Pandas

What is Pandas?

- *Pandas* is a package built on top of *NumPy*, and provides an efficient implementation of a *DataFrame*.
- *DataFrames* are essentially multidimensional arrays with attached row and column labels, and often with heterogeneous types and/or missing data.
- Pandas implements a number of powerful data operations familiar to users of both database frameworks and spreadsheet programs.

Why Pandas?

- NumPy's `ndarray` data structure provides essential features for the type of clean, well-organized data typically seen in numerical computing tasks.
- However, its limitations become clear when we need more flexibility (e.g., attaching labels to data, working with missing data, etc.) and when attempting to analyze the less structured data available in many forms in the world around us.
- Pandas, and in particular its `Series` and `DataFrame` objects, builds on the NumPy array structure and provides efficient access to these sorts of "data munging" tasks that occupy much of a data scientist's time.

Import Pandas

In [4]:
```
import pandas as pd
```

Pandas Version

In [5]:
```
pd.__version__
```

Out[5]:
```
'1.2.4'
```

In [63]:

```
# import numpy as well for some usage
import numpy as np
```

Pandas Objects

- Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the **rows** and **columns** are identified with **labels** rather than simple integer indices.
- Three fundamental Pandas objects (data structures):
    - Series
    - DataFrame
    - Index

Pandas `Series` Object

A Pandas `Series` is a one-dimensional array of indexed data. It can be created from a list or array as follows:

In [6]:

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

Out[6]:

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

As we see in the output, the `Series` wraps both a sequence of values and a sequence of indices, which we can access with the `values` and `index` attributes. The `values` are simply a familiar NumPy array:

In [7]:

```
data.values
```

Out[7]:

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

In [8]:
```
data.index
```

Out[8]:
```
RangeIndex(start=0, stop=4, step=1)
```

Like with a NumPy array, data can be accessed by the associated index via the familiar Python square-bracket (`[]`) notation:

In [11]:
```
data[0]
```

Out[11]:
```
0.5
```

In [12]:
```
data[1]
```

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

In [13]:
```
data[1:3]
```

Out[13]:
```
1    0.50
2    0.75
dtype: float64
```

As we will see, though, the Pandas `Series` is much more general and flexible than the one-dimensional NumPy array that it emulates.

`Series` as generalized NumPy array

- The essential difference between `Series` object and NumPy array object is the presence of the **index**.
- The Numpy Array has an implicitly defined integer index used to access the values, the Pandas `Series` has an explicitly defined index associated with the values.
- This explicit index definition gives the `Series` object additional capabilities.
- For example, the index need not be an integer, but can consist of values of any desired type.

In [21]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd']) # character type indices
data
```

Out[21]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In [18]:
```
data['b']
```

Out[18]:
```
0.5
```

In [19]:
```
data['a':'c']
```

Out[19]:
```
a    0.25
b    0.50
c    0.75
dtype: float64
```

In [22]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=[2, 5, 3, 7]) # non-contiguous or non-sequential indices
data
```

Out[22]:
```
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

In [23]:
```
data[5]
```

Out[23]:
```
0.5
```

`Series` as specialized dictionary

In [26]:
```python
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
population
```

Out[26]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

By default, a `Series` will be created where the index is drawn from the sorted keys.

In [27]:
```python
# typical dictionary-style item access can be performed
population['California']
```

Out[27]:
```
38332521
```

In [29]:
```python
# Unlike a dictionary, though, the Series also supports array-style operations such as slicing
population['California':'Florida']
```

Out[29]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
dtype: int64
```

Constructing Series objects

The `Series` objects contruct by using the following forms:

`pd.Series(data, index=index)`

where `index` is an optional argument, and `data` can be one of many entities (such as, a list or NumPy array).

In [71]:
```
# From a list or array, in which index defaults to an integer sequence
# --------------------------------------------------------------------

pd.Series([2, 4, 6])
```

Out[71]:
```
0    2
1    4
2    6
dtype: int64
```

In [77]:
```
# From a scalar data, which is repeated to fill the specified index
# -----------------------------------------------------------------

pd.Series(5, index=[100, 200, 300])
```

Out[77]:
```
100    5
200    5
300    5
dtype: int64
```

In [78]:
```
# From a dictionary, in which index defaults to the sorted dictionary keys
# ------------------------------------------------------------------------

pd.Series({2:'a', 1:'b', 3:'c'})
```

Out[78]:
```
2    a
1    b
3    c
dtype: object
```

In each case, the `index` can be explicitly set if a different result is preferred:

In [80]:
```python
# Here, the Series is populated only with the explicitly identified keys.
pd.Series({2:'a', 1:'b', 3:'c'}, index=[3, 2])
```

Out[80]:
```
3    c
2    a
dtype: object
```

Pandas `DataFrame` Object

- `Series` is an analog of a one-dimensional array with flexible indices,
- `DataFrame` is an analog of a two-dimensional array with both flexible row indices and flexible column names.
- Like `Series` object, `DataFrame` can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary.

DataFrame as a generalized NumPy array

`DataFrame` can be considered as a sequence of aligned Series objects. Here, by "aligned" we mean that they share the same index.

In [31]:
```python
# We already have a population series
population
```

Out[31]:
```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

In [32]:

```python
# Create another series with same index as population series
area_dict = {
    'California': 423967,
    'Texas': 695662,
    'New York': 141297,
    'Florida': 170312,
    'Illinois': 149995
}
area = pd.Series(area_dict)
area
```

Out[32]:

```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
dtype: int64
```

In [33]:

```python
# By combining two Series, create the a DataFrame
states = pd.DataFrame({'population': population,
                       'area': area})
states
```

Out[33]:

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

Like the `Series` object, the `DataFrame` has an `index` attribute that gives access to the index labels:

In [34]:

```python
states.index
```

Out[34]: Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='object')

Additionally, the `DataFrame` has a `columns` attribute, which is an `Index` object holding the column labels:

In [35]:
```
states.columns
```

Out[35]: Index(['population', 'area'], dtype='object')

Thus the `DataFrame` can be thought of as a generalization of a two-dimensional NumPy array, where both the rows and columns have a generalized index for accessing the data.

In [50]:
```
# return the area of 'Texas'
states['area']['Texas']
```

Out[50]: 695662

In [51]:
```
# return the population of 'Texas'
states['population']['Texas']
```

Out[51]: 26448193

DataFrame as specialized dictionary

A `DataFrame` can also be thought as a specialization of a dictionary, because, a `DataFrame` maps a column name to a `Series` of column data.

In [45]:
```
# 'area' attribute (column name) returns the Series object containing the areas
states['area']
```

Out[45]:
```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
Name: area, dtype: int64
```

Notice the potential point of confusion here: in a two-dimesnional NumPy array, `data[0]` will return the first row. For a DataFrame, `data['col0']` will return the first column.

Because of this, it is probably better to think about **DataFrames as generalized dictionaries** rather than generalized arrays, though both ways of looking at the situation can be useful.

Constructing DataFrame objects

In [57]:

```
# From a single Series object
# ---------------------------

pd.DataFrame(population, columns=['population'])
```

Out[57]:

	population
California	38332521
Texas	26448193
New York	19651127
Florida	19552860
Illinois	12882135

In [59]:

```
# From a list of dictionaries
# ---------------------------

data = [{'a': i, 'b': 2 * i} for i in range(3)] # list comprehension
pd.DataFrame(data)
```

Out[59]:

	a	b
0	0	0
1	1	2
2	2	4

If some keys in the dictionary are missing, Pandas will fill them in with `NaN` (i.e., "not a number") values.

In [55]:

```
pd.DataFrame([{'a': 1, 'b': 2}, {'b': 3, 'c': 4}])
```

Out[55]:

	a	b	c
0	1.0	2	NaN
1	NaN	3	4.0

In [60]:

```
# From a dictionary of Series objects
# ----------------------------------

pd.DataFrame({'population': population,
              'area': area})
```

Out[60]:

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

In [68]:

```python
# From a two-dimensional NumPy array
# -----------------------------------

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

Out[68]:

	foo	bar
a	0.583383	0.754358
b	0.274044	0.715939
c	0.215631	0.524222

In [69]:

```python
# Note that, if `columns` or `index` is omitted, an integer index will be used for each.
pd.DataFrame(np.random.rand(3, 2))
```

Out[69]:

	0	1
0	0.023630	0.784373
1	0.794738	0.505967
2	0.821421	0.028745

Pandas Index Object

- So far, we have seen both the `Series` and `DataFrame` objects contain an explicit `index` that lets you reference and modify data.
- This `Index` object can be thought of either as an immutable array or as an ordered set.

In [81]:
```
# let's construct an Index from a list of integers:
ind = pd.Index([2, 3, 5, 7, 11])
ind
```

Out[81]:
```
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Index as immutable array

The `Index` in many ways operates like an array.

In [82]:
```
ind[1] # Python indexing notation to retrieve values
```

Out[82]:
```
3
```

In [83]:
```
ind[::2] # Python indexing notation for slicing
```

Out[83]:
```
Int64Index([2, 5, 11], dtype='int64')
```

In [85]:
```
# Index objects also have many of the attributes familiar from NumPy arrays:

print(ind.size, ind.shape, ind.ndim, ind.dtype)
```

```
5 (5,) 1 int64
```

One difference between `Index` objects and NumPy arrays is that indices are immutable–that is, they cannot be modified via the normal means:

In [86]:
```
ind[1] = 0 # tring to modiy but generates error
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-86-acc359bf9bf0> in <module>
----> 1 ind[1] = 0 # tring to modiy but generates error

~/.local/share/virtualenvs/p4ds-notebooks--5YdjQw8/lib/python3.8/site-packages/pandas/core/indexes/base.py in __se
titem__(self, key, value)
   4275     @final
   4276     def __setitem__(self, key, value):
-> 4277         raise TypeError("Index does not support mutable operations")
   4278 
   4279     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

This immutability makes it safer to share indices between multiple DataFrames and arrays,

Index as ordered set

The `Index` object follows many of the conventions used by Python's built-in `set` data structure, so that unions, intersections, differences, and other combinations can be computed in a familiar way:

In [88]:
```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
```

In [90]:
```
indA.intersection(indB)  # intersection
```

Out[90]:
```
Int64Index([3, 5, 7], dtype='int64')
```

In [93]:
```
indA.union(indB) # union
```

Out[93]:
```
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
```

In [95]:
```
indA.symmetric_difference(indB) # symmetric difference
```

Out[95]:
```
Int64Index([1, 2, 9, 11], dtype='int64')
```