Python Bootcamp 4 Part 1

pandas dataframes (I)

- Pandas is one of the most commonly used Python packages/libraries for data science. Developed by Wes McKinney in January 2008.
- Pandas is Python's answer for making two dimensional tables (like Excel).
- Pandas calls a table a "DataFrame".
- A DataFrame is a data structure that organizes data into a 2-dimensional table of rows and columns, much like a spreadsheet
- Pandas DataFrames are also used by Python's other packages for statistical analysis, data manipulation, and data visualization.
- Pandas DataFrames can be exported as .csv and other files.

About these two parts

These two parts are designed as an introduction to using the package known as `pandas`.

By the end of these two parts, you should be able to:
-read a csv into a dataframe
-filter by columns
-run some basic statistics on that dataframe
-graph the data using a second package called `seaborn`.

Introduction to Pandas

Pandas is the essential data analysis library for Python programmers. It provides fast and flexible data structures built on top of numpy (→ the fundamental package for scientific computing in Python, such as mathematical, logical, shape manipulation, linear algebra, basic statistical operations, etc.)

It is well suited to handle "tabular" data (that might be found in a spreadsheet), time series data, or pretty much anything you care to put in a matrix with rows and named columns.

It contains two primary data structures, the `Series` (1-dimensional) and the `DataFrame` (2-dimensional) as well as a host of convenience methods for loading and working with data.

The main point that makes pandas is that all data is *intrinsically aligned*. That means each data structure, `DataFrame` or `Series` has something called an **Index** that links data values with a label. That link will always be there (unless you explicitly break or change it) and it's what allows pandas to quickly and efficiently "do the right thing" when working with data.

```
!pip install pandas
```

```
Requirement already satisfied: pandas in
/home/codespace/.local/lib/python3.12/site-packages (2.2.3)
Requirement already satisfied: numpy>=1.26.0 in
/home/codespace/.local/lib/python3.12/site-packages (from pandas)
(2.2.0)
Requirement already satisfied: python-dateutil>=2.8.2 in
/home/codespace/.local/lib/python3.12/site-packages (from pandas)
(2.9.0.post0)
Requirement already satisfied: pytz>=2020.1 in
/home/codespace/.local/lib/python3.12/site-packages (from pandas)
(2024.2)
Requirement already satisfied: tzdata>=2022.7 in
/home/codespace/.local/lib/python3.12/site-packages (from pandas)
(2024.2)
Requirement already satisfied: six>=1.5 in
/home/codespace/.local/lib/python3.12/site-packages (from python-
dateutil>=2.8.2->pandas) (1.17.0)

[notice] A new release of pip is available: 24.3.1 -> 25.0
[notice] To update, run: python3 -m pip install --upgrade pip
```

import pandas

Because pandas is one of the most commonly used Python packages, it often gets imported as a shortened version of it's actual name. This makes it quicker to type.

```
import pandas as pd

import numpy as np
```

The `Series` Object

A `Series` is a one-dimensional labeled array of indexed data, capable of holding data of any type (integer, string, float, python objects, etc.)

```
data = pd.Series([0.1, 0.2, 0.3, 0.4])

data
```

```
0    0.1
1    0.2
2    0.3
3    0.4
dtype: float64
```

↑ On a 64-bit system, default types will be 64-bit

```
type(data)
```

```
pandas.core.series.Series
```

The `Series` wraps a 1-d `ndarray` from numpy and an `Index` object.

```
print(data.values)
```

```
[0.1 0.2 0.3 0.4]
```

```
type(data.values)
```

```
numpy.ndarray
```

```
data.index
```

```
RangeIndex(start=0, stop=4, step=1)
```

```
# This particular index type, the `RangeIndex`, let us use the
# same square-bracket notation as a `ndarray` to access elements:
data[0]
```

```
np.float64(0.1)
```

```
data.values[0]
```

```
np.float64(0.1)
```

```
# or even a slice:
data[1:3]
```

```
1    0.2
2    0.3
dtype: float64
```

```
data.values[1:3]
```

```
array([0.2, 0.3])
```

We don't have to use this auto-generated list of integers as the index though. Index values can be specified manually and don't even have to be integers.

```
data = pd.Series([0.1, 0.2, 0.3, 0.4], index=['a', 'b', 'c', 'd'])
```

```
data
```

```
a    0.1
b    0.2
c    0.3
d    0.4
dtype: float64
```

```
data.index
```

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

# Item access works just like before, with square brackets,
# even though the index values are strings
data['a']

np.float64(0.1)

#once you have labels, you can also access them this way (assuming no
spaces in name)
data.a

np.float64(0.1)

# slices still work! But note the last element is included this time.
# This is the default behavior for indexes.
data['a':'d']

a    0.1
b    0.2
c    0.3
d    0.4
dtype: float64

# We could create a non-sequential integer index:
data = pd.Series([0.1, 0.2, 0.3, 0.4], index=[5, 8, 2, 1])
data

5    0.1
8    0.2
2    0.3
1    0.4
dtype: float64

data.index

Index([5, 8, 2, 1], dtype='int64')

data[1]

np.float64(0.4)

# Why?
data.values[1]

np.float64(0.2)
```

Remember that the values method (data.values) is converting the column into a numpy array. This means any indexing follows the numpy rules (which are based on position), not the pandas rules (which are based on index)

`Series` are in fact a cross between a numpy array and a python dictionary. You can think of them as a dictionary with *typed* keys and *typed* values.

```python
# in fact it is easy to convert a dictionary into a series
max_depths_dict = {
    'Erie': 64,
    'Huron': 229,
    'Michigan': 281,
    'Ontario': 244,
    'Superior': 406,
}

max_depths_dict
```

```
{'Erie': 64, 'Huron': 229, 'Michigan': 281, 'Ontario': 244,
'Superior': 406}
```

```python
type(max_depths_dict)
```

```
dict
```

```python
max_depths = pd.Series(max_depths_dict)

max_depths
```

```
Erie         64
Huron       229
Michigan    281
Ontario     244
Superior    406
dtype: int64
```

```python
type(max_depths)
```

```
pandas.core.series.Series
```

```python
# it looks like a dictionary!
max_depths['Michigan']
```

```
np.int64(281)
```

```python
max_depths_dict['Michigan']
```

```
281
```

```python
# Notice the index in this case was constructed automatically from the
dictionary keys.
max_depths.index
```

```
Index(['Erie', 'Huron', 'Michigan', 'Ontario', 'Superior'],
dtype='object')
```

Numpy and `Series`

Because the values in a `Series` are contained in a numpy `ndarray`, `Series` provides all the benefits of numpy! Namely, this means we get ultra-fast vectorized math operations on the elements of a `Series`.

```
max_depths * 10
```

```
Erie          640
Huron        2290
Michigan     2810
Ontario      2440
Superior     4060
dtype: int64
```

You can use most numpy functions directly on a `Series` object (and later, we'll see `DataFrame` objects as well), but pandas also provides access to these numpy functions through the `Series` object methods.

```
np.sin(max_depths)
```

```
Erie         0.920026
Huron        0.329962
Michigan    -0.985151
Ontario     -0.864536
Superior    -0.670252
dtype: float64
```

```
np.mean(max_depths)
```

```
np.float64(244.8)
```

```
max_depths.mean()
```

```
np.float64(244.8)
```

```
#and if you are lazy and just want a bunch of standard stats
max_depths.describe()

# how to keep 2 digits next to the decimal point?
```

```
count      5.000000
mean     244.800000
std      122.713895
min       64.000000
25%      229.000000
50%      244.000000
75%      281.000000
max      406.000000
dtype: float64
```

`Series` objects also support Boolean mask indexing (aka boolean indexing, is a feature in Python NumPy that allows for the filtering of values in numpy arrays):

```
max_depths[max_depths > max_depths.mean()] # pass a condition in the indexing brackets, [], of an array. The condition can be any comparison.

Michigan    281
Superior    406
dtype: int64
```

And so-called "fancy indexing", i.e. using a list or array to specify values to access:

```
max_depths

Erie         64
Huron       229
Michigan    281
Ontario     244
Superior    406
dtype: int64

max_depths[['Erie', 'Ontario']]

Erie        64
Ontario    244
dtype: int64

max_depths['Erie':'Ontario']

Erie         64
Huron       229
Michigan    281
Ontario     244
dtype: int64
```

For `df[[colname(s)]]`, the **interior square brackets** are for **list**, and the **outside square brackets** are **indexing** operator, i.e. you must use double brackets if you select two or more columns.

With one column name, single pair of brackets returns a **Series**, while double brackets return a **dataframe**.

The DataFrame Object

Much like the `Series` is a one-dimensional array of indexed data, a `DataFrame` is a two-dimensional array of indexed data.

You can think of a `DataFrame` as a sequence of `Series` objects all sharing the same index.

```python
avg_depths_dict = {
    'Erie': 19,
    'Huron': 59,
    'Michigan': 85,
    'Ontario': 86,
    'Superior': 149,
}

avg_depths = pd.Series(avg_depths_dict)

avg_depths
```

```
Erie         19
Huron        59
Michigan     85
Ontario      86
Superior    149
dtype: int64
```

```python
# We've already created this series:
max_depths
```

```
Erie         64
Huron       229
Michigan    281
Ontario     244
Superior    406
dtype: int64
```

```python
lakes = pd.DataFrame({'Max Depth (m)': max_depths, 'Avg Depth (m)': avg_depths})
# Or pd.DataFrame({'Max Depth (m)': max_depths_dict, 'Avg Depth (m)': avg_depths_dict})

lakes
```

```
          Max Depth (m)  Avg Depth (m)
Erie                 64             19
Huron               229             59
Michigan            281             85
Ontario             244             86
Superior            406            149
```

Just like the `Series`, a `DataFrame` has an `index` property.

```python
lakes.index
```

```
Index(['Erie', 'Huron', 'Michigan', 'Ontario', 'Superior'], dtype='object')
```

And a `values` property that exposes the underlying `ndarray`.

```
lakes.values
```

```
array([[ 64,  19],
       [229,  59],
       [281,  85],
       [244,  86],
       [406, 149]])
```

And unlike the Series, the DataFrame has a `columns` property, which is also an index.

```
lakes.columns
```

```
Index(['Max Depth (m)', 'Avg Depth (m)'], dtype='object')
```

We can get the shape of a dataframe, just like a numpy ndarray:

```
lakes.shape
```

```
(5, 2)
```

We can do dictionary-style lookups into the dataframe by column name to get a single `Series`:

```
lakes['Max Depth (m)']
```

```
Erie          64
Huron        229
Michigan     281
Ontario      244
Superior     406
Name: Max Depth (m), dtype: int64
```

To select more than one column put a list of column names inside the dictionary-style square brackets:

```
lakes['Max Depth (m)': 'Avg Depth (m)']
```

```
Empty DataFrame
Columns: [Max Depth (m), Avg Depth (m)]
Index: []
```

For `df[[colname(s)]]`, the **interior square brackets** are for **list**, and the **outside square brackets** are **indexing** operator, i.e. you must use double brackets if you select two or more columns.

With one column name, single pair of brackets returns a **Series**, while double brackets return a **dataframe**.

```
lakes
```

```
          Max Depth (m)  Avg Depth (m)
Erie                 64             19
Huron               229             59
Michigan            281             85
Ontario             244             86
Superior            406            149
```

Creating new columns

Once we have a `DataFrame`, creating new columns is done through simple assignment.

```
surface_area = pd.Series({
    'Superior': 82097,
    'Michigan': 57753,
    'Huron': 59565,
    'Erie': 25655,
    'Ontario': 19009,
})

lakes['Surface Area (sq km)'] = surface_area

lakes
```

```
          Max Depth (m)  Avg Depth (m)  Surface Area (sq km)
Erie                 64             19                 25655
Huron               229             59                 59565
Michigan            281             85                 57753
Ontario             244             86                 19009
Superior            406            149                 82097
```

Notice how the index values allowed pandas to "align" the new data with the existing data!

It's also possible to create new columns from existing columns. Say for example we wanted a column to track the difference between the avg depth and max depth. We'll call this the "depth spread".

```
lakes['Depth Spread'] = lakes['Max Depth (m)'] - lakes['Avg Depth (m)']

lakes
```

```
          Max Depth (m)  Avg Depth (m)  Surface Area (sq km)  Depth
Spread
Erie                 64             19                 25655
45
Huron               229             59                 59565
170
Michigan            281             85                 57753
196
Ontario             244             86                 19009
```

```
158
Superior               406            149                  82097
257
```

DataFrames can be created from many different kinds of data structures (`Series` objects, lists, dictionaries, numpy arrays, etc.)

If you don't specify an index explicitly when creating the DataFrame, or you are using data without implicit indexes, pandas will create a `RangeIndex` for you:

```
call_signs = ['WLUW', 'WNUR', 'WBEZ', 'WXRT', 'WFMT']

type(call_signs)

list

frequencies = [88.7, 89.3333, 91.5, 93.1, 98.7]

formats = ['College', 'College', 'Public Radio', 'Adult Album
Alternative', 'Classical']

radio_station_df = pd.DataFrame({'Call Sign': call_signs, 'Frequency':
frequencies, 'Format': formats})

radio_station_df

  Call Sign  Frequency                   Format
0      WLUW    88.7000                  College
1      WNUR    89.3333                  College
2      WBEZ    91.5000             Public Radio
3      WXRT    93.1000  Adult Album Alternative
4      WFMT    98.7000                Classical

radio_station_df[['Frequency']].round(1)

   Frequency
0       88.7
1       89.3
2       91.5
3       93.1
4       98.7
```

Setting the index

You may want to "move" one of the columns to be the index. You can do this with the DataFrame's `set_index` method. By default this returns a new DataFrame with the index replaced with the values in the chosen column.

The `inplace` parameter will make the change to the existing DataFrame rather than returning a new one.

```python
radio_station_df.set_index('Call Sign', inplace=True)
radio_station_df
```

```
           Frequency                     Format
Call Sign
WLUW         88.7000                    College
WNUR         89.3333                    College
WBEZ         91.5000               Public Radio
WXRT         93.1000  Adult Album Alternative
WFMT         98.7000                  Classical
```

```python
# If you want, you can remove the name of index ('Call Sign')
radio_station_df.index.name = None
radio_station_df
```

```
      Frequency                     Format
WLUW    88.7000                    College
WNUR    89.3333                    College
WBEZ    91.5000               Public Radio
WXRT    93.1000  Adult Album Alternative
WFMT    98.7000                  Classical
```

It is possible to move the index back to a column with the `reset_index` method:

```python
radio_station_df.reset_index(inplace=True)
radio_station_df
```

```
  index  Frequency                     Format
0  WLUW    88.7000                    College
1  WNUR    89.3333                    College
2  WBEZ    91.5000               Public Radio
3  WXRT    93.1000  Adult Album Alternative
4  WFMT    98.7000                  Classical
```

Data Indexing and Selection

Now that we can load data into pandas objects, we need to be able to access it. Pandas offers a variety of methods for accessing the data we need.

First, both `Series` and `DataFrame` objects support dictionary-style access with square brackets. Think of index label values as dictionary keys:

```python
# We saw this above -- access a series like a dictionary to get a
single value.
#avg_depths
avg_depths['Michigan']
```

```
np.int64(85)
```

```
# DataFrame dictionary-style access returns the Series with that column index label:
lakes['Avg Depth (m)']
```

```
Erie         19
Huron        59
Michigan     85
Ontario      86
Superior    149
Name: Avg Depth (m), dtype: int64
```

Boolean masking and fancy indexing work with DataFrames, just like Series objects:

```
lakes
```

```
          Max Depth (m)  Avg Depth (m)  Surface Area (sq km)  Depth
Spread
Erie                 64             19                 25655
45
Huron               229             59                 59565
170
Michigan            281             85                 57753
196
Ontario             244             86                 19009
158
Superior            406            149                 82097
257
```

```
# use a Boolean mask to select just the items we want:
lakes[(avg_depths == 59) | (avg_depths == 86)]
```

```
         Max Depth (m)  Avg Depth (m)  Surface Area (sq km)  Depth
Spread
Huron              229             59                 59565
170
Ontario            244             86                 19009
158
```

This works because the Boolean mask creates a new `Series` with the same index values!

```
avg_depths > 60
```

```
Erie        False
Huron       False
Michigan     True
Ontario      True
Superior     True
dtype: bool
```

```python
# There is a potential problem with non-sequential integer indexes:
data_implicit = pd.Series([100, 200, 300, 400])
data_explicit = pd.Series([100, 200, 300, 400], index=[4, 9, 8, 1])
print('data_implicit')
print(data_implicit)
print()
print('data_explicit')
print(data_explicit)
```

```
data_implicit
0    100
1    200
2    300
3    400
dtype: int64

data_explicit
4    100
9    200
8    300
1    400
dtype: int64
```

To handle this potential confusion between label-based and position-based access and make data access easier in general, pandas provides two "indexers": `Series` and `DataFrame` attributes that expose differents ways to access the data.

- `iloc`: always integer position-based
- `loc`: always label-based

```python
data_implicit.iloc[1]
```

```
np.int64(200)
```

```python
data_explicit.iloc[1]
```

```
np.int64(200)
```

```python
#data_implicit.loc[4]  # Note that this should result in an error
```

```python
data_explicit.loc[4] # # Note that this does not result in an error
```

```
np.int64(100)
```

```python
# We can use slices to select more than one value as well. Here, get all values after the first one:
data_implicit.iloc[1:]
```

```
1    200
2    300
3    400
dtype: int64
```

```
# Let's get all rows of the lakes dataframe except the last one:
lakes.iloc[0:-1]
```

```
          Max Depth (m)  Avg Depth (m)  Surface Area (sq km)  Depth
Spread
Erie                 64             19                 25655
45
Huron               229             59                 59565
170
Michigan            281             85                 57753
196
Ontario             244             86                 19009
158
```

These indexers (`.iloc` and `.loc`) take two parameters: the row index values to include, and the *column* index values to include. By default, all columns of a DataFrame are included, but it is possible to retrieve only a subset:

```
lakes
```

```
          Max Depth (m)  Avg Depth (m)  Surface Area (sq km)  Depth
Spread
Erie                 64             19                 25655
45
Huron               229             59                 59565
170
Michigan            281             85                 57753
196
Ontario             244             86                 19009
158
Superior            406            149                 82097
257
```

```
lakes[["Max Depth (m)","Avg Depth (m)"]]["Erie":"Michigan"]
```

```
          Max Depth (m)  Avg Depth (m)
Erie                 64             19
Huron               229             59
Michigan            281             85
```

```
# The first two rows and first two columns only
lakes.iloc[:2, :2]

#How to print the following without using .iloc or .loc?
lakes[["Max Depth (m)", "Avg Depth (m)"]]["Erie": "Huron"]
```

```
       Max Depth (m)  Avg Depth (m)
Erie              64             19
Huron            229             59
```

```
#lakes.loc['Michigan']
lakes.loc[1] #will result in an error
#lakes
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call
last)
File
~/.local/lib/python3.12/site-packages/pandas/core/indexes/base.py:3805
, in Index.get_loc(self, key)
   3804 try:
-> 3805     return self._engine.get_loc(casted_key)
   3806 except KeyError as err:

File index.pyx:167, in pandas._libs.index.IndexEngine.get_loc()

File index.pyx:196, in pandas._libs.index.IndexEngine.get_loc()

File pandas/_libs/hashtable_class_helper.pxi:7081, in
pandas._libs.hashtable.PyObjectHashTable.get_item()

File pandas/_libs/hashtable_class_helper.pxi:7089, in
pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 1

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

KeyError                                  Traceback (most recent call
last)
Cell In[87], line 2
      1 #lakes.loc['Michigan']
----> 2 lakes.loc[1] #will result in an error
      3 #lakes

File
~/.local/lib/python3.12/site-packages/pandas/core/indexing.py:1191, in
_LocationIndexer.__getitem__(self, key)
   1189 maybe_callable = com.apply_if_callable(key, self.obj)
   1190 maybe_callable = self._check_deprecated_callable_usage(key,
maybe_callable)
-> 1191 return self._getitem_axis(maybe_callable, axis=axis)

File
~/.local/lib/python3.12/site-packages/pandas/core/indexing.py:1431, in
_LocIndexer._getitem_axis(self, key, axis)
   1429 # fall thru to straight lookup
   1430 self._validate_key(key, axis)
-> 1431 return self._get_label(key, axis=axis)
```

```
File
~/.local/lib/python3.12/site-packages/pandas/core/indexing.py:1381, in
_LocIndexer._get_label(self, label, axis)
   1379 def _get_label(self, label, axis: AxisInt):
   1380     # GH#5567 this will fail if the label is not present in
the axis.
-> 1381     return self.obj.xs(label, axis=axis)

File
~/.local/lib/python3.12/site-packages/pandas/core/generic.py:4301, in
NDFrame.xs(self, key, axis, level, drop_level)
   4299             new_index = index[loc]
   4300 else:
-> 4301     loc = index.get_loc(key)
   4303     if isinstance(loc, np.ndarray):
   4304         if loc.dtype == np.bool_:

File
~/.local/lib/python3.12/site-packages/pandas/core/indexes/base.py:3812
, in Index.get_loc(self, key)
   3807     if isinstance(casted_key, slice) or (
   3808         isinstance(casted_key, abc.Iterable)
   3809         and any(isinstance(x, slice) for x in casted_key)
   3810     ):
   3811         raise InvalidIndexError(key)
-> 3812     raise KeyError(key) from err
   3813 except TypeError:
   3814     # If we have a listlike key, _check_indexing_error will
raise
   3815     #  InvalidIndexError. Otherwise we fall through and re-
raise
   3816     #  the TypeError.
   3817     self._check_indexing_error(key)

KeyError: 1
```

`loc` accepts the following types of inputs:

- a single label (as above)
- a list or array of labels, e.g. ['a', 'b', 'c']
- a slice object with labels e.g. 'a':'c' (note that contrary to usual python slices, both the start and the stop are **included**!)
- A boolean array
- A callable function with one argument (the calling Series, DataFrame or Panel) that returns valid output for indexing (one of the above)

```
lakes.loc[['Michigan', 'Superior'], ['Max Depth (m)']]
```

It is also possible to assign to the values at the locations you specify with the `iloc` and `loc` indexers! They aren't read-only.

```
import random

random.seed(123)
df = pd.DataFrame(np.random.randint(0, 10, (3, 3)), columns = ['A',
'B', 'C'])
#
https://numpy.org/doc/stable/reference/random/generated/numpy.random.r
andint.html
df
```

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

```
# Assign the value 100 to the cells 0,B and 1,B.
# Remember with label-based access, which `loc` uses, the "stop" of
the slice is **included**.
df.loc[:1, 'B'] = 100
df.iloc[:1, 1] = 400
df
```

```
   A    B  C
0  2  400  0
1  6  100  1
2  5    8  5
```

A few more examples with `.loc()`:

```
lakes['Max Depth (m)'].loc[['Erie', 'Michigan']]
```

```
Erie         64
Michigan    281
Name: Max Depth (m), dtype: int64
```

```
lakes[['Max Depth (m)', 'Avg Depth (m)']].loc[['Erie', 'Michigan']]
```

```
lakes[['Max Depth (m)', 'Avg Depth (m)']].loc['Erie':'Michigan']
```

```
lakes.loc['Erie':'Michigan', ['Max Depth (m)', 'Avg Depth (m)']]
```

Examining Data

While you can manipulate and operate on your data in any way you can dream up, pandas does provide basic descriptive statistics and sorting functionality for you. I **highly** recommend reading the Pandas documentation to see what methods are available and save yourself some work!

The `describe` method is very useful with numeric data:

```
round(lakes.describe(),2)
```

```
       Max Depth (m)  Avg Depth (m)  Surface Area (sq km)  Depth
Spread
count           5.00           5.00                  5.00
5.0
mean          244.80          79.60              48815.80
165.2
std           122.71          47.39              26114.77
77.3
min            64.00          19.00              19009.00
45.0
25%           229.00          59.00              25655.00
158.0
50%           244.00          85.00              57753.00
170.0
75%           281.00          86.00              59565.00
196.0
max           406.00         149.00              82097.00
257.0
```

We can get the highest value for a given Series with `max`:

```
lakes['Max Depth (m)'].max()
```

```
np.int64(406)
```

But what if we wanted the top 2? `sort_values` is the answer:

```
lakes['Max Depth (m)'].sort_values(ascending = False).head(5)
```

```
Superior    406
Michigan    281
Ontario     244
Huron       229
Erie         64
Name: Max Depth (m), dtype: int64
```

This is so common that there is actually a shortcut for it:

```
max_depths.nlargest(2)
```

```
Superior    406
Michigan    281
dtype: int64
```

Which naturally works on DataFrames as well:

```
lakes.nlargest(2, 'Avg Depth (m)')
```

```
          Max Depth (m)  Avg Depth (m)  Surface Area (sq km)  Depth
Spread
Superior            406            149                 82097
257
Ontario             244             86                 19009
158
```

Combining DataFrames

Often you will need to combine data from multiple data sets together. There are three types of combinations in pandas: concatenations and merges (aka joins).

Concatenating means taking multiple DataFrame objects and appending their rows together to make a new DataFrame. In general you will do this when your datasets contain the same columns and you are combining observations of the same type together into one dataset that contains all the rows from all the datasets.

Merging is joining DataFrames together SQL-style by using common values. This is useful when you have multiple datasets with common keys and you want to combine them into one dataset that contains columns from all the datasets being merged.

```
# Concatenation example
df1 = pd.DataFrame({'Site': [1, 2, 3],
                    'Observed Value': [8.1, 5.5, 6.9]})

df2 = pd.DataFrame({'Site': [7, 8, 9],
                    'Observed Value': [10.5, 11.5, 12.0]})

print("df1: ")
df1
```

```
df1:
```

```
   Site  Observed Value
0     1             8.1
1     2             5.5
2     3             6.9
```

```
print("df2: ")
df2
```

```
df2:
```

```
   Site  Observed Value
0     7            10.5
1     8            11.5
2     9            12.0
```

```
print("concatenated along rows: ")
df3 = pd.concat([df1, df2])
```

```
df3
# How to set index?
```

```
concatenated along rows:
```

```
   Site  Observed Value
0     1             8.1
1     2             5.5
2     3             6.9
0     7            10.5
1     8            11.5
2     9            12.0
```

```
print("concatenated along columns: ")
pd.concat([df1, df2], axis = 1)
```

```
concatenated along columns:
```

```
   Site  Observed Value  Site  Observed Value
0     1             8.1     7            10.5
1     2             5.5     8            11.5
2     3             6.9     9            12.0
```

```
# Merge example
df1 = pd.DataFrame({'Site': [3, 1, 2],
                    'Observed Value': [8.1, 5.5, 6.9]})

df2 = pd.DataFrame({'Site': [1, 2, 3, 4],
                    'Temperature': [27.1, 18.2, 29.8, 30.4]})

print("df1: ")
df1
```

```
df1:
```

```
   Site  Observed Value
0     3             8.1
1     1             5.5
2     2             6.9
```

```
print("df2: ")
df2
```

```
df2:
```

```
   Site  Temperature
0     1         27.1
1     2         18.2
2     3         29.8
3     4         30.4
```

```python
print("merged: ")
pd.merge(df1, df2) # inner/intersection
```

```
merged:
```

```
   Site  Observed Value  Temperature
0     3             8.1         29.8
1     1             5.5         27.1
2     2             6.9         18.2
```

```python
print("merged: ")
print(pd.merge(df1, df2, how = 'outer')) # outer/union
```

```
merged:
   Site  Observed Value  Temperature
0     1             5.5         27.1
1     2             6.9         18.2
2     3             8.1         29.8
3     4             NaN         30.4
```