UNIT 5

PANDAS and INTRODUCTION to SCIPY

Contents

Introduction to Pandas, Pandas Objects,

Operations in Pandas: Object creation, viewing data, selection,

Different ways of creating DataFrame,

Handling missing data, merge and concat operations.

Pivot and Pivot table.

SciPy: Introduction to Optimization and Minimization, Interpolation, Integration, Statistics

Introducing Pandas Objects

At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices. Pandas provides a host of useful tools, methods, and functionality on top of the basic data structures, but nearly everything that follows will require an understanding of what these structures are. Thus, before we go any further,
let's introduce these three fundamental Pandas data structures: the Series, DataFrame, and Index.

Series

A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its *index*. The simplest Series is formed from only an array of data:

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

In [5]: obj
Out[5]:
0    4
1    7
2   -5
3    3
```

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 [6]: obj.values
Out[6]: array([ 4,  7, -5,  3])

In [7]: obj.index
Out[7]: Int64Index([0, 1, 2, 3])
```

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

```
In [8]: obj2 = Series([4, 7, -5, 3], index=['d', 'b', 'a', 'c'])

In [9]: obj2
Out[9]:
d    4
b    7
a   -5
c    3
```

```
In [10]: obj2.index
Out[10]: Index([d, b, a, c], dtype=object)
```

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

```
In [11]: obj2['a']
Out[11]: -5

In [12]: obj2['d'] = 6

In [13]: obj2[['c', 'a', 'd']]
Out[13]:
c    3
a   -5
d    6
```

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

```
In [14]: obj2
Out[14]:
d    6
b    7
a   -5
c    3
```

```
In [15]: obj2[obj2 > 0]
Out[15]:
d    6
b    7
c    3
```

```
In [16]: obj2 * 2
Out[16]:
d    12
b    14
a   -10
c     6
```

```
In [17]: np.exp(obj2)
Out[17]:
d     403.428793
b    1096.633158
a       0.006738
c      20.085537
```

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 substituted into many functions that expect a dict:

```
In [18]: 'b' in obj2
Out[18]: True

In [19]: 'e' in obj2
Out[19]: False
```

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

```
In [20]: sdata = {'Ohio': 35000, 'Texas': 71000, 'Oregon': 16000, 'Utah': 5000}
In [21]: obj3 = Series(sdata)
```

When only passing a dict, the index in the resulting Series will have the dict's keys

```
In [23]: states = ['California', 'Ohio', 'Oregon', 'Texas']

In [24]: obj4 = Series(sdata, index=states)
```

In this case, 3 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. I will use the terms "missing" or "NA" to refer to missing data. The isnull and notnull functions in pandas should be used to detect missing data:

```
In [26]: pd.isnull(obj4)      In [27]: pd.notnull(obj4)
```

Series also has these as instance methods:

```
In [28]: obj4.isnull()
```

DataFrame

A DataFrame represents a tabular, spreadsheet-like data structure containing 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 (one for all sharing the same index)

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

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

If you specify a sequence of columns, the DataFrame's columns will be exactly what you pass:

```
In [39]: DataFrame(data, columns=['year', 'state', 'pop'])
Out[39]:
   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
```

As with Series, if you pass a column that isn't contained in data, it will appear with NA values in the result:

```
In [40]: frame2 = DataFrame(data, columns=['year', 'state', 'pop', 'debt'],
   ....:                    index=['one', 'two', 'three', 'four', 'five'])

In [41]: frame2
Out[41]:
       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

In [42]: frame2.columns
Out[42]: 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 [43]: frame2['state']
Out[43]:
one        Ohio
```

```
In [44]: frame2.year
Out[44]:
one       2000
```

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 by a couple of methods, such as the ix indexing field (much more on this later):

```
In [45]: frame2.ix['three']
Out[45]:
year      2002
state     Ohio
pop        3.6
debt       NaN
Name: three
```

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

```
In [46]: frame2['debt'] = 16.5

In [47]: frame2
Out[47]:
       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

In [48]: frame2['debt'] = np.arange(5.)
```

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

```
In [50]: val = Series([-1.2, -1.5, -1.7], index=['two', 'four', 'five'])

In [51]: frame2['debt'] = val

In [52]: frame2
```

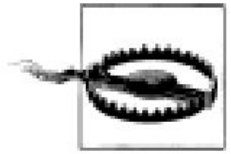
Assigning a column that doesn't exist will create a new column. The del keyword will delete columns as with a dict:

```
In [53]: frame2['eastern'] = frame2.state == 'Ohio'

In [54]: frame2
Out[54]:
       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

In [55]: del frame2['eastern']

In [56]: frame2.columns
Out[56]: Index([year, state, pop, debt], dtype=object)
```



The column returned when 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 using the Series's `copy` method.

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

```
In [57]: pop = {'Nevada': {2001: 2.4, 2002: 2.9},
   ....:        'Ohio': {2000: 1.5, 2001: 1.7, 2002: 3.6}}
```

If passed to DataFrame, it will interpret the outer dict keys as the columns and the inner keys as the row indices:

```
In [58]: frame3 = DataFrame(pop)

In [59]: frame3
Out[59]:
      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
2002     2.9   3.6
```

Of course you can always transpose the result:

```
In [60]: frame3.T
```

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

```
In [61]: DataFrame(pop, index=[2001, 2002, 2003])
Out[61]:
      Nevada  Ohio
2001     2.4   1.7
2002     2.9   3.6
2003     NaN   NaN
```

Dicts of Series are treated much in the same way:

```
In [62]: pdata = {'Ohio': frame3['Ohio'][:-1],
   ....:          'Nevada': frame3['Nevada'][:2]}

In [63]: DataFrame(pdata)
Out[63]:
      Nevada  Ohio
2000     NaN   1.5
2001     2.4   1.7
```

For a complete list of things you can pass the DataFrame constructor, see Table 5-1.

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 DataFrame. All sequences must be the same length.
NumPy structured/record array	Treated as the "dict of arrays" case
dict of Series	Each value becomes a column. Indexes from each Series are unioned together to form the 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 row index as in the "dict of Series" case.
list of dicts or Series	Each item becomes a row in the DataFrame. Union of dict keys or Series indexes become the 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 passed
NumPy MaskedArray	Like the "2D ndarray" case except masked values become NA/missing in the DataFrame result

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

```
In [64]: frame3.index.name = 'year'; frame3.columns.name = 'state'

In [65]: frame3
Out[65]:
state  Nevada  Ohio
year
2000      NaN   1.5
2001      2.4   1.7
2002      2.9   3.6
```

Like Series, the values attribute returns the data contained in the DataFrame as a 2D ndarray:

```
In [66]: frame3.values
Out[66]:
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 accomodate all of the columns:

```
In [67]: frame2.values
Out[67]:
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]], dtype=object)
```

Handling Missing Data

Missing data is common in most data analysis applications. One of the goals in designing pandas was to make working with missing data as painless as possible. For example, all of the descriptive statistics on pandas objects exclude missing data as

pandas uses the floating point value `NaN` (Not a Number) to represent missing data in both floating as well as in non-floating point arrays. It is just used as a *sentinel* that can be easily detected:

```
In [229]: string_data = Series(['aardvark', 'artichoke', np.nan, 'avocado'])

In [230]: string_data
Out[230]:
0     aardvark
1    artichoke
2          NaN
3      avocado

In [231]: string_data.isnull()
Out[231]:
0    False
1    False
2     True
3    False
```

The built-in Python `None` value is also treated as NA in object arrays:

```
In [232]: string_data[0] = None

In [233]: string_data.isnull()
Out[233]:
0     True
1    False
2     True
3    False
```

Table 5-12. NA handling methods

Argument	Description
`dropna`	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
`fillna`	Fill in missing data with some value or using an interpolation method such as `'ffill'` or `'bfill'`.
`isnull`	Return like-type object containing boolean values indicating which values are missing / NA.
`notnull`	Negation of `isnull`.

Filtering Out Missing Data

You have a number of options for filtering out missing data. While doing it by hand is always an option, dropna can be very helpful. On a Series, it returns the Series with only the non-null data and index values:

```
In [234]: from numpy import nan as NA

In [235]: data = Series([1, NA, 3.5, NA, 7])

In [236]: data.dropna()
Out[236]:
```

Naturally, you could have computed this yourself by boolean indexing:

```
In [237]: data[data.notnull()]
Out[237]:
0    1.0
2    3.5
4    7.0
```

With DataFrame objects, these are a bit more complex. You may want to drop rows or columns which are all NA or just those containing any NAs. dropna by default drops any row containing a missing value:

```
In [238]: data = DataFrame([[1., 6.5, 3.], [1., NA, NA],
   .....:                   [NA, NA, NA], [NA, 6.5, 3.]])

In [239]: cleaned = data.dropna()

In [240]: data
Out[240]:
    0    1   2
0   1  6.5   3
1   1  NaN NaN
2 NaN  NaN NaN
3 NaN  6.5   3

In [241]: cleaned
Out[241]:
   0    1  2
0  1  6.5  3
```

Passing how='all' will only drop rows that are all NA:

```
In [242]: data.dropna(how='all')
Out[242]:
    0    1   2
0   1  6.5   3
1   1  NaN NaN
3 NaN  6.5   3
```

Dropping columns in the same way is only a matter of passing axis=1:

```
In [243]: data[4] = NA

In [244]: data

In [245]: data.dropna(axis=1, how='all')
```

A related way to filter out DataFrame rows tends to concern time series data. Suppose you want to keep only rows containing a certain number of observations. You can indicate this with the thresh argument:

```
In [246]: df = DataFrame(np.random.randn(7, 3))

In [247]: df.ix[:4, 1] = NA; df.ix[:2, 2] = NA
```

```
In [248]: df
```

```
In [249]: df.dropna(thresh=3)
```

Filling in Missing Data

Rather than filtering out missing data (and potentially discarding other data along with it), you may want to fill in the "holes" in any number of ways. For most purposes, the fillna method is the workhorse function to use. Calling fillna with a constant replaces missing values with that value:

```
In [250]: df.fillna(0)
Out[250]:
          0         1         2
0 -0.577087  0.000000  0.000000
1  0.523772  0.000000  0.000000
2 -0.713544  0.000000  0.000000
3 -1.860761  0.000000  0.560145
4 -1.265934  0.000000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

Calling fillna with a dict you can use a different fill value for each column:

```
In [251]: df.fillna({1: 0.5, 3: -1})
Out[251]:
          0         1         2
0 -0.577087  0.500000       NaN
1  0.523772  0.500000       NaN
2 -0.713544  0.500000       NaN
3 -1.860761  0.500000  0.560145
4 -1.265934  0.500000 -1.063512
5  0.332883 -2.359419 -0.199543
6 -1.541996 -0.970736 -1.307030
```

fillna returns a new object, but you can modify the existing object in place:

```
# always returns a reference to the filled object
In [252]: _ = df.fillna(0, inplace=True)

In [253]: df
```

The same interpolation methods available for reindexing can be used with fillna:

```
In [254]: df = DataFrame(np.random.randn(6, 3))

In [255]: df.ix[2:, 1] = NA; df.ix[4:, 2] = NA

In [256]: df
Out[256]:
          0         1         2
0  0.286350  0.377984 -0.753887
1  0.331286  1.349742  0.069877
2  0.246674       NaN  1.004812
3  1.327195       NaN -1.549106
4  0.022185       NaN       NaN
5  0.862580       NaN       NaN

In [257]: df.fillna(method='ffill')      In [258]: df.fillna(method='ffill', limit=2)
```

With fillna you can do lots of other things with a little creativity. For example, you might pass the mean or median value of a Series:

```
In [259]: data = Series([1., NA, 3.5, NA, 7])

In [260]: data.fillna(data.mean())
Out[260]:
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
```

See Table 5-13 for a reference on fillna.

Table 5-13. fillna function arguments

Argument	Description
value	Scalar value or dict-like object to use to fill missing values
method	Interpolation, by default 'ffill' if function called with no other arguments
axis	Axis to fill on, default axis=0
inplace	Modify the calling object without producing a copy
limit	For forward and backward filling, maximum number of consecutive periods to fill

Combining and Merging Data Sets

Data contained in pandas objects can be combined together in a number of built-in ways:

- `pandas.merge` connects rows in DataFrames based on one or more keys. This will be familiar to users of SQL or other relational databases, as it implements database *join* operations.
- `pandas.concat` glues or stacks together objects along an axis.

Database-style DataFrame Merges

Merge or *join* operations combine data sets by linking rows using one or more *keys*. These operations are central to relational databases. The `merge` function in pandas is the main entry point for using these algorithms on your data.

Let's start with a simple example:

```
In [15]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                  'data1': range(7)})

In [16]: df2 = DataFrame({'key': ['a', 'b', 'd'],
   ....:                  'data2': range(3)})

In [17]: df1
Out[17]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   a
6      6   b

In [18]: df2
Out[18]:
   data2 key
0      0   a
1      1   b
2      2   d
```

This is an example of a *many-to-one* merge situation; the data in df1 has multiple rows labeled a and b, whereas df2 has only one row for each value in the key column. Calling merge with these objects we obtain:

```
In [19]: pd.merge(df1, df2)
Out[19]:
   data1 key  data2
0      2   a      0
1      4   a      0
2      5   a      0
3      0   b      1
4      1   b      1
5      6   b      1
```

Note that I didn't specify which column to join on. If not specified, merge uses the overlapping column names as the keys. It's a good practice to specify explicitly, though:

```
In [20]: pd.merge(df1, df2, on='key')
Out[20]:
   data1 key  data2
0      2   a      0
1      4   a      0
2      5   a      0
3      0   b      1
4      1   b      1
5      6   b      1
```

If the column names are different in each object, you can specify them separately:

```
In [21]: df3 = DataFrame({'lkey': ['b', 'b', 'a', 'c', 'a', 'a', 'b'],
   ....:                  'data1': range(7)})
```

```
In [22]: df4 = DataFrame({'rkey': ['a', 'b', 'd'],
   ....:                  'data2': range(3)})

In [23]: pd.merge(df3, df4, left_on='lkey', right_on='rkey')
Out[23]:
   data1 lkey  data2 rkey
0      2    a      0    a
1      4    a      0    a
2      5    a      0    a
3      0    b      1    b
4      1    b      1    b
5      6    b      1    b
```

You probably noticed that the 'c' and 'd' values and associated data are missing from the result. By default merge does an 'inner' join; the keys in the result are the intersection. Other possible options are 'left', 'right', and 'outer'. The outer join takes the union of the keys, combining the effect of applying both left and right joins:

```
In [24]: pd.merge(df1, df2, how='outer')
Out[24]:
   data1 key  data2
0      2   a      0
1      4   a      0
2      5   a      0
3      0   b      1
4      1   b      1
5      6   b      1
6      3   c    NaN
7    NaN   d      2
```

Many-to-many merges have well-defined though not necessarily intuitive behavior. Here's an example:

```
In [25]: df1 = DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
   ....:                  'data1': range(6)})

In [26]: df2 = DataFrame({'key': ['a', 'b', 'a', 'b', 'd'],
   ....:                  'data2': range(5)})
```

```
In [27]: df1
Out[27]:
   data1 key
0      0   b
1      1   b
2      2   a
3      3   c
4      4   a
5      5   b
```

```
In [28]: df2
Out[28]:
   data2 key
0      0   a
1      1   b
2      2   a
3      3   b
4      4   d
```

```
In [29]: pd.merge(df1, df2, on='key', how='left')
Out[29]:
    data1 key  data2
0       2   a      0
1       2   a      2
2       4   a      0
3       4   a      2
4       0   b      1
5       0   b      3
6       1   b      1
7       1   b      3
8       5   b      1
9       5   b      3
10      3   c    NaN
```

Many-to-many joins form the Cartesian product of the rows. Since there were 3 'b' rows in the left DataFrame and 2 in the right one, there are 6 'b' rows in the result. The join method only affects the distinct key values appearing in the result:

```
In [30]: pd.merge(df1, df2, how='inner')
Out[30]:
   data1 key  data2
0      2   a      0
1      2   a      2
2      4   a      0
3      4   a      2
4      0   b      1
5      0   b      3
6      1   b      1
7      1   b      3
8      5   b      1
9      5   b      3
```

To merge with multiple keys, pass a list of column names:

```
In [31]: left = DataFrame({'key1': ['foo', 'foo', 'bar'],
   ....:                   'key2': ['one', 'two', 'one'],
   ....:                   'lval': [1, 2, 3]})

In [32]: right = DataFrame({'key1': ['foo', 'foo', 'bar', 'bar'],
   ....:                    'key2': ['one', 'one', 'one', 'two'],
   ....:                    'rval': [4, 5, 6, 7]})

In [33]: pd.merge(left, right, on=['key1', 'key2'], how='outer')
Out[33]:
  key1 key2  lval  rval
0  bar  one     3     6
1  bar  two   NaN     7
2  foo  one     1     4
3  foo  one     1     5
4  foo  two     2   NaN
```

A last issue to consider in merge operations is the treatment of overlapping column names. While you can address the overlap manually (see the later section on renaming axis labels), merge has a suffixes option for specifying strings to append to overlapping names in the left and right DataFrame objects:

```
In [34]: pd.merge(left, right, on='key1')
Out[34]:
  key1 key2_x  lval key2_y  rval
0  bar    one     3    one     6
1  bar    one     3    two     7
2  foo    one     1    one     4
3  foo    one     1    one     5
4  foo    two     2    one     4
5  foo    two     2    one     5

In [35]: pd.merge(left, right, on='key1', suffixes=('_left', '_right'))
Out[35]:
  key1 key2_left  lval key2_right  rval
0  bar       one     3        one     6
1  bar       one     3        two     7
2  foo       one     1        one     4
3  foo       one     1        one     5
4  foo       two     2        one     4
5  foo       two     2        one     5
```

See Table 7-1 for an argument reference on merge. Joining on index is the subject of the next section.

Table 7-1. merge function arguments

Argument	Description
`left`	DataFrame to be merged on the left side
`right`	DataFrame to be merged on the right side
how	One of `'inner'`, `'outer'`, `'left'` or `'right'`. `'inner'` by default
on	Column names to join on. Must be found in both DataFrame objects. If not specified and no other join keys given, will use the intersection of the column names in `left` and `right` as the join keys
`left_on`	Columns in `left` DataFrame to use as join keys
`right_on`	Analogous to `left_on` for `left` DataFrame
`left_index`	Use row index in `left` as its join key (or keys, if a MultiIndex)
`right_index`	Analogous to `left_index`
`sort`	Sort merged data lexicographically by join keys; `True` by default. Disable to get better performance in some cases on large datasets
`suffixes`	Tuple of string values to append to column names in case of overlap; defaults to `('_x', '_y')`. For example, if `'data'` in both DataFrame objects, would appear as `'data_x'` and `'data_y'` in result
`copy`	If `False`, avoid copying data into resulting data structure in some exceptional cases. By default always copies

Merging on Index

In some cases, the merge key or keys in a DataFrame will be found in its index. In this case, you can pass `left_index=True` or `right_index=True` (or both) to indicate that the index should be used as the merge key:

```
In [36]: left1 = DataFrame({'key': ['a', 'b', 'a', 'a', 'b', 'c'],
   ....:                    'value': range(6)})

In [37]: right1 = DataFrame({'group_val': [3.5, 7]}, index=['a', 'b'])

In [38]: left1
Out[38]:
  key  value
0   a      0
1   b      1
2   a      2
3   a      3
4   b      4
5   c      5

In [39]: right1
Out[39]:
   group_val
a        3.5
b        7.0

In [40]: pd.merge(left1, right1, left_on='key', right_index=True)
Out[40]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
```

Since the default merge method is to intersect the join keys, you can instead form the union of them with an outer join:

```
In [41]: pd.merge(left1, right1, left_on='key', right_index=True, how='outer')
Out[41]:
  key  value  group_val
0   a      0        3.5
2   a      2        3.5
3   a      3        3.5
1   b      1        7.0
4   b      4        7.0
5   c      5        NaN
```

Concatenating Along an Axis

Another kind of data combination operation is alternatively referred to as concatenation, binding, or stacking. NumPy has a concatenate function for doing this with raw NumPy arrays:

```
In [58]: arr = np.arange(12).reshape((3, 4))

In [59]: arr
Out[59]:
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])

In [60]: np.concatenate([arr, arr], axis=1)
Out[60]:
array([[ 0,  1,  2,  3,  0,  1,  2,  3],
       [ 4,  5,  6,  7,  4,  5,  6,  7],
       [ 8,  9, 10, 11,  8,  9, 10, 11]])
```

In the context of pandas objects such as Series and DataFrame, having labeled axes enable you to further generalize array concatenation.

The concat function in pandas provides a consistent way to address each of these concerns. I'll give a number of examples to illustrate how it works. Suppose we have three Series with no index overlap:

```
In [61]: s1 = Series([0, 1], index=['a', 'b'])

In [62]: s2 = Series([2, 3, 4], index=['c', 'd', 'e'])

In [63]: s3 = Series([5, 6], index=['f', 'g'])
```

Calling concat with these object in a list glues together the values and indexes:

```
In [64]: pd.concat([s1, s2, s3])
Out[64]:
a    0
b    1
c    2
d    3
e    4
f    5
g    6
```

By default concat works along `axis=0`, producing another Series. If you pass `axis=1`, the result will instead be a DataFrame (`axis=1` is the columns):

```
In [65]: pd.concat([s1, s2, s3], axis=1)
Out[65]:
     0   1   2
a    0 NaN NaN
b    1 NaN NaN
c  NaN   2 NaN
d  NaN   3 NaN
e  NaN   4 NaN
f  NaN NaN   5
g  NaN NaN   6
```

Pivot

pandas.pivot(index, columns, values) function produces pivot table based on 3 columns of the DataFrame. Uses unique values from index / columns and fills with values.

Parameters:
index[ndarray] : *Labels to use to make new frame's index*
columns[ndarray] : *Labels to use to make new frame's columns*
values[ndarray] : *Values to use for populating new frame's values*
Returns: *Reshaped DataFrame*

Pivot Table

A *pivot table* is a data summarization tool frequently found in spreadsheet programs and other data analysis software. It aggregates a table of data by one or more keys, arranging the data in a rectangle with some of the group keys along the rows and some along the columns. Pivot tables in Python with pandas are made possible using the groupby facility described in this chapter combined with reshape operations utilizing hierarchical indexing. DataFrame has a `pivot_table` method, and additionally there is a top-level `pandas.pivot_table` function. In addition to providing a convenience interface to `groupby`, `pivot_table` also can add partial totals, also known as *margins*.

pandas.pivot_table(data, values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All')

Create a spreadsheet-style pivot table as a DataFrame.
Levels in the pivot table will be stored in MultiIndex objects (hierarchical indexes) on the index and columns of the result DataFrame.

Parameters:
data : *DataFrame*
values : *column to aggregate, optional*
index: *column, Grouper, array, or list of the previous*
columns: *column, Grouper, array, or list of the previous*
aggfunc: *function, list of functions, dict, default numpy.mean*

fill_value[scalar, default None] : *Value to replace missing values with*
margins[boolean, default False] : *Add all row / columns (e.g. for subtotal / grand totals)*
dropna[boolean, default True] : *Do not include columns whose entries are all NaN*
margins_name[string, default 'All'] : *Name of the row / column that will contain the totals when margins is True.*

Returns: *DataFrame*

Table 9-2. pivot_table options

Function name	Description
`values`	Column name or names to aggregate. By default aggregates all numeric columns
`rows`	Column names or other group keys to group on the rows of the resulting pivot table
`cols`	Column names or other group keys to group on the columns of the resulting pivot table
`aggfunc`	Aggregation function or list of functions; `'mean'` by default. Can be any function valid in a groupby context
`fill_value`	Replace missing values in result table
`margins`	Add row/column subtotals and grand total, False by default

SciPy

With NumPy we can achieve fast solutions with simple coding. Where does SciPy come into the picture? It's a package that utilizes NumPy arrays and manipulations to take on standard problems that scientists and engineers commonly face: integration, determining a function's maxima or minima, finding eigenvectors for large sparse matrices, testing whether two distributions are the same, and much more.

Introduction to:

- **Optimization and Minimization**
- **Interpolation**
- **Integration**
- **Statistics**

Optimization and Minimization

- In mathematics, computer science and economics, an **optimization problem** is the problem of finding the *best* solution from all feasible solutions.
- Optimization problems can be divided into two categories, depending on whether the variables are continuous or discrete:
- An optimization problem with discrete variables is known as a *discrete optimization*
- A problem with continuous variables is known as a *continuous optimization*, in which an optimal value from a continuous function must be found.

The standard form of a Minimization Problem

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & f(x) \\ \text{subject to} \quad & g_i(x) \leq 0, \quad i = 1, \ldots, m \\ & h_j(x) = 0, \quad j = 1, \ldots, p \end{aligned}$$

where

- $f : \mathbb{R}^n \to \mathbb{R}$ is the **objective function** to be minimized over the n-variable vector x,
- $g_i(x) \leq 0$ are called **inequality constraints**
- $h_j(x) = 0$ are called **equality constraints**, and
- $m \geq 0$ and $p \geq 0$.

If $m = p = 0$, the problem is an unconstrained optimization problem. By convention, the standard form defines a **minimization problem**. A **maximization problem** can be treated by negating the objective function.

classic examples are performing linear regression, finding a function's minimum and maximum values, determining the root of a function, and finding where two functions intersect.

Many important applied problems involve finding the best way to accomplish some task. Often this involves finding the maximum or minimum value of some function: the minimum time to make a certain journey, the minimum cost for doing a task, the maximum power that can be generated by a device, and so on. Many of these problems can be solved by finding the appropriate function and then using techniques of calculus to find the maximum or the minimum value required.

Generally such a problem will have the following mathematical form: Find the largest (or smallest) value of $f(x)$ when $a \leq x \leq b$. Sometimes a or b are infinite, but frequently the real world imposes some ***constraint*** on the values that x may have.

Linear regression

```
from scipy.optimize import curve_fit
```

Finding the roots of an equation

```
from scipy.optimize import fsolve
```

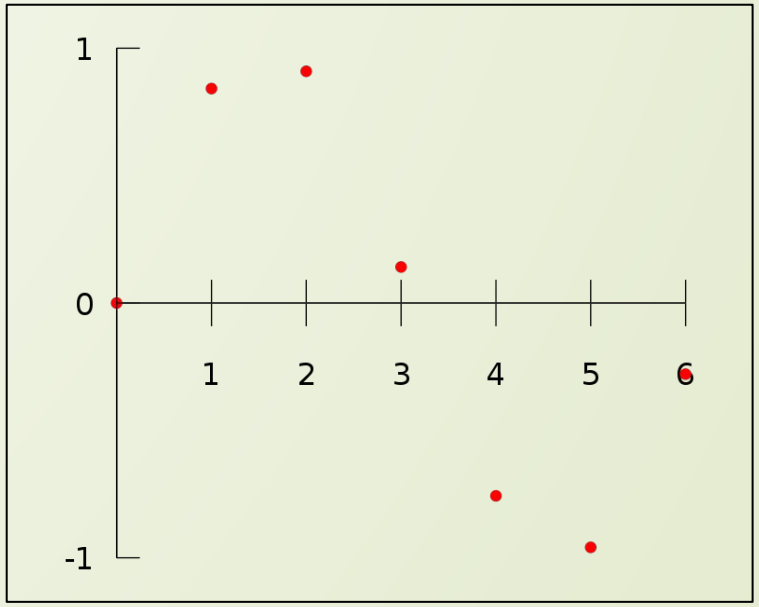
Interpolation

Data containing information has a functional form.

Interpolation is a type of estimation, a method of constructing new data points within the range of known data points.

Example:

x	f(x)
0	0
1	0.8415
2	0.9093
3	0.1411
4	-0.7568
5	-0.9589
6	-0.2794



Interpolation provides a means of estimating the function at intermediate points, such as x=2.5

Basic methods of Interpolation

(1) Fit one function to an entire dataset

```
from scipy.interpolate import interp1d
```

(2) Fit different parts of the dataset with several functions where the joints of each function are joined smoothly – **Spline Interpolation**

```
from scipy.interpolate import UnivariateSpline
```

```
from scipy.interpolate import SmoothBivariateSpline as SBS
```

Integration

Integration is a crucial tool in math and science, as differentiation and integration are the two key components of calculus. Given a curve from a function or a dataset, we can calculate the area below it. In the traditional classroom setting we would integrate a function analytically, but data in the research setting is rarely given in this form, and we need to approximate its definite integral.

SciPy has a range of different functions to integrate equations and data.

Analytic Integration

A function is given that needs to be integrated

$$\int_0^3 \cos^2(e^x)\,dx$$

```
from scipy.integrate import quad
```

Numerical Integration

Data is given instead of an equation

```
from scipy.integrate import quad, trapz
```

The `quad` integrator can only work with a callable function, whereas `trapz` is a numerical integrator that utilizes data points.

Statistics

In NumPy there are basic statistical functions like `mean`, `std`, `median`, `argmax`, and `argmin`. Moreover, the `numpy.arrays` have built-in methods that allow us to use most of the NumPy statistics easily.

```
import numpy as np

# Constructing a random array with 1000 elements
x = np.random.randn(1000)

# Calculating several of the built-in methods
# that numpy.array has
mean = x.mean()
std = x.std()
var = x.var()
```

For quick calculations these methods are useful, but more is usually needed for quantitative research. SciPy offers an extended collection of statistical tools such as distributions (continuous or discrete) and functions.

Continuous and Discrete Distributions

There are roughly 80 continuous distributions and over 10 discrete distributions. Twenty of the continuous functions are shown in Figure 3-12 as probability density functions (PDFs) to give a visual impression of what the `scipy.stats` package provides. These distributions are useful as random number generators, similar to the functions found in `numpy.random`. Yet the rich variety of functions SciPy provides stands in contrast to the `numpy.random` functions, which are limited to uniform and Gaussian-like distributions.

When we call a distribution from `scipy.stats`, we can extract its information in several ways: probability density functions (PDFs), cumulative distribution functions (CDFs), random variable samples (RVSs), percent point functions (PPFs), and more.

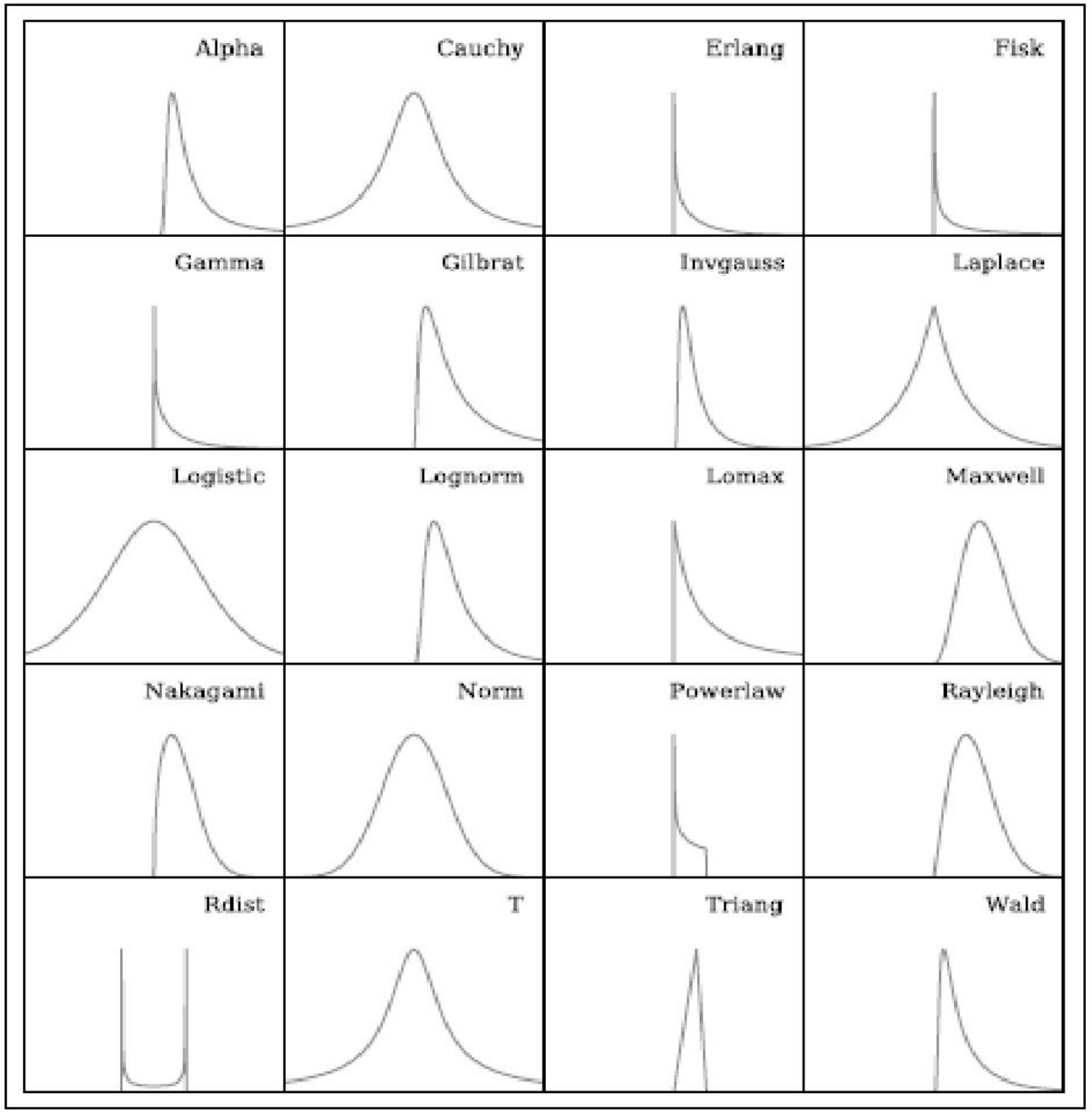


Figure 3-12. A sample of 20 continuous distributions in SciPy.

For Normal distribution

```
import scipy.stats import norm
```

For Geometric distribution

```
from scipy.stats import geom
```

Functions

There are more than 60 statistical functions in SciPy, which can be overwhelming to digest if you simply are curious about what is available. The best way to think of the statistics functions is that they either describe or test samples—for example, the frequency of certain values or the Kolmogorov-Smirnov test, respectively.

```
from scipy import stats
```

In the `stats` package, there are a number of functions such as `kstest` and `normaltest` that test samples. These distribution tests can be very helpful in determining whether a sample comes from some particular distribution or not.

Researchers commonly use descriptive functions for statistics. Some descriptive functions that are available in the `stats` package include the geometric mean (`gmean`), the skewness of a sample (`skew`), and the frequency of values in a sample (`itemfreq`).