Lecture 2: Data wrangling with Pandas

Why Pandas?

Pandas (https://pandas.pydata.org/) is a very useful package for data wrangling.

Particularly useful when working with real data, which can be messy.

Why Pandas?

Pandas (https://pandas.pydata.org/) is a very useful package for data wrangling.

Particularly useful when working with real data, which can be messy.

Combines advantages of a number of different data structures (NumPy arrays, dictionaries, relational databases).

Can also be more efficient than native Python data structures for certain operators (as we will see).

Particularly useful for dealing with:

- Labelled data
- Missing data
- Heteterogenous types
- Groupings

Particularly useful for dealing with:

- Labelled data
- Missing data
- Heteterogenous types
- Groupings

We will focus mostly on Pandas `Series` and `DataFrame` objects.

Import Pandas

In [2]:

```
import pandas as pd
import numpy as np
```

Documentation

Recall can check documentation with `pd?`, `pd.<TAB>`, and/or print documentation for specific function with `print(pd.<function_name>.__doc__)`.

In [3]:
```
#pd?
```

In [4]:
```
#pd.
```

In [5]:
```
#pd.concat?
```

In [6]:
```
#print(pd.concat.__doc__)
```

Pandas `Series`

A Pandas `Series` is a *1D* array of *indexed* data.

Pandas `Series`

A Pandas `Series` is a *1D* array of *indexed* data.

Can be created from a list or array:

In [7]:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
data
```

Out[7]:

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

The `Series` wraps both a sequence of *values* and a sequence of *indices*, which we can access with the `values` and `index` attributes.

In [8]:
```
data
```

Out[8]:
```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

In [9]:
```
data.values
```

Out[9]:
```
array([0.25, 0.5 , 0.75, 1.  ])
```

In [10]:
```
data.index
```

Out[10]:
```
RangeIndex(start=0, stop=4, step=1)
```

`Series` as generalized NumPy array

Values are simply NumPy array.

Index need not be an integer, but can consist of values of any desired type.

`Series` as generalized NumPy array

Values are simply NumPy array.

Index need not be an integer, but can consist of values of any desired type.

In [11]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

Out[11]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

`Series` as generalized NumPy array

Values are simply NumPy array.

Index need not be an integer, but can consist of values of any desired type.

In [11]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

Out[11]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In [12]:
```
data['b']
```

Out[12]:
```
0.5
```

`Series` as specialized dictionary

Can also think of a Pandas `Series` like a specialization of a Python dictionary.

`Series` as specialized dictionary

Can also think of a Pandas `Series` like a specialization of a Python dictionary.

In [13]:
```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict) # Instantiate from dictionary
```

In [14]:
```
type(population_dict), type(population)
```

Out[14]:
```
(dict, pandas.core.series.Series)
```

`Series` as specialized dictionary

Can also think of a Pandas `Series` like a specialization of a Python dictionary.

In [13]:
```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict) # Instantiate from dictionary
```

In [14]:
```
type(population_dict), type(population)
```

Out[14]:
```
(dict, pandas.core.series.Series)
```

In [15]:
```
population['California']
```

Out[15]:
```
38332521
```

- Python dictionary: maps *arbitrary* keys to *arbitrary* values.
- Pandas `Series`: maps *typed* indices to *typed* values.

- Python dictionary: maps *arbitrary* keys to *arbitrary* values.
- Pandas `Series`: maps *typed* indices to *typed* values.

Type information of Pandas `Series` makes it much more efficient than Python dictionaries for certain operations.

Pandas `DataFrame`

`DataFrame` can be thought of as a sequence of aligned `Series` objects, with *indices* and *columns*.

Pandas `DataFrame`

`DataFrame` can be thought of as a sequence of aligned `Series` objects, with *indices* and *columns*.

In [16]:

```
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
```

Out[16]:

	foo	bar
a	0.323654	0.593432
b	0.793729	0.217120
c	0.405826	0.327011

`DataFrame` as generalized NumPy array

`DataFrame` is an analog of a two-dimensional array with both flexible row indices and flexible column names.

`DataFrame` as generalized NumPy array

`DataFrame` is an analog of a two-dimensional array with both flexible row indices and flexible column names.

Contstruct another `Series` with same indices.

In [17]:

```
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
```

Combine two `Series` into a `DataFrame`.

In [18]:
```
states = pd.DataFrame({'population': population,
                       'area': area})
states
```

Out[18]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

`DataFrame` has both `index` and `column` attributes.

In [19]:
```
states
```

Out[19]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

`DataFrame` has both `index` and `column` attributes.

In [19]:
```
states
```

Out[19]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

In [20]:
```
states.index
```

Out[20]:
```
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'], dtype='objec
t')
```

`DataFrame` has both `index` and `column` attributes.

In [19]:
```
states
```

Out[19]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

In [20]:
```
states.index
```

Out[20]:
```
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas'], dtype='objec
t')
```

In [21]:
```
states.columns
```

Out[21]:
```
Index(['area', 'population'], dtype='object')
```

`DataFrame` as specialized dictionary

Can also think of a Pandas `DataFrame` like a specialization of a Python dictionary.

`DataFrame` as specialized dictionary

Can also think of a Pandas `DataFrame` like a specialization of a Python dictionary.

`DataFrame` maps a column name to a `Series`.

```
In [22]: states['area']
```

```
Out[22]: California    423967
         Florida       170312
         Illinois      149995
         New York      141297
         Texas         695662
         Name: area, dtype: int64
```

```
In [23]: type(states['area'])
```

```
Out[23]: pandas.core.series.Series
```

Pandas `Index`

Both Pandas `Series` and `DataFrame` contain `Index` object(s).

Can be thought of as *immutable array* (i.e. cannot be changed) or *ordered multi-set* (may contain repeated values).

Pandas `Index`

Both Pandas `Series` and `DataFrame` contain `Index` object(s).

Can be thought of as *immutable array* (i.e. cannot be changed) or *ordered multi-set* (may contain repeated values).

In [24]:
```
ind = pd.Index([2, 3, 5, 7, 11])
ind
```

Out[24]:
```
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

`Index` as immutable array

Immutability makes it safer to share indices between multiple `DataFrames`.

In [25]:
```
ind[1]
```

Out[25]:
```
3
```

In [26]:
```
#ind[1] = 0
```

`Index` as ordered multi-set

`Index` objects support many set operations, e.g. joins, unions, intersections, differences.

Example: Compute the intersection and union of the following two `Index` objects.

In [27]:

```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
```

Example: Compute the intersection and union of the following two `Index` objects.

In [27]:
```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
```

In [28]:
```
indA & indB  # intersection
```

Out[28]:
```
Int64Index([3, 5, 7], dtype='int64')
```

Example: Compute the intersection and union of the following two `Index` objects.

In [27]:
```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
```

In [28]:
```
indA & indB  # intersection
```

Out[28]:
```
Int64Index([3, 5, 7], dtype='int64')
```

In [29]:
```
indA | indB  # union
```

Out[29]:
```
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
```

Data indexing and selection

Data selection in a `Series`

In additional to acting like a dictionary, a `Series` also provies array-style selection like NumPy arrays.

In [30]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

Out[30]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In [30]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

Out[30]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In [31]:
```
# slicing by explicit index
data['a':'c']
```

Out[31]:
```
a    0.25
b    0.50
c    0.75
dtype: float64
```

In [30]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

Out[30]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In [31]:
```
# slicing by explicit index
data['a':'c']
```

Out[31]:
```
a    0.25
b    0.50
c    0.75
dtype: float64
```

In [32]:
```
# slicing by implicit integer index
data[0:2]
```

Out[32]:
```
a    0.25
b    0.50
dtype: float64
```

In [30]:
```python
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

Out[30]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In [31]:
```python
# slicing by explicit index
data['a':'c']
```

Out[31]:
```
a    0.25
b    0.50
c    0.75
dtype: float64
```

In [32]:
```python
# slicing by implicit integer index
data[0:2]
```

Out[32]:
```
a    0.25
b    0.50
dtype: float64
```

When slicing by an explicit index (e.g. `data['a':'c']`), the final index *is* included.

When slicing by an implicit index (e.g. `data[0:2]`), the final index *is not* included.

This can be a source of much confusion.

Consider a Series with integer indices.

In [33]:
```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```

Out[33]:
```
1    a
3    b
5    c
dtype: object
```

In [34]:
```
# explicit index when indexing
data[1]
```

Out[34]:
```
'a'
```

In [35]:
```
# implicit index when slicing
data[1:3]
```

Out[35]:
```
3    b
5    c
dtype: object
```

Indexers

Indexers `loc` (explicit) and `iloc` (implicit) are introduced to avoid confusion.

In [36]:
```
data
```

Out[36]:
```
1    a
3    b
5    c
dtype: object
```

In [37]:
```
data.loc[1]
```

Out[37]:
```
'a'
```

In [38]:
```
data.loc[1:3]
```

Out[38]:
```
1    a
3    b
dtype: object
```

In [39]:
```
data.iloc[1]
```

Out[39]:
```
'b'
```

Data selection in a `DataFrame`

In additional to acting like a dictionary of `Series` objects with the same index, a `DataFrame` also provies array-style selection like NumPy arrays.

In [41]:

```
area = pd.Series({'California': 423967, 'Texas': 695662,
                  'New York': 141297, 'Florida': 170312,
                  'Illinois': 149995})
pop = pd.Series({'California': 38332521, 'Texas': 26448193,
                 'New York': 19651127, 'Florida': 19552860,
                 'Illinois': 12882135})
data = pd.DataFrame({'area':area, 'population':pop})
data
```

Out[41]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

Indexers

Indexers `loc` (explicit) and `iloc` (implicit) are also available to avoid confusion when selecting data.

Note that index and column labels are preserved in the result.

In [42]:

```
data
```

Out[42]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

In [43]:
```
data.iloc[:3, :2]
```

Out[43]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

In [44]:
```
data.loc[:'Illinois', :'population']
```

Out[44]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135

Additional array-style selection

Other NumPy selection approaches can also be applied (e.g. masking).

Exercise: Create a `DataFrame`

Create a `DataFrame` containing only those states that have an area greater than 150,000 and a population greater than 20 million.

In [45]:
```
data
```

Out[45]:

	area	population
California	423967	38332521
Florida	170312	19552860
Illinois	149995	12882135
New York	141297	19651127
Texas	695662	26448193

In [46]:
```
data[(data.area > 150e3) & (data.population > 20e6)]
```

Out[46]:

	area	population
California	423967	38332521
Texas	695662	26448193

(Pandas raises an error if you try to convert something to `bool`, hence use bitwise logical operations. Read more [here (http://pandas.pydata.org/pandas-docs/version/0.15/gotchas.html)](http://pandas.pydata.org/pandas-docs/version/0.15/gotchas.html).)

Operating on data in Pandas

Elementwise operations in Pandas automatically aligns indices and preserves index/column labels.

Can avoid many errors and bugs in data wrangling.

Index preservation

In [47]:
```
rng = np.random.RandomState(42)
ser = pd.Series(rng.randint(0, 10, 4))
ser
```

Out[47]:
```
0    6
1    3
2    7
3    4
dtype: int64
```

In [48]:
```
np.exp(ser)
```

Out[48]:
```
0     403.428793
1      20.085537
2    1096.633158
3      54.598150
dtype: float64
```

In [49]:
```python
df = pd.DataFrame(rng.randint(0, 10, (3, 4)),
                  columns=['A', 'B', 'C', 'D'])
df
```

Out[49]:

	A	B	C	D
0	6	9	2	6
1	7	4	3	7
2	7	2	5	4

In [50]:
```python
np.exp(df)
```

Out[50]:

	A	B	C	D
0	403.428793	8103.083928	7.389056	403.428793
1	1096.633158	54.598150	20.085537	1096.633158
2	1096.633158	7.389056	148.413159	54.598150

Index alignment

Consider the following two series.

In [51]:
```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967}, name='area')
population = pd.Series({'California': 38332521, 'Texas': 26448193,
                        'New York': 19651127}, name='population')
```

Exercise: Compute the population density for each state (where possible).

Index alignment

Consider the following two series.

In [51]:

```
area = pd.Series({'Alaska': 1723337, 'Texas': 695662,
                  'California': 423967}, name='area')
population = pd.Series({'California': 38332521, 'Texas': 26448193,
                        'New York': 19651127}, name='population')
```

Exercise: Compute the population density for each state (where possible).

In [52]:

```
population / area
```

Out[52]:

```
Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64
```

In [53]:
```
population / area
```

Out[53]:
```
Alaska              NaN
California    90.413926
New York            NaN
Texas         38.018740
dtype: float64
```

The Pandas `Series` given by `population/area` contains indicies of the *union* of the two `Series` considered, with the density computed for states where both the area and population are available.

When one of the area or population are not available NaN is returned, which is how Pandas represents missing data.

Index alignment works similarly for `DataFrames`.

In [54]:
```
A = pd.DataFrame(rng.randint(0, 20, (2, 2)),
                 columns=list('AB'))
A
```

Out[54]:

	A	B
0	1	11
1	5	1

In [55]:
```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                 columns=list('BAC'))
B
```

Out[55]:

	B	A	C
0	4	0	9
1	5	8	0
2	9	2	6

In [56]:

```
A + B
```

Out[56]:

	A	B	C
0	1.0	15.0	NaN
1	13.0	6.0	NaN
2	NaN	NaN	NaN

Operations between `DataFrame` and `Series` objects

In [57]:
```
A = rng.randint(10, size=(3, 4))
df = pd.DataFrame(A, columns=list('QRST'))
df
```

Out[57]:

	Q	R	S	T
0	3	8	2	4
1	2	6	4	8
2	6	1	3	8

In [58]:
```
s = df.iloc[0]
s
```

Out[58]:
```
Q     3
R     8
S     2
T     4
Name: 0, dtype: int64
```

Difference between the `DataFrame` and `Series`:

In [59]:
```
df - s
```

Out[59]:

	Q	R	S	T
0	0	0	0	0
1	-1	-2	2	4
2	3	-7	1	4

Convention is to operate row-wise.

Can also operate column-wise using object methods.

In [60]:
```
df.subtract(df['R'], axis=0)
```

Out[60]:

	Q	R	S	T
0	-5	0	-6	-4
1	-4	0	-2	2
2	5	0	2	7

Handling missing data

Real data is messy. Often some data are missing.

Various conventions can be considered to handle missing data.

We will focus on the use of the floating point IEEE value NaN (not a number) to represent missing data.

Various conventions can be considered to handle missing data.

We will focus on the use of the floating point IEEE value NaN (not a number) to represent missing data.

Pandas interprets NaN as Null values.

(Pandas also supports `None` but we will focus on NaN here.)

Arithematic operations with NaN values result in NaN.

In [61]:
```
1 + np.nan
```

Out[61]:
```
nan
```

Operating on Null values

Several useful methods exist to work with NaNs, for example to detect, drop or replace:

- `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.

Detecting null values

Pandas `isnull` and `notnull` are useful for detecting null values.

Exercise: Detecting null values

Consider the following series.

In [62]:
```
data = pd.Series([1, np.nan, 'hello', np.nan])
data
```

Out[62]:
```
0        1
1      NaN
2    hello
3      NaN
dtype: object
```

Compute a new Series of bools that specify whether each entry in the above Series is *not* NaN. Using this Series, construct a new series from the original data that does not contain the NaN entries.

In [63]:
```
data
```

Out[63]:
```
0        1
1      NaN
2    hello
3      NaN
dtype: object
```

In [64]:
```
not_null = data.notnull()
not_null
```

Out[64]:
```
0     True
1    False
2     True
3    False
dtype: bool
```

In [63]:
```
data
```

Out[63]:
```
0        1
1      NaN
2    hello
3      NaN
dtype: object
```

In [64]:
```
not_null = data.notnull()
not_null
```

Out[64]:
```
0     True
1    False
2     True
3    False
dtype: bool
```

In [65]:
```
data[not_null]
```

Out[65]:
```
0        1
2    hello
dtype: object
```

Dropping null values

Direct routines may be used to drop null values (i.e. `dropna`), rather than constructing masks as performed above.

Exercise: Remove null values directly

Remove null values from the previous data `Series` directly.

Exercise: Remove null values directly

Remove null values from the previous data `Series` directly.

In [66]:
```
data.dropna()
```

Out[66]:
```
0        1
2    hello
dtype: object
```

Dropping null values from `DataFrames`

For `DataFrames`, there are multiple ways null values can be dropped.

In [67]:
```
df = pd.DataFrame([[1,      np.nan, 2],
                   [2,      3,      5],
                   [np.nan, 4,      6]])
df
```

Out[67]:

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

Dropping null values from `DataFrames`

For `DataFrames`, there are multiple ways null values can be dropped.

In [67]:

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

Out[67]:

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

By default `dropna` operates row-wise and drops all rows that contain any NaNs.

In [68]:

```
df.dropna()
```

Out[68]:

	0	1	2
1	2.0	3.0	5

Can also operate column-wise.

In [69]:
```
df
```

Out[69]:

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

In [70]:
```
df.dropna(axis='columns')
```

Out[70]:

	2
0	2
1	5
2	6

Can also operate column-wise.

In [69]:
```
df
```

Out[69]:

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

In [70]:
```
df.dropna(axis='columns')
```

Out[70]:

	2
0	2
1	5
2	6

More sophisticated approaches can also be considered (e.g. only dropping rows/columns if all entries or a certain number of NaNs appear).

Replacing null values

Null values can be easily replaced using `fillna`.

In [71]:
```
df
```

Out[71]:

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

In [72]:
```
df.fillna(0.0)
```

Out[72]:

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

Can also fill using adjacent values.

In [73]:
```
df
```

Out[73]:

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

Can also fill using adjacent values.

```
In [73]: df
```

Out[73]:

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

```
In [74]: df.fillna(method='ffill', axis='columns')
```

Out[74]:

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

In [75]:
```python
df.fillna(method='ffill', axis='rows')
```

Out[75]:

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

In [75]:
```
df.fillna(method='ffill', axis='rows')
```

Out[75]:

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

In [76]:
```
df.fillna(method='bfill', axis='columns')
```

Out[76]:

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

Combining data-sets

Define helper functions

In [77]:

```
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[77]:

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

In [78]:

```
class display(object):
    """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):
        return '\n'.join(self.template.format(a, eval(a)._repr_html_())
                         for a in self.args)

    def __repr__(self):
        return '\n\n'.join(a + '\n' + repr(eval(a))
                           for a in self.args)
```

Concatenation

Can concatenate `Series` and `DataFrame` objects with `pd.concat()`.

Concatenation

Can concatenate `Series` and `DataFrame` objects with `pd.concat()`.

Default is to concatenate over rows.

In [79]:
```
df1 = make_df('AB', [1, 2])
df2 = make_df('AB', [3, 4])
display('df1', 'df2', 'pd.concat([df1, df2])')
```

Out[79]:

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

Can also concatenate over columns.

Duplicated indices

Can have duplicated indices.

Duplicated indices

Can have duplicated indices.

In [81]:
```
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[81]:

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

Ignoring index

Can ignore index.

In [82]:
```
display('x', 'y', 'pd.concat([x, y], ignore_index=True)')
```

Out[82]:

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

Concantenation with joins

Can join `DataFrames` with different column names.

Concantenation with joins

Can join `DataFrames` with different column names.

In [83]:
```
df5 = make_df('ABC', [1, 2])
df6 = make_df('BCD', [3, 4])
display('df5', 'df6', 'pd.concat([df5, df6])')
```

Out[83]:

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])

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

Entries with no data are filled with NaN.

Default join is the *union* of the columns of the two `DataFrames`.

Can also perform different types of joins.

For example, the *intersection* of the columns of the two DataFrames.

In [84]:
```
display('df5', 'df6',
        "pd.concat([df5, df6], join='inner')")
```

Out[84]:

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')

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

Relational combinations

Pandas also provides functionality to perform relational algebra (cf. relational databases).

Hence, Pandas data structures provide analogy not only of NumPy array and dictionary, but also relational database.

Relational combinations

Pandas also provides functionality to perform relational algebra (cf. relational databases).

Hence, Pandas data structures provide analogy not only of NumPy array and dictionary, but also relational database.

Functionality provied by `pd.merge()` function.

One-to-one join

In [85]:
```
df1 = pd.DataFrame({'employee': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'group': ['Accounting', 'Engineering', 'Engineering', 'HR']})
df2 = pd.DataFrame({'employee': ['Lisa', 'Bob', 'Jake', 'Sue'],
                    'hire_date': [2004, 2008, 2012, 2014]})
df3 = pd.merge(df1, df2)
display('df1', 'df2', 'df3')
```

Out[85]:

df1

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

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

df3

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

Many-to-one joins

In [86]:
```
df4 = pd.DataFrame({'group': ['Accounting', 'Engineering', 'HR'],
                    'supervisor': ['Carly', 'Guido', 'Steve']})
display('df3', 'df4', 'pd.merge(df3, df4)')
```

Out[86]:

df3

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

df4

	group	supervisor
0	Accounting	Carly
1	Engineering	Guido
2	HR	Steve

pd.merge(df3, df4)

	employee	group	hire_date	supervisor
0	Bob	Accounting	2008	Carly
1	Jake	Engineering	2012	Guido
2	Lisa	Engineering	2004	Guido
3	Sue	HR	2014	Steve

The `on` keyword

In [87]:
```
display('df1', 'df2', "pd.merge(df1, df2, on='employee')")
```

Out[87]:

df1

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

df2

	employee	hire_date
0	Lisa	2004
1	Bob	2008
2	Jake	2012
3	Sue	2014

pd.merge(df1, df2, on='employee')

	employee	group	hire_date
0	Bob	Accounting	2008
1	Jake	Engineering	2012
2	Lisa	Engineering	2004
3	Sue	HR	2014

The `left_on` and `right_on` keywords

Employee and name both included now, so may want to drop one.

In [89]:
```
pd.merge(df1, df3, left_on="employee", right_on="name")
```

Out[89]:

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

Employee and name both included now, so may want to drop one.

In [89]:
```
pd.merge(df1, df3, left_on="employee", right_on="name")
```

Out[89]:

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

In [90]:
```
pd.merge(df1, df3, left_on="employee", right_on="name").drop('name', axis='column
s')
```

Out[90]:

	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

Often one wants to join on index.

In [91]:

```
df1a = df1.set_index('employee')
df2a = df2.set_index('employee')
display('df1a', 'df2a')
```

Out[91]:

df1a

	group
employee	
Bob	Accounting
Jake	Engineering
Lisa	Engineering
Sue	HR

df2a

	hire_date
employee	
Lisa	2004
Bob	2008
Jake	2012
Sue	2014

In [92]:

```
display("pd.merge(df1a, df2a, left_index=True, right_index=True)")
```

Out[92]:

```
pd.merge(df1a, df2a, left_index=True, right_index=True)
```

	group	hire_date
employee		
Bob	Accounting	2008
Jake	Engineering	2012
Lisa	Engineering	2004
Sue	HR	2014

Set arithmetic for joins

Have so far been considering relational joins based on *intersection* (also called *inner* join).

In [93]:

```
df6 = pd.DataFrame({'name': ['Peter', 'Paul', 'Mary'],
                    'food': ['fish', 'beans', 'bread']},
                   columns=['name', 'food'])
df7 = pd.DataFrame({'name': ['Mary', 'Joseph'],
                    'drink': ['wine', 'beer']},
                   columns=['name', 'drink'])
display('df6', 'df7', "pd.merge(df6, df7, how='inner')")
```

Out[93]:

df6

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

df7

	name	drink
0	Mary	wine
1	Joseph	beer

pd.merge(df6, df7, how='inner')

	name	food	drink
0	Mary	bread	wine

Outer join

Can also join based on *union* (missing entries filled with NaNs).

In [94]:
```
display('df6', 'df7', "pd.merge(df6, df7, how='outer')")
```

Out[94]:

```
df6
```

	name	food
0	Peter	fish
1	Paul	beans
2	Mary	bread

```
df7
```

	name	drink
0	Mary	wine
1	Joseph	beer

```
pd.merge(df6, df7, how='outer')
```

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

Left and right join

Can also join based on *left* or *right* entries.

Overlapping column names

Possible for DataFrames to have conflicting columns.

In [96]:
```
df8 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [1, 2, 3, 4]})
df9 = pd.DataFrame({'name': ['Bob', 'Jake', 'Lisa', 'Sue'],
                    'rank': [3, 1, 4, 2]})
display('df8', 'df9')
```

Out[96]:

df8

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

df9

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

In [97]:
```
display('pd.merge(df8, df9, on="name", suffixes=["_L", "_R"])')
```

Out[97]:

```
pd.merge(df8, df9, 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