Python: Pandas

The pandas package provides very efficient data structures and tools for analyzing data. There are two basic data types created by pandas: `Series` and `DataFrame`. The `Series` class is built to store a column of data. That is, one characteristic and multiple observations. The `DataFrame` class is built to store several columns of related data. We will learn how to work with both classes. Pandas is built-on top of NumPy, and there are several other packages that rely on Pandas for data management.

Another interesting package that is built on top of Pandas is the Statsmodels package, which provides functions for statistical estimation (time series, method of moments, linear regression). We won't cover this package during this course, but you should read its documentation if econometrics interests you.

1 Install

Pandas is bundled with Anaconda and should already be installed on your system. In order to use it, you will need to import it:

```
import pandas as pd
```

Notice that we use the alias `pd` for the Pandas package.

2 Series

We start by creating a Pandas Series. The Series object is like a single column of a spreadsheet. Each row of the column is one observation of a single characteristic. Below, we will create a Series to hold the last return of a Stock, and each row will represent the company name:

```
import numpy as np
import pandas as pd

data = np.random.random(5)
print(data)
s = pd.Series(data=data, name='returns')
print(s)
```

You can think of each row as a return observation for a stock. Or each row as representing a different company, and the characteristic is the return for a month. The Series object can only hold data of a single type. The values of the column are stored in a numpy array:

```
print(s.values, type(s.values))
```

However, `s` itself is another class with more features. And only one of its attributes is the numpy array.

```
print(type(s), dir(s))
```

Mathematical operations work as they would with a Numpy array:

```
s + 1
s * 100
np.exp(s)
np.log(s)
```

All mathematical operations are applied element-wise, since they are fundamentally operations done on a numpy array. Notice that these operations use the underlying numpy array to do the computations, but return still return a pandas Series (or DataFrame).

The index of the series (the column of numbers) serves as a way to find the data we want. While Pandas supports indexing with `[]`, it is better to use the method `loc` to get the elements of a Series. To use `loc`, we need to specify the label of the row we want and pass it inside `[]`:

```
s.loc[0]
s.loc[1]
s.loc[-1]  # -1 is not a label!
s.loc[4]
s.loc[0:4]  # pandas always includes the last value
```

The use of square brackets for indexing changes depending on the context. If we are using a Series, then `[]` will index rows, but for a DataFrame (several rows and several columns) `[]` indexes columns. The method `loc` works as expected for both the Series and DataFrame objects.

But, we can change the way the data is indexed. For example, we can change from numbers to text:

```
print(s.index)
s.index = ['SPY', 'AAPL', 'TSLA', 'AMZN', 'COST']
print(s)
print(s.index)
```

Notice that now the type of the index is `object`, instead of `float_`. We can access the rows using the labels with `loc`:

```
s.loc['SPY']
s.loc['AAPL']
s.loc[0]                          # 0 is not a label anymore!
```

But they are more flexible. For example, we can acess multiple rows:

```
s.loc[['SPY', 'AAPL']]
s.loc[['SPY', 'AAPL', 'COST']]
```

We can also use the slicing notation:

```
s.loc['SPY': 'COST']
s.loc['AAPL': 'TSLA']
```

We can use the in operator to test against the series index:

```
print('SPY' in s)
print('GOOG' in s)
```

3 DataFrame

A `DataFrame` is a collection of many `Series`. It is like a spreadsheet, where each column represents one variable and each row an observation.

```
data = np.random.random((5, 3))
df = pd.DataFrame(data=data)
# DataFrame does not take a name argument.
print(type(df))
print(df)
```

Notice that rows and columns were labeled with numbers since we did not specify the labels we wanted to use.

The rows are indexed by numbers:

```
print(df.index)
df.index = ['SPY', 'AAPL', 'TSLA', 'AMZN', 'COST']
print(df)
```

The columns are also indexed by numbers, but we can also change the indexing to represent the characteristics:

```
print(df.columns)
df.columns = ['return', 'last dividend', 'last price']
print(df)
```

To index the elements of a DataFrame we can still use `loc`. Remember that `loc` takes labels, but now we need to specify two sets of labels: one for rows and another for columns.

```
df.loc[:, 'return']
df.loc['SPY':, 'return']
df.loc[:'COST', 'return']
df.loc[:, ['last dividend', 'return']]
df.loc[:, ['last price', 'return']]
df.loc[['AAPL', 'AMZN'], ['last price', 'return']]
```

Notice that the DataFrame columns are reordered depending on the order of the columns.

It is also possible to slice a DataFrame via integers, instead of just relying on labels. To do so, we use the method `iloc`, but instead of specifying labels we specify indices:

```
print(df)
df.iloc[0, :]
df.iloc[:, 0]
df.iloc[0:2, :]  # slicing with indices works as before: excludes last
    integer
df.iloc[2:, :2]
df.iloc[[0, 2, 4], [0, 2]]
```

The method `iloc` is for selectin rows and columns by **integer** indexing.

Notice that for a DataFrame, we can also use `[]` for indexing columns:

```
df["return"]
df[["return", "last dividend"]]
df[:]
```

4 Reading Data

Pandas also implements several methods for reading data into a `Series` or a `DataFrame`. Let's use `pd.read_csv` to read the .csv file for a Stock.

```
import os
import pandas as pd

filename = os.path.join("/", "Volumes", "SD", "Data", "Stocks5Min", "SPY.
    csv")
data = pd.read_csv(filename)
print(data)
```

Notice that the first row of the file was used as headers. To avoid that:

```
pd.read_csv(filename, header=None)
```

How do I know that I need to change the input `header`? By simply reading the functions documentation:

```
help(pd.read_csv)
```

Notice that `pd.read_csv` is a function that allows for several different use cases. We will see some of these cases, but you should read the documentation yourself to see what this function is capable of.

We want to name the columns of our data. To do so:

```
pd.read_csv(filename, header=None, names=["Date", "Time", "Price"])
```

Now the data is being imported into a DataFrame and assigned useful column names.

Notice that the index of the data is a list of integers. We can select another index:

```
data = pd.read_csv(filename, header=None, names=["Date", "Time", "Price"])
data = data.set_index("Date")
```

We can get a summary description of the data by calling the method `describe`:

```
data.describe()
# change number of decimal places displayed
pd.set_option('precision', 2)   # only for printing, not for storing
data.describe()
```

The index of the data is given by the dates in our data. We can get the values in the index by calling:

```
data.index
```

Notice that the index is a list of integers.

5 Dates and Times

Pandas works well with `datetime` objects, which can be used to represent dates and times. We can use the function `pd.to_datetime` to convert data to the `datetime` format:

```
dates_index = pd.to_datetime(data.index, format="%Y%m%d")
print(dates_index)
data = data.set_index(dates_index)
print(data.index)
```

For a full list of possible format codes, see this refence page on Format Codes.

What's the usefulness of this? Well, Pandas has methods for slicing data by dates (and also times), and we can also group data by dates (more on this later).

```
# Select a year of data
data.loc["2007"]
data.loc["2007", :]
# Select a month
data.loc["2007-12"]
# Select a day
data.loc["2007-12-03"]
```

We can also use the index to find the dates directly:

```
data.index.date
data.loc[data.index.date[0]]
# There are many repetitions, so get the values that do not repeat
dates = np.unique(data.index.date)
print(dates)
data.loc[dates[0]]
data.loc[dates[1]]
```

And the dates can be used to slice the data on a range:

```
data.loc[dates[0]:dates[10]]
data.loc["2007":"2008"]
```

Let's incorporate the times into the index.

```
dates = data.index.strftime("%Y%m%d")
times = data["Time"].astype(str)
datetime = pd.to_datetime(dates + times, format="%Y%m%d%H%M")
data = data.set_index(datetime)
data = data.drop(columns=["Time"])
print(data)
```

We can now slice by time:

```
data.loc["2007-01-03 09:35:00":"2007-01-03 10:30:00"]
```

We can also use the datetime indices to create plots with the correct axis labels. Before we move on to plots, notice that the `pd.read_csv` function can parse dates when the data is being imported:

```
data = pd.read_csv(
    filename,
    header=None,
    names=["Date", "Time", "Price"],
    parse_dates=[[0, 1]],  # combine columns 0 and 1 to parse
    index_col=0            # use the first column as the index
)
print(data)
```

Notice that Pandas couldn't identify the date and time format automatically, so we need to specify it ourselves:

```
data = pd.read_csv(
    filename,
    header=None,
    names=["Date", "Time", "Price"],
```

```
    parse_dates=[[0, 1]],  # combine columns 0 and 1 to parse
    index_col=0,           # use the first column as the index
    date_parser=lambda x: pd.datetime.strptime(x, "%Y%m%d %H%M"),
)
print(data)
```

6 Plotting

Pandas is also built on top of Matplotlib. The `Series` and `DataFrame` classes provide methods to automatically plot what is stored in these objects.

```
import matplotlib.pyplot as plt

with plt.style.context("ggplot"):
    ax = data.plot(
        kind="line",
        figsize=(16, 4),
        title="Evolution of the Market Index",
        legend=False,
        grid=True,
        color="black",
    )
    ax.set(xlabel="Time", ylabel="Price (in dollars)")
```

All of the options available in Matplotlib are also available in Pandas.

We can zoom in by indexing the data we want to plot:

```
from matplotlib.dates import DateFormatter

with plt.style.context("ggplot"):
    ax = data.loc["2007"].plot()
    ax.xaxis.set_major_formatter(DateFormatter("%b"))
    ax.axhline(y=data.loc["2007"].mean().values, color="black", linestyle="
    --")
    ax.set(xlabel="Time", ylabel="Price (in dollars)", title="Market Index
    in 2007")
    ax.legend(["SPY", "Average SPY"])
```

7 Panel Data and Merging

Let's load data for another stock:

```
def load_stock(folder, ticker):
    return pd.read_csv(
        os.path.join(folder, f"{ticker}.csv"),
        header=None,
        names=["Date", "Time", "Price"],
        parse_dates=[[0, 1]],  # combine columns 0 and 1 to parse
        index_col=0,  # use the first column as the index
        date_parser=lambda x: pd.datetime.strptime(x, "%Y%m%d %H%M"),
    )


folder = os.path.join("/", "Volumes", "SD", "Data", "Stocks5Min")
```

```
spy = load_stock(folder, "SPY")
csco = load_stock(folder, "CSCO")

prices = pd.merge(
    left=spy, right=csco, left_index=True, right_index=True, suffixes=("
    _SPY", "_CSCO")
)


prices
```

The function `pd.merge` can be used for joining multiple data frames together. In this case, we have two data frames that share indices containing the exact same values. We used these indices to join the data. Given the values in one index (`left_index`), Pandas looks for a unique matching value on the other index (`right_index`), and then joins the values of both columns in a new row. For more information on merging, refer to the Method, join and concatenate reference page.

Let's create a matrix of returns:

```
returns = np.log(prices).diff(axis=0)
print(returns)
```

The `np.log` function comes from Numpy and applies the logarithm element by element on the data frame. The data frame's method `diff` takes differences row by row.

The returns we computed include overnight returns (except for the very first row). Therefore, we need to drop those rows:

```
returns.iloc[::78, :] = np.nan
print(returns.iloc[::78, :])
returns = returns.dropna()
print(returns)
returns.columns = ["Return_SPY", "Return_CSCO"]
print(returns)
```

Another possibility to computing returns is by indexing the data day by day.

8 Group, Apply and Aggregate

Pandas can automatically split the data by the values in a column. For example, the column containing the indices can be used to splits the data day by day.

```
dates = prices.index.date
grouped = prices.groupby(dates)
```

The variable grouped now contains a list-like object. This list is a list of slices of the original data, where each slice is a single date of data. Pandas can slice the data by day because it looks for values of the index that match the values in the `dates` variable. Each value in the `dates` variable is a specific day, which is matched to the values in `returns.index` and allows Pandas to select the correct rows for that day.

We can go over the variable `grouped` directly via a for-loop. Each element of `grouped` is a tuple containing two elements. First, is the value of `dates` that was matched to `returns.index`. Second, is a DataFrame containing all of the rows for the matching values (all of the data for a given day).

```
for date, df in prices.groupby(dates):
    break
print(date, type(date))
print(df)
```

We can then use the DataFrame with that day's data to compute the returns for that day:

```
day_by_day = []
for date, df in prices.groupby(dates):
    day_returns = np.log(df).diff(axis=0)
    day_by_day.append(day_returns)
print(day_by_day[0])
print(day_by_day[-1])
```

We can combine all of the data frames stored in a list with the function `pd.concat`:

```
returns = pd.concat(day_by_day)
returns.columns = ["Return_SPY", "Return_CSCO"]
returns = returns.dropna()
print(returns)
```

Let's use the same idea to compute the realized variances:

```
rv_day_by_day = []
dates = []
for date, df in returns.groupby(returns.index.date):
    rv_day_by_day.append((df ** 2).sum(axis=0))
    dates.append(date)
print(dates[0])
print(rv_day_by_day[0])
rv = pd.DataFrame(data=rv_day_by_day, index=dates)
```

Here we did not use `pd.concat` because the result of computing the realized variance is a Series instead of a DataFrame. Concatenating many Series would lead to a long Series with various repeated labels. We use the `pd.DataFrame` function to create a new DataFrame from the data we just accumulated. Each Series in `rv_day_by_day` will be transformed into a row, and we use the `dates` to create the index.

Notice that there is a pattern here:

- Break up a big data frame into smaller pieces;
- Use each piece to compute some measure of interest;
- Aggregate all results into a DataFrame.

This pattern is built into Pandas. The `grouped` variable has many methods, one of them is `apply`. This method is used to apply a function to each slice of the data frame (i.e., each day), and also takes care of joining all results together.

```
returns = prices.groupby(prices.index.date).apply(lambda df: np.log(df).
    diff(axis=0))
print(returns)
returns = returns.dropna()
returns.columns = ["Return_SPY", "Return_CSCO"]
print(returns)
```

We can use the same idea to compute the realized variance:

```
rv = returns.groupby(returns.index.date).apply(lambda df: (df**2).sum(axis
    =0))
print(rv)
rv.columns = ["RV_SPY", "RV_CSCO"]
print(rv)
```

9 Resampling

So far we have worked with data sampled every 5-minutes or faster. We can use the high-frequency data to compute returns at a lower frequency. To do so, we need to resample our data. We need to split the data day by day, and, for each day, sum up the log-returns to obtain the return for a given day.

```
daily_returns = returns.groupby(returns.index.date).apply(lambda df: df.sum
    (axis=0))
```

When working with data index by time, we can leverage Pandas functions to resample our data to different frequencies. Pandas has a built-in function for resampling a DataFrame in various ways, as long as the index is a list of datetime objects. The resampling functionality of Pandas works in a similar way to `groupby`:

- Split the data, but now we split the data into a new frequency. For example, if the new frequency is hourly, then the new DataFrame will be index on an hour by hour basis. The original data will be split by hour, so if we have 5-minutes observations, we will end up with a list of data frames each containing 12 observations.
- Now we need to use each data frame in the group to compute some measure of interest. If we want to obtain hourly returns, then we would need to sum up all log-returns in an hour.
- The values we computed are then aggregated in a new data frame.

This functionality is available in the method `resample`:

```
# Method 1
daily_returns = returns.resample("D").sum()  # D: calendar days
daily_returns = daily_returns.loc[np.unique(returns.index.date), :]
print(daily_returns)
# Method 2
daily_returns = returns.resample("B").sum()  # B: business days
daily_returns = daily_returns.loc[np.unique(returns.index.date), :]
print(daily_returns)
# Method 3
daily_returns = returns.resample("D").apply(
    lambda df: df.sum() if not df.empty else None
)
daily_returns = daily_returns.dropna()
print(daily_returns)
# Method 4
daily_returns = prices.resample("D").apply(
    lambda df: np.log(df.iloc[0, :] / df.iloc[-1, :]) if not df.empty else
    None
)
daily_returns = daily_returns.dropna()
print(daily_returns)
```

See the Frequency (Offset Aliases) reference page for the possible frequencies that can be used with the `resample` method. For a more in depth understanding of these topics you should read the Time Series and Date functionality reference page.

10 MultiIndex

We have stored different variables in different data frames, but we can put all of the data together (or most of it) in a single data frame. To do so, we need the data to have the same indices, and to use a MultiIndex object to store the columns. Let's load stock prices and returns for two stocks:

```
def load_stock(folder, ticker):
    return pd.read_csv(
        os.path.join(folder, f"{ticker}.csv"),
        header=None,
        names=["Date", "Time", "Price"],
        parse_dates=[[0, 1]],  # combine columns 0 and 1 to parse
        index_col=0,  # use the first column as the index
        date_parser=lambda x: pd.datetime.strptime(x, "%Y%m%d %H%M"),
    )


folder = os.path.join("/", "Volumes", "SD", "Data", "Stocks5Min")
spy = load_stock(folder, "SPY")
csco = load_stock(folder, "CSCO")
prices = pd.merge(left=spy, right=csco, left_index=True, right_index=True,
    suffixes=("_SPY", "_CSCO"))
returns = prices.groupby(prices.index.date).apply(lambda df: np.log(df).
    diff(axis=0))
returns.columns = ["Return_SPY", "Return_CSCO"]
```

Both data frames contain data about two stocks, SPY and CSCO, but one data frame has information about prices and the other has information about returns. The idea behind a `MultiIndex` is to split column labels in levels. In our case, the first level would be the stock tickers, and the second level would be the information about each stock (prices and returns). This allows us to put the entire data in a single data frame, which can be useful for doing computations on a large set of data, for storing your data set, or even for sharing results in a table format. To create a `MultiIndex` we need to specify what are the labels on each level:

```
columns = pd.MultiIndex.from_product(
    [["SPY", "CSCO"], ["Price"]], names=["Stock", "Information"]
)
print(columns)
```

We can use this `MultiIndex` to substitute our column names:

```
prices.columns = columns
print(prices)
```

Our columns are now divided by stock, and for each stock we have a further division for price. Notice that we gave names to our columns' names: Stock refers to the different tickers we have, and Information refers to the types of data we have (which is just one for now).

Let's create another `MultiIndex` for the returns:

```
columns = pd.MultiIndex.from_product(
    [["SPY", "CSCO"], ["Return"]], names=["Stock", "Information"]
)
print(columns)
returns.columns = columns
print(returns)
```

We can now combine both data frames into one:

```
stocks = pd.merge(left=prices, right=returns, left_index=True, right_index=
    True)
print(stocks)
```

Let's organize the data by the stock ticker:

```
stocks = stocks.sort_index(axis=1, level="Stock")
```

The MultiIndex is a **hierarchical structure**, and the order it follows was the order that was given when it was constructed. In this case, the labels first in the hierarchy are the `Stock` names, SPY and CSCO, next are the `Information` names, Price and Return.

We can select columns of the data by using the slicing operator:

```
print(stocks.loc[:, "SPY"])
```

The code above goes into the column of the stock SPY, and displays all the values for this country. We go down 1-level in the hierarchy, and now there are only two divisions for columns, the first is for Price and the second for Return.

```
print(stocks.loc[:, ("SPY", "Price")])
print(stocks.loc[:, ["SPY", "Return"]])
print(stocks.loc[:, ("SPY", ("Price", "Return"))])
```

Notice the use of a tuple to specify a complete **multi-level** key, where each element of the tuple specifies the keys for one level.

It is possible to select all values of a level in a MultiIndex by using the syntax `slice(None)`, similar to how `:` works with lists.

```
print(stocks.loc[:, (slice(None), "Return")])
```

We can even add more stock data to our data frame:

```
aapl = load_stock(folder, "AAPL")
aapl["Return"] = aapl.groupby(aapl.index.date).apply(lambda df: np.log(df).
    diff(axis=0))
aapl.columns = pd.MultiIndex.from_product(
    [["AAPL"], ["Price", "Return"]], names=["Stock", "Information"]
)
stocks = pd.merge(left=stocks, right=aapl, left_index=True, right_index=
    True)
print(stocks)
```

At this point we have a data structure that contains most of the information we need, and which is a good starting point for computing other estimators.

As an example, let's compute the realized variance for all stocks:

```
rv = (
    stocks.loc[:, (slice(None), "Return")]
    .groupby(stocks.index.date)
    .apply(lambda df: (df ** 2).sum(axis=0))
```

```
)

tickers = rv.columns.get_level_values(level="Stock")
rv.columns = pd.MultiIndex.from_product(
    [tickers, ["Realized Variance"]], names=["Stock", "Information"]
)
print(rv)
```

You can repeat the process for the bipower variance, for example, and then aggregate the resulting data frames into a single frame.