Contents

1 *pandas*

pandas is a high-performance module that provides a comprehensive set of structures for working with data. *pandas* excels at handling structured data, such as data sets containing many variables, working with missing values and merging across multiple data sets.

While extremely useful, *pandas* is not an essential component of the Python scientific stack unlike NumPy, SciPy or matplotlib, and so while *pandas* doesn't make data analysis possible in Python, it makes it much easier. *pandas* also provides high-performance, robust methods for importing from and exporting to a wide range of formats.

1.1 Data Structures

pandas provides a set of data structures which include Series,DataFrames and Panels. Series are 1-dimensional arrays. DataFrames are collections of Series and so are 2-dimensional, and Panels are collections ofDataFrames, and so are 3-dimensional. Note that the Panel type is not covered in this chapter. Series are the primary building block of the data structures in pandas, and in many ways a Series behaves similarly to a NumPy array. A Series is initialized using a list or tupel, or directly from a NumPy array.

```
>>> a = array([0.1, 1.2, 2.3, 3.4, 4.5])
>>> a
>>> from pandas import Series
>>> s = Series([0.1, 1.2, 2.3, 3.4, 4.5])
>>> s
>>> s = Series(a) # NumPy array to Series
```

Series, like arrays, are sliceable. However, unlike a 1-dimensional array, a Series has an additional column – an index – which is a set of values which are associated with the rows of the Series. In this example, pandas has automatically generated an index using the sequence 0, 1, . . . since none was provided. It is also possible to use other values as the index when initializing the Series using a keyword argument.

```
>>> s = Series([0.1, 1.2, 2.3, 3.4, 4.5], index = ['a','b','c','d','e'])
>>> s
```

The index enhances the usefulness of the pandas's data structures (Series and-DataFraame) and allows for dictionary-like access to elements in the index (in addition to both numeric slicing and logical indices).

```
>>> s['a']
0.10000000000000001
>>> s[0]
0.10000000000000001
>>> s[['a','c']]
a 0.1
c 2.3
dtype: float64
>>> s[[0,2]]
a 0.1
c 2.3
dtype: float64
>>> s[:2]
a 0.1
b 1.2
dtype: float64
>>> s[s>2]
c 2.3
d 3.4
e 4.5
dtype: float64
```

In this examples, 'a' and 'c' behave in the same manner as 0 and 2 would in a standard NumPy array. The elements of an index do not have to be unique which another way in which a Series generalizes a NumPy array.

```
>>> s = Series([0.1, 1.2, 2.3, 3.4, 4.5], index = ['a','b','c','a','b'])
```

Using numeric index values other than the default sequence will break scalar selection since there is ambiguity between numerical slicing and index access. For this reason, custom numerical indices should be used with care.

```
>>> s
a 0.1
b 1.2
c 2.3
```

```
a 3.4
b 4.5
dtype: float64
>>> s['a']
a 0.1
a 3.4
dtype: float64
```

Series can also be initialized directly from dictionaries.

```
>>> s = Series({'a': 0.1, 'b': 1.2, 'c': 2.3})
>>> s
```

Series are like NumPy arrays in that they support most numerical operations.

```
>>> s = Series({'a': 0.1, 'b': 1.2, 'c': 2.3})
>>> s * 2.0
>>> s- 1.0
```

However, Series are different from arrays when math operations are performed across two Series. In particular, math operations involving two series operate by aligning indices. The mathematical operation is performed in two steps.
First, the union of all indices is created, and then the mathematical operation is performed on matching indices. Indices that do not match are given the value `NaN` (not a number), and values are computed for all unique pairs of repeated indices.

```
>>> s1 = Series({'a': 0.1, 'b': 1.2, 'c': 2.3})
>>> s2 = Series({'a': 1.0, 'b': 2.0, 'c': 3.0})
>>> s3 = Series({'c': 0.1, 'd': 1.2, 'e': 2.3})
>>> s1 + s2
>>>
>>> s1 * s2
>>>
>>> s1 + s3
>>>
```

Mathematical operations performed on series which have non-unique indices will broadcast the operation to all indices which are common. For example, when one array has 2 elements with the same index, and another has 3, adding the two will produce 6 outputs.

```
>>> s1 = Series([1.0,2,3],index=['a']*3)
>>> s2 = Series([4.0,5],index=['a']*2)
>>> s1 + s2
a 5
a 6
a 6
a 7
a 7
a 8
dtype: float64
```

The underlying NumPy array is accessible through the values property, and the index is accessible the index property, which returns an Index type. The NumPy array underlying the index can be retrieved using values on the Index object returned.

```
>>> s1 = Series([1.0,2,3])
>>> s1.values
>>> s1.index
>>> s1.index.values
>>> s1.index = ['cat','dog','elephant']
>>> s1.index
```

1.2 Notable Methods and Properties

Series provide a large number of methods to manipulate data. These can broadly be categorized into mathematical and non-mathematical functions. The mathematical functions are generally very similar

to those in NumPy due to the underlying structure of a Series, and generally do not warrant a separate discussion. In contrast, the non-mathematical methods are unique to pandas.

`head` and `tail`

`head()` shows the first 5 rows of a series, and `tail()` shows the last 5 rows. An optional argument can be used to return a different number of entries, as in `head(10)`.

`isnull` and `notnull`

- `isnull()` returns a Series with the same indices containing Boolean values indicating True for null values which include NaN and None, among others.
- `notnull()` returns the negation of `isnull()` – that is, True for non-null values, and False otherwise.

`ix`

`ix` is the indexing function and `s.ix[0:2]` is the same as `s[0:2]`. `ix` is quite useful for DataFrames.

`describe`

`describe()` returns a simple set of summary statistics about a Series. The values returned is a series where the index contains name of the statistics computed.

```
>>> s1 = Series(arange(10.0,20.0))
>>> s1.describe()
>>> summ = s1.describe()
>>> summ['mean']
```

`unique` and `nunique`

`unique()` returns the unique elements of a series and `nunique()` returns the number of unique values in a Series.

`drop` and `dropna`

`drop(labels)` drop elements with the selected labels from a Series.

```
drop(labels) dro~+p elements with the selected labels forma Series.
>>> s1 = Series(arange(1.0,6),index=['a','a','b','c','d'])
>>> s1
>>> s1.drop('a')
```

`dropna()` is similar to `drop()` except that it only drops null values – NaN or similar.

```
>>> s1 = Series(arange(1.0,4.0),index=['a','b','c'])
>>> s2 = Series(arange(1.0,4.0),index=['c','d','e'])
>>> s3 = s1 + s2
>>> s3
>>> s3.dropna()
```

Both return Series and so it is necessary to assign the values to have a series with the selected elements dropped.

`fillna`

`fillna(value)` fills all null values in a series with a specific value.

```
>>> s1 = Series(arange(1.0,4.0),index=['a','b','c'])
>>> s2 = Series(arange(1.0,4.0),index=['c','d','e'])
>>> s3 = s1 + s2
>>> s3.fillna(1.0)
a 1
b 1
c 4
d 1
e 1
dtype: float64
```

append

append(series) appends one series to another, and is similar to list.append.

replace

replace(list,values) replaces a set of values in a Series with a new value. replace is similar to fillna except that replace also replaces non-null values.

update

update(series) replaces values in a series with those in another series, matching on the index, and is similar to a SQL update operation.

```
>>> s1 = Series(arange(1.0,4.0),index=['a','b','c'])
>>> s1

dtype: float64
>>> s2 = Series(1.0
* arange(1.0,4.0),index=['c','d','e'])
>>> s1.update(s2)
>>> s1
```

1.3 DataFrame

While the Series class is the building block of data structures in pandas, the DataFrame is the work-horse. DataFrames collect multiple series in the same way that a spreadsheet collects multiple columns of data. In a simple sense, a DataFrame is like a 2-dimensional NumPy array – and when all data is numeric and of the same type (e.g. float64), it is virtually indistinguishable. However, a DataFrame is composed of Series and each Series has its own data type, and so not allDataFrames are representable as homogeneous NumPy arrays. A number of methods are available to initialize a DataFrame. The simplest is from a homogeneous NumPy array.

```
>>> from pandas import DataFrame
>>> a = array([[1.0,2],[3,4]])
>>> df = DataFrame(a)
>>> df
0 1
0 1 2
177
1 3 4
```

Like a Series, a DataFrame contains the input data as well as row labels. However, since a DataFrame is a collection of columns, it also contains column labels (located along the top edge). When none are provided, the numeric sequence 0, 1, . . . is used. Column names are entered using a keyword argument or later by assigning to columns.

```
>>> df = DataFrame(array([[1,2],[3,4]]),columns=['a','b'])
>>> df
a b
0 1 2
1 3 4
>>> df = DataFrame(array([[1,2],[3,4]]))
>>> df.columns = ['dogs','cats']
>>> df
dogs cats
0 1 2
1 3 4
```

Index values are similarly assigned using either the keyword argument index or by setting the index property.

```
>>> df = DataFrame(array([[1,2],[3,4]]), columns=['dogs','cats'], index=['Alice','Bob']
>>> df
dogs cats
Alice 1 2
Bob 3 4
```

DataFrames can also be created from NumPy arrays with structured data.

```
>>> import datetime
>>> t = dtype([('datetime', 'O8'), ('value', 'f4')])
>>> x = zeros(1,dtype=t)
>>> x[0][0] = datetime.datetime(2013,01,01)
>>> x[0][1] = 99.99
>>> x
array([(datetime.datetime(2013, 1, 1, 0, 0), 99.98999786376953)],
dtype=[('datetime', 'O'), ('value', '<f4')])
>>> df = DataFrame(x)
>>> df
datetime value
0 20130101
99.989998
```

In the previous part, the DataFrame has automatically pulled the column names and column types from the NumPy structured data. The final method to create a DataFrame uses a dictionary containing Series, where the keys contain the column names. The DataFrame will automatically align the data using the common indices.

```
>>> s1 = Series(arange(0.0,5))
>>> s2 = Series(arange(1.0,3))
>>> DataFrame({'one': s1, 'two': s2})
178
one two
0 0 1
1 1 2
2 2 3
3 3 4
4 4 5
>>> s3 = Series(arange(0.0,3))
>>> DataFrame({'one': s1, 'two': s2, 'three': s3})
one three two
0 0 0 1
1 1 1 2
2 2 2 3
3 3 NaN 4
4 4 NaN 5
```

In the final example, the third series (s3) has fewer values and the DataFrame automatically fills missing values as NaN. Note that is possible to create DataFrames from Series which do not have unique index values, although in these cases the index values of the two series must match exactly – that is, have the same index values in the same order.

1.4 Manipulating DataFrames

The use of DataFrames will be demonstrated using a data set containing a mix of data types using statelevel GDP data from the US. The data set contains both the GDP level between 2009 and 2012 (constant 2005 US$) and the growth rates for the same years as well as a variable containing the region of the state. The data is loaded directly into a DataFrame using read_excel, which is described in Section 17.4.

```
>>> from pandas import read_excel
>>> state_gdp = read_excel('US_state_GDP.xls','Sheet1')
>>> state_gdp.head()
state_code state gdp_2009 gdp_2010 gdp_2011 gdp_2012
0 AK Alaska 44215 43472 44232 44732
1 AL Alabama 149843 153839 155390 157272
2 AR Arkansas 89776 92075 92684 93892
3 AZ Arizona 221405 221016 224787 230641
4 CA California 1667152 1672473 1692301 1751002
gdp_growth_2009 gdp_growth_2010 gdp_growth_2011 gdp_growth_2012 region
0 7.7 1.7
1.7 1.1 FW
1 3.9
2.7 1.0 1.2 SE
2 2.0
2.6 0.7 1.3 SE
3 8.2
0.2
1.7 2.6 SW
4 5.1
0.3 1.2 3.5 FW
```

1.5 Selecting Columns

Single columns are selectable using the columnname, as in `state_gdp['state']`, and the value returned in a Series. Multiple columns are similarly selected using a list of column names as in `state_gdp [['state_code', 'state']]`, or equivalently using an Index object. Note that these two methods are slightly different – selecting a single column returns a Series while selecting multiple columns returns a DataFrame. This is similar to how NumPy's scalar selection returns an array with a lower dimension. Use a list of column names containing a single name to return a DataFrame with a single column.

```
>>> state_gdp['state_code'].head() # Series
0 AK
```

```
1 AL
2 AR
3 AZ
4 CA
Name: state_code, dtype: object
>>> state_gdp[['state_code']].head() # DataFrame
state_code
0 AK
1 AL
2 AR
3 AZ
4 CA
>>> state_gdp[['state_code','state']].head()
state_code state
0 AL Alabama
1 AK Alaska
2 AZ Arizona
3 AR Arkansas
4 CA California
>>> index = state_gdp.index
>>> state_gdp[index[1:3]].head() # Elements 1 and 2 (0based
counting)
state gdp_2009
0 Alabama 149843
1 Alaska 44215
2 Arizona 221405
3 Arkansas 89776
4 California 1667152
```

Finally, single columns can also be selected using dot-notation and the column name.2 This is identical to using df['column'] and so the value returned is a Series.

```
>>> state_gdp.state_code.head()
0 AL
1 AK
2 AZ
3 AR
4 CA
Name: state_code, dtype: object
```

The column name must be a legal Python variable name, and so cannot contain spaces or reserved notation.

```
>>> type(state_gdp.state_code)
pandas.core.series.Series
```

Selecting Rows Rows can be selected using standard numerical slices.

```
>>> state_gdp[1:3]
state_code state gdp_2009 gdp_2010 gdp_2011 gdp_2012
1 AL Alabama 149843 153839 155390 157272
2 AR Arkansas 89776 92075 92684 93892
gdp_growth_2009 gdp_growth_2010 gdp_growth_2011 gdp_growth_2012 region
1 3.9
2.7 1.0 1.2 SE
2 2.0
2.6 0.7 1.3 SE
```

A function version is also available using iloc[rows] which is identical to the standard slicing syntax. Labeled rows can also be selected using the method loc[label] or loc[list of labels] to elect multiple rows using their label . Finally, rows can also be selected using logical selection using a Boolean array with the same number of elements as the number of rows as the DataFrame.

```
>>> state_long_recession = state_gdp['gdp_growth_2010']<0
>>> state_gdp[state_long_recession].head()
state_code state gdp_2009 gdp_2010 gdp_2011 gdp_2012
1 AK Alaska 44215 43472 44232 44732
2 AZ Arizona 221405 221016 224787 230641
28 NV Nevada 110001 109610 111574 113197
50 WY Wyoming 32439 32004 31231 31302
gdp_growth_2009 gdp_growth_2010 gdp_growth_2011 gdp_growth_2012
1 7.7 1.7
1.7 1.1
2 8.2
0.2
```

```
1.7 2.6
28 8.2
0.4
1.8 1.5
50 3.4 1.3
2.4
0.2
```

1.6 Selecting Rows and Columns

Since the behavior of slicing depends on whether the input is text (selects columns) or numeric/Boolean (selects rows), it isn't possible to use standard slicing to select both rows and columns.
Instead, the selector method `ix[rowselector,colselector]` allows joint selection where `rowselector` is either a scalar selector, a slice selector, a Boolean array, a numeric selector or a row label or list of row labels and colselector is a scalar selector, a slice selector, a Boolean array, a numeric selector or a column name or list of column names.

```
>>> state_gdp.ix[state_long_recession,'state']
1 Alaska
2 Arizona
28 Nevada
50 Wyoming
181
Name: state, dtype: object
>>> state_gdp.ix[state_long_recession,['state','gdp_growth_2009','gdp_growth_2010']]
state gdp_growth_2009 gdp_growth_2010
1 Alaska 7.7 1.7
2 Arizona 8.2
0.2
28 Nevada 8.2
0.4
50 Wyoming 3.4 1.3
>>> state_gdp.ix[10:15,0] # Slice and scalar
10 GA
11 HI
12 IA
13 ID
14 IL
15 IN
```

```
>>> state_gdp.ix[10:15,:2] # Slice and slice
state_code state
10 GA Georgia
11 HI Hawaii
12 IA Iowa
13 ID Idaho
14 IL Illinois
15 IN Indiana
```

1.7 Adding Columns

Columns are added using one of three methods. The most obvious is to add a Series merging along the index using a dictionary-like syntax.

```
>>> state_gdp_2012 = state_gdp[['state','gdp_2012']]
>>> state_gdp_2012.head()
state gdp_2012
0 Alabama 157272
1 Alaska 44732
2 Arizona 230641
3 Arkansas 93892
4 California 1751002
>>> state_gdp_2012['gdp_growth_2012'] = state_gdp['gdp_growth_2012']
>>> state_gdp_2012.head()
state gdp_2012 gdp_growth_2012
0 Alabama 157272 1.2
1 Alaska 44732 1.1
2 Arizona 230641 2.6
3 Arkansas 93892 1.3
```

This syntax always adds the column at the end. `insert(location,column_name,series)` inserts a Series at an specified location, where location uses 0-based indexing (i.e. 0 places the column first, 1 places it 182 second, etc.), `column_name` is the name of the column to be added and `series` is the series data. series is either a Series or another object that is readily convertible into a Series such as a NumPy array.

```
>>> state_gdp_2012 = state_gdp[['state','gdp_2012']]
>>> state_gdp_2012.insert(1,'gdp_growth_2012',state_gdp['gdp_growth_2012'])
>>> state_gdp_2012.head()
state gdp_growth_2012 gdp_2012
0 Alabama 1.2 157272
1 Alaska 1.1 44732
2 Arizona 2.6 230641
3 Arkansas 1.3 93892
4 California 3.5 1751002
```

Formally this type of join performs a left join which means that only index values in the base DataFrame will appear in the combined DataFrame, and so inserting columns with different indices or fewer items than the DataFrame results in a DataFrame with the original indices with NaN-filled missing values in the new Series.

```
>>> state_gdp_2012 = state_gdp.ix[0:2,['state','gdp_2012']]
>>> state_gdp_2012
state gdp_2012
0 Alabama 157272
1 Alaska 44732
2 Arizona 230641
>>> gdp_2011 = state_gdp.ix[1:4,'gdp_2011']
>>> state_gdp_2012['gdp_2011'] = gdp_2011
state gdp_2012 gdp_2011
0 Alabama 157272 NaN
1 Alaska 44732 44232
2 Arizona 230641 224787
```

1.8 Deleting Columns

Columns are deleted using the del keyword, using pop(column) on the DataFrame or by calling `drop(list of columns,axis=1)` . The behavior of these differs slightly: `del` will simply delete the Series from the DataFrame.
`pop()` will both delete the Series and return the Series as an output, and `drop()` will return a DataFrame with the Series dropped by will not modify the original DataFrame.

```
>>> state_gdp_copy = state_gdp.copy()
>>> state_gdp_copy = state_gdp_copy[['state_code',
 'gdp_growth_2011','gdp_growth_2012']]
>>> state_gdp_copy.index = state_gdp['state_code']
>>> state_gdp_copy.head()
>>>
>>> gdp_growth_2012 = state_gdp_copy.pop('gdp_growth_2012')
>>> gdp_growth_2012.head()
>>>
>>> state_gdp_copy.head()
>>> del state_gdp_copy['gdp_growth_2011']
>>> state_gdp_copy.head()
>>> state_gdp_copy = state_gdp.copy()
>>> state_gdp_copy = state_gdp_copy[['state_code',
 'gdp_growth_2011','gdp_growth_2012']]
```

```
>>> state_gdp_dropped = state_gdp_copy.drop(['state_code'
 ,'gdp_growth_2011'],axis=1)
>>> state_gdp_dropped.head()
```

1.9 Notable Properties and Methods

drop, dropna and drop_duplicates

`drop()`, `dropna()` and `drop_duplicates()` can all be used to drop rows or columns from a DataFrame. drop(labels) drops rows based on the rowlabels in a label or list labels. drop(column_name,axis=1) drops columns based on a column name or list column names.

`dropna()` drops rows with anyNaN(or null) values. It can be used with the keyword argument dropna(how='all') to only drop rows which have missing values for all variables. It can also be used with the keyword argument dropna(axis=1) to drop columns with missing values. Finally, `drop_duplicates()` removes rows which are duplicates or other rows, and is used with the keyword argument `drop_duplicates(cols=col_list)` to only consider a subset of all columns when checking for duplicates.

values and index

values retrieves a the NumPy array (structured if the data columns are heterogeneous) underlying the DataFrame, and index returns the index of the DataFrame or can be assigned to to set the index.

fillna

`fillna()` fills NaN or other null values with other values. The simplest use fill all `NaN`s with a single value and is called `fillna(value=value )`. Using a dictionary allows for more sophisticated na-filling with column names as the keys and the replacements as the values.

```
>>> df = DataFrame(array([[1, nan],[nan, 2]]))
>>> df.columns = ['one','two']
>>> replacements = {'one':1,
'two':2}
>>> df.fillna(value=replacements)
```

`T` and `transpose`

`T` and `transpose` are identical – both swap rows and columns of a DataFrame. `T` operates like a property, while `transpose` is used as a method.

`sort` and `sort_index`

`sort` and `sort_index` are identical in their outcome and only differ in the inputs. The default behavior of sort is to sort using the index. Using a keyword argument axis=1 sorts the DataFrame by the column names.

Both can also be used to sort by the data in the DataFrame. sort does this using the keyword argument columns, which is either a single column name or a list of column names, and using a list of column names produces a nested sort. `sort_index` uses the keyword argument by to do the same.

Another keyword argument determines the direction of the sort (ascending by default). `sort(ascending=False)` will produce a descending sort, and when using a nested sort, the sort direction is specified using a list
`sort(columns=['one','two'], ascending=[True,False])`
where each entry corresponds to the columns used to sort.

```
>>> df = DataFrame(array([[1, 3],[1, 2],[3, 2],[2,1]]),
 columns=['one','two'])
>>> df.sort(columns='one')
>>>
>>> df.sort(columns=['one','two'])
>>>
>>> df.sort(columns=['one','two'], ascending=[0,1])
```

The default behavior is to not sort in-place and so it is necessary to assign the output of a sort. Using the keyword argument `inplace=True` will change the default behavior.

`pivot`

`pivot` reshapes a table using column values when reshaping. `pivot` takes three inputs. The first, index, defines the column to use as the index of the pivoted table. The second, columns, defines the column to use to formthe column names, and values defines the columns to for the data in the constructed DataFrame.
The following example showhowa flatDataFrame with repeated values is transformed into a more meaningful representation.

```
>>> prices = [101.0,102.0,103.0]
>>> tickers = ['GOOG','AAPL']
>>> import itertools
>>> data = [v for v in itertools.product(tickers,prices)]
>>> import pandas as pd
>>> dates = pd.date_range('20130103',
periods=3)
>>> df = DataFrame(data, columns=['ticker','price'])
>>> df['dates'] = dates.append(dates)
>>> df
>>>
>>> df.pivot(index='dates',columns='ticker',values='price')
>>>
```

`stack` and `unstack`

`stack` and `unstack` transform a DataFrame to a Series (stack) and back to a DataFrame (`unstack`). The stacked DataFrame (a Series) uses an index containing both the original row and column labels.

`concat` and `append`

append appends rows of another DataFrame to the end of an existing DataFrame. If the data appended has a different set of columns, missing values are `NaN`-filled. The keyword argument `ignore_index=True` instructs append to ignore the existing index in the appendedDataFrame. This is useful when index values are not meaningful, such as when they are simple numeric values.

`pd.concat` is a core function which concatenates two or more DataFrames using an outer join by default. An outer join is a method of joining DataFrames which will returns a DataFrame using the union of the indices of input Data Frames. This differs fromthe left join that is used when adding a Series to an existing DataFrame using dictionary syntax.

The keyword argument `join='inner'` can be used to perform an inner join, which will return a DataFrame using the intersection of the indices in the input DataFrames. Be default pd.concat will concatenate using column names, and the keyword argument axis=1 can be used to join using index labels.

```
>>> df1 = DataFrame([1,2,3],index=['a','b','c'],columns=['one'])
>>> df2 = DataFrame([4,5,6],index=['c','d','e'],columns=['two'])
>>> pd.concat((df1,df2), axis=1)
one two
a 1 NaN
b 2 NaN
c 3 4
d NaN 5
e NaN 6
>>> pd.concat((df1,df2), axis=1, join='inner')
one two
c 3 4
```

`reindex`, `reindex_like` and `reindex_axis`

`reindex` changes the labels while null-filling any missing values, which is useful for selecting subsets of a DataFrameor re-ordering rows. `reindex_like` behaves similarly, but uses the index from another DataFrame. The keyword argument axis directs `reindex_axis` to alter either rows or columns.

```

>>> original = DataFrame([[1,1],[2,2],[3.0,3]],index=['a','b','c'],
columns=['one','two'])
>>> original.reindex(index=['b','c','d'])
>>>
>>> different = DataFrame([[1,1],[2,2],[3.0,3]],index=['c','d','e'],
columns=['one','two'])
>>> original.reindex_like(different)
>>>
>>> original.reindex_axis(['two','one'], axis = 1)
```

`merge` and `join`

`merge` and `join` provide SQL-like operations for merging the DataFrames using row labels or the contents of columns.
The primary difference between the two is that `merge` defaults to using column contents while `join` defaults to using index labels. Both commands take a large number of optional inputs. The important keyword arguments are:

- how, which must be one of `'left', 'right', 'outer', 'inner'` describes which set of indices to use when performing the join.
- '`left`' uses the indices of the DataFrame that is used to call the method and '`right`' uses the DataFrame input into merge or join.
- '`outer`' uses a union of all indices from both DataFrames and '`inner`' uses an intersection from the two DataFrames.
- `on` is a single column name of list of column names to use in the merge. on assumes the names are common. If no value is given for on or `left_on/right_on`, then the common column names are used.
- `left_on` and `right_on` allow for a merge using columns with different names. When left_on and right_on contains the same column names, the behavior is the same as on.
- `left_index` and `right_index` indicate that the index labels are the join key for the left and right DataFrames.

```
>>> left = DataFrame([[1,2],[3,4],[5,6]],columns=['one','two'])
>>> right = DataFrame([[1,2],[3,4],[7,8]],columns=['one','three'])
>>> left.merge(right,on='one') # Same as how='inner'
>>>
>>> left.merge(right,on='one', how='left')
>>>
>>> left.merge(right,on='one', how='right')
>>>
>>> left.merge(right,on='one', how='outer')
>>>
```

`update`

`update` updates the values in one DataFrame using the non-null values from another DataFrame, using the index labels to determine which records to update.

```
>>> left = DataFrame([[1,2],[3,4],[5,6]],columns=['one','two'])
>>> left
>>> right = DataFrame([[nan,12],[13,nan],[nan,8]],
 columns=['one','two'],index=[1,2,3])
>>> right
>>>
>>> left.update(right) # Updates values in left
>>> left
```

`groupby`

`groupby` produces a `DataFrameGroupBy` object which is a `groupedDataFrame`, and is useful when a DataFrame has columns containing group data (e.g. sex or race in cross-sectional data). By itself, `groupby` does not produce any output, and so it is necessary to use other functions on the output `DataFrameGroupBy`.

```
>>> subset = state_gdp[['gdp_growth_2009','gdp_growth_2010','region']]
>>> subset.head()
>>>
>>> grouped_data = subset.groupby(by='region')
>>> grouped_data.groups # Lists group names and index labels for group
>>>
>>> grouped_data.mean()
>>> grouped_data.std() # Can use other methods
```

`apply`

`apply` executes a function along the columns or rows of a DataFrame. The following example applies the mean function both down columns and across rows, which is a trivial since mean could be executed on the DataFrame directly. `apply` is more general since it allows custom functions to be applied to a DataFrame.

```
>>> subset = state_gdp[['gdp_growth_2009','gdp_growth_2010','gdp_growth_2011','gdp_grow
>>> subset.index = state_gdp['state_code'].values
>>> subset.head()
gdp_growth_2009 gdp_growth_2010 gdp_growth_2011 gdp_growth_2012
AK 7.7 1.7
1.7 1.1
AL 3.9
2.7 1.0 1.2
AR 2.0
2.6 0.7 1.3
AZ 8.2
0.2
1.7 2.6
CA 5.1
0.3 1.2 3.5
>>> subset.apply(mean) # Same as subset.mean()
gdp_growth_2009 2.313725
gdp_growth_2010 2.462745
gdp_growth_2011 1.590196
gdp_growth_2012 2.103922
dtype: float64
>>> subset.apply(mean, axis=1).head() # Same as subset.mean(axis=1)
```

`applymap`

`applymap` is similar to apply, only that it applies element-by-element rather than column- or row-wise.

`pivot_table`

`pivot_table` provides a method to summarize data by groups. A pivot table first forms groups based using the keyword argument index and then returns an aggregate of all values within the group (using mean by default). The keyword argument `aggfun` allows for other aggregation function.

```
>>> subset = state_gdp[['gdp_growth_2009','gdp_growth_2010','region']]
>>> subset.head()
gdp_growth_2009 gdp_growth_2010 region
0 7.7 1.7
```

```
FW
1 3.9
2.7 SE
2 2.0
2.6 SE
3 8.2
0.2
SW
4 5.1
0.3 FW
>>> subset.pivot_table(index='region')
gdp_growth_2009 gdp_growth_2010
region
FW 2.483333
1.550000
GL 5.400000
3.660000
MW 1.250000
2.433333
NE 2.350000
2.783333
PL 1.357143
2.900000
RM 0.940000
1.380000
SE 2.633333
2.850000
SW 2.175000
1.325000
```

`pivot_table` differs from pivot since an aggregation function is used when transforming the data.

1.10 Statistical Function

pandas Series and DataFrame are derived from NumPy arrays and so the vast majority of simple statistical functions are available. This list includes `sum, mean, std, var, skew, kurt, prod, median, quantile, abs, cumsum`, and `cumprod`. DataFrame also supports `cov` and `corr` – the keyword argument axis determines the direction of the operation (0 for down columns, 1 for across rows). Some Particular statistical routines are described below.

count

`count` returns number of non-null values – that is, those which are not `NaN` or another null value such as `None` or `NaT` (not a time, for datetimes).

describe

`describe` provides a summary of the Series or DataFrame.

```
state_gdp.describe()
```

`value_counts`

`value_counts` performs frequency counting (“histogramming”) of a Series or DataFrame.

```
>>> state_gdp.region.value_counts()
SE 12
PL 7
NE 6
FW 6
MW 6
GL 5
RM 5
SW 4
dtype: int64
```

2 More pandas Functionality

2.1 Time-series Data

A TimeSeries is basically a series where the index contains datetimes index values (more formally the class TimeSeries inherits from Series), and Series constructor will automatically promote a Series with datetime index values to a TimeSeries. The TimeSeries examples all make use of US real GDP data from the Federal Reserve Economic Database (FRED).

The pandas `TimeSeries` object is currently limited to a span of about 585 years centered at 1970. While this is unlikely to create problems, it may not be appropriate for some applications. pandas includes a substantial number of routines which are primarily designed to work with timeseries data.

```
>>> GDP\_data = read\_excel('GDP.xls','GDP',skiprows=19)
>>> GDP\_data.head()
DATE VALUE
0 19470101
00:00:00 243.1
1 19470401
00:00:00 246.3
2 19470701
00:00:00 250.1
3 19471001
00:00:00 260.3
4 19480101
00:00:00 266.2
>>> type(GDP\_data.VALUE)
pandas.core.series.Series
>>> gdp = GDP\_data.VALUE
>>> gdp.index = GDP\_data.DATE
>>> gdp.head()
DATE
19470101
243.1
19470401
246.3
19470701
250.1
19471001
260.3
19480101
266.2
Name: VALUE, dtype: float64
>>> type(gdp.index)
pandas.tseries.index.DatetimeIndex
```

TimeSeries have some useful indexing tricks. For example, all of the data for a particular year can retrieved using `gdp['yyyy']` syntax where yyyy is a year.

```
>>> gdp['2009']
DATE
20090101
14381.2
20090401
14342.1
20090701
14384.4
20091001
14564.1
Name: VALUE, dtype: float64
>>> gdp['200904']
# All for a particular month
DATE
20090401
14342.1
Name: VALUE, dtype: float6
Dates can also be used for slicing using the notation gdp['d1:d2:'] where d1 and d2 are
formats (e.g '2009' or '20090101')
>>> gdp['2009':'2010']
DATE
20090101
14381.2
20090401
14342.1
20090701
14384.4
20091001
14564.1
20100101
14672.5
20100401
14879.2
20100701
15049.8
20101001
15231.7
Name: VALUE, dtype: float64
>>> gdp['20090601':'
20100601']
DATE
```

```
20090701
14384.4
20091001
14564.1
20100101
14672.5
20100401
14879.2
Name: VALUE, dtype: float64
```

Slicing indexing can also be accomplished using datetime, for example `gdp[ datetime(2009,01,01): datetime(2011,12,31)]` where datetime has been imported using from pandas `import datetime`.

2.2 `date_range`

`date_range` is a very useful function provided by pandas to generate ranges of dates (from pandas `import date_range`). The basic use is either `date_range(beginning_date,ending_date)` which will produce a daily series between the two dates (inclusive) or `date_range(beginning_date, periods=periods)` which will produce a daily series starting at `beginning_date` with periods periods.

```
>>> from pandas import date\_range
>>> date\_range('20130103','
20130105')
<class 'pandas.tseries.index.DatetimeIndex'>
[20130103
00:00:00, ..., 20130105
00:00:00]
Length: 3, Freq: D, Timezone: None
>>> date\_range('20130103',
periods = 3)
<class 'pandas.tseries.index.DatetimeIndex'>
[20130103
00:00:00, ..., 20130105
00:00:00]
Length: 3, Freq: D, Timezone: None
```

The keyword argument freq changes the frequency, and common choices include Scaling the frequency produces skips that are a multiple of the default, such as in 2D which uses every other day. Combining multiple frequencies produces less regular skips, e.g. 2H10T.

```
>>> date\_range('20130103',
periods=4, freq='Q').values
array(['20130331T00:
00:00.000000000+0000',
'20130630T01:
00:00.000000000+0100',
'20130930T01:
00:00.000000000+0100',
'20131231T00:
00:00.000000000+0000'], dtype='datetime64[ns]')
>>> date\_range('20130103',
periods=4, freq='7D4H').values
array(['20130103T00:
00:00.000000000+0000',
'20130110T04:
00:00.000000000+0000',
'20130117T08:
00:00.000000000+0000',
'20130124T12:
00:00.000000000+0000'], dtype='datetime64[ns]')
```

Note that the underlying array uses NumPy's datetime64 as the data type (with nano-second resolution, indicated by [ns]).

resample

pandas supports sophisticated resampling which is useful for aggregating form a higher frequency to a lower one using resample. This example uses annual ('A') and alternative aggregation functions.

```
>>> gdp.resample('A',how=mean).tail() # Annual average
DATE
20091231
14417.950
20101231
14958.300
```

```
20111231
15533.825
20121231
16244.575
20131231
16601.600
Freq: ADEC,
dtype: float64
>>> gdp.resample('A',how=max).tail() # Maximum
DATE
20091231
14564.1
20101231
15231.7
20111231
15818.7
20121231
16420.3
20131231
16667.9
Freq: ADEC,
dtype: float64
```

pct_change

Growth rates are computed using `pct_change`. The keyword argument periods constructs overlapping growth rates which are useful when using seasonal data.

```
>>> gdp.pct\_change().tail()
DATE
20120401
0.007406
20120701
0.012104
20121001
0.003931
20130101
0.007004
20130401
```

```
0.008019
Name: VALUE, dtype: float64
>>> gdp.pct\_change(periods=4).tail() # Quarterly data, annual difference
DATE
20120401
0.045176
20120701
0.047669
20121001
0.038031
20130101
0.030776
20130401
0.031404
Name: VALUE, dtype: float64
```

2.3 Importing and Exporting Data

In addition to providing data management tools, pandas also excels at importing and exporting data. pandas supports reading and Excel, csv and other delimited files, Stata files, fixed-width text, html, json, HDF5 and from SQL databases. The functions to read follow the common naming convention `read_type` where type is the file type, e.g. excel or csv. The writers are all methods of Series or DataFrame and follow the naming convention `to_type`.

2.3.1 Reading Data

read_excel
`read_excel` supports reading data fromboth xls (Excel 2003) and xlsx (Excel 2007/10/13) formats. The basic usage required two inputs, the file name and the sheet name. Other notable keyword arguments include:

- `header`, an integer indicating which row to use for the column labels. The default is 0 (top) row, and if skiprows is used, this value is relative.
- `skiprows`, typically an integer indicating the number of rows at the top of the sheet to skip before reading the file. The default is 0.
- `skip_footer`, typically an integer indicating the number of rows at the bottom of the sheet to skip when reading the file. The default is 0.
- `index_col`, an integer or column name indicating the column to use as the index. If not provided, a basic numeric index is generated.

- parse_cols, None, an integer, a list of integers or strings, tells pandas whether to attempt to parse a column. The default is `None` which will parse all columns. Alternatively, if an integer is provided then the value is interpreted as the last column to parse.
- Finally, if a list of integers is provided, the values are interpreted as the columns to parse (0-based, e.g. `[0,2,5]`).
- The string version takes one of the forms 'A', 'A,C,D', 'A:D' or a mix of the latter two ('A,C:D,G,W:Z').

read_csv
`read_csv` reads comma separated value files. The basic use only requires one input, a file name. `read_csv` also accepts valid URLs (http, ftp, or s3 (Amazon) if the boto package is available) or any object that provides a read method in places of the file name. A huge range of options are available, and so only the most relevant are presented in the list below.

- delimiter, the delimiter used to separate values. The default is ','. Complicated delimiters are matched using a regular expression.
- delim_whitespace, Boolean indicating that the delimiter is white space (a space or tab). This is preferred to using a regular expression to detect white space.
- header, an integer indicating the row number to use for the column names. The default is 0.
- skiprows, similar to skiprows in read_excel.
- skip_footer, similar to skip_footer in read_excel.
- index_col, similar to index_col in read_excel.
- names, a list of column names to use in-place of any found in the fileMust use header=0 (the default value).
- parse_dates, either a Boolean indicating whether to parse dates encountered, or a list of integers or strings indicating which columns to parse. Supportsmore complicated options to combine columns.
- date_parser, a function to use when parsing dates. The default parser is dateutil.parser.
- dayfirst, a Boolean indicating whether to use European date format (DD/MM, True) or American date format (MM/DD False) when encountering dates. The default is False.
- error_bad_lines, when True stops processing on a bad line. If False, continues skipping any bad lines encountered.

- encoding, a string containing the file encoding (e.g. 'utf8' or 'latin1').
- converters, a dictionary of functions for converting values in certain columns, where keys can either integers (column-number) or column labels.
- nrows, an integer, indicates the maximum number of rows to read. This is useful for reading a subset of a file.
- usecols, a list of integers or column names indicating which column to retain
- dtype A data type to use for the read data or a dictionary of data types using the column names as keys. If not provided, the type is inferred.

read_table
read_table is similar to read_csv and both are wrappers around a private read function provided by pandas. **read_hdf**
read_hdf is primarily for reading pandas DataTables which were written using DataTable.`to_hdf`

2.3.2 Writing Data

Writing data froma Series or DataFrame is muc simpler since the starting point (the Series or theDataFrame) is well understood by pandas. While the file writing methods all have a number of options, most can safely be ignored.

```
>>> state\_gdp.to\_excel('state\_gdp\_from\_dataframe.xls')
>>> state\_gdp.to\_excel('state\_gdp\_from\_dataframe\_sheetname.xls', sheet\_name='Sta
>>> state\_gdp.to\_excel('state\_gdp\_from\_dataframe.xlsx')
>>> state\_gdp.to\_csv('state\_gdp\_from\_dataframe.csv')
>>> import cStringIO
>>> sio = cStringIO.StringIO()
>>> state\_gdp.to\_json(sio)
>>> sio.seek(0)
>>> sio.read(50)
'{"state\_code":{"0":"AK","1":"AL","2":"AR","3":"AZ"'
>>> state\_gdp.to\_string()[:50]
u' state\_code state gdp\_2009 gdp'
```

One writer, `to_hdf` is worth special mention. to_hdf writes pandasDataFrames toHDF5files which are binary files which support compression. HDF5 files can achieve fantastic compression ratios when data are regular, and so are often much more useful than csv or xlsx (which is also compressed). The usage of to_hdf is not meaningfully different from the other writers except that:

- In addition to the filename, an argument is required containing the key, which is usually the variable name.
- Two additional arguments must be passed for the output file to be compressed. These two keyword arguments are complib and complevel, which I recommend to setting to 'zlib' and 6, respectively.

```
>>> df = DataFrame(zeros((1000,1000)))
>>> df.to\_csv('size\_test.csv')
>>> df.to\_hdf('size\_test.h5','df') # h5 is the usual extension for HDF5
# h5 is the usual extension for HDF5
>>> df.to\_hdf('size\_test\_compressed.h5','df',complib='zlib',complevel=6)
>>> ls size\_* # Ignore
09/19/2013 04:16 PM 4,008,782 size\_test.csv
198
09/19/2013 04:16 PM 8,029,160 size\_test.h5
09/19/2013 04:16 PM 33,812 size\_test\_compressed.h5
>>> import gzip
>>> f = gzip.open('size\_test.csvz','w')
>>> df.to\_csv(f)
>>> f.close()
>>> ls size\_test.csvz # Ignore
09/19/2013 04:18 PM 10,533 size\_test.csvz
>>> from pandas import read\_csv
>>> df\_from\_csvz = read\_csv('size\_test.csvz',compression='gzip')
```

The final block of lines shows how a csv with gzip compression is written and directly read using pandas. This method also achieves a very high level of compression. Any NumPy array is easily written to a file using a single, simple line using pandas.

```
>>> x = randn(100,100)
>>> DataFrame(x).to\_csv('numpy\_array.csv',header=False,index=False)
```

2.3.3 HDFStore

`HDFStore` is the Class that underlies `to_hdf`, and is useful for storing multiple Series or DataFrames to a single HDF file. It's use is similar to that of any generic file writing function in Python – it must be opened, data can be read or written, and then it must be closed. Storing data is as simple as inserting objects into a dictionary. The basic use of a HDFStore for saving data is

```
>>> from pandas import HDFStore
>>> store = HDFStore('store.h5',mode='w',complib='zlib',complevel=6)
```

which opens the HDFStore named store for writing (mode='w') with compression. Stores can also be opened for reading (mode='r') or appending (mode='a'). When opened for reading, the compression options are not needed. Data can then be stored in the HDFStore using dictionary syntax.

```
>>> store['a'] = DataFrame([[1,2],[3,4]])
>>> store['b'] = DataFrame(np.ones((10,10)))
and finally the store must be closed.
>>> store.close()
```

The data can then be read using similar commands,

```
>>> store = HDFStore('store.h5',mode='r')
>>> a = store['a']
>>> b = store['b']
>>> store.close()
```

which will read the data with key 'a' in a variable named a, and similarly for b. A slightly better method for using a store is to use the Python keyword with and `get_store`. This is similar to opening the store, reading/writing some data, and then calling close(), only that the close() is automatically called when the store is no longer required. For example, with pd.get_store('store.h5') as store:

```
a = store['a']
b = store['b']
```

is equivalent to the previous code block, only the `close()` is called implicitly after the variables are read. `get_store` can be used with the same keyword arguments as HDFStore to enable compression or set the mode for opening the file.

2.4 Graphics

pandas provides a set of useful plotting routines based on matplotlib which makes use of the structure of a DataFrame. Everything in *pandas* plot library is reproducible using matplotlib, although often at the cost of additional typing and code complexity (for example, axis labeling).

plot

`plot` is the main plotting method, and by default will produce a line graph of the data in a DataFrame. Calling `plot` on a DataFrame will plot all series using different colours and generate a legend. A number of keyword argument are available to affect the contents and appearance of the plot.

- `style`, a list of matplotlib styles, one for each series plotted. A dictionary using column names as keys and the line styles as values allows for further customization.
- `title`, a string containing the figure title.
- subplots, a Boolean indicating whether to plot using one subplot per series (True). The default it False.
- legend, a Boolean indicating whether to show a legend
- `secondary_y`, a Boolean indicating whether to plot a series on a secondary set of axis values. See the example below.
- `ax`, a matplotlib axis object to use for the plot. If no axis is provided, then a new axis is created.
- kind, a string, one of:
  - '`line`', the default
  - '`bar`' to produce a bar chart. Can also use the keyword argument stacked=True to produce a stacked bar chart.
  - '`barh`' to produce a horizontal bar chart. Also support stacked=True.
  - '`kde`' or '`density`' to produce a kernel density plot.

2.4.1 `hist`

`hist` produces a histogram plot, and is similar to producing a bar plot using the output of `value_count`.

2.4.2 boxplot

`boxplot` produces box plots of the series in a DataFrame.

2.4.3 scatter_plot

`scatter_plot` produce a scatter plot from two series in a DataFrame. Three inputs are required: the DataFrame, the column name for the x-axis data and the column name for the y-axis data. `scatter_plot` is located in `pandas.tools.plotting`.

2.4.4 scatter_matrix

`scatter_matrix` produces a *n by n* set of subplots where each subplot contains the bivariate scatter of two series. One input is required, the DataFrame. `scatter_matrix` is located in `pandas.tools.plotting`. By default, the diagonal elements are histograms, and the keyword argument `diagonal='kde'` produces a ***kernel density plot***.

2.4.5 lag_plot

`lag_plot` produces a scatter plot of a series against its lagged value. The keyword argument `lag` chooses the lag used in the plot (default is 1). This is quite useful in ***Time Series Analysis***.