It contains data structures and data manipulation tools designed to make data cleaning and analysis fast and easy in Python.

pandas is often used in tandem with numerical computing tools like NumPy and SciPy, analytical libraries like statsmodels and scikit-learn, and data visualization libraries like matplotlib. pandas adopts significant parts of NumPy's idiomatic style of array-based computing, especially array-based functions and a preference for data processing without for loops.

While pandas adopts many coding idioms from NumPy, the biggest difference is that pandas is designed for working with tabular or heterogeneous data. NumPy, by contrast, is best suited for working with homogeneous numerical array data.

Since becoming an open source project in 2010, pandas has matured into a quite large library that's applicable in a broad set of real-world use cases. The developer community has grown to over 800 distinct contributors, who've been helping build the project as they've used it to solve their day-to-day data problems.

In [1]:
```
import pandas as pd
```

You may also find it eas-ier to import Series and DataFrame into the local namespace since they are so frequently used:

In [2]:
```
from pandas import Series, DataFrame
```

To get started with pandas, you will need to get comfortable with its two workhorse data structures: Series and DataFrame. While they are not a universal solution for every problem, they provide a solid, easy-to-use basis for most applications.

Series

A Series is a one-dimensional array-like object containing a sequence of values (of similar types to NumPy types) and an associated array of data labels, called its index. The simplest Series is formed from only an array of data:

In [3]:
```
obj = pd.Series([4, 7, -5, 3])
obj
```

Out[3]:
```
0    4
1    7
2   -5
3    3
dtype: int64
```

The string representation of a Series displayed interactively shows the index on the left and the values on the right. Since we did not specify an index for the data, a default one consisting of the integers 0 through N - 1 (where N is the length of the data) is created. You can get the array representation and index object of the Series via its values and index attributes, respectively:

In [4]:
```
obj.values
```

Out[4]:
```
array([ 4,  7, -5,  3], dtype=int64)
```

In [5]:
```
obj.index # like range(4)
```

Out[5]:
```
RangeIndex(start=0, stop=4, step=1)
```

Often it will be desirable to create a Series with an index identifying each data point with a label:

In [6]:
```
obj2 = pd.Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])
obj2
```

Out[6]:
```
d    4
b    7
a   -5
c    3
dtype: int64
```

In [7]:
```
obj2.index
```

Out[7]:
```
Index(['d', 'b', 'a', 'c'], dtype='object')
```

Compared with NumPy arrays, you can use labels in the index when selecting single values or a set of values:

In [8]:
```
 obj2['a']
```

Out[8]:
```
-5
```

In [9]:
```
obj2['d'] = 6
```

In [10]:
```
obj2
```

Out[10]:
```
d    6
b    7
a   -5
c    3
dtype: int64
```

In [11]:
```
obj2[['c', 'a', 'd']]
```

Out[11]:
```
c    3
a   -5
d    6
dtype: int64
```

Here ['c', 'a', 'd'] is interpreted as a list of indices, even though it contains strings instead of integers.

Using NumPy functions or NumPy-like operations, such as filtering with a boolean array, scalar multiplication, or applying math functions, will preserve the index-value link:

In [12]:
```
obj2[obj2 > 0]
```

Out[12]:
```
d    6
b    7
c    3
dtype: int64
```

In [13]:
```
obj2 * 2
```

Out[13]:
```
d    12
b    14
a   -10
c     6
dtype: int64
```

In [15]:
```
import numpy as np
np.exp(obj2)
```

Out[15]:
```
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
dtype: float64
```

Another way to think about a Series is as a fixed-length, ordered dict, as it is a mapping of index values to data values. It can be used in many contexts where you might use a dict:

In [16]:
```
'b' in obj2
```

Out[16]:
```
True
```

In [17]:
```
'e' in obj2
```

Out[17]:
```
False
```

Should you have data contained in a Python dict, you can create a Series from it by passing the dict:

In [18]:
```
sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
obj3 = pd.Series(sdata)
obj3
```

Out[18]:
```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

When you are only passing a dict, the index in the resulting Series will have the dict's keys in sorted order. You can override this by passing the dict keys in the order you want them to appear in the resulting Series:

In [19]:
```
states = ['California', 'Ohio', 'Oregon', 'Texas']
```

In [20]:
```
obj4 = pd.Series(sdata, index=states)
```

In [21]:
```
obj4
```

Out[21]:
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

Here, three values found in sdata were placed in the appropriate locations, but since no value for 'California' was found, it appears as NaN (not a number), which is considered in pandas to mark missing or NA values. Since 'Utah' was not included in states, it is excluded from the resulting object.

I will use the terms “missing” or “NA” interchangeably to refer to missing data. The isnull and notnull functions in pandas should be used to detect missing data:

In [22]:
```
pd.isnull(obj4)
```

Out[22]:
```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

In [23]:
```
pd.notnull(obj4)
```

Out[23]:
```
California    False
Ohio           True
Oregon         True
Texas          True
dtype: bool
```

Series also has these as instance methods:

In [24]:
```
obj4.isnull()
```

Out[24]:
```
California     True
Ohio          False
Oregon        False
Texas         False
dtype: bool
```

A useful Series feature for many applications is that it automatically aligns by index label in arithmetic operations:

In [25]:
```
obj3
```

Out[25]:
```
Ohio      35000
Texas     71000
Oregon    16000
Utah       5000
dtype: int64
```

In [26]:
```
obj4
```

Out[26]:
```
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
dtype: float64
```

In [27]:
```
obj3 + obj4
```

Out[27]:
```
California         NaN
Ohio           70000.0
Oregon         32000.0
Texas         142000.0
Utah               NaN
dtype: float64
```

Data alignment features will be addressed in more detail later. If you have experience with databases, you can think about this as being similar to a join operation.

Both the Series object itself and its index have a name attribute, which integrates with other key areas of pandas functionality:

In [28]:
```
obj4.name = 'population'
obj4.index.name = 'state'
```

In [29]:
```
obj4
```

Out[29]:
```
state
California        NaN
Ohio          35000.0
Oregon        16000.0
Texas         71000.0
Name: population, dtype: float64
```

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

In [30]:
```
obj
```

Out[30]:
```
0    4
1    7
2   -5
3    3
dtype: int64
```

In [31]:
```
obj.index = ['Bob', 'Steve', 'Jeff', 'Ryan']
```

In [32]:
```
obj
```

Out[32]:
```
Bob      4
Steve    7
Jeff    -5
Ryan     3
dtype: int64
```

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.

While a DataFrame is physically two-dimensional, you can use it to represent higher dimensional data in a tabular format using hierarchical indexing, a subject we will discuss in Chapter 8 and an ingredient in some of the more advanced data-handling features in pandas.

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 [33]:
```
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:

In [34]:
```
frame
```

Out[34]:

	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

If you are using the Jupyter notebook, pandas DataFrame objects will be displayed as a more browser-friendly HTML table.

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

In [35]:
```
frame.head()
```

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

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

In [36]:
```
pd.DataFrame(data, columns=['year', 'state', 'pop'])
```

Out[36]:

	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',
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
```

Attribute-like access (e.g., frame2.year) and tab completion of column names in IPython is provided as a convenience. frame2[column] works for any column name, but frame2.column only works when the column name is a valid Python variable name

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 (much more on this later):

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
```

In [44]:
```
frame2
```

Out[44]:

	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 [45]:
```
frame2['debt'] = np.arange(6.)
```

In [46]:
```
frame2
```

Out[46]:

	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 [48]:
```
val = pd.Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])
```

In [49]:
```
frame2['debt'] = val
```

In [50]:
```
frame2
```

Out[50]:

	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 [51]:
```
frame2['eastern'] = frame2.state == 'Ohio'
```

In [52]:
```
frame2
```

Out[52]:

	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

New columns cannot be created with the frame2.eastern syntax

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

In [53]:
```
del frame2['eastern']
```

In [54]:
```
frame2.columns
```

Out[54]:
```
Index(['year', 'state', 'pop', 'debt'], dtype='object')
```

The column returned from indexing a DataFrame is a view on the underlying data, not a copy. Thus, any in-place modifications to the Series will be reflected in the DataFrame. The column can be explicitly copied with the Series's copy method.

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

In [55]:
```
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 [56]:
```
frame3 = pd.DataFrame(pop)
```

In [57]:
```
frame3
```

Out[57]:

	Nevada	Ohio
2001	2.4	1.7
2002	2.9	3.6
2000	NaN	1.5

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

In [58]:
```
frame3.T
```

Out[58]:

	2001	2002	2000
Nevada	2.4	2.9	NaN
Ohio	1.7	3.6	1.5

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 [59]:
```
pd.DataFrame(pop, index=[2001, 2002, 2003])
```

Out[59]:

	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 [60]:
```
pdata = {'Ohio': frame3['Ohio'][:-1],'Nevada': frame3['Nevada'][:2]}
```

In [61]:
```
pd.DataFrame(pdata)
```

Out[61]:

	Ohio	Nevada
2001	1.7	2.4
2002	3.6	2.9

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

In [62]:
```
frame3.index.name = 'year'; frame3.columns.name = 'state'
```

In [63]:
```
frame3
```

Out[63]:

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

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

In [64]:
```
frame3.values
```

Out[64]:
```
array([[2.4, 1.7],
       [2.9, 3.6],
       [nan, 1.5]])
```

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

In [65]:
```
frame2.values
```

Out[65]:
```
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)
```

In [66]:
```
# Table 5-1. --> Possible data inputs to DataFrame constructor
# ---------------------------------------------------------------
# Type --> Notes
# ----------------
```

```
# 2D ndarray --> A matrix of data, passing optional row and column labels
# dict of arrays, lists, or tuples --> Each sequence becomes a column in the DataFr
# NumPy structured/record array --> Treated as the "dict of arrays" case
# dict of Series --> Each value becomes a column; indexes from each Series are unio
# result's row index if no explicit index is passed
# dict of dicts --> Each inner dict becomes a column; keys are unioned to form the
# Series" case
# List of dicts or Series --> Each item becomes a row in the DataFrame; union of di
# DataFrame's column labels
# List of lists or tuples --> Treated as the "2D ndarray" case
# Another DataFrame --> The DataFrame's indexes are used unless different ones are
# NumPy MaskedArray --> Like the "2D ndarray" case except masked values become NA/m
```

Index Objects

pandas's Index objects are responsible for holding the axis labels and other metadata (like the axis name or names). Any array or other sequence of labels you use when constructing a Series or DataFrame is internally converted to an Index:

In [67]:
```
obj = pd.Series(range(3), index=['a', 'b', 'c'])
```

In [68]:
```
index = obj.index
```

In [69]:
```
index
```

Out[69]:
```
Index(['a', 'b', 'c'], dtype='object')
```

In [70]:
```
index[1:]
```

Out[70]:
```
Index(['b', 'c'], dtype='object')
```

Index objects are immutable and thus can't be modified by the user:

In [71]:
```
index[1] = 'd' # TypeError
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-71-1a15f4fda8a8> in <module>
----> 1 index[1] = 'd' # TypeError

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\indexes\base.py in __
setitem__(self, key, value)
   4082
   4083     def __setitem__(self, key, value):
-> 4084         raise TypeError("Index does not support mutable operations")
   4085
   4086     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

Immutability makes it safer to share Index objects among data structures:

In [72]:
```
labels = pd.Index(np.arange(3))
```

In [74]:
```
labels
```

Out[74]:
```
Int64Index([0, 1, 2], dtype='int64')
```

In [75]:
```
obj2 = pd.Series([1.5, -2.5, 0], index=labels)
```

In [76]:
```
obj2
```

Out[76]:
```
0    1.5
1   -2.5
2    0.0
dtype: float64
```

In [77]:
```
obj2.index is labels
```

Out[77]:
```
True
```

Some users will not often take advantage of the capabilities provided by indexes, but because some operations will yield results containing indexed data, it's important to understand how they work

In addition to being array-like, an Index also behaves like a fixed-size set:

In [78]:
```
frame3
```

Out[78]:

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

In [79]:
```
 frame3.columns
```

Out[79]:
```
Index(['Nevada', 'Ohio'], dtype='object', name='state')
```

In [80]:
```
'Ohio' in frame3.columns
```

Out[80]:
```
True
```

In [81]:
```
2003 in frame3.index
```

Out[81]:
```
False
```

Unlike Python sets, a pandas Index can contain duplicate labels:

In [82]:
```
dup_labels = pd.Index(['foo', 'foo', 'bar', 'bar'])
```

In [83]:
```
dup_labels
```

Out[83]:
```
Index(['foo', 'foo', 'bar', 'bar'], dtype='object')
```

Each Index has a number of methods and properties for set logic, which answer other common questions about the data it contains. Some useful ones are summarized in Table 5-2

In [84]:
```
# Table 5-2. --> Some Index methods and properties
# ----------------------------------------------------
# Method --> Description
# -----------------------
# append --> Concatenate with additional Index objects, producing a new Index
# difference --> Compute set difference as an Index
# intersection --> Compute set intersection
# union --> Compute set union
# isin --> Compute boolean array indicating whether each value is contained in the
# delete --> Compute new Index with element at index i deleted
# drop --> Compute new Index by deleting passed values
# insert --> Compute new Index by inserting element at index i
# is_monotonic --> Returns True if each element is greater than or equal to the pre
# is_unique --> Returns True if the Index has no duplicate values
# unique --> Compute the array of unique values in the Index
```

5.2 Essential Functionality

This section will walk you through the fundamental mechanics of interacting with the data contained in a Series or DataFrame

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

In [85]:
```
obj = pd.Series([4.5, 7.2, -5.3, 3.6], index=['d', 'b', 'a', 'c'])
obj
```

Out[85]:
```
d    4.5
b    7.2
a   -5.3
c    3.6
dtype: float64
```

Calling reindex on this Series rearranges the data according to the new index, introducing missing values if any index values were not already present:

In [86]:
```
obj2 = obj.reindex(['a', 'b', 'c', 'd', 'e'])
```

In [87]:
```
obj2
```

Out[87]:
```
a   -5.3
b    7.2
c    3.6
d    4.5
e    NaN
dtype: float64
```

For ordered data like time series, it may be desirable to do some interpolation or filling of values when reindexing. The method option allows us to do this, using a method such as ffill, which forward-fills the values:

In [88]:
```
obj3 = pd.Series(['blue', 'purple', 'yellow'], index=[0, 2, 4])
obj3
```

Out[88]:
```
0      blue
2    purple
4    yellow
dtype: object
```

In [89]:
```
obj3.reindex(range(6), method='ffill')
```

Out[89]:
```
0      blue
1      blue
2    purple
3    purple
4    yellow
5    yellow
dtype: object
```

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

In [90]:
```
frame = pd.DataFrame(np.arange(9).reshape((3, 3)),index=['a', 'c', 'd'],columns=['O
```

In [91]:
```
frame
```

Out[91]:

	Ohio	Texas	California
a	0	1	2
c	3	4	5
d	6	7	8

In [92]:
```
frame2 = frame.reindex(['a', 'b', 'c', 'd'])
frame2
```

Out[92]:

	Ohio	Texas	California
a	0.0	1.0	2.0
b	NaN	NaN	NaN
c	3.0	4.0	5.0
d	6.0	7.0	8.0

The columns can be reindexed with the columns keyword:

In [93]:
```
states = ['Texas', 'Utah', 'California']
frame.reindex(columns=states)
```

Out[93]:

	Texas	Utah	California
a	1	NaN	2
c	4	NaN	5
d	7	NaN	8

As we'll explore in more detail, you can reindex more succinctly by label-indexing with loc, and many users prefer to use it exclusively:

In [94]:
```
frame.loc[['a', 'b', 'c', 'd'], states]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-94-074c26e96084> in <module>
----> 1 frame.loc[['a', 'b', 'c', 'd'], states]

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\indexing.py in __geti
tem__(self, key)
    871                 # AttributeError for IntervalTree get_value
    872                 pass
--> 873         return self._getitem_tuple(key)
    874     else:
    875         # we by definition only have the 0th axis

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\indexing.py in _getit
em_tuple(self, tup)
   1051         # ugly hack for GH #836
   1052         if self._multi_take_opportunity(tup):
-> 1053             return self._multi_take(tup)
   1054 
   1055         return self._getitem_tuple_same_dim(tup)

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\indexing.py in _multi
_take(self, tup)
   1003         d = {
   1004             axis: self._get_listlike_indexer(key, axis)
-> 1005             for (key, axis) in zip(tup, self.obj._AXIS_ORDERS)
   1006         }
   1007         return self.obj._reindex_with_indexers(d, copy=True, allow_dups=Tru
e)

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\indexing.py in <dictc
omp>(.0)
   1003         d = {
   1004             axis: self._get_listlike_indexer(key, axis)
-> 1005             for (key, axis) in zip(tup, self.obj._AXIS_ORDERS)
   1006         }
   1007         return self.obj._reindex_with_indexers(d, copy=True, allow_dups=Tru
e)

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\indexing.py in _get_l
istlike_indexer(self, key, axis, raise_missing)
   1252             keyarr, indexer, new_indexer = ax._reindex_non_unique(keyarr)
   1253 
-> 1254         self._validate_read_indexer(keyarr, indexer, axis, raise_missing=rai
se_missing)
   1255         return keyarr, indexer
   1256 

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\indexing.py in _valid
ate_read_indexer(self, key, indexer, axis, raise_missing)
```

```
   1314                 with option_context("display.max_seq_items", 10, "display.wi
dth", 80):
   1315                     raise KeyError(
-> 1316                         "Passing list-likes to .loc or [] with any missing l
abels "
   1317                         "is no longer supported. "
   1318                         f"The following labels were missing: {not_found}. "

KeyError: "Passing list-likes to .loc or [] with any missing labels is no longer sup
ported. The following labels were missing: Index(['b'], dtype='object'). See http
s://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#deprecate-loc-rein
dex-listlike"
```

In [95]:
```
# Table 5-3. --> reindex function arguments
# ------------------------------------------
# Argument --> Description
# ------------------------
# index --> New sequence to use as index. Can be Index instance or any other sequen
# Index will be used exactly as is without any copying.
# method --> Interpolation (fill) method; 'ffill' fills forward, while 'bfill' fill
# fill_value --> Substitute value to use when introducing missing data by reindexin
# limit --> When forward- or backfilling, maximum size gap (in number of elements)
# tolerance --> When forward- or backfilling, maximum size gap (in absolute numeric
# level --> Match simple Index on level of MultiIndex; otherwise select subset of.
# copy --> If True, always copy underlying data even if new index is equivalent to
# the data when the indexes are equivalent.
```

Dropping Entries from an Axis

Dropping one or more entries from an axis is easy if you already have an index array or list without those entries. As that can require a bit of munging and set logic, the drop method will return a new object with the indicated value or values deleted from an axis:

In [96]:
```
obj = pd.Series(np.arange(5.), index=['a', 'b', 'c', 'd', 'e'])
```

In [97]:
```
obj
```

Out[97]:
```
a    0.0
b    1.0
c    2.0
d    3.0
e    4.0
dtype: float64
```

In [98]:
```
new_obj = obj.drop('c')
```

In [99]:
```
new_obj
```

Out[99]:
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

In [100…
```
obj.drop(['d', 'c'])
```

Out[100…
```
a    0.0
b    1.0
e    4.0
dtype: float64
```

With DataFrame, index values can be deleted from either axis. To illustrate this, we first create an example DataFrame:

In [101…
```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),index=['Ohio', 'Colorado', 'Utah'
```

In [102…
```
data
```

Out[102…

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

Calling drop with a sequence of labels will drop values from the row labels (axis 0):

In [103…
```
data.drop(['Colorado', 'Ohio'])
```

Out[103…

	one	two	three	four
Utah	8	9	10	11
New York	12	13	14	15

You can drop values from the columns by passing axis=1 or axis='columns':

In [104…
```
data.drop('two', axis=1)
```

Out[104…

	one	three	four
Ohio	0	2	3
Colorado	4	6	7
Utah	8	10	11
New York	12	14	15

In [105…
```
data.drop(['two', 'four'], axis='columns')
```

Out[105…

	one	three
Ohio	0	2
Colorado	4	6
Utah	8	10
New York	12	14

Many functions, like drop, which modify the size or shape of a Series or DataFrame, can manipulate an object in-place without returning a new object:

In [106…
```
obj.drop('c', inplace=True)
obj
```

Out[106…
```
a    0.0
b    1.0
d    3.0
e    4.0
dtype: float64
```

Be careful with the inplace, as it destroys any data that is dropped.

Indexing, Selection, and Filtering

Series indexing (obj[...]) works analogously to NumPy array indexing, except you can use the Series's index values instead of only integers. Here are some examples of this:

In [108…
```
obj = pd.Series(np.arange(4.), index=['a', 'b', 'c', 'd'])
```

In [109…
```
obj
```

Out[109…
```
a    0.0
b    1.0
c    2.0
d    3.0
dtype: float64
```

In [110…
```
obj['b']
```

Out[110…
```
1.0
```

In [111…
```
obj[1]
```

Out[111…
```
1.0
```

In [112…
```
obj[2:4]
```

Out[112…
```
c    2.0
d    3.0
dtype: float64
```

In [113…
```
obj[['b', 'a', 'd']]
```

Out[113…
```
b    1.0
a    0.0
d    3.0
dtype: float64
```

In [114…
```
obj[[1, 3]]
```

Out[114…
```
b    1.0
d    3.0
dtype: float64
```

In [115…
```
obj[obj < 2]
```

Out[115…
```
a    0.0
b    1.0
dtype: float64
```

Slicing with labels behaves differently than normal Python slicing in that the end- point is inclusive:

In [116…
```
obj['b':'c']
```

Out[116…
```
b    1.0
c    2.0
dtype: float64
```

Setting using these methods modifies the corresponding section of the Series:

In [117…
```
obj['b':'c'] = 5
obj
```

Out[117…
```
a    0.0
b    5.0
c    5.0
d    3.0
dtype: float64
```

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

In [118…
```
data = pd.DataFrame(np.arange(16).reshape((4, 4)),index=['Ohio', 'Colorado', 'Utah'
```

In [119…
```
data
```

Out[119…

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

In [120…

```
data['two']
```

Out[120…

```
Ohio         1
Colorado     5
Utah         9
New York    13
Name: two, dtype: int32
```

In [121…

```
data[['three', 'one']]
```

Out[121…

	three	one
Ohio	2	0
Colorado	6	4
Utah	10	8
New York	14	12

Indexing like this has a few special cases. First, slicing or selecting data with a boolean array:

In [122…

```
data[:2]
```

Out[122…

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7

In [123…

```
data[data['three'] > 5]
```

Out[123…

	one	two	three	four
Colorado	4	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

The row selection syntax data[:2] is provided as a convenience. Passing a single element or a list to the [] operator selects columns.

Another use case is in indexing with a boolean DataFrame, such as one produced by a scalar comparison:

In [124...
```
data < 5
```

Out[124...

	one	two	three	four
Ohio	True	True	True	True
Colorado	True	False	False	False
Utah	False	False	False	False
New York	False	False	False	False

In [125...
```
data[data < 5] = 0
```

In [126...
```
data
```

Out[126...

	one	two	three	four
Ohio	0	0	0	0
Colorado	0	5	6	7
Utah	8	9	10	11
New York	12	13	14	15

This makes DataFrame syntactically more like a two-dimensional NumPy array in this particular case.

Selection with loc and iloc

For DataFrame label-indexing on the rows, I introduce the special indexing operators loc and iloc. They enable you to select a subset of the rows and columns from a DataFrame with NumPy-like notation using either axis labels (loc) or integers (iloc).

In [127...
```
data.loc['Colorado', ['two', 'three']]
```

Out[127...
```
two      5
three    6
Name: Colorado, dtype: int32
```

We'll then perform some similar selections with integers using iloc:

In [128...
```
data.iloc[2, [3, 0, 1]]
```

Out[128…

```
four     11
one       8
two       9
Name: Utah, dtype: int32
```

In [129…

```
data.iloc[2]
```

Out[129…

```
one       8
two       9
three    10
four     11
Name: Utah, dtype: int32
```

In [130…

```
data.iloc[[1, 2], [3, 0, 1]]
```

Out[130…

	four	one	two
Colorado	7	0	5
Utah	11	8	9

Both indexing functions work with slices in addition to single labels or lists of labels:

In [131…

```
data.loc[:'Utah', 'two']
```

Out[131…

```
Ohio        0
Colorado    5
Utah        9
Name: two, dtype: int32
```

In [132…

```
data.iloc[:, :3][data.three > 5]
```

Out[132…

	one	two	three
Colorado	0	5	6
Utah	8	9	10
New York	12	13	14

In [133…

```
# Table 5-4. --> Indexing options with DataFrame
# -----------------------------------------------
# Type --> Notes
# --------------
# df[val] --> Select single column or sequence of columns from the DataFrame; speci
# conveniences: boolean array (filter rows), slice (slice rows), or boolean DataFra
# (set values based on some criterion)
# df.loc[val] --> Selects single row or subset of rows from the DataFrame by label
# df.loc[:, val] --> Selects single column or subset of columns by label
# df.loc[val1, val2] --> Select both rows and columns by label
# df.iloc[where] --> Selects single row or subset of rows from the DataFrame by int
# df.iloc[:, where] --> Selects single column or subset of columns by integer posit
# df.iloc[where_i, where_j] --> Select both rows and columns by integer position
# df.at[label_i, label_j] --> Select a single scalar value by row and column label
```

```
# df.iat[i, j] --> Select a single scalar value by row and column position (integer
# reindex method --> Select either rows or columns by labels
# get_value, set_value methods --> Select single value by row and column label
```

Integer Indexes

Working with pandas objects indexed by integers is something that often trips up new users due to some differences with indexing semantics on built-in Python data structures like lists and tuples. For example, you might not expect the following code to generate an error:

In [134…

```
ser = pd.Series(np.arange(3.))
ser
```

Out[134…

```
0    0.0
1    1.0
2    2.0
dtype: float64
```

In [135…

```
ser[-1]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\indexes\range.py in g
et_loc(self, key, method, tolerance)
    354                 try:
--> 355                     return self._range.index(new_key)
    356                 except ValueError as err:

ValueError: -1 is not in range

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
<ipython-input-135-44969a759c20> in <module>
----> 1 ser[-1]

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\series.py in __getite
m__(self, key)
    880 
    881         elif key_is_scalar:
--> 882             return self._get_value(key)
    883 
    884         if is_hashable(key):

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\series.py in _get_val
ue(self, label, takeable)
    988 
    989         # Similar to Index.get_value, but we do not fall back to positional
--> 990         loc = self.index.get_loc(label)
    991         return self.index._get_values_for_loc(self, loc, label)
    992 

~\OneDrive - Reliance Corporate IT Park Limited\Desktop\Practice_Code\Python_Practic
e\Python_For_Data_Analysis\myenv\lib\site-packages\pandas\core\indexes\range.py in g
et_loc(self, key, method, tolerance)
    355                     return self._range.index(new_key)
    356                 except ValueError as err:
--> 357                     raise KeyError(key) from err
    358             raise KeyError(key)
    359         return super().get_loc(key, method=method, tolerance=tolerance)

KeyError: -1
```

In this case, pandas could “fall back” on integer indexing, but it’s difficult to do this in general without introducing subtle bugs. Here we have an index containing 0, 1, 2, but inferring what the user wants (label-based indexing or position-based) is difficult

On the other hand, with a non-integer index, there is no potential for ambiguity:

In [136…

```
ser2 = pd.Series(np.arange(3.), index=['a', 'b', 'c'])
```

In [137…
```
ser2[-1]
```
Out[137…
```
2.0
```

To keep things consistent, if you have an axis index containing integers, data selection will always be label-oriented. For more precise handling, use loc (for labels) or iloc (for integers):

In [138…
```
ser[:1]
```
Out[138…
```
0    0.0
dtype: float64
```

In [139…
```
ser.loc[:1]
```
Out[139…
```
0    0.0
1    1.0
dtype: float64
```

In [140…
```
ser.iloc[:1]
```
Out[140…
```
0    0.0
dtype: float64
```

Arithmetic and Data Alignment

An important pandas feature for some applications is the behavior of arithmetic between objects with different indexes. When you are adding together objects, if any index pairs are not the same, the respective index in the result will be the union of the index pairs. For users with database experience, this is similar to an automatic outer join on the index labels. Let’s look at an example:

In [141…
```
s1 = pd.Series([7.3, -2.5, 3.4, 1.5], index=['a', 'c', 'd', 'e'])
s2 = pd.Series([-2.1, 3.6, -1.5, 4, 3.1],index=['a', 'c', 'e', 'f', 'g'])
```

In [142…
```
s1
```
Out[142…
```
a    7.3
c   -2.5
d    3.4
e    1.5
dtype: float64
```

In [143…
```
s2
```
Out[143…
```
a   -2.1
c    3.6
e   -1.5
f    4.0
g    3.1
dtype: float64
```

In [144…
```
s1 + s2
```

Out[144…
```
a    5.2
c    1.1
d    NaN
e    0.0
f    NaN
g    NaN
dtype: float64
```

The internal data alignment introduces missing values in the label locations that don't overlap. Missing values will then propagate in further arithmetic computations.

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

In [145…
```
df1 = pd.DataFrame(np.arange(9.).reshape((3, 3)), columns=list('bcd'),index=['Ohio'
df2 = pd.DataFrame(np.arange(12.).reshape((4, 3)), columns=list('bde'),index=['Utah
df1
```

Out[145…

	b	c	d
Ohio	0.0	1.0	2.0
Texas	3.0	4.0	5.0
Colorado	6.0	7.0	8.0

In [146…
```
df2
```

Out[146…

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

Adding these together returns a DataFrame whose index and columns are the unions of the ones in each DataFrame:

In [147…
```
df1 + df2
```

Out[147…

	b	c	d	e
Colorado	NaN	NaN	NaN	NaN
Ohio	3.0	NaN	6.0	NaN
Oregon	NaN	NaN	NaN	NaN
Texas	9.0	NaN	12.0	NaN
Utah	NaN	NaN	NaN	NaN

If you add DataFrame objects with no column or row labels in common, the result will contain all nulls:

In [148…

```
df1 = pd.DataFrame({'A': [1, 2]})
df2 = pd.DataFrame({'B': [3, 4]})
df1
```

Out[148…

	A
0	1
1	2

In [150…

```
df2
```

Out[150…

	B
0	3
1	4

In [151…

```
df1 - df2
```

Out[151…

	A	B
0	NaN	NaN
1	NaN	NaN

Arithmetic methods with fill values

In [152…

```
df1 = pd.DataFrame(np.arange(12.).reshape((3, 4)),columns=list('abcd'))
df2 = pd.DataFrame(np.arange(20.).reshape((4, 5)),columns=list('abcde'))
df2.loc[1, 'b'] = np.nan
df1
```

Out[152…

	a	b	c	d
0	0.0	1.0	2.0	3.0
1	4.0	5.0	6.0	7.0
2	8.0	9.0	10.0	11.0

In [153…

```
df2
```

Out[153…

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	4.0
1	5.0	NaN	7.0	8.0	9.0
2	10.0	11.0	12.0	13.0	14.0
3	15.0	16.0	17.0	18.0	19.0

Adding these together results in NA values in the locations that don’t overlap:

In [154…

```
df1 + df2
```

Out[154…

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	NaN
1	9.0	NaN	13.0	15.0	NaN
2	18.0	20.0	22.0	24.0	NaN
3	NaN	NaN	NaN	NaN	NaN

Using the add method on df1, I pass df2 and an argument to fill_value:

In [155…

```
df1.add(df2, fill_value=0)
```

Out[155…

	a	b	c	d	e
0	0.0	2.0	4.0	6.0	4.0
1	9.0	5.0	13.0	15.0	9.0
2	18.0	20.0	22.0	24.0	14.0
3	15.0	16.0	17.0	18.0	19.0

In [156…

```
1 / df1
```

Out[156…

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250	0.200000	0.166667	0.142857
2	0.125	0.111111	0.100000	0.090909

In [157…

```
df1.rdiv(1)
```

Out[157…

	a	b	c	d
0	inf	1.000000	0.500000	0.333333
1	0.250	0.200000	0.166667	0.142857
2	0.125	0.111111	0.100000	0.090909

Relatedly, when reindexing a Series or DataFrame, you can also specify a different fill value:

In [158…

```
df1.reindex(columns=df2.columns, fill_value=0)
```

Out[158…

	a	b	c	d	e
0	0.0	1.0	2.0	3.0	0
1	4.0	5.0	6.0	7.0	0
2	8.0	9.0	10.0	11.0	0

In [1]:

```
# Table 5-5. --> Flexible arithmetic methods
# ------------------------------------------
# Method --> Description
# --------------------------
# add, radd --> Methods for addition (+)
# sub, rsub --> Methods for subtraction (-)
# div, rdiv --> Methods for division (/)
# floordiv, rfloordiv --> Methods for foor division (//)
# mul, rmul --> Methods for multiplication (*)
# pow, rpow --> Methods for exponentiation (**)
```

Operations between DataFrame and Series

As with NumPy arrays of different dimensions, arithmetic between DataFrame and Series is also defined. First, as a motivating example, consider the difference between a two-dimensional array and one of its rows:

In [160…

```
arr = np.arange(12.).reshape((3, 4))
```

In [161…

```
arr
```

Out[161…

```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

In [162…

```
arr[0]
```

Out[162…

```
array([0., 1., 2., 3.])
```

In [163…

```
arr - arr[0]
```

Out[163…

```
array([[0., 0., 0., 0.],
       [4., 4., 4., 4.],
       [8., 8., 8., 8.]])
```

When we subtract arr[0] from arr, the subtraction is performed once for each row. This is referred to as broadcasting and is explained in more detail as it relates to general NumPy arrays in Appendix A. Operations between a DataFrame and a Series are similar:

In [164…

```
frame = pd.DataFrame(np.arange(12.).reshape((4, 3)),columns=list('bde'),index=['Uta
series = frame.iloc[0]
frame
```

Out[164…

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [165…

```
series
```

Out[165…

```
b    0.0
d    1.0
e    2.0
Name: Utah, dtype: float64
```

By default, arithmetic between DataFrame and Series matches the index of the Series on the DataFrame's columns, broadcasting down the rows:

In [166…

```
frame - series
```

Out[166…

	b	d	e
Utah	0.0	0.0	0.0
Ohio	3.0	3.0	3.0
Texas	6.0	6.0	6.0
Oregon	9.0	9.0	9.0

If an index value is not found in either the DataFrame's columns or the Series's index, the objects will be reindexed to form the union:

In [167...
```
series2 = pd.Series(range(3), index=['b', 'e', 'f'])
frame + series2
```

Out[167...

	b	d	e	f
Utah	0.0	NaN	3.0	NaN
Ohio	3.0	NaN	6.0	NaN
Texas	6.0	NaN	9.0	NaN
Oregon	9.0	NaN	12.0	NaN

If you want to instead broadcast over the columns, matching on the rows, you have to use one of the arithmetic methods. For example:

In [168...
```
series3 = frame['d']
frame
```

Out[168...

	b	d	e
Utah	0.0	1.0	2.0
Ohio	3.0	4.0	5.0
Texas	6.0	7.0	8.0
Oregon	9.0	10.0	11.0

In [169...
```
series3
```

Out[169...
```
Utah       1.0
Ohio       4.0
Texas      7.0
Oregon    10.0
Name: d, dtype: float64
```

In [170...
```
frame.sub(series3, axis='index')
```

Out[170...

	b	d	e
Utah	-1.0	0.0	1.0
Ohio	-1.0	0.0	1.0
Texas	-1.0	0.0	1.0
Oregon	-1.0	0.0	1.0

The axis number that you pass is the axis to match on. In this case we mean to match on the DataFrame's row index (axis='index' or axis=0) and broadcast across.

Function Application and Mapping

NumPy ufuncs (element-wise array methods) also work with pandas objects:

In [171…
```
frame = pd.DataFrame(np.random.randn(4, 3), columns=list('bde'),index=['Utah', 'Ohi
```

In [172…
```
frame
```

Out[172…

	b	d	e
Utah	0.546979	0.220407	-2.005660
Ohio	0.166589	1.285297	2.852149
Texas	-0.259623	0.490741	-0.231152
Oregon	0.635178	-0.666540	0.805573

In [173…
```
np.abs(frame)
```

Out[173…

	b	d	e
Utah	0.546979	0.220407	2.005660
Ohio	0.166589	1.285297	2.852149
Texas	0.259623	0.490741	0.231152
Oregon	0.635178	0.666540	0.805573

Another frequent operation is applying a function on one-dimensional arrays to each column or row. DataFrame's apply method does exactly this:

In [174…
```
f = lambda x: x.max() - x.min()
```

In [175…
```
frame.apply(f)
```

Out[175…
```
b    0.894801
d    1.951837
e    4.857809
dtype: float64
```

Here the function f, which computes the difference between the maximum and minimum of a Series, is invoked once on each column in frame. The result is a Series having the columns of frame as its index.

If you pass axis='columns' to apply, the function will be invoked once per row instead:

In [176…
```
frame.apply(f, axis='columns')
```
Out[176…
```
Utah      2.552639
Ohio      2.685560
Texas     0.750364
Oregon    1.472112
dtype: float64
```

Many of the most common array statistics (like sum and mean) are DataFrame methods, so using apply is not necessary.

The function passed to apply need not return a scalar value; it can also return a Series with multiple values:

In [177…
```
def f(x):
    return pd.Series([x.min(), x.max()], index=['min', 'max'])
```
In [178…
```
frame.apply(f)
```
Out[178…

	b	d	e
min	-0.259623	-0.666540	-2.005660
max	0.635178	1.285297	2.852149

Element-wise Python functions can be used, too. Suppose you wanted to compute a formatted string from each floating-point value in frame. You can do this with apply map:

In [179…
```
format = lambda x: '%.2f' % x
frame.applymap(format)
```
Out[179…

	b	d	e
Utah	0.55	0.22	-2.01
Ohio	0.17	1.29	2.85
Texas	-0.26	0.49	-0.23
Oregon	0.64	-0.67	0.81

The reason for the name applymap is that Series has a map method for applying an element-wise function:

In [180…
```
frame['e'].map(format)
```
Out[180…
```
Utah      -2.01
Ohio       2.85
Texas     -0.23
Oregon     0.81
Name: e, dtype: object
```

Sorting and Ranking

In [181…
```
obj = pd.Series(range(4), index=['d', 'a', 'b', 'c'])
```

In [182…
```
obj.sort_index()
```

Out[182…
```
a    1
b    2
c    3
d    0
dtype: int64
```

In [183…
```
frame = pd.DataFrame(np.arange(8).reshape((2, 4)),index=['three', 'one'],columns=['
```

In [184…
```
frame.sort_index()
```

Out[184…

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

In [185…
```
frame.sort_index(axis=1)
```

Out[185…

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

The data is sorted in ascending order by default, but can be sorted in descending order, too:

In [186…
```
frame.sort_index(axis=1, ascending=False)
```

Out[186…

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

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

In [187…
```
obj = pd.Series([4, 7, -3, 2])
obj.sort_values()
```

Out[187…
```
2   -3
3    2
0    4
1    7
dtype: int64
```

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

In [188…
```
obj = pd.Series([4, np.nan, 7, np.nan, -3, 2])
obj.sort_values()
```

Out[188…
```
4   -3.0
5    2.0
0    4.0
2    7.0
1    NaN
3    NaN
dtype: float64
```

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

In [189…
```
frame = pd.DataFrame({'b': [4, 7, -3, 2], 'a': [0, 1, 0, 1]})
frame
```

Out[189…

	b	a
0	4	0
1	7	1
2	-3	0
3	2	1

In [190…
```
frame.sort_values(by='b')
```

Out[190…

	b	a
2	-3	0
3	2	1
0	4	0
1	7	1

In [191…
```
frame.sort_values(by=['a', 'b'])
```

Out[191…

	b	a
2	-3	0
0	4	0
3	2	1
1	7	1

Ranking assigns ranks from one through the number of valid data points in an array. The rank methods for Series and DataFrame are the place to look; by default rank breaks ties by assigning each group the mean rank:

In [192...
```
obj = pd.Series([7, -5, 7, 4, 2, 0, 4])
obj.rank()
```

Out[192...
```
0    6.5
1    1.0
2    6.5
3    4.5
4    3.0
5    2.0
6    4.5
dtype: float64
```

Ranks can also be assigned according to the order in which they're observed in the data:

In [193...
```
obj.rank(method='first')
```

Out[193...
```
0    6.0
1    1.0
2    7.0
3    4.0
4    3.0
5    2.0
6    5.0
dtype: float64
```

Here, instead of using the average rank 6.5 for the entries 0 and 2, they instead have been set to 6 and 7 because label 0 precedes label 2 in the data.

You can rank in descending order, too:

In [194...
```
# Assign tie values the maximum rank in the group
obj.rank(ascending=False, method='max')
```

Out[194...
```
0    2.0
1    7.0
2    2.0
3    4.0
4    5.0
5    6.0
6    4.0
dtype: float64
```

DataFrame can compute ranks over the rows or the columns:

In [195...
```
frame = pd.DataFrame({'b': [4.3, 7, -3, 2], 'a': [0, 1, 0, 1],'c': [-2, 5, 8, -2.5]
frame
```

Out[195...

	b	a	c
0	4.3	0	-2.0
1	7.0	1	5.0
2	-3.0	0	8.0
3	2.0	1	-2.5

In [196...

```
frame.rank(axis='columns')
```

Out[196...

	b	a	c
0	3.0	2.0	1.0
1	3.0	1.0	2.0
2	1.0	2.0	3.0
3	3.0	2.0	1.0

In [197...

```
# Table 5-6. --> Tie-breaking methods with rank
# ---------------------------------------------
# Method --> Description
# -----------------------
# 'average' --> Default: assign the average rank to each entry in the equal group
# 'min' --> Use the minimum rank for the whole group
# 'max' --> Use the maximum rank for the whole group
# 'first' --> Assign ranks in the order the values appear in the data
# 'dense' --> Like method='min', but ranks always increase by 1 in between groups r
# elements in a group
```

Axis Indexes with Duplicate Labels

While many pandas functions (like reindex) require that the labels be unique, it's not mandatory. Let's consider a small Series with duplicate indices:

In [198...

```
obj = pd.Series(range(5), index=['a', 'a', 'b', 'b', 'c'])
```

In [199...

```
obj
```

Out[199...

```
a    0
a    1
b    2
b    3
c    4
dtype: int64
```

The index's is_unique property can tell you whether its labels are unique or not:

In [200...

```
obj.index.is_unique
```

Out[200…
```
False
```

Data selection is one of the main things that behaves differently with duplicates. Indexing a label with multiple entries returns a Series, while single entries return a scalar value:

In [201…
```
obj['a']
```

Out[201…
```
a    0
a    1
dtype: int64
```

In [202…
```
obj['c']
```

Out[202…
```
4
```

This can make your code more complicated, as the output type from indexing can vary based on whether a label is repeated or not.

The same logic extends to indexing rows in a DataFrame:

In [203…
```
df = pd.DataFrame(np.random.randn(4, 3), index=['a', 'a', 'b', 'b'])
```

In [204…
```
df
```

Out[204…

	0	1	2
a	0.138858	-0.518643	-0.467778
a	-2.183903	-0.641348	0.631495
b	-0.982207	-0.916507	0.722714
b	-0.441702	1.533297	0.116198

In [205…
```
 df.loc['b']
```

Out[205…

	0	1	2
b	-0.982207	-0.916507	0.722714
b	-0.441702	1.533297	0.116198

5.3 Summarizing and Computing Descriptive Statistics

pandas objects are equipped with a set of common mathematical and statistical methods. Most of these fall into the category of reductions or summary statistics, methods that extract a single value (like the sum or mean) from a Series or a Series of values from the rows or columns of a DataFrame. Compared with the similar methods found on NumPy arrays, they have built-in handling for missing data. Consider a small DataFrame

In [206…
```
df = pd.DataFrame([[1.4, np.nan], [7.1, -4.5],[np.nan, np.nan], [0.75, -1.3]],index
df
```

Out[206…

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

calling DataFrame’s sum method returns a Series containing column sums:

In [207…
```
df.sum()
```

Out[207…
```
one    9.25
two   -5.80
dtype: float64
```

Passing axis='columns' or axis=1 sums across the columns instead:

In [208…
```
df.sum(axis='columns')
```

Out[208…
```
a    1.40
b    2.60
c    0.00
d   -0.55
dtype: float64
```

NA values are excluded unless the entire slice (row or column in this case) is NA. This can be disabled with the skipna option:

In [209…
```
df.mean(axis='columns', skipna=False)
```

Out[209…
```
a      NaN
b    1.300
c      NaN
d   -0.275
dtype: float64
```

In [210…
```
# Table 5-7. --> Options for reduction methods
# --------------------------------------------
# Method --> Description
# ----------------------
# axis --> Axis to reduce over; 0 for DataFrame’s rows and 1 for columns
# skipna --> Exclude missing values; True by default
# level --> Reduce grouped by level if the axis is hierarchically indexed (MultiInd
```

Some methods, like idxmin and idxmax, return indirect statistics like the index value where the minimum or maximum values are attained:

In [211…
```
df.idxmax()
```

Out[211…
```
one      b
two      d
dtype: object
```

In [212…
```
df.cumsum()
```

Out[212…

	one	two
a	1.40	NaN
b	8.50	-4.5
c	NaN	NaN
d	9.25	-5.8

describe is one such example, producing multiple summary statistics in one shot:

In [213…
```
df.describe()
```

Out[213…

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

On non-numeric data, describe produces alternative summary statistics:

In [214…
```
obj = pd.Series(['a', 'a', 'b', 'c'] * 4)
obj.describe()
```

Out[214…
```
count      16
unique      3
top         a
freq        8
dtype: object
```

In [215…
```
# Table 5-8. --> Descriptive and summary statistics
# -----------------------------------------------
# Method --> Description
# ----------------------
# count --> Number of non-NA values
# describe --> Compute set of summary statistics for Series or each DataFrame colum
# min, max --> Compute minimum and maximum values
# argmin, argmax --> Compute index locations (integers) at which minimum or maximum
# idxmin, idxmax --> Compute index labels at which minimum or maximum value obtaine
# quantile --> Compute sample quantile ranging from 0 to 1
# sum --> Sum of values
# mean --> Mean of values
# median --> Arithmetic median (50% quantile) of values
# mad --> Mean absolute deviation from mean value
# prod --> Product of all values
# var --> Sample variance of values
# std --> Sample standard deviation of values
# skew --> Sample skewness (third moment) of values
# kurt --> Sample kurtosis (fourth moment) of values
# cumsum --> Cumulative sum of values
# cummin, cummax --> Cumulative minimum or maximum of values, respectively
# cumprod --> Cumulative product of values
# diff --> Compute first arithmetic difference (useful for time series)
# pct_change --> Compute percent changes
```

Correlation and Covariance

In [5]:
```
# ! pip install pandas-datareader
```

I use the pandas_datareader module to download some data for a few stock tickers:

In [6]:
```
import pandas_datareader.data as web
all_data = {ticker: web.get_data_yahoo(ticker) for ticker in ['AAPL', 'IBM', 'MSFT'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[6], line 2
      1 import pandas_datareader.data as web
----> 2 all_data = {ticker: web.get_data_yahoo(ticker) for ticker in ['AAPL', 'IBM',
'MSFT', 'GOOG']}

Cell In[6], line 2, in <dictcomp>(.0)
      1 import pandas_datareader.data as web
----> 2 all_data = {ticker: web.get_data_yahoo(ticker) for ticker in ['AAPL', 'IBM',
'MSFT', 'GOOG']}

File ~\ankit\ankit\myenv\lib\site-packages\pandas_datareader\data.py:80, in get_data
_yahoo(*args, **kwargs)
     79 def get_data_yahoo(*args, **kwargs):
---> 80     return YahooDailyReader(*args, **kwargs).read()

File ~\ankit\ankit\myenv\lib\site-packages\pandas_datareader\base.py:253, in _DailyB
aseReader.read(self)
    251 # If a single symbol, (e.g., 'GOOG')
    252 if isinstance(self.symbols, (string_types, int)):
--> 253     df = self._read_one_data(self.url, params=self._get_params(self.symbol
s))
    254 # Or multiple symbols, (e.g., ['GOOG', 'AAPL', 'MSFT'])
    255 elif isinstance(self.symbols, DataFrame):

File ~\ankit\ankit\myenv\lib\site-packages\pandas_datareader\yahoo\daily.py:153, in
YahooDailyReader._read_one_data(self, url, params)
    151 try:
    152     j = json.loads(re.search(ptrn, resp.text, re.DOTALL).group(1))
--> 153     data = j["context"]["dispatcher"]["stores"]["HistoricalPriceStore"]
    154 except KeyError:
    155     msg = "No data fetched for symbol {} using {}"

TypeError: string indices must be integers
```

In [ ]:
```
price = pd.DataFrame({ticker: data['Adj Close'] for ticker, data in all_data.items(
volume = pd.DataFrame({ticker: data['Volume'] for ticker, data in all_data.items()}
```

In [ ]:
```
returns = price.pct_change()
returns.tail()
```

The corr method of Series computes the correlation of the overlapping, non-NA, aligned-by-index values in two Series. Relatedly, cov computes the covariance:

In [ ]:
```
returns['MSFT'].corr(returns['IBM'])
```

In [ ]:
```
returns['MSFT'].cov(returns['IBM'])
```

Since MSFT is a valid Python attribute, we can also select these columns using more concise syntax:

In [ ]:
```
returns.MSFT.corr(returns.IBM)
```

DataFrame’s corr and cov methods, on the other hand, return a full correlation or covariance matrix as a DataFrame, respectively:

In [ ]:
```
returns.corr()
```

In [ ]:
```
returns.cov()
```

Using DataFrame’s corrwith method, you can compute pairwise correlations between a DataFrame’s columns or rows with another Series or DataFrame. Passing a Series returns a Series with the correlation value computed for each column:

In [ ]:
```
returns.corrwith(returns.IBM)
```

Passing a DataFrame computes the correlations of matching column names. Here I compute correlations of percent changes with volume:

In [ ]:
```
returns.corrwith(volume)
```

Passing axis='columns' does things row-by-row instead. In all cases, the data points are aligned by label before the correlation is computed.

In [ ]:
```
obj = pd.Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

In [ ]:
```
uniques = obj.unique()
```

In [ ]:
```
uniques
```

The unique values are not necessarily returned in sorted order, but could be sorted after the fact if needed (uniques.sort()). Relatedly, value_counts computes a Series containing value frequencies:

In [ ]:
```
obj.value_counts()
```

The Series is sorted by value in descending order as a convenience. value_counts is also available as a top-level pandas method that can be used with any array or sequence

In [ ]:
```
pd.value_counts(obj.values, sort=False)
```

isin performs a vectorized set membership check and can be useful in filtering a dataset down to a subset of values in a Series or column in a DataFrame:

In [ ]:
```
obj
```

In [ ]:
```
mask = obj.isin(['b', 'c'])
mask
```

In [ ]:
```
obj[mask]
```

Related to isin is the Index.get_indexer method, which gives you an index array from an array of possibly non-distinct values into another array of distinct values:

In [ ]:
```
to_match = pd.Series(['c', 'a', 'b', 'b', 'c', 'a'])
unique_vals = pd.Series(['c', 'b', 'a'])
pd.Index(unique_vals).get_indexer(to_match)
```

In [ ]:
```
# Table 5-9. --> Unique, value counts, and set membership methods
# ------------------------------------------------------------------
# Method --> Description
# -----------------------------
# isin --> Compute boolean array indicating whether each Series value is contained
# values
# match --> Compute integer indices for each value in an array into another array o
# alignment and join-type operations
# unique --> Compute array of unique values in a Series, returned in the order obse
# value_counts --> Return a Series containing unique values as its index and freque
# descending order
```

In some cases, you may want to compute a histogram on multiple related columns in a DataFrame. Here's an example:

In [ ]:
```
data = pd.DataFrame({'Qu1': [1, 3, 4, 3, 4],'Qu2': [2, 3, 1, 2, 3],'Qu3': [1, 5, 2,
data
```

Passing pandas.value_counts to this DataFrame's apply function gives:

In [ ]:
```
result = data.apply(pd.value_counts).fillna(0)
result
```

Here, the row labels in the result are the distinct values occurring in all of the col- umns. The values are the respective counts of these values in each column.

In [ ]:
```

```