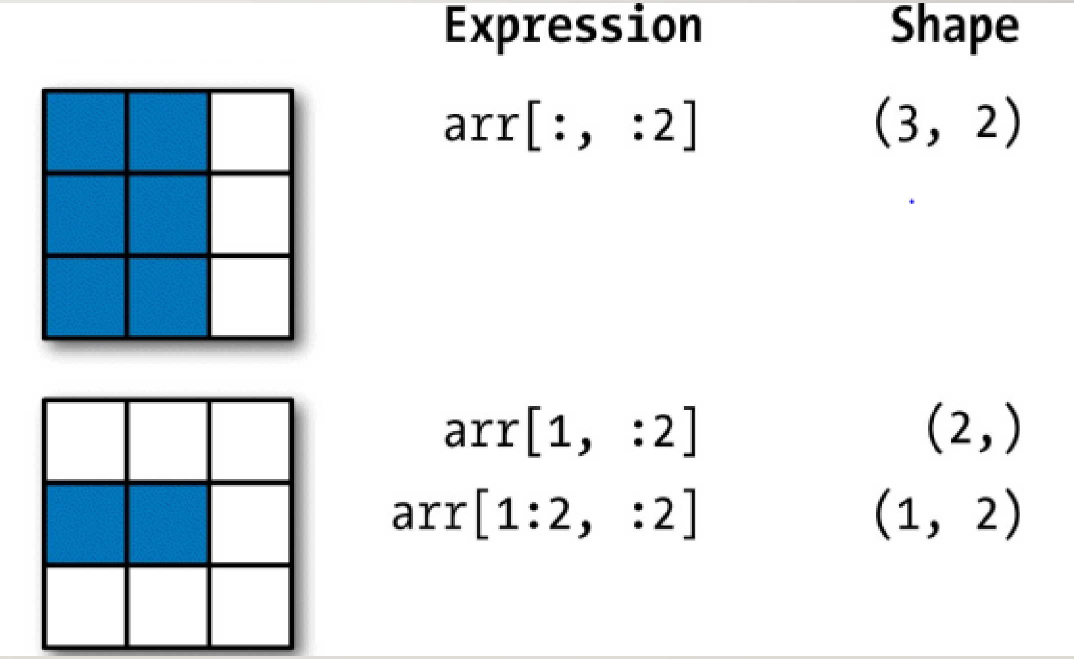
  arr2d[1, :2]
  - Multiple slices is just like you can pass multiple indexes:

  Try: arr2d[:2, 1:]

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- For two-dimensional array slicing:





ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- For Boolean indexing,
  - Let's have some data in an array and an array of names with duplicates:

  Try:

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
data = np.random.randn(7, 4)
```

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- For Boolean indexing,
  - Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob‘:

Try:

```
names == 'Bob'
data[names == 'Bob']
data[names == 'Bob', 2:]
data[names == 'Bob', 3]
names != 'Bob‘
data[~(names == 'Bob')]
```

Note: The boolean array must be of the same length as the array axis it’s indexing

```
mask = (names == 'Bob') | (names == 'Will')
data[mask]
```

Note: To combine multiple Boolean conditions, use arithmetic: & (for and) and | (for or)

ndarray: MULTIDIMENSIONAL ARRAY OBJECT

- For transposing arrays,
  - Transposing is a special form of reshaping that similarly returns a view on the underlying data without copying anything.
  - Arrays have the transpose method and also the special T attribute

Try:

arr = np.arange(15).reshape((3, 5))

arr

Arr.T

np.dot(arr.T, arr)

UNIVERSAL FUNCTION: FAST ELEMENT-WISE ARRAY FUNCTION

- A universal function, or ufunc, is a function that performs element-wise operations on data in ndarrays.
- Many ufuncs are simple element-wise transformations, like sqrt or exp.
  - ➢These are referred to as unary ufuncs.

  Try: arr = np.arange(10)
  np.sqrt(arr)
  np.exp(arr)
- Two arrays (binary ufuncs) return a single array as the result.
  - ➢Example: add or maximum

  Try: x = np.random.randn(8),
  y = np.random.randn(8)
  np.maximum(x, y)

Note: Other unary and binary ufuncs can be obtained in Table 4-3 and Table 4-4

ARRAY-ORIENTED PROGRAMMING

- Using NumPy arrays enables you to express many kinds of data processing tasks as concise array expressions that might otherwise require writing loops.
- In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact in any kind of numerical computations.

Try:

```
points = np.arange(-5, 5, 0.01) # 1000 equally spaced points

xs, ys = np.meshgrid(points, points)

z = np.sqrt(xs ** 2 + ys ** 2)
```

ARRAY-ORIENTED PROGRAMMING

- A set of mathematical functions that compute statistics about an entire array or about the data along an axis are accessible as methods of the array class.

Try:

```
arr = np.random.randn(5, 4)
arr.mean() #equivalent to np.mean(arr)
arr.sum()
```

- Functions like mean and sum take an optional axis argument that computes the statistic over the given axis, resulting in an array with one fewer dimension:

Try:

```
arr.mean(axis=1) #axis=1 compute mean across the columns
arr.sum(axis=0) #axis=0 compute sum down the rows
```

Note: Other basic array statistical methods can be obtained in Table 4-5.

ARRAY-ORIENTED PROGRAMMING

- NumPy arrays can be sorted in-place with the sort method.

Try:

```
arr = np.random.randn(6)
arr.sort()
arr1 = np.random.randn(5, 3)
arr.sort(1)
```

- np.unique returns the sorted unique values in an array.

Try:

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
np.unique(names)
```

Note: Other array set operations can be obtained in Table 4-6.

LINEAR ALGEBRA

- Linear algebra, like matrix multiplication, decompositions, determinants, and other square matrix math, is an important part of any array library. Try:

```
x = np.array([[1., 2., 3.], [4., 5., 6.]])
y = np.array([[6., 23.], [-1, 7], [8, 9]])
x.dot(y) #equivalent to np.dot(x, y)
```

- To consider a standard set of matrix decompositions and things like inverse and determinant, use numpy.linalg. Try:

```
from numpy.linalg import inv
X = np.random.randn(5, 5)
mat = X.T.dot(X)
inv(mat)
```

Note: Other commonly used numpy.linalg functions can be obtained in Table 4-6.

PSEUDORANDOM NUMBER GENERATION

- The numpy.random module supplements the built-in Python random with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions (such as normal, beta, chi-square, gamma, binomial, etc.).
- For example, you can get a 4 × 4 array of samples from the standard normal distribution using normal:

```
samples = np.random.normal(size=(4, 4))
```

Note: Other commonly used list of numpy.random functions can be obtained in Table 4-6.

PANDAS

DATA STRUCTURES AND DATA MANIPULATION TOOLS

NOR HAMIZAH MISWAN

GETTING STARTED

- Pandas contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python.
- Pandas is often used in tandem with numerical computing tools like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and data visualization libraries like matplotlib.
- The biggest difference of NumPy and Pandas:
  - Pandas is designed for working with tabular or heterogeneous data.
  - NumPy is best suited for working with homogeneous numerical array data.

GETTING STARTED

- To import pandas: import pandas as pd
- It easier to import Series and DataFrame into the local namespace since they are so frequently used: from pandas import Series, DataFrame
- Series:
  - A one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index
- DataFrame:
  - represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.)..

SERIES

- A one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index.
- The simplest Series is formed from only an array of data: `obj = pd.Series([4, 7, -5, 3])`.
- We can create a Series with an index identifying each data point with a label:

```
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
obj2['a']
```

- To replace certain number: `obj2['d'] = 6`

```
obj2[['c', 'a', 'd']]
```

SERIES

- Can also used with NumPy functions or NumPy-like operations, such as scalar multiplication, or applying math functions:

```
obj2[obj2 > 0]
obj2 * 2
np.exp(obj2)
```

- If you have data contained in a Python dictionary, you can create a Series from it by passing the dictionary:

```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj3 = pd.Series(sdata)
states = ['California', 'Ohio', 'Oregon', 'Texas']
obj4 = pd.Series(sdata, index=states)
```

SERIES

- To detect missing data, use isnull or notnull functions:

```
pd.isnull(obj4)  #silimar to obj4.isnull()
pd.notnull(obj4)
```

- A Series's index can be altered in-place by assignment:

```
obj = pd.Series([4, 7, -5, 3])
obj
obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
```

DATAFRAME

- DataFrame,
  - represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.).
  - has both a row and column index; it can be thought of as a dictionary of Series all sharing the same index. ate arrays of 0s or 1s.
- To construct DataFrame from a dict of equal-length lists or NumPy arrays:

```
data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
        'year': [2000, 2001, 2002, 2001, 2002, 2003],
        'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
frame = pd.DataFrame(data)
frame.head() #specify the header (top five)
```

DATAFRAME

- To specify a sequence of columns: pd.DataFrame(data, columns=['year', 'state', 'pop'])
- If you pass a column that isn't contained in the dict, it will appear with missing values in the result:

```
frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                      index=['one', 'two', 'three', 'four', 'five', 'six'])
```

- A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute: frame2['state']
  frame2.year
- To retrieved specific row by position or name using loc function: frame2.loc['three']
- To insert/change the values in a column: frame2['debt'] = 16.5
  frame2['eastern'] = frame2.state == 'Ohio'

DATAFRAME

- The del method can then be used to remove this column: del frame2['eastern']
- To transpose the DataFrame:

```
pop = {'Nevada': {2001: 2.4, 2002: 2.9}, 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
frame3 = pd.DataFrame(pop)
frame3.T
```

INDEXING AND REINDEXING

- An important method on pandas objects is reindex, which means to create a new object with the data conformed to a new index:

```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
obj3.reindex(range(6), method='ffill') #ffill refered to forward-fills the values.
```

INDEXING AND REINDEXING

- With DataFrame, reindex can alter either the (row) index, columns, or both. When passed only a sequence, it reindexes the rows in the result:

```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)), index=['a', 'c', 'd'],
        columns=['Ohio', 'Texas', 'California']
frame2 = frame.reindex(['a', 'b', 'c', 'd'])
states = ['Texas', 'Utah', 'California']
frame.reindex(columns=states) #The columns can be reindexed with the columns keyword
```

- You can reindex by label indexing with loc function:

```
frame.loc[['a', 'b', 'c', 'd'], states]
```

DROP ENTRIES

- Dropping one or more entries from an axis is easy if you already have an index array or list without those entries.

```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
new_obj = obj.drop('c')
obj.drop(['d', 'c'])
data = pd.DataFrame(np.arange(16).reshape((4, 4)), index=['Ohio', 'Colorado', 'Utah', 'New York'], columns=['one', 'two', 'three', 'four'])
data.drop(['Colorado', 'Ohio'])
data.drop('two', axis=1) #drop values from the columns by passing axis=1 or axis='columns'
```

INDEXING, SLICING, FILTERING

- Slicing with labels behaves differently than normal Python slicing in that the endpoint is inclusive.

```
obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
obj['b'] #same as obj[1]
obj[2:4]
obj[['b', 'a', 'd']]
obj[[1, 3]]
obj['b':'c']
obj['b':'c'] = 5 #modifies the corresponding value
```

INDEXING, SLICING, FILTERING

- Indexing into a DataFrame is for retrieving one or more columns either with a single value or sequence:

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)), index=['Ohio', 'Colorado', 'Utah', 'New York'], columns=['one', 'two', 'three', 'four'])
data['two']
data[['three', 'one']]
data[:2] #this will select based on row
```

INDEXING, SLICING, FILTERING

- Selection with loc and iloc,
  - enable you to select a subset of the rows and columns from a DataFrame with NumPy-like notation using either axis labels (loc) or integers (iloc).

```
data = pd.DataFrame(np.arange(16).reshape((4, 4)), index=['Ohio', 'Colorado', 'Utah', 'New York'], columns=['one', 'two', 'three', 'four'])
data.loc['Colorado', ['two', 'three']]
data.iloc[2, [3, 0, 1]]
data.iloc[2]
data.iloc[[1, 2], [3, 0, 1]]
data.loc[:'Utah', 'two']
```

ARITHMETIC METHOD

- An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes.
- When you are adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs.

```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1], index=['a', 'c', 'e', 'f', 'g'])
s1 + s2
```

ARITHMETIC METHOD

- In the case of DataFrame, alignment is performed on both the rows and the columns:

```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'), index=['Ohio',
        'Texas', 'Colorado'])
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'), index=['Utah',
        'Ohio', 'Texas', 'Oregon'])
df1 + df2
```

- Since the 'c' and 'e' columns are not found in both DataFrame objects, they appear as all missing in the result.
- Also, try:

```
1 / df1 #Table 5-5 shows the flexible arithmetic methods
```

SORTING

- Sorting a dataset by some criterion is another important built-in operation.
- To sort lexicographically by row or column index, use the sort_index method, which returns a new, sorted object:

```
obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
obj.sort_index()
```

- With a DataFrame, you can sort by index on either axis:

```
frame = pd.DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'], columns=['d', 'a',
        'b', 'c'])
frame.sort_index() #by default, it will sort in row
frame.sort_index(axis=1) #axis=1will sort by column
frame.sort_index(axis=1, ascending=False) #n be sorted in descending order
```

SORTING

- To sort a Series by its values, use its sort_values method:

```
obj = pd.Series([4, 7, -3, 2])
obj.sort_values()
```

- Any missing values are sorted to the end of the Series by default:

```
obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
obj.sort_values()
```

- When sorting a DataFrame, you can use the data in one or more columns as the sort keys. To do so, pass one or more column names to the by option of sort_values:

```
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
frame.sort_values(by='b')
frame.sort_values(by=['a', 'b']) #rank a first, then b
```

DESCRIPTIVE STATISTICS

- Pandas objects are equipped with a set of common mathematical and statistical methods.
- Compared with the similar methods found on NumPy arrays, they have built-in handling for missing data.

```
df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5], [np.nan, np.nan], [0.75, -1.3]], index=['a', 'b', 'c',
        'd'], columns=['one', 'two'])
df.sum()
df.sum(axis='columns') #or axis=1 add the column for each row
df.mean(axis='columns', skipna=False) #skip adding with NA using skipna function
```

Note: refer Table 5-8 for list of descriptive statistics