

# Intro to Pandas for Data Analytics

PROFESSOR LESLIE ALBERT  
SAN JOSE STATE UNIVERSITY

In this tutorial, you will learn how to use Python Pandas library to explore data. Participants will need a Gmail account to access Google Colaboratory. You'll need a computer and browser, a GMAIL account, and the provided data: TableauSalesData.xlsx.

## INTRO TO PANDAS FOR DATA ANALYSIS PLAN

1. Learn how to open Google Colab, upload a data file, import Pandas and data
2. Introduce project goals: investigate underperforming sub-categories in terms of profits for Office Solutions, an online office supplies retailer.
3. Preview data and columns, use the .describe method to see data descriptive statistics
4. Use the .unique method to see values inside Sub-Category columns
5. Determine Sub-Category Profits and identify those with negative Profits
6. Look at individual products within the Tables Sub-Category and determine their Profits and Sales
7. Look at regional and customer Segment effects on Table Profits and Sales
8. Look at average Discounts on products in the Tables Sub-Category along with Profits
9. Look at the correlation between Discounts and Sales, Discounts and Profits
10. Look at yearly differences in Tables Profits and Sales
11. Summarize findings, make a recommendation

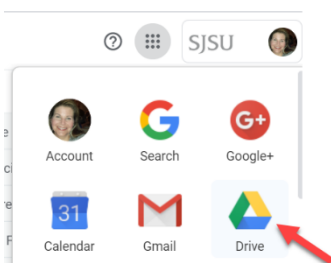
**BACKGROUND:** An office supply retailer, Office Solutions, has recently hired you to help analyze their sales data. Your boss, Natasha, is looking to boost profits and sales. One suggestion to improve profits is to get rid unprofitable or low profit products. As such, she has asked you to determine low performing sub-categories of merchandise and has given you a spreadsheet, TableauSalesData.xlsx, with 4 years of sales data to analyze. As you go through the exercises below, look beyond the code and pay attention to how I'm drilling down into the data, exploring it from many different angles to build a "story," a logical flow of data insights that builds a case for my recommendation.

## PART 1A: IF YOU'VE NEVER USED GOOGLE COLABORATORY (GOOGLE COLAB).

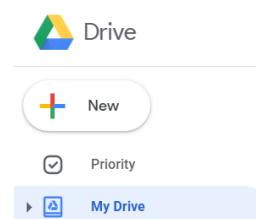
Before you can use Colab you need to add it to your Google Drive.

1. Navigate to: <https://colab.research.google.com>
2. Open a Colab Notebook by clicking **Google Drive** and selecting **New Notebook**.
3. Once the notebook is open, go back to File and select Save the Notebook to Drive, this will create a copy of the file stored in your own Google drive. You will now see Colab as an option under the New button under the More option (see Part 1B below)
4. If you don't see Colab listed under the New Button you may need to refresh your Drive window,
5. The next time you want to use Colab, skip this step and start with Part 1B, below.

## PART 1B: GETTING STARTED WITH COLAB



1. Download the Excel file "TableauSalesData.xlsx" from seminar website
2. Open your Gmail account and under "Apps" select Google Drive
3. In Drive, click the "New" button on the left, scroll down to "More" and select "Google Colaboratory."



## PART 2: IMPORTING DATA

1. In Colab, click the folder icon on the top left, then click "Upload." Browse to where you saved the `TableauSalesData.xlsx` file and click "Open"
2. Right click on the `TableauSalesData.xlsx` file and "copy path" (this is the name and location of the file).
3. Add a code cell by clicking the + Code at the top of the page and type the code below into it. Replacing the location & file name in the example below with your file path, be sure to keep it in quotes. Click the arrow to run your code. Please note that every time you end then restart a Colab session you'll need to re-upload your data file...it's not saved but your code is. You may also need to rerun code cells that build **dataframes** you want to use.



FIGURE 1: IMPORT DATA

4. So what's happening in the code in Figure 1 above? We are using the **Python pandas library** to import data from an Excel spreadsheet and create a **dataframe**. Note the abbreviation of pandas as `pd`...this is convention and the abbreviation is called an "alias." The next line, we create `xl` and use it to tell pandas where the data is. The easiest way to get the file path is to right click on your file icon on the left (`TableauSalesData.xlsx`) and select copy path then control + V to paste it in the parentheses. Don't forget the quotes. Note you could combine lines 2 and 3 in the above if you wish. We are reading in our data with `ExcelFile` but there are other ways to import data including `pd.read_excel()` or converting our Excel file into a `.csv` and using `read_csv()` though the latter could alter your data. You may import the data however you like as long as you end up with a dataframe we're good.
5. Now let's print the first 10 rows of our data to take a look at it by adding the code `print(SalesData.head(10))` and clicking the run arrow – see Figure 2 below. The `.head()` prints from the top row of our dataframe. The default is 5 rows but we can specify more or less. If you've never used Python before you should also know, it is case sensitive. This means Python sees `ExcelFile` and `Excelfile` as different words. And if you write `Print` instead of `print` for a print statement you'll get an error, so be careful when reproducing the code provided. Also, you can name your dataframe something else but it can help to have names that tell you what's in your dataframes and Python won't allow spaces in variable names.

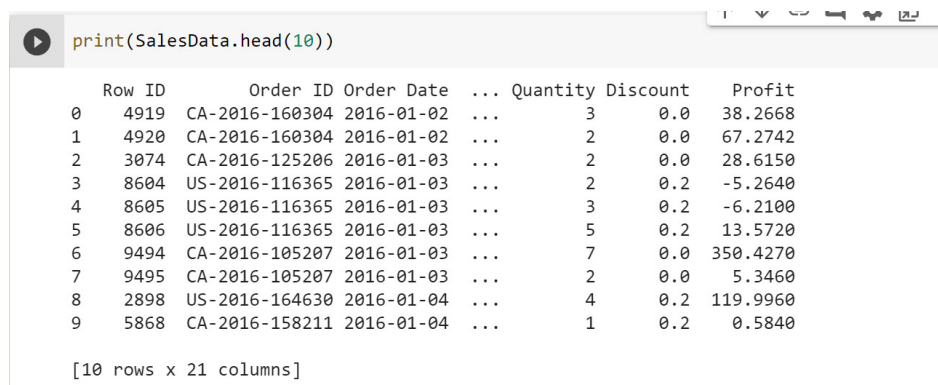


FIGURE 2: PRINTING THE FIRST 10 ROWS IN OUR DATAFRAME

You can't see all of the columns but you can get an idea of the data in our dataframe. A **dataframe** is a Pandas Data Structure but you can think of it like a **spreadsheet**. It has columns of data (called series) with column labels and rows identified in this case by index numbers there on the far left. A better option for viewing dataframes is the display function, illustrated below in Figure 3. My screen capture couldn't show all of it but display shows more of the data and formats it much more nicely than does print. Sometimes print is better, sometimes display is better. I'll use both in this tutorial.

display(SalesData.head(10))

	Row ID	Order ID	Order Date	Ship Date	Ship Mode	Customer ID	Customer Name	Segment	Country	City	State	Postal Code	Region	Product ID	Category	Sub-Category	Product Name	Sales	Qua
0	4919	CA-2016-160304	2016-01-02	2016-01-07	Standard Class	BM-11575	Brendan Murry	Corporate	United States	Gaithersburg	Maryland	20877	East	FUR-BO-10004709	Furniture	Bookcases	Bush Westfield Collection Bookcases, Medium Ch...	173.940	
1	4920	CA-2016-160304	2016-01-02	2016-01-07	Standard Class	BM-11575	Brendan Murry	Corporate	United States	Gaithersburg	Maryland	20877	East	TEC-PH-10000455	Technology	Phones	GE 30522EE2	231.980	
2	3074	CA-2016-125206	2016-01-03	2016-01-05	First Class	LR-16915	Lena Radford	Consumer	United States	Los Angeles	California	90045	West	OFF-ST-10003692	Office Supplies	Storage	Recycled Steel Personal File for Hanging File ...	114.460	
3	8604	US-2016-116365	2016-01-03	2016-01-08	Standard Class	CA-12310	Christine Abelman	Corporate	United States	San Antonio	Texas	78207	Central	TEC-AC-10002217	Technology	Accessories	Imation Clip USB flash drive - 8 GB	30.080	
4	8605	US-2016-116365	2016-01-03	2016-01-08	Standard Class	CA-12310	Christine Abelman	Corporate	United States	San Antonio	Texas	78207	Central	TEC-AC-10002942	Technology	Accessories	WD My Passport Ultra 1TB Portable External Har...	165.600	

FIGURE 3: DISPLAYING FIRST 5 ROWS

## PART 3: LEARNING ABOUT THE DATA

1. **LIST OF COLUMNS.** Add a new code cell then type the one line of code below to see the column names. When you use the arrow to run your cell you should see output like Figure 4, below. Seeing these is really helpful because we'll be pulling data from our dataframe using column names...these names must be exact, even capitalized the same as they are in these results:

```
print(SalesData.columns)
```

Index(['Row ID', 'Order ID', 'Order Date', 'Ship Date', 'Ship Mode', 'Customer ID', 'Customer Name', 'Segment', 'Country', 'City', 'State', 'Postal Code', 'Region', 'Product ID', 'Category', 'Sub-Category', 'Product Name', 'Sales', 'Quantity', 'Discount', 'Profit'], dtype='object')

FIGURE 4: .COLUMNS() FUNCTION

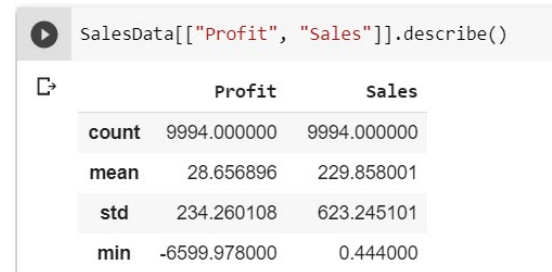
2. **.DESCRIBE FOR DATA DESCRIPTIVE STATISTICS:** Add a new code cell then type and run the following code: SalesData.describe() and you'll see the output in Figure 5.

	Row ID	Postal Code	Sales	Quantity	Discount	Profit
count	9994.000000	9994.000000	9994.000000	9994.000000	9994.000000	9994.000000
mean	4997.500000	55190.379428	229.858001	3.789574	0.156203	28.656896
std	2885.163629	32063.693350	623.245101	2.225110	0.206452	234.260108
min	1.000000	1040.000000	0.444000	1.000000	0.000000	-6599.978000
25%	2499.250000	23223.000000	17.280000	2.000000	0.000000	1.728750
50%	4997.500000	56430.500000	54.490000	3.000000	0.200000	8.666500
75%	7495.750000	90008.000000	209.940000	5.000000	0.200000	29.364000
max	9994.000000	99301.000000	22638.480000	14.000000	0.800000	8399.976000

FIGURE 5: .DESCRIBE()

The `.describe()` is a predefined function that runs descriptive statistics on the numerical values in your dataframe. But not all of it makes sense. For example, what is the average of postal codes? Pandas see postal codes (zip codes) as numbers and treats them as such. Now this could be in issue in other applications of this data. There are states with postal codes that begin with zeros, like those in the northeast US like Connecticut, Massachusetts, Maine. When we treat a postal code like a number, they're really text with no numerical value, that leading zero is dropped. It's unlikely that postal codes without leading zeros will cause issues in our project but you should be aware of them.

3. **.DESCRIBE()** FOR SPECIFIC COLUMNS: If we are interesting in just seeing specific columns we can specify them after the dataframe name as I did in Figure 6. If you have more than 1 column you want to describe, please be sure to include two sets of square brackets or you'll get an error. The inner set of `[]` is because we are passing the function a list rather than a single value...more on that later.



```
SalesData[['Profit', 'Sales']].describe()
```

	Profit	Sales
count	9994.000000	9994.000000
mean	28.656896	229.858001
std	234.260108	623.245101
min	-6599.978000	0.444000

FIGURE 6: .DESCRIBE ON SPECIFIC COLUMNS

4. **.UNIQUE()** TO GET A LIST OF UNIQUE VALUES IN A COLUMN.

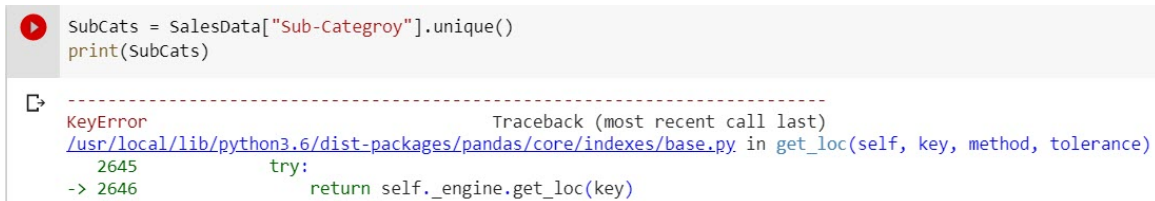
We've been given the challenge of identifying poor performing sub-categories so let's look at the types of sub-categories we have. I've created a new code cell, see Figure 7. In it I've asked Pandas to use the `.unique()` method. This is a predefined function that returns all the unique values in the column we specify and saves them in a numpy array, I've called SubCats. Numpy is another Python library that Pandas is built on top of. If you've never heard of an array, don't worry, it's like a list, just less flexible - all of the elements in an array have to be of the same data type. I have also printed my SubCats array so I can see all the sub-categories.

```
[ ] SubCats = SalesData["Sub-Category"].unique()
    print( SubCats)
```

```
[ ] ['Bookcases' 'Phones' 'Storage' 'Accessories' 'Tables' 'Binders' 'Copiers'
     'Art' 'Furnishings' 'Paper' 'Envelopes' 'Chairs' 'Fasteners' 'Appliances'
     'Labels' 'Machines' 'Supplies']
```

FIGURE 7: .UNIQUE TO IDENTIFY ALL THE UNIQUE VALUES IN A COLUMN

Be careful that "Sub-Category" is spelled correctly in your code...you'll get the `KeyError` below if it's not (you can see that I misspelled Sub-Category as Sub-Categroy in Figure 8, below). This error happens because Pandas is looking for an exact column name in the dataframe and cannot find Sub-Categroy in the list of columns.



```
SubCats = SalesData["Sub-Categroy"].unique()
print(SubCats)
```

```
KeyError                                Traceback (most recent call last)
/usr/local/lib/python3.6/dist-packages/pandas/core/indexes/base.py in get_loc(self, key, method, tolerance)
    2645         try:
-> 2646             return self._engine.get_loc(key)
```

FIGURE 8: KEY ERROR FROM MISSPELLED COLUMN NAME

Also, Python is case sensitive to subcats is not the same as SubCats. The SubCats array name must match in the `unique()` and `print` statements below.

5. **MODIFYING THE .UNIQUE()** CODE TO SEE VALUES IN OTHER COLUMNS. You can also modify the correct code to see the unique values in other columns like I did below in Figure 9 for "Category" and for "Segment" which are customer types.

```
[4] Categories = SalesData["Category"].unique()
    print(Categories)

['Furniture' 'Technology' 'Office Supplies']

Segment = SalesData["Segment"].unique()
print(Segment)

['Corporate' 'Consumer' 'Home Office']
```

FIGURE 9: .UNIQUE LOOKING AT OTHER COLUMNS

## PART 4: GROUP, SUM, & SORT

In this section we'll first determine those sub-categories that aren't helping Office Solutions meet its financial goals.

1. **WEEDING OUT UNNEEDED DATA:** The first thing we need to do to understand Sub-Category Profits is to tell Pandas to only look at certain columns in our dataframe and to disregard the others. Much of the work in analyzing data with Pandas is getting rid of data we don't need and that could complicate our analyses. In the code in Figure 10 below, I am naming a reference to three columns in my SalesData dataframe, Sub-Category, Profit, and Sales. It looks like we are creating new dataframe called SubCatProfits with only three columns. That's not what's really happening...instead we are telling Pandas just to look at these columns when we write SubCatProfits. If you are new to Python and Pandas, this can be confusing so don't worry. We are going to treat our new SubCatProfits like a new dataframe with just three columns in it in our next step. Try adding the code below. The results below is just a partial image of the output. You might notice that the Accessories Sub-Category appears twice. That's because we haven't done any grouping and these are just the first 10 rows of sales data in my dataframe. **NOTE: Be sure there are two sets of square brackets around Sub-Category, Profit, and Sales!**

```
SubCatProfits = SalesData[["Sub-Category", "Profit", "Sales"]]
print(SubCatProfits)
```

	Sub-Category	Profit	Sales
0	Bookcases	38.2668	173.940
1	Phones	67.2742	231.980
2	Storage	28.6150	114.460
3	Accessories	-5.2640	30.080
4	Accessories	-6.2100	165.600

FIGURE 10: PICKING ONLY THE COLUMNS WE WANT TO WORK WITH

2. **.GROUPBY, .SUM(), AND .SORT\_VALUES():** What we really want to see is the total of all Profits for each Sub-Category so we want Pandas to group records by Sub-Category and total up the Profits and Sales. That's just what the next code does in Figure 11. I'm also going to sort the resulting values by Profit so I can quickly see lowest to highest. With sort\_values, sorting values in ascending order is the default so strings will appear in alphabetical order, our Profits, below, are smallest to largest. Also, groupby will automatically sort by your first column so if you just want Sub-Categories sorted in alphabetical order, you don't need sort\_values.

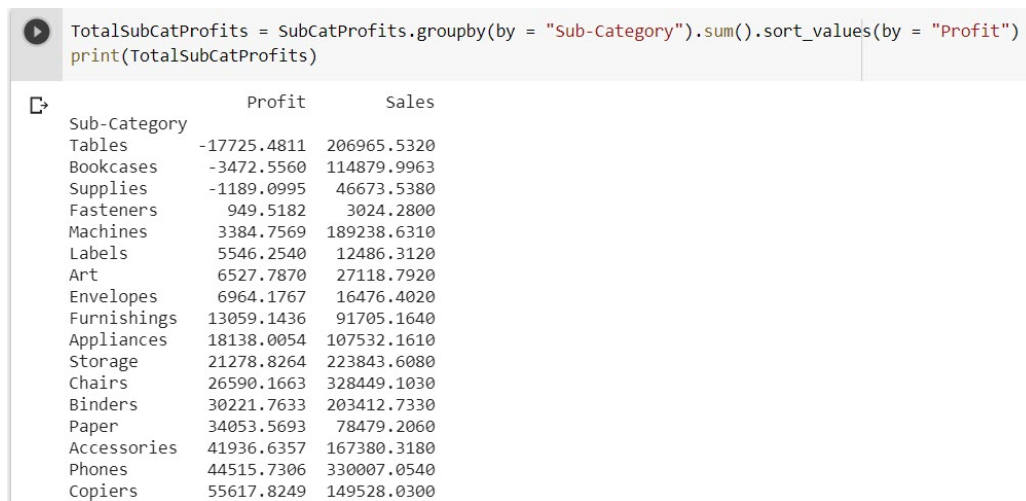


FIGURE 11: GROUPBY, SUM, AND SORT\_VALUES

So what's happening in the code above? First, I ask Pandas to look through all 9994 rows of Sub-Category data, group them together and total up their Profits and Sales, then assign a reference to these with a new dataframe name TotalSubCatProfits. Of course, I still need a print or a display statement to see my results. As you can see above, Tables, Bookcases, and Supplies are all negative! These aren't just underperforming; they are hurting the Office Solution's total profits. Now your instinct may be to say, "Let's get rid of these sub-categories!" but we don't know much about them. What if these sub-categories are much better performing in some geographical Regions than others or maybe there are some highly profitable products in these sub-categories but they are overshadowed by highly negative products when we total them up. Let's take a look in there to see what might be going on.

3. Let's look at another example of grouping. Maybe instead of looking at the Sub-Category level I want to look at Category Profits and Sales. In Figure 12 you'll see this code plus how you can condense the code above into a single line. I also use the display here instead of print but when you are looking at small amounts of output the difference between the two isn't so great. Note you can add another argument to sort\_values to reverse the sort order: .sort\_values(by= "Profit", ascending = False)



FIGURE 12: CATEGORY PROFIT AND SALES

4. Multi-level grouping. We can also group by more than one value. In Figure 13 you can see how I grouped first by Category and then by Sub-Category. You need to use the [] as you are passing a list to the groupby function, not just the single value it is expecting. Also, you can see that I split the column selection and grouping into two lines...this is mainly for legibility. I also dropped the sort\_values and let the group by order the output based on category. I chose print instead of display too. Display is prettier but print is more compact. Be sure that if you are modifying the code in Figure 12 to make the code in Figure 13 than you are printing the correct dataframe, printing the second dataframe in Figure 12 TotalCatSubCat, not the first, CategoryProfitsSales. I make this mistake all the time! You can group on as many columns as you like but it does have its limits – groupby needs repeated values to work. If you try grouping on a column with many unique values, like Profit, there isn't much this function will be able to do for you.



```

CategoryProfitsSales = SalesData[["Category", "Sub-Category", "Profit", "Sales" ]]
TotalCatSubCat = CategoryProfitsSales.groupby(by=["Category", "Sub-Category"]).sum()
print(TotalCatSubCat)

```

Category	Sub-Category	Profit	Sales
Furniture	Bookcases	-3472.5560	114879.9963
	Chairs	26590.1663	328449.1030
	Furnishings	13059.1436	91705.1640
	Tables	-17725.4811	206965.5320
Office Supplies	Appliances	18138.0054	107532.1610
	Art	6527.7870	27118.7920
	Binders	30221.7633	203412.7330
	Envelopes	6964.1767	16476.4020
	Fasteners	949.5182	3024.2800
	Labels	5546.2540	12486.3120
	Paper	34053.5693	78479.2060
	Storage	21278.8264	223843.6080
	Supplies	-1189.0995	46673.5380
	Accessories	41936.6357	167380.3180
Technology	Copiers	55617.8249	149528.0300
	Machines	3384.7569	189238.6310
	Phones	44515.7306	330007.0540

FIGURE 13: MULTILEVEL GROUPING CATEGORY AND SUB-CATEGORY

## PART 5. USING THE .LOC FUNCTION TO FIND ROWS CONTAINING SPECIFIC VALUES PLUS A FILTER EXAMPLE

- To dig into the data analyses I'm going to focus on just one of the negative sub-categories we found previously: Tables. I'll show you how to modify the code to see Bookcases in a bit. The first thing I'm going to do is to ask Pandas to find me all the rows in SalesData, my original dataframe, where the value in the Sub-Category column is equal to Tables. For this we are going to use the .loc method in Figure 14, below. We will use the dataframe JustTables created below in several exercises so you'll want to reproduce it carefully.

```

JustTables = SalesData.loc[SalesData["Sub-Category"]=="Tables"]
print(JustTables.head(10))

```

	Row ID	Order ID	Order Date	...	Quantity	Discount	Profit
6	9494	CA-2016-105207	2016-01-03	...	7	0.0	350.4270
37	3236	CA-2016-140746	2016-01-15	...	1	0.3	-15.5826
69	9359	CA-2016-168046	2016-01-25	...	3	0.4	-99.3453
82	8927	CA-2016-168032	2016-01-30	...	3	0.5	-538.4460
113	8144	CA-2016-100993	2016-02-05	...	4	0.2	6.9716
130	7813	CA-2016-134334	2016-02-14	...	3	0.3	-47.1798
172	1083	US-2016-143819	2016-03-01	...	8	0.4	-264.9208
183	4003	CA-2016-145730	2016-03-03	...	3	0.3	-127.5792
204	8985	CA-2016-110898	2016-03-06	...	2	0.5	-99.2664
233	1462	US-2016-128902	2016-03-11	...	2	0.3	-31.3722

[10 rows x 21 columns]

FIGURE 14: LOCATING JUST THOSE ROWS WITH TABLES IN THE SUB-CAT COLUMN

What we've done in this code is to start over with our original full set of data in the SalesData dataframe, then **located only those rows with sales data about Tables**. I'm using the .head() to limit my output to print only the first 10 rows. That's because it without doing this, the results will be 9994 rows and our code cell won't print all that...it's just too much data. However, 21 columns are there noted at the bottom of the output. Be sure you add both == in the first line about. This is how Python makes a comparison, like A equals B. A single = is an assignment statement were we assign values like we did for JustTables. If you want to look at Bookcases instead of Tables the code would like that in Figure 15. You can modify this code for any Sub-Category you want to look at.

```
JustBookcases = SalesData.loc[SalesData["Sub-Category"]=="Bookcases"]
print(JustTables.head(10))
```

FIGURE 15: LOCATING JUST BOOKCASES

2. **TOTAL PROFITS AND SALES FOR EACH PRODUCT IN THE TABLE SUB-CATEGORY.** Now I need a new dataframe that only looks at my three columns of interest, Product Name, Sales, and Profit. You can see below that we are doing something similar to before, only we have 3 columns we are working with instead of two and we are grouping by Product Name rather than Sub-Category. You may be wondering why we didn't include Sub-Category as a column. Well do we really need it? We already weeded out all the Sub-Categories that weren't Tables so we could have a column that listed table in each row but it wouldn't add much to our output. Also, you'll see that I didn't specify which columns I wanted .sum() to add up. We don't need to, .sum() will total up any and all numerical data for us – even if it doesn't make much sense, like postal code, which is why we want to remove any columns we don't need as we play with the data. I've combined the column selection and groupby / sum into one line in Figure 16 but you can do it in two steps if you prefer.

```
TableProdProfSales = JustTables[["Product Name", "Profit", "Sales"]].groupby("Product Name").sum().sort_values("Profit")
display(TableProdProfSales)
```

	Profit	Sales
Product Name		
Chromcraft Bull-Nose Wood Oval Conference Tables & Bases	-2876.1156	9917.6400
Bush Advantage Collection Racetrack Conference Table	-1934.3976	9544.7250
Balt Solid Wood Round Tables	-1201.0581	6518.7540
BoxOffice By Design Rectangular and Half-Moon Meeting Room Tables	-1148.4375	1706.2500
Riverside Furniture Oval Coffee Table, Oval End Table, End Table with Drawer	-1147.4000	4446.1750
Bretford "Just In Time" Height-Adjustable Multi-Task Work Tables	-964.1940	5634.9000
Bevis Oval Conference Table, Walnut	-856.0144	6942.0680
BPI Conference Tables	-795.9725	2241.8675

FIGURE 16: A FEW OF THE NEGATIVE PROFIT TABLE PRODUCTS

Figure 16 above doesn't show all the output from this code but in your code cell you'll see that many of the table products are negative and those that aren't really aren't making the company much money. Even the best performing table product at the end of our list only made Office Solutions about \$600 in 4 years!

What if you wanted to find only those rows that contain two or more specific values, for example Table sales to Corporate customers in just one line of code? For this we can build a statement containing multiple criteria for or .loc. Here's an example of what that would look like, note how we use the & to tell the .loc that both stated conditions needs to be met for the row to be selected:

```
SubCatSalesProf = SalesData[["Segment", "Sub-Category", "Sales", "Profit"]]
CorpTableSalesProf = SubCatSalesProf.loc[(SubCatSalesProf["Segment"]=="Corporate") & (SubCatSalesProf["Sub-Category"]=="Tables")]
TotalCorpTablesSalesProf = CorpTableSalesProf.groupby(by=["Segment", "Sub-Category"]).sum().round()
display(TotalCorpTablesSalesProf)
```

		Sales	Profit
Segment	Sub-Category		
Corporate	Tables	70872.0	-4906.0

FIGURE 17: MULTI-CRITERIA LOC

3. **FINDING ONLY NEGATIVE TABLE PRODUCTS.** If we are only interested in finding negative profit table products, we can use a filter as shown in Figure 18. Note that I group and sum before filtering so I'm only finding those tables products with total negative profits over 4 years, not each instance where a table sold for a negative profit. I only provide a portion of the output in Figure 18.



```

JustTables = SalesData.loc[SalesData["Sub-Category"]=="Tables"]
TableProf = JustTables[["Product Name", "Profit"]].groupby("Product Name").sum()
NegTables = TableProf[TableProf["Profit"] < 0.0]
display(NegTables)

```

Bevis Oval Conference Table, Walnut	-856.0144
Bevis Rectangular Conference Tables	-586.8396
Bevis Round Bullnose 29" High Table Top	-192.1854
Bevis Round Conference Room Tables and Bases	-39.4438
Bevis Round Conference Table Top, X-Base	-519.9410

FIGURE 18: NEGATIVE TABLES ONLY

## PART 6. DIGGING DEEPER TO UNDERSTAND TABLE PROFITS AND SALES

1. At this point you may be thinking, "Tables are a really underperforming Sub-Category. Let's discontinue Tables! While it is true that OS is losing money on almost all of its Table products, we need to look further into the data to understand what's going on and why so many Tables are unprofitable. There can also be repercussions to discontinuing whole Sub-Categories. For example, what if our best customers by lots of tables from us? Do we want to lose their business if they make us money on other products? Is there a difference in table profitability depending on whom we sell them to or where we sell them? We don't have a full picture yet so before we advise Natasha, we should dig deeper into the data. Let's look at two columns in particular: Segment (i.e. customer segment) and Region. Before we do that though, keep in mind that we are still working with the JustTables dataframe we created earlier. If you stopped working through this tutorial and are now in a new Colab project then you'll need to upload your data and import it and then run this code again:

```
JustTables = SalesData.loc[SalesData["Sub-Category"]=="Tables"]
```

Below in Figure 19 is the code for Segment and Regional analyses of the Table Sub-Category's Profit and Sales. You may be wondering, why look at regional effects or consumer Segment effects? Before we make a recommendation, we want to understand everything we can about the data. What if we lose money on Tables in three regions but make a ton of money in the fourth...profits that were obscured at a higher level of analysis by other regions' losses? If so, that could change a recommendation from discontinue Tables to only marketing Tables in the one profitable region. Alternatively, what if sell a lot of Tables in one Segment and that Segment is one we really care about, like the Corporate Segment? This is something Natasha may want to consider before discontinuing Tables

```

[22] TableSegment = JustTables[["Segment", "Profit", "Sales"]]
      SegTotalProfSales = TableSegment.groupby(by="Segment").sum().sort_values(by = "Profit")
      print(SegTotalProfSales)

```

	Profit	Sales
Segment		
Consumer	-9728.0378	99933.7950
Corporate	-4906.4986	70871.7175
Home Office	-3090.9447	36160.0195

```

TableRegion = JustTables[["Region", "Profit", "Sales"]]
RegionTotalProfSales = TableRegion.groupby(by="Region").sum().sort_values(by = "Profit")
print(RegionTotalProfSales)

```

	Profit	Sales
Region		
East	-11025.3801	39139.807
South	-4623.0579	43916.192
Central	-3559.6504	39154.971
West	1482.6073	84754.562

FIGURE 19: TABLE PROFITS AND SALES BY CUSTOMER SEGMENT & REGION

Again, we are only working with the table data but what we can see from above is that we lose money on Tables in each Segment, though we lose the most to individual Consumers, and we lose money on Tables in all the Regions but the West.

**2. TOTAL TABLE PROFITS AND SALES FOR EACH CUSTOMER SEGMENT IN JUST THE WEST REGION.** We can combine these to investigate further to see which, if any, of our customer Segments makes a profit on Tables in the West Region. For that, we will use the .loc method again. It can get tricky though as we create a new dataframe reference on each line, building on the work we did on the previous line. Track your dataframes carefully to make sure you are using / printing the right one. Note we could have used a multi-criteria .loc statement like we did back in Figure 17 to find all the Tables in the Western Region. However, we've already been working with the JustTables dataframe for a while so now we just need to isolate those records in the West.



**FIGURE 20: BREAK DOWN OF SEGMENT PROFITS AND SALES IN THE WESTERN REGION**

Let's break down what's happening above. First, we are starting with the JustTables dataframe...it only has those rows that have Tables in the Sub-Category column but it has all the original SalesData columns. Next, we create a new dataframe reference TableSegRegion that only looks at 4 of the columns in our JustTables dataframe. In line two we use the .loc method again to find only those rows that have West in the Region column and to ignore the other rows. Once we have a dataframe, WestTableSegment, that has only Table Sales in the West Region, we grouped by Segment and totaled up our Sales and Profits. As you can see from the output, we do make some money on Tables in the Consumer and Corporate Segments in the West Region, not a lot but it's important for us to give Natasha as a complete a picture of table performance as we can. Maybe the evidence we have is enough for her to want to discontinue Tables...or drastically reduce the Table products OS offers. But we still don't know why we lose so much on Tables. Looking at our columns of data, we don't have cost of goods sold but we do have data on discounts. Maybe there's a lead there.

**3. .MEAN TO SEE AVERAGE DISCOUNTS ON TABLES.** Below, in Figure 21, is code that will help us see what the average overall Discount is on Tables and what the Discount is on some of our table products. The .mean() is similar to the .sum() we used before and only works on numbers.



**FIGURE 21: CALCULATED THE AVERAGE (MEAN) DISCOUNT ON TABLES**

Below is a part of the output from this code. At the top, we see that the average Discount on Tables is high, 26%, even while the average Profit is negative. Next, we see the most discounted Tables some of which are discounted almost 50%! Also, notice how the Profit on all these highly discounted table products are negative! Looks like we might have found at least one issue with our poorly performing Tables Sub-Category!

Discount	0.261285	
Profit	-55.565771	
dtype:	float64	
	Discount	Profit
Product Name		
BoxOffice By Design Rectangular and Half-Moon M...	0.483333	-382.812500
Bevis Rectangular Conference Tables	0.475000	-146.709900
Balt Split Level Computer Training Table	0.380000	-71.595000
BPI Conference Tables	0.370000	-159.194500
Riverside Furniture Stanwyck Manor Table Series	0.366667	-138.644167
Bush Advantage Collection Racetrack Conference ...	0.350000	-276.342514
Bevis Traditional Conference Table Top, Plinth ...	0.333333	-152.234400
Chromcraft Bull-Nose Wood 48" x 96" Rectangular...	0.333333	-101.931300
Hon 30" x 60" Table with Locking Drawer	0.333333	-59.117067
KI Conference Tables	0.325000	-59.902050
Hon Rectangular Conference Tables	0.325000	-73.953750
Chromcraft Rectangular Conference Tables	0.322222	-50.027000

FIGURE 22: OUTPUT OF MEAN TABLE DISCOUNT

## PART 7. CORRELATION BETWEEN TABLE DISCOUNTS AND SALES & DATES (.DT)

1. Companies typically give discounts to drive up sales or to clearance out products. Sometimes, companies will even lose money to attract customers. An example of this are loss leaders...products discounted deeply to get you into the store to spend your money on other products – think Door Busters on Black Friday. However, OS is losing money on most of their Table products so this probably isn't what's happening. Let's see if there's any relationship (correlation) between Discounts and Table Sales. If the discounts are driving sales as we hope, there should be a moderate to strong positive correlation between the two. In other words, when discounts increase, sales also increase. If sales decrease and discounts increase, that would be a negative correlation and certainly not what we want as discounts can drive down profits. Below, in Figure 23, is the code to determine this correlation. Note I had to import a new library, `scipy.stats`.

```
import scipy.stats
x = "Discount"
y = "Sales"
GetCorrelation = scipy.stats.spearmanr(JustTables[x], JustTables[y])
Corr = GetCorrelation[0]
CorrRounded = str(Corr.round(2))
pValue = GetCorrelation[1]
pValueRounded = str(pValue.round(5))
print("The correlation between " + x + " and " + y + " is: " + CorrRounded )
print("At p value: " + pValueRounded)
```

The correlation between Discount and Sales is: -0.29  
At p value: 0.0

FIGURE 23: CORRELATION BETWEEN DISCOUNTS & SALES

So what's happening in the code? First, we need an x and a y to correlate. Yes, we could write these into our `spearmanr` code but by assigning them to variables, we can change them easily if we want. We also just want to look at the correlation between sales and discounts for Tables only so I'm using my `JustTables` dataframe. `GetCorrelation` is a numpy array that holds the results of this calculation as two values, the correlation in the first slot and the p value in the second (pandas is built on top of numpy so we don't have to import that library). I've rounded the values for readability. For correlation, two decimal places is enough, for p-values, they can get quite small so I rounded to five decimal places. After that, we just need to convert to strings to print plus an explanation of what we are seeing. Our results tell us there is a moderate negative correlation between discounts and sales. The p values is significant but given how many records we have that's probably not meaningful – when you have lots of data even small, un-meaningful relationships can appear significant. However, what is important to know is that OS's current approach to discounts on Tables is backfiring and hurting, not helping, sales.

2. **.DT (DATE TIME) AND CREATING A NEW COLUMN TO SEE ANNUAL SALES OF TABLES.** For the last analysis we'll do together, let's see if there's a year-over-year trend in table Sales. This is a bit more complicated so we'll do it in two steps.

STEP 1: We have a column called Order Date – the date when the order was place. Before we can group by year, we first need to get the Order Date's year all by itself. Originally, it was in a day, month, and year format. So we create a new column in our JustTablesYears dataframe called Year then use the .dt method (date time) to pull the year out of each Order Date and put it in the Year column. You can replace .dt.year with .dt.month or .dt.quarter etc. to see other periods. Using .dt.quarter will break the year into 4 quarters that align with the seasons (Q1 = Jan., Feb., March = winter) if you are interested in looking at seasonal effects. You can see our new year column in our list of columns, in Figure 24 below.

```
[53] #make a copy to preserve previous dataframe
JustTablesYears = JustTables.copy()
#create a new column called year and fill it with the year portion of each row's order date
JustTablesYears["Year"] = JustTablesYears["Order Date"].dt.year
#check to make sure our new year column is there (at the end)
print(JustTablesYears.columns)
```

```
Index(['Row ID', 'Order ID', 'Order Date', 'Ship Date', 'Ship Mode',
      'Customer ID', 'Customer Name', 'Segment', 'Country', 'City', 'State',
      'Postal Code', 'Region', 'Product ID', 'Category', 'Sub-Category',
      'Product Name', 'Sales', 'Quantity', 'Discount', 'Profit', 'Year'],
      dtype='object')
```

FIGURE 24: ADDING A NEW SERIES (COLUMN) TO HOLD THE YEAR PORTION OF ORDER DATE

STEP 2: Now we can use the same groupby and sum methods we've used previously. From the output below we can see Table Sales are negative each year and there doesn't seem to be a trend over the 4 years in terms of Profits but it looks like total Sales of Tables is on the decline (Figure 25).

```
#now that we have the year isolated, we can group by year and total up profit and sales
YearlyProfSales = JustTablesYears[["Year", "Profit", "Sales"]].groupby(by="Year").sum()
print("The yearly profit and sales for tables are:")
print(YearlyProfSales)
```

```
The yearly profit and sales for tables are:
      Profit      Sales
Year
2016 -2950.9418  60833.2005
2017 -8140.6947  60893.5425
2018 -3124.0427  46088.3655
2019 -3509.8019  39150.4235
```

FIGURE 25: GROUPING & SUMMING ANNUAL TABLE SALES

## PART 8: COMBINING SEVERAL LEVELS OF ANALYSIS AT ONCE

We've looked at the Tables data in quite a few different ways and could probably recommended the company discontinue Tables sales, or perhaps only sell them to Corporate accounts. But let's look at just a couple more analyses. The embedded green comments explain each line.

### 1. AVERAGE TABLE DISCOUNTS BY REGION.

```
JustTables = SalesData.loc[SalesData["Sub-Category"]=="Tables"]
#Grouping by region and averaging discount
RegJustTableDiscount = JustTables[["Region", "Discount"]].groupby(by = "Region").mean()
print(RegJustTableDiscount)
```

```
Region      Discount
Central    0.262500
East       0.373750
South      0.222549
West       0.200000
```

FIGURE 26: AVERAGE TABLE DISCOUNTS BY REGION

## 2. AVERAGE TABLE DISCOUNTS BY REGION FOR EACH YEAR (ADDING A LOOP).

```
#Code to look at Table regional discounts by year
#create a dataframe reference to only those rows where Sub-Cat value is Tables
JustTables = SalesData.loc[SalesData["Sub-Category"]=="Tables"]
#Making a copy of JustTable
JustTablesYear = JustTables.copy()
#adding the Year column and using .dt to fill in the yr portion of Order Date
JustTablesYear["Year"] = JustTablesYear["Order Date"].dt.year
#Just checking column with a print
#print(JustTablesYear.columns)
Years = JustTablesYear.Year.unique()
#can check array of years with a print too (just uncomment)
#print(Years)
# just keeping what we need
RegJustTableYear = JustTablesYear[["Region", "Year", "Discount"]]
#looping through the array of years
for year in Years:
    # locating one year's worth of data at a time
    OneYear = RegJustTableYear.loc[RegJustTableYear["Year"] == year]
    #getting rid of the year column - not needed after .loc and .mean will try to average
    NoYear = OneYear[["Region", "Discount"]]
    #group by Region, sum up sales and profits
    YearlyRegDiscount = NoYear.groupby(by="Region").mean()
    # print an output title
    print("\nRegional table discounts for the year: " + str(year))
    #results
    print(YearlyRegDiscount)
    print("*"*40)
```

FIGURE 27: LOOPING THROUGH YEARS TO SEE ANNUAL REGIONAL DISCOUNTS ON TABLES

```
Regional table discounts for the year: 2016
Discount
Region
Central  0.205882
East     0.368182
South    0.250000
West     0.240625
*****

Regional table discounts for the year: 2017
Discount
Region
Central  0.292308
East     0.373913
South    0.270588
West     0.178947
*****

Regional table discounts for the year: 2018
Discount
Region
Central  0.326667
East     0.380000
South    0.113636
West     0.208000
*****

Regional table discounts for the year: 2019
Discount
Region
Central  0.207143
East     0.373333
South    0.218750
West     0.166667
*****
```

FIGURE 28: OUTPUT OF AVERAGE TABLE DISCOUNTS BY REGION FOR EACH YEAR



3. AVERAGE TABLE DISCOUNTS BY CUSTOMER SEGMENT FOR EACH YEAR. You'll notice the code is a combination of what we've done before. We are still looping through the years but we've swapped the Region column for Segment.

```
[26] #Code to look at Table customer segment discounts by year
#create a dataframe reference to only those rows where Sub-Cat value is Tables
JustTables = SalesData.loc[SalesData["Sub-Category"]=="Tables"]
#Making a copy of JustTable
JustTablesYear = JustTables.copy()
#adding the Year column and using .dt to fill in the yr portion of Order Date
JustTablesYear["Year"] = JustTablesYear["Order Date"].dt.year
#Just checking column with a print
#print(JustTablesYear.columns)
Years = JustTablesYear.Year.unique()
#can check array of years with a print too (just uncomment)
#print(Years)
# just keeping what we need
SegJustTableYear = JustTablesYear[["Segment", "Year", "Discount"]]
#looping through the array of years
for year in Years:
    # locating one year's worth of data at a time
    OneYear = SegJustTableYear.loc[SegJustTableYear["Year"] == year]
    #getting rid of the year column - not needed after .loc and .mean will try to average
    NoYear = OneYear[["Segment", "Discount"]]
    #group by Region, sum up sales and profits
    YearlySegDiscount = NoYear.groupby(by="Segment").mean()
    # print an output title
    print("\nTable discounts by customer segment for the year: " + str(year))
    #results
    print(YearlySegDiscount)
    print("*"*40)
```

FIGURE 29: ANNUAL SEGMENT TABLE DISCOUNTS CODE

```
-
Table discounts by customer segment for the year: 2016
      Segment      Discount
Consumer    0.263095
Corporate   0.265714
Home Office 0.300000
*****

Table discounts by customer segment for the year: 2017
      Segment      Discount
Consumer    0.254444
Corporate   0.279032
Home Office 0.267857
*****

Table discounts by customer segment for the year: 2018
      Segment      Discount
Consumer    0.293243
Corporate   0.261538
Home Office 0.162500
*****

Table discounts by customer segment for the year: 2019
      Segment      Discount
Consumer    0.255714
Corporate   0.200000
Home Office 0.222222
*****
```

FIGURE 30: ANNUAL SEGMENT TABLE DISCOUNTS OUTPUT

4. TOTAL TABLE PROFITS AND SALES FOR EACH CUSTOMER SEGMENT BY REGION LOOP. We could also look at all Segments in all Regions at the same time but that requires a loop too.

```
#create a numpy array of unique regions
Regions = JustTables.Region.unique()
TableSegRegion = JustTables[["Segment", "Region", "Profit", "Sales"]]
#loop through the regions array.
for region in Regions:
    #with each loop the variable region takes on a different region value
    #the .loc then finds rows with that specific region, ignoring the rest for the remainder of the loop.
    TableSegment = TableSegRegion.loc[TableSegRegion["Region"]==region]
    TableSegmentProfSales = TableSegment.groupby(by="Segment").sum().sort_values(by = "Profit")
    #the next line is just an output lable to improve legibility
    print("\nThe table sub-category profits and sales for each customer segment in the " + region + " are as follows:")
    #data output, different with each loop
    print(TableSegmentProfSales)
    #visual formatting break
    print(""*40)
```

FIGURE 31: SEGMENT TABLE PROFITS & SALES BY REGION

```
The table sub-category profits and sales for each customer segment in the Central are as follows:
Profit      Sales
Segment
Consumer    -3964.1571  20835.000
Home Office -302.6739   5757.378
Corporate    707.1806   12562.593
*****

The table sub-category profits and sales for each customer segment in the East are as follows:
Profit      Sales
Segment
Corporate   -5299.6557  17551.558
Consumer    -3708.3528  14082.467
Home Office -2017.3716   7505.782
*****

The table sub-category profits and sales for each customer segment in the West are as follows:
Profit      Sales
Segment
Home Office -113.5648  16009.185
Consumer    521.9717   40607.486
Corporate   1074.2004   28137.891
*****

The table sub-category profits and sales for each customer segment in the South are as follows:
Profit      Sales
Segment
Consumer    -2577.4996  24408.8420
Corporate   -1388.2239  12619.6755
Home Office -657.3344   6887.6745
*****
```

FIGURE 32: SEGMENT TABLE PROFITS & SALES BY REGION OUTPUT

5. AVERAGE ANNUAL TABLE DISCOUNTS FOR EACH SEGMENT BY REGION. This requires a loop within a loop, in Figure 33. In our first loop, we isolate data for 1 year. Once we have that we loop through all of the four regions for that year in the inner loop before hopping back to the outer loop to grab the data from our second year. A partial output follows in Figure 34.

```

#Code to look at Table customer segment discounts by year and by region
JustTables = SalesData.loc[SalesData["Sub-Category"]=="Tables"]
JustTablesYear = JustTables.copy()
JustTablesYear["Year"] = JustTablesYear["Order Date"].dt.year
Years = JustTablesYear.Year.unique()
Regions = JustTablesYear.Region.unique()
SegJustTableYear = JustTablesYear[["Segment", "Year", "Discount", "Region"]]
#looping through the array of years
for year in Years:
    OneYear = SegJustTableYear.loc[SegJustTableYear["Year"] == year]
    NoYear = OneYear[["Segment", "Discount", "Region"]]
    #looping through the array of regions
    for region in Regions:
        OneRegion = NoYear.loc[NoYear["Region"] == region]
        YearlySegRegDiscount = OneRegion.groupby(by="Segment").mean()
        print("\n" + str(year) + " Table discounts by customer segment in the " + region + " region")
        print(YearlySegRegDiscount)
        print(""*40)

```

FIGURE 33: AVERAGE ANNUAL TABLE DISCOUNTS FOR EACH SEGMENT BY REGION CODE

```

2016 Table discounts by customer segment in the Central region
Discount
Segment
Consumer      0.32
Corporate      0.06
Home Office    0.00
*****

2016 Table discounts by customer segment in the East region
Discount
Segment
Consumer      0.336364
Corporate      0.400000
Home Office    0.400000
*****

2016 Table discounts by customer segment in the West region
Discount
Segment
Consumer      0.208333
Corporate      0.220000
Home Office    0.380000
*****

2016 Table discounts by customer segment in the South region
Discount
Segment
Consumer      0.183333
Corporate      0.350000
*****

2017 Table discounts by customer segment in the Central region
Discount
Segment
Consumer      0.293333
Corporate      0.320000
Home Office    0.266667
*****

2017 Table discounts by customer segment in the East region
Discount
Segment
Consumer      0.371429
Corporate      0.362500
Home Office    0.387500

```

FIGURE 34: AVERAGE ANNUAL TABLE DISCOUNTS FOR EACH SEGMENT BY REGION OUTPUT

## 6. COMBINING ANALYSES WITH .GROUPBY

If you want to do both .sum and .mean in the same output, things can get a bit trickier. In the figure below I use .agg to allow me to sum Profit while also take the mean of Discounts. The first 5 records of the output are also included.

```
TablesProduct = JustTables[["Product Name", "Profit", "Discount"]]
TablesTotalProfAveDiscount = TablesProduct.groupby("Product Name").agg({"Profit": 'sum', "Discount": "mean"})
display(TablesTotalProfAveDiscount)
```

Bush Advantage Collection Racetrack Conference Table	-1934.3976	0.350000
Bush Advantage Collection Round Conference Table	210.4740	0.237500
Bush Andora Conference Table, Maple/Graphite Gray Finish	-143.6232	0.300000
Bush Cubix Conference Tables, Fully Assembled	311.8230	0.200000
Chromcraft 48" x 96" Racetrack Double Pedestal Table	-404.0064	0.316667

FIGURE 35: .AGG TO COMBINE SUM AND MEAN

## PART 9: BASIC GRAPHS

In this section, I will show you how to create two basic graphs. Be sure to think about the story you are trying to tell when selecting a graph, chart, or table...they each have their strengths and limitations.

1. Importing the needed libraries. Here are two very helpful libraries for graphs and charts, matplotlib and seaborn. Matplotlib is a very powerful library that will let you control many aspects of your graphs. Seaborn layers on top of matplotlib and reduces the amount of code you need to write to create a graph. The first thing we need to do is import both of these as shown in Figure 36. Make sure you run this code cell after you add your import statements.

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
xl=pd.ExcelFile("/content/TableauSalesData.xlsx")
SalesData = xl.parse("Orders")
```

FIGURE 36: IMPORTING MATPLOTLIB & SEABORN

2. I'm going to use part of the yearly profit and sales code we created previously to create a pie chart of annual table sales. If you haven't used your Colab file in a while, you will need to reload your data file, run the first code cell wherein we created the

```
JustTables = SalesData.loc[SalesData["Sub-Category"]=="Tables"]
JustTablesYears = JustTables.copy()
JustTablesYears["Year"] = JustTablesYears["Order Date"].dt.year
```

FIGURE 37: CREATING THE JUSTTABLESYEARS DATAFRAME

SalesData dataframe, as well as the cells that create the JustTables dataframe and the JustTableYears dataframes. Once that's done, we need can determine the total Tables sales by year, using groupby and sum as we've done several times. In order to graph this data though we will need to

reset our dataframe's index using .reset\_index(). Why? Because of the way groupby works. Look at the output in Figure 38 below. In the first output you'll see what how the JustTableYearsSales dataframe looks. Do you see how the Year column label is offset from the Sales column label and how there are no index numbers? This is because the groupby uses the Year column values as indices. In other words, if you wanted to look at the first row of the dataframe you'd look at row 2016, not row 0.

```
JustTableYearsSales = JustTablesYears[["Year", "Sales"]].groupby(by="Year").sum()
print(JustTableYearsSales)
#reset_index() is an important step for graphs when we've used .groupby.
JustTableYearsSales = JustTableYearsSales.reset_index()
print("\n*****")
print(JustTableYearsSales)
```

```

      Sales
Year
2016  60833.2005
2017  60893.5425
2018  46088.3655
2019  39150.4235

*****
   Year      Sales
0  2016  60833.2005
1  2017  60893.5425
2  2018  46088.3655
3  2019  39150.4235
```

FIGURE 38: BEFORE & AFTER RESET INDEX

Now check out the second print that runs after we reset the index. Do you see the index column has returned and how Year is a proper column (i.e. a series) again? This is what our chart and graphs will need to work. I'll show you the error you'll get if you skip this step in a moment. First, let try a simple pie chart using matplotlib. Add the code in the figure below. Green comments provide explanations. You'll see it takes very few lines of code with matplotlib to create a very professional looking pie chart.

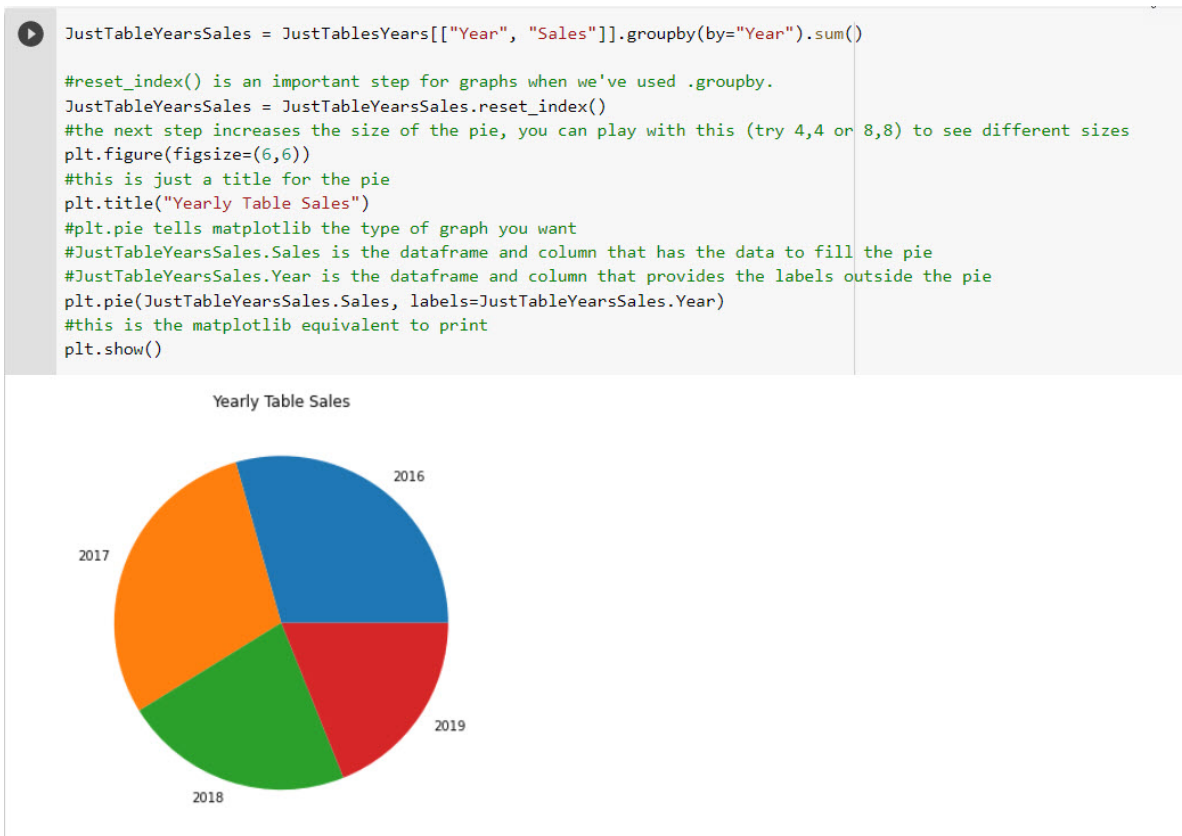


FIGURE 39: BASIC PIE CHART

Here's what happens when you don't reset the index, see Figure 40...matplotlib can't find the column Year (because it's an indexing column thanks to groupby) and throws an attribute error. If you don't reset\_index(), you'll get a value error:

```

JustTableYearsSales = JustTablesYears[["Year", "Sales"]].groupby(by="Year").sum()
plt.figure(figsize=(6,6))
plt.title("Yearly Table Sales")
plt.pie(JustTableYearsSales.Sales, labels=JustTableYearsSales.Year)
plt.show()

```

```

-----
AttributeError                                Traceback (most recent call last)
<ipython-input-15-7a43f1305352> in <module>()
      2 plt.figure(figsize=(6,6))
      3 plt.title("Yearly Table Sales")
----> 4 plt.pie(JustTableYearsSales.Sales, labels=JustTableYearsSales.Year)
      5 plt.show()

/usr/local/lib/python3.6/dist-packages/pandas/core/generic.py in __getattr__(self, name)
    5139         if self._info_axis._can_hold_identifiers_and_holds_name(name):
    5140             return self[name]
-> 5141         return object.__getattr__(self, name)
    5142
    5143     def __setattr__(self, name: str, value) -> None:

AttributeError: 'DataFrame' object has no attribute 'Year'

```

FIGURE 40: ATTRIBUTE ERROR FROM MATPLOTLIB FROM NOT USING RESET INDEX



3. So maybe you like the pie chart but think a bar chart would be more effecting. For our bar chart, let's use seaborn. I've included the groupby, sum and reset index code in the figure below for completeness but if you've already run that code in a different cell, perhaps in creating your pie chart, you need not repeat it.



FIGURE 41: BASIC BAR CHART WITH SEABORN

There are many other graphs available through these libraries and you can customize them by size, color, orientation, etc.

Now let's wrap everything up and make a data-justified business recommendation (see next page).

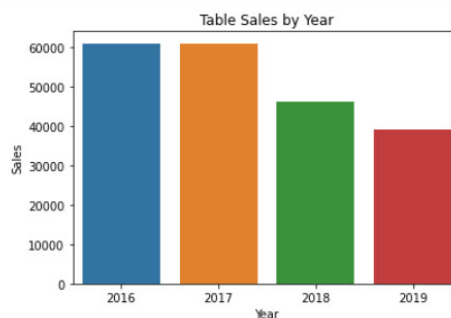
## PART 10: MAKING A RECOMMENDATION – ONE EXAMPLE YOU CAN FOLLOW

Natasha has stated a goal of increasing profits by 10% in the next year. One way to accomplish this is to minimize losses by reducing or discontinuing underperforming products. We present our analyses of Office Solutions' (OS) underperforming Sub-Category, Tables, below.

- First, we calculated the profits and sales for all sub-categories. The Tables Sub-Category has the highest negative profit of all sub-categories, with total losses over 4 years of \$17,725. Sales are strong however, totaling over \$207,000. The Bookcases and Supplies Sub-Categories are also unprofitable at (-\$3,473 and -\$1,189, respectively).
- We then investigated the profitability of individual table products and found most have negative profits. Only 14 table products are profitable and these barely so with the best performing table product earning only \$600 in profit in 4 years.
- We also examined how the Tables Sub-Category performs in different segments and regions:
  - Tables are unprofitable in all three customer segments, earning -\$9,728 in the Consumer Segment, -\$4,907 in the Corporate Segment, and -\$3,091 in the Home Office Segment.
  - The Tables Sub-Category is unprofitable in all sales regions (East -\$11,035, South -\$4,623, Central -\$3,560) but the West where it generated \$1,483 in profits.
  - In the West Region specifically, tables generated profits of \$522 in the Consumer Segment and \$1,074 in the Corporate Segment but lost \$114 in the Home Office Segment.
  - We also looked into segment effects in the other Regions finding that Corporate customers also generate \$700 in profits on tables in the Central Region. Table sales to the Home Office Segment are unprofitable in all regions. All Tables sales in the South and East Regions return negative profits.
- To try to understand why tables are unprofitable, we investigated discounts given in this sub-category. Discounts on tables are high, averaging 26% and reaching almost 50% on some highly unprofitable products. Table 1 provides the average discounts given on tables for each region. Further, discounts exhibit a moderate negative correlation with sales suggesting that the current discount strategy decreases sales rather than increasing them.
- After conducting a year-over-year analysis of both table profits and sales we find OS lost money on tables each year though no annual profit trend is apparent. Sales of tables have declined over the 4 years, as maybe seen in Figure 1, suggesting waning demand for OS table products.

**TABLE 1: REGIONAL TABLE DISCOUNT AVERAGES**

Region	Discount
Central	0.262500
East	0.373750
South	0.222549
West	0.200000



**FIGURE 1: TABLES SALES BY YEAR**

**Recommendation:** Based on the analyses above, we recommend OS reduce discounts offered on tables until they are profitable or, given the decline in sales, clearance table inventory and discontinue all products in this sub-category.