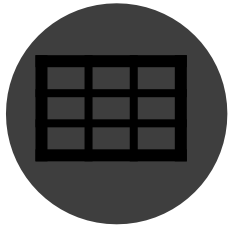
DATAFRAMES

DATAFRAMES



In this section we'll introduce Pandas **DataFrames**, the Python equivalent of an Excel or SQL table which we'll use to store and analyze data

TOPICS WE'LL COVER:

- DataFrame Basics
- Exploring DataFrames
- Accessing & Dropping Data
- Blank & Duplicate Values
- Sorting & Filtering
- Modifying Columns
- Pandas Data Types
- Memory Optimization

GOALS FOR THIS SECTION:

- Learn to read in DataFrames, explore their contents, and access data within them
- Manipulate DataFrames by filtering & sorting rows, handling missing data, and applying Pandas functions
- Understand the different data types in Pandas DataFrames, as well as how to optimize memory consumption by converting & downcasting them

THE PANDAS DATAFRAME

DataFrames are Pandas "tables" made up from columns and rows

- Each column of data in a DataFrame is a Pandas Series that shares the same row index
- The column headers work as a column index that contains the Series names

The row index points to the corresponding row in each Series

(axis = 0)

	id	date	store_nbr	family	sales	onpromotion
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0
1	1	2013-01-01	1	BABY CARE	0.0	0
2	2	2013-01-01	1	BEAUTY	0.0	0
3	3	2013-01-01	1	BEVERAGES	0.0	0
4	4	2013-01-01	1	BOOKS	0.0	0

The column index points to each individual Series

(axis = 1)

Each column is a Pandas Series

DATAFRAME PROPERTIES

DataFrames have these key properties:

- **shape** – the number of rows and columns in a DataFrame *(the index is not considered a column)*
- **index** – the row index in a DataFrame, represented as a range of integers *(axis=0)*
- **columns** – the column index in a DataFrame, represented by the Series names *(axis=1)*
- **axes** – the row and column indices in a DataFrame
- **dtypes** – the data type for each Series in a DataFrame *(they can be different!)*

```
df.shape
```

```
(3000888, 6)
```

```
df.index
```

```
RangeIndex(start=0, stop=3000888, step=1)
```

```
df.dtypes
```

```
id              int64
date           object
store_nbr       int64
family         object
sales         float64
onpromotion     int64
dtype: object
```

Pandas will try to guess the data types when creating a DataFrame

(we'll modify them later!)

```
df.columns
```

```
Index(['id', 'date', 'store_nbr', 'family', 'sales', 'onpromotion'], dtype='object')
```

```
df.axes
```

```
[RangeIndex(start=0, stop=3000888, step=1),
 Index(['id', 'date', 'store_nbr', 'family', 'sales', 'onpromotion'], dtype='object')]
```

CREATING A DATAFRAME

You can **create a DataFrame** from a Python dictionary or NumPy array by using the Pandas DataFrame() function

```
pd.DataFrame(
    {"id": [1, 2],
     "store_nbr": [1, 2],
     "family": ["POULTRY", "PRODUCE"]
    }
)
```

This creates a DataFrame from a Python dictionary

Note that the keys are used as column names

	id	store_nbr	family
0	1	1	POULTRY
1	2	2	PRODUCE

CREATING A DATAFRAME

You'll more likely **create a DataFrame** by reading in a flat file *(csv, txt, or tsv)* with Pandas read_csv() function

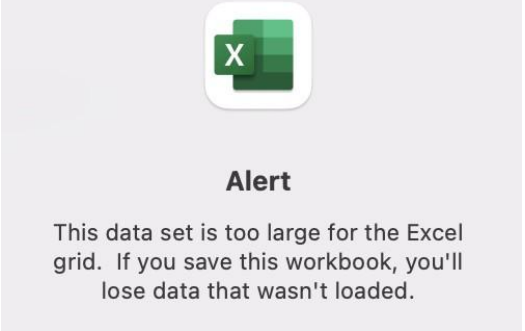
```
import pandas as pd

retail_df = pd.read_csv("retail/train.csv")

retail_df
```

While the read_csv() function has more arguments for manipulating the data during the import process, all you need is the file path and name to get started!

	id	date	store_nbr	family	sales	onpromotion
0	0	2013-01-01	1	AUTOMOTIVE	0.000	0
1	1	2013-01-01	1	BABY CARE	0.000	0
2	2	2013-01-01	1	BEAUTY	0.000	0
3	3	2013-01-01	1	BEVERAGES	0.000	0
4	4	2013-01-01	1	BOOKS	0.000	0
...	...	...	...	...	...	...
3000883	3000883	2017-08-15	9	POULTRY	438.133	0
3000884	3000884	2017-08-15	9	PREPARED FOODS	154.553	1
3000885	3000885	2017-08-15	9	PRODUCE	2419.729	148
3000886	3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.000	8
3000887	3000887	2017-08-15	9	SEAFOOD	16.000	0

3000888 rows × 6 columns

PRO TIP: Pandas is a great alternative to Excel when dealing with large datasets!



ASSIGNMENT: DATAFRAME BASICS

NEW MESSAGE

July 5, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Transactions Data**

Hi there,

I heard you did some great work for our finance department. I need some help analyzing our transactions to make sure there aren't any irregularities in the numbers. The data is stored in retail_2016_2017.csv.

Can you read this data in and remind me how many rows are in the data, which columns are in the data, and their datatypes? More to come soon!

Thanks!

section03_DataFrames.ipynb

Reply Forward

Results Preview

```
transactions
```

	date	store_nbr	transactions
0	2013-01-01	25	770
1	2013-01-02	1	2111
2	2013-01-02	2	2358
3	2013-01-02	3	3487
4	2013-01-02	4	1922

```
83487
```

```
date            object
store_nbr        int64
transactions     int64
dtype: object
```

SOLUTION: DATAFRAME BASICS

NEW MESSAGE

July 5, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Transactions Data**

Hi there,

I heard you did some great work for our finance department. I need some help analyzing our transactions to make sure there aren't any irregularities in the numbers. The data is stored in retail_2016_2017.csv.

Can you read this data in and remind me how many rows are in the data, which columns are in the data, and their datatypes? More to come soon!

Thanks!

section03_DataFrames.ipynb

Reply Forward

Solution Code

```
transactions = pd.read_csv('../retail/transactions.csv

transactions
```

	date	store_nbr	transactions
0	2013-01-01	25	770
1	2013-01-02	1	2111
2	2013-01-02	2	2358
3	2013-01-02	3	3487
4	2013-01-02	4	1922

```
transactions.index.max()
```

```
83487
```

```
transactions.dtypes
```

```
date            object
store_nbr        int64
transactions     int64
dtype: object
```

EXPLORING A DATAFRAME

You can **explore a DataFrame** with these Pandas methods:

head	*Returns the first n rows of the DataFrame (5 by default)*	**df.head(*nrows*)**
tail	*Returns the last n rows of the DataFrame (5 by default)*	**df.tail(*nrows*)**
sample	*Returns n rows from a random sample (1 by default)*	**df.sample(*n*)rows**
info	*Returns key details on a DataFrame's size , columns, and memory usage*	**df.info()**
describe	*Returns descriptive statistics for the columns in a DataFrame (only numeric columns by default; use the 'include' argument to specify more columns)*	**df.describe(*include*)**

HEAD & TAIL

The **.head()** and **.tail()** methods return the top or bottom rows in a DataFrame

- This is a great way to QA data upon import!

```
retail_df.head()
```

	id	date	store_nbr	family	sales	onpromotion
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0
1	1	2013-01-01	1	BABY CARE	0.0	0
2	2	2013-01-01	1	BEAUTY	0.0	0
3	3	2013-01-01	1	BEVERAGES	0.0	0
4	4	2013-01-01	1	BOOKS	0.0	0

*This returns the top **5** rows by default*

```
retail_df.tail(3)
```

	id	date	store_nbr	family	sales	onpromotion
3000885	3000885	2017-08-15	9	PRODUCE	2419.729	148
3000886	3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.0	8
3000887	3000887	2017-08-15	9	SEAFOOD	16.0	0

You can specify the number of rows to return, in this case the bottom 3

SAMPLE

The **.sample()** method returns a random sample of rows from a DataFrame

```
retail_df.sample()
```

	id	date	store_nbr	family	sales	onpromotion
1476628	1476628	2015-04-11	40	EGGS	77.0	0

*This returns **1** row by default*

```
retail_df.sample(5, random_state=12345)
```

	id	date	store_nbr	family	sales	onpromotion
2862475	2862475	2017-05-30	25	LIQUOR,WINE,BEER	92.0	2
940501	940501	2014-06-13	48	BABY CARE	0.0	0
1457967	1457967	2015-04-01	17	PLAYERS AND ELECTRONICS	0.0	0
1903307	1903307	2015-12-07	12	SEAFOOD	3.0	0
196280	196280	2013-04-21	16	PREPARED FOODS	66.0	0

You can specify the number of rows to return, in this case 5, and set a random_state to ensure your sample can be reproduced later in needed

INFO

The .info() method returns details on a DataFrame's properties and memory usage

```
retail_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3000888 entries, 0 to 3000887
Data columns (total 6 columns):
 #   Column       Dtype
---  ------       -----
 0   id           int64
 1   date         object
 2   store_nbr    int64
 3   family       object
 4   sales        float64
 5   onpromotion  int64
dtypes: float64(1), int64(3), object(2)
memory usage: 137.4+ MB
```

Rows & columns in the DataFrame

Position, name, and data type for each column

Memory usage

INFO

The .info() method returns details on a DataFrame's properties and memory usage

```
retail_df.info(show_counts=True)
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3000888 entries, 0 to 3000887
Data columns (total 6 columns):
 #   Column       Non-Null Count    Dtype
---  ------       --------------    -----
 0   id           3000888 non-null  int64
 1   date         3000888 non-null  object
 2   store_nbr    3000888 non-null  int64
 3   family       3000888 non-null  object
 4   sales        3000888 non-null  float64
 5   onpromotion  3000888 non-null  int64
dtypes: float64(1), int64(3), object(2)
memory usage: 137.4+ MB
```

*The .info() method will show non-null counts on a DataFrame with less than ~1.7m rows, but you can specify **show_counts=True** to ensure they are always displayed*

This is a great way to quickly identify missing values - if the non-null count is less than the total number of rows, then the difference is the number of NaN values in that column!

(In this case there are none)

DESCRIBE

The **.describe()** method returns key statistics on a DataFrame's columns

```
retail_df.describe()
```

	id	store_nbr	sales	onpromotion
count	3.000888e+06	3.000888e+06	3.000888e+06	3.000888e+06
mean	1.500444e+06	2.750000e+01	3.577757e+02	2.602770e+00
std	8.662819e+05	1.558579e+01	1.101998e+03	1.221888e+01
min	0.000000e+00	1.000000e+00	0.000000e+00	0.000000e+00
25%	7.502218e+05	1.400000e+01	0.000000e+00	0.000000e+00
50%	1.500444e+06	2.750000e+01	1.100000e+01	0.000000e+00
75%	2.250665e+06	4.100000e+01	1.958473e+02	0.000000e+00
max	3.000887e+06	5.400000e+01	1.247170e+05	7.410000e+02

Only numeric columns by default

Non-null values

Mean & standard deviation values

Quartile values

DESCRIBE

The **.describe()** method returns key statistics on a DataFrame's columns

```
retail_df.describe(include="all").round()
```

*Use **include="all"** to return statistics for all columns, or choose a specific data type to include*

*Note that the **.round()** method suppresses scientific notation and makes the output more readable*

	id	date	store_nbr	family	sales	onpromotion
count	3000888.0	3000888	3000888.0	3000888	3000888.0	3000888.0
unique	NaN	1684	NaN	33	NaN	NaN
top	NaN	2013-01-01	NaN	AUTOMOTIVE	NaN	NaN
freq	NaN	1782	NaN	90936	NaN	NaN
mean	1500444.0	NaN	28.0	NaN	358.0	3.0
std	866282.0	NaN	16.0	NaN	1102.0	12.0
min	0.0	NaN	1.0	NaN	0.0	0.0
25%	750222.0	NaN	14.0	NaN	0.0	0.0
50%	1500444.0	NaN	28.0	NaN	11.0	0.0
75%	2250665.0	NaN	41.0	NaN	196.0	0.0
max	3000887.0	NaN	54.0	NaN	124717.0	741.0

Unique values, most common value (top), and its frequency

ASSIGNMENT: EXPLORING DATAFRAMES

NEW MESSAGE

July 7, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Quick Statistics**

Hi there,

Can you dive a bit more deeply into the retail data and check if there are any missing values?

What about the number of unique dates? I want to make sure we didn't leave any out.

Finally, can you report the mean, median, min and max of "transactions"? I want to check for any anomalies in our data.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Results Preview

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 83488 entries, 0 to 83487
Data columns (total 3 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   date          83488 non-null  object
 1   store_nbr     83488 non-null  int64
 2   transactions  83488 non-null  int64
dtypes: int64(2), object(1)
memory usage: 1.9+ MB
```

	date	store_nbr	transactions
count	83488	83488.0	83488.0
unique	1682	NaN	NaN
top	2017-08-15	NaN	NaN
freq	54	NaN	NaN
mean	NaN	27.0	1695.0
std	NaN	16.0	963.0
min	NaN	1.0	5.0
25%	NaN	13.0	1046.0
50%	NaN	27.0	1393.0
75%	NaN	40.0	2079.0
max	NaN	54.0	8359.0

SOLUTION: EXPLORING DATAFRAMES

NEW MESSAGE

July 7, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Quick Statistics**

Hi there,

Can you dive a bit more deeply into the retail data and check if there are any missing values?

What about the number of unique dates? I want to make sure we didn't leave any out.

Finally, can you report the mean, median, min and max of "transactions"? I want to check for any anomalies in our data.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Solution Code

```
transactions.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 83488 entries, 0 to 83487
Data columns (total 3 columns):
 #   Column        Non-Null Count  Dtype 
---  ------        --------------  ----- 
 0   date          83488 non-null  object
 1   store_nbr     83488 non-null  int64 
 2   transactions  83488 non-null  int64 
dtypes: int64(2), object(1)
memory usage: 1.9+ MB
```

```
transactions.describe(include='all').round()
```

	date	store_nbr	transactions
count	83488	83488.0	83488.0
unique	1682	NaN	NaN
top	2017-08-15	NaN	NaN
freq	54	NaN	NaN
mean	NaN	27.0	1695.0
std	NaN	16.0	963.0
min	NaN	1.0	5.0
25%	NaN	13.0	1046.0
50%	NaN	27.0	1393.0
75%	NaN	40.0	2079.0
max	NaN	54.0	8359.0

ACCESING DATAFRAME COLUMNS

You can **access a DataFrame column** by using bracket or dot notation

- Dot notation only works for valid Python variable names *(no spaces, special characters, etc.)*, and if the column name is not the same as an existing variable or method

```
retail_df['family']
```

```
0                            AUTOMOTIVE
1                            BABY CARE
2                               BEAUTY
3                            BEVERAGES
4                                BOOKS
                              ...
3000883                        POULTRY
3000884                 PREPARED FOODS
3000885                        PRODUCE
3000886    SCHOOL AND OFFICE SUPPLIES
3000887                        SEAFOOD
Name: family, Length: 3000888, dtype: object
```

```
retail_df.family
```

```
0                            AUTOMOTIVE
1                            BABY CARE
2                               BEAUTY
3                            BEVERAGES
4                                BOOKS
                              ...
3000883                        POULTRY
3000884                 PREPARED FOODS
3000885                        PRODUCE
3000886    SCHOOL AND OFFICE SUPPLIES
3000887                        SEAFOOD
Name: family, Length: 3000888, dtype: object
```

PRO TIP: Even though you'll see many examples of dot notation in use, stick to bracket notation for single columns of data as it is less likely to cause issues

ACCESING DATAFRAME COLUMNS

You can **use Series operations** on DataFrame columns *(each column is a Series!)*

Number of unique values in a column

```
retail_df["family"].nunique()
```

```
33
```

Mean of values in a column

```
retail_df["sales"].mean()
```

```
357.77574911262707
```

First 5 unique values in a column with their frequencies

```
retail_df["family"].value_counts()[:5]
```

```
AUTOMOTIVE                    90936
HOME APPLIANCES               90936
SCHOOL AND OFFICE SUPPLIES    90936
PRODUCE                       90936
PREPARED FOODS                90936
Name: family, dtype: int64
```

Rounded sum of values in a column

```
retail_df["sales"].sum().round()
```

```
1073644952.0
```

ACCESING DATAFRAME COLUMNS

You can **select multiple columns** with a list of column names between brackets

- This is ideal for selecting non-consecutive columns in a DataFrame

```
retail_df[['family', 'store_nbr']]
```

	family	store_nbr
0	AUTOMOTIVE	1
1	BABY CARE	1
2	BEAUTY	1
3	BEVERAGES	1
4	BOOKS	1
...	...	...
3000883	POULTRY	9
3000884	PREPARED FOODS	9
3000885	PRODUCE	9
3000886	SCHOOL AND OFFICE SUPPLIES	9
3000887	SEAFOOD	9

3000888 rows × 2 columns

```
retail_df[['family', 'store_nbr']][:5]
```

	family	store_nbr
0	AUTOMOTIVE	1
1	BABY CARE	1
2	BEAUTY	1
3	BEVERAGES	1
4	BOOKS	1

PRO TIP: Use .loc() to access more than one column of data – column bracket notation should primarily be used for creating new columns and quick exploration

ACCESING DATA WITH ILOC

The .iloc() accessor filters DataFrames by their row and column indices

- The first parameter accesses rows, and the second accesses columns

First 5 rows, all columns

```
retail_df.iloc[:5, :]
```

	id	date	store_nbr	family	sales	onpromotion
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0
1	1	2013-01-01	1	BABY CARE	0.0	0
2	2	2013-01-01	1	BEAUTY	0.0	0
3	3	2013-01-01	1	BEVERAGES	0.0	0
4	4	2013-01-01	1	BOOKS	0.0	0

ACCESING DATA WITH ILOC

The .iloc() accessor filters DataFrames by their row and column indices

- The first parameter accesses rows, and the second accesses columns

All rows, 2-4 columns

```
retail_df.iloc[:, 1:4]
```

	date	store_nbr	family
0	2013-01-01	1	AUTOMOTIVE
1	2013-01-01	1	BABY CARE
2	2013-01-01	1	BEAUTY
3	2013-01-01	1	BEVERAGES
4	2013-01-01	1	BOOKS
...	...	...	...
3000883	2017-08-15	9	POULTRY
3000884	2017-08-15	9	PREPARED FOODS
3000885	2017-08-15	9	PRODUCE
3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES
3000887	2017-08-15	9	SEAFOOD

ACCESING DATA WITH ILOC

The .iloc() accessor filters DataFrames by their row and column indices

- The first parameter accesses rows, and the second accesses columns

First 5 rows, 2-4 columns

```
retail_df.iloc[:5, 1:4]
```

	date	store_nbr	family
0	2013-01-01	1	AUTOMOTIVE
1	2013-01-01	1	BABY CARE
2	2013-01-01	1	BEAUTY
3	2013-01-01	1	BEVERAGES
4	2013-01-01	1	BOOKS

ACCESING DATA WITH LOC

The .loc() accessor filters DataFrames by their row and column labels

- The first parameter accesses rows, and the second accesses columns

All rows, "date" column

```
retail_df.loc[:, "date"]
```

```
0           2013-01-01
1           2013-01-01
2           2013-01-01
3           2013-01-01
4           2013-01-01
               ...
3000883     2017-08-15
3000884     2017-08-15
3000885     2017-08-15
3000886     2017-08-15
3000887     2017-08-15
Name: date, Length: 3000888
```

This is a Series

```
retail_df.loc[:, ["date"]]
```

	date
0	2013-01-01
1	2013-01-01
2	2013-01-01
3	2013-01-01
4	2013-01-01
...	...
3000883	2017-08-15
3000884	2017-08-15
3000885	2017-08-15
3000886	2017-08-15
3000887	2017-08-15

3000888 rows × 1 columns

Wrap single columns in brackets to return a DataFrame

ACCESING DATA WITH LOC

The .loc() accessor filters DataFrames by their row and column labels

- The first parameter accesses rows, and the second accesses columns

All rows, "date" & "sales" columns

*(**list** of columns)*

```
retail_df.loc[:, ["date", "sales"]]
```

	date	sales
0	2013-01-01	0.000
1	2013-01-01	0.000
2	2013-01-01	0.000
3	2013-01-01	0.000
4	2013-01-01	0.000
...	...	...
3000883	2017-08-15	438.133
3000884	2017-08-15	154.553
3000885	2017-08-15	2419.729
3000886	2017-08-15	121.000
3000887	2017-08-15	16.000

3000888 rows × 2 columns

All rows, "date" through "sales" columns

*(**slice** of columns)*

```
retail_df.loc[:, "date":"sales"]
```

	date	store_nbr	family	sales
0	2013-01-01	1	AUTOMOTIVE	0.0
1	2013-01-01	1	BABY CARE	0.0
2	2013-01-01	1	BEAUTY	0.0
3	2013-01-01	1	BEVERAGES	0.0
4	2013-01-01	1	BOOKS	0.0
...	...	...	...	...
3000883	2017-08-15	9	POULTRY	438.133
3000884	2017-08-15	9	PREPARED FOODS	154.553
3000885	2017-08-15	9	PRODUCE	2419.729
3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.0
3000887	2017-08-15	9	SEAFOOD	16.0

3000888 rows × 4 columns

ASSIGNMENT: ACCESSING DATA

NEW MESSAGE

July 9, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **A Few More Stats....**

Hi there,

I noticed that the first row is the only one from 2013-01-01.

Can you get me a copy of the DataFrame that excludes that row, and only includes "store_nbr" and "transactions"?

Also, can you report the number of unique store numbers?

Finally, report the total number of transactions in millions.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Results Preview

	store_nbr	transactions
1	1	2111
2	2	2358
3	3	3487
4	4	1922
5	5	1903
...	...	...
83483	50	2804
83484	51	1573
83485	52	2255
83486	53	932
83487	54	802

83487 rows × 2 columns

```
54
```

```
141.478945
```

SOLUTION: ACCESSING DATA

NEW MESSAGE

July 9, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **A Few More Stats....**

Hi there,

I noticed that the first row is the only one from 2013-01-01.

Can you get me a copy of the DataFrame that excludes that row, and only includes "store_nbr" and "transactions"?

Also, can you report the number of unique store numbers?

Finally, report the total number of transactions in millions.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Solution Code

```
transactions.loc[1:, ['store_nbr', 'transactions']]
```

	store_nbr	transactions
1	1	2111
2	2	2358
3	3	3487
4	4	1922
5	5	1903
...	...	...
83483	50	2804
83484	51	1573
83485	52	2255
83486	53	932
83487	54	802

83487 rows × 2 columns

```
transactions.loc[:, 'store_nbr'].nunique()
```

```
54
```

```
transactions.transactions.sum() / 1000000
```

```
141.478945
```

DROPPING ROWS & COLUMNS

The .drop() method **drops rows and columns** from a DataFrame

- Specify axis=0 to drop rows by label, and axis=1 to drop columns

```
retail_df.drop("id", axis=1).head()
```

	date	store_nbr	family	sales	onpromotion
0	2013-01-01	1	AUTOMOTIVE	0.0	0
1	2013-01-01	1	BABY CARE	0.0	0
2	2013-01-01	1	BEAUTY	0.0	0
3	2013-01-01	1	BEVERAGES	0.0	0
4	2013-01-01	1	BOOKS	0.0	0

This returns the first 5 rows of the retail_df DataFrame without the "id" column

```
retail_df.drop(["id", "onpromotion"], inplace=True, axis=1)

retail_df.head()
```

	date	store_nbr	family	sales
0	2013-01-01	1	AUTOMOTIVE	0.0
1	2013-01-01	1	BABY CARE	0.0
2	2013-01-01	1	BEAUTY	0.0
3	2013-01-01	1	BEVERAGES	0.0
4	2013-01-01	1	BOOKS	0.0

*You can specify **inplace=True** to permanently remove rows or columns from a DataFrame*

PRO TIP: Drop unnecessary columns early in your workflow to save memory and make DataFrames more manageable *(ideally, they shouldn't be imported - more on that later!)*

DROPPING ROWS & COLUMNS

The .drop() method **drops rows and columns** from a DataFrame

- Specify axis=0 to drop rows by label, and axis=1 to drop columns

```
retail_df.drop([0], axis=0).head()
```

	date	store_nbr	family	sales
1	2013-01-01	1	BABY CARE	0.0
2	2013-01-01	1	BEAUTY	0.0
3	2013-01-01	1	BEVERAGES	0.0
4	2013-01-01	1	BOOKS	0.0
5	2013-01-01	1	BREAD/BAKERY	0.0

This returns the first 5 rows of the retail_df DataFrame after removing the first row

Note that the row label is passed as a list

```
retail_df.drop(range(5), axis=0).head()
```

	date	store_nbr	family	sales
5	2013-01-01	1	BREAD/BAKERY	0.0
6	2013-01-01	1	CELEBRATION	0.0
7	2013-01-01	1	CLEANING	0.0
8	2013-01-01	1	DAIRY	0.0
9	2013-01-01	1	DELI	0.0

You can pass a range to remove rows with consecutive labels, in this case 0-4

You'll typically drop rows via **slicing** or **filtering**, but it's worth being aware that .drop() can be used as well

IDENTIFYING DUPLICATE ROWS

The .duplicated() method **identifies duplicate rows** of data

- Specify subset=column(s) to look for duplicates across a subset of columns

```
product_df
```

	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

```
product_df.shape
```

```
(5, 2)
```

```
product_df.nunique()
```

```
product    3
price      4
dtype: int64
```

If the number of unique values for a column is less than the total number of rows, then that column contains duplicate values

IDENTIFYING DUPLICATE ROWS

The .duplicated() method **identifies duplicate rows** of data

- Specify subset=column(s) to look for duplicates across a subset of columns

```
product_df
```

	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

```
product_df.duplicated()
```

```
0    False
1     True
2    False
3    False
4    False
dtype: bool
```

The .duplicated() method returns True for the second row here because it is a duplicate of the first row

IDENTIFYING DUPLICATE ROWS

The .duplicated() method **identifies duplicate rows** of data

- Specify subset=column(s) to look for duplicates across a subset of columns

```
product_df
```

	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

```
product_df.duplicated(subset='product')
```

```
0    False
1     True
2     True
3    False
4    False
dtype: bool
```

*Specifying **subset='product'** will only look for duplicates in that column*

In this case rows 2 and 3 are duplicates of the first row ("Dairy")

DROPPING DUPLICATE ROWS

The .drop_duplicates() method **drops duplicate rows** from a DataFrame

- Specify subset=column(s) to look for duplicates across a subset of columns

```
product_df
```

	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

```
product_df.drop_duplicates()
```

	product	price
0	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

This removed the second row from the product_df DataFrame, as it is a duplicate of the first row

Note that the row index now has a gap between 0 & 2

DROPPING DUPLICATE ROWS

The .drop_duplicates() method **drops duplicate rows** from a DataFrame

- Specify subset=column(s) to look for duplicates across a subset of columns

```
product_df
```

	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

```
product_df.drop_duplicates(subset="product", keep="last", ignore_index=True)
```

	product	price
0	Dairy	4.55
1	Vegetables	2.74
2	Fruits	5.44

How does this code work?

- **subset="product"** will look for duplicates in the product column *(index 0, 1, and 2 for "Dairy")*
- **keep="last"** will keep the final duplicate row, and drop the rest
- **ignore_index=True** will reset the index so there are no gaps

ASSIGNMENT: DROPPING DATA

NEW MESSAGE

July 9, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Reducing The Data**

Hi there,

Can you drop the first row of data this time? A slice is fine for viewing but we want this permanently removed. Also, drop the date column, but not in place.

Then, can you get me a DataFrame that includes only the last row for each of our stores? I want to take a look at the most recent data for each.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Results Preview

```
transactions [illegible]

transactions.head()
```

	date	store_nbr	transactions
1	2013-01-02	1	2111
2	2013-01-02	2	2358
3	2013-01-02	3	3487
4	2013-01-02	4	1922
5	2013-01-02	5	1903

```
transactions [illegible] .head()
```

	store_nbr	transactions
1	1	2111
2	2	2358
3	3	3487
4	4	1922
5	5	1903

```
transactions [illegible] .head()
```

	date	store_nbr	transactions
83434	2017-08-15	1	1693
83435	2017-08-15	2	1737
83436	2017-08-15	3	2956
83437	2017-08-15	4	1283
83438	2017-08-15	5	1310

SOLUTION: DROPPING DATA

NEW MESSAGE

July 9, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Reducing The Data**

Hi there,

Can you drop the first row of data this time? A slice is fine for viewing but we want this permanently removed. Also, drop the date column, but not in place.

Then, can you get me a DataFrame that includes only the last row for each of our stores? I want to take a look at the most recent data for each.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Solution Code

```
transactions.drop(0, axis=0, inplace=True)

transactions.head()
```

	date	store_nbr	transactions
1	2013-01-02	1	2111
2	2013-01-02	2	2358
3	2013-01-02	3	3487
4	2013-01-02	4	1922
5	2013-01-02	5	1903

```
transactions.drop('date', axis=1, inplace=False).head()
```

	store_nbr	transactions
1	1	2111
2	2	2358
3	3	3487
4	4	1922
5	5	1903

```
transactions.drop_duplicates(subset='store_nbr', keep='last').head()
```

	date	store_nbr	transactions
83434	2017-08-15	1	1693
83435	2017-08-15	2	1737
83436	2017-08-15	3	2956
83437	2017-08-15	4	1283
83438	2017-08-15	5	1310

IDENTIFYING MISSING DATA

You can **identify missing data** by column using the .isna() and .sum() methods

- The .info() method can also help identify null values

```
product_df
```

	product	price	product_id
0	<NA>	2.56	1
1	Dairy	<NA>	2
2	Dairy	4.55	3
3	<NA>	2.74	4
4	Fruits	<NA>	5

```
product_df.isna().sum()
```

```
product       2
price         2
product_id    0
dtype: int64
```

This is a Series

*The **.isna()** method returns a DataFrame with Boolean values (True for NAs, False for others)*

*The **.sum()** method adds these for each column (True=1, False=0) and returns the summarized results*

```
product_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   product     3 non-null      object
 1   price       3 non-null      object
 2   product_id  5 non-null      int64
dtypes: int64(1), object(2)
memory usage: 248.0+ bytes
```

The difference between the total entries and non-null values for each column gives you the missing values in each

HANDLING MISSING DATA

Like with Series, the .dropna() and .fillna() methods let you **handle missing data** in a DataFrame by either removing them or replacing them with other values

```
product_df
```

	product	price
0	<NA>	2.56
1	Dairy	<NA>
2	Dairy	4.55
3	<NA>	2.74
4	Fruits	<NA>

```
product_df.fillna(0)
```

	product	price
0	0	2.56
1	Dairy	0.00
2	Dairy	4.55
3	0	2.74
4	Fruits	0.00

```
product_df.fillna({"price": 0})
```

Use a dictionary to specify a value for each column

	product	price
0	<NA>	2.56
1	Dairy	0.00
2	Dairy	4.55
3	<NA>	2.74
4	Fruits	0.00

```
product_df.dropna()
```

This drops any row with missing values

	product	price
2	Dairy	4.55

```
product_df.dropna(subset=["price"])
```

Use subset to drop rows with missing values in specified columns

	product	price
0	<NA>	2.56
2	Dairy	4.55
3	<NA>	2.74

ASSIGNMENT: MISSING DATA

NEW MESSAGE

July 12, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Missing Price Data**

Hi again,

I was reviewing some of Rachel Revenue's work on the oil price data as I was closing the books and I noticed quite a few missing dates and values, so now I'm concerned…

Can you tell if any dates or prices are missing?

I'd like to see the mean oil price if we fill in the missing values with 0 and compare it to the mean oil price if we fill them in with the mean.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Results Preview

```
oil = pd.read_csv('../retail/oil.csv')
```

```
date           0
dcoilwtico    43
dtype: int64
```

```
65.32379310344835
```

```
67.71436595744696
```

SOLUTION: MISSING DATA

NEW MESSAGE

July 12, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Missing Price Data**

Hi again,

I was reviewing some of Rachel Revenue's work on the oil price data as I was closing the books and I noticed quite a few missing dates and values, so now I'm concerned…

Can you tell if any dates or prices are missing?

I'd like to see the mean oil price if we fill in the missing values with 0 and compare it to the mean oil price if we fill them in with the mean.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Solution Code

```
oil = pd.read_csv('../retail/oil.csv')
```

```
oil.isna().sum()
```

```
date           0
dcoilwtico    43
dtype: int64
```

```
oil['dcoilwtico'].fillna(0).mean()
```

```
65.32379310344835
```

```
oil['dcoilwtico'].fillna(oil['dcoilwtico'].mean()).mean()
```

```
67.71436595744696
```

FILTERING DATAFRAMES

You can **filter the rows in a DataFrame** by passing a logical test into the .loc[] accessor, just like filtering a Series or a NumPy array

```
retail_df.loc[retail_df["date"] == "2016-10-28"]
```

	id	date	store_nbr	family	sales	onpromotion
2482326	2482326	2016-10-28	1	AUTOMOTIVE	8.000	0
2482327	2482327	2016-10-28	1	BABY CARE	0.000	0
2482328	2482328	2016-10-28	1	BEAUTY	9.000	1
2482329	2482329	2016-10-28	1	BEVERAGES	2576.000	38
2482330	2482330	2016-10-28	1	BOOKS	0.000	0
...	...	...	...	...	...	...
2484103	2484103	2016-10-28	9	POULTRY	391.292	24
2484104	2484104	2016-10-28	9	PREPARED FOODS	78.769	1
2484105	2484105	2016-10-28	9	PRODUCE	993.760	5
2484106	2484106	2016-10-28	9	SCHOOL AND OFFICE SUPPLIES	0.000	0
2484107	2484107	2016-10-28	9	SEAFOOD	3.000	1

1782 rows × 6 columns

This filters the retail_df DataFrame and only returns rows where the date is equal to "2016-10-28"

FILTERING DATAFRAMES

You can **filter the columns in a DataFrame** by passing them into the .loc[] accessor as a list or a slice

```
retail_df.loc[retail_df["date"] == "2016-10-28", ["date", "sales"]].head()
```

Row filter

Column filter

	date	sales
2482326	2016-10-28	8.0
2482327	2016-10-28	0.0
2482328	2016-10-28	9.0
2482329	2016-10-28	2576.0
2482330	2016-10-28	0.0

This filters the retail_df DataFrame to the columns selected, and only returns rows where the date is equal to "2016-10-28"

FILTERING DATAFRAMES

You can **apply multiple filters** by joining the logical tests with an "&" operator

- Try creating a Boolean mask for creating filters with complex logic

```
conds = retail_df["family"].isin(["CLEANING", "DAIRY"]) & (retail_df["sales"] > 0)

retail_df.loc[conds]
```

	id	date	store_nbr	family	sales	onpromotion
568	568	2013-01-01	25	CLEANING	186.0	0
569	569	2013-01-01	25	DAIRY	143.0	0
1789	1789	2013-01-02	1	CLEANING	1060.0	0
1790	1790	2013-01-02	1	DAIRY	579.0	0
1822	1822	2013-01-02	10	CLEANING	1110.0	0

*The Boolean mask here is filtering the DataFrame for rows where the family is "CLEANING" or "DAIRY", **and** the sales are greater than 0*

PRO TIP: QUERY

The **.query()** method lets you use SQL-like syntax to filter DataFrames

- You can specify any number of filtering conditions by using the *"and"* & *"or"* keywords

```
retail_df.query("family in ['CLEANING', 'DAIRY'] and sales > 0")
```

	id	date	store_nbr	family	sales	onpromotion
568	568	2013-01-01	25	CLEANING	186.0	0
569	569	2013-01-01	25	DAIRY	143.0	0
1789	1789	2013-01-02	1	CLEANING	1060.0	0
1790	1790	2013-01-02	1	DAIRY	579.0	0
1822	1822	2013-01-02	10	CLEANING	1110.0	0
...	...	...	...	...	...	...
3000797	3000797	2017-08-15	7	DAIRY	1279.0	25
3000829	3000829	2017-08-15	8	CLEANING	1198.0	13
3000830	3000830	2017-08-15	8	DAIRY	1330.0	24
3000862	3000862	2017-08-15	9	CLEANING	1439.0	25
3000863	3000863	2017-08-15	9	DAIRY	835.0	19

*This query filters rows where the family is "CLEANING" or "DAIRY", **and** the sales are greater than 0*

Note that you don't need to call the DataFrame name repeatedly, saving keystrokes and making the `filter` easier to interpret

PRO TIP: QUERY

The .query() method lets you use SQL-like syntax to filter DataFrames

- You can specify any number of filtering conditions by using the *"and"* & *"or"* keywords
- You can reference variables by using the *"@"* symbol

```
avg_sales = retail_df.loc[:, "sales"].mean()

avg_sales
```

```
357.77574911262707
```

```
retail_df.query("family in ['CLEANING', 'DAIRY'] and sales > @avg_sales")
```

	id	date	store_nbr	family	sales	onpromotion
1789	1789	2013-01-02	1	CLEANING	1060.0	0
1790	1790	2013-01-02	1	DAIRY	579.0	0
1822	1822	2013-01-02	10	CLEANING	1110.0	0
1855	1855	2013-01-02	11	CLEANING	3260.0	0
1888	1888	2013-01-02	12	CLEANING	1092.0	0

... output truncated

*This query filters rows where the family is "CLEANING" or "DAIRY", **and** the sales are greater than the avg_sales value (from the variable!)*

ASSIGNMENT: FILTERING DATAFRAMES

NEW MESSAGE

July 12, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Store 25 Deep Dive**

I need some quick research on store 25:

- First, calculate the percentage of times ALL stores had more than 2000 transactions
- Then, calculate the percentage of times store 25 had more than 2000 transactions, and calculate the sum of transactions on these days
- Finally, sum the transactions for stores 25 and 3, that occurred in May or June, and had less than 2000 transactions

section03_DataFrames.ipynb

Reply Forward

Results Preview

```
0.266808006036868
```

```
0.03469640644361834
```

```
144903
```

```
644910
```

SOLUTION: FILTERING DATAFRAMES

NEW MESSAGE

July 12, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Store 25 Deep Dive**

I need some quick research on store 25:

- First, calculate the percentage of times ALL stores had more than 2000 transactions
- Then, calculate the percentage of times store 25 had more than 2000 transactions, and calculate the sum of transactions on these days
- Finally, sum the transactions for stores 25 and 3, that occurred in May or June, and had less than 2000 transactions

section03_DataFrames.ipynb

Reply Forward

Solution Code

```
(transactions['transactions'] > 2000).mean()
```

```
0.266808006036868
```

```
mask = (transactions["store_nbr"] == 25) & (transactions["transactions"] > 2000)

(transactions.loc[mask].count()
 / transactions.loc[transactions["store_nbr"] == 25].count()
).iloc[2]
```

```
0.03469640644361834
```

```
transactions[mask].loc[:, "transactions"].sum()
```

```
144903
```

```
transactions.query(
    "store_nbr in [25, 31] & date.str[6] in ['5', '6'] & transactions < 2000"
).transactions.sum()
```

```
644910
```

SORTING DATAFRAMES BY INDICES

You can **sort a DataFrame by its indices** using the .sort_index() method

- This sorts rows (axis=0) by default, but you can specify axis=1 to sort the columns

```
condition = retail_df.family.isin(["BEVERAGES", "DELI", "DAIRY"])

sample_df = retail_df[condition].sample(5, random_state=2021)

sample_df
```

	id	date	store_nbr	family	sales	onpromotion
2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24
2356175	2356175	2016-08-18	2	DAIRY	714.0	7
1691390	1691390	2015-08-10	17	DAIRY	613.0	0
1435443	1435443	2015-03-19	35	DELI	134.0	24
2939747	2939747	2017-07-12	43	DAIRY	628.0	120

This creates a sample DataFrame by filtering rows for the 3 specified product families, and grabbing 5 random rows

```
sample_df.sort_index(ascending=False)
```

	id	date	store_nbr	family	sales	onpromotion
2939747	2939747	2017-07-12	43	DAIRY	628.0	120
2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24
2356175	2356175	2016-08-18	2	DAIRY	714.0	7
1691390	1691390	2015-08-10	17	DAIRY	613.0	0
1435443	1435443	2015-03-19	35	DELI	134.0	24

+

-

This sorts the sample DataFrame in descending order by its row index

(it sorts in ascending order by default)

SORTING DATAFRAMES BY INDICES

You can **sort a DataFrame by its indices** using the .sort_index() method

- This sorts rows (axis=0) by default, but you can specify axis=1 to sort the columns

```
condition = retail_df.family.isin(["BEVERAGES", "DELI", "DAIRY"])

sample_df = retail_df[condition].sample(5, random_state=2021)

sample_df
```

	id	date	store_nbr	family	sales	onpromotion
2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24
2356175	2356175	2016-08-18	2	DAIRY	714.0	7
1691390	1691390	2015-08-10	17	DAIRY	613.0	0
1435443	1435443	2015-03-19	35	DELI	134.0	24
2939747	2939747	2017-07-12	43	DAIRY	628.0	120


```
sample_df.sort_index(axis=1, inplace=True)

sample_df
```

	date	family	id	onpromotion	sales	store_nbr
2782266	2017-04-15	BEVERAGES	2782266	24	5516.0	25
2356175	2016-08-18	DAIRY	2356175	7	714.0	2
1691390	2015-08-10	DAIRY	1691390	0	613.0	17
1435443	2015-03-19	DELI	1435443	24	134.0	35
2939747	2017-07-12	DAIRY	2939747	120	628.0	43

This sorts the sample DataFrame in ascending order by its column index, and modifies the underlying values

SORTING DATAFRAMES BY VALUES

You can **sort a DataFrame by its values** using the .sort_values() method

- You can sort by a single column or by multiple columns

```
sample_df.sort_values("store_nbr")
```

	date	family	id	onpromotion	sales	store_nbr
2356175	2016-08-18	DAIRY	2356175	7	714.0	2
1691390	2015-08-10	DAIRY	1691390	0	613.0	17
2782266	2017-04-15	BEVERAGES	2782266	24	5516.0	25
1435443	2015-03-19	DELI	1435443	24	134.0	35
2939747	2017-07-12	DAIRY	2939747	120	628.0	43

This sorts the sample DataFrame by the values in the store_nbr column in ascending order by default

```
sample_df.sort_values(["family", "sales"], ascending=[True, False])
```

	id	date	store_nbr	family	sales	onpromotion
2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24
2356175	2356175	2016-08-18	2	DAIRY	714.0	7
2939747	2939747	2017-07-12	43	DAIRY	628.0	120
1691390	1691390	2015-08-10	17	DAIRY	613.0	0
1435443	1435443	2015-03-19	35	DELI	134.0	24

This sorts the sample DataFrame by the values in the family column in ascending order, then by the values in the sales column in descending order within each family

ASSIGNMENT: SORTING DATAFRAMES

NEW MESSAGE

July 13, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Sorting help!**

Hi there,

Can you get me a dataset that includes the 5 days with the highest transactions counts? Any similarities between them?

Then, can you get me a dataset sorted by date from earliest to most recent, but with the highest transactions first and the lowest transactions last for each day?

Finally, sort the columns in reverse alphabetical order.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Results Preview

	date	store_nbr	transactions
52011	2015-12-23	44	8359
71010	2016-12-23	44	8307
16570	2013-12-23	44	8256
33700	2014-12-23	44	8120
16572	2013-12-23	46	8001

	date	store_nbr	transactions
40	2013-01-02	46	4886
38	2013-01-02	44	4821
39	2013-01-02	45	4208
41	2013-01-02	47	4161
11	2013-01-02	11	3547

	transactions	store_nbr	date
1	2111	1	2013-01-02
2	2358	2	2013-01-02
3	3487	3	2013-01-02
4	1922	4	2013-01-02
5	1903	5	2013-01-02

SOLUTION: SORTING DATAFRAMES

NEW MESSAGE

July 13, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Sorting help!**

Hi there,

Can you get me a dataset that includes the 5 days with the highest transactions counts? Any similarities between them?

Then, can you get me a dataset sorted by date from earliest to most recent, but with the highest transactions first and the lowest transactions last for each day?

Finally, sort the columns in reverse alphabetical order.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Solution Code

```
transactions.sort_values(by=['transactions'], ascending=False).iloc[:5, :]
```

	date	store_nbr	transactions
52011	2015-12-23	44	8359
71010	2016-12-23	44	8307
16570	2013-12-23	44	8256
33700	2014-12-23	44	8120
16572	2013-12-23	46	8001

```
transactions.sort_values(by=["date", "transactions"], ascending=[True, False])
```

	date	store_nbr	transactions
40	2013-01-02	46	4886
38	2013-01-02	44	4821
39	2013-01-02	45	4208
41	2013-01-02	47	4161
11	2013-01-02	11	3547

```
transactions.sort_index(axis=1, ascending=False)
```

	transactions	store_nbr	date
1	2111	1	2013-01-02
2	2358	2	2013-01-02
3	3487	3	2013-01-02
4	1922	4	2013-01-02
5	1903	5	2013-01-02
...	...	...	...

RENAMING COLUMNS

Rename columns in place via assignment using the "columns" property

```
product_df
```

	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

```
product_df.columns = ["product_name", "cost"]

product_df
```

*Simply assign a list with the new column names using the **columns** property*

	product_name	cost
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

```
product_df.columns = [col.upper() for col in product_df.columns]

product_df
```

*Use a **list comprehension** to clean or standardize column titles using methods like .upper()*

	PRODUCT_NAME	COST
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

RENAMING COLUMNS

You can also **rename columns** with the .rename() method

```
product_df
```

	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

```
product_df.rename(columns={'product': 'product_name', 'price': 'cost'})
```

Use a dictionary to map the new column names to the old names

	product_name	cost
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

```
product_df.rename(columns=lambda x: x.upper())
```

*Use **lambda functions** to clean or standardize column titles using methods like .upper()*

	PRODUCT	PRICE
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

Note that the .rename() method **doesn't rename in place by default**, so you can chain methods together

REORDERING COLUMS

Reorder columns with the .reindex() method when sorting won't suffice

```
product_df
```

	product	price	product_id
0	Dairy	2.56	1
1	Dairy	2.56	2
2	Dairy	4.55	3
3	Vegetables	2.74	4
4	Fruits	5.44	5

```
product_df.reindex(labels=["product_id", "product", "price"], axis=1)
```

	product_id	product	price
0	1	Dairy	2.56
1	2	Dairy	2.56
2	3	Dairy	4.55
3	4	Vegetables	2.74
4	5	Fruits	5.44

Pass a list of the existing columns in their desired order, and specify axis=1

ASSIGNMENT: MODIFYING COLUMNS

NEW MESSAGE

July 15, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Cosmetic Changes**

Hi again,

Just some quick work, but can you send me the transaction data with the columns renamed?

I want them looking a bit prettier for my presentation – you can find more details in the notebook.

Also, could you reorder the columns so date is first, then transaction count, then store number?

Thanks!

section03_DataFrames.ipynb

Reply Forward

Results Preview

```
transactions.head()
```

	date	store_number	transaction_count
1	2013-01-02	1	2111
2	2013-01-02	2	2358
3	2013-01-02	3	3487
4	2013-01-02	4	1922
5	2013-01-02	5	1903

	date	transaction_count	store_number
1	2013-01-02	2111	1
2	2013-01-02	2358	2
3	2013-01-02	3487	3
4	2013-01-02	1922	4
5	2013-01-02	1903	5

SOLUTION: MODIFYING COLUMNS

NEW MESSAGE

July 15, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Cosmetic Changes**

Hi again,

Just some quick work, but can you send me the transaction data with the columns renamed?

I want them looking a bit prettier for my presentation – you can find more details in the notebook.

Also, could you reorder the columns so date is first, then transaction count, then store number?

Thanks!

section03_DataFrames.ipynb

Reply Forward

Solution Code

```
transactions = transactions.rename(
    columns={"transactions": "transaction_count", "store_nbr": "store_number"}
)
```

```
transactions.head()
```

	date	store_number	transaction_count
1	2013-01-02	1	2111
2	2013-01-02	2	2358
3	2013-01-02	3	3487
4	2013-01-02	4	1922
5	2013-01-02	5	1903

```
transactions.reindex(
    labels=["date", "transaction_count", "store_number"], axis=1
).head()
```

	date	transaction_count	store_number
1	2013-01-02	2111	1
2	2013-01-02	2358	2
3	2013-01-02	3487	3
4	2013-01-02	1922	4
5	2013-01-02	1903	5

ARITHMETIC COLUMN CREATION

You can **create columns with arithmetic** by assigning them Series operations

- Simply specify the new column name and assign the operation of interest

```
baby_books
```

	id	date	store_nbr	family	sales	onpromotion
2390356	2390356	2016-09-06	29	BABY CARE	3.0	0
2691613	2691613	2017-02-23	31	BABY CARE	1.0	0
2538925	2538925	2016-11-28	47	BOOKS	6.0	0
2022637	2022637	2016-02-13	11	BABY CARE	1.0	0
2585356	2585356	2016-12-24	5	BOOKS	2.0	0

```
baby_books["tax_amount"] = baby_books["sales"] * 0.05
baby_books["total"] = baby_books["sales"] + baby_books["tax_amount"]
baby_books
```

*This creates a new **tax_amount** column equal to **sales** * **0.05***

*This creates a new **total** column equal to **sales** + **tax_amount***

	id	date	store_nbr	family	sales	onpromotion	tax_amount	total
2390356	2390356	2016-09-06	29	BABY CARE	3.0	0	0.15	3.15
2691613	2691613	2017-02-23	31	BABY CARE	1.0	0	0.05	1.05
2538925	2538925	2016-11-28	47	BOOKS	6.0	0	0.30	6.30
2022637	2022637	2016-02-13	11	BABY CARE	1.0	0	0.05	1.05
2585356	2585356	2016-12-24	5	BOOKS	2.0	0	0.10	2.10

The new columns are added to the end of the DataFrame by default

BOOLEAN COLUMN CREATION

You can **create Boolean columns** by assigning them a logical test

```
baby_books["taxable_category"] = baby_books["family"] != "BABY CARE"

baby_books
```

	id	date	store_nbr	family	sales	onpromotion	taxable_category
2390356	2390356	2016-09-06	29	BABY CARE	3.0	0	False
2691613	2691613	2017-02-23	31	BABY CARE	1.0	0	False
2538925	2538925	2016-11-28	47	BOOKS	6.0	0	True
2022637	2022637	2016-02-13	11	BABY CARE	1.0	0	False
2585356	2585356	2016-12-24	5	BOOKS	2.0	0	True

*This creates a new **taxable_category** column with Boolean values - True if the family is not "BABY CARE", and False if it is*

```
baby_books["tax_amount"] = (
    baby_books["sales"] * 0.05 * (baby_books["family"] != "BABY CARE")
)

baby_books
```

	id	date	store_nbr	family	sales	onpromotion	tax_amount
2390356	2390356	2016-09-06	29	BABY CARE	3.0	0	0.0
2691613	2691613	2017-02-23	31	BABY CARE	1.0	0	0.0
2538925	2538925	2016-11-28	47	BOOKS	6.0	0	0.3
2022637	2022637	2016-02-13	11	BABY CARE	1.0	0	0.0
2585356	2585356	2016-12-24	5	BOOKS	2.0	0	0.1

*This creates a new **tax_amount** column by leveraging both Boolean logic & arithmetic:*

*If the family is not "BABY CARE", then calculate the sales tax (sales * 0.05 * 1), otherwise return zero (sales * 0.05 * 0)*

ASSIGNMENT: COLUMN CREATION

NEW MESSAGE

July 18, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Bonus Calculations**

Doing some work on bonus estimates and I need help!!!

Create a 'pct_to_target' column that divides transactions by 2500 – our transaction target for all stores. Then create a 'met_target' column that is True if 'pct_to_target' is greater than or equal to 1, and False if not. Finally, create a 'bonus_payable' column that equals 100 if 'met_target' is True, and 0 if not, then sum the total 'bonus_payable' amount.

Finally, create columns for month and day of week as integers. I put some code for the date parts in the notebook.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Results Preview

	date	store_number	transaction_count	pct_to_target	met_target	bonus_payable	month	day_of_week	day
1	2013-01-02	1	2111	0.8444	False	0	1	2	2
2	2013-01-02	2	2358	0.9432	False	0	1	2	2
3	2013-01-02	3	3487	1.3948	True	100	1	2	2
4	2013-01-02	4	1922	0.7688	False	0	1	2	2
5	2013-01-02	5	1903	0.7612	False	0	1	2	2
...	...	...	...	...	...	...	...	...	...
83483	2017-08-15	50	2804	1.1216	True	100	8	15	15
83484	2017-08-15	51	1573	0.6292	False	0	8	15	15
83485	2017-08-15	52	2255	0.9020	False	0	8	15	15
83486	2017-08-15	53	932	0.3728	False	0	8	15	15
83487	2017-08-15	54	802	0.3208	False	0	8	15	15

```
transactions["bonus_payable"].sum()
```

```
1448300
```

SOLUTION: COLUMN CREATION

NEW MESSAGE

July 18, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Bonus Calculations**

Doing some work on bonus estimates and I need help!!!

Create a 'pct_to_target' column that divides transactions by 2500 – our transaction target for all stores. Then create a 'met_target' column that is True if 'pct_to_target' is greater than or equal to 1, and False if not. Finally, create a 'bonus_payable' column that equals 100 if 'met_target' is True, and 0 if not, then sum the total 'bonus_payable' amount.

Finally, create columns for month and day of week as integers. I put some code for the date parts in the notebook.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Solution Code

```
# Bonus Columns
transactions["pct_to_target"] = transactions["transaction_count"] / 2500
transactions["met_target"] = transactions["pct_to_target"] >= 1
transactions["bonus_payable"] = (transactions["pct_to_target"] >= 1) * 100

# Date Columns
transactions["date"] = transactions.date.astype("Datetime64")
transactions["month"] = transactions["date"].dt.month
transactions["day_of_week"] = transactions["date"].dt.dayofweek

transactions
```

	date	store_number	transaction_count	pct_to_target	met_target	bonus_payable	month	day_of_week
1	2013-01-02	1	2111	0.8444	False	0	1	2
2	2013-01-02	2	2358	0.9432	False	0	1	2
3	2013-01-02	3	3487	1.3948	True	100	1	2
4	2013-01-02	4	1922	0.7688	False	0	1	2
5	2013-01-02	5	1903	0.7612	False	0	1	2
...	...	...	...	...	...	...	...	...
83483	2017-08-15	50	2804	1.1216	True	100	8	1
83484	2017-08-15	51	1573	0.6292	False	0	8	1
83485	2017-08-15	52	2255	0.9020	False	0	8	1
83486	2017-08-15	53	932	0.3728	False	0	8	1
83487	2017-08-15	54	802	0.3208	False	0	8	1

83487 rows × 8 columns

PRO TIP: NUMPY SELECT

NumPy's **select()** function lets you create columns based on multiple conditions

- This is more flexible than NumPy's where() function or Pandas' .where() method

```
conditions = [
    (baby_books["date"] == "2017-02-23") & (baby_books["family"] == "BABY CARE"),
    (baby_books["date"] == "2016-12-24") & (baby_books["family"] == "BOOKS"),
    (baby_books["date"] == "2016-09-06") & (baby_books["store_nbr"] > 28),
]

choices = ["Winter Clearance", "Christmas Eve", "New Store Special"]

baby_books["Sale_Name"] = np.select(conditions, choices, default="No Sale")

baby_books
```

*Specify a set of **conditions** and outcomes (**choices**) for each condition*

*Then use **np.select** and pass in the conditions, the choices, and an optional default outcome if none of the conditions are met to the new Sale_Name column*

	id	date	store_nbr	family	sales	onpromotion	tax_amount	total	taxable_category	Sale_Name
2390356	2390356	2016-09-06	29	BABY CARE	3.0	0	0.0	3.0	False	New Store Special
2691613	2691613	2017-02-23	31	BABY CARE	1.0	0	0.0	1.0	False	Winter Clearance
2538925	2538925	2016-11-28	47	BOOKS	6.0	0	0.3	6.3	True	No Sale
2022637	2022637	2016-02-13	11	BABY CARE	1.0	0	0.0	1.0	False	No Sale
2585356	2585356	2016-12-24	5	BOOKS	2.0	0	0.1	2.1	True	Christmas Eve

The first condition is met, so the first choice is returned

ASSIGNMENT: NUMPY SELECT

NEW MESSAGE

July 20, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Seasonal Bonuses**

Create a 'seasonal_bonus' column that applies to these dates:

- All days in December
- Sundays in May
- Mondays in August

Call the December bonus 'Holiday Bonus', the May bonus 'Corporate Month', and the August bonus 'Summer Special'. If no bonus applies, the column should display 'None'.

Finally, calculate the total bonus owed at $100 per day.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Results Preview

```
transactions["seasonal_bonus"].value_counts()
```

```
None              75258
Holiday Bonus      6028
Summer Special     1103
Corporate Month    1098
Name: seasonal_bonus, dtype: int64
```

```
7409700
```

SOLUTION: NUMPY SELECT

NEW MESSAGE

July 20, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Seasonal Bonuses**

Create a 'seasonal_bonus' column that applies to these dates:

- All days in December
- Sundays in May
- Mondays in August

Call the December bonus 'Holiday Bonus', the May bonus 'Corporate Month', and the August bonus 'Summer Special'. If no bonus applies, the column should display 'None'.

Finally, calculate the total bonus owed at $100 per day.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Solution Code

```
conditions = [
    (transactions["month"] == 12),
    (transactions["month"] == 5) & (transactions["day_of_week"] == 6),
    (transactions["month"] == 7) & (transactions["day_of_week"] == 0),
]

choices = ["Holiday Bonus", "Corporate Month", "Summer Special"]

transactions["seasonal_bonus"] = np.select(conditions, choices, default="None")
```

```
transactions.loc[transactions['seasonal_bonus'] != 'None'].count().mul(100).sum()
```

```
7409700
```

MAPPING VALUES TO COLUMNS

The .map() method **maps values to a column** or an entire DataFrame

- You can pass a dictionary with existing values as the keys, and new values as the values

```
mapping_dict = {"Dairy": "Non-Vegan", "Vegetables": "Vegan", "Fruits": "Vegan"}

product_df["Vegan?"] = product_df["product"].map(mapping_dict)

product_df
```

The dictionary keys will be mapped to the values in the column selected

*This creates a new **Vegan?** column by mapping the dictionary keys to the values in the **product** column and returning the corresponding dictionary values in each row*

	product_id	product	price	Vegan?
0	1	Dairy	2.56	Non-Vegan
1	2	Dairy	2.56	Non-Vegan
2	3	Dairy	4.55	Non-Vegan
3	4	Vegetables	2.74	Vegan
4	5	Fruits	5.44	Vegan
		Keys		*Values*

MAPPING VALUES TO COLUMNS

The .map() method **maps values to a column** or an entire DataFrame

- You can pass a dictionary with existing values as the keys, and new values as the values
- You can apply lambda functions *(and others!)*

```
product_df["price"] = product_df["price"].map(lambda x: f"${x}")

product_df
```

	product_id	product	price	Vegan?
0	1	Dairy	$2.56	Non-Vegan
1	2	Dairy	$2.56	Non-Vegan
2	3	Dairy	$4.55	Non-Vegan
3	4	Vegetables	$2.74	Vegan
4	5	Fruits	$5.44	Vegan

*This overwrites the **price** column by adding a dollar sign to each of the previous values*

PRO TIP: COLUMN CREATION WITH ASSIGN

The .assign() method **creates multiple columns** at once and returns a DataFrame

- This can be chained together with other data processing methods

```
sample_df.assign(tax_amount=sample_df["sales"] * 0.05)
```

	id	date	store_nbr	family	sales	onpromotion	tax_amount
2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24	275.80
2356175	2356175	2016-08-18	2	DAIRY	714.0	7	35.70
1691390	1691390	2015-08-10	17	DAIRY	613.0	0	30.65
1435443	1435443	2015-03-19	35	DELI	134.0	24	6.70
2939747	2939747	2017-07-12	43	DAIRY	628.0	120	31.40

*To create a column using **.assign()**, simply specify the column name and assign its values as you normally would*

(arithmetic, Boolean logic, mapping, etc.)

```
sample_df.assign(
    tax_amount=(sample_df["sales"] * 0.05).round(2).map(lambda x: f"${x}"),
    on_promotion_flag=np.where(sample_df["onpromotion"] > 0, True, False),
    year=sample_df["date"].str.slice(0, 4).astype("int"),
).query("family == 'DAIRY'")
```

	id	date	store_nbr	family	sales	onpromotion	tax_amount	on_promotion_flag	year
2356175	2356175	2016-08-18	2	DAIRY	714.0	7	$35.7	True	2016
1691390	1691390	2015-08-10	17	DAIRY	613.0	0	$30.65	False	2015
2939747	2939747	2017-07-12	43	DAIRY	628.0	120	$31.4	True	2017

*To create multiple columns using **.assign()**, simply separate them using commas*

*Note that this is chained with **.query()** at the end to filter the DataFrame once the new columns are created*

PRO TIP: COLUMN CREATION WITH ASSIGN

How does this code work?

```
sample_df.assign(
    tax_amount=(sample_df["sales"] * 0.05).round(2).map(lambda x: f"${x}"),
    on_promotion_flag=np.where(sample_df["onpromotion"] > 0, True, False),
    year=sample_df["date"].str.slice(0, 4).astype("int"),
).query("family == 'DAIRY'")
```

	date	family	id	onpromotion	sales	store_nbr
2782266	2017-04-15	BEVERAGES	2782266	24	5516.0	25
2356175	2016-08-18	DAIRY	2356175	7	714.0	2
1691390	2015-08-10	DAIRY	1691390	0	613.0	17
1435443	2015-03-19	DELI	1435443	24	134.0	35
2939747	2017-07-12	DAIRY	2939747	120	628.0	43

*Let's create some columns for the **sample_df** DataFrame!*

PRO TIP: COLUMN CREATION WITH ASSIGN

How does this code work?

```
sample_df.assign(
  tax_amount=(sample_df["sales"] * 0.05).round(2).map(lambda x: f"${x}"),
    on_promotion_flag=np.where(sample_df["onpromotion"] > 0, True, False),
    year=sample_df["date"].str.slice(0, 4).astype("int"),
).query("family == 'DAIRY'")
```

	date	family	id	onpromotion	sales	store_nbr	tax_amount
2782266	2017-04-15	BEVERAGES	2782266	24	5516.0	25	$275.8
2356175	2016-08-18	DAIRY	2356175	7	714.0	2	$35.7
1691390	2015-08-10	DAIRY	1691390	0	613.0	17	$30.65
1435443	2015-03-19	DELI	1435443	24	134.0	35	$6.7
2939747	2017-07-12	DAIRY	2939747	120	628.0	43	$31.4

*First, a column called **tax_amount**:*

- *Equal to **sales** * **0.05***
- *Rounded to **2** decimals*
- *Starting with **"$"***

PRO TIP: COLUMN CREATION WITH ASSIGN

How does this code work?

```
sample_df.assign(
    tax_amount=(sample_df["sales"] * 0.05).round(2).map(lambda x: f"${x}"),
 👉 on_promotion_flag=np.where(sample_df["onpromotion"] > 0, True, False),
    year=sample_df["date"].str.slice(0, 4).astype("int"),
).query("family == 'DAIRY'")
```

	date	family	id	onpromotion	sales	store_nbr	tax_amount	on_promotion_flag
2782266	2017-04-15	BEVERAGES	2782266	24	5516.0	25	$275.8	True
2356175	2016-08-18	DAIRY	2356175	7	714.0	2	$35.7	True
1691390	2015-08-10	DAIRY	1691390	0	613.0	17	$30.65	False
1435443	2015-03-19	DELI	1435443	24	134.0	35	$6.7	True
2939747	2017-07-12	DAIRY	2939747	120	628.0	43	$31.4	True

*Next, a column called **on_promotion_flag**:*

- *If **onpromotion > 0** return **True***
- *Otherwise, return **False***

PRO TIP: COLUMN CREATION WITH ASSIGN

How does this code work?

```
sample_df.assign(
    tax_amount=(sample_df["sales"] * 0.05).round(2).map(lambda x: f"${x}"),
    on_promotion_flag=np.where(sample_df["onpromotion"] > 0, True, False),
 👉 year=sample_df["date"].str.slice(0, 4).astype("int"),
).query("family == 'DAIRY'")
```

	date	family	id	onpromotion	sales	store_nbr	tax_amount	on_promotion_flag	year
2782266	2017-04-15	BEVERAGES	2782266	24	5516.0	25	$275.8	True	2017
2356175	2016-08-18	DAIRY	2356175	7	714.0	2	$35.7	True	2016
1691390	2015-08-10	DAIRY	1691390	0	613.0	17	$30.65	False	2015
1435443	2015-03-19	DELI	1435443	24	134.0	35	$6.7	True	2015
2939747	2017-07-12	DAIRY	2939747	120	628.0	43	$31.4	True	2017

*Next, a last column called **year**:*

- *Equal to the first 4 characters on the **date** column*
- *Converted to an integer*

PRO TIP: COLUMN CREATION WITH ASSIGN

How does this code work?

```
sample_df.assign(
    tax_amount=(sample_df["sales"] * 0.05).round(2).map(lambda x: f"${x}"),
    on_promotion_flag=np.where(sample_df["onpromotion"] > 0, True, False),
    year=sample_df["date"].str.slice(0, 4).astype("int"),
).query("family == 'DAIRY'")
```

	date	family	id	onpromotion	sales	store_nbr	tax_amount	on_promotion_flag	year
2356175	2016-08-18	DAIRY	2356175	7	714.0	2	$35.7	True	2016
1691390	2015-08-10	DAIRY	1691390	0	613.0	17	$30.65	False	2015
2939747	2017-07-12	DAIRY	2939747	120	628.0	43	$31.4	True	2017

Finally, return the DataFrame and filter it:

- *For rows where **family** is **"DAIRY"***

ASSIGNMENT: ASSIGN

NEW MESSAGE

July 23, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Cleaning Up The Code**

Hi there,

Time to clean up our workflow!

Drop the columns that have been created so far (keep only date, store_number, and transaction count), and recreate them using the assign method.

Then sum the seasonal bonus owed once again to make sure the numbers are correct.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Results Preview

```
transactions
```

	date	store_number	transaction_count	target_pct	met_target	bonus_payable	month	day_of_week	seasonal_bonus
1	2013-01-02	1	2111	0.8444	False	0	1	2	None
2	2013-01-02	2	2358	0.9432	False	0	1	2	None
3	2013-01-02	3	3487	1.3948	True	100	1	2	None
4	2013-01-02	4	1922	0.7688	False	0	1	2	None
5	2013-01-02	5	1903	0.7612	False	0	1	2	None

```
7409700
```

SOLUTION: ASSIGN

NEW MESSAGE

July 23, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Cleaning Up The Code**

Hi there,

Time to clean up our workflow!

Drop the columns that have been created so far (keep only date, store_number, and transaction count), and recreate them using the assign method.

Then sum the seasonal bonus owed once again to make sure the numbers are correct.

Thanks!

section03_DataFrames.ipynb

Reply Forward

Solution Code

```
transactions.assign(
    target_pct = transactions["transaction_count"] / 2500,
    met_target = (transactions["transaction_count"] / 2500) >= 1,
    bonus_payable = ((transactions["transaction_count"] / 2500) >= 1) * 100,
    month = transactions.date.dt.month,
    day_of_week = transactions.date.dt.dayofweek,
    seasonal_bonus = np.select(conditions, choices, default="None"),
)

transactions
```

	date	store_number	transaction_count	target_pct	met_target	bonus_payable	month	day_of_week	seasonal_bonus
1	2013-01-02	1	2111	0.8444	False	0	1	2	None
2	2013-01-02	2	2358	0.9432	False	0	1	2	None
3	2013-01-02	3	3487	1.3948	True	100	1	2	None
4	2013-01-02	4	1922	0.7688	False	0	1	2	None
5	2013-01-02	5	1903	0.7612	False	0	1	2	None

```
transactions.loc[transactions['seasonal_bonus'] != 'None'].count().mul(100).sum()
```

```
7409700
```

REVIEW: PANDAS DATA TYPES

Pandas data types mostly expand on their base Python and NumPy equivalents

Numeric:

Data Type	Description	Bit Sizes
bool	Boolean True/False	8
int64 *(default)*	Whole numbers	8, 16, 32, 64
float64 *(default)*	Decimal numbers	8, 16, 32, 64
boolean	Nullable Boolean True/False	8
Int64 *(default)*	Nullable whole numbers	8, 16, 32, 64
Float64 *(default)*	Nullable decimal numbers	32, 64

**Gray* = *NumPy data type*
**Yellow* = *Pandas data type*

Object / Text:

Data Type	Description
object	Any Python object
string	Only contains strings or text
category	Maps categorical data to a numeric array for efficiency

Time Series:

Data Type	Description
datetime	A single moment in time *(January 4, 2015, 2:00:00 PM)*
timedelta	The duration between two dates or times *(10 days, 3 seconds, etc.)*
period	A span of time *(a day, a week, etc.)*

THE CATEGORICAL DATA TYPE

The Pandas **categorical data type** stores text data with repeated values efficiently

- Python maps each unique category to an integer to save space
- As a rule of thumb, only consider this data type when unique categories < number of rows / 2

```
sample_df.astype({"family": "category"})
```

	id	date	store_nbr	family	sales	onpromotion
2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24
2356175	2356175	2016-08-18	2	DAIRY	714.0	7
1691390	1691390	2015-08-10	17	DAIRY	613.0	0
1435443	1435443	2015-03-19	35	DELI	134.0	24
2939747	2939747	2017-07-12	43	DAIRY	628.0	120

family
0
1
1
2
1

These are now stored as integers in the backend

The categorical data type has some quirks during some data manipulation operations that will **force it back into an object data type**, but it's not something we'll cover in depth in this course

TYPE CONVERSION

Like Series, you can **convert data types** in a DataFrame by using the .astype() method and specifying the desired data type *(if compatible)*

```
sample_df["sales_int"] = sample_df["sales"].astype("int")

sample_df
```

	id	date	store_nbr	family	sales	onpromotion	sales_int
2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24	5516
2356175	2356175	2016-08-18	2	DAIRY	714.0	7	714
1691390	1691390	2015-08-10	17	DAIRY	613.0	0	613
1435443	1435443	2015-03-19	35	DELI	134.0	24	134
2939747	2939747	2017-07-12	43	DAIRY	628.0	120	628

This creates a new 'sales_int' column by converting 'sales' to integers

```
sample_df = sample_df.astype({"date": "Datetime64", "onpromotion": "float"})

sample_df
```

	id	date	store_nbr	family	sales	onpromotion	sales_int
2782266	2782266	2017-04-15	25	BEVERAGES	5516.0	24.0	5516
2356175	2356175	2016-08-18	2	DAIRY	714.0	7.0	714
1691390	1691390	2015-08-10	17	DAIRY	613.0	0.0	613
1435443	1435443	2015-03-19	35	DELI	134.0	24.0	134
2939747	2939747	2017-07-12	43	DAIRY	628.0	120.0	628

You can use the .astype() method on the entire DataFrame and pass a dictionary with the columns as keys and the desired data type as values

TYPE CONVERSION

Like Series, you can **convert data types** in a DataFrame by using the .astype() method and specifying the desired data type *(if compatible)*

```
product_df
```

	product_id	product	price	Vegan?
0	1	Dairy	$2.56	Non-Vegan
1	2	Dairy	$2.56	Non-Vegan
2	3	Dairy	$4.55	Non-Vegan
3	4	Vegetables	$2.74	Vegan
4	5	Fruits	$5.44	Vegan

```
product_df.astype({"price": "float"})
```

```
ValueError: could not convert string to float: '$2.56'
```

.astype() will return a ValueError if the data type is incompatible

```
product_df.assign(price=product_df["price"].str.strip("$").astype("float"))
```

But applying cleaning steps to the data can make it work

	product_id	product	price	Vegan?
0	1	Dairy	2.56	Non-Vegan
1	2	Dairy	2.56	Non-Vegan
2	3	Dairy	4.55	Non-Vegan
3	4	Vegetables	2.74	Vegan
4	5	Fruits	5.44	Vegan

PRO TIP: MEMORY OPTIMIZATION

DataFrames are stored entirely in memory, so **memory optimization** is key in working with large datasets in Pandas

Memory Optimization Best Practices *(in order)*:

1. Drop unnecessary columns *(when possible, avoid reading them in at all)*
2. Convert object types to numeric or datetime datatypes where possible
3. Downcast numeric data to the smallest appropriate bit size
4. Use the categorical datatype for columns where the number of unique values < rows / 2

```
class_data = pd.read_csv("class_data.csv")

class_data.memory_usage(deep=True)
```

```
Index                128
id                    40
start_date           335
title                331
class_level           40
price                299
students_enrolled    294
dtype: int64
```

*The **.memory_usage()** method returns the memory used by each column in a DataFrame (in bytes), and **deep=True** provides more accurate results*

```
class_data.memory_usage(deep=True).sum()
```

```
1467
```

← *The total memory usage is 1,472 bytes*

PRO TIP: A good rule of thumb is to have around 5-10 times the RAM as the size of your DataFrame

STEP 1: DROP COLUMNS

Dropping unnecessary columns is an easy way to free up significant space

- You may not know which columns are important when reading in a dataset for the first time
- If you do, you can limit the columns you read in to begin with *(more on that later!)*

```
class_data
```

	id	start_date	title	class_level	price	students_enrolled
0	0	2022-01-11	Algebra I	8.0	$299	102
1	1	2022-02-22	Algebra 2	9.0	-	43
2	2	2022-03-03	Geometry	NaN	$399	16
3	3	2022-04-04	Trigonometry	11.0	$599	8
4	4	2022-05-05	Calculus	12.0	-	-

Note that the id column is identical to the index

```
class_data.drop("id", axis=1, inplace=True)

class_data.memory_usage(deep=True).sum()
```

```
1427
```

By dropping the 'id' column, around 40 bytes were freed (~4% of memory use)

Note that the memory usage went down from 1,470 bytes

STEP 2: CONVERTING OBJECT DATA TYPES

Try to **convert object data types** to numeric or datetime whenever possible

```
class_data
```

	start_date	title	class_level	price	students_enrolled
0	2022-01-11	Algebra I	8.0	$299	102
1	2022-02-22	Algebra 2	9.0	-	43
2	2022-03-03	Geometry	NaN	$399	16
3	2022-04-04	Trigonometry	11.0	$599	8
4	2022-05-05	Calculus	12.0	-	-

Note that the missing values in 'price' and 'students_enrolled' are not NaN values

*Use **memory_usage="deep"** with the .info() method to get total memory usage along with the column data types*

```
class_data.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   start_date         5 non-null      object
 1   title              5 non-null      object
 2   class_level        4 non-null      float64
 3   price              5 non-null      object
 4   students_enrolled  5 non-null      object
dtypes: float64(1), object(4)
memory usage: 1.4 KB
```

Text data, usually including dates, is read in as an object by default

Numeric data is usually read in as 64-bit (like 'class_level') but errors in the data can cause it to be read in as an object

STEP 2: CONVERTING OBJECT DATA TYPES

Try to **convert object data types** to numeric or datetime whenever possible

```
class_data.memory_usage(deep=True)
```

```
Index                128
start_date           335
title                331
class_level           40
price                299
students_enrolled    294
dtype: int64
```

```
class_data = class_data.astype({"start_date": "Datetime64", "title": "string"})

class_data.memory_usage(deep=True)
```

```
Index                128
start_date            40
title                331
class_level           40
price                299
students_enrolled    294
dtype: int64
```

Note that by converting 'start_date' to a datetime data type, around 295 bytes were freed (~20% of memory use)

On the other hand, nothing changed by converting 'title' to a string data type

STEP 2: CONVERTING OBJECT DATA TYPES

Try to **convert object data types** to numeric or datetime whenever possible

```
class_data
```

	start_date	title	class_level	price	students_enrolled
0	2022-01-11	Algebra I	8.0	$299	102
1	2022-02-22	Algebra 2	9.0	-	43
2	2022-03-03	Geometry	NaN	$399	16
3	2022-04-04	Trigonometry	11.0	$599	8
4	2022-05-05	Calculus	12.0	-	-

```
class_data["students_enrolled"] = class_data["students_enrolled"].replace("-",np.nan).astype("float")
class_data["price"] = class_data["price"].replace("-",np.nan).str.strip("$").astype("float")

class_data.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 5 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   start_date          5 non-null      datetime64[ns]
 1   title               5 non-null      string
 2   class_level         4 non-null      float64
 3   price               3 non-null      float64
 4   students_enrolled   4 non-null      float64
dtypes: datetime64[ns](1), float64(3), string(1)
memory usage: 619.0 bytes
```

This is now only 619 bytes!

Note that additional methods were chained to convert 'price' and 'students_enrolled' to floats:

- *"-" was replaced by NaN values on both*
- *"$" was stripped on 'price'*

STEP 3: DOWNCAST NUMERIC DATA

Integers and floats are cast as 64-bit by default to handle any possible value, but you can **downcast numeric data** to a smaller bit size to save space if possible

- 8-bits = -128 to 127
- 16-bits = -32,768 to 32,767
- 32-bits = -2,147,483,648 to 2,147,483,647
- 64-bits = -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

```
class_data
```

	start_date	title	class_level	price	students_enrolled
0	2022-01-11	Algebra I	8.0	299.0	102.0
1	2022-02-22	Algebra 2	9.0	NaN	43.0
2	2022-03-03	Geometry	NaN	399.0	16.0
3	2022-04-04	Trigonometry	11.0	599.0	8.0
4	2022-05-05	Calculus	12.0	NaN	NaN

Based on the ranges above, these columns can be downcast as follows:

- *'class_level' to Int8*
- *'price' to Int16*
- *'students_enrolled' to Int16*

STEP 3: DOWNCAST NUMERIC DATA

Integers and floats are cast as 64-bit by default to handle any possible value, but you can **downcast numeric data** to a smaller bit size to save space if possible

```
class_data = class_data.astype(
    {
        "class_level": "Int8",
        "price": "Int16",
        "students_enrolled": "Int16"
    }
)

class_data.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   start_date         5 non-null      datetime64[ns]
 1   title              5 non-null      string
 2   class_level        4 non-null      Int8
 3   price              3 non-null      Int16
 4   students_enrolled  4 non-null      Int16
dtypes: Int16(2), Int8(1), datetime64[ns](1), string(1)
memory usage: 539.0 bytes
```

This is now only 539 bytes!

```
class_data
```

	start_date	title	class_level	price	students_enrolled
0	2022-01-11	Algebra I	8	299	102
1	2022-02-22	Algebra 2	9	<NA>	43
2	2022-03-03	Geometry	<NA>	399	16
3	2022-04-04	Trigonometry	11	599	8
4	2022-05-05	Calculus	12	<NA>	<NA>

Note that, since these are Pandas' Nullable data types, the NumPy NaN values are now Pandas NA values

STEP 4: USING CATEGORICAL DATA TYPES

Use the categorical data type if you have string columns where the number of unique values is less than half of the total number of rows

```
class_data["title"] = class_data["title"].astype("category")

class_data.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 5 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   start_date        5 non-null      datetime64[ns]
 1   title             5 non-null      category
 2   class_level       4 non-null      Int8
 3   price             3 non-null      Int16
 4   students_enrolled  4 non-null      Int16
dtypes: Int16(2), Int8(1), category(1), datetime64[ns](1)
memory usage: 716.0 bytes
```

This went up to 716 bytes!

```
class_data
```

	start_date	title	class_level	price	students_enrolled
0	2022-01-11	Algebra I	8	299	102
1	2022-02-22	Algebra 2	9	<NA>	43
2	2022-03-03	Geometry	<NA>	399	16
3	2022-04-04	Trigonometry	11	599	8
4	2022-05-05	Calculus	12	<NA>	<NA>

In this case, there are five unique categories in five rows, so this just added overhead

(the demo will show the memory reduction)

RECAP: MEMORY OPTIMIZATION

- DataFrame Basics
- Exploring DataFrames
- Accessing & Dropping Data
- Blank & Duplicate Values
- Sorting & Filtering
- Modifying Columns
- Pandas Data Types
- Memory Optimization

BEFORE

```
class_data
```

	id	start_date	title	class_level	price	students_enrolled
0	0	2022-01-11	Algebra I	8.0	$299	102
1	1	2022-02-22	Algebra 2	9.0	-	43
2	2	2022-03-03	Geometry	NaN	$399	16
3	3	2022-04-04	Trigonometry	11.0	$599	8
4	4	2022-05-05	Calculus	12.0	-	-

```
class_data.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 6 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   id                 5 non-null      int64
 1   start_date         5 non-null      object
 2   title              5 non-null      object
 3   class_level        4 non-null      float64
 4   price              5 non-null      object
 5   students_enrolled  5 non-null      object
dtypes: float64(1), int64(1), object(4)
memory usage: 1.4 KB
```

AFTER

```
class_data
```

	start_date	title	class_level	price	students_enrolled
0	2022-01-11	Algebra I	8	299	102
1	2022-02-22	Algebra 2	9	<NA>	43
2	2022-03-03	Geometry	<NA>	399	16
3	2022-04-04	Trigonometry	11	599	8
4	2022-05-05	Calculus	12	<NA>	<NA>

```
class_data.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5 entries, 0 to 4
Data columns (total 5 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   start_date         5 non-null      datetime64[ns]
 1   title              5 non-null      string
 2   class_level        4 non-null      Int8
 3   price              3 non-null      Int16
 4   students_enrolled  4 non-null      Int16
dtypes: Int16(2), Int8(1), datetime64[ns](1), string(1)
memory usage: 539.0 bytes
```

63% less memory!

A NOTE ON EFFICIENCY

"Premature optimization is the root of all evil" - *Don Knuth*

- Data analysis is an **iterative process**
  - *It's normal to not know the best or most efficient version of a dataset from the beginning*
  - *Once you understand the data and have an analysis path, then you can optimize*
- Efficiency's importance **depends on the use case**
  - *If all you need is a quick analysis, fully optimizing your code will only waste time*
  - *If you're building a pipeline that will run frequently, efficiency & optimization are critical*
- Build **efficient habits** & **workflows**
  - *As you work more with Python and Pandas, take time to review your code and note areas for improvement - you'll be able to incorporate better practices next time!*

ASSIGNMENT: MEMORY OPTIMIZATION

NEW MESSAGE

July 25, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Saving Disc Space**

Hi,

I'm going to take my work home this weekend, and I'll be staying at my grandparents' house.

They have a very, very old computer, so do you think you can reduce the memory usage to below 5MB, without losing any of the information?

Thanks!

section03_DataFrames.ipynb

Reply Forward

Results Preview

```
transactions.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 83487 entries, 1 to 83487
Data columns (total 9 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   [illegible]
 1   [illegible]
 2   [illegible]
 3   [illegible]
 4   [illegible]
 5   [illegible]
 6   [illegible]
 7   [illegible]
 8   [illegible]
[illegible]
memory usage: 3.3 MB
```

SOLUTION: MEMORY OPTIMIZATION

NEW MESSAGE

July 25, 2022

From: **Chandler Capital** *(Accountant)*

Subject: **Saving Disc Space**

Hi,

I'm going to take my work home this weekend, and I'll be staying at my grandparents' house.

They have a very, very old computer, so do you think you can reduce the memory usage to below 5MB, without losing any of the information?

Thanks!

section03_DataFrames.ipynb

Reply Forward

Solution Code

```
transactions.info(memory_usage="deep")
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 83487 entries, 1 to 83487
Data columns (total 9 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   date               83487 non-null  datetime64[ns]
 1   store_number       83487 non-null  Int16
 2   transaction_count  83487 non-null  Int32
 3   target_pct         83487 non-null  float64
 4   met_target         83487 non-null  bool
 5   bonus_payable      83487 non-null  Int16
 6   month              83487 non-null  Int8
 7   day_of_week        83487 non-null  Int8
 8   seasonal_bonus     83487 non-null  category
dtypes: Int16(2), Int32(1), Int8(2), bool(1), category(1), datetime64[ns](1), float64(1)
memory usage: 3.3 MB
```

KEY TAKEAWAYS

Pandas DataFrames are **data tables** with rows & columns

- *They are technically collections of Pandas Series that share an index, and are the primary data structure that data analysts work with in Python*

Use **exploration methods** to quickly understand the data in a DataFrame

- *The head, tail, describe, and info methods let you get a glimpse of the data and its characteristics to identify the cleaning steps needed*

You can easily **filter**, **sort**, and **modify** DataFrames with methods & functions

- *DataFrames rows & columns can be sorted by index or values, and filtered using multiple conditions*
- *Columns can be created with arithmetic or complex logic, and multiple columns can be created with .assign()*

Memory optimization is critical in working with large datasets in Pandas

- *Once you understand the data, dropping unnecessary columns, converting object data types, downcasting numerical data types, and using the categorical data types when possible will help save significant memory*

AGGREGATING & RESHAPING

AGGREGATING & RESHAPING

In this section we'll cover **aggregating & reshaping** DataFrames, including grouping columns, performing aggregation calculations, and pivoting & unpivoting data

TOPICS WE'LL COVER:

- Grouping Columns
- Multi-Index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

GOALS FOR THIS SECTION:

- Group DataFrames by one or more columns and calculate aggregate statistics by group
- Learn to access multi-index DataFrames and reset them to return to a single index
- Create Excel-style PivotTables to summarize data
- Melt "wide" tables of data into a "long" tabular form

AGGREGATING DATAFRAMES

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

You can **aggregate a DataFrame** column by using aggregation methods *(like Series!)*

```
retail.loc[:, ['sales', 'onpromotion']].sample(100).sum().round(2)
```

```
sales          33729.57
onpromotion      240.00
dtype: float64
```

```
retail.loc[:, ['sales', 'onpromotion']].sample(100).mean().round(2)
```

```
sales          299.68
onpromotion      2.36
dtype: float64
```

But what if you want multiple aggregate statistics, or summarized statistics by groups?

GROUPING DATAFRAMES

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

Grouping a DataFrame allows you to aggregate the data at a different level

- For example, transform daily data into monthly, roll up transaction level data by store, etc.

Original DataFrame

	id	date	store_nbr	family	sales	onpromotion
0	0	2013-01-01	1	AUTOMOTIVE	0.0	0
1	1	2013-01-01	1	BABY CARE	0.0	0
2	2	2013-01-01	1	BEAUTY	0.0	0
3	3	2013-01-01	1	BEVERAGES	0.0	0
4	4	2013-01-01	1	BOOKS	0.0	0

Raw sales data by transaction

Grouped by Family

	sales
family	
AUTOMOTIVE	264.0
BABY CARE	0.0
BEAUTY	121.0
BEVERAGES	29817.0
BOOKS	0.0

Total sales by family

GROUPING DATAFRAMES

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

To group data, use the **.groupby()** method and specify a column to group by

- The grouped column becomes the index by default

Just specify the columns to group by

```
small_retail.groupby('family')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f9fe9e30910>
```

This returns a "groupby" object

```
small_retail.groupby('family')['sales'].sum().head()
```

```
family
AUTOMOTIVE      264.0
BABY CARE         0.0
BEAUTY          121.0
BEVERAGES     29817.0
BOOKS             0.0
Name: sales, dtype: float64
```

Using single brackets returns a Series

To return the groups created, you need to calculate aggregate statistics:

1. *Specify the column for the calculations*
2. *Apply an aggregation method*

GROUPING DATAFRAMES

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

To group data, use the **.groupby()** method and specify a column to group by

- The grouped column becomes the index by default

Just specify the columns to group by

```
small_retail.groupby('family')
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x7f9fe9e30910>
```

This returns a "groupby" object

```
small_retail.groupby('family')[['sales']].sum().head()
```

Using double brackets returns a DataFrame

family	sales
AUTOMOTIVE	264.0
BABY CARE	0.0
BEAUTY	121.0
BEVERAGES	29817.0
BOOKS	0.0

To return the groups created, you need to calculate aggregate statistics:

1. *Specify the column for the calculations*
2. *Apply an aggregation method*

ASSIGNMENT: GROUPBY

NEW MESSAGE

August 3, 2022

From: **Phoebe Product** *(Merchandising)*

Subject: **Top Stores by Transactions**

Hi there, it's Phoebe!

I want to create some custom displays for our busiest stores.

Can you return a table containing the top 10 stores by total transactions in the data?

Make sure they're sorted from highest to lowest.

Thanks!

P.S. Let me know if you want to hear my music!

section04_aggregations.ipynb

Reply Forward

Results Preview

store_nbr	transactions
44	7273093
47	6535810
45	6201115
46	5990113
3	5366350
48	5107785
8	4637971
49	4574103
50	4384444
11	3972488

SOLUTION: GROUPBY

1 **NEW MESSAGE**

August 3, 2022

From: **Phoebe Product** *(Merchandising)*

Subject: **Top Stores by Transactions**

Hi there, it's Phoebe!

I want to create some custom displays for our busiest stores.

Can you return a table containing the top 10 stores by total transactions in the data?

Make sure they're sorted from highest to lowest.

Thanks!

P.S. Let me know if you want to hear my music!

section04_aggregations.ipynb

Reply Forward

Solution Code

```
transactions.groupby(["store_nbr"])[["transactions"]].sum().sort_values(
    by="transactions", ascending=False
).iloc[:10]
```

	transactions
store_nbr	
44	7273093
47	6535810
45	6201115
46	5990113
3	5366350
48	5107785
8	4637971
49	4574103
50	4384444
11	3972488

GROUPING BY MULTIPLE COLUMNS

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

You can **group by multiple columns** by passing the list of columns into .groupby()

- This creates a multi-index object with an index for each column the data was grouped by

```
small_retail.groupby(['family', 'store_nbr'])[['sales']].sum()
```

		sales
family	store_nbr	
AUTOMOTIVE	11	8.000000
	12	9.000000
	13	7.000000
	16	5.000000
	19	5.000000
...	...	...
SEAFOOD	47	48.239998
	49	78.718000
	50	18.583000
	53	2.000000
	8	57.757000

780 rows × 1 columns

This is a multi-index DataFrame

This returns the sum of sales for each combination of 'family' and 'store_nbr'

GROUPING BY MULTIPLE COLUMNS

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

You can **group by multiple columns** by passing the list of columns into .groupby()

- This creates a multi-index object with an index for each column the data was grouped by
- Specify **as_index=False** to prevent the grouped columns from becoming indices

```
sales_sums = small_retail.groupby(['family', 'store_nbr'],
                                  as_index=False)[['sales']].sum()

sales_sums
```

	family	store_nbr	sales
0	AUTOMOTIVE	11	8.000000
1	AUTOMOTIVE	12	9.000000
2	AUTOMOTIVE	13	7.000000
3	AUTOMOTIVE	16	5.000000
4	AUTOMOTIVE	19	5.000000
...	...	...	...
775	SEAFOOD	47	48.239998
776	SEAFOOD	49	78.718000
777	SEAFOOD	50	18.583000
778	SEAFOOD	53	2.000000
779	SEAFOOD	8	57.757000

780 rows × 3 columns

This still returns the sum of sales for each combination of 'family' and 'store_nbr', but keeps a numeric index

ASSIGNMENT: GROUPBY MULTIPLE COLUMNS

NEW MESSAGE

August 4, 2022

From: **Phoebe Product** *(Merchandising)*

Subject: **Transactions by Store and Month**

Hi there, it's Phoebe again!

Taking this analysis a layer deeper...

Can you get me the total transactions by store and month?

Sort the table from first month to last, then by highest transactions to lowest within each month.

I'll likely analyze this data further in a dashboard software, but this will help me set up special seasonal displays.

Thanks!

section04_aggregations.ipynb

Reply Forward

Results Preview

		transactions
store_nbr	**month**	
44	**1**	628438
47	**1**	568824
45	**1**	538370
46	**1**	522763
3	**1**	463260
...	**...**	...
32	**12**	86167
21	**12**	84128
42	**12**	76741
29	**12**	76627
22	**12**	50650

641 rows × 1 columns

SOLUTION: GROUPBY MULTIPLE COLUMNS

NEW MESSAGE

August 4, 2022

From: **Phoebe Product** *(Merchandising)*

Subject: **Transactions by Store and Month**

Hi there, it's Phoebe again!

Taking this analysis a layer deeper...

Can you get me the total transactions by store and month?

Sort the table from first month to last, then by highest transactions to lowest within each month.

I'll likely analyze this data further in a dashboard software, but this will help me set up special seasonal displays.

Thanks!

section04_aggregations.ipynb

Reply Forward

Solution Code

```
(transactions.groupby(["store_nbr", "month"])[["transactions"]]
 .sum()
 .sort_values(by=["month", "transactions"], ascending=[True, False]))
```

		transactions
store_nbr	**month**	
44	**1**	628438
47	**1**	568824
45	**1**	538370
46	**1**	522763
3	**1**	463260
...	**...**	...
32	**12**	86167
21	**12**	84128
42	**12**	76741
29	**12**	76627
22	**12**	50650

641 rows × 1 columns

MULTI-INDEX DATAFRAMES

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

Multi-index DataFrames are generally created through aggregation operations

- They are stored as a list of tuples, with an item for each layer of the index

```
sales_sums
```

		sales
family	store_nbr	
AUTOMOTIVE	11	8.000000
	12	9.000000
	13	7.000000
	16	5.000000
	19	5.000000
...	...	...
SEAFOOD	47	48.239998
	49	78.718000
	50	18.583000
	53	2.000000
	8	57.757000

780 rows × 1 columns

```
sales_sums.index
```

```
MultiIndex([('AUTOMOTIVE', '11'),
            ('AUTOMOTIVE', '12'),
            ('AUTOMOTIVE', '13'),
            ('AUTOMOTIVE', '16'),
            ('AUTOMOTIVE', '19'),
            ('AUTOMOTIVE', '21'),
            ('AUTOMOTIVE', '22'),
            ('AUTOMOTIVE', '26'),
            ('AUTOMOTIVE', '27'),
            ('AUTOMOTIVE', '28'),
            ...
            (   'SEAFOOD', '34'),
            (   'SEAFOOD', '35'),
            (   'SEAFOOD', '39'),
            (   'SEAFOOD', '43'),
            (   'SEAFOOD', '45'),
            (   'SEAFOOD', '47'),
            (   'SEAFOOD', '49'),
            (   'SEAFOOD', '50'),
            (   'SEAFOOD', '53'),
            (   'SEAFOOD',  '8')],
           names=['family', 'store_nbr'], length=780)
```

ACCESSING MULTI-INDEX DATAFRAMES

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

The .loc[] accessor lets you **access multi-index DataFrames** in different ways:

1. Access rows via the **outer index** only

```
sales_sums
```

		sales
family	store_nbr	
AUTOMOTIVE	11	8.000000
	12	9.000000
	13	7.000000
	16	5.000000
	19	5.000000
...	...	...
SEAFOOD	47	48.239998
	49	78.718000
	50	18.583000
	53	2.000000
	8	57.757000

780 rows × 1 columns

```
sales_sums.loc['AUTOMOTIVE'].head()
```

	sales
store_nbr	
11	8.0
12	9.0
13	7.0
16	5.0
19	5.0

All rows for "AUTOMOTIVE" (note that 'family' is dropped)

```
sales_sums.loc["AUTOMOTIVE":"BEAUTY"]
```

		sales
family	store_nbr	
AUTOMOTIVE	11	8.0
	12	9.0
	13	7.0
	16	5.0
	19	5.0
...	...	...
BEAUTY	45	10.0
	47	34.0
	48	8.0
	52	0.0
	9	7.0

72 rows × 1 columns

All rows from "AUTOMOTIVE" to "BEAUTY" (inclusive)

ACCESSING MULTI-INDEX DATAFRAMES

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

The .loc[] accessor lets you **access multi-index DataFrames** in different ways:

2. Access rows via the **outer & inner indices**

```
sales_sums
```

		sales
family	store_nbr	
AUTOMOTIVE	11	8.000000
	12	9.000000
	13	7.000000
	16	5.000000
	19	5.000000
...	...	...
SEAFOOD	47	48.239998
	49	78.718000
	50	18.583000
	53	2.000000
	8	57.757000

780 rows × 1 columns

```
sales_sums.loc[('AUTOMOTIVE', '12'), :]
```

```
sales     9.0
Name: (AUTOMOTIVE, 12), dtype: float64
```

All rows for "AUTOMOTIVE" and "12"
(note that 'family' and 'store_nbr' are dropped)

```
sales_sums.loc[("AUTOMOTIVE", "11"):("BEAUTY", "11"), :]
```

family	store_nbr	
AUTOMOTIVE	11	8.0
	12	9.0
	13	7.0

All rows from "AUTOMOTIVE" and "11" to "BEAUTY" and "11" (inclusive)

....All rows in Automotive and Baby Care

BEAUTY	1	7.0
	10	1.0
	11	11.0

MODIFYING MULTI-INDEX DATAFRAMES

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

There are several ways to **modify multi-index DataFrames**:

Reset the index

Moves the index levels back to DataFrame columns

```
sales_sums.reset_index()
```

	family	store_nbr	sales
0	AUTOMOTIVE	11	8.000000
1	AUTOMOTIVE	12	9.000000
2	AUTOMOTIVE	13	7.000000
3	AUTOMOTIVE	16	5.000000
4	AUTOMOTIVE	19	5.000000
...	...	...	...

Swap the index level

Changes the hierarchy for the index levels

```
sales_sums.swaplevel()
```

		sales
store_nbr	**family**	
11	**AUTOMOTIVE**	8.000000
12	**AUTOMOTIVE**	9.000000
13	**AUTOMOTIVE**	7.000000
16	**AUTOMOTIVE**	5.000000
19	**AUTOMOTIVE**	5.000000
...	**...**	...

Drop an index level

Drops an index level from the DataFrame entirely

```
sales_sums.droplevel('family')
```

	sales
store_nbr	
11	8.000000
12	9.000000
13	7.000000
16	5.000000
19	5.000000
...	...

Be careful! You **may lose important information**

PRO TIP: In most cases it's best to reset the index and avoid multi-index DataFrames – they're not very intuitive!

ASSIGNMENT: MULTI-INDEX DATAFRAMES

NEW MESSAGE

August 5, 2022

From: **Ross Retail** *(Head of Analytics)*

Subject: **Multi-Index Help!**

Hey, I've looked at your work – folks are really impressed.

Can you help me access rows and columns with multiple indices? I've been struggling with multi-index DataFrames.

Also, I might prefer to just return to an integer-based index and drop the second column index that got created when I performed multiple aggregations.

Do you know how to do this?

I attached the notebook and added a few questions in there.

section04_aggregations.ipynb

Reply Forward

Results Preview

```
grouped.head()
```

		transactions	
		sum	mean
store_nbr	month		
44	1	628438	4246.202703
47	1	568824	3843.405405
45	1	538370	3637.635135
46	1	522763	3532.182432
3	1	463260	3151.428571

```
[redacted]
```

			sum	mean
0	44	1	628438	4246.202703
1	47	1	568824	3843.405405
2	45	1	538370	3637.635135
3	46	1	522763	3532.182432
4	3	1	463260	3151.428571

SOLUTION: MULTI-INDEX DATAFRAMES

NEW MESSAGE

August 5, 2022

From: **Ross Retail** *(Head of Analytics)*

Subject: **Multi-Index Help!**

Hey, I've looked at your work – folks are really impressed.

Can you help me access rows and columns with multiple indices? I've been struggling with multi-index DataFrames.

Also, I might prefer to just return to an integer-based index and drop the second column index that got created when I performed multiple aggregations.

Do you know how to do this?

I attached the notebook and added a few questions in there.

section04_aggregations.ipynb

Reply Forward

Solution Code

```
grouped.loc[(3, 1)]
```

```
transactions  sum     463260.000000
              mean      3151.428571
Name: (3, 1), dtype: float64
```

```
grouped.iloc[4]
```

```
transactions  sum     463260.000000
              mean      3151.428571
Name: (3, 1), dtype: float64
```

```
grouped.loc[:, ("transactions", "mean")].head(1)
```

```
store_nbr  month
44         1        4246.202703
Name: (transactions, mean), dtype: float64
```

```
grouped.iloc[:, 1].head(1)
```

```
store_nbr  month
44         1        4246.202703
Name: (transactions, mean), dtype: float64
```

```
grouped.reset_index().droplevel(0, axis=1).head()
```

			sum	mean
0	44	1	628438	4246.202703
1	47	1	568824	3843.405405
2	45	1	538370	3637.635135
3	46	1	522763	3532.182432
4	3	1	463260	3151.428571

THE AGG METHOD

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

The .agg() method lets you perform multiple aggregations on a "groupby" object

	date	store_nbr	family	sales	onpromotion
2862475	2017-05-30	25	LIQUOR,WINE,BEER	92.0	2
940501	2014-06-13	48	BABY CARE	0.0	0
1457967	2015-04-01	17	PLAYERS AND ELECTRONICS	0.0	0
1903307	2015-12-07	12	SEAFOOD	3.0	0
196280	2013-04-21	16	PREPARED FOODS	66.0	0

```
small_retail.groupby(['store_nbr', 'family']).agg('sum')
```

		sales	onpromotion
store_nbr	**family**		
1	**BEAUTY**	7.000	1
	BREAD/BAKERY	822.484	9
	CELEBRATION	2.000	0
	CLEANING	682.000	1
	DAIRY	585.000	3
...	**...**	...	...
9	**LADIESWEAR**	5.000	0
	LAWN AND GARDEN	8.000	0
	LINGERIE	15.000	0
	LIQUOR,WINE,BEER	69.000	0
	PET SUPPLIES	0.000	0

780 rows × 2 columns

The .agg() method will perform the aggregation on all compatible columns, in this case 'sales' and 'onpromotion', which are numeric

MULTIPLE AGGREGATIONS

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

You can perform **multiple aggregations** by passing a list of aggregation functions

```
small_retail.groupby(['family', 'store_nbr']).agg(['sum', 'mean'])
```

		sales		onpromotion	
		sum	mean	sum	mean
family	store_nbr				
AUTOMOTIVE	11	8.000000	8.000000	0	0.0
	12	9.000000	9.000000	0	0.0
	13	7.000000	3.500000	0	0.0
	16	5.000000	5.000000	0	0.0
	19	5.000000	5.000000	0	0.0
...	...	...	...	...	...
SEAFOOD	47	48.239998	48.239998	0	0.0
	49	78.718000	78.718000	0	0.0
	50	18.583000	18.583000	5	5.0
	53	2.000000	1.000000	0	0.0
	8	57.757000	57.757000	0	0.0

780 rows × 4 columns

This creates two levels in the column index, one for the original column names, and another for the aggregations performed

MULTIPLE AGGREGATIONS

You can perform **specific aggregations by column** by passing a dictionary with column names as keys, and lists of aggregation functions as values

```
(small_retail
 .groupby(['family', 'store_nbr'])
 .agg({'sales': ['sum', 'mean'],
       'onpromotion':['min', 'max']})
)
```

The calculates the sum and mean for 'sales' and the min and max for 'onpromotion'

		sales		onpromotion	
		sum	mean	min	max
family	store_nbr				
AUTOMOTIVE	11	8.000000	8.000000	0	0
	12	9.000000	9.000000	0	0
	13	7.000000	3.500000	0	0
	16	5.000000	5.000000	0	0
	19	5.000000	5.000000	0	0
...	...	...	...	...	...
SEAFOOD	47	48.239998	48.239998	0	0
	49	78.718000	78.718000	0	0
	50	18.583000	18.583000	5	5
	53	2.000000	1.000000	0	0
	8	57.757000	57.757000	0	0

780 rows × 4 columns

NAMED AGGREGATIONS

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

You can **name aggregated columns** upon creation to avoid multi-index columns

Specify the new column name and assign it a tuple with the column you want to aggregate and the aggregation to perform

```
(small_retail
 .groupby(['family', 'store_nbr'])
 .agg(sales_sum=('sales', 'sum'),
      sales_avg=('sales', 'mean'),
      on_promotion_max=('onpromotion', 'max'))
)
```

A single column index!

family	store_nbr	sales_sum	sales_avg	on_promotion_max
AUTOMOTIVE	11	8.000000	8.000000	0
	12	9.000000	9.000000	0
	13	7.000000	3.500000	0
	16	5.000000	5.000000	0
	19	5.000000	5.000000	0
...	...	...	...	...
SEAFOOD	47	48.239998	48.239998	0
	49	78.718000	78.718000	0
	50	18.583000	18.583000	5
	53	2.000000	1.000000	0
	8	57.757000	57.757000	0

780 rows × 3 columns

ASSIGNMENT: THE AGG METHOD

NEW MESSAGE

August 6, 2022

From: **Chandler Capital** *(Accounting)*

Subject: **Bonus Rate and Bonus Payable**

Hey again,

I'm performing some further analysis on our bonuses.

Can you create a table that has the average number of days each store hit the target?

Calculate the total bonuses payable to each store and sort the DataFrame from highest bonus owed to lowest.

Then do the same for day of week and month.

Thanks!

section04_aggregations.ipynb

Reply Forward

Results Preview

By Store:

store_nbr	met_target	bonus_payable
47	0.999404	167600
44	0.998807	167500
45	0.997615	167300
3	0.998210	167300
46	0.989267	165900

By Month:

month	met_target	bonus_payable
12	0.255640	154100
5	0.170792	131800
3	0.169461	130400
4	0.174469	129700
7	0.162486	126300

By Weekday:

day_of_week	met_target	bonus_payable
5	0.222204	266400
6	0.204001	241700
4	0.179007	213000
0	0.160214	191600
2	0.160572	191000

***NOTE**: Only the top 5 rows for each DataFrame are included here*

SOLUTION: THE AGG METHOD

NEW MESSAGE

August 6, 2022

From: **Chandler Capital** *(Accounting)*

Subject: **Bonus Rate and Bonus Payable**

Hey again,

I'm performing some further analysis on our bonuses.

Can you create a table that has the average number of days each store hit the target?

Calculate the total bonuses payable to each store and sort the DataFrame from highest bonus owed to lowest.

Then do the same for day of week and month.

Thanks!

section04_aggregations.ipynb

Reply Forward

Solution Code

By Store:

```
transactions.groupby("store_nbr").agg(
    {"met_target": "mean", "bonus_payable": "sum"}
).sort_values(by=["bonus_payable"], ascending=False)
```

By Month:

```
transactions.groupby("month").agg(
    {"met_target": "mean", "bonus_payable": "sum"}
).sort_values(by=["bonus_payable"], ascending=False)
```

By Weekday:

```
transactions.groupby("day_of_week").agg(
    {"met_target": "mean", "bonus_payable": "sum"}
).sort_values(by=["bonus_payable"], ascending=False)
```

PRO TIP: TRANSFORM

The **.transform()** method can be used to perform aggregations without reshaping

- This is useful for calculating group-level statistics to perform row-level analysis

```
small_retail.assign(store_sales = (small_retail
                                   .groupby('store_nbr')['sales']
                                   .transform('sum')))
```

*This uses **.assign()** to create a new DataFrame column, and **.transform()** calculates the sum of 'sales' by 'store_nbr' and applies the corresponding value to each row*

	date	store_nbr	family	sales	onpromotion	store_sales
2862475	2017-05-30	25	LIQUOR,WINE,BEER	92.000	2	4800.79800
940501	2014-06-13	48	BABY CARE	0.000	0	4058.12603
1457967	2015-04-01	17	PLAYERS AND ELECTRONICS	0.000	0	1099.15101
1903307	2015-12-07	12	SEAFOOD	3.000	0	3288.39100
196280	2013-04-21	16	PREPARED FOODS	66.000	0	4365.89000
...	...	...	...	...	...	...
1875481	2015-11-21	31	PERSONAL CARE	273.000	0	3816.56100
2129111	2016-04-12	48	HOME APPLIANCES	0.000	0	4058.12603
1967389	2016-01-13	10	POULTRY	139.781	0	5272.78100
480840	2013-09-27	5	PRODUCE	7.000	0	11566.84600
23887	2013-01-14	29	POULTRY	0.000	0	4291.98500

1000 rows × 6 columns

The value for rows with store 48 is the same!

ASSIGNMENT: TRANSFORM

NEW MESSAGE

August 6, 2022

From: **Chandler Capital** *(Accounting)*

Subject: **Store Transactions Vs. Average**

Hey again,

I need some more data on store performance.

This time I want a column added to the transactions data that has the average transactions for each store… but I don't want to lose the rows for each store/day.

Once you've created that column, can you create a new column that contains the difference between the store's average and its transactions on that day?

Thanks!

section04_aggregations.ipynb

Reply Forward

Results Preview

	date	store_nbr	transactions	target_pct	met_target	bonus_payable	month	day_of_week	store_avg_trans	trans_vs_avg
0	2013-01-01	25	770	0.3080	False	0	1	1	941.400619	-171.400619
1	2013-01-02	1	2111	0.8444	False	0	1	2	1523.844272	587.155728
2	2013-01-02	2	2358	0.9432	False	0	1	2	1920.036374	437.963626
3	2013-01-02	3	3487	1.3948	True	100	1	2	3201.879475	285.120525
4	2013-01-02	4	1922	0.7688	False	0	1	2	1502.987470	419.012530

SOLUTION: TRANSFORM

NEW MESSAGE

August 6, 2022

From: **Chandler Capital** *(Accounting)*

Subject: **Store Transactions Vs. Average**

Hey again,

I need some more data on store performance.

This time I want a column added to the transactions data that has the average transactions for each store… but I don't want to lose the rows for each store/day.

Once you've created that column, can you create a new column that contains the difference between the store's average and its transactions on that day?

Thanks!

section04_aggregations.ipynb

Reply Forward

Solution Code

```
transactions["store_avg_trans"] =(transactions
                                  .groupby("store_nbr")["transactions"]
                                  .transform("mean"))

transactions["trans_vs_avg"] = (
    transactions["transactions"] - transactions["store_avg_trans"]
)

transactions.head()
```

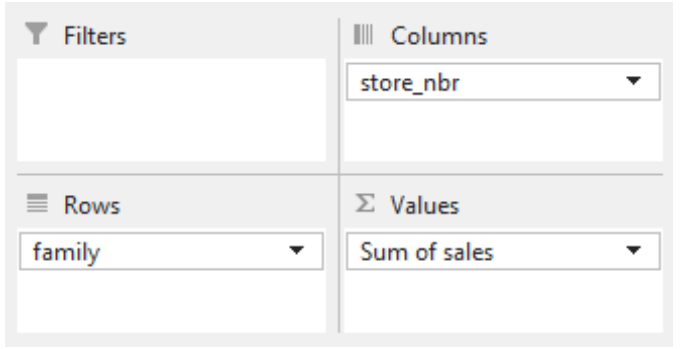
	date	store_nbr	transactions	target_pct	met_target	bonus_payable	month	day_of_week	store_avg_trans	trans_vs_avg
0	2013-01-01	25	770	0.3080	False	0	1	1	941.400619	-171.400619
1	2013-01-02	1	2111	0.8444	False	0	1	2	1523.844272	587.155728
2	2013-01-02	2	2358	0.9432	False	0	1	2	1920.036374	437.963626
3	2013-01-02	3	3487	1.3948	True	100	1	2	3201.879475	285.120525
4	2013-01-02	4	1922	0.7688	False	0	1	2	1502.987470	419.012530

PIVOT TABLES

The .pivot_table() method let's you create Excel-style **pivot tables**

```
smaller_retail.pivot_table(index='family',
                           columns='store_nbr',
                           values='sales',
                           aggfunc='sum')
```

store_nbr	1	2	3	4
family				
AUTOMOTIVE	5475.0	9100.0	15647.0	6767.0
BABY CARE	0.0	84.0	672.0	24.0
BEAUTY	4056.0	7936.0	16189.0	6890.0



Sum of sales	store_nbr			
family	1	2	3	4
AUTOMOTIVE	5575.0	9100.0	15647.0	6767.0
BABY CARE	0.0	84.0	672.0	24.0
BEAUTY	4056.0	7936.0	16189.0	6890.0

Unlike Excel, Pandas pivot tables **don't have a "filter" argument"**, but you can filter your DataFrame before pivoting to return a filtered pivot table

PIVOT TABLE ARGUMENTS

The .pivot_table() method has these arguments:

- **index**: returns a row index with distinct values from the specified column
- **columns**: returns a column index with distinct values from the specified column
- **values**: the column, or columns, to perform the aggregations on
- **aggfunc**: defines the aggregation function, or functions, to perform on the "values"
- **margins**: returns row and column totals when True *(False by default)*

```
smaller_retail.pivot_table(index='family',
                           columns='store_nbr',
                           values='sales',
                           aggfunc='sum')
```

store_nbr	1	2	3	4
family				
AUTOMOTIVE	5475.0	9100.0	15647.0	6767.0
BABY CARE	0.0	84.0	672.0	24.0
BEAUTY	4056.0	7936.0	16189.0	6890.0

This returns distinct 'family' values as rows, distinct 'store_nbr' values as columns, and sums the 'sales' for each combination of 'family' and 'str_nbr' as the values

PIVOT TABLE ARGUMENTS

Grouping Columns

Multi-index DataFrames

Aggregating Groups

Pivot Tables

Melting DataFrames

The .pivot_table() method has these arguments:

- **index**: returns a row index with distinct values from the specified column
- **columns**: returns a column index with distinct values from the specified column
- **values**: the column, or columns, to perform the aggregations on
- **aggfunc**: defines the aggregation function, or functions, to perform on the "values"
- **margins**: returns row and column totals when True *(False by default)*

```
smaller_retail.pivot_table(index='family',
                           columns='store_nbr',
                           values='sales',
                           aggfunc='sum',
                           margins=True)
```

store_nbr	1	2	3	4	All
family					
AUTOMOTIVE	5475.0	9100.0	15647.0	6767.0	36989.0
BABY CARE	0.0	84.0	672.0	24.0	780.0
BEAUTY	4056.0	7936.0	16189.0	6890.0	35071.0
All	9531.0	17120.0	32508.0	13681.0	72840.0

*Specifying **margins=True** adds row and column totals based on the aggregation (the corner represents the grand total)*

MULTIPLE AGGREGATION FUNCTIONS

Multiple aggregation functions can be passed to the "aggfunc" argument

- The new values are added as additional columns

```
smaller_retail.pivot_table(index='family',
                           columns='store_nbr',
                           values='sales',
                           aggfunc=('min','max'))
```

The functions are passed as a tuple

	max				min			
store_nbr	**1**	**2**	**3**	**4**	**1**	**2**	**3**	**4**
family								
AUTOMOTIVE	19.0	23.0	48.0	22.0	0.0	0.0	0.0	0.0
BABY CARE	0.0	5.0	11.0	3.0	0.0	0.0	0.0	0.0
BEAUTY	12.0	108.0	93.0	19.0	0.0	0.0	0.0	0.0

There is a column for each store_nbr min and max (this can create a very wide dataset very quickly)

MULTIPLE AGGREGATION FUNCTIONS

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

Multiple aggregation functions can be passed to the "aggfunc" argument

- The new values are added as additional columns

```
smaller_retail.pivot_table(
    index="family",
    columns="store_nbr",
    aggfunc={"sales": ["sum", "mean"], "onpromotion": "max"},
)
```

Use a dictionary to apply specific functions to specific columns

	onpromotion				sales							
	max				mean				sum			
store_nbr	1	2	3	4	1	2	3	4	1	2	3	4
family												
AUTOMOTIVE	1	1	1	1	3.251188	5.403800	9.291568	4.018409	5475.0	9100.0	15647.0	6767.0
BABY CARE	0	0	1	0	0.000000	0.049881	0.399050	0.014252	0.0	84.0	672.0	24.0
BEAUTY	2	2	2	2	2.408551	4.712589	9.613420	4.091449	4056.0	7936.0	16189.0	6890.0

PIVOT TABLES VS. GROUPBY

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

If the column argument isn't specified in a pivot table, it will return a table that's identical to one grouped by the index columns

```
smaller_retail.pivot_table(index=('family', 'store_nbr'),
                           aggfunc=('sum','mean'),
                           values=('sales', 'onpromotion'))
```

		onpromotion		sales	
		mean	sum	mean	sum
family	store_nbr				
AUTOMOTIVE	1	0.008314	14.0	3.251188	5475.0
	2	0.007720	13.0	5.403800	9100.0
	3	0.007720	13.0	9.291568	15647.0
	4	0.005938	10.0	4.018409	6767.0
BABY CARE	1	0.000000	0.0	0.000000	0.0
	2	0.000000	0.0	0.049881	84.0
	3	0.000594	1.0	0.399050	672.0
	4	0.000000	0.0	0.014252	24.0
BEAUTY	1	0.134204	226.0	2.408551	4056.0
	2	0.171615	289.0	4.712589	7936.0
	3	0.232185	391.0	9.613420	16189.0
	4	0.159739	269.0	4.091449	6890.0

=

```
smaller_retail.groupby(["family", "store_nbr"]).agg(
    {"onpromotion": ["mean", "sum"],
     "sales": ["mean", "sum"]})
```

		onpromotion		sales	
		mean	sum	mean	sum
family	store_nbr				
AUTOMOTIVE	1	0.008314	14	3.251188	5475.0
	2	0.007720	13	5.403800	9100.0
	3	0.007720	13	9.291568	15647.0
	4	0.005938	10	4.018409	6767.0
BABY CARE	1	0.000000	0	0.000000	0.0
	2	0.000000	0	0.049881	84.0
	3	0.000594	1	0.399050	672.0
	4	0.000000	0	0.014252	24.0
BEAUTY	1	0.134204	226	2.408551	4056.0
	2	0.171615	289	4.712589	7936.0
	3	0.232185	391	9.613420	16189.0
	4	0.159739	269	4.091449	6890.0

PIVOT TABLES VS. GROUPBY

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

If the column argument isn't specified in a pivot table, it will return a table that's identical to one grouped by the index columns

```
smaller_retail.pivot_table(index=('family', 'store_nbr'),
                           aggfunc=('sum','mean'),
                           values=('sales', 'onpromotion'))
```

		onpromotion		sales	
		mean	sum	mean	sum
family	store_nbr				
AUTOMOTIVE	1	0.008314	14.0	3.251188	5475.0
	2	0.007720	13.0	5.403800	9100.0
	3	0.007720	13.0	9.291568	15647.0
	4	0.005938	10.0	4.018409	6767.0
BABY CARE	1	0.000000	0.0	0.000000	0.0
	2	0.000000	0.0	0.049881	84.0
	3	0.000594	1.0	0.399050	672.0
	4	0.000000	0.0	0.014252	24.0

≠

```
smaller_retail.groupby(["family", "store_nbr"]).agg(
    on_promo_avg=("onpromotion", "mean"),
    on_promo_sum=("onpromotion", "sum"),
    sales_avg=("sales", "mean"),
    sales_sum=("sales", "sum"),
)
```

		on_promo_avg	on_promo_sum	sales_avg	sales_sum
family	store_nbr				
AUTOMOTIVE	1	0.008314	14	3.251188	5475.0
	2	0.007720	13	5.403800	9100.0
	3	0.007720	13	9.291568	15647.0
	4	0.005938	10	4.018409	6767.0
BABY CARE	1	0.000000	0	0.000000	0.0
	2	0.000000	0	0.049881	84.0
	3	0.000594	1	0.399050	672.0
	4	0.000000	0	0.014252	24.0
BEAUTY	1	0.134204	226	2.408551	4056.0
	2	0.171615	289	4.712589	7936.0
	3	0.232185	391	9.613420	16189.0
	4	0.159739	269	4.091449	6890.0

PRO TIP: Use groupby if you don't need columns in the pivot, as you can use named aggregations to flatten the column index

PRO TIP: HEATMAPS

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

You can style a DataFrame based on its values to create a **heatmap**

- Simply chain .style.background_gradient() to your DataFrame and add a "cmap" argument

axis=None adds a red-yellow-green heatmap to the whole table

```
(smaller_retail.pivot_table(
        index="family",
    columns="store_nbr",
    values="sales",
    aggfunc="sum"
    ).style.background_gradient(cmap="RdYlGn", axis=None)
)
```

store_nbr	1	2	3	4
family				
AUTOMOTIVE	5475.000000	9100.000000	15647.000000	6767.000000
BABY CARE	0.000000	84.000000	672.000000	24.000000
BEAUTY	4056.000000	7936.000000	16189.000000	6890.000000

axis=1 adds a red-yellow-green heatmap to each row

```
(
    smaller_retail.pivot_table(
        index="family",
        columns="store_nbr",
        values="sales",
        aggfunc="sum"
    ).style.background_gradient(cmap="RdYlGn", axis=1)
)
```

store_nbr	1	2	3	4
family				
AUTOMOTIVE	5475.000000	9100.000000	15647.000000	6767.000000
BABY CARE	0.000000	84.000000	672.000000	24.000000
BEAUTY	4056.000000	7936.000000	16189.000000	6890.000000

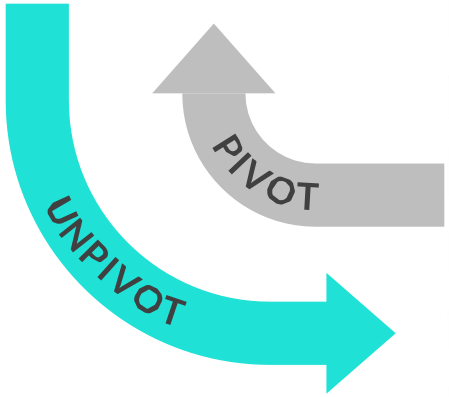
MELT

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

The **.melt()** method will *unpivot* a DataFrame, or convert columns into rows

```
country_revenue
```

	country	2000	2001	2002	2003
0	Algeria	151	171	176	184
1	Egypt	204	214	227	241
2	Nigeria	277	301	311	342
3	South Africa	190	211	224	243




```
country_revenue.melt()
```

	variable	value
0	country	Algeria
1	country	Egypt
2	country	Nigeria
3	country	South Africa
4	2000	151
5	2000	204
6	2000	277
7	2000	190
8	2001	171
9	2001	214
10	2001	301
11	2001	211
12	2002	176
13	2002	227
14	2002	311
15	2002	224
16	2003	184
17	2003	241
18	2003	342
19	2003	243

How does this code work?

- The original column names *(country, 2000, etc.)* are turned into a single "variable" column
- The values for each original column are placed on a single "value" column next to its corresponding column name

Note that the resulting table isn't perfect, as .melt() **unpivots a DataFrame around its index**, while ideally you'd want to pivot this around the country values

MELT

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

Use the "id_vars" argument to **specify the column** to unpivot the DataFrame by

```
country_revenue
```

	country	2000	2001	2002	2003
0	Algeria	151	171	176	184
1	Egypt	204	214	227	241
2	Nigeria	277	301	311	342
3	South Africa	190	211	224	243

UNPIVOT

```
country_revenue.melt(id_vars="country")
```

	country	variable	value
0	Algeria	2000	151
1	Egypt	2000	204
2	Nigeria	2000	277
3	South Africa	2000	190
4	Algeria	2001	171
5	Egypt	2001	214
6	Nigeria	2001	301
7	South Africa	2001	211
8	Algeria	2002	176
9	Egypt	2002	227
10	Nigeria	2002	311
11	South Africa	2002	224
12	Algeria	2003	184
13	Egypt	2003	241
14	Nigeria	2003	342
15	South Africa	2003	243

How does this code work?

- The "id_vars" column *(country)* is kept in the DataFrame
- The rest of the DataFrame columns are "melted" around the countries in matching variable/value pairs

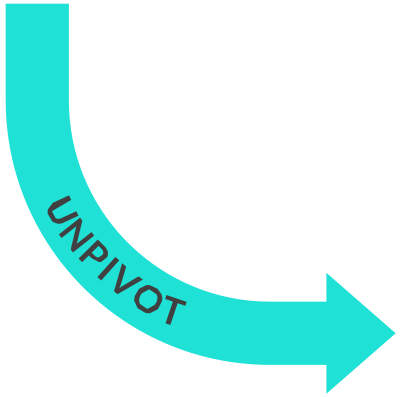
MELT

- Grouping Columns
- Multi-index DataFrames
- Aggregating Groups
- Pivot Tables
- Melting DataFrames

You can also select the columns to melt and name the "variable" & "value" columns

```
country_revenue
```

	country	2000	2001	2002	2003
0	Algeria	151	171	176	184
1	Egypt	204	214	227	241
2	Nigeria	277	301	311	342
3	South Africa	190	211	224	243




```
country_revenue.melt(
    id_vars="country",
    value_vars=["2001", "2002", "2003"],
    var_name="year",
    value_name="GDP",
)
```

	country	year	GDP
0	Algeria	2001	171
1	Egypt	2001	214
2	Nigeria	2001	301
3	South Africa	2001	211
4	Algeria	2002	176
5	Egypt	2002	227
6	Nigeria	2002	311
7	South Africa	2002	224
8	Algeria	2003	184
9	Egypt	2003	241
10	Nigeria	2003	342
11	South Africa	2003	243

How does this code work?

- **id_vars** melts the DataFrame around the "country" column
- **value_vars** selects 2001, 2002, and 2003 as the columns to melt *(omitting 2000)*
- **var_name** & **value_name** set "year" and "GDP" as column names instead of "variable" and "value"

ASSIGNMENT: PIVOT & MELT

NEW MESSAGE

August 7, 2022

From: **Chandler Capital** *(Accounting)*

Subject: **Store Transactions Vs. Average**

Hey again,

I need to summarize store numbers 1 through 11 by total bonus payable for each day of the week.

Can you create a pivot table that has the sum of bonus payable by day of week? Make sure to filter out any rows that had 0 bonus payable first and add a heatmap across the rows.

Then unpivot the table so we have one row for each store and day of the week with the corresponding total owed.

Thanks

section04_aggregations.ipynb

Reply Forward

Results Preview

day_of_week	0	1	2	3	4	5	6
store_nbr							
1	200.000000	300.000000	300.000000	200.000000	100.000000	nan	nan
2	300.000000	600.000000	500.000000	400.000000	400.000000	500.000000	200.000000
3	24000.000000	23900.000000	23900.000000	23900.000000	23900.000000	24000.000000	23700.000000
4	200.000000	300.000000	300.000000	200.000000	100.000000	200.000000	nan
5	200.000000	300.000000	300.000000	100.000000	100.000000	100.000000	nan
6	400.000000	500.000000	500.000000	300.000000	200.000000	900.000000	300.000000
7	200.000000	300.000000	300.000000	200.000000	100.000000	100.000000	nan
8	22000.000000	18800.000000	23800.000000	18000.000000	22900.000000	23400.000000	20000.000000
9	1200.000000	800.000000	800.000000	700.000000	400.000000	7900.000000	5100.000000
11	3500.000000	4800.000000	3200.000000	3000.000000	2000.000000	15600.000000	17600.000000

	store_nbr	day_of_week	bonus_payable
0	1	0	200.0
1	2	0	300.0
2	3	0	24000.0
3	4	0	200.0
4	5	0	200.0

SOLUTION: PIVOT & MELT

NEW MESSAGE

August 7, 2022

From: **Chandler Capital** *(Accounting)*

Subject: **Store Transactions Vs. Average**

Hey again,

I need to summarize store numbers 1 through 11 by total bonus payable for each day of the week.

Can you create a pivot table that has the sum of bonus payable by day of week? Make sure to filter out any rows that had 0 bonus payable first and add a heatmap across the rows.

Then unpivot the table so we have one row for each store and day of the week with the corresponding total owed.

Thanks

section04_aggregations.ipynb

Reply Forward

Solution Code

```
transactions.loc[transactions["bonus_payable"] != 0].pivot_table(
    index="store_nbr", columns="day_of_week", values="bonus_payable", aggfunc="sum",
).iloc[:10].style.background_gradient(cmap="RdYlGn", axis=1)
```

day_of_week	0	1	2	3	4	5	6
store_nbr							
1	200.000000	300.000000	300.000000	200.000000	100.000000	nan	nan
2	300.000000	600.000000	500.000000	400.000000	400.000000	500.000000	200.000000
3	24000.000000	23900.000000	23900.000000	23900.000000	23900.000000	24000.000000	23700.000000
4	200.000000	300.000000	300.000000	200.000000	100.000000	200.000000	nan
5	200.000000	300.000000	300.000000	100.000000	100.000000	100.000000	nan
6	400.000000	500.000000	500.000000	300.000000	200.000000	900.000000	300.000000
7	200.000000	300.000000	300.000000	200.000000	100.000000	100.000000	nan
8	22000.000000	18800.000000	23800.000000	18000.000000	22900.000000	23400.000000	20000.000000
9	1200.000000	800.000000	800.000000	700.000000	400.000000	7900.000000	5100.000000
11	3500.000000	4800.000000	3200.000000	3000.000000	2000.000000	15600.000000	17600.000000

```
transactions.loc[transactions["bonus_payable"] != 0].pivot_table(
    index="store_nbr", columns="day_of_week", values="bonus_payable", aggfunc="sum",
).reset_index().melt(id_vars="store_nbr", value_name="bonus_payable")
```

	store_nbr	day_of_week	bonus_payable
0	1	0	200.0
1	2	0	300.0
2	3	0	24000.0
3	4	0	200.0
4	5	0	200.0

KEY TAKEAWAYS

- Use the **.groupby()** method to aggregate a DataFrame by specific columns
  - *You also need to specify a column of values to aggregate and an aggregate function*

- Avoid working with **multi-index DataFrames** whenever possible
  - *Multi-index DataFrames are created by default when grouping by more than one column*
  - *It's worth knowing how to access multi-index DataFrames, but it's advised to avoid them by resetting the index*

- Use the **.agg()** method to specify multiple aggregation functions when grouping
  - *Named aggregations allow you to set intuitive column names and prevent multi-index columns*

- The **.pivot_table()** and **.melt()** methods let you pivot and unpivot DataFrames
  - *Pandas pivot tables work just like Excel, and make data "wide" by converting unique row values into columns*
  - *With .melt(), you can make "wide" tables "long" in order to analyze the data traditionally*