ALBUKHARY INTERNATIONAL UNIVERSITY

Artificial Intelligence

LAB Four

Pandas Library

Pandas is a Python library in Python that allows for you to easily work with tabular data that is often stored in Excel files or .csv files. It is often considered one of the most vital Python libraries for data analysis because of its ease of use.

It has functions for analyzing, cleaning, exploring, and manipulating data.

The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Features of Pandas

- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets.
- Label-based slicing, indexing and subsetting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality

Installation of Pandas

Pandas is an easy package to install. Install it using this command:

In [ ]:
```
! pip install pandas
```

The ! at the beginning runs cells as if they were in a terminal.

To import pandas we usually import it with a shorter name since it's used so much:

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

Core components of pandas: Series and DataFrames

The primary two components of pandas are the **1. Series**. **2. DataFrame**.

1. Series

A Pandas Series is like a column in a table. It is a one-dimensional array holding data of any type. The row labels of series are called the index. We can easily convert the list, tuple, and dictionary into series using "series' method. A Series cannot contain multiple columns. It has one parameter:

Data: It can be any list, dictionary, or scalar value.

Creating Series

In [ ]:
```
import pandas as pd

x = ["Tamim", "Afrah", "Atteib"]

y = pd.Series(x)

y
```

Labels

If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.

This label can be used to access a specified value.

In [ ]:
```
print(y[0])
```

Create Labels

With the index argument, you can name your own labels.

In [ ]:
```
import pandas as pd

x = ["Tamim", "Afrah", "Atteib"]

y = pd.Series(x, index = ["a", "b", "c"])

print(y)
```

When you have created labels, you can access an item by referring to the label.

In [ ]:
```
print(y["b"])
```

2. DataFrame

It is a widely used data structure of pandas and works with a two-dimensional array with labeled axes (rows and columns). DataFrame is defined as a standard way to store data and has two different indexes, i.e., row index and column index. It consists of the following properties:

1. The columns can be heterogeneous types like int, bool, and so on.
2. It can be seen as a dictionary of Series structure where both the rows and columns are indexed. It is denoted as "columns" in case of columns and "index" in case of rows.

Series is like a column, a DataFrame is the whole table.

Creating DataFrame

There are many ways to create a DataFrame from scratch, but a great option is to just use a simple dict.

In [ ]:
```
data = {
    'Names': ["Tamim", "Afrah", "Atteib"],
    'Age': [23, 20, 22]
}
```

And then pass it to the pandas DataFrame constructor:

In [ ]:
```
MyData = pd.DataFrame(data)
```

In [ ]:
```
MyData
```

Each (key, value) item in data corresponds to a column in the resulting DataFrame.

The **Index** of this DataFrame was given to us on creation as the numbers 0-3, but we could also create our own when we initialize the DataFrame

How to Save DataFrame to CSV

Often times when you convert your data into a DataFrame, you will process it and then ultimately save it to disk. To do this, we have a few different options, such as CSV and JSON. To save your DataFrame to a CSV file, you can write the following command:

In [ ]:
```
MyData.to_csv("names.csv")
```

How to Read DataFrame from CSV

Now that we have the data saved to a CSV file, let us create a new DataFrame, name2, and read that data. We can do this with the command pd.read_csv(). As with to_csv, we can pass multiple arguments here, but for now, we will stick with the one mandatory one, a string of the file that we wish to open. In this case, it is the same file we just created. Let's open it and print it off.

In [ ]:
```
MyData2 = pd.read_csv("names.csv")
```

In [ ]:
```
MyData2
```

Notice that this DataFrame looks a bit off from what we saved to disk. Why is that? It is because of how we saved the file. If we donnot specify an index, Pandas will automatically create one for us. In order to correctly save our file, we need to pass an extra keyword argument, specifically index=False. Let us try and save this file again under a different name: "names_no_index.csv".

In [ ]:
```
MyData.to_csv("names_no_index.csv", index=False)
```

Let's create a new DataFrame, MyData3, and reopen and print off the data.

In [ ]:
```
MyData3 = pd.read_csv("names_no_index.csv")
```

In [ ]:
```
MyData3
```

important DataFrame operations

DataFrames possess hundreds of methods and other operations that are crucial to any analysis. As a beginner, you should know the operations that perform simple transformations of your data and those that provide fundamental statistical analysis.

In [ ]:
```
customers = pd.read_csv("Mall_Customers.csv")
```

We're loading this dataset from a CSV and designating the CustomerID to be our index.

Viewing your data

The first thing to do when opening a new dataset is print out a few rows to keep as a visual reference. We accomplish this with **.head()**:

In [ ]:
```
customers.head()
```

.head() outputs the first five rows of your DataFrame by default, but we could also pass a number as well: *customers.head(10)* would output the top ten rows, for example.

In [ ]:
```
customers.head(10)
```

To see the last five rows use **.tail()**.

tail() also accepts a number, and in this case we printing the bottom two rows.:

In [ ]:
```
customers.tail(5)
```

Typically when we load in a dataset, we like to view the first five or so rows to see what's under the hood. Here we can see the names of each column, the index, and examples of values in each row.

Getting info about your data

.info() should be one of the very first commands you run after loading your data:

In [ ]:
```
customers.info()
```

.info() provides the essential details about your dataset, such as the number of rows and columns, the number of non-null values, what type of data is in each column, and how much memory your DataFrame is using.

Another fast and useful attribute is **.shape**, which outputs just a tuple of (rows, columns):

In [ ]:
```
customers.shape
```

Note that **.shape** has no parentheses and is a simple tuple of format (rows, columns). So we have 200 rows and 5 columns in our movies DataFrame.

You will be going to **.shape** a lot when cleaning and transforming data. For example, you might filter some rows based on some criteria and then want to know quickly how many rows were removed.

Handling duplicates

This dataset does not have duplicate rows, but it is always important to verify you aren't aggregating duplicate rows.

To demonstrate, let us simply just double up our customer DataFrame by appending it to itself:

In [ ]:
```
customer_df = customers.append(customers)

customer_df.shape
```

Using **append()** will return a copy without affecting the original DataFrame. We are capturing this copy in customer so we are not working with the real data.

Notice call **.shape** quickly proves our DataFrame rows have doubled.

Now we can try dropping duplicates:

In [ ]:
```
customer_df = customers.append(customers)  # make a new copy

customer_df.drop_duplicates(inplace=True, keep=False)

customer_df.shape
```

Just like **append()**, the **drop_duplicates()** method will also return a copy of your DataFrame, but this time with duplicates removed. Calling .shape confirms we're back to the 200 rows of our original dataset.

It's a little verbose to keep assigning DataFrames to the same variable like in this example. For this reason, pandas has the inplace keyword argument on many of its methods. Using **inplace=True** will modify the DataFrame object in place:

In [ ]:
```
customer_df.drop_duplicates(inplace=True)
```

Now our *customer_df* will have the transformed data automatically.

Another important argument for **drop_duplicates()** is keep, which has three possible options:

- **first:** (default) Drop duplicates except for the first occurrence.
- **last:** Drop duplicates except for the last occurrence.
- **False:** Drop all duplicates.

Since we did not define the keep arugment in the previous example it was defaulted to first. This means that if two rows are the same pandas will drop the second row and keep the first row. Using last has the opposite effect: the first row is dropped.

keep, on the other hand, will drop all duplicates. If two rows are the same then both will be dropped. Watch what happens to customer_df:

In [ ]:
```
customer_df = customers.append(customers)  # make a new copy

customer_df.drop_duplicates(inplace=True, keep=False)

customer_df.shape
```

Since all rows were duplicates, keep=False dropped them all resulting in zero rows being left over. If you arre wondering why you would want to do this, one reason is that it allows you to locate all duplicates in your dataset. When conditional selections are shown below you will see how to do that.

In [ ]:
```
customers.columns
```

Not only does **.columns** come in handy if you want to rename columns by allowing for simple copy and paste, it's also useful if you need to understand why you are receiving a Key Error when selecting data by column.

We can use the **.rename()** method to rename certain or all columns via a dict.

In [ ]:
```
customers.rename(columns={
        'Annual Income (k$)': 'Annual Income'
    }, inplace=True)


customers.columns
```

How to work with missing values

When exploring data, you will most likely encounter missing or null values, which are essentially placeholders for non-existent values.

There are two options in dealing with nulls:

1. Get rid of rows or columns with nulls
2. Replace nulls with non-null values, a technique known as imputation

Let us calculate to total number of nulls in each column of our dataset. The first step is to check which cells in our DataFrame are null:

In [ ]:
```
customers.isnull()
```

Notice **isnull()** returns a DataFrame where each cell is either True or False depending on that cell's null status.

To count the number of nulls in each column we use an aggregate function for summing:

In [ ]:
```
customers.isnull().sum()
```

.isnull() just by iteself is not very useful, and is usually used in conjunction with other methods, like sum().

We can see now that our data has 8 missing values for Annual Income.

Removing null values

Data Scientists and Analysts regularly face the dilemma of dropping or imputing null values, and is a decision that requires intimate knowledge of your data and its context. Overall, removing null data is only suggested if you have a small amount of missing data.

Remove nulls is pretty simple:

In [ ]:
```
customers.dropna()
```

This operation will delete any row with at least a single null value, but it will return a new DataFrame without altering the original one. You could specify **inplace=True** in this method as well.

So in the case of our dataset, this operation would remove 8 rows where Annual Income is null.

This obviously seems like a waste since there is perfectly good data in the other columns of those dropped rows. That is why we will look at imputation next.

Other than just dropping rows, you can also drop columns with null values by setting **axis=1:**

In [ ]:
```
customers.dropna(axis=1)
```

Imputation

Imputation is a conventional feature engineering technique used to keep valuable data that have null values.

There may be instances where dropping every row with a null value removes too big a chunk from your dataset, so instead we can impute that null with another value, usually the **mean** or the **median** of that column.

Let us look at imputing the missing values in the Annual Income column. First we will extract that column into its own variable:

In [ ]:
```
income = customers['Annual Income']
```

Using square brackets is the general way we select columns in a DataFrame.

The *income* now contains a Series:

In [ ]:
```
income.head()
```

We will impute the missing values of revenue using the **mean**. Here is the **mean** value:

In [ ]:
```
income_mean = income.mean()

income_mean
```

With the mean, let us fill the nulls using **fillna()**:

In [ ]:
```
income.fillna(income_mean, inplace=True)
```

We have now replaced all nulls in *income* with the mean of the column.

Notice that by using **inplace=True** we have actually affected the original customers:

In [ ]:
```
customers.isnull().sum()
```

Understanding our Variables

Using **describe()** on an entire DataFrame we can get a summary of the distribution of continuous variables:

In [ ]:
```
customers.describe()
```

Understanding which numbers are continuous also comes in handy when thinking about the type of plot to use to represent your data visually.

.describe() can also be used on a categorical variable to get the count of rows, unique count of categories, top category, and freq of top category:

In [ ]:
```
customers["Genre"].describe()
```

This tells us that the genre column has 2 unique values, the top value is Female, which shows up 112 times (freq).

.value_counts() can tell us the frequency of all values in a column:

In [ ]:
```
customers['Genre'].value_counts().head(10)
```

Relationships between Continuous Variables

By using the correlation method **.corr()** we can generate the relationship between each continuous variable:

In [ ]:
```
customers.corr()
```

Correlation tables are a numerical representation of the bivariate relationships in the dataset.

Positive numbers indicate a positive correlation — one goes up the other goes up — and negative numbers represent an inverse correlation — one goes up the other goes down. 1.0 indicates a perfect correlation.

DataFrame Slicing, Selecting, Extracting

Below are some methods of slicing, selecting, and extracting you will need to use constantly.

It's important to note that, although many methods are the same, DataFrames and Series have different attributes, so you will need be sure to know which type you are working with or else you will receive attribute errors.

Let us look at working with columns first.

By column

You already saw how to extract a column using square brackets like this:

In [ ]:
```
Genre_col = customers['Genre']

type(Genre_col)
```

This will return a Series. To extract a column as a DataFrame, you need to pass a list of column names. In our case that is just a single column:

In [ ]:
```
Genre_col = customers[['Genre']]

type(Genre_col)
```

Since it's just a list, adding another column name is easy:

In [ ]:
```
subset = customers[['Genre', 'Annual Income']]

subset.head()
```

Now we will look at getting data by rows.

By rows

For rows, we have two options:

1. **.loc**: Locates by name
2. **.iloc**: Locates by numerical index

In [ ]:
```
# Creating the DataFrame
data1 = pd.DataFrame({'Name':['Tamim', 'Abdirrahman', 'Maryam', 'Osman', 'Omar'],
                      'Age':[24, 23, 22, 22, 23]})
# Create the index
index1 = ['A', 'B', 'C', 'D', 'E']
# Set the index
data1.index = index1
```

In [ ]:
```
data1
```

In [ ]:
```
Data_rows = data1.loc["B","Name"]

Data_rows
```

On the other hand, with iloc we give it the numerical index of Name:

In [ ]:
```
customer = data1.iloc[1]
customer
```

loc and **iloc** can be thought of as similar to Python list slicing.

In [ ]:
```

```