Pandas_quick_tour

January 28, 2020

[1]:
```
# My standard magic !  You will see this in almost all my notebooks.

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

# Reload all modules imported with %aimport
%load_ext autoreload
%autoreload 1

%matplotlib inline
```

1 Pandas

VanderPlas Chapter 3, Geron notebook

[2]:
```
import numpy as np
import pandas as pd
```

1.1 Series

A Series is like a NumPy ndarray but with *symbolic indexing* (like a Dictionary)

[3]:
```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

[3]:
```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

You select elements by using members of the index

[4]:
```
print("data at 'b': ", data["b"])
```

```
data at 'b':  0.5
```

A Series looks a little like a Dictionary, but with useful operations like NumPy ndarrays

[5]:
```
data.sum()
```

[5]:
```
2.5
```

You can access the index and the values directly. The values are a NumPy ndarray so you can easily integrate with NumPy.

The Index is it's own type.

[6]:
```
data.index

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

[6]:
```
Index(['a', 'b', 'c', 'd'], dtype='object')
```

[6]:
```
array([0.25, 0.5 , 0.75, 1.  ])
```

[6]:
```
numpy.ndarray
```

Symbolic indexing is super convenient ! No more "parallel" arrays of values and labels.

[7]:
```
data = pd.Series( np.arange(0,3), index=["2018-01-01", "2018-01-02",␣
↪"2018-12-31"])
data.index
```

[7]:
```
Index(['2018-01-01', '2018-01-02', '2018-12-31'], dtype='object')
```

1.2 Pandas DataFrame

A DataFrame (note the capital "F") looks like a table or a 2-D ndarray but with *symbolic indexing* of rows and columns.

Unlike an ndarray, the columns can be different types.

1.2.1 Constructor

There are several ways to construct a DataFrame. Common methods: - list of tuples, each tuple representing a row - list of Dictionaries, each Dictionary representing a row (key/value pairs) - A Dictionary - where keys are column names - values are the Series representing a column - the index of the Series becomes *row* names of the DataFrame

[8]:
```
```

```
prices     = pd.Series( { "FB": 150, "AAPL": 156, "AMZN": 1700, "NFLX": 340,␣
 ↪"GOOG": 1100 } )
full_names = pd.Series( { "FB": "Facebook", "AAPL": "Apple", "AMZN": "Amazon",
                          "NFLX": "Netflix", "GOOG": "Google"
                        } )

stocks = pd.DataFrame({ "price": prices, "name": full_names })
stocks
```

[8]:

	price	name
FB	150	Facebook
AAPL	156	Apple
AMZN	1700	Amazon
NFLX	340	Netflix
GOOG	1100	Google

[9]:
```
ticker = "AMZN"
print( "Ticker {t}: price: {p}, full name: {fn}".format(t=ticker,
                                                        p=stocks.loc[ticker,␣
 ↪"price"],

                                                        fn=stocks.loc[ticker,␣
 ↪"name"]

                                                       )
    )
```

```
Ticker AMZN: price: 1700, full name: Amazon
```

Notice how we indexed symbolically into the DataFrame: `stocks.loc[ticker, "price"]`

More on indexing.

1.2.2 Display

[10]:
```
stocks.head(2)

stocks.tail(3)
```

[10]:

	price	name
FB	150	Facebook
AAPL	156	Apple

[10]:

	price	name
AMZN	1700	Amazon
NFLX	340	Netflix
GOOG	1100	Google

1.2.3 Data Indexing and Selection

VanderPlas

There is more than one way to index, and it gets confusing !

Personally, I just use one-way: the `.loc`indexer

[11]:
```
print("Price column via .price attribute:\n", stocks.price)

# What would happen if "price" were in both the row index and the column index ?
print("\nPrice column via named column (implicitly the column):\n",␣
 ↪stocks["price"])

print("\nPrice column via .loc indexer:\n", stocks.loc[:, "price"])
```

```
Price column via .price attribute:
 FB       150
AAPL     156
AMZN    1700
NFLX     340
GOOG    1100
Name: price, dtype: int64

Price column via named column (implicitly the column):
 FB       150
AAPL     156
AMZN    1700
NFLX     340
GOOG    1100
Name: price, dtype: int64

Price column via .loc indexer:
 FB       150
AAPL     156
AMZN    1700
NFLX     340
GOOG    1100
Name: price, dtype: int64
```

Common "gotcha" Note the difference in return type for the two slightly statements

[12]:
```
print("Arg. is an array with a single element")
stocks.loc[:, ["price"] ]
type(stocks.loc[:, ["price"] ] )

print("Arg. is a singleton")
stocks.loc[:,  "price" ]
```

```
type(stocks.loc[:,  "price" ])
```

Arg. is an array with a single element

[12]:
```
       price
FB       150
AAPL     156
AMZN    1700
NFLX     340
GOOG    1100
```

[12]:
```
pandas.core.frame.DataFrame
```

Arg. is a singleton

[12]:
```
FB       150
AAPL     156
AMZN    1700
NFLX     340
GOOG    1100
Name: price, dtype: int64
```

[12]:
```
pandas.core.series.Series
```

Index alignment Indices are more than a convenient feature for indexing. They have semantic meaning in that they align two Series or DataFrames

[13]:
```
series1 = pd.Series( { "a": 1,   "b": 2,          "d": 4})
series2 = pd.Series( { "a": 10,         "c": 30, "d": 44 })

series1
series2

series1 + series2
series2 + series1
```

[13]:
```
a    1
b    2
d    4
dtype: int64
```

[13]:
```
a    10
c    30
d    44
dtype: int64
```

[13]:
```
a    11.0
b     NaN
c     NaN
d    48.0
dtype: float64
```

[13]:
```
a    11.0
b     NaN
c     NaN
d    48.0
dtype: float64
```

Slices You can use slices in Index !

Note that, unlike in Python, the upper bound is **inclusive**

Slices are particularly useful when you have DatetimeIndex (an index consisting of dates)

[14]:
```
stocks.loc["AMZN": "GOOG"]
```

[14]:

	price	name
AMZN	1700	Amazon
NFLX	340	Netflix
GOOG	1100	Google

Adding a column

[15]:
```
stocks["sector"] = pd.Series( { "FB": "Tech", "AAPL": "Tech", "AMZN": "ConsD",␣
↪"NFLX": "ConsD"})
stocks
```

[15]:

	price	name	sector
FB	150	Facebook	Tech
AAPL	156	Apple	Tech
AMZN	1700	Amazon	ConsD
NFLX	340	Netflix	ConsD
GOOG	1100	Google	NaN

Note the missing values.

1.3 Operations on Series and DataFrames

The usual suspects, plus vector operations like NumPy.

Note that the default `axis=0`, just like NumPy

[16]:
```
stocks.loc[:, "price"].mean()
```

[16]: 689.2

1.3.1 apply

There are lots of methods (hard to remember them all). Don't forget the old standby: `apply`

The `apply` method to applies your own function, either column-wise or row-wise (depending on axis chosen)

[17]:
```
def my_func(s):
    # If you don't understand what is passed (Series or DataFrame), or whether it it row or column, try this:
    print("Type of s is {t}, shape is {sh}".format(t=type(s), sh=s.shape))

    # s is a series
    return s.mean()

stocks.loc[:, ["price"] ].apply(my_func, axis=0)
```

```
Type of s is <class 'pandas.core.series.Series'>, shape is (5,)
```

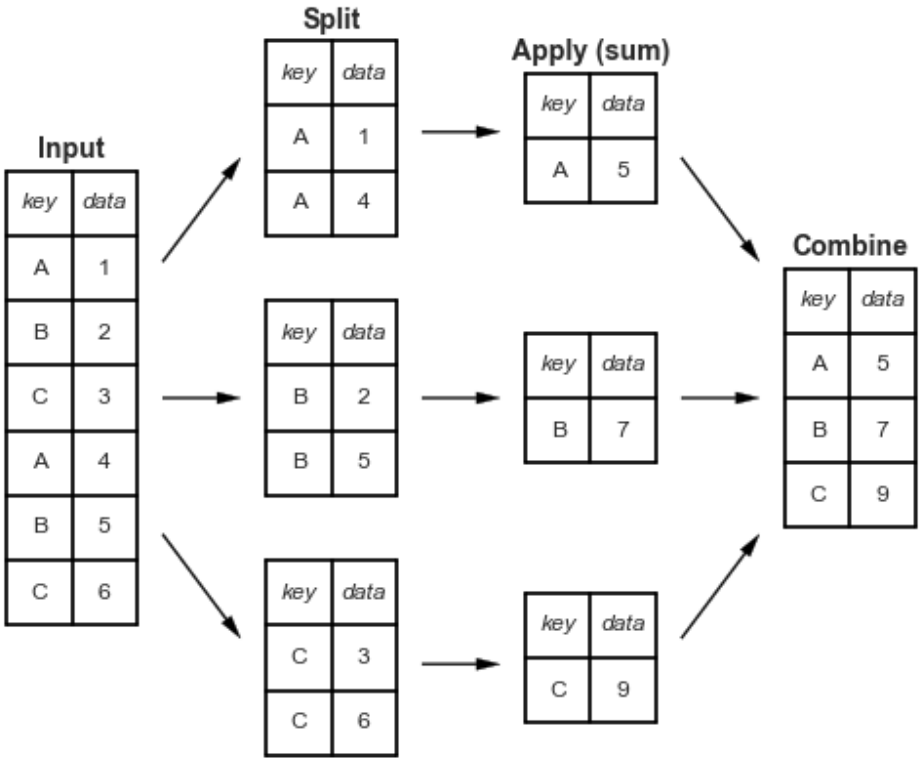
[17]:
```
price    689.2
dtype: float64
```

1.3.2 Database (and Spreadsheet) like operations

Aggregation and grouping VanderPlas

The "split-apply-combine" paradigm (VanderPlas) should be familiar to users of SQL, where the operation is called `group by`

- split the DataFrame into groups by filtering the rows on some criteria
- apply a function to each group, returning another DataFrame
- combine the DataFrame from each group result into a single DataFrame



[18]:
```
stocks.groupby("sector")
```

[18]:
```
<pandas.core.groupby.groupby.DataFrameGroupBy object at 0x7f8505b34e80>
```

[19]:
```
def my_group_func(df):
    return df.loc[:, "price"].mean()

stocks.loc[:, ["price", "sector"] ].groupby("sector").aggregate(my_group_func)
```

[19]:
```
          price
sector
ConsD   1020.0
Tech     153.0
```

Joining, concatenation, pivot VanderPlas

It is very rare that you are given a single dataset organized exactly as you like. Often it comes in separate pieces (i.e., from different database tables or vendors) that must be joined.

You can combine two DataFrames in a manner just like an SQL join of two tables, where the join column is the Index of the two DataFrames. Remember: Index is used for alignment as well as convenience.

We will explore this in our module on Data Transformations.