Getting Started with pandas

Part 2

Introduction to pandas Data Structures

Part 2

DataFrame

- A DataFrame represents a rectangular table of data and contains an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.).
- The DataFrame has both a row and column index; it can be thought of as a dict of Series all sharing the same index.
- Under the hood, the data is stored as one or more two-dimensional blocks rather than a list, dict, or some other collection of one-dimensional arrays.

- There are many ways to construct a DataFrame, though one of the most common is from a dict of equal-length lists or NumPy arrays:

```
In [34]: data = {'state': ['Ohio', 'Ohio', 'Ohio', 'Nevada', 'Nevada', 'Nevada'],
                 'year': [2000, 2001, 2002, 2001, 2002, 2003],
                 'pop': [1.5, 1.7, 3.6, 2.4, 2.9, 3.2]}
         frame = pd.DataFrame(data)
```

- The resulting DataFrame will have its index assigned automatically as with Series, and the columns are placed in sorted order. (*Changed in version 0.23.0:* If data is a dict, argument order is maintained for Python 3.6 and later.)

```
In [35]: frame
```

Out[35]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9
5	Nevada	2003	3.2

- For large DataFrames, the `head` method selects only the first five rows:

```
In [36]: frame.head()
```

Out[36]:

	state	year	pop
0	Ohio	2000	1.5
1	Ohio	2001	1.7
2	Ohio	2002	3.6
3	Nevada	2001	2.4
4	Nevada	2002	2.9

- If you specify a sequence of columns, the DataFrame's columns will be arranged in that order:

```
In [37]: pd.DataFrame(data, columns=['year', 'state', 'pop'])
```

Out[37]:

	year	state	pop
0	2000	Ohio	1.5
1	2001	Ohio	1.7
2	2002	Ohio	3.6
3	2001	Nevada	2.4
4	2002	Nevada	2.9
5	2003	Nevada	3.2

- If you pass a column that isn't contained in the dict, it will appear with missing values in the result:

```
In [38]: frame2 = pd.DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
                               index=['one', 'two', 'three', 'four',
                                      'five', 'six'])
         frame2
```

Out[38]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	NaN
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	NaN
five	2002	Nevada	2.9	NaN
six	2003	Nevada	3.2	NaN

```
In [39]: frame2.columns
Out[39]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

- A column in a DataFrame can be retrieved as a Series either by dict-like notation or by attribute:

```
In [40]: frame2['state']

Out[40]: one        Ohio
         two        Ohio
         three      Ohio
         four     Nevada
         five     Nevada
         six      Nevada
         Name: state, dtype: object

In [41]: frame2.year

Out[41]: one      2000
         two      2001
         three    2002
         four     2001
         five     2002
         six      2003
         Name: year, dtype: int64
```

- Note that the returned Series have the same index as the DataFrame, and their `name` attribute has been appropriately set.

- Rows can also be retrieved by position or name with the special `loc` attribute:

```
In [42]: frame2.loc['three']

Out[42]: year     2002
         state    Ohio
         pop       3.6
         debt      NaN
         Name: three, dtype: object
```

- Columns can be modified by assignment.
- For example, the empty `'debt'` column could be assigned a scalar value or an array of values:

```
In [43]: frame2['debt'] = 16.5
         frame2
```

Out[43]:

	year	state	pop	debt
one	2000	Ohio	1.5	16.5
two	2001	Ohio	1.7	16.5
three	2002	Ohio	3.6	16.5
four	2001	Nevada	2.4	16.5
five	2002	Nevada	2.9	16.5
six	2003	Nevada	3.2	16.5

```
In [44]: frame2['debt'] = np.arange(6.)
         frame2
```

Out[44]:

	year	state	pop	debt
one	2000	Ohio	1.5	0.0
two	2001	Ohio	1.7	1.0
three	2002	Ohio	3.6	2.0
four	2001	Nevada	2.4	3.0
five	2002	Nevada	2.9	4.0
six	2003	Nevada	3.2	5.0

- When you are assigning lists or arrays to a column, the value's length must match the length of the DataFrame.
- If you assign a Series, its labels will be realigned exactly to the DataFrame's index, inserting missing values in any holes:

```
In [45]: val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
         frame2['debt'] = val
         frame2
```

Out[45]:

	year	state	pop	debt
one	2000	Ohio	1.5	NaN
two	2001	Ohio	1.7	-1.2
three	2002	Ohio	3.6	NaN
four	2001	Nevada	2.4	-1.5
five	2002	Nevada	2.9	-1.7
six	2003	Nevada	3.2	NaN

- Assigning a column that doesn’t exist will create a new column.
- The `del` keyword will delete columns as with a dict.

- As an example of `del`, I first add a new column of boolean values where the state column equals `'Ohio'`:

```
In [46]: frame2['eastern'] = frame2.state == 'Ohio'
         frame2
```

Out[46]:

	year	state	pop	debt	eastern
one	2000	Ohio	1.5	NaN	True
two	2001	Ohio	1.7	-1.2	True
three	2002	Ohio	3.6	NaN	True
four	2001	Nevada	2.4	-1.5	False
five	2002	Nevada	2.9	-1.7	False
six	2003	Nevada	3.2	NaN	False

- The `del` method can then be used to remove this column:

```
In [47]: del frame2['eastern']
         frame2.columns

Out[47]: Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

- Another common form of data is a nested dict of dicts:

```
In [48]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
                'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

- If the nested dict is passed to the DataFrame, pandas will interpret the outer dict keys as the columns and the inner keys as the row indices:

```
In [49]: frame3 = pd.DataFrame(pop)
         frame3
```

Out[49]:

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

- You can transpose the DataFrame (swap rows and columns) with similar syntax to a NumPy array:

```
In [50]: frame3.T
```

Out[50]:

	2000	2001	2002
Nevada	NaN	2.4	2.9
Ohio	1.5	1.7	3.6

- The keys in the inner dicts are combined and sorted to form the index in the result.
- This isn’t true if an explicit index is specified:

```
In [47]: pop
Out[47]: {'Nevada': {2001: 2.4, 2002: 2.9}, 'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}

In [48]: pd.DataFrame(pop, index=[2001, 2002, 2003])
Out[48]:
```

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2003	NaN	NaN

- Dicts of Series are treated in much the same way:

```
In [49]: frame3
```

Out[49]:

	Nevada	Ohio
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

```
In [50]: pdata = {'Ohio': frame3['Ohio'][:-1],
                  'Nevada': frame3['Nevada'][:2]}
         pdata
```

```
Out[50]: {'Ohio': 2000    1.5
          2001    1.7
          Name: Ohio, dtype: float64, 'Nevada': 2000    NaN
          2001    2.4
          Name: Nevada, dtype: float64}
```

```
In [51]: pd.DataFrame(pdata)
```

Out[51]:

	Ohio	Nevada
2000	1.5	NaN
2001	1.7	2.4

- If a DataFrame's `index` and `columns` have their `name` attributes set, these will also be displayed:

```
In [52]: frame3.index.name = 'year'; frame3.columns.name = 'state'
         frame3
```

Out[52]:

state	Nevada	Ohio
year		
2000	NaN	1.5
2001	2.4	1.7
2002	2.9	3.6

- As with Series, the `values` attribute returns the data contained in the DataFrame as a two-dimensional ndarray:

```
In [53]: frame3.values

Out[53]: array([[nan, 1.5],
                [2.4, 1.7],
                [2.9, 3.6]])
```

- If the DataFrame’s columns are different dtypes, the dtype of the values array will be chosen to accommodate all of the columns:

```
In [54]: frame2.values

Out[54]: array([[2000, 'Ohio', 1.5, nan],
                [2001, 'Ohio', 1.7, -1.2],
                [2002, 'Ohio', 3.6, nan],
                [2001, 'Nevada', 2.4, -1.5],
                [2002, 'Nevada', 2.9, -1.7],
                [2003, 'Nevada', 3.2, nan]], dtype=object)
```