Lecture 9: Data Cleaning and Preprocessing

Data Science, DST, UIC

The difference between data found in many tutorials and data in the real world is that real-world data is **rarely clean and homogeneous**. Real-world data is often incomplete, inconsistent, and/or lacking in certain behaviors or trends, and is likely to contain many errors. Data cleaning and preprocessing are proven methods of resolving such issues. In this lecture, we will introduce several common tasks in data cleaning and preprocessing, including handling missing data, combining datasets and data transformation.

1 Handling Missing Data

Generally, most data will have some missing values. There could be various reasons for this: the source system which collects the data might not have collected the values or the values may never have existed. To make matters even more complicated, different data sources may indicate missing data in different ways.

A number of schemes have been developed to indicate the presence of missing data in a table or DataFrame. Generally, they revolve around one of two strategies:

- using a *mask* that globally indicates missing values, e.g., the mask might be an entirely separate Boolean array, or
- choosing a *sentinel value* that indicates a missing entry, e.g, NaN, -9999 or some data-specific convention.

None of these approaches is without trade-offs:

- Use a separate mask array requires allocation of an additional Boolean array, which adds overhead in both storage and computation.
- A sentinel value reduces the range of valid values that can be represented, and may require extra (often non-optimized) logic in CPU and GPU arithmetic. Common special values like NaN are not available for all data types.

1.1 Missing Data in Pandas

The way in which Pandas handles missing values is constrained by its reliance on the NumPy package, which does not have a built-in notion of NA values for nonfloating-point data types. Pandas chose to use sentinels for missing data, and further chose to use two already-existing Python null values: the special floating-point `NaN` value, and the Python `None` object. This choice has some side effects, as we will see, but in practice ends up being a good compromise in most cases of interest.

`None` : Pythonic missing data

The first sentinel value used by Pandas is `None` , a Python singleton object that is often used for missing data in Python code. Because it is a Python object, `None` cannot be used in any arbitrary NumPy/Pandas array, but only in arrays with data type `'object'` (i.e., arrays of Python objects):

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

In [2]:
```
vals1 = np.array([1, None, 3, 4])
vals1
```

Out[2]:
```
array([1, None, 3, 4], dtype=object)
```

This `dtype=object` means that the best common type representation NumPy could infer for the contents of the array is that they are Python objects. While this kind of object array is useful for some purposes, any operations on the data will be done at the Python level, with much more overhead than the typically fast operations seen for arrays with native types:

In [3]:
```
for dtype in ['object', 'int']:
    print("dtype =", dtype)
    # arrange: return evenly spaced values within a given interval.
    %timeit np.arange(1E6, dtype=dtype).sum()
    print()
```

```
dtype = object
62 ms ± 2.25 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

dtype = int
2.37 ms ± 206 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

The use of Python objects in an array also means that if you perform aggregations like `sum()` or `min()` across an array with a `None` value, you will generally get an error:

In [4]:
```
# vals1.sum()
```

This reflects the fact that addition between an integer and `None` is undefined.

`NaN` : Missing numerical data

The other missing data representation, `NaN` (acronym for *Not a Number*), is different; it is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation:

In [5]:
```
vals2 = np.array([1, np.nan, 3, 4])
vals2.dtype
```

Out[5]:
```
dtype('float64')
```

Notice that NumPy chose a native floating-point type for this array: this means that unlike the object array from before, this array supports fast operations pushed into compiled code. You should be aware that `NaN` is a bit like a data virus–it infects any other object it touches. Regardless of the operation, the result of arithmetic with `NaN` will be another `NaN` :

In [6]:
```
print(1 + np.nan)
print(0*np.nan)
```

```
nan
nan
```

Note that this means that aggregates over the values are well defined (i.e., they don't result in an error) but not always useful:

In [7]:
```
vals2.sum(), vals2.min(), vals2.max()
```

Out[7]:
```
(nan, nan, nan)
```

NumPy does provide some special aggregations that will ignore these missing values:

In [8]:
```
# Return the sum of array elements over a given axis treating Not a Numbers (NaNs) as zero.
np.nansum(vals2), np.nanmin(vals2), np.nanmax(vals2)
```

Out[8]:
```
(8.0, 1.0, 4.0)
```

Keep in mind that `NaN` is specifically a floating-point value; there is no equivalent NaN value for integers, strings, or other types.

NaN and None in Pandas

`NaN` and `None` both have their place, and Pandas is built to handle the two of them **nearly interchangeably**, converting between them where appropriate:

In [9]:
```
pd.Series([1, np.nan, 2, None])
```

Out[9]:
```
0    1.0
1    NaN
2    2.0
3    NaN
dtype: float64
```

For types that don't have an available sentinel value, Pandas automatically type-casts when NA values are present. For example, if we set a value in an integer array to `np.nan`, it will automatically be upcast to a floating-point type to accommodate the NA:

In [10]:
```
x = pd.Series(range(2), dtype=int)
x
```

Out[10]:
```
0    0
1    1
dtype: int32
```

In [11]:
```
x[0] = None
x
```

Out[11]:
```
0    NaN
1    1.0
dtype: float64
```

Notice that in addition to casting the integer array to floating point, Pandas automatically converts the `None` to a `NaN` value.

The following table lists the upcasting conventions in Pandas when NA values are introduced:

Typeclass	Conversion When Storing NAs	NA Sentinel Value
`floating`	No change	`np.nan`
`object`	No change	`None` or `np.nan`
`integer`	Cast to `float64`	`np.nan`
`boolean`	Cast to `object`	`None` or `np.nan`

In [12]:
```
y = pd.Series([np.nan, False, True, True])
y
```

Out[12]:
```
0      NaN
1    False
2     True
3     True
dtype: object
```

1.2 Operating on Null Values

As we have seen, Pandas treats `None` and `NaN` as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful methods for detecting, removing, and replacing null values in Pandas data structures. They are:

- `isnull()` : Generate a boolean mask indicating missing values
- `notnull()` : Opposite of `isnull()`
- `dropna()` : Return a filtered version of the data
- `fillna()` : Return a copy of the data with missing values filled or imputed

We will conclude this section with a brief exploration and demonstration of these routines.

Detecting null values

Pandas data structures have two useful methods for detecting null data: `isnull()` and `notnull()` . Either one will return a Boolean mask over the data. For example:

In [13]:
```
data = pd.Series([1, np.nan, 'hello', None])
data
```

Out[13]:
```
0        1
1      NaN
2    hello
3     None
dtype: object
```

In [14]:
```
data.isnull()
```

Out[14]:
```
0    False
1     True
2    False
3     True
dtype: bool
```

As mentioned in data indexing and selection, Boolean masks can be used directly as a `Series` or `DataFrame` index:

In [15]:
```
data[data.notnull()]
```

Out[15]:
```
0        1
2    hello
dtype: object
```

The `isnull()` and `notnull()` methods produce similar Boolean results for `DataFrame` s.

Dropping null values

In addition to the masking used before, there are the convenience methods, `dropna()` (which removes NA values) and `fillna()` (which fills in NA values). For a `Series`, the result is straightforward:

In [16]:
```
data.dropna()
```

Out[16]:
```
0        1
2    hello
dtype: object
```

For a `DataFrame`, there are more options. Consider the following `DataFrame`:

In [17]:
```
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])
df
```

Out[17]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

We cannot drop single values from a `DataFrame`; we can only drop full rows or full columns. Depending on the application, you might want one or the other, so `dropna()` gives a number of options for a `DataFrame`.

By default, `dropna()` will drop all rows in which *any* null value is present:

In [18]:
```
df.dropna()
```

Out[18]:

	0	1	2
1	2.0	3.0	5

Alternatively, you can drop NA values along a different axis; `axis=1` or `axis = 'columns'` drops all columns containing a null value:

In [19]:
```
df.dropna(axis='columns')
```

Out[19]:

	2
0	2
1	5
2	6

But this drops some good data as well; you might rather be interested in dropping rows or columns with *all* NA values, or a majority of NA values. This can be specified through the `how` or `thresh` parameters, which allow fine control of the number of nulls to allow through.

The default is `how='any'`, such that a row or column (depending on the `axis` keyword) containing *any* null value will be dropped. You can also specify `how='all'`, which will only drop rows/columns that are *all* null values:

In [20]:
```
df[3] = np.nan
df
```

Out[20]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

In [21]:
```
df.dropna(axis=1, how='all')
```

Out[21]:

	0	1	2
0	1.0	NaN	2
1	2.0	3.0	5
2	NaN	4.0	6

For finer-grained control, the `thresh` parameter lets you specify a **minimum number of non-null values for the row/column to be kept**:

In [22]:
```
df.dropna(axis='rows', thresh=3)
```

Out[22]:

	0	1	2	3
1	2.0	3.0	5	NaN

Here the first and last row have been dropped, because they contain only two non-null values.

Filling null values

Sometimes rather than dropping NA values, you'd rather replace them with a valid value. This value might be a single number like zero, or it might be some sort of imputation or interpolation from the good values. You could do this **in-place** using the `isnull()` method as a mask, but because it is such a common operation, Pandas provides the `fillna()` method, which returns **a copy of** the array with the null values replaced.

Consider the following `Series`:

In [23]:
```
data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
data
```

Out[23]:
```
a    1.0
b    NaN
c    2.0
d    NaN
e    3.0
dtype: float64
```

We can fill NA entries with a single value, such as zero:

In [24]:
```
data.fillna(0)
```

Out[24]:
```
a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

We can specify a forward-fill to propagate the previous value forward:

In [25]:
```
# forward-fill
data.fillna(method='ffill')
```

Out[25]:
```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

Or we can specify a back-fill to propagate the next values backward:

In [26]:
```
# back-fill
data.fillna(method='bfill')
```

Out[26]:
```
a    1.0
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```

For `DataFrame` s, the options are similar, but we can also specify an `axis` along which the fills take place:

In [27]:
```
df
```

Out[27]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	NaN	4.0	6	NaN

In [28]:
```
df.fillna(method='ffill', axis=1)
```

Out[28]:

	0	1	2	3
0	1.0	1.0	2.0	2.0
1	2.0	3.0	5.0	5.0
2	NaN	4.0	6.0	6.0

In [29]:
```
df.fillna(0)
```

Out[29]:

	0	1	2	3
0	1.0	0.0	2	0.0
1	2.0	3.0	5	0.0
2	0.0	4.0	6	0.0

Notice that if a previous value is not available during a forward fill, the NA value remains.

In [30]:
```
df.fillna(method='ffill', axis=0)
```

Out[30]:

	0	1	2	3
0	1.0	NaN	2	NaN
1	2.0	3.0	5	NaN
2	2.0	4.0	6	NaN

2 Combining Datasets

Some of the most interesting studies of data come from combining different data sources. These operations can involve anything from very straightforward concatenation of two different datasets, to more complicated database-style joins and merges that correctly handle any overlaps between the datasets. Pandas `Series` and `DataFrame` s are built with this type of operation in mind, and includes functions and methods that make this sort of data wrangling fast and straightforward.

Here we'll take a look at

- simple **concatenation** of `Series` and `DataFrame` s with the `pd.concat` function
- more sophisticated in-memory **merges** and **joins** implemented in Pandas.

For convenience, we'll define this function which creates a `DataFrame` of a particular form that will be useful below:

In [31]:
```
def make_df(cols, ind):
    """Quickly make a DataFrame"""
    data = {c: [str(c) + str(i) for i in ind]
            for c in cols}
    return pd.DataFrame(data, ind)

# example DataFrame
make_df('ABC', range(3))
```

Out[31]:

	A	B	C
0	A0	B0	C0
1	A1	B1	C1
2	A2	B2	C2

In addition, we'll create a quick class that allows us to display multiple `DataFrame` s side by side. The code makes use of the special `_repr_html_` method, which IPython uses to implement its rich object display:

- If you add a `_repr_html_` method returning a string of HTML to any Python class, Jupyter notebooks will render that HTML inline to represent that object. (Note that these are surrounded by single, not double underscores.)

In [32]:
```python
class display():
    """Display HTML representation of multiple objects"""
    template = """<div style="float: left; padding: 10px;">
    <p style='font-family:"Courier New", Courier, monospace'>{0}</p>{1}
    </div>"""
    def __init__(self, *args):
        self.args = args

    def _repr_html_(self):
        # str.join(sequence): use str to join the given sequence
        # eval(expression): return the result of the expression
        # dataframe object has _repr_html_ method
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)
```

In [33]:
```python
dfa = make_df([0,1],['a','b'])
df._repr_html_()
```

Out[33]:
```
'<div>\n<style scoped>\n    .dataframe tbody tr th:only-of-type {\n        vertical-align: middle;\n    }\n\n    .dataframe tbody tr th {\n        vertical-align: top;\n    }\n\n    .dataframe thead th {\n        text-align: right;\n    }\n</style>\n<table border="1" class="dataframe">\n  <thead>\n    <tr style="text-align: right;">\n      <th></th>\n      <th>0</th>\n      <th>1</th>\n      <th>2</th>\n      <th>3</th>\n    </tr>\n  </thead>\n  <tbody>\n    <tr>\n      <th>0</th>\n      <td>1.0</td>\n      <td>NaN</td>\n      <td>2</td>\n      <td>NaN</td>\n    </tr>\n    <tr>\n      <th>1</th>\n      <td>2.0</td>\n      <td>3.0</td>\n      <td>5</td>\n      <td>NaN</td>\n    </tr>\n    <tr>\n      <th>2</th>\n      <td>NaN</td>\n      <td>4.0</td>\n      <td>6</td>\n      <td>NaN</td>\n    </tr>\n  </tbody>\n</table>\n</div>'
```

In [34]:
```python
dfb = make_df([0,1],['c','d'])
display('dfa','dfb')
```

Out[34]:

dfa

	0	1
a	0a	1a
b	0b	1b

dfb

	0	1
c	0c	1c
d	0d	1d

The use of this will become clearer as we continue our discussion in the following section.

2.1 Concat and Append

Simple Concatenation with `pd.concat`

Pandas has a function, `pd.concat()`, which has a similar syntax to `np.concatenate` but contains a number of options that we'll discuss:

```python
# Signature in Pandas v0.18
pd.concat(objs, axis=0, join='outer', join_axes=None, ignore_index=False,
          keys=None, levels=None, names=None, verify_integrity=False,
          copy=True)
```

`pd.concat()` can be used for a simple concatenation of `Series` or `DataFrame` objects, just as `np.concatenate()` can be used for simple concatenations of arrays:

In [35]:
```
ser1 = pd.Series(['A', 'B', 'C'], index=[1, 2, 3])
ser2 = pd.Series(['D', 'E', 'F'], index=[4, 5, 6])
pd.concat([ser1, ser2])
```

Out[35]:
```
1    A
2    B
3    C
4    D
5    E
6    F
dtype: object
```

In [36]:
```
pd.concat([ser1,ser2], axis='columns')
```

Out[36]:

	0	1
1	A	NaN
2	B	NaN
3	C	NaN
4	NaN	D
5	NaN	E
6	NaN	F

It also works to concatenate higher-dimensional objects, such as `DataFrame` s:

In [37]:
```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
display('df1', 'df2', 'pd.concat([df1, df2])')
```

Out[37]:

df1

	A	B
1	A1	B1
2	A2	B2

df2

	A	B
3	A3	B3
4	A4	B4

pd.concat([df1, df2])

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

By default, the concatenation takes place row-wise within the `DataFrame` (i.e., `axis=0`). Like `np.concatenate` , `pd.concat` allows specification of an axis along which concatenation will take place. Consider the following example:

In [38]:
```
df3 = make_df('AB', [0, 1])
df4 = make_df('CD', [0, 1])
display('df3', 'df4', "pd.concat([df3, df4], axis='columns')")
```

Out[38]:

df3

	A	B
0	A0	B0
1	A1	B1

df4

	C	D
0	C0	D0
1	C1	D1

pd.concat([df3, df4], axis='columns')

	A	B	C	D
0	A0	B0	C0	D0
1	A1	B1	C1	D1

We could have equivalently specified `axis=1` ; here we've used the more intuitive `axis='columns'` .

Duplicate indices

One important difference between `np.concatenate` and `pd.concat` is that Pandas concatenation *preserves indices*, even if the result will have duplicate indices! Consider this simple example:

In [39]:
```
x = make_df('AB', [0, 1])
y = make_df('AB', [2, 3])
y.index = x.index  # make duplicate indices!
display('x', 'y', 'pd.concat([x, y])')
```

Out[39]:

x

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

pd.concat([x, y])

	A	B
0	A0	B0
1	A1	B1
0	A2	B2
1	A3	B3

Notice the repeated indices in the result. While this is valid within `DataFrame` s, the outcome is often undesirable. `pd.concat()` gives us a few ways to handle it.

Catching the repeats as an error

If you'd like to simply verify that the indices in the result of `pd.concat()` do not overlap, you can specify the `verify_integrity` flag. With this set to True, the concatenation will raise an exception if there are duplicate indices. Here is an example, where for clarity we'll catch and print the error message:

In [40]:
```
try:
    pd.concat([x, y], verify_integrity=True)
except ValueError as e:
    print("ValueError:", e)
```

```
ValueError: Indexes have overlapping values: Int64Index([0, 1], dtype='int64')
```

Ignoring the index

Sometimes the index itself does not matter, and you would prefer it to simply be ignored. This option can be specified using the `ignore_index` flag. With this set to true, the concatenation will create a new integer index for the resulting `Series` :

In [41]:
```
display('x', 'y', 'pd.concat([x, y], ignore_index=True)')
```

Out[41]:

x

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

pd.concat([x, y], ignore_index=True)

	A	B
0	A0	B0
1	A1	B1
2	A2	B2
3	A3	B3

Adding MultiIndex keys

Another option is to use the `keys` option to specify a label for the data sources; the result will be a hierarchically indexed series containing the data:

In [42]:
```
display('x', 'y', "pd.concat([x, y], keys=['x', 'y'])")
```

Out[42]:

x

	A	B
0	A0	B0
1	A1	B1

y

	A	B
0	A2	B2
1	A3	B3

pd.concat([x, y], keys=['x', 'y'])

		A	B
x	0	A0	B0
	1	A1	B1
y	0	A2	B2
	1	A3	B3

The result is a multiply indexed `DataFrame`, and we can use the tools discussed in *hierarchical indexing* to transform this data into the representation we're interested in.

In [43]:
```
x_c_y = pd.concat([x,y], keys=['x','y'])
x_c_y
```

Out[43]:

		A	B
x	0	A0	B0
	1	A1	B1
y	0	A2	B2
	1	A3	B3

In [44]:
```
z = x_c_y.unstack()
z
```

Out[44]:

	A		B	
	0	1	0	1
x	A0	A1	B0	B1
y	A2	A3	B2	B3

```
In [45]: # select values with column index 'A' and 1 for each level
z.loc[:, ('A', 1)]
```

```
Out[45]: x    A1
y    A3
Name: (A, 1), dtype: object
```

```
In [46]: # select values with column index 1 on second level
idx = pd.IndexSlice
z.loc[:, idx[:, 1]]
```

Out[46]:

	A	B
	1	1
x	A1	B1
y	A3	B3

Concatenation with joins

In the simple examples we just looked at, we were mainly concatenating `DataFrame` s with shared column names. In practice, data from different sources might have different sets of column names, and `pd.concat` offers several options in this case. Consider the concatenation of the following two `DataFrame` s, which have some (but not all!) columns in common:

```
In [47]: df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
display('df5', 'df6', 'pd.concat([df5, df6], sort=True)')
```

Out[47]:

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

pd.concat([df5, df6], sort=True)

	A	B	C	D
1	A1	B1	C1	NaN
2	A2	B2	C2	NaN
3	NaN	B3	C3	D3
4	NaN	B4	C4	D4

By default, the entries for which no data is available are filled with NA values. To change this, we can specify one of several options for the `join` and `join_axes` parameters of the concatenate function. By default, the join is a union of the input columns (`join='outer'`), but we can change this to an intersection of the columns using `join='inner'` :

```
In [48]: display('df5', 'df6', "pd.concat([df5, df6], join='inner', sort=True)")
```

Out[48]:

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

pd.concat([df5, df6], join='inner', sort=True)

	B	C
1	B1	C1
2	B2	C2
3	B3	C3
4	B4	C4

In [49]:
```
display('df5', 'df6', "pd.concat([df5,df6], join='outer', sort=True)")
```

Out[49]:

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

pd.concat([df5,df6], join='outer', sort=True)

	A	B	C	D
1	A1	B1	C1	NaN
2	A2	B2	C2	NaN
3	NaN	B3	C3	D3
4	NaN	B4	C4	D4

Another option is to directly specify the index of the remaininig colums using the `join_axes` argument, which takes a list of index objects. Here we'll specify that the returned columns should be the same as those of the first input:

In [50]:
```
display('df5', 'df6',
        "pd.concat([df5, df6], join_axes=[df5.columns])")
```

Out[50]:

df5

	A	B	C
1	A1	B1	C1
2	A2	B2	C2

df6

	B	C	D
3	B3	C3	D3
4	B4	C4	D4

pd.concat([df5, df6], join_axes=[df5.columns])

	A	B	C
1	A1	B1	C1
2	A2	B2	C2
3	NaN	B3	C3
4	NaN	B4	C4

The combination of options of the `pd.concat` function allows a wide range of possible behaviors when joining two datasets; keep these in mind as you use these tools for your own data.

The `append()` method

Because direct array concatenation is so common, `Series` and `DataFrame` objects have an `append` method that can accomplish the same thing in fewer keystrokes. For example, rather than calling `pd.concat([df1, df2])`, you can simply call `df1.append(df2)`:

In [51]:
```
display('df1', 'df2', "df1.append(df2)")
# Append rows of `other` to the end of this frame
```

Out[51]:

df1

	A	B
1	A1	B1
2	A2	B2

df2

	A	B
3	A3	B3
4	A4	B4

df1.append(df2)

	A	B
1	A1	B1
2	A2	B2
3	A3	B3
4	A4	B4

2.2 Merge and Join

One essential feature offered by Pandas is its high-performance, in-memory join and merge operations. If you have ever worked with databases, you should be familiar with this type of data interaction. The main interface for this is the `pd.merge` function, and we'll see few examples of how this can work in practice.

Relational Algebra

- The behavior implemented in `pd.merge()` is a subset of what is known as *relational algebra*, which is a formal set of rules for manipulating relational data, and forms the conceptual foundation of operations available in most databases.
- The strength of the relational algebra approach is that it proposes several primitive operations, which become the building blocks of more complicated operations on any dataset.
- Pandas implements several of these fundamental building-blocks in the `pd.merge()` function and the related `join()` method of `Series` and `Dataframe` s.

As we will see, these let you efficiently link data from different sources.

Categories of Joins

The `pd.merge()` function implements a number of types of joins:

- *one-to-one*
- *many-to-one* and
- *many-to-many* joins.

All three types of joins are accessed via an identical call to the `pd.merge()` interface; the type of join performed depends on the form of the input data. Here we will show simple examples of the three types of merges, and discuss detailed options further below.

One-to-one joins

Perhaps the simplest type of merge expresion is the one-to-one join, which is in many ways very similar to the column-wise concatenation. As a concrete example, consider the following two `DataFrames` which contain information on several employees in a company:

In [52]:
```
df7 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df8 = pd.DataFrame({'employee': ['Sue', 'Bob', 'Jake','Tom'],
                    'hire_date': [2004, 2008, 2012, 2009]})
display('df7', 'df8')
```

Out[52]:

df7

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df8

	employee	hire_date
0	Sue	2004
1	Bob	2008
2	Jake	2012
3	Tom	2009

To combine this information into a single `DataFrame`, we can use the `pd.merge()` function:

In [53]:
```
df9 = pd.merge(df7, df8)
df9
```

Out[53]:

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Sue	HR	2004

The `pd.merge()` function recognizes that each `DataFrame` has an "employee" column, and automatically joins using this column as a key.

The result of the merge is a new `DataFrame` that combines the information from the two inputs. Notice that the order of entries in each column is not necessarily maintained: in this case, the order of the "employee" column differs between `df1` and `df2`, and the `pd.merge()` function correctly accounts for this. Additionally, keep in mind that the merge in general discards the index, except in the special case of merges by index (see the `left_index` and `right_index` keywords, discussed momentarily).

Many-to-one joins

Many-to-one joins are joins in which one of the two key columns contains duplicate entries. For the many-to-one case, the resulting `DataFrame` will preserve those duplicate entries as appropriate. Consider the following example of a many-to-one join:

In [54]:
```
df10 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                     'supervisor': ['Carly', 'Guido', 'Steve']})
display('df7', 'df10', 'pd.merge(df7, df10)')
```

Out[54]:

df7

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df10

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido
2	HR	Steve

pd.merge(df7, df10)

	employee	group	supervisor
0	Bob	Accounting	Carly
1	Jake	Engineering	Guido
2	Lisa	Engineering	Guido
3	Sue	HR	Steve

The resulting `DataFrame` has an aditional column with the "supervisor" information, where the information is repeated in one or more locations as required by the inputs.

Many-to-many joins

Many-to-many joins are a bit confusing conceptually, but are nevertheless well defined. If the key column in both the left and right array contains duplicates, then the result is a many-to-many merge. This will be perhaps most clear with a concrete example. Consider the following, where we have a `DataFrame` showing one or more skills associated with a particular group. By performing a many-to-many join, we can recover the skills associated with any individual person:

In [55]:
```python
df11 = pd.DataFrame({'group': ['Accounting', 'Accounting',
                              'Engineering', 'Engineering', 'HR', 'HR'],
                    'skills': ['math', 'spreadsheets', 'coding', 'linux',
                               'spreadsheets', 'organization']})
display('df7', 'df11', "pd.merge(df7, df11)")
```

Out[55]:

df7

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df11

	group	skills
0	Accounting	math
1	Accounting	spreadsheets
2	Engineering	coding
3	Engineering	linux
4	HR	spreadsheets
5	HR	organization

pd.merge(df7, df11)

	employee	group	skills
0	Bob	Accounting	math
1	Bob	Accounting	spreadsheets
2	Jake	Engineering	coding
3	Jake	Engineering	linux
4	Lisa	Engineering	coding
5	Lisa	Engineering	linux
6	Sue	HR	spreadsheets
7	Sue	HR	organization

These three types of joins can be used with other Pandas tools to implement a wide array of functionality. But in practice, datasets are rarely as clean as the one we're working with here. In the following section we'll consider some of the options provided by `pd.merge()` that enable you to tune how the join operations work.

Specification of the Merge Key

We've already seen the default behavior of `pd.merge()` : it looks for one or more matching column names between the two inputs, and uses this as the key. However, often the column names will not match so nicely, and `pd.merge()` provides a variety of options for handling this.

The `on` keyword

Most simply, you can explicitly specify the name of the key column using the `on` keyword, which takes a column name or a list of column names:

In [56]:
```python
display('df7', 'df8', "pd.merge(df7, df8, on='employee')")
```

Out[56]:

df7

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df8

	employee	hire_date
0	Sue	2004
1	Bob	2008
2	Jake	2012
3	Tom	2009

pd.merge(df7, df8, on='employee')

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Sue	HR	2004

This option works only if both the left and right `DataFrame` s have the specified column name.

The `left_on` and `right_on` keywords

At times you may wish to merge two datasets with different column names; for example, we may have a dataset in which the employee name is labeled as "name" rather than "employee". In this case, we can use the `left_on` and `right_on` keywords to specify the two column names:

In [57]:
```
df12 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'salary': [70000, 80000, 120000, 90000]})
display('df7', 'df12', 'pd.merge(df7, df12, left_on="employee", right_on="name")')
```

Out[57]:

df7

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df12

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

pd.merge(df7, df12, left_on="employee", right_on="name")

	employee	group	name	salary
0	Bob	Accounting	Bob	70000
1	Jake	Engineering	Jake	80000
2	Lisa	Engineering	Lisa	120000
3	Sue	HR	Sue	90000

The result has a redundant column that we can drop if desired–for example, by using the `drop()` method of `DataFrame` s:

In [58]:
```
pd.merge(df7, df12, left_on="employee", right_on="name").drop('name', axis=1)
```

Out[58]:

	employee	group	salary
0	Bob	Accounting	70000
1	Jake	Engineering	80000
2	Lisa	Engineering	120000
3	Sue	HR	90000

The `left_index` and `right_index` keywords

Sometimes, rather than merging on a column, you would instead like to merge on an index. For example, your data might look like this:

In [59]:
```
df7a = df7.set_index('employee')
df8a = df8.set_index('employee')
display('df7','df8','df7a', 'df8a')
```

Out[59]:

df7

	employee	group
0	Bob	Accounting
1	Jake	Engineering
2	Lisa	Engineering
3	Sue	HR

df8

	employee	hire_date
0	Sue	2004
1	Bob	2008
2	Jake	2012
3	Tom	2009

df7a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df8a

	hire_date
employee	
Sue	2004
Bob	2008
Jake	2012
Tom	2009

You can use the index as the key for merging by specifying the `left_index` and/or `right_index` flags in `pd.merge()` :

In [60]:
```
display('df7a', 'df8a',
        "pd.merge(df7a, df8a, left_index=True, right_index=True)")
```

Out[60]:

df7a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df8a

	hire_date
employee	
Sue	2004
Bob	2008
Jake	2012
Tom	2009

pd.merge(df7a, df8a, left_index=True, right_index=True)

	group	hire_date
employee		
Bob	Accounting	2008
Jake	Engineering	2012
Sue	HR	2004

For convenience, `DataFrame` s implement the `join()` method, which performs a merge that defaults to joining on indices:

In [61]:
```
display('df7a', 'df8a', "df7a.join(df8a)")
```

Out[61]:

df7a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df8a

	hire_date
employee	
Sue	2004
Bob	2008
Jake	2012
Tom	2009

df7a.join(df8a)

	group	hire_date
employee		
Bob	Accounting	2008.0
Jake	Engineering	2012.0
Lisa	Engineering	NaN
Sue	HR	2004.0

If you'd like to mix indices and columns, you can combine `left_index` with `right_on` or `left_on` with `right_index` to get the desired behavior:

In [62]:
```
display('df7a', 'df12', "pd.merge(df7a, df12, left_index=True, right_on='name')")
```

Out[62]:

df7a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df12

	name	salary
0	Bob	70000
1	Jake	80000
2	Lisa	120000
3	Sue	90000

pd.merge(df7a, df12, left_index=True, right_on='name')

	group	name	salary
0	Accounting	Bob	70000
1	Engineering	Jake	80000
2	Engineering	Lisa	120000
3	HR	Sue	90000

All of these options also work with multiple indices and/or multiple columns; the interface for this behavior is very intuitive. For more information on this, see the "Merge, Join, and Concatenate" section (http://pandas.pydata.org/pandas-docs/stable/merging.html) of the Pandas documentation.

Specifying Set Arithmetic for Joins

In all the preceding examples we have glossed over one important consideration in performing a join: the type of set arithmetic used in the join. This comes up when a value appears in one key column but not the other. Consider this example:

In [63]:
```
df13 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                     'food': ['fish', 'beans', 'bread']},
                    columns=['name', 'food'])
df14 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                     'drink': ['wine', 'beer']},
                    columns=['name', 'drink'])
display('df13', 'df14', 'pd.merge(df13, df14)')
```

Out[63]:

df13

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df14

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df13, df14)

	name	food	drink
0	Mary	bread	wine

Here we have merged two datasets that have only a single "name" entry in common: Mary. By default, the result contains the *intersection* of the two sets of inputs; this is what is known as an *inner join*. We can specify this explicitly using the `how` keyword, which defaults to `"inner"`:

In [64]:
```
pd.merge(df13, df14, how='inner')
```

Out[64]:

	name	food	drink
0	Mary	bread	wine

Other options for the `how` keyword are `'outer'`, `'left'`, and `'right'`. An *outer join* returns a join over the union of the input columns, and fills in all missing values with NAs:

In [65]:
```
display('df13', 'df14', "pd.merge(df13, df14, how='outer')")
```

Out[65]:

df13

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df14

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df13, df14, how='outer')

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine
3	Joseph	NaN	beer

The *left join* and *right join* return joins over the left entries and right entries, respectively. For example:

In [66]:
```
display('df13', 'df14', "pd.merge(df13, df14, how='left')")
```

Out[66]:

df13

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df14

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df13, df14, how='left')

	name	food	drink
0	Peter	fish	NaN
1	Paul	beans	NaN
2	Mary	bread	wine

The output rows now correspond to the entries in the left input. Using `how='right'` works in a similar manner.

All of these options can be applied straightforwardly to any of the preceding join types.

Overlapping Column Names: The `suffixes` Keyword

Finally, you may end up in a case where your two input `DataFrame` s have conflicting column names. Consider this example:

In [67]:
```
df15 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'rank': [1, 2, 3, 4]})
df16 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                     'rank': [3, 1, 4, 2]})
display('df15', 'df16', 'pd.merge(df15, df16, on="name")')
```

Out[67]:

df15

	name	rank
0	Bob	1
1	Jake	2
2	Lisa	3
3	Sue	4

df16

	name	rank
0	Bob	3
1	Jake	1
2	Lisa	4
3	Sue	2

pd.merge(df15, df16, on="name")

	name	rank_x	rank_y
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

Because the output would have two conflicting column names, the merge function automatically appends a suffix `_x` or `_y` to make the output columns unique. If these defaults are inappropriate, it is possible to specify a custom suffix using the `suffixes` keyword:

In [68]:
```
display('df15', 'df16', 'pd.merge(df15, df16, on="name", suffixes=["_L","_R"])')
```

Out[68]:

df15

	name	rank
0	Bob	1
1	Jake	2
2	Lisa	3
3	Sue	4

df16

	name	rank
0	Bob	3
1	Jake	1
2	Lisa	4
3	Sue	2

pd.merge(df15, df16, on="name", suffixes=["_L","_R"])

	name	rank_L	rank_R
0	Bob	1	3
1	Jake	2	1
2	Lisa	3	4
3	Sue	4	2

These suffixes work in any of the possible join patterns, and work also if there are multiple overlapping columns.

Example: US States Data

Merge and join operations come up most often when combining data from different sources. Here we will consider an example of some data about US states and their populations.

Let's take a look at the three datasets, using the Pandas `read_csv()` function:

In [69]:
```
# Read CSV (comma-separated) file into DataFrame
pop = pd.read_csv('state-population.csv')
areas = pd.read_csv('state-areas.csv')
abbrevs = pd.read_csv('state-abbrevs.csv')
# df.head(n=5): return the first 'n' rows
display('pop.head()', 'areas.head()', 'abbrevs.head()')
```

Out[69]:

pop.head()

	state/region	ages	year	population
0	AL	under18	2012	1117489.0
1	AL	total	2012	4817528.0
2	AL	under18	2010	1130966.0
3	AL	total	2010	4785570.0
4	AL	under18	2011	1125763.0

areas.head()

	state	area (sq. mi)
0	Alabama	52423
1	Alaska	656425
2	Arizona	114006
3	Arkansas	53182
4	California	163707

abbrevs.head()

	state	abbreviation
0	Alabama	AL
1	Alaska	AK
2	Arizona	AZ
3	Arkansas	AR
4	California	CA

Given this information, say we want to compute a relatively straightforward result: rank US states and territories by their 2010 population density. We clearly have the data here to find this result, but we'll have to combine the datasets to find the result.

- We'll start with a many-to-one merge that will give us the full state name within the population `DataFrame`. We want to merge based on the `state/region` column of `pop`, and the `abbreviation` column of `abbrevs`. We'll use `how='outer'` to make sure no data is thrown away due to mismatched labels.

In [70]:
```
merged = pd.merge(pop, abbrevs, how='outer',
                  left_on='state/region', right_on='abbreviation')
merged = merged.drop('abbreviation', axis=1) # drop duplicate info
merged.head()
```

Out[70]:

	state/region	ages	year	population	state
0	AL	under18	2012	1117489.0	Alabama
1	AL	total	2012	4817528.0	Alabama
2	AL	under18	2010	1130966.0	Alabama
3	AL	total	2010	4785570.0	Alabama
4	AL	under18	2011	1125763.0	Alabama

Let's double-check whether there were any mismatches here, which we can do by looking for rows with nulls:

In [71]:
```
merged.isnull().any()
```

Out[71]:
```
state/region    False
ages            False
year            False
population       True
state            True
dtype: bool
```

Some of the `population` info is null; let's figure out which these are!

In [72]:
```
merged[merged['population'].isnull()].head()
```

Out[72]:

	state/region	ages	year	population	state
2448	PR	under18	1990	NaN	NaN
2449	PR	total	1990	NaN	NaN
2450	PR	total	1991	NaN	NaN
2451	PR	under18	1991	NaN	NaN
2452	PR	total	1993	NaN	NaN

It appears that all the null population values are from Puerto Rico prior to the year 2000; this is likely due to this data not being available from the original source.

More importantly, we see also that some of the new `state` entries are also null, which means that there was no corresponding entry in the `abbrevs` key! Let's figure out which regions lack this match:

In [73]:
```
merged.loc[merged['state'].isnull(), 'state/region'].unique()
```

Out[73]:
```
array(['PR', 'USA'], dtype=object)
```

We can quickly infer the issue: our population data includes entries for Puerto Rico (PR) and the United States as a whole (USA), while these entries do not appear in the state abbreviation key. We can fix these quickly by filling in appropriate entries:

In [74]:
```
merged.loc[merged['state/region'] == 'PR', 'state'] = 'Puerto Rico'
merged.loc[merged['state/region'] == 'USA', 'state'] = 'United States'
merged.isnull().any()
```

Out[74]:
```
state/region    False
ages            False
year            False
population       True
state           False
dtype: bool
```

No more nulls in the `state` column: we're all set!

Now we can merge the result with the area data using a similar procedure. Examining our results, we will want to join on the `state` column in both:

In [75]:
```
final = pd.merge(merged, areas, on='state', how='left')
final.head()
```

Out[75]:

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

Again, let's check for nulls to see if there were any mismatches:

In [76]:
```
final.isnull().any(axis=0)
```

Out[76]:
```
state/region     False
ages             False
year             False
population        True
state            False
area (sq. mi)     True
dtype: bool
```

There are nulls in the `area` column; we can take a look to see which regions were ignored here:

In [77]:
```
# dataframe[column][row] to access elements
final['state'][final['area (sq. mi)'].isnull()].unique()
```

Out[77]:
```
array(['United States'], dtype=object)
```

We see that our `areas` `DataFrame` does not contain the area of the United States as a whole. We could insert the appropriate value (using the sum of all state areas, for instance), but in this case we'll just drop the null values because the population density of the entire United States is not relevant to our current discussion:

In [78]:
```
# inplace=true will change the content of the original data
final.dropna(inplace=True)
final.head()
```

Out[78]:

	state/region	ages	year	population	state	area (sq. mi)
0	AL	under18	2012	1117489.0	Alabama	52423.0
1	AL	total	2012	4817528.0	Alabama	52423.0
2	AL	under18	2010	1130966.0	Alabama	52423.0
3	AL	total	2010	4785570.0	Alabama	52423.0
4	AL	under18	2011	1125763.0	Alabama	52423.0

Now we have all the data we need. To answer the question of interest, let's first select the portion of the data corresponding with the year 2010, and the total population. We'll use the `query()` function to do this quickly (pandas query method offers a simple way for making selections. The main advantage of this method, is that it allows writing cleaner and more readable code for getting the exact pieces of data you want):

In [79]:
```python
data2010 = final.query("year == 2010 & ages == 'total'")
data2010.head()
```

Out[79]:

	state/region	ages	year	population	state	area (sq. mi)
3	AL	total	2010	4785570.0	Alabama	52423.0
91	AK	total	2010	713868.0	Alaska	656425.0
101	AZ	total	2010	6408790.0	Arizona	114006.0
189	AR	total	2010	2922280.0	Arkansas	53182.0
197	CA	total	2010	37333601.0	California	163707.0

Now let's compute the population density and display it in order. We'll start by re-indexing our data on the state, and then compute the result:

In [80]:
```python
data2010.set_index('state', inplace=True)
density = data2010['population'] / data2010['area (sq. mi)']
density.head()
```

Out[80]:
```
state
Alabama        91.287603
Alaska          1.087509
Arizona        56.214497
Arkansas       54.948667
California    228.051342
dtype: float64
```

In [81]:
```python
density.sort_values(ascending=False, inplace=True)
density.head()
```

Out[81]:
```
state
District of Columbia    8898.897059
Puerto Rico             1058.665149
New Jersey              1009.253268
Rhode Island             681.339159
Connecticut              645.600649
dtype: float64
```

The result is a ranking of US states plus Washington, DC, and Puerto Rico in order of their 2010 population density, in residents per square mile. We can see that by far the densest region in this dataset is Washington, DC (i.e., the District of Columbia); among states, the densest is New Jersey.

We can also check the end of the list:

In [82]:
```python
density.tail()
```

Out[82]:
```
state
South Dakota    10.583512
North Dakota     9.537565
Montana          6.736171
Wyoming          5.768079
Alaska           1.087509
dtype: float64
```

We see that the least dense state, by far, is Alaska, averaging slightly over one resident per square mile.

This type of messy data merging is a common task when trying to answer questions using real-world data sources. This example has given you an idea of the ways you can combine tools we've covered in order to gain insight from your data!

3 Aggregation and Grouping

An essential piece of analysis of large data is efficient summarization: computing aggregations like sum(), mean(), median(), min(), and max(), in which a single number gives insight into the nature of a potentially large dataset.

In this section, we'll explore aggregations in Pandas, from simple operations akin to what we've seen on NumPy arrays, to more sophisticated operations based on the concept of a groupby.

3.1 Planets Data

Here we will use the Planets dataset, which gives information on planets that astronomers have discovered around other stars (known as *extrasolar planets* or *exoplanets* for short).

In [83]:
```
planets = pd.read_csv("planets.csv")
planets.shape
```

Out[83]:
```
(1035, 6)
```

In [84]:
```
planets.head()
```

Out[84]:

	method	number	orbital_period	mass	distance	year
0	Radial Velocity	1	269.300	7.10	77.40	2006
1	Radial Velocity	1	874.774	2.21	56.95	2008
2	Radial Velocity	1	763.000	2.60	19.84	2011
3	Radial Velocity	1	326.030	19.40	110.62	2007
4	Radial Velocity	1	516.220	10.50	119.47	2009

This has some details on the 1,000+ extrasolar planets discovered up to 2014.

3.2 Simple Aggregation in Pandas

Earlier, we explored some of the data aggregations available for NumPy arrays. As with a one-dimensional NumPy array, for a Pandas `Series` the aggregates return a single value:

In [85]:
```
rng = np.random.RandomState(42)
ser = pd.Series(rng.rand(5))
ser
```

Out[85]:
```
0    0.374540
1    0.950714
2    0.731994
3    0.598658
4    0.156019
dtype: float64
```

In [86]:
```
ser.sum()
```

Out[86]:
```
2.811925491708157
```

In [87]:
```
ser.mean()
```

Out[87]:
```
0.5623850983416314
```

For a `DataFrame` , by default the aggregates return results within each **column**:

In [88]:
```
df = pd.DataFrame({'A': rng.rand(5),
                   'B': rng.rand(5)})
df
```

Out[88]:

	A	B
0	0.155995	0.020584
1	0.058084	0.969910
2	0.866176	0.832443
3	0.601115	0.212339
4	0.708073	0.181825

In [89]:
```
df.mean()
```

Out[89]:
```
A    0.477888
B    0.443420
dtype: float64
```

By specifying the `axis` argument, you can instead aggregate within each row:

In [90]:
```
df.mean(axis='columns')
```

Out[90]:
```
0    0.088290
1    0.513997
2    0.849309
3    0.406727
4    0.444949
dtype: float64
```

Pandas `Series` and `DataFrame` s include all of the common aggregates as introduced in numpy chapter. In addition, there is a convenience method `describe()` that computes several common aggregates for each column and returns the result. Let's use this on the Planets data, for now dropping rows with missing values:

In [91]:
```
# planets.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)
# Remove missing values, axis = 0 (default), drop rows which contain missing values
planets.dropna().describe()
```

Out[91]:

	number	orbital_period	mass	distance	year
count	498.00000	498.000000	498.000000	498.000000	498.000000
mean	1.73494	835.778671	2.509320	52.068213	2007.377510
std	1.17572	1469.128259	3.636274	46.596041	4.167284
min	1.00000	1.328300	0.003600	1.350000	1989.000000
25%	1.00000	38.272250	0.212500	24.497500	2005.000000
50%	1.00000	357.000000	1.245000	39.940000	2009.000000
75%	2.00000	999.600000	2.867500	59.332500	2011.000000
max	6.00000	17337.500000	25.000000	354.000000	2014.000000

This can be a useful way to begin understanding the overall properties of a dataset.

For example, we see in the `year` column that although exoplanets were discovered as far back as 1989, half of all known expolanets were not discovered until 2010 or after. This is largely thanks to the *Kepler* mission, which is a space-based telescope specifically designed for finding eclipsing planets around other stars.

The following table summarizes some other built-in Pandas aggregations:

Aggregation	Description
`count()`	Total number of items
`first()` , `last()`	First and last item
`mean()` , `median()`	Mean and median
`min()` , `max()`	Minimum and maximum
`std()` , `var()`	Standard deviation and variance
`mad()`	Mean absolute deviation
`prod()`	Product of all items
`sum()`	Sum of all items

These are all methods of `DataFrame` and `Series` objects.

To go deeper into the data, however, simple aggregates are often not enough.

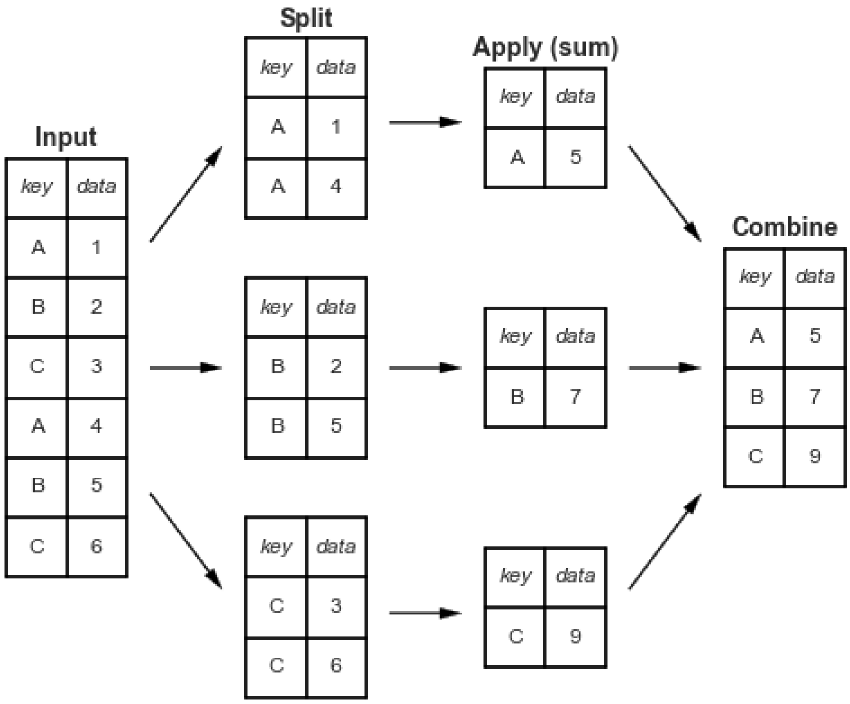
The next level of data summarization is the `groupby` operation, which allows you to **quickly and efficiently compute aggregates on subsets of data**.

3.3 GroupBy: Split, Apply, Combine

Simple aggregations can give you a flavor of your dataset, but often we would prefer to aggregate conditionally on some label or index: this is implemented in the so-called `groupby` operation. The name "group by" comes from a command in the SQL database language, but it is perhaps more illuminative to think of it in the terms first coined by Hadley Wickham of Rstats fame: *split, apply, combine*.

Split, apply, combine

A canonical example of this split-apply-combine operation, where the "apply" is a summation aggregation, is illustrated in this figure:



This makes clear what the `groupby` accomplishes:

- The *split* step involves breaking up and grouping a `DataFrame` depending on the value of the specified key.
- The *apply* step involves computing some function, usually an aggregate, transformation, or filtering, within the individual groups.
- The *combine* step merges the results of these operations into an output array.

While this could certainly be done manually using some combination of the masking, aggregation, and merging commands covered earlier, an important realization is that *the intermediate splits do not need to be explicitly instantiated*. Rather, the `GroupBy` can (often) do this in a single pass over the data, updating the sum, mean, count, min, or other aggregate for each group along the way.

As a concrete example, let's take a look at using Pandas for the computation shown in this diagram. We'll start by creating the input `DataFrame`:

In [92]:
```
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data': range(6)}, columns=['key', 'data'])
df
```

Out[92]:

	key	data
0	A	0
1	B	1
2	C	2
3	A	3
4	B	4
5	C	5

The most basic split-apply-combine operation can be computed with the `groupby()` method of `DataFrame` s, passing the name of the desired key column:

In [93]:
```
df.groupby('key')
```
Out[93]: `<pandas.core.groupby.groupby.DataFrameGroupBy object at 0x0000024137C38B70>`

Notice that what is returned is not a set of `DataFrame` s, but a `DataFrameGroupBy` object. This object is where the magic is: you can think of it as a special view of the `DataFrame` , which is poised to dig into the groups but does no actual computation until the aggregation is applied.

To produce a result, we can apply an aggregate to this `DataFrameGroupBy` object, which will perform the appropriate apply/combine steps to produce the desired result:

In [94]:
```
df.groupby('key').sum()
```
Out[94]:

	data
key	
A	3
B	5
C	7

The `sum()` method is just one possibility here; you can apply virtually any common Pandas or NumPy aggregation function, as well as virtually any valid `DataFrame` operation, as we will see in the following discussion.

The GroupBy object

The `GroupBy` object is a very flexible abstraction. In many ways, you can simply treat it as if it's a collection of `DataFrame` s, and it does the difficult things under the hood. Let's see some examples using the Planets data.

Column indexing

The `GroupBy` object supports column indexing in the same way as the `DataFrame` , and returns a modified `GroupBy` object. For example:

In [95]:
```
planets.groupby('method')
```
Out[95]: `<pandas.core.groupby.groupby.DataFrameGroupBy object at 0x0000024137C38A90>`

In [96]:
```
planets.groupby('method')['orbital_period']
```
Out[96]: `<pandas.core.groupby.groupby.SeriesGroupBy object at 0x0000024137C38A20>`

Here we've selected a particular `Series` group from the original `DataFrame` group by reference to its column name. As with the `GroupBy` object, no computation is done until we call some aggregate on the object:

In [97]:
```
planets.groupby('method')['orbital_period'].median()
```

Out[97]:
```
method
Astrometry                         631.180000
Eclipse Timing Variations         4343.500000
Imaging                          27500.000000
Microlensing                      3300.000000
Orbital Brightness Modulation        0.342887
Pulsar Timing                       66.541900
Pulsation Timing Variations       1170.000000
Radial Velocity                    360.200000
Transit                              5.714932
Transit Timing Variations           57.011000
Name: orbital_period, dtype: float64
```

This gives an idea of the general scale of orbital periods (in days) that each method is sensitive to.

Aggregate, filter, apply

The preceding discussion focused on aggregation for the combine operation, but there are more options available. In particular, `GroupBy` objects have `aggregate()` , `filter()` and `apply()` methods that efficiently implement a variety of useful operations before combining the grouped data.

For the purpose of the following subsections, we'll use this `DataFrame` :

In [98]:
```
rng = np.random.RandomState(0)
df = pd.DataFrame({'key': ['A', 'B', 'C', 'A', 'B', 'C'],
                   'data1': range(6),
                   'data2': rng.randint(0, 10, 6)},
                   columns = ['key', 'data1', 'data2'])
df
```

Out[98]:

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

Aggregation

We're now familiar with `GroupBy` aggregations with `sum()` , `median()` , and the like, but the `aggregate()` method allows for even more flexibility.

It can take a *string*, a *function*, or *a list thereof*, and compute all the aggregates at once. Here is a quick example combining all these:

```
In [99]: df.groupby('key').aggregate(['min', np.median, max])
```

Out[99]:

	data1			data2		
	min	median	max	min	median	max
key						
A	0	1.5	3	3	4.0	5
B	1	2.5	4	0	3.5	7
C	2	3.5	5	3	6.0	9

Another useful pattern is to pass a dictionary mapping column names to operations to be applied on that column:

```
In [100]: df.groupby('key').aggregate({'data1': 'min',
                                       'data2': max})
```

Out[100]:

	data1	data2
key		
A	0	5
B	1	7
C	2	9

Filtering

A filtering operation allows you to drop data **based on the group properties**. The `filter()` method lets you apply an arbitrary function to the group results. For example, we might want to keep all groups in which the standard deviation is larger than some critical value:

```
In [101]: def filter_func(x):
              return x['data2'].std() > 4

          display('df', "df.groupby('key').std()", "df.groupby('key').filter(filter_func)")
```

Out[101]:

`df`

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

`df.groupby('key').std()`

	data1	data2
key		
A	2.12132	1.414214
B	2.12132	4.949747
C	2.12132	4.242641

`df.groupby('key').filter(filter_func)`

	key	data1	data2
1	B	1	0
2	C	2	3
4	B	4	7
5	C	5	9

The filter function should return a Boolean value specifying whether the group passes the filtering. Here because group A does not have a standard deviation greater than 4, it is dropped from the result.

The apply() method

The `apply()` method lets you apply an arbitrary function to the group results. The function should take a `DataFrame`, and return either a Pandas object (e.g., `DataFrame`, `Series`) or a scalar; the combine operation will be tailored to the type of output returned.

For example, here is an `apply()` that normalizes the first column by the sum of the second:

In [102]:
```
def norm_by_data2(x):
    # x is a DataFrame of group values
    x['data1'] /= x['data2'].sum()
    return x

display('df', "df.groupby('key').apply(norm_by_data2)")
```

Out[102]:

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby('key').apply(norm_by_data2)

	key	data1	data2
0	A	0.000000	5
1	B	0.142857	0
2	C	0.166667	3
3	A	0.375000	3
4	B	0.571429	7
5	C	0.416667	9

`apply()` within a `groupby` is quite flexible: the only criterion is that the function takes a `DataFrame` and returns a Pandas object or scalar; what you do in the middle is up to you!

Specifying the split key

In the simple examples presented before, we split the `DataFrame` on a single column name by using `groupby`. This is just one of many options by which the groups can be defined, and we'll go through some other options for group specification here.

A list, array, series, or index providing the grouping keys

The key can be any series or list with a length matching that of the `DataFrame`. For example:

In [103]:
```
L = [0, 1, 0, 1, 2, 0]
# L=['A','B','C','A','B','C']
display('df', 'df.groupby(L).sum()')
```

Out[103]:

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby(L).sum()

	data1	data2
0	7	17
1	4	3
2	4	7

Of course, this means there's another, more verbose way of accomplishing the `df.groupby('key')` from before:

In [104]:
```
display('df', "df.groupby(df['key']).sum()")
```

Out[104]:

df

	key	data1	data2
0	A	0	5
1	B	1	0
2	C	2	3
3	A	3	3
4	B	4	7
5	C	5	9

df.groupby(df['key']).sum()

	data1	data2
key		
A	3	8
B	5	7
C	7	12

A dictionary or series mapping index to group

Another method is to provide a dictionary that maps index values to the group keys:

In [105]:
```
df2 = df.set_index('key')
mapping = {'A': 'vowel', 'B': 'consonant', 'C': 'consonant'}
display('df2', 'df2.groupby(mapping).sum()')
```

Out[105]:

df2

	data1	data2
key		
A	0	5
B	1	0
C	2	3
A	3	3
B	4	7
C	5	9

df2.groupby(mapping).sum()

	data1	data2
consonant	12	19
vowel	3	8

Any Python function

Similar to mapping, you can pass any Python function that will input the index value and output the group:

In [106]:
```
display('df2', 'df2.groupby(str.lower).mean()')
```

Out[106]:

df2

key	data1	data2
A	0	5
B	1	0
C	2	3
A	3	3
B	4	7
C	5	9

df2.groupby(str.lower).mean()

	data1	data2
a	1.5	4.0
b	2.5	3.5
c	3.5	6.0

A list of valid keys

Further, any of the preceding key choices can be combined to group on a multi-index:

In [107]:
```
df2.groupby([mapping, str.lower]).mean()
```

Out[107]:

		data1	data2
consonant	b	2.5	3.5
	c	3.5	6.0
vowel	a	1.5	4.0

Grouping example

As an example of this, in a couple lines of Python code we can put all these together and count discovered planets by method and by decade:

An essential piece of analysis of large data is efficient summarization: computing aggregations like `sum()`, `mean()`, `median()`, `min()`, and `max()`, in which a single number gives insight into the nature of a potentially large dataset.

In this section, we'll explore aggregations in Pandas, from simple operations akin to what we've seen on NumPy arrays, to more sophisticated operations based on the concept of a `groupby`.

In [108]:
```
decade = 10 * (planets['year'] // 10)
decade = decade.astype(str) + 's'
decade.name = 'decade'
planets.groupby(['method', decade])['number'].sum().unstack().fillna(0)
```

Out[108]:

decade	1980s	1990s	2000s	2010s
method				
Astrometry	0.0	0.0	0.0	2.0
Eclipse Timing Variations	0.0	0.0	5.0	10.0
Imaging	0.0	0.0	29.0	21.0
Microlensing	0.0	0.0	12.0	15.0
Orbital Brightness Modulation	0.0	0.0	0.0	5.0
Pulsar Timing	0.0	9.0	1.0	1.0
Pulsation Timing Variations	0.0	0.0	1.0	0.0
Radial Velocity	1.0	52.0	475.0	424.0
Transit	0.0	0.0	64.0	712.0
Transit Timing Variations	0.0	0.0	0.0	9.0

This shows the power of combining many of the operations we've discussed up to this point when looking at realistic datasets. We immediately gain a coarse understanding of when and how planets have been discovered over the past several decades!