Informatics Practices (New Syllabus)
Unit 2: Data Handling (DH-1)

Introduction to data structures in Pandas

Pandas is an open-source, BSD-licensed Python library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.

Pandas deals with the following three data structures −

- Series
- DataFrame
- Panel

These data structures are built on top of Numpy array, which means they are fast.

Dimension & Description

The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure. For example, DataFrame is a container of Series, Panel is a container of DataFrame.

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, sizeimmutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure with potentially heterogeneously typed columns.
Panel	3	General 3D labeled, size-mutable array.

Building and handling two or more dimensional arrays is a tedious task, burden is placed on the user to consider the orientation of the data set when writing functions. But using Pandas data structures, the mental effort of the user is reduced.

For example, with tabular data (DataFrame) it is more semantically helpful to think of the **index** (the rows) and the **columns** rather than axis 0 and axis 1.

Mutability

All Pandas data structures are value mutable (can be changed) and except Series all are size mutable. Series is size immutable.

Note − DataFrame is widely used and one of the most important data structures. Panel is used much less.

Series

Series is a one-dimensional array like structure with homogeneous data. For example, the following series is a collection of integers 10, 23, 56, …

10	23	56	17	52	61	73	90	26	72

Key Points

- Homogeneous data
- Size Immutable
- Values of Data Mutable

DataFrame

DataFrame is a two-dimensional array with heterogeneous data. For example,

Name	Age	Gender	Rating
Steve	32	Male	3.45
Lia	28	Female	4.6
Vin	45	Male	3.9
Katie	38	Female	2.78

The table represents the data of a sales team of an organization with their overall performance rating. The data is represented in rows and columns. Each column represents an attribute and each row represents a person.

Data Type of Columns

The data types of the four columns are as follows −

Column	**Type**
Name	String
Age	Integer
Gender	String
Rating	Float

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

Panel

Panel is a three-dimensional data structure with heterogeneous data. It is hard to represent the panel in graphical representation. But a panel can be illustrated as a container of DataFrame.

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

Informatics Practices (New Syllabus)
Unit 2: Data Handling (DH-1)

Operations on a Series

Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index.

pandas.Series

A pandas Series can be created using the following constructor −

```
pandas.Series( data, index, dtype, copy)
```

The parameters of the constructor are as follows −

S.No	Parameter & Description
1	**data -** data takes various forms like ndarray, list, constants
2	**index -** Index values must be unique and hashable, same length as data. Default **np.arrange(n)** if no index is passed.
3	**dtype -** dtype is for data type. If None, data type will be inferred
4	**copy -** Copy data. Default False

A series can be created using various inputs like −

- Array
- Dict
- Scalar value or constant

Create an Empty Series

A basic series, which can be created is an Empty Series.

Example

```
#import the pandas library and aliasing as pd
import pandas as pd
s = pd.Series()
print s
```

Its **output** is as follows −

```
Series([], dtype: float64)
```

Create a Series from ndarray

If data is an ndarray, then index passed must be of the same length. If no index is passed, then by default index will be **range(n)** where **n** is array length, i.e., [0,1,2,3…. **range(len(array))-1].**

Example 1

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print s
```

Its **output** is as follows −

```
0   a
1   b
2   c
3   d
dtype: object
```

We did not pass any index, so by default, it assigned the indexes ranging from 0 to **len(data)-1**, i.e., 0 to 3.

Example 2

```
#import the pandas library and aliasing as pd
```

```
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data,index=[100,101,102,103])
print s
```

Its **output** is as follows −

```
100 a
101 b
102 c
103 d
dtype: object
```

We passed the index values here. Now we can see the customized indexed values in the output.

Create a Series from dict

A **dict** can be passed as input and if no index is specified, then the dictionary keys are taken in a sorted order to construct index. If **index** is passed, the values in data corresponding to the labels in the index will be pulled out.

Example 1

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data)
print s
```

Its **output** is as follows −

```
a 0.0
b 1.0
c 2.0
```

```
dtype: float64
```

Observe − Dictionary keys are used to construct index.

Example 2

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data,index=['b','c','d','a'])
print s
```

Its **output** is as follows −

```
b 1.0
c 2.0
d NaN
a 0.0
dtype: float64
```

Observe − Index order is persisted and the missing element is filled with NaN (Not a Number).

Create a Series from Scalar

If data is a scalar value, an index must be provided. The value will be repeated to match the length of **index**

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
s = pd.Series(5, index=[0, 1, 2, 3])
print s
```

Its **output** is as follows −

```
0 5
```

```
1 5
2 5
3 5
dtype: int64
```

pandas.Series.head

`Series.head`(*n=5*)

Return the first *n* rows.

This function returns the first *n* rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

Parameters:	**n** : int, default 5 Number of rows to select.
Returns:	**obj_head** : type of caller The first *n* rows of the caller object.

Returns the last *n* rows.

Examples

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
 animal
0 alligator
1  bee
2 falcon
3  lion
4 monkey
5 parrot
6 shark
7 whale
8 zebra
```

Viewing the first 5 lines

```
>>> df.head()
 animal
0 alligator
1  bee
2 falcon
3  lion
4 monkey
```

Viewing the first *n* lines (three in this case)

```
>>> df.head(3)
 animal
0 alligator
1  bee
2 falcon
```

pandas.Series.tail

`Series.tail`(*n=5*)

Return the last *n* rows.

This function returns last *n* rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

Parameters:	**n** : int, default 5 Number of rows to select.
Returns:	**type of caller** The last *n* rows of the caller object.

The first *n* rows of the caller object.

Examples

```
>>> df = pd.DataFrame({'animal':['alligator', 'bee', 'falcon', 'lion',
```

```
'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
 animal
0 alligator
1  bee
2 falcon
3  lion
4 monkey
5 parrot
6 shark
7 whale
8 zebra
```

Viewing the last 5 lines

```
>>> df.tail()
  animal
4 monkey
5 parrot
6  shark
7  whale
8  zebra
```

Viewing the last *n* lines (three in this case)

```
>>> df.tail(3)
 animal
6 shark
7 whale
8 zebra
```

Here we discuss a lot of the essential functionality common to the pandas data structures. Here's how to create some of the objects used in the examples from the previous section:

```
In [1]: index = pd.date_range('1/1/2000', periods=8)
```

```
In [2]: s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])

In [3]: df = pd.DataFrame(np.random.randn(8, 3), index=index,
   ...:    columns=['A', 'B', 'C'])
   ...:

In [4]: wp = pd.Panel(np.random.randn(2, 5, 4), items=['Item1', 'Item2'],
   ...:    major_axis=pd.date_range('1/1/2000', periods=5),
   ...:    minor_axis=['A', 'B', 'C', 'D'])
   ...:
```

Head and Tail

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

```
In [5]: long_series = pd.Series(np.random.randn(1000))

In [6]: long_series.head()
Out[6]:
0   0.229453
1   0.304418
2   0.736135
3   -0.859631
4   -0.424100
dtype: float64

In [7]: long_series.tail(3)
Out[7]:
997   -0.351587
998   1.136249
999   -0.448789
dtype: float64
```

Attributes and the raw ndarray(s)

pandas objects have a number of attributes enabling you to access the metadata

- **shape**: gives the axis dimensions of the object, consistent with ndarray
- Axis labels
  - **Series**: *index* (only axis)
  - **DataFrame**: *index* (rows) and *columns*
  - **Panel**: *items*, *major_axis*, and *minor_axis*

Note, **these attributes can be safely assigned to**!

```
In [8]: df[:2]
Out[8]:
    A   B  C
2000-01-01 0.048869 -1.360687 -0.47901
2000-01-02 -0.859661 -0.231595 -0.52775

In [9]: df.columns = [x.lower() for x in df.columns]

In [10]: df
Out[10]:
    a   b   c
2000-01-01 0.048869 -1.360687 -0.479010
2000-01-02 -0.859661 -0.231595 -0.527750
2000-01-03 -1.296337 0.150680 0.123836
2000-01-04 0.571764 1.555563 -0.823761
2000-01-05 0.535420 -1.032853 1.469725
2000-01-06 1.304124 1.449735 0.203109
2000-01-07 -1.032011 0.969818 -0.962723
2000-01-08 1.382083 -0.938794 0.669142
```

Unit 2: Data Handling (DH-1)

Introduction to data structures in Pandas

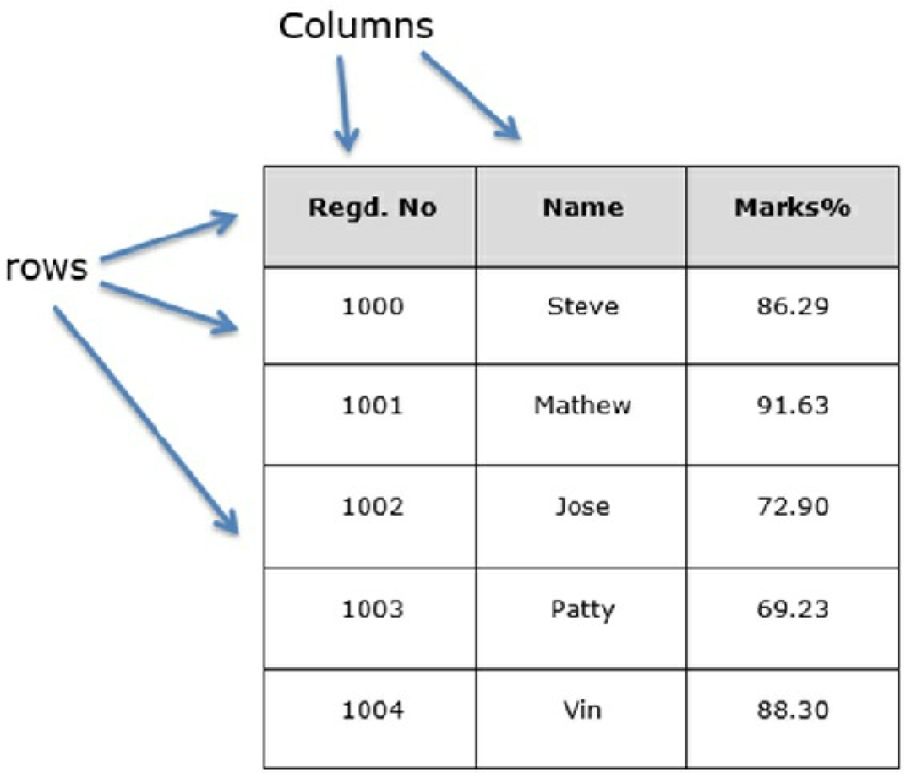
A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

Features of Data Frame

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns

Structure

Let us assume that we are creating a data frame with student's data.



Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

You can think of it as an SQL table or a spreadsheet data representation.

pandas.DataFrame

A pandas DataFrame can be created using the following constructor −

```
pandas.DataFrame( data, index, columns, dtype, copy)
```

The parameters of the constructor are as follows −

S.No	Parameter & Description
1	**data -** data takes various forms like ndarray, series, map, lists, dict, constants and also another DataFrame.
2	**index -** For the row labels, the Index to be used for the resulting frame is Optional Default np.arrange(n) if no index is passed.
3	**columns -** For column labels, the optional default syntax is - np.arrange(n). This is only true if no index is passed.
4	**dtype -** Data type of each column.
4	**copy -** This command (or whatever it is) is used for copying of data, if the default is False.

Create DataFrame

A pandas DataFrame can be created using various inputs like −

- Lists
- dict
- Series
- Numpy ndarrays
- Another DataFrame

In the subsequent sections of this chapter, we will see how to create a DataFrame using these inputs.

Create an Empty DataFrame

A basic DataFrame, which can be created is an Empty Dataframe.

Example

```
#import the pandas library and aliasing as pd
import pandas as pd
df = pd.DataFrame()
print df
```

Its **output** is as follows −

Empty DataFrame

Columns: []

Index: []

Create a DataFrame from Lists

The DataFrame can be created using a single list or a list of lists.

Example 1

import pandas as pd

data = [1,2,3,4,5]

df = pd.DataFrame(data)

print df

Its **output** is as follows −

```
  0
0  1
1  2
2  3
3  4
4  5
```

Example 2

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
print df
```

Its **output** is as follows −

```
 Name Age
0 Alex 10
1 Bob 12
2 Clarke 13
```

Example 3

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'],dtype=float)
print df
```

Its **output** is as follows −

```
 Name Age
0 Alex 10.0
1 Bob 12.0
2 Clarke 13.0
```

Note − Observe, the **dtype** parameter changes the type of Age column to floating point.

Create a DataFrame from Dict of ndarrays / Lists

All the **ndarrays** must be of same length. If index is passed, then the length of the index should equal to the length of the arrays.

If no index is passed, then by default, index will be range(n), where **n** is the array length.

Example 1

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data)
print df
```

Its **output** is as follows −

```
 Age Name
0 28 Tom
1 34 Jack
2 29 Steve
3 42 Ricky
```

Note − Observe the values 0,1,2,3. They are the default index assigned to each using the function range(n).

Example 2

Let us now create an indexed DataFrame using arrays.

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print df
```

Its **output** is as follows −

```
 Age Name
rank1 28 Tom
rank2 34 Jack
rank3 29 Steve
rank4 42 Ricky
```

Note − Observe, the **index** parameter assigns an index to each row.

Create a DataFrame from List of Dicts

List of Dictionaries can be passed as input data to create a DataFrame. The dictionary keys are by default taken as column names.

Example 1

The following example shows how to create a DataFrame by passing a list of dictionaries.

```
import pandas as pd
```

```
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
print df
```

Its **output** is as follows −

```
 a b c
0 1 2 NaN
1 5 10 20.0
```

Note − Observe, NaN (Not a Number) is appended in missing areas.

Example 2

The following example shows how to create a DataFrame by passing a list of dictionaries and the row indices.

```
import pandas as pd
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['first', 'second'])
print df
```

Its **output** is as follows −

```
 a b c
first 1 2 NaN
second 5 10 20.0
```

Example 3

The following example shows how to create a DataFrame with a list of dictionaries, row indices, and column indices.

```
import pandas as pd
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]

#With two column indices, values same as dictionary keys
df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])
```

```
#With two column indices with one index with other name
df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])
print df1
print df2
```

Its **output** is as follows −

```
#df1 output
 a b
first 1 2
second 5 10

#df2 output
 a b1
first 1 NaN
second 5 NaN
```

Note − Observe, df2 DataFrame is created with a column index other than the dictionary key; thus, appended the NaN's in place. Whereas, df1 is created with column indices same as dictionary keys, so NaN's appended.

Create a DataFrame from Dict of Series

Dictionary of Series can be passed to form a DataFrame. The resultant index is the union of all the series indexes passed.

Example

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
 'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print df
```

Its **output** is as follows −

```
 one two
a 1.0 1
b 2.0 2
c 3.0 3
d NaN 4
```

Note − Observe, for the series one, there is no label **'d'** passed, but in the result, for the **d** label, NaN is appended with NaN.

Let us now understand **column selection, addition**, and **deletion** through examples.

Column Selection

We will understand this by selecting a column from the DataFrame.

Example

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
 'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print df ['one']
```

Its **output** is as follows −

```
a 1.0
b 2.0
c 3.0
d NaN
Name: one, dtype: float64
```

Column Addition

We will understand this by adding a new column to an existing data frame.

Example

```
import pandas as pd
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
 'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
# Adding a new column to an existing DataFrame object with column label by p
print ("Adding a new column by passing as Series:")
df['three']=pd.Series([10,20,30],index=['a','b','c'])
print df
print ("Adding a new column using the existing columns in DataFrame:")
df['four']=df['one']+df['three']
print df
```

Its **output** is as follows −

```
Adding a new column by passing as Series:
 one two three
a 1.0 1 10.0
b 2.0 2 20.0
c 3.0 3 30.0
d NaN 4 NaN

Adding a new column using the existing columns in DataFrame:
 one two three four
a 1.0 1 10.0 11.0
b 2.0 2 20.0 22.0
c 3.0 3 30.0 33.0
d NaN 4 NaN NaN
```

Column Deletion

Columns can be deleted or popped; let us take an example to understand how.

Example

```
# Using the previous DataFrame, we will delete a column
# using del function
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
 'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
 'three' : pd.Series([10,20,30], index=['a','b','c'])}

df = pd.DataFrame(d)
print ("Our dataframe is:")
print df

# using del function
print ("Deleting the first column using DEL function:")
del df['one']
print df

# using pop function
print ("Deleting another column using POP function:")
df.pop('two')
print df
```

Its **output** is as follows −

```
Our dataframe is:
 one three two
a 1.0 10.0 1
b 2.0 20.0 2
c 3.0 30.0 3
d NaN NaN 4

Deleting the first column using DEL function:
 three two
a 10.0 1
```

```
b 20.0 2
c 30.0 3
d NaN 4

Deleting another column using POP function:
 three
a 10.0
b 20.0
c 30.0
d NaN
```

Row Selection, Addition, and Deletion

We will now understand row selection, addition and deletion through examples. Let us begin with the concept of selection.

Selection by Label

Rows can be selected by passing row label to a **loc** function.

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
 'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print df.loc['b']
```

Its **output** is as follows −

```
one 2.0
two 2.0
Name: b, dtype: float64
```

The result is a series with labels as column names of the DataFrame. And, the Name of the series is the label with which it is retrieved.

Selection by integer location

Rows can be selected by passing integer location to an **iloc** function.

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
 'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print df.iloc[2]
```

Its **output** is as follows −

```
one 3.0
two 3.0
Name: c, dtype: float64
```

Slice Rows

Multiple rows can be selected using ' : ' operator.

```
import pandas as pd

d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
 'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}

df = pd.DataFrame(d)
print df[2:4]
```

Its **output** is as follows −

```
 one two
c 3.0 3
d NaN 4
```

Addition of Rows

Add new rows to a DataFrame using the **append** function. This function will append the rows at the end.

```
import pandas as pd

df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)
print df
```

Its **output** is as follows −

```
  a b
0 1 2
1 3 4
0 5 6
1 7 8
```

Deletion of Rows

Use index label to delete or drop rows from a DataFrame. If label is duplicated, then multiple rows will be dropped.

If you observe, in the above example, the labels are duplicate. Let us drop a label and will see how many rows will get dropped.

```
import pandas as pd

df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)

# Drop rows with label 0
df = df.drop(0)
```

```
print df
```

Its **output** is as follows −

```
  a  b
1  3  4
1  7  8
```

In the above example, two rows were dropped because those two contain the same label 0.

Unit 2: Data Handling (DH-1)

Binary operations in a Data Frame

pandas.DataFrame.add

`DataFrame.add`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *add*).

Equivalent to `dataframe + other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters:	**other** : Series, DataFrame, or constant **axis** : {0, 1, ‘index’, ‘columns’} For Series input, axis to match Series index on **level** : int or name Broadcast across a level, matching Index values on the passed MultiIndex level **fill_value** : None or float value, default None Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing
Returns:	**result** : DataFrame

Notes : Mismatched indices will be unioned together

Examples

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
```

```
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

pandas.DataFrame.sub

`DataFrame.sub`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Subtraction of dataframe and other, element-wise (binary operator *sub*).

Equivalent to `dataframe - other`, but with support to substitute a fill_value for missing data in one of the inputs.

	other : Series, DataFrame, or constant **axis** : {0, 1, ‘index’, ‘columns’} For Series input, axis to match Series index on **level** : int or name

Parameters:	Broadcast across a level, matching Index values on the passed MultiIndex level **fill_value** : None or float value, default None Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing
Returns:	**result** : DataFrame

Notes : Mismatched indices will be unioned together

Examples

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.sub(b, fill_value=0)
   one  two
a  1.0  -3.0
b  1.0  -2.0
```

```
c   1.0   NaN
d   -1.0   NaN
e   NaN   -2.0
```

pandas.DataFrame.mul

`DataFrame.mul`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Multiplication of dataframe and other, element-wise (binary operator *mul*).

Equivalent to `dataframe * other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters:	**other** : Series, DataFrame, or constant **axis** : {0, 1, 'index', 'columns'} For Series input, axis to match Series index on **level** : int or name Broadcast across a level, matching Index values on the passed MultiIndex level **fill_value** : None or float value, default None Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing
Returns:	**result** : DataFrame

Notes : Mismatched indices will be unioned together

pandas.DataFrame.div

`DataFrame.div`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Floating division of dataframe and other, element-wise (binary operator *truediv*).

Equivalent to `dataframe / other`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters:	**other** : Series, DataFrame, or constant **axis** : {0, 1, ‘index’, ‘columns’} For Series input, axis to match Series index on **level** : int or name Broadcast across a level, matching Index values on the passed MultiIndex level **fill_value** : None or float value, default None Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing
Returns:	**result** : DataFrame

Notes : Mismatched indices will be unioned together

pandas.DataFrame.radd

`DataFrame.radd`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Addition of dataframe and other, element-wise (binary operator *radd*).

Equivalent to `other + dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters:	**other** : Series, DataFrame, or constant **axis** : {0, 1, ‘index’, ‘columns’} For Series input, axis to match Series index on **level** : int or name Broadcast across a level, matching Index values on the passed MultiIndex level **fill_value** : None or float value, default None Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing

Returns:	result : DataFrame

Notes : Mismatched indices will be unioned together

Examples

```
>>> a = pd.DataFrame([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  1.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[np.nan, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  NaN
b  NaN  2.0
d  1.0  NaN
e  NaN  2.0
>>> a.add(b, fill_value=0)
   one  two
a  2.0  NaN
b  1.0  2.0
c  1.0  NaN
d  1.0  NaN
e  NaN  2.0
```

pandas.DataFrame.rsub

`DataFrame.rsub`(*other*, *axis='columns'*, *level=None*, *fill_value=None*)

Subtraction of dataframe and other, element-wise (binary operator *rsub*).

Equivalent to `other - dataframe`, but with support to substitute a fill_value for missing data in one of the inputs.

Parameters:	**other** : Series, DataFrame, or constant **axis** : {0, 1, 'index', 'columns'} For Series input, axis to match Series index on **level** : int or name Broadcast across a level, matching Index values on the passed MultiIndex level **fill_value** : None or float value, default None Fill existing missing (NaN) values, and any new element needed for successful DataFrame alignment, with this value before computation. If data in both corresponding DataFrame locations is missing the result will be missing
Returns:	**result** : DataFrame

Notes : Mismatched indices will be unioned together

Examples

```
>>> a = pd.DataFrame([2, 1, 1, np.nan], index=['a', 'b', 'c', 'd'],
...                  columns=['one'])
>>> a
   one
a  2.0
b  1.0
c  1.0
d  NaN
>>> b = pd.DataFrame(dict(one=[1, np.nan, 1, np.nan],
...                       two=[3, 2, np.nan, 2]),
...                  index=['a', 'b', 'd', 'e'])
>>> b
   one  two
a  1.0  3.0
```

```
b   NaN  2.0
d   1.0  NaN
e   NaN  2.0
>>> a.sub(b, fill_value=0)
    one  two
a   1.0  -3.0
b   1.0  -2.0
c   1.0  NaN
d   -1.0  NaN
e   NaN  -2.0
```

Unit 2: Data Handling (DH-1)

Matching and Broadcasting operations

Matching / broadcasting behavior

DataFrame has the methods `add()`, `sub()`, `mul()`, `div()` and related functions `radd()`, `rsub()`, … for carrying out binary operations. For broadcasting behavior, Series input is of primary interest. Using these functions, you can use to either match on the *index* or *columns* via the **axis** keyword:

```
In [14]: df = pd.DataFrame({'one' : pd.Series(np.random.randn(3),
                             index=['a', 'b', 'c']),
   ....: 'two' : pd.Series(np.random.randn(4), index=['a', 'b', 'c', 'd']),
   ....: 'three' : pd.Series(np.random.randn(3), index=['b', 'c', 'd'])})
   ....:

In [15]: df
Out[15]:
        one       two     three
a -1.101558  1.124472       NaN
b -0.177289  2.487104 -0.634293
c  0.462215 -0.486066  1.931194
d       NaN -0.456288 -1.222918

In [16]: row = df.iloc[1]

In [17]: column = df['two']

In [18]: df.sub(row, axis='columns')
Out[18]:
        one       two     three
```

```
a -0.924269 -1.362632       NaN
b  0.000000  0.000000  0.000000
c  0.639504 -2.973170  2.565487
d       NaN -2.943392 -0.588625

In [19]: df.sub(row, axis=1)
Out[19]:
        one       two     three
a -0.924269 -1.362632       NaN
b  0.000000  0.000000  0.000000
c  0.639504 -2.973170  2.565487
d       NaN -2.943392 -0.588625

In [20]: df.sub(column, axis='index')
Out[20]:
        one  two     three
a -2.226031  0.0       NaN
b -2.664393  0.0 -3.121397
c  0.948280  0.0  2.417260
d       NaN  0.0 -0.766631

In [21]: df.sub(column, axis=0)
Out[21]:
        one  two     three
a -2.226031  0.0       NaN
b -2.664393  0.0 -3.121397
c  0.948280  0.0  2.417260
d       NaN  0.0 -0.766631
```

Furthermore you can align a level of a multi-indexed DataFrame with a Series.

```
In [22]: dfmi = df.copy()

In [23]: dfmi.index = pd.MultiIndex.from_tuples([(1,'a'),
```

```
                                 (1,'b'),(1,'c'),(2,'a')],
   ....:                 names=['first','second'])
   ....:

In [24]: dfmi.sub(column, axis=0, level='second')
Out[24]:
                   one      two     three
first second
1     a      -2.226031  0.00000       NaN
      b      -2.664393  0.00000 -3.121397
      c       0.948280  0.00000  2.417260
2     a            NaN -1.58076 -2.347391
```

With Panel, describing the matching behavior is a bit more difficult, so the arithmetic methods instead (and perhaps confusingly?) give you the option to specify the *broadcast axis*. For example, suppose we wished to demean the data over a particular axis. This can be accomplished by taking the mean over an axis and broadcasting over the same axis:

```
In [25]: major_mean = wp.mean(axis='major')

In [26]: major_mean
Out[26]:
      Item1     Item2
A -0.878036 -0.092218
B -0.060128  0.529811
C  0.099453 -0.715139
D  0.248599 -0.186535

In [27]: wp.sub(major_mean, axis='major')
Out[27]:
<class 'pandas.core.panel.Panel'>
Dimensions: 2 (items) x 5 (major_axis) x 4 (minor_axis)
Items axis: Item1 to Item2
Major_axis axis: 2000-01-01 00:00:00 to 2000-01-05 00:00:00
```

```
Minor_axis axis: A to D
```

And similarly for `axis="items"` and `axis="minor"`.

Note

I could be convinced to make the **axis** argument in the DataFrame methods match the broadcasting behavior of Panel. Though it would require a transition period so users can change their code...

Series and Index also support the `divmod()` builtin. This function takes the floor division and modulo operation at the same time returning a two-tuple of the same type as the left hand side. For example:

```
In [28]: s = pd.Series(np.arange(10))

In [29]: s
Out[29]:
0    0
1    1
2    2
3    3
4    4
5    5
6    6
7    7
8    8
9    9
dtype: int64

In [30]: div, rem = divmod(s, 3)

In [31]: div
Out[31]:
0    0
```

```
1    0
2    0
3    1
4    1
5    1
6    2
7    2
8    2
9    3
dtype: int64

In [32]: rem
Out[32]:
0    0
1    1
2    2
3    0
4    1
5    2
6    0
7    1
8    2
9    0
dtype: int64

In [33]: idx = pd.Index(np.arange(10))

In [34]: idx
Out[34]: Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype='int64')

In [35]: div, rem = divmod(idx, 3)

In [36]: div
```

```
Out[36]: Int64Index([0, 0, 0, 1, 1, 1, 2, 2, 2, 3], dtype='int64')

In [37]: rem
Out[37]: Int64Index([0, 1, 2, 0, 1, 2, 0, 1, 2, 0], dtype='int64')
```

We can also do elementwise `divmod()`:

```
In [38]: div, rem = divmod(s, [2, 2, 3, 3, 4, 4, 5, 5, 6, 6])

In [39]: div
Out[39]:
0    0
1    0
2    0
3    1
4    1
5    1
6    1
7    1
8    1
9    1
dtype: int64

In [40]: rem
Out[40]:
0    0
1    1
2    2
3    0
4    0
5    1
6    1
7    2
8    2
```

```
9    3
dtype: int64
```

Broadcasting

The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python. It does this without making needless copies of data and usually leads to efficient algorithm implementations. There are, however, cases where broadcasting is a bad idea because it leads to inefficient use of memory that slows computation.

NumPy operations are usually done on pairs of arrays on an element-by-element basis. In the simplest case, the two arrays must have exactly the same shape, as in the following example:

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = np.array([2.0, 2.0, 2.0])
>>> a * b
array([ 2.,  4.,  6.])
```

NumPy’s broadcasting rule relaxes this constraint when the arrays’ shapes meet certain constraints. The simplest broadcasting example occurs when an array and a scalar value are combined in an operation:

```
>>> a = np.array([1.0, 2.0, 3.0])
>>> b = 2.0
>>> a * b
array([ 2.,  4.,  6.])
```

The result is equivalent to the previous example where `b` was an array. We can think of the scalar `b` being *stretched* during the arithmetic operation into an array with the same shape as `a`. The new elements in `b` are simply copies of the original scalar. The stretching analogy is only conceptual. NumPy is smart enough to use the original scalar value without actually making copies, so that broadcasting operations are as memory and computationally efficient as possible.

The code in the second example is more efficient than that in the first because broadcasting moves less memory around during the multiplication (b is a scalar rather than an array).

General Broadcasting Rules

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing dimensions, and works its way forward. Two dimensions are compatible when

1. they are equal, or
2. one of them is 1

If these conditions are not met, a `ValueError: frames are not aligned` exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the maximum size along each dimension of the input arrays.

Arrays do not need to have the same *number* of dimensions. For example, if you have a `256x256x3` array of RGB values, and you want to scale each color in the image by a different value, you can multiply the image by a one-dimensional array with 3 values. Lining up the sizes of the trailing axes of these arrays according to the broadcast rules, shows that they are compatible:

```
Image  (3d array): 256 x 256 x 3
Scale  (1d array):             3
Result (3d array): 256 x 256 x 3
```

When either of the dimensions compared is one, the other is used. In other words, dimensions with size 1 are stretched or "copied" to match the other.

In the following example, both the `A` and `B` arrays have axes with length one that are expanded to a larger size during the broadcast operation:

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

Here are some more examples:

```
A      (2d array):  5 x 4
B      (1d array):      1
Result (2d array):  5 x 4

A      (2d array):  5 x 4
B      (1d array):      4
Result (2d array):  5 x 4

A      (3d array):  15 x 3 x 5
B      (3d array):  15 x 1 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 5
Result (3d array):  15 x 3 x 5

A      (3d array):  15 x 3 x 5
B      (2d array):       3 x 1
Result (3d array):  15 x 3 x 5
```

Here are examples of shapes that do not broadcast:

```
A      (1d array):  3
B      (1d array):  4 # trailing dimensions do not match

A      (2d array):      2 x 1
B      (3d array):  8 x 4 x 3 # second from last dimensions mismatched
```

An example of broadcasting in practice:

```
>>> x = np.arange(4)
>>> xx = x.reshape(4,1)
>>> y = np.ones(5)
>>> z = np.ones((3,4))
```

```
>>> x.shape
(4,)

>>> y.shape
(5,)

>>> x + y
<type 'exceptions.ValueError'>: shape mismatch: objects cannot
                           be broadcast to a single shape

>>> xx.shape
(4, 1)

>>> y.shape
(5,)

>>> (xx + y).shape
(4, 5)

>>> xx + y
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 2.,  2.,  2.,  2.,  2.],
       [ 3.,  3.,  3.,  3.,  3.],
       [ 4.,  4.,  4.,  4.,  4.]])

>>> x.shape
(4,)

>>> z.shape
(3, 4)

>>> (x + z).shape
(3, 4)
```

```
>>> x + z
array([[ 1.,  2.,  3.,  4.],
       [ 1.,  2.,  3.,  4.],
       [ 1.,  2.,  3.,  4.]])
```

Broadcasting provides a convenient way of taking the outer product (or any other outer operation) of two arrays. The following example shows an outer addition operation of two 1-d arrays:

```
>>> a = np.array([0.0, 10.0, 20.0, 30.0])
>>> b = np.array([1.0, 2.0, 3.0])
>>> a[:, np.newaxis] + b
array([[  1.,   2.,   3.],
       [ 11.,  12.,  13.],
       [ 21.,  22.,  23.],
       [ 31.,  32.,  33.]])
```

Here the `newaxis` index operator inserts a new axis into `a`, making it a two-dimensional `4x1` array. Combining the `4x1` array with `b`, which has shape `(3,)`, yields a `4x3` array.

Unit 2: Data Handling (DH-1)

Missing data and filling values

Missing data is always a problem in real life scenarios. Areas like machine learning and data mining face severe issues in the accuracy of their model predictions because of poor quality of data caused by missing values. In these areas, missing value treatment is a major point of focus to make their models more accurate and valid.

When and Why Is Data Missed?

Let us consider an online survey for a product. Many a times, people do not share all the information related to them. Few people share their experience, but not how long they are using the product; few people share how long they are using the product, their experience but not their contact information. Thus, in some or the other way a part of data is always missing, and this is very common in real time.

Let us now see how we can handle missing values (say NA or NaN) using Pandas.

```
# import the pandas library
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df
```

Its **output** is as follows −

```
         one        two      three
```

```
a   0.077988   0.476149   0.965836
b        NaN        NaN        NaN
c  -0.390208  -0.551605  -2.301950
d        NaN        NaN        NaN
e  -2.000303  -0.788201   1.510072
f  -0.930230  -0.670473   1.146615
g        NaN        NaN        NaN
h   0.085100   0.532791   0.887415
```

Using reindexing, we have created a DataFrame with missing values. In the output, **NaN** means **Not a Number.**

Check for Missing Values

To make detecting missing values easier (and across different array dtypes), Pandas provides the **isnull()** and **notnull()** functions, which are also methods on Series and DataFrame objects −

Example 1

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df['one'].isnull()
```

Its **output** is as follows −

```
a  False
b  True
c  False
d  True
```

```
e  False
f  False
g  True
h  False
Name: one, dtype: bool
```

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df['one'].notnull()
```

Its **output** is as follows −

```
a  True
b  False
c  True
d  False
e  True
f  True
g  False
h  True
Name: one, dtype: bool
```

Calculations with Missing Data

- When summing data, NA will be treated as Zero
- If the data are all NA, then the result will be NA

Example 1

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df['one'].sum()
```

Its **output** is as follows −

```
2.02357685917
```

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame(index=[0,1,2,3,4,5],columns=['one','two'])
print df['one'].sum()
```

Its **output** is as follows −

```
nan
```

Cleaning / Filling Missing Data

Pandas provides various methods for cleaning the missing values. The fillna function can “fill in” NA values with non-null data in a couple of ways, which we have illustrated in the following sections.

Replace NaN with a Scalar Value

The following program shows how you can replace "NaN" with "0".

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

```
df = pd.DataFrame(np.random.randn(3, 3), index=['a', 'c', 'e'],
columns=['one', 'two', 'three'])
df = df.reindex(['a', 'b', 'c'])
print df
print ("NaN replaced with '0':")
print df.fillna(0)
```

Its **output** is as follows −

```
        one        two     three
a  -0.576991  -0.741695  0.553172
b        NaN        NaN       NaN
c   0.744328  -1.735166  1.749580

NaN replaced with '0':
        one        two     three
a  -0.576991  -0.741695  0.553172
b   0.000000   0.000000  0.000000
c   0.744328  -1.735166  1.749580
```

Here, we are filling with value zero; instead we can also fill with any other value.

Fill NA Forward and Backward

Using the concepts of filling discussed in the ReIndexing Chapter we will fill the missing values.

Method	**Action**
pad/fill	Fill methods Forward
bfill/backfill	Fill methods Backward

Example 1

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

```
df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])
df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])

print df.fillna(method='pad')
```

Its **output** is as follows −

```
        one        two      three
a  0.077988   0.476149   0.965836
b  0.077988   0.476149   0.965836
c -0.390208  -0.551605  -2.301950
d -0.390208  -0.551605  -2.301950
e -2.000303  -0.788201   1.510072
f -0.930230  -0.670473   1.146615
g -0.930230  -0.670473   1.146615
h  0.085100   0.532791   0.887415
```

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print df.fillna(method='backfill')
```

Its **output** is as follows −

```
        one        two      three
a  0.077988   0.476149   0.965836
b -0.390208  -0.551605  -2.301950
c -0.390208  -0.551605  -2.301950
d -2.000303  -0.788201   1.510072
```

```
e  -2.000303  -0.788201   1.510072
f  -0.930230  -0.670473   1.146615
g   0.085100   0.532791   0.887415
h   0.085100   0.532791   0.887415
```

Drop Missing Values

If you want to simply exclude the missing values, then use the **dropna**function along with the **axis** argument. By default, axis=0, i.e., along row, which means that if any value within a row is NA then the whole row is excluded.

Example 1

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print df.dropna()
```

Its **output** is as follows −

```
        one        two      three
a   0.077988   0.476149   0.965836
c  -0.390208  -0.551605  -2.301950
e  -2.000303  -0.788201   1.510072
f  -0.930230  -0.670473   1.146615
h   0.085100   0.532791   0.887415
```

Example 2

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(5, 3), index=['a', 'c', 'e', 'f',
```

```
'h'],columns=['one', 'two', 'three'])

df = df.reindex(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
print df.dropna(axis=1)
```

Its **output** is as follows −

```
Empty DataFrame
Columns: [ ]
Index: [a, b, c, d, e, f, g, h]
```

Replace Missing (or) Generic Values

Many times, we have to replace a generic value with some specific value. We can achieve this by applying the replace method.

Replacing NA with a scalar value is equivalent behavior of the **fillna()**function.

Example 1

```
import pandas as pd
import numpy as np
df = pd.DataFrame({'one':[10,20,30,40,50,2000],
'two':[1000,0,30,40,50,60]})
print df.replace({1000:10,2000:60})
```

Its **output** is as follows −

```
   one  two
0   10   10
1   20    0
2   30   30
3   40   40
4   50   50
5   60   60
```

Example 2

```
import pandas as pd
import numpy as np
df = pd.DataFrame({'one':[10,20,30,40,50,2000],
'two':[1000,0,30,40,50,60]})
print df.replace({1000:10,2000:60})
```

Its **output** is as follows −

```
   one  two
0   10   10
1   20    0
2   30   30
3   40   40
4   50   50
5   60   60
```

Unit 2: Data Handling (DH-1)

Comparisons and Boolean Reductions

Flexible Comparisons

Series and DataFrame have the binary comparison methods `eq`, `ne`, `lt`, `gt`, `le`, and `ge` whose behavior is analogous to the binary arithmetic operations described above:

```
In [45]: df.gt(df2)
Out[45]:
   one  two three
a False False False
b False False False
c False False False
d False False False

In [46]: df2.ne(df)
Out[46]:
   one  two three
a False False  True
b False False False
c False False False
d  True False False
```

These operations produce a pandas object of the same type as the left-hand-side input that is of dtype `bool`. These `boolean` objects can be used in indexing operations.

Boolean Reductions

You can apply the reductions: `empty`, `any()`, `all()`, and `bool()` to provide a way to summarize a boolean result.

```
In [47]: (df > 0).all()
Out[47]:
one    False
two    False
three  False
dtype: bool

In [48]: (df > 0).any()
Out[48]:
one    True
two    True
three  True
dtype: bool
```

You can reduce to a final boolean value.

```
In [49]: (df > 0).any().any()
Out[49]: True
```

You can test if a pandas object is empty, via the `empty` property.

```
In [50]: df.empty
Out[50]: False

In [51]: pd.DataFrame(columns=list('ABC')).empty
Out[51]: True
```

To evaluate single-element pandas objects in a boolean context, use the method `bool()`:

```
In [52]: pd.Series([True]).bool()
Out[52]: True

In [53]: pd.Series([False]).bool()
Out[53]: False

In [54]: pd.DataFrame([[True]]).bool()
```

```
Out[54]: True

In [55]: pd.DataFrame([[False]]).bool()
Out[55]: False
```

Warning

You might be tempted to do the following:

```
>>> if df:
   ...
```

Or

```
>>> df and df2
```

These will both raise errors, as you are trying to compare multiple values.

```
ValueError: The truth value of an array is ambiguous.
Use a.empty, a.any() or a.all().
```

See gotchas for a more detailed discussion.

Comparing if objects are equivalent

Often you may find that there is more than one way to compute the same result. As a simple example, consider `df+df` and `df*2`. To test that these two computations produce the same result, given the tools shown above, you might imagine using `(df+df == df*2).all()`. But in fact, this expression is False:

```
In [56]: df+df == df*2
Out[56]:
   one  two three
a  True True False
b  True True  True
c  True True  True
d False True  True
```

```
In [57]: (df+df == df*2).all()
Out[57]:
one      False
two       True
three    False
dtype: bool
```

Notice that the boolean DataFrame `df+df == df*2` contains some False values! This is because NaNs do not compare as equals:

```
In [58]: np.nan == np.nan
Out[58]: False
```

So, NDFrames (such as Series, DataFrames, and Panels) have an `equals()` method for testing equality, with NaNs in corresponding locations treated as equal.

```
In [59]: (df+df).equals(df*2)
Out[59]: True
```

Note that the Series or DataFrame index needs to be in the same order for equality to be True:

```
In [60]: df1 = pd.DataFrame({'col':['foo', 0, np.nan]})

In [61]: df2 = pd.DataFrame({'col':[np.nan, 0, 'foo']}, index=[2,1,0])

In [62]: df1.equals(df2)
Out[62]: False

In [63]: df1.equals(df2.sort_index())
Out[63]: True
```

Comparing array-like objects

You can conveniently perform element-wise comparisons when comparing a pandas data structure with a scalar value:

```
In [64]: pd.Series(['foo', 'bar', 'baz']) == 'foo'
Out[64]:
0     True
1    False
2    False
dtype: bool

In [65]: pd.Index(['foo', 'bar', 'baz']) == 'foo'
Out[65]: array([ True, False, False], dtype=bool)
```

Pandas also handles element-wise comparisons between different array-like objects of the same length:

```
In [66]:pd.Series(['foo', 'bar', 'baz']) == pd.Index(['foo', 'bar', 'qux'])
Out[66]:
0     True
1     True
2    False
dtype: bool

In [67]:pd.Series(['foo', 'bar', 'baz']) == np.array(['foo', 'bar', 'qux'])
Out[67]:
0     True
1     True
2    False
dtype: bool
```

Trying to compare `Index` or `Series` objects of different lengths will raise a ValueError:

```
In [55]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo', 'bar'])
ValueError: Series lengths must match to compare

In [56]: pd.Series(['foo', 'bar', 'baz']) == pd.Series(['foo'])
ValueError: Series lengths must match to compare
```

Note that this is different from the NumPy behavior where a comparison can be broadcast:

```
In [68]: np.array([1, 2, 3]) == np.array([2])
Out[68]: array([False, True, False], dtype=bool)
```

or it can return False if broadcasting can not be done:

```
In [69]: np.array([1, 2, 3]) == np.array([1, 2])
Out[69]: False
```

Combining overlapping data sets

A problem occasionally arising is the combination of two similar data sets where values in one are preferred over the other. An example would be two data series representing a particular economic indicator where one is considered to be of “higher quality”. However, the lower quality series might extend further back in history or have more complete data coverage. As such, we would like to combine two DataFrame objects where missing values in one DataFrame are conditionally filled with like-labeled values from the other DataFrame. The function implementing this operation is `combine_first()`, which we illustrate:

```
In [70]: df1 = pd.DataFrame({'A' : [1., np.nan, 3., 5., np.nan],
   ....:                    'B' : [np.nan, 2., 3., np.nan, 6.]})
   ....:

In [71]: df2 = pd.DataFrame({'A' : [5., 2., 4., np.nan, 3., 7.],
   ....:                    'B' : [np.nan, np.nan, 3., 4., 6., 8.]})
   ....:

In [72]: df1
Out[72]:
     A    B
0  1.0  NaN
1  NaN  2.0
2  3.0  3.0
3  5.0  NaN
4  NaN  6.0
```

```
In [73]: df2
Out[73]:
     A    B
0  5.0  NaN
1  2.0  NaN
2  4.0  3.0
3  NaN  4.0
4  3.0  6.0
5  7.0  8.0

In [74]: df1.combine_first(df2)
Out[74]:
     A    B
0  1.0  NaN
1  2.0  2.0
2  3.0  3.0
3  5.0  4.0
4  3.0  6.0
5  7.0  8.0
```

General DataFrame Combine

The `combine_first()` method above calls the more general `DataFrame.combine()`. This method takes another DataFrame and a combiner function, aligns the input DataFrame and then passes the combiner function pairs of Series (i.e., columns whose names are the same).

So, for instance, to reproduce `combine_first()` as above:

```
In [75]: combiner = lambda x, y: np.where(pd.isna(x), y, x)

In [76]: df1.combine(df2, combiner)
Out[76]:
     A    B
0  1.0  NaN
```

```
1 2.0 2.0
2 3.0 3.0
3 5.0 4.0
4 3.0 6.0
5 7.0 8.0
```

Unit 2: Data Handling (DH-1)

Transfer data CSV SQL DataFrame

A **DataFrame** is a table much like in SQL or Excel. It's similar in structure, too, making it possible to use similar operations such as aggregation, filtering, and pivoting. However, because DataFrames are built in Python, it's possible to use Python to program more advanced operations and manipulations than SQL and Excel can offer. As a bonus, the creators of pandas have focused on making the DataFrame operate very quickly, even over large datasets.

DataFrames are particularly useful because powerful methods are built into them. In Python, methods are associated with objects, so you need your data to be in the DataFrame to use these methods. DataFrames can load data through a number of **different data structures and files**, including lists and dictionaries, csv files, excel files, and database records.

The Pandas library documentation defines a DataFrame as a "two-dimensional, size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns)". In plain terms, think of a DataFrame as a table of data, i.e. a single set of formatted two-dimensional data, with the following characteristics:

- There can be multiple rows and columns in the data.
- Each row represents a sample of data,
- Each column contains a different variable that describes the samples (rows).
- The data in every column is usually the same type of data – e.g. numbers, strings, dates.
- Usually, unlike an excel data set, DataFrames avoid having missing values, and there are no gaps and empty values between rows or columns.

By way of example, the following data sets that would fit well in a Pandas DataFrame:

- **In a school system DataFrame** – each row could represent a single student in the school, and columns may represent the students name (string), age (number), date of birth (date), and address (string).

- **In an economics DataFrame,** each row may represent a single city or geographical area, and columns might include the the name of area (string), the population (number), the average age of the population (number), the number of households (number), the number of schools in each area (number) etc.
- **In a shop or e-commerce system DataFrame**, each row in a DataFrame may be used to represent a customer, where there are columns for the number of items purchased (number), the date of original registration (date), and the credit card number (string).

pandas.DataFrame.to_sql

`DataFrame.to_sql`(*name*, *con*, *schema=None*, *if_exists='fail'*, *index=True*, *index_label=None*, *chunksize=None*, *dtype=None*)

Write records stored in a DataFrame to a SQL database.

Databases supported by SQLAlchemy are supported. Tables can be newly created, appended to, or overwritten.

Parameters:	**name** : string Name of SQL table. **con** : sqlalchemy.engine.Engine or sqlite3.Connection Using SQLAlchemy makes it possible to use any DB supported by that library. Legacy support is provided for sqlite3.Connection objects. **schema** : string, optional Specify the schema (if database flavor supports this). If None, use default schema. **if_exists** : {'fail', 'replace', 'append'}, default 'fail' How to behave if the table already exists. • fail: Raise a ValueError. • replace: Drop the table before inserting new values. • append: Insert new values to the existing table. **index** : boolean, default True Write DataFrame index as a column. Uses *index_label* as the column name in

	the table. **index_label** : string or sequence, default None Column label for index column(s). If None is given (default) and *index* is True, then the index names are used. A sequence should be given if the DataFrame uses MultiIndex. **chunksize** : int, optional Rows will be written in batches of this size at a time. By default, all rows will be written at once. **dtype** : dict, optional Specifying the datatype for columns. The keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 legacy mode.
Raises:	**ValueError** When the table already exists and *if_exists* is 'fail' (the default).

Examples

Create an in-memory SQLite database.

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('sqlite://', echo=False)
```

Create a table from scratch with 3 rows.

```
>>> df = pd.DataFrame({'name' : ['User 1', 'User 2', 'User 3']})
>>> df
     name
0  User 1
1  User 2
2  User 3
```

```
>>> df.to_sql('users', con=engine)
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3')]
```

```
>>> df1 = pd.DataFrame({'name' : ['User 4', 'User 5']})
```

```
>>> df1.to_sql('users', con=engine, if_exists='append')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 1'), (1, 'User 2'), (2, 'User 3'),
 (0, 'User 4'), (1, 'User 5')]
```

Overwrite the table with just `df1`.

```
>>> df1.to_sql('users', con=engine, if_exists='replace',
...            index_label='id')
>>> engine.execute("SELECT * FROM users").fetchall()
[(0, 'User 4'), (1, 'User 5')]
```

Specify the dtype (especially useful for integers with missing values). Notice that while pandas is forced to store the data as floating point, the database supports nullable integers. When fetching the data with Python, we get back integer scalars.

```
>>> df = pd.DataFrame({"A": [1, None, 2]})
>>> df
     A
0  1.0
1  NaN
2  2.0
```

```
>>> from sqlalchemy.types import Integer
>>> df.to_sql('integers', con=engine, index=False,
...           dtype={"A": Integer()})
```

```
>>> engine.execute("SELECT * FROM integers").fetchall()
[(1,), (None,), (2,)]
```

Moving data to SQL, CSV, Pandas etc.

CSV

This uses the standard library `csv` module:

```
"""Export to CSV."""
import sys
```

```
import csv
from dbfread import DBF

table = DBF('files/people.dbf')
writer = csv.writer(sys.stdout)

writer.writerow(table.field_names)
for record in table:
    writer.writerow(list(record.values()))
```

The output is:

```
NAME,BIRTHDATE
Alice,1987-03-01
Bob,1980-11-12
```

Pandas Data Frames

```
"""
Load content of a DBF file into a Pandas data frame.

The iter() is required because Pandas doesn't detect that the DBF
object is iterable.
"""
from dbfread import DBF
from pandas import DataFrame

dbf = DBF('files/people.dbf')
frame = DataFrame(iter(dbf))

print(frame)
```

This will print:

```
    BIRTHDATE   NAME
0  1987-03-01  Alice
```

```
1   1980-11-12      Bob
```

The `iter()` is required. Without it Pandas will not realize that it can iterate over the table.

Pandas will create a new list internally before converting the records to data frames. This means they will all be loaded into memory. There seems to be no way around this at the moment.

dataset (SQL)

The dataset package makes it easy to move data to a modern database. Here's how you can insert the `people` table into an SQLite database:

```
"""
Convert a DBF file to an SQLite table.

Requires dataset: https://dataset.readthedocs.io/
"""
import dataset
from dbfread import DBF

# Change to "dataset.connect('people.sqlite')" if you want a file.
db = dataset.connect('sqlite:///:memory:')
table = db['people']

for record in DBF('files/people.dbf', lowernames=True):
    table.insert(record)

# Select and print a record just to show that it worked.
print(table.find_one(name='Alice'))
```

(This also creates the schema.)

dbf2sqlite

You can use the included example program `dbf2sqlite` to insert tables into an SQLite database:

```
dbf2sqlite -o example.sqlite table1.dbf table2.dbf
```

This will create one table for each DBF file. You can also omit the `-o example.sqlite` option to have the SQL printed directly to stdout.

If you get character encoding errors you can pass `--encoding` to override the encoding, for example:

```
dbf2sqlite --encoding=latin1 ...
```