

# DATA ANALYSIS WITH PYTHON & PANDAS

★★★★★ *With Expert Python Instructor Chris Bruehl*



# COURSE STRUCTURE

---



This is a **project-based** course for students looking for a practical, hands-on, and highly engaging approach to learning data analysis with Python using the Pandas library

*Additional resources include:*

- ★ **Downloadable PDF** to serve as a helpful reference when you're offline or on the go
- ★ **Quizzes & Assignments** to test and reinforce key concepts, with step-by-step solutions
- ★ **Interactive demos** to keep you engaged and apply your skills throughout the course

# COURSE OUTLINE

---

1

## Intro to Pandas & NumPy

Introduce Pandas & NumPy, two critical Python libraries that help structure data in arrays & DataFrames and contain built-in functions for data analysis

2

## Pandas Series

Introduce Pandas Series, the Python equivalent of a column of data, and cover their basic properties, creation, manipulation, and useful functions for analysis

3

## DataFrames

Work with Pandas DataFrames, the Python equivalent of an Excel or SQL table, and use them to store, manipulate, and analyze data efficiently

4

## Aggregating & Reshaping

Aggregate & reshape data in DataFrames by grouping columns, performing aggregation calculations, and pivoting & unpivoting data

5

## Data Visualization

Learn the basics of data visualization in Pandas, and use the plot method to create & customize line charts, bar charts, scatterplots, and histograms

# COURSE OUTLINE

---

6

## Mid-Course Project

Put your skills to the test on a new dataset by analyzing key metrics for a new retailer and share insights into its potential acquisition by Maven MegaMart

7

## Time Series

Learn how to work with the datetime data type in Pandas to extract date components, group by dates, and perform calculations like moving averages

8

## Importing & Exporting Data

Read in data from flat files and apply processing steps during import, create DataFrames by querying SQL tables, and write data back out to its source

9

## Combining DataFrames

Combine multiple DataFrames by joining data from related fields to add new columns, and appending data with the same fields to add new rows

10

## Final Project

Put the finishing touches on your project by joining a new table, performing time series analysis, and optimizing your workflow, and writing out your results

# INTRODUCING THE COURSE PROJECT



## THE **SITUATION**

You've just been hired as a Data Analyst for **Maven MegaMart**, a multinational corporation that operates a chain of retail and grocery stores. They recently received a sample of data from a new retailer they're looking to acquire, and they need you to identify and deliver key insights about their sales history.



## THE **ASSIGNMENT**

Your task is to **analyze over 2 million transactions** by product, household, and store to get a better understanding of the retailer's main strengths. From there, you need to review the company's discount scheme to assess whether they can expect to attract customers without losing margin.



## THE **OBJECTIVES**

### Use Python to:

- Read in multiple flat files efficiently
- Join tables to provide a single source of information
- Shape & aggregate sales data to calculate KPIs
- Visualize the data to communicate your findings



# SETTING EXPECTATIONS

---



This course covers the **core Pandas functionality**

- We'll cover critical functions, methods, and best practices for manipulating and analyzing data using Pandas DataFrames, but won't dive into advanced statistical analysis or data prep for machine learning



We will only cover **basic data visualization** in Python

- We'll use Matplotlib to create & customize common charts like histograms, scatterplots, bar & line charts, but we'll cover data visualization in depth in a separate course



We'll use **Jupyter Notebooks** as our primary coding environment

- Jupyter Notebooks are free to use, and the industry standard for conducting data analysis with Python (we'll introduce Google Colab as an alternative, cloud-based environment as well)



You do **NOT** need to be a Python expert to take this course

- It is strongly recommended that you complete our Python Essentials for Analysts course, or have a solid understanding of Base Python concepts like data types, variables, conditional logic, loops, and functions

# INSTALLATION & SETUP



# INSTALLING ANACONDA (MAC)

Installing  
Anaconda

Appending

Joining

1) Go to [anaconda.com/products/distribution](https://anaconda.com/products/distribution) and click

Download

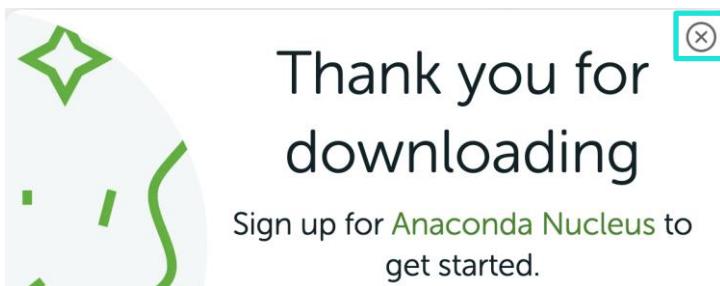
Individual Edition is now

## ANACONDA DISTRIBUTION

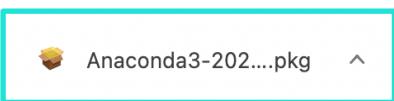
The world's most popular open-source Python distribution platform



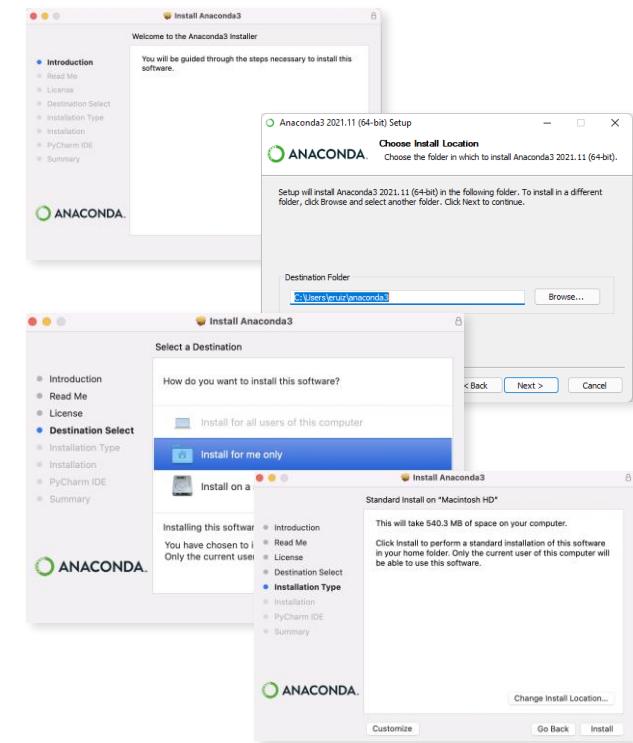
2) Click **X** on the Anaconda Nucleus pop-up  
(no need to launch)



3) Launch the downloaded Anaconda **pkg** file



4) Follow the **installation steps**  
(default settings are OK)





# INSTALLING ANACONDA (PC)

Installing  
Anaconda

Appending

Joining

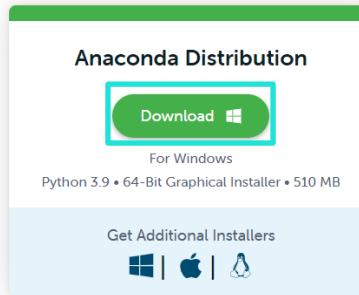
1) Go to [anaconda.com/products/distribution](https://anaconda.com/products/distribution) and click

[Download](#)

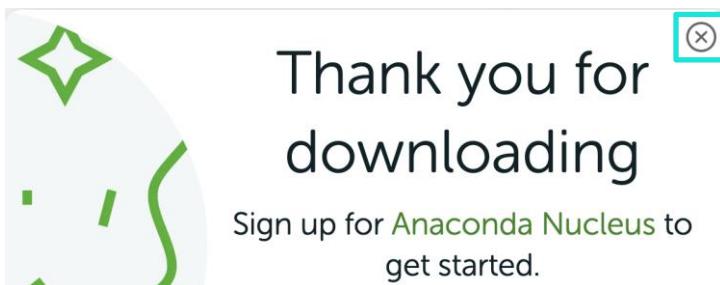
Individual Edition is now

## ANACONDA DISTRIBUTION

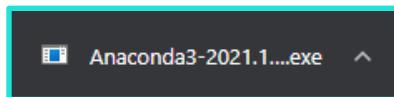
The world's most popular open-source Python distribution platform



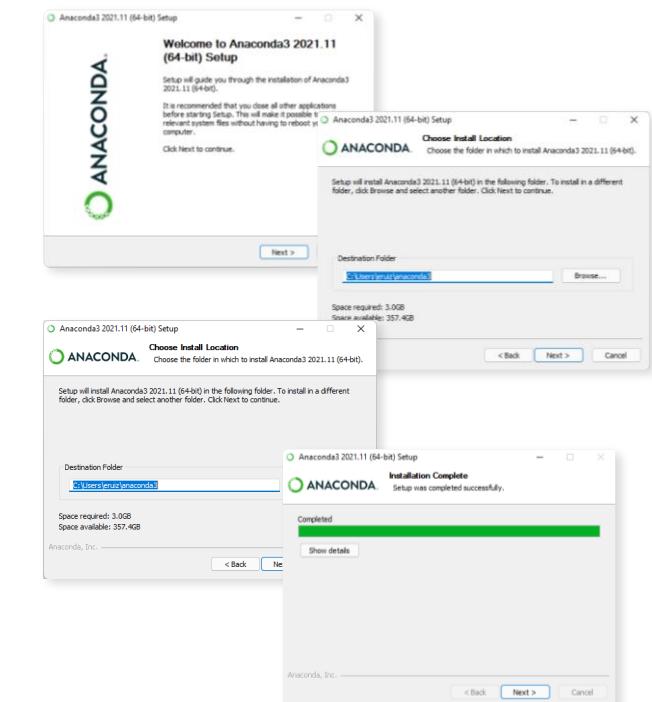
2) Click **X** on the Anaconda Nucleus pop-up  
(no need to launch)



3) Launch the downloaded Anaconda **exe** file



4) Follow the **installation steps**  
(default settings are OK)





# LAUNCHING JUPYTER

1) Launch **Anaconda Navigator**

The screenshot shows the Anaconda Navigator application window. On the left, there's a sidebar with buttons for 'Home', 'Environments', 'Learning', and 'Community'. The main area is titled 'ANACONDA.NAVIGATOR' and shows a grid of application cards. One card for 'Jupyter Notebook' is highlighted with a red border. The card for 'Jupyter Notebook' includes a brief description: 'Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.' and a 'Launch' button.

2) Find **Jupyter Notebook** and click **Launch**

Installing  
Anaconda

Launching  
Jupyter

Joining



# YOUR FIRST JUPYTER NOTEBOOK

Installing  
Anaconda

Launching  
Jupyter

Joining

- Once inside the Jupyter interface, **create a folder** to store your notebooks for the course

The screenshot shows the Jupyter interface with the 'Files' tab selected. A context menu is open over a folder named 'Python 3 (ipykernel)'. The menu options include 'Upload', 'New', and a dropdown menu with 'Notebook', 'Text File', 'Folder', and 'Terminal'. The 'Folder' option is highlighted with a red arrow. To the right, the file list shows a newly created folder named 'Untitled Folder'.

**NOTE:** You can rename your folder by clicking "Rename" in the top left corner

- Open your new coursework folder and **launch your first Jupyter notebook!**

The screenshot shows the Jupyter interface with the 'Files' tab selected. A context menu is open over a file named 'Python 3 (ipykernel)'. The menu options include 'Upload', 'New', and a dropdown menu with 'Notebook', 'Text File', 'Folder', and 'Terminal'. The 'Notebook' option is highlighted with a red arrow.

The screenshot shows the Jupyter interface with the 'File' tab selected. The title bar indicates the notebook is titled 'Untitled' and was last checked at a minute ago. The toolbar includes 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', 'Help', 'Trusted', and a Python 3 (ipykernel) icon. Below the toolbar, the code editor shows the prompt 'In [ ]:'.

**NOTE:** You can rename your notebook by clicking on the title at the top of the screen



# THE NOTEBOOK SERVER

Installing  
Anaconda

Launching  
Jupyter

Joining

**NOTE:** When you launch a Jupyter notebook, a terminal window may pop up as well; this is called a **notebook server**, and it powers the notebook interface

```
Last login: Tue Jan 25 14:04:12 on ttys002
(base) chrisb@Chriss-MBP ~ % jupyter notebook
[I 2022-01-26 08:45:53.886 LabApp] JupyterLab extension loaded from /Users/chrisb/opt/anaconda3/lib/python3.9/site-packages/jupyterlab
[I 2022-01-26 08:45:53.886 LabApp] JupyterLab application directory is /Users/chrisb/opt/anaconda3/share/jupyter/lab
[I 08:45:53.890 NotebookApp] Serving notebooks from local directory: /Users/chrisb
[I 08:45:53.890 NotebookApp] Jupyter Notebook 6.4.5 is running at:
[I 08:45:53.890 NotebookApp] http://localhost:8888/?token=3159cf032d9e6841d04910e257db2b24b6df6dfc878d6d5f
[I 08:45:53.890 NotebookApp] or http://127.0.0.1:8888/?token=3159cf032d9e6841d04910e257db2b24b6df6dfc878d6d5f
[I 08:45:53.890 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 08:45:53.893 NotebookApp]

To access the notebook, open this file in a browser:
  file:///Users/chrisb/Library/Jupyter/runtime/nbserver-27175-open.html
Or copy and paste one of these URLs:
  http://localhost:8888/?token=3159cf032d9e6841d04910e257db2b24b6df6dfc878d6d5f
  or http://127.0.0.1:8888/?token=3159cf032d9e6841d04910e257db2b24b6df6dfc878d6d5f
[W 08:46:05.829 NotebookApp] Notebook Documents/Maven_Coursework/Python_Intro.ipynb
```



If you close the server window,  
**your notebooks will not run!**

Depending on your OS, and method of launching Jupyter, one may not open. As long as you can run your notebooks, don't worry!



# ALTERNATIVE: GOOGLE COLAB

**Google Colab** is Google's cloud-based version of Jupyter Notebooks

## To create a Colab notebook:

1. Log in to a Gmail account
2. Go to [colab.research.google.com](https://colab.research.google.com)
3. Click “new notebook”

Google Colab



Colab is very similar to Jupyter Notebooks (*they even share the same file extension*); the main difference is that you are connecting to **Google Drive** rather than your machine, so files will be stored in Google's cloud

The screenshot shows the Google Colab interface. At the top, there are tabs for 'Examples', 'Recent', 'Google Drive', 'GitHub', and 'Upload'. The 'Recent' tab is selected. Below it, a table lists recent notebooks:

Title	Last opened	First opened	Actions
Welcome To Colaboratory	1:34 PM	January 4	
Scratch_Work.ipynb	January 25	January 4	

Below this is a 'Drive' interface window. It shows a sidebar with 'New', 'Priority', 'My Drive' (which is selected), 'Shared drives', 'Shared with me', 'Recent', 'Starred', 'Trash', and 'Storage'. The main area shows a file named '1\_Python\_Intro.ipynb' with the note 'You edited today'. A modal window is open at the bottom right, containing a 'New notebook' button and a 'Cancel' button.

# INTRO TO PANDAS & NUMPY

# INTRO TO PANDAS & NUMPY



In this section we'll introduce **Pandas** & **NumPy**, two critical Python libraries that help structure data in arrays & DataFrames and contain built-in functions for data analysis

## TOPICS WE'LL COVER:

Intro to Pandas & NumPy

NumPy Array Basics

Array Creation

Array Indexing & Slicing

Array Operations

Vectorization & Broadcasting

## GOALS FOR THIS SECTION:

- Convert Python lists to NumPy arrays, and create new arrays from scratch using functions
- Apply array indexing, slicing, methods, and functions to perform operations on NumPy arrays
- Understand the concepts of vectorization and broadcasting, which are critical in making NumPy and Pandas more efficient than base Python



# MEET PANDAS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting



**Pandas** is Python's most widely used library for data analysis, and contains functions for accessing, aggregating, joining, and analyzing data

Its data structure, the DataFrame, is analogous to SQL tables or Excel worksheets

Custom indices & column titles make working with datasets more intuitive!



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



# MEET NUMPY

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting



**NumPy** is an open-source library that is the universal standard for working with numerical data in Python, and forms the foundation of other libraries like Pandas. Pandas DataFrames are built on NumPy arrays and can leverage NumPy functions.

The indices, column names,  
and data columns are all  
stored as NumPy arrays

Pandas just adds convenient  
wrappers and functions!

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



# NUMPY ARRAYS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

**NumPy arrays** are fixed-size containers of items that are more efficient than Python lists or tuples for data processing

- They only store a single data type (*mixed data types are stored as a string*)
- They can be one dimensional or multi-dimensional
- Array elements can be modified, but the array size cannot change

```
import numpy as np
sales = [0, 5, 155, 0, 518, 0, 1827, 616, 317, 325]
sales_array = np.array(sales)
sales_array
array([ 0,  5, 155,  0, 518,  0, 1827, 616, 317, 325])
```

'np' is the standard alias for the NumPy library

} NumPy's **array** function converts Python lists and tuples into NumPy arrays



# ARRAY PROPERTIES

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

NumPy arrays have these key properties:

- **ndim** – the number of dimensions (axes) in the array
- **shape** – the size of the array for each dimension
- **size** – the total number of elements in the array
- **dtype** – the data type of the elements in the array

```
sales = [0, 5, 155, 0, 518, 0, 1827, 616, 317, 325]
sales_array = np.array(sales)
type(sales_array)
numpy.ndarray
```

```
print(f"ndim: {sales_array.ndim}")
print(f"shape: {sales_array.shape}")
print(f"size: {sales_array.size}")
print(f"dtype: {sales_array.dtype}")
```

ndim: 1 → The sales\_array has 1 dimension  
shape: (10,) → The dimension has a size of 10  
size: 10 → The array has 10 elements total  
dtype: int64 → The elements are stored as 64-bit integers

NumPy arrays are a **ndarray** Python data type,  
which stands for n-dimensional array



# ARRAY PROPERTIES

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

NumPy arrays have these key properties:

- **ndim** – the number of dimensions (axes) in the array
- **shape** – the size of the array for each dimension
- **size** – the total number of elements in the array
- **dtype** – the data type of the elements in the array

```
sales = [[0, 5, 155, 0, 518], [0, 1827, 616, 317, 325]]  
  
sales_array = np.array(sales)  
  
array([[ 0, 5, 155, 0, 518],  
       [0, 1827, 616, 317, 325]])
```

Converting a nested list creates a multi-dimensional array, where each nested list is a dimension

**NOTE:** The nested lists must be of equal length

```
print(f"ndim: {sales_array.ndim}")  
print(f"shape: {sales_array.shape}")  
print(f"size: {sales_array.size}")  
print(f"dtype: {sales_array.dtype}")
```

**ndim: 2** → The sales\_array has 2 dimensions  
**shape: (2, 5)** → The first dimension has a size of 2 (rows) and the second a size of 5 (columns)  
**size: 10** → It has 10 elements total  
**dtype: int64** → The elements are stored as 64-bit integers

# ASSIGNMENT: ARRAY BASICS



## NEW MESSAGE

June 16, 2022

From: **Ross Retail** (Head of Analytics)  
Subject: NumPy?

Hi there, welcome to the Maven MegaMart!

Your resume mentions you have basic Python experience.  
Our finance team has been asking us to cut software costs –  
can you help us dig into Python as an analysis tool?

I know NumPy is foundational, but not much beyond that.

Can you convert a Python list into a NumPy array and help me  
get familiar with their properties?

Thanks!

[numpy\\_assignments.ipynb](#)

Reply

Forward

## Results Preview

```
my_list = [x * 10 for x in range(1, 11)]
```

```
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
```

```
ndim: 1
shape: (10,)
size: 10
dtype: int64
```

# SOLUTION: ARRAY BASICS



## NEW MESSAGE

June 16, 2022

From: **Ross Retail** (Head of Analytics)  
Subject: NumPy?

Hi there, welcome to the Maven MegaMart!

Your resume mentions you have basic Python experience. Our finance team has been asking us to cut software costs – can you help us dig into Python as an analysis tool?

I know NumPy is foundational, but not much beyond that.

Can you convert a Python list into a NumPy array and help me get familiar with their properties?

Thanks!

numpy\_assignments.ipynb

Reply

Forward

## Solution Code

```
my_list = [x * 10 for x in range(1, 11)]  
  
my_array = np.array(my_list)  
  
my_array  
  
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100])
```

```
print(f"ndim: {my_array.ndim}")  
print(f"shape: {my_array.shape}")  
print(f"size: {my_array.size}")  
print(f"dtype: {my_array.dtype}")
```

```
ndim: 1  
shape: (10,)  
size: 10  
dtype: int64
```



# ARRAY CREATION

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

As an alternative to converting lists, you can **create arrays** using functions

**ones**

*Creates an array of ones of a given size, as float by default*

`np.ones((rows, cols), dtype)`

**zeros**

*Creates an array of zeros of a given size, as float by default*

`np.zeros((rows, cols), dtype)`

**arange**

*Creates an array of integers with given start & stop values, and a step size (only stop is required, and is not inclusive)*

`np.arange(start, stop, step)`

**linspace**

*Creates an array of floats with given start & stop values with n elements, separated by a consistent step size (stop is inclusive)*

`np.linspace(start, stop, n)`

**reshape**

*Changes an array into the specified dimensions, if compatible*

`np.array.reshape(rows, cols)`



# ARRAY CREATION

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

As an alternative to converting lists, you can **create arrays** using functions

`np.ones((rows, cols), dtype)`

```
np.ones(4,)
```

```
array([1., 1., 1., 1.])
```

`np.zeros((rows, cols), dtype)`

```
np.zeros((2, 5), dtype=int)
```

```
array([[0, 0, 0, 0, 0],  
       [0, 0, 0, 0, 0]])
```

`np.arange(start, stop, step)` start is 0 and step is 1 by default

```
np.arange(10)
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

stop is not inclusive

`np.linspace(start, stop, n)`

```
np.linspace(0, 100, 5)
```

```
array([ 0., 25., 50., 75., 100.])
```

stop is inclusive

`np.array.reshape(rows, cols)`

```
np.arange(1, 9, 2).reshape(2, 2)
```

```
array([[1, 3],  
       [5, 7]])
```



# RANDOM NUMBER ARRAYS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

You can create **random number arrays** from a variety of distributions using NumPy functions and methods (great for sampling and simulation!)

**default\_rng**

Creates a random number generator (the seed is for reproducibility)

`np.default_rng(seed)`

**random**

Returns n random numbers from a uniform distribution between 0 and 1

`rng.random(n)`

**normal**

Returns n random numbers from a normal distribution with a given mean and standard deviation

`rng.normal(mean, stdev, n)`

'rng' is the standard variable name for  
the default\_rng number generator



# RANDOM NUMBER ARRAYS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

You can create **random number arrays** from a variety of distributions using NumPy functions and methods (*great for sampling and simulation!*)

```
from numpy.random import default_rng

rng = default_rng(12345)

random_array = rng.random(10)
random_array
```

array([0.22733602, 0.31675834, 0.79736546, 0.67625467, 0.39110955,  
 0.33281393, 0.59830875, 0.18673419, 0.67275604, 0.94180287])

First, we're creating a random number generator with a seed of 12345 and assigning it to 'rng' using **default\_rng**

Then we're using the **random** method on 'rng' to return an array with 10 random numbers

```
rng = default_rng(12345)
mean, stddev = 5, 1
random_normal = rng.normal(mean, stddev, size=10)
random_normal
```

array([3.57617496, 6.26372846, 4.12933826, 4.74082677, 4.92465669,  
 4.25911535, 3.6322073 , 5.6488928 , 5.36105811, 3.04713694])

Here we're using the **normal** method on 'rng' to return an array with 10 random numbers from a normal distribution with a mean of 5 and a st. deviation of 1



**PRO TIP:** Even though it's optional, make sure to **set a seed** when generating random numbers to ensure you and others can recreate the work you've done (the value for the seed is less important)

# ASSIGNMENT: ARRAY CREATION

 NEW MESSAGE  
June 17, 2022

**From:** Ross Retail (Head of Analytics)  
**Subject:** Array Creation

Thanks for your help last time – I'm starting to understand NumPy Arrays!

Are there any NumPy functions that can create arrays so we don't have to convert from a Python list? Recreate the array from the first assignment but make it 5 rows by 2 columns.

Once you've done that, create an array of random numbers between 0 and 1 in a 3x3 shape. One of our data scientists has been asking about this so I want to make sure it's possible.

Thanks!

 numpy\_assignments.ipynb

## Results Preview

```
array([[ 10.,  20.],  
       [ 30.,  40.],  
       [ 50.,  60.],  
       [ 70.,  80.],  
       [ 90., 100.]])
```

```
random_array
```

```
array([[0.24742606, 0.09299006, 0.61176337],  
       [0.06066207, 0.66103343, 0.75515778],  
       [0.1108689 , 0.04305584, 0.41441747]])
```

# SOLUTION: ARRAY CREATION



NEW MESSAGE

June 17, 2022

From: **Ross Retail** (Head of Analytics)  
Subject: **Array Creation**

Thanks for your help last time – I'm starting to understand NumPy Arrays!

Are there any NumPy functions that can create arrays so we don't have to convert from a Python list? Recreate the array from the first assignment but make it 5 rows by 2 columns.

Once you've done that, create an array of random numbers between 0 and 1 in a 3x3 shape. One of our data scientists has been asking about this so I want to make sure it's possible.

Thanks!

numpy\_assignments.ipynb

Reply

Forward

## Solution Code

```
my_array = np.linspace(10, 100, 10).reshape(5, 2)
```

```
my_array
```

```
array([[ 10.,  20.],
       [ 30.,  40.],
       [ 50.,  60.],
       [ 70.,  80.],
       [ 90., 100.]])
```

```
from numpy.random import default_rng
```

```
rng = default_rng(2022)
```

```
random_array = rng.random(9).reshape(3, 3)
```

```
random_array
```

```
array([[0.24742606, 0.09299006, 0.61176337],
       [0.06066207, 0.66103343, 0.75515778],
       [0.1108689 , 0.04305584, 0.41441747]])
```



# INDEXING & SLICING ARRAYS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

**Indexing & slicing** one-dimensional arrays is the same as base Python

- **array[index]** – indexing to access a single element (*0-indexed*)
- **array[start:stop:step size]** – slicing to access a series of elements (*stop is not inclusive*)

```
product_array
```

```
array(['fruits', 'vegetables', 'cereal', 'dairy', 'eggs', 'snacks',
       'beverages', 'coffee', 'tea', 'spices'], dtype='<U10')
```

```
print(product_array[1])
print(product_array[-1])
```

```
vegetables
spices
```

```
product_array[:5]
```

```
array(['fruits', 'vegetables', 'cereal', 'dairy', 'eggs'], dtype='<U10')
```

```
product_array[5::2]
```

```
array(['snacks', 'coffee', 'spices'], dtype='<U10')
```

This grabs the **second** and **last** elements of product\_array

This grabs the **first five** elements of product\_array

This starts at the **sixth** element and grabs **every other** element **until the end** of product\_array



# INDEXING & SLICING ARRAYS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

**Indexing & slicing** two-dimensional arrays requires an extra index or slice

- **array[row index, column index]** – indexing to access a single element (0-indexed)
- **array[start:stop:step size, start:stop:step size]** – slicing to access a series of elements

```
product_array2D = product_array.reshape(2, 5)
product_array2D
```

```
array([['fruits', 'vegetables', 'cereal', 'dairy', 'eggs'],
       ['snacks', 'beverages', 'coffee', 'tea', 'spices']], dtype='<U10')
```

```
product_array2D[1, 2]
```

```
'coffee'
```

```
product_array2D[:, 2:]
```

```
array([['cereal', 'dairy', 'eggs'],
       ['coffee', 'tea', 'spices']], dtype='<U10')
```

```
product_array2D[1:, :]
```

```
array([['snacks', 'beverages', 'coffee', 'tea', 'spices']], dtype='<U10')
```

This goes to the **second** row  
and grabs the **third** element

This goes to **all** rows and grabs  
all the elements starting from  
the **third** in each row

This goes to the **second** row  
and grabs **all** its elements

# ASSIGNMENT: ARRAY ACCESS

  NEW MESSAGE

June 17, 2022

From: Ross Retail (Head of Analytics)

Subject: Indexing & Slicing Arrays

Ok, last ‘theoretical’ exercise before we start working with real data.

I am familiar with indexing and slicing in base Python but have no idea how it works in multiple dimensions.

I’ve provided a few different ‘cuts’ of the data in the notebook – can you slice and dice the random array we created in the last exercise?

Thanks!

 section01\_Numpy.ipynb

## Results Preview

```
array([[0.24742606, 0.09299006, 0.61176337],  
       [0.06066207, 0.66103343, 0.75515778],  
       [0.1108689 , 0.04305584, 0.41441747]])
```

**First two ‘rows’**

```
array([[0.24742606, 0.09299006, 0.61176337],  
       [0.06066207, 0.66103343, 0.75515778]])
```

**First ‘column’**

```
array([0.24742606, 0.06066207, 0.1108689 ])
```

**Second number in third ‘row’:**

```
0.04305584439252108
```

# ASSIGNMENT: ARRAY ACCESS

  NEW MESSAGE

June 17, 2022

From: Ross Retail (Head of Analytics)

Subject: Indexing & Slicing Arrays

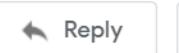
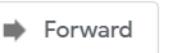
Ok, last ‘theoretical’ exercise before we start working with real data.

I am familiar with indexing and slicing in base Python but have no idea how it works in multiple dimensions.

I’ve provided a few different ‘cuts’ of the data in the notebook – can you slice and dice the random array we created in the last exercise?

Thanks!

 section01\_Numpy.ipynb

## Results Preview

```
array([[0.24742606, 0.09299006, 0.61176337],  
       [0.06066207, 0.66103343, 0.75515778],  
       [0.1108689 , 0.04305584, 0.41441747]])
```

*First two ‘rows’*

```
random_array[:2]
```

```
array([[0.24742606, 0.09299006, 0.61176337],  
       [0.06066207, 0.66103343, 0.75515778]])
```

*First ‘column’*

```
random_array[:, 0]
```

```
array([0.24742606, 0.06066207, 0.1108689 ])
```

*Second number in third ‘row’:*

```
random_array[2, 1]
```

```
0.04305584439252108
```



# ARRAY OPERATIONS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

Arithmetic operators can be used to perform **array operations**

```
sales = [[0, 5, 155, 0, 518], [0, 1827, 616, 317, 325]]  
  
sales_array = np.array(sales)  
sales_array
```

```
array([[ 0, 5, 155, 0, 518],  
       [ 0, 1827, 616, 317, 325]])
```

```
sales_array + 2
```

```
array([[ 2, 7, 157, 2, 520],  
       [ 2, 1829, 618, 319, 327]])
```

```
quantity = sales_array[0, :]  
price = sales_array[1, :]
```

```
quantity * price
```

```
array([ 0, 9135, 95480, 0, 168350])
```



Array operations are applied via **vectorization** and **broadcasting**, which eliminates the need to loop through the array's elements

This **adds 2** to every element in the array

This assigns **all** the elements in the **first row** to 'quantity'  
Then assigns **all** the elements in the **second row** to 'price'  
Finally, it multiplies the corresponding elements in each array:  
(0\*0, 5\*1827, 155\*616, 0\*317, and 518\*325)

# ASSIGNMENT: ARRAY OPERATIONS



## NEW MESSAGE

June 18, 2022

From: **Ross Retail** (Head of Analytics)  
Subject: Random Discount

Ok, so now that we've gotten the basics down, we can start using NumPy for our first tasks. As part of a promotion, we want to apply a random discount to surprise our customers and generate social media buzz.

First, add a flat shipping cost of 5 to our prices to get the 'total' amount owed.

The numbers in the random array represent 'discount percent'. To get the 'percent owed', subtract the first 6 numbers in the random array from 1, then multiply 'percent owed' by 'total' to get the final amount owed.

numpy\_assignments.ipynb

Reply

Forward

## Results Preview

```
prices = np.array([5.99, 6.99, 22.49, 99.99, 4.99, 49.99])
```

```
total
```

```
array([ 10.99, 11.99, 27.49, 104.99, 9.99, 54.99])
```

```
print(discount_pct)
print(pct_owed)
print(final_owed.round(2))
```

```
[0.24742606 0.09299006 0.61176337 0.06066207 0.66103343 0.75515778]
[0.75257394 0.90700994 0.38823663 0.93933793 0.33896657 0.24484222]
[ 8.27 10.88 10.67 98.62  3.39 13.46]
```



The `.round()` array method rounds the elements of the array to the specified decimals

# SOLUTION: ARRAY OPERATIONS



## NEW MESSAGE

June 18, 2022

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

## Subject: Random Discount

Ok, so now that we've gotten the basics down, we can start using NumPy for our first tasks. As part of a promotion, we want to apply a random discount to surprise our customers and generate social media buzz.

First, add a flat shipping cost of 5 to our prices to get the ‘total’ amount owed.

The numbers in the random array represent ‘discount percent’. To get the ‘percent owed’, subtract the first 6 numbers in the random array from 1, then multiply ‘percent owed’ by ‘total’ to get the final amount owed.



*numpy assignments.ipynb*



# **Solution Code**

```
prices = np.array([5.99, 6.99, 22.49, 99.99, 4.99, 49.99])  
  
total = prices + 5  
  
total  
  
array([ 10.99,  11.99,  27.49, 104.99,   9.99,  54.99])
```

```
discount_pct = random_array[:2].reshape(6)

pct_owed = 1 - discount_pct

final_owed = total * pct_owed

final_owed.round(2)
```

The `.round()` array method rounds the elements of the array to the specified decimals



# FILTERING ARRAYS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

You can **filter arrays** by indexing them with a logical test

- Only the array elements in positions where the logical test returns True are returned

```
sales_array
```

```
array([[ 0,    5,   155,    0,   518],  
       [ 0, 1827,   616,   317,   325]])
```

```
sales_array != 0
```

```
array([[False,  True,  True, False,  True],  
       [False,  True,  True,  True,  True]])
```

```
sales_array[sales_array != 0]
```

```
array([ 5, 155, 518, 1827, 616, 317, 325])
```

Performing a logical test on a NumPy array returns a **Boolean array** with the results of the logical test on each array element

Indexing an array with a Boolean array returns an array with the elements where the Boolean value is **True**



# FILTERING ARRAYS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

You can filter arrays with **multiple logical tests**

- Use `|` for **or** conditions and `&` for **and** conditions

```
sales_array
```

```
array([[ 0,    5,  155,    0,  518],  
       [ 0, 1827,   616,  317,  325]])
```

```
sales_array[(sales_array == 616) | (sales_array < 100)]
```

```
array([ 0,    5,    0,    0,  616])
```

```
sales_array[(sales_array > 100) & (sales_array < 500)]
```

```
array([155, 317, 325])
```

```
mask = (sales_array > 100) & (sales_array < 500)
```

```
sales_array[mask]
```

This returns an array with elements equal to 616 **or** less than 100

This returns an array with elements greater than 100 **and** less than 500



**PRO TIP:** Store complex filtering criteria in a variable (known as a Boolean mask)



# FILTERING ARRAYS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

You can filter arrays based on **values in other arrays**

- Use the Boolean array returned from the other array to index the array you want to filter

```
sales_array
```

```
array([  0,    5, 155,    0, 518])
```

```
product_array
```

```
array(['fruits', 'vegetables', 'cereal', 'dairy', 'eggs'], dtype='<U10')
```

```
product_array[sales_array > 0]
```

```
array(['vegetables', 'cereal', 'eggs'], dtype='<U10')
```



This returns the elements from `product_array` where values in `sales_array` are greater than 0



# MODIFYING ARRAY VALUES

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

You can **modify array values** by assigning new ones

```
sales_array
```

```
array([ 0,  5, 155,  0, 518])
```

```
sales_array[1] = 25
```

```
sales_array
```

```
array([ 0, 25, 155,  0, 518])
```

```
sales_array[sales_array == 0] = 5
```

```
sales_array
```

```
array([ 5, 25, 155,  5, 518])
```

This assigns a single value via **indexing**

This **filters** the zero values in sales\_array and assigns them a new value of 5



# THE WHERE FUNCTION

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

The **where()** NumPy function performs a logical test and returns a given value if the test is True, or another if the test is False

```
np.where(logical test,  
         value if True,  
         value if False)
```

Calls the NumPy  
function library

A logical expression that  
evaluates to True or False

Value to return when  
the expression is True

Value to return when  
the expression is False



# THE WHERE FUNCTION

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

The **where()** NumPy function performs a logical test and returns a given value if the test is True, or another if the test is False

```
inventory_array
```

```
array([ 12, 102, 18, 0, 0])
```

```
product_array
```

```
array(['fruits', 'vegetables', 'cereal', 'dairy', 'eggs'], dtype='<U10')
```

```
np.where(inventory_array <= 0, "Out of Stock", "In Stock")
```

```
array(['In Stock', 'In Stock', 'In Stock', 'Out of Stock', 'Out of Stock'],  
      dtype='<U12')
```

```
np.where(inventory_array <= 0, "Out of Stock", product_array)
```

```
array(['fruits', 'vegetables', 'cereal', 'Out of Stock', 'Out of Stock'],  
      dtype='<U12')
```

If *inventory* is zero or negative,  
assign 'Out of Stock', otherwise  
assign 'In Stock'

If *inventory* is zero or negative,  
assign 'Out of Stock', otherwise  
assign the *product\_array* value

# ASSIGNMENT: FILTERING ARRAYS



NEW MESSAGE

June 19, 2022

From: **Ross Retail** (Head of Analytics)  
Subject: Subsetting Our Data

Hey there,

We're working on some more promotions. Can you filter our product list to only include prices greater than 25?

Once you've done that, modify your logic to force cola into the list. Call this array 'fancy\_feast\_special'.

Finally, we need to modify our shipping logic. Create a new shipping cost array, but this time if price is greater than 20, shipping cost is 0, otherwise shipping cost is 5.

Thanks!

numpy\_assignments.ipynb

Reply

Forward

## Results Preview

```
products[ ... ]
```

```
array(['rare tomato', 'gourmet ice cream'], dtype='<U17')
```

```
fancy_feast_special
```

```
array(['rare tomato', 'cola', 'gourmet ice cream'], dtype='<U17')
```

```
shipping_cost
```

```
array([5, 5, 0, 0, 5, 0])
```

# SOLUTION: FILTERING ARRAYS



NEW MESSAGE

June 19, 2022

From: **Ross Retail** (Head of Analytics)  
Subject: Subsetting Our Data

Hey there,

We're working on some more promotions. Can you filter our product list to only include prices greater than 25?

Once you've done that, modify your logic to force cola into the list. Call this array 'fancy\_feast\_special'.

Finally, we need to modify our shipping logic. Create a new shipping cost array, but this time if price is greater than 20, shipping cost is 0, otherwise shipping cost is 5.

Thanks!

numpy\_assignments.ipynb

Reply

Forward

## Solution Code

```
products[prices > 25]
array(['rare tomato', 'gourmet ice cream'], dtype='<U17')

mask = (prices > 25) | (products == "cola")
fancy_feast_special = products[mask]
fancy_feast_special
array(['rare tomato', 'cola', 'gourmet ice cream'], dtype='<U17')

shipping_cost = np.where(prices > 20, 0, 5)
shipping_cost
array([5, 5, 0, 0, 5, 0])
```



# ARRAY AGGREGATION METHODS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

**Array aggregation methods** let you calculate metrics like sum, mean, and max

`sales_array`

```
array([[ 0,  5, 155,  0, 518],  
       [ 0, 1827, 616, 317, 325]])
```

**array.sum()** Returns the sum of all values in an array

`sales_array.sum()`

3763

**array.mean()** Returns the average of the values in an array

`sales_array.mean()`

376.3

**array.max()** Returns the largest value in an array

`sales_array.max()`

1827

**array.min()** Returns the smallest value in an array

`sales_array.min()`

0



# ARRAY AGGREGATION METHODS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

You can also aggregate across **rows** or **columns**

`sales_array`

```
array([[ 0,    5,   155,    0,   518],  
       [ 0, 1827,   616,   317,   325]])
```

`array.sum()` Returns the sum of all values in an array

`sales_array.sum()`

3763

`array.sum(axis=0)` Aggregates across rows

`sales_array.sum(axis=0)`

```
array([ 0, 1832, 771, 317, 84
```

`array.sum(axis=1)` Aggregates across columns

`sales_array.sum(axis=1)`

```
array([ 678, 3085])
```





# ARRAY FUNCTIONS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

**Array functions** let you perform other aggregations like median and percentiles

`sales_array`

```
array([[ 0,    5,   155,    0,   518],  
       [ 0, 1827,   616,   317,   325]])
```

`np.median(array)` Returns the median value in an array

`np.median(sales_array)`

236.0

`np.percentile(array, n)` Returns a value in the  $n^{\text{th}}$  percentile in an array

`np.percentile(sales_array, 90)`

737.0999999999996 ← This uses linear interpolation by default



# ARRAY FUNCTIONS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

You can also return a **unique** list of values or the **square root** for each number

`sales_array`

```
array([[ 0,    5,   155,    0,   518],  
       [ 0, 1827,   616,   317,   325]])
```

`np.unique(array)` *Returns the unique values in an array*

`np.unique(sales_array)`

```
array([ 0,    5,   155,   317,   325,   518,   616, 1827])
```

`np.sqrt(array)` *Returns the square root of each value in an array*

`np.sqrt(sales_array)`

```
array([[ 0.          ,  2.23606798, 12.4498996 ,  0.          , 22.75961335],  
       [ 0.          , 42.74342055, 24.81934729, 17.80449381, 18.02775638]])
```



# SORTING ARRAYS

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

The sort() method will **sort arrays** in place

- Use the axis argument to specify the dimension to sort by

```
sales_array
```

```
array([[ 0,  5, 155,  0, 518],  
       [ 0, 1827, 616, 317, 325]])
```

```
sales_array.sort()
```

```
sales_array
```

```
array([[ 0,  0,  5, 155, 518],  
       [ 0, 317, 325, 616, 1827]])
```

```
sales_array.sort(axis=0)
```

```
sales_array
```

```
array([[ 0,  5, 155,  0, 325],  
       [ 0, 1827, 616, 317, 518]])
```

} *axis=1 by default, which sorts a two-dimensional array row by row*

} *axis=0 will sort by columns*

# ASSIGNMENT: SORTING AND AGGREGATING

  NEW MESSAGE  
June 19, 2022

**From:** Ross Retail (Head of Analytics)  
**Subject:** Top Tier Products

Hey there,

Thanks for all your hard work. I know we're working with small sample sizes, but we're proving that analysis can be done in Python!

Can you calculate the mean, min, max, and median of our 3 most expensive product prices? Sorting the array first should help!

Then, calculate the number of unique price tiers we have.

Thanks!

 [numpy\\_assignments.ipynb](#)

## Results Preview

### Top 3 Price Stats

Mean: 40.98667062008267  
Min: 10.875049160510919  
Max: 98.62108889965567  
Median: 13.46387380008141

### Unique Price Tiers



3

# SOLUTION: SORTING AND AGGREGATING

  NEW MESSAGE

June 19, 2022

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

**Subject:** Top Tier Products

Hey there,

Thanks for all your hard work. I know we're working with small sample sizes, but we're proving that analysis can be done in Python!

Can you calculate the mean, min, max, and median of our 3 most expensive product prices? Sorting the array first should help!

Then, calculate the number of unique price tiers we have.

Thanks!

 `numpy_assignments.ipynb`

## Solution Code

### Top 3 Price Stats

```
Mean: 40.98667062008267  
Min: 10.875049160510919  
Max: 98.62108889965567  
Median: 13.46387380008141
```

### Unique Price Tiers

```
np.unique(products).size
```

3



# VECTORIZATION

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

**Vectorization** is the process of pushing array operations into optimized C code, which is easier and more efficient than writing for loops

```
def for_loop_multiply_lists(list1, list2):
    product_list = []
    for element1, element2 in zip(tuple1, tuple2):
        product_list.append(element1 * element2)
    return product_list
```

} Function that multiplies two Python **lists**

```
def multiply_arrays(array1, array2):
    return array1 * array2
```

} Function that multiplies two NumPy **arrays**

```
list1 = list(range(1000))
list2 = list(range(1000))
```

} Generating and multiplying two lists

```
%%timeit -r 5 -n 10000
for_loop_multiply_lists(list1, list2)
```

```
75.8 µs ± 2 µs per loop (mean ± std. dev. of 5 runs, 10000 loops each)
```

```
array1 = np.array(list1)
array2 = np.array(list2)
```

```
%%timeit -r 5 -n 10000
multiply_arrays(array1, array2)
```

```
876 ns ± 44.7 ns per loop (mean ± std. dev. of 5 runs, 10000 loops each)
```

} Converting and multiplying two arrays  
**~86 times faster!**



# VECTORIZATION

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

**Vectorization** is the process of pushing array operations into optimized C code, which is easier and more efficient than writing for loops

```
def for_loop_multiply_lists(list1, list2):
    product_list = []
    for element1, element2 in zip(tuple1, tuple2):
        product_list.append(element1 * element2)
    return product_list

def multiply_arrays(array1, array2):
    return array1 * array2

list1 = list(range(1000))
list2 = list(range(1000))

%%timeit -r 5 -n 10000
for_loop_multiply_lists(list1, list2)

75.8 µs ± 2 µs per loop (mean ± std. dev. of 5 runs, 10000 loops each)

array1 = np.array(list1)
array2 = np.array(list2)

%%timeit -r 5 -n 10000
multiply_arrays(array1, array2)

876 ns ± 44.7 ns per loop (mean ± std. dev. of 5 runs, 10000 loops each)
```



**PRO TIP:** Use vectorized operations whenever possible when manipulating data, and avoid writing loops

Converting and multiplying two arrays  
**~86 times faster!**



# BROADCASTING

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

**Broadcasting** lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to ‘fit’ the larger one

- Single values (scalars) can be broadcast into arrays of any dimension

```
test_array = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
```

```
test_array
```

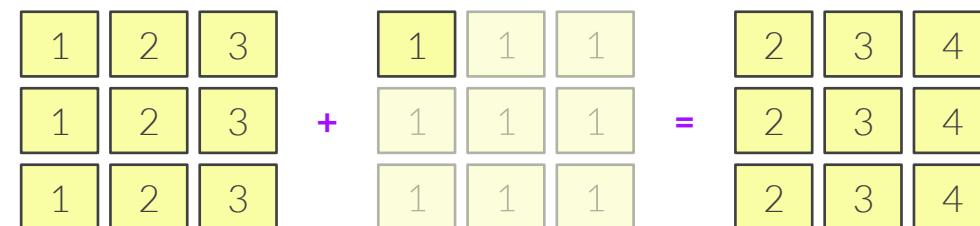
```
array([[1, 2, 3],  
       [1, 2, 3],  
       [1, 2, 3]])
```

```
test_array + 1
```

```
array([[2, 3, 4],  
       [2, 3, 4],  
       [2, 3, 4]])
```



How does this code work?





# BROADCASTING

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

**Broadcasting** lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to ‘fit’ the larger one

- Single values (scalars) can be broadcast into arrays of any dimension
- Dimensions with a length greater than one must be the same size

```
test_array = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
```

```
test_array
```

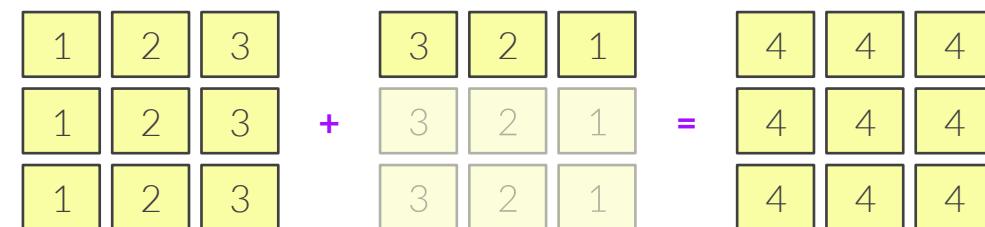
```
array([[1, 2, 3],  
       [1, 2, 3],  
       [1, 2, 3]])
```

```
test_array + np.array([3, 2, 1])
```

```
array([[4, 4, 4],  
       [4, 4, 4],  
       [4, 4, 4]])
```



How does this code work?





# BROADCASTING

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

**Broadcasting** lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to ‘fit’ the larger one

- Single values (scalars) can be broadcast into arrays of any dimension
- Dimensions with a length greater than one must be the same size

```
test_array = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
```

```
test_array
```

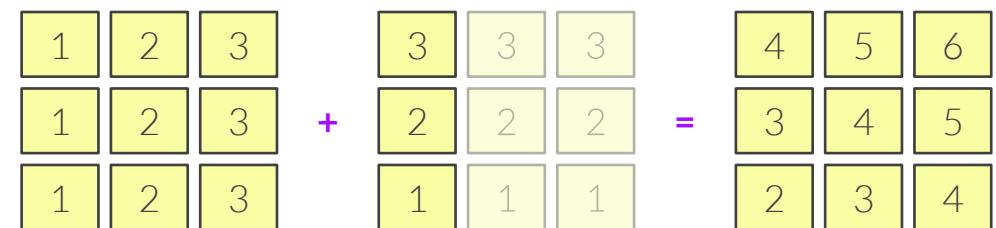
```
array([[1, 2, 3],  
       [1, 2, 3],  
       [1, 2, 3]])
```

```
test_array + np.array([3, 2, 1]).reshape(3, 1)
```

```
array([[4, 5, 6],  
       [3, 4, 5],  
       [2, 3, 4]])
```



How does this code work?





# BROADCASTING

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

**Broadcasting** lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to 'fit' the larger one

- Single values (scalars) can be broadcast into arrays of any dimension
- Dimensions with a length greater than one must be the same size

```
test_array = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
```

```
test_array
```

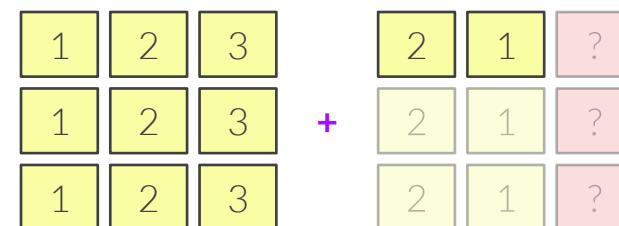
```
array([[1, 2, 3],  
       [1, 2, 3],  
       [1, 2, 3]])
```

```
test_array + np.array([2, 1])
```

```
ValueError: operands could not be broadcast  
together with shapes (3,3) (2,)
```



How does this code work?





# BROADCASTING

Intro to Pandas & NumPy

NumPy Array Basics

Array Creation

Array Indexing & Slicing

Array Operations

Vectorization & Broadcasting

**Broadcasting** lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to ‘fit’ the larger one

- Single values (scalars) can be broadcast into arrays of any dimension
- Dimensions with a length greater than one must be the same size

```
test_array = np.array([[1, 2, 3], [1, 2, 3], [1, 2, 3]])
```

```
test_array
```

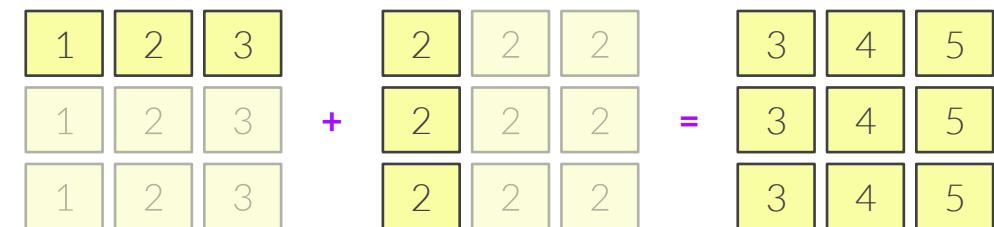
```
array([[1, 2, 3],  
       [1, 2, 3],  
       [1, 2, 3]])
```

```
test_array[0, :] + test_array[:, 1].reshape(3, 1)
```

```
array([[3, 4, 5],  
       [3, 4, 5],  
       [3, 4, 5]])
```



How does this code work?





# BROADCASTING

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

**Broadcasting** lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to ‘fit’ the larger one

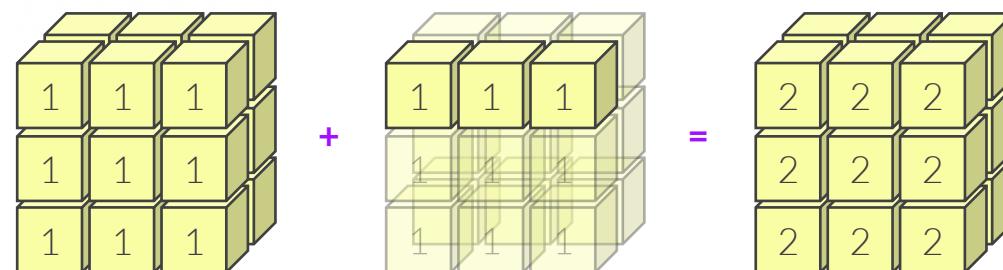
- Single values (scalars) can be broadcast into arrays of any dimension
- Dimensions with a length greater than one must be the same size

```
np.ones((2, 3, 3), dtype=int) + np.ones(3, dtype=int)
```

```
array([[[2, 2, 2],  
        [2, 2, 2],  
        [2, 2, 2]],  
  
       [[2, 2, 2],  
        [2, 2, 2],  
        [2, 2, 2]])
```



How does this code work?





# BROADCASTING

Intro to Pandas  
& NumPy

NumPy Array  
Basics

Array Creation

Array Indexing  
& Slicing

Array Operations

Vectorization &  
Broadcasting

**Broadcasting** lets you perform vectorized operations with arrays of different sizes, where NumPy will expand the smaller array to ‘fit’ the larger one

- Single values (scalars) can be broadcast into arrays of any dimension
- Dimensions with a length greater than one must be the same size



**Compatible** shapes:

Array 1	Array 2	Output
(3, 3)	(100, 3, 3)	(100, 3, 3)
(3, 1, 5, 1)	(4, 1, 6)	(3, 4, 5, 6)



**Incompatible** shapes:

Array 1	Array 2
(4, 3)	(100, 3, 3)
(3, 1, 5, 1)	(4, 2, 6)

# ASSIGNMENT: BRINGING IT ALL TOGETHER



## NEW MESSAGE

June 20, 2022

From: **Ross Retail** (Head of Analytics)  
Subject: Final NumPy Challenge

Alright, our new data scientist set up a little test case for us. She provided code to read in data from a csv and convert two columns to arrays.

Filter 'sales\_array' to only include sales that had the product family 'PRODUCE' in the 'family\_array'. Call this produce\_sales.

Then randomly sample half of the remaining sales and calculate the mean of those sales.

Thanks!

Reply

Forward

numpy\_assignments.ipynb

## Results Preview

```
import pandas as pd  
  
retail_df = pd.read_csv("../retail/retail.csv").sample(1000, random_state=100)  
  
family_array = np.array(retail_df["family"])  
sales_array = np.array(retail_df["sales"])
```

1556.4381428571428

# SOLUTION: BRINGING IT ALL TOGETHER



## NEW MESSAGE

June 20, 2022

From: **Ross Retail** (Head of Analytics)  
Subject: Final NumPy Challenge

Alright, our new data scientist set up a little test case for us. She provided code to read in data from a csv and convert two columns to arrays.

Filter 'sales\_array' to only include sales that had the product family 'PRODUCE' in the 'family\_array'. Call this produce\_sales.

Then randomly sample half of the remaining sales and calculate the mean of those sales.

Thanks!

numpy\_assignments.ipynb

Reply

Forward

## Solution Code

```
import pandas as pd  
  
retail_df = pd.read_csv("../retail/retail.csv").sample(1000, random_state=100)  
  
family_array = np.array(retail_df["family"])  
sales_array = np.array(retail_df["sales"])  
  
produce_sales = sales_array[family_array == "PRODUCE"]  
  
produce_sales[rng.random(30) < 0.5].mean()  
  
1556.4381428571428
```

# KEY TAKEAWAYS

---



NumPy forms the **foundation for Pandas**

- As an analyst, it's important to be comfortable working with NumPy arrays & functions in order to use Pandas data structures & functions properly



NumPy arrays are **more efficient** than base Python lists and tuples

- They are semi-mutable data structures capable of storing many data types
- Their values can be modified, but their size cannot



Array operations let you **aggregate, filter, and sort** data

- Broadcasting and vectorization make these operations convenient and efficient without the use of loops
- The syntax for NumPy array operations is very similar to Pandas

# PANDAS SERIES

# SERIES



In this section we'll introduce Pandas **Series**, the Python equivalent of a column of data, and cover their basic properties, creation, manipulation, and useful functions for analysis

## TOPICS WE'LL COVER:

Pandas Series Basics

Series Indexing

Sorting & Filtering

Operations & Aggregations

Handling Missing Data

Applying Custom Functions

## GOALS FOR THIS SECTION:

- Understand the relationship between Pandas Series and NumPy arrays
- Use the `.loc()` and `.iloc()` methods to access Series data by their indices or values
- Learn to sort, filter, and aggregate Pandas Series using methods and functions
- Apply custom functions using conditional logic to Pandas Series



# PANDAS SERIES

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

**Series** are Pandas data structures built on top of NumPy arrays

- Series also contain an **index** and an **optional name**, in addition to the array of data
- They can be created from other data types, but are usually imported from external sources
- Two or more Series grouped together form a Pandas DataFrame

```
import numpy as np
import pandas as pd
```

```
sales = [0, 5, 155, 0, 518, 0, 1827, 616, 317, 325]
sales_series = pd.Series(sales, name="Sales")
sales_series
```

The index is an array  
of integers starting at  
0 by default, but it can  
be modified

0	0
1	5
2	155
3	0
4	518
5	0
6	1827
7	616
8	317
9	325

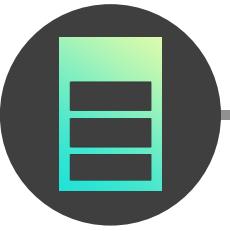
Name: Sales, dtype: int64

'pd' is the standard alias for the Pandas library

Pandas' Series function converts Python lists  
and NumPy arrays into Pandas Series

The name argument lets you specify a name

The series name and data type are stored as well



# SERIES PROPERTIES

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

Pandas Series have these key properties:

- **values** – the data array in the Series
- **index** – the index array in the Series
- **name** – the optional name for the Series (*useful for accessing columns in a DataFrame*)
- **dtype** – the data type of the elements in the values array

`sales_series.values`

`array([ 0, 5, 155, 0, 518, 0, 1827, 616, 317, 325])`

The index is a  
range of integers  
from 0 to 10

`sales_series.index`

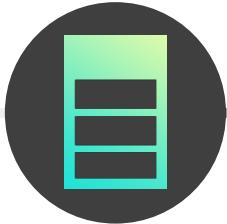
`RangeIndex(start=0, stop=10, step=1)`

`sales_series.name`

`'Sales'`

`sales_series.dtype`

`dtype('int64')`



# PANDAS DATA TYPES

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

**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



We'll review the nuances of data types  
in depth when covering **DataFrames**

## 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
datetime64	A single moment in time (January 4, 2015, 2:00:00 PM)
timedelta64	The duration between two dates or times (10 days, 3 seconds, etc.)
period	A span of time (a day, a week, etc.)

# TYPE CONVERSION



Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

You can **convert the data type** in a Pandas Series by using the `.astype()` method and specifying the desired data type (if compatible)

`sales_series`

0	0
1	5
2	155
3	0
4	518

These are integers

Name: Sales, dtype: int64

`sales_series.astype("float")`

0	0.0
1	5.0
2	155.0
3	0.0
4	518.0

This converts them to floats

Name: Sales, dtype: float64

`sales_series.astype("bool")`

0	False
1	True
2	True
3	False
4	True

This converts them to Booleans  
(0 is False, others are True)

Name: Sales, dtype: bool

`sales_series.astype("datetime64")`

This attempts to convert them to the Datetime datatype, but isn't compatible

`ValueError: The 'datetime64' dtype has no unit.`

# ASSIGNMENT: SERIES BASICS



**NEW MESSAGE**

June 21, 2022

**From:** **Rachel Revenue** (Financial Analyst)

**Subject:** Oil Price Series

Hi there, glad to have you on the team!

I work in the finance department, and I'm working on an analysis on the impact of oil prices on our sales. Our last analyst read in oil data and created a NumPy array, can you convert that to a Pandas Series and report back on properties of the Series?

Make sure to include name, dtype, size, index, then take the mean of the values array. Finally, convert the series to an integer data type and recalculate the mean.

Thanks!

[Reply](#) [Forward](#)

## Results Preview

`oil_series`

```
0      52.22  
1      51.44  
2      51.98  
3      52.01  
4      52.82  
...  
95     45.84  
96     47.28  
97     47.81  
98     47.83  
99     48.86
```

`Name: oil_prices`

`dtype: float64`

`size: 100`

`index: RangeIndex(start=0, stop=100, step=1)`

`51.128299999999996`

`50.66`

# SOLUTION: SERIES BASICS



**NEW MESSAGE**

June 21, 2022

**From:** **Rachel Revenue** (Financial Analyst)

**Subject:** Oil Price Series

Hi there, glad to have you on the team!

I work in the finance department, and I'm working on an analysis on the impact of oil prices on our sales. Our last analyst read in oil data and created a NumPy array, can you convert that to a Pandas Series and report back on properties of the Series?

Make sure to include name, dtype, size, index, then take the mean of the values array. Finally, convert the series to an integer data type and recalculate the mean.

Thanks!

[Reply](#) [Forward](#)

## Solution Code

```
oil_series
```

```
0      52.22  
1      51.44  
2      51.98  
3      52.01  
4      52.82  
...  
95     45.84  
96     47.28  
97     47.81  
98     47.83  
99     48.86
```

```
Name: oil_prices
```

```
dtype: float64
```

```
size: 100
```

```
index: RangeIndex(start=0, stop=100, step=1)
```

```
oil_series.values.mean()
```

```
51.128299999999996
```

```
oil_series.astype("int").values.mean()
```

```
50.66
```



# THE INDEX

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

The **index** lets you easily access “rows” in a Pandas Series or DataFrame

```
sales = [0, 5, 155, 0, 518]
sales_series = pd.Series(sales, name="Sales")
sales_series
```

Here we're using the  
default integer index,  
which is preferred

```
{ 0      0
 1      5
 2    155
 3      0
 4    518
Name: Sales, dtype: int64
```

```
sales_series[2]
```

155

```
sales_series[2:4]
```

```
2    155
3      0
```

```
Name: Sales, dtype: int64
```

You can **index** and **slice** Series like  
other sequence data types, but we'll  
learn a better method

# CUSTOM INDICES



Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

There are cases where it's applicable to use a **custom index** for accessing rows

```
sales = [0, 5, 155, 0, 518]
items = ["coffee", "bananas", "tea", "coconut", "sugar"]

sales_series = pd.Series(sales, index=items, name="Sales")
```

```
sales_series
```

coffee	0
bananas	5
tea	155
coconut	0
sugar	518

Name: Sales, dtype: int64

Custom indices can be assigned when  
*creating the series* or by assignment

```
sales_series.index = ["coffee", "bananas", "tea", "coconut", "sugar"]
```

sales\_series

coffee	0
bananas	5
tea	155
coconut	0
sugar	518

Name: Sales, dtype: int64



This will become more relevant  
when working with **datetimes**  
(covered later in the course!)



# CUSTOM INDICES

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

There are cases where it's applicable to use a **custom index** for accessing rows

```
sales = [0, 5, 155, 0, 518]
items = ["coffee", "bananas", "tea", "coconut", "sugar"]

sales_series = pd.Series(sales, index=items, name="Sales")

sales_series
```

```
coffee      0
bananas     5
tea         155
coconut     0
sugar       518
Name: Sales, dtype: int64
```

```
sales_series["tea"]
```

```
155
```

```
sales_series["bananas":"coconut"]
```

```
bananas     5
tea         155
coconut     0
Name: Sales, dtype: int64
```

Note that slicing custom indices  
makes the stop point **inclusive**

You can still **index** and **slice**  
to retrieve Series values using  
the custom indices



# THE ILOC METHOD

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

The `.iloc[]` method is the preferred way to access values by their positional index

- This method works even when Series have a custom, non-integer index
- It is more efficient than slicing and is recommended by Pandas' creators

`df.iloc[row position, column position]`

Series or DataFrame  
to access values from

The row position(s) for the  
value(s) you want to access

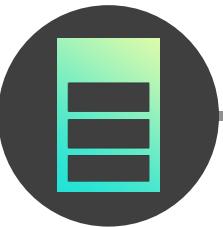
The column position(s) for the  
value(s) you want to access

Examples:

- `0` (single row)
- `[5, 9]` (multiple rows)
- `[0:11]` (range of rows)



We'll use the column position  
argument once we start working  
with Pandas **DataFrames**



# THE ILOC METHOD

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

The `.iloc[]` method is the preferred way to access values by their positional index

- This method works even on Series with a custom, non-integer index
- It is more efficient than slicing and is recommended by Pandas' creators

Note that this Series  
has a custom index

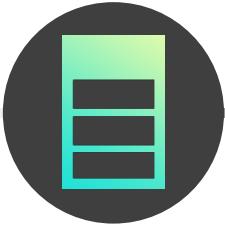
```
sales_series
coffee      0
bananas     5
tea        155
coconut      0
sugar       518
Name: Sales, dtype: int64
```

```
sales_series.iloc[2]
155
```

This returns the value in the 3<sup>rd</sup> position (0-indexed), even  
though the custom index for that value is "tea"

```
sales_series.iloc[2:4]
tea        155
coconut      0
Name: Sales, dtype: int64
```

This returns the values from the 3<sup>rd</sup> to the 4<sup>th</sup> position (stop  
is non-inclusive)



# THE LOC METHOD

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

The `.loc[]` method is the preferred way to access values by their custom labels

`df.loc[row_label, column_label]`

Series or DataFrame  
to access values from

The custom row index for the  
value(s) you want to access

The custom column index for  
the value(s) you want to access

## Examples:

- `"pizza"` (single row)
- `["mike", "ike"]` (multiple rows)
- `["jan":"dec"]` (range of rows)

# THE LOC METHOD



Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

The `.loc[]` method is the preferred way to access values by their custom labels

*The custom indices  
are the labels*

```
sales_series
coffee      0
bananas     5
tea        155
coconut      0
sugar       518
Name: Sales, dtype: int64
```

```
sales_series.loc["tea"]
```

155

```
sales_series.loc["bananas":"coconut"]
```

```
bananas     5
tea        155
coconut      0
Name: Sales, dtype: int64
```

Note that slices are **inclusive**  
when using custom labels



The `.loc[]` method works even  
when the indices are integers,  
but if they are custom integers  
not ordered from 0 to n-1, the  
rows will be returned based on  
the **labels** themselves and NOT  
their numeric position



# DUPLICATE INDEX VALUES

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

It is possible to have **duplicate index values** in a Pandas Series or DataFrame

- Accessing these indices by their label using `.loc[]` returns all corresponding rows

```
sales = [0, 5, 155, 0, 518]
items = ["coffee", "coffee", "tea", "coconut", "sugar"]

sales_series = pd.Series(sales, index=items, name="Sales")
```

```
sales_series
```

coffee	0
coffee	5
tea	155
coconut	0
sugar	518

Name: Sales, dtype: int64

```
sales_series.loc["coffee"]
```

coffee	0
coffee	5

Name: Sales, dtype: int64

This returns both rows with the same label



Warning! Duplicate index values are **generally not advised**, but there are some edge cases where they are useful



# RESETTING THE INDEX

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

You can **reset the index** in a Pandas Series or DataFrame back to the default range of integers by using the `.reset_index()` method

- By default, the existing index will become a new column in a DataFrame

`sales_series`

```
coffee      0
coffee      5
tea       155
coconut     0
sugar      518
Name: Sales, dtype: int64
```

`sales_series.reset_index()`

	index	Sales
0	coffee	0
1	coffee	5
2	tea	155
3	coconut	0
4	sugar	518

This returns a DataFrame by default, with the previous index values stored as a new column

`sales_series.reset_index(drop=True)`

```
0      0
1      5
2    155
3      0
4    518
Name: Sales, dtype: int64
```

Use `drop=True` when resetting the index if you don't want the previous index values stored

# ASSIGNMENT: ACCESSING SERIES DATA

 NEW MESSAGE  
June 21, 2022

**From:** Rachel Revenue (Financial Analyst)  
**Subject:** Oil Price Series w/Dates

Thanks for picking up this work, but this data isn't really useful without dates since I need to understand trends over time to improve my forecasts.

Can you set the date series to be the index?

Then, take the mean of the first 10 and last 10 prices. After that, can you grab all oil prices from January 1st, 2017 to January 7th, 2017 and revert the index of this slice back to integers?

Thanks!

 section02\_Series.ipynb

Reply Forward

## Results Preview

oil\_series

date	oil_prices
2016-12-20	52.22
2016-12-21	51.44
2016-12-22	51.98
2016-12-23	52.01
2016-12-27	52.82
...	
2017-05-09	45.84
2017-05-10	47.28
2017-05-11	47.81
2017-05-12	47.83
2017-05-15	48.86

Name: oil\_prices, Length: 100, dtype: float64

52.765

47.129999999999995

0	52.36
1	53.26
2	53.77
3	53.98

Name: oil\_prices, dtype: float64

# SOLUTION: ACCESSING SERIES DATA



**NEW MESSAGE**

June 21, 2022

**From:** Rachel Revenue (Financial Analyst)

**Subject:** Oil Price Series w/Dates

Thanks for picking up this work, but this data isn't really useful without dates since I need to understand trends over time to improve my forecasts.

Can you set the date series to be the index?

Then, take the mean of the first 10 and last 10 prices. After that, can you grab all oil prices from January 1st, 2017 to January 7th, 2017 and revert the index of this slice back to integers?

Thanks!

[section02\\_Series.ipynb](#)

[Reply](#) [Forward](#)

## Solution Code

```
oil_series.index = dates  
oil_series
```

```
date  
2016-12-20    52.22  
2016-12-21    51.44  
2016-12-22    51.98  
2016-12-23    52.01  
2016-12-27    52.82  
...  
2017-05-09    45.84  
2017-05-10    47.28  
2017-05-11    47.81  
2017-05-12    47.83  
2017-05-15    48.86  
Name: oil_prices, Length: 100, dtype: float64
```

```
oil_series.iloc[:10].mean()
```

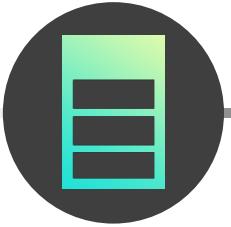
```
52.765
```

```
oil_series.iloc[-10:].mean()
```

```
47.129999999999995
```

```
oil_series.loc["2017-01-01":"2017-01-07"].reset_index(drop=True)
```

```
0      52.36  
1      53.26  
2      53.77  
3      53.98  
Name: oil_prices, dtype: float64
```



# FILTERING SERIES

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

You can **filter a Series** by passing a logical test into the `.loc[]` accessor (like arrays!)

```
sales_series
```

```
coffee      0
coffee      5
tea       155
coconut     0
sugar      518
Name: Sales, dtype: int64
```

```
sales_series.loc[sales_series > 0]
```

```
coffee      5
tea       155
sugar      518
Name: Sales, dtype: int64
```

```
mask = (sales_series > 0) & (sales_series.index == "coffee")
```

```
sales_series.loc[mask]
```

```
coffee      5
Name: Sales, dtype: int64
```

This returns all rows from `sales_series` with a value greater than 0

This uses a **mask** to store complex logic and returns all rows from `sales_series` with a greater than 0 and an index equal to "coffee"



# LOGICAL OPERATORS & METHODS

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

You can use these **operators** & **methods** to create Boolean filters for logical tests

Description	Python Operator	Pandas Method
Equal	==	.eq()
Not Equal	!=	.ne()
Less Than or Equal	<=	.le()
Less Than	<	.lt()
Greater Than or Equal	>=	.ge()
Greater Than	>	.gt()
Membership Test	in	.isin()
Inverse Membership Test	not in	~.isin()

`sales_series`

```
coffee      0
bananas    5
tea       155
coconut     0
sugar      518
Name: Sales, dtype: int64
```

**Python Operator:**

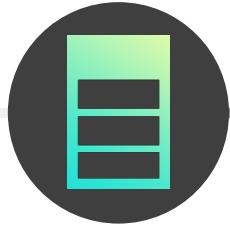
```
sales_series == 5
```

```
coffee  False
coffee  True
tea   False
coconut False
sugar  False
Name: Sales, dtype: bool
```

**Pandas Method:**

```
sales_series.eq(5)
```

```
coffee  False
coffee  True
tea   False
coconut False
sugar  False
Name: Sales, dtype: bool
```



# LOGICAL OPERATORS & METHODS

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

You can use these **operators** & **methods** to create Boolean filters for logical tests

Description	Python Operator	Pandas Method
Equal	==	.eq()
Not Equal	!=	.ne()
Less Than or Equal	<=	.le()
Less Than	<	.lt()
Greater Than or Equal	>=	.ge()
Greater Than	>	.gt()
<b>Membership Test</b>	<b>in</b>	<b>.isin()</b>
<b>Inverse Membership Test</b>	<b>not in</b>	<b>~.isin()</b>

The Python operators 'in' and 'not in' won't work for many operations, so the Pandas method must be used

`sales_series`

```
coffee      0
bananas    5
tea       155
coconut     0
sugar      518
Name: Sales, dtype: int64
```

`sales_series.index.isin(["coffee", "tea"])`

```
array([ True,  True,  True, False, False])
```

`~sales_series.index.isin(["coffee", "tea"])`

```
array([False, False, False,  True,  True])
```

The tilde '`~`' inverts Boolean values!

# SORTING SERIES



Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

You can **sort Series** by their values or their index

1. The **.sort\_values()** method sorts a Series by its values in ascending order

`sales_series.sort_values()`

```
coffee      0
coconut     0
coffee      5
tea        155
sugar      518
Name: Sales, dtype: int64
```

`sales_series.sort_values(ascending=False)`

```
sugar      518
tea        155
coffee      5
coffee      0
coconut     0
Name: Sales, dtype: int64
```

Specify `ascending=False`  
to sort in descending order

2. The **.sort\_index()** method sorts a Series by its index in ascending order

`sales_series.sort_index()`

```
coconut     0
coffee      0
coffee      5
sugar      518
tea        155
Name: Sales, dtype: int64
```

`sales_series.sort_index(ascending=False)`

```
tea        155
sugar      518
coffee      0
coffee      5
coconut     0
Name: Sales, dtype: int64
```

# ASSIGNMENT: SORTING & FILTERING SERIES



**NEW MESSAGE**  
June 22, 2022

**From:** **Rachel Revenue** (Financial Analyst)  
**Subject:** Oil Price Anomalies

Hi again, your work has been super helpful already!  
I need to look at this data from a few more angles.  
First, can you get me the 10 lowest prices from the data, sorted by date, starting with the most recent and ending with the oldest?  
After that, return to the original data. I've provided a list of dates I want to narrow down to, and I also want to look only at prices less than or equal to 50 dollars per barrel.  
Thanks!

 section02\_Series.ipynb

## Results Preview

```
date
2017-05-10    47.28
2017-05-09    45.84
2017-05-08    46.46
2017-05-05    46.23
2017-05-04    45.55
2017-03-27    47.02
2017-03-23    47.00
2017-03-22    47.29
2017-03-21    47.02
2017-03-14    47.24
Name: oil_prices, dtype: float64
```

```
date
2017-03-21    47.02
2017-05-03    47.79
Name: oil_prices, dtype: float64
```

# SOLUTION: SORTING & FILTERING SERIES



**NEW MESSAGE**

June 22, 2022

**From:** Rachel Revenue (Financial Analyst)

**Subject:** Oil Price Anomalies

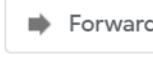
Hi again, your work has been super helpful already!

I need to look at this data from a few more angles.

First, can you get me the 10 lowest prices from the data, sorted by date, starting with the most recent and ending with the oldest?

After that, return to the original data. I've provided a list of dates I want to narrow down to, and I also want to look only at prices less than or equal to 50 dollars per barrel.

Thanks!

## Solution Code

```
oil_series.sort_values().iloc[:10].sort_index(ascending=False)
```

```
date
2017-05-10    47.28
2017-05-09    45.84
2017-05-08    46.46
2017-05-05    46.23
2017-05-04    45.55
2017-03-27    47.02
2017-03-23    47.00
2017-03-22    47.29
2017-03-21    47.02
2017-03-14    47.24
```

```
Name: oil_prices, dtype: float64
```

```
mask = oil_series.index.isin(dates) & (oil_series <= 50)
```

```
oil_series.loc[mask]
```

```
date
2017-03-21    47.02
2017-05-03    47.79
```

```
Name: oil_prices, dtype: float64
```



# ARITHMETIC OPERATORS & METHODS

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

You can use these **operators** & **methods** to perform numeric operations on Series

Operation	Python Operator	Pandas Method
Addition	+	.add()
Subtraction	-	.sub(), .subtract()
Multiplication	*	.mul(), .multiply()
Division	/	.div(), .truediv(), .divide()
Floor Division	//	.floordiv()
Modulo	%	.mod()
Exponentiation	**	.pow()

monday\_sales

0	0
1	5
2	155
3	0
4	518
dtype: int64	

monday\_sales + 2

0	2
1	7
2	157
3	2
4	520
dtype: int64	

monday\_sales.add(2)

0	2
1	7
2	157
3	2
4	520
dtype: int64	

These both add two to every row

"\$" + monday\_sales.astype("float").astype("string")

0	\$0.0
1	\$5.0
2	\$155.0
3	\$0.0
4	\$518.0
dtype: string	

This uses string arithmetic to add a dollar sign,  
converts to float to add decimals (cents), then  
converts back to a string



# STRING METHODS

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

The Pandas str accessor lets you access many **string methods**

- These methods all return a Series (*split returns multiple series*)

String Method	Description
.strip(), .lstrip(), .rstrip()	Removes all leading and/or trailing characters (spaces by default)
.upper(), .lower()	Converts all characters to upper or lower case
.slice(start:stop:step)	Applies a slice to the strings in a Series
.count("string")	Counts all instances of a given string
.contains("string")	Returns True if a given string is found; False if not
.replace("a", "b")	Replaces instances of string "a" with string "b"
.split("delimiter", expand=True)	Splits strings based on a given delimiter string, and returns a DataFrame with a Series for each split
.len()	Returns the length of each string in a Series
.startswith("string"), .endswith("string")	Returns True if a string starts or ends with given string; False if not

```
prices
```

```
0    $3.99
1    $5.99
2   $22.99
3    $7.99
4   $33.99
dtype: object
```

```
prices.str.contains("3")
```

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

```
clean = prices.str.strip("$").astype("float")
```

```
clean
```

```
0    3.99
1    5.99
2   22.99
3    7.99
4   33.99
dtype: float64
```

The **str** accessor lets you access the string methods

This is removing the dollar sign, then converting to float

# ASSIGNMENT: SERIES OPERATIONS



**NEW MESSAGE**

June 22, 2022

**From:** Rachel Revenue (Financial Analyst)

**Subject:** Sensitivity Analysis

Hey there,

I'm doing some 'stress testing' on my models. I want to look at the financial impact if oil prices were 10% higher and add an additional two dollars per barrel on top of that.

Once you've done that, create a series that represents the percent difference between each price and the max price.

Finally, extract the month from the string dates in the index, and store them as an integer.

Thanks!

## Results Preview

<pre>date 2016-12-20    59.442 2016-12-21    58.584 2016-12-22    59.178 2016-12-23    59.211 2016-12-27    60.102 ... 2017-05-09    52.424 2017-05-10    54.008 2017-05-11    54.591 2017-05-12    54.613 2017-05-15    55.746 Name: oil_prices, Length: 100, dtype: float64</pre>	<pre>max_price 54.48</pre>
<pre>max_price_differential date 2016-12-20   -0.041483 2016-12-21   -0.055800 2016-12-22   -0.045888 2016-12-23   -0.045338 2016-12-27   -0.030470 ... 2017-05-09   -0.158590 2017-05-10   -0.132159 2017-05-11   -0.122430 2017-05-12   -0.122063 2017-05-15   -0.103157 Name: oil_prices, Length: 100, dtype: float64</pre>	<pre>0      12 1      12 2      12 3      12 4      12 ... 95     5 96     5 97     5 98     5 99     5 Name: date, Length: 100, dtype: int64</pre>

# SOLUTION: SERIES OPERATIONS



**NEW MESSAGE**

June 22, 2022

**From:** Rachel Revenue (Financial Analyst)

**Subject:** Sensitivity Analysis

Hey there,

I'm doing some 'stress testing' on my models. I want to look at the financial impact if oil prices were 10% higher and add an additional two dollars per barrel on top of that.

Once you've done that, create a series that represents the percent difference between each price and the max price.

Finally, extract the month from the string dates in the index, and store them as an integer.

Thanks!

[Reply](#) [Forward](#)

## Solution Code

```
oil_series * 1.1 + 2
```

date	oil_prices
2016-12-20	59.442
2016-12-21	58.584
2016-12-22	59.178
2016-12-23	59.211
2016-12-27	60.102
...	
2017-05-09	52.424
2017-05-10	54.008
2017-05-11	54.591
2017-05-12	54.613
2017-05-15	55.746

Name: oil\_prices, Length: 100, dtype: float64

```
max_price = oil_series.max()  
max_price
```

54.48

```
max_price_differential = (oil_series - max_price) / max_price
```

date	oil_prices
2016-12-20	-0.041483
2016-12-21	-0.055800
2016-12-22	-0.045888
2016-12-23	-0.045338
2016-12-27	-0.030470
...	
2017-05-09	-0.158590
2017-05-10	-0.132159
2017-05-11	-0.122430
2017-05-12	-0.122063
2017-05-15	-0.103157

Name: oil\_prices, Length: 100, dtype: float64

```
string_dates.str[5:7].astype('int')
```

date	month
2016-12-20	12
2016-12-21	12
2016-12-22	12
2016-12-23	12
2016-12-27	12
...	
2017-05-09	5
2017-05-10	5
2017-05-11	5
2017-05-12	5
2017-05-15	5

Name: date, Length: 100, dtype: int64



# NUMERIC SERIES AGGREGATION

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

You can use these methods to **aggregate numerical Series**

Method	Description
.count()	Returns the number of items
.first(), .last()	Returns the first or last item
.mean(), .median()	Calculates the mean or median
.min(), .max()	Returns the smallest or largest value
.argmax(), .argmin()	Returns the index for the smallest or largest values
.std(), .var()	Calculates the standard deviation or variance
.mad()	Calculates the mean absolute deviation
.prod()	Calculates the product of all the items
.sum()	Calculates the sum of all the items
.quantile()	Returns a specified percentile, or list of percentiles

`sales_series`

```
coffee      0.0
coffee      5.0
tea       155.0
coconut     NaN
sugar      518.0
Name: Sales, dtype: float64
```

`sales_series.sum()`

678.0

`sales_series.loc["coffee"].sum()`

5.0

`sales_series.quantile([0.25, 0.50, 0.75])`

```
0.25      3.75
0.50     80.00
0.75    245.75
Name: Sales, dtype: float64
```



# CATEGORICAL SERIES AGGREGATION

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

You can use these methods to **aggregate categorical Series**

Method	Description
.unique()	Returns an array of unique items in a Series
.nunique()	Returns the number of unique items
.value_counts()	Returns a Series of unique items and their frequency

items

```
0    coffee
1    coffee
2      tea
3   coconut
4     sugar
dtype: object
```



items.value\_counts()

```
coffee    2
tea      1
coconut   1
sugar     1
dtype: int64
```



items.value\_counts(normalize=True)

```
coffee    0.4
tea      0.2
coconut  0.2
sugar    0.2
dtype: float64
```

Specify **normalize=True** to  
return the percentage of total  
for each category

items.unique()



items.nunique()

array(['coffee', 'tea', 'coconut', 'sugar'], dtype=object)

# ASSIGNMENT: SERIES AGGREGATIONS



**NEW MESSAGE**

June 23, 2022

**From:** Rachel Revenue (Financial Analyst)

**Subject:** Additional Metrics

Hi again!

I need a few more metrics. Can you calculate the sum and mean of prices in the month of march? Next, how many prices did we have in Jan and Feb?

Then, calculate the 10th and 90th percentiles across all data.

Finally, how often did integer dollar values (e.g. 51, 52) occur in the dataNormalize the results to a percentage.

Thanks!

 section02\_Series.ipynb

 Reply     Forward

## Results Preview

```
1134.54
```

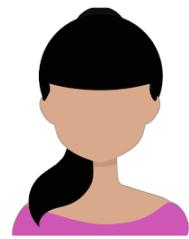
```
49.32782608695651
```

```
39
```

```
0.1    47.299
0.9    53.811
Name: oil_prices, dtype: float64
```

```
53    0.26
52    0.22
47    0.13
48    0.10
51    0.07
50    0.07
49    0.06
54    0.05
45    0.02
46    0.02
Name: oil_prices, dtype: float64
```

# SOLUTION: SERIES AGGREGATIONS



## NEW MESSAGE

June 23, 2022

From: **Rachel Revenue** (Financial Analyst)

Subject: Additional Metrics

Hi again!

I need a few more metrics. Can you calculate the sum and mean of prices in the month of march? Next, how many prices did we have in Jan and Feb?

Then, calculate the 10th and 90th percentiles across all data.

Finally, how often did integer dollar values (e.g. 51, 52) occur in the data. Normalize the results to a percentage.

Thanks!

Reply

Forward

section02\_Series.ipynb

## Solution Code

```
oil_series[oil_series.index.str[6:7] == "3"].sum().round(2)
```

```
1134.54
```

```
oil_series[oil_series.index.str[6:7] == "3"].mean()
```

```
49.32782608695651
```

```
oil_series[oil_series.index.str[5:7].isin(["01", "02"])].count()
```

```
39
```

```
oil_series.quantile([0.1, 0.9])
```

```
0.1    47.299  
0.9    53.811  
Name: oil_prices, dtype: float64
```

```
oil_series.astype("int").value_counts(normalize=True)
```

```
53    0.26  
52    0.22  
47    0.13  
48    0.10  
51    0.07  
50    0.07  
49    0.06  
54    0.05  
45    0.02  
46    0.02  
Name: oil_prices, dtype: float64
```

# MISSING DATA



Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

**Missing data** in Pandas is often represented by NumPy “NaN” values

- This is more efficient than Python’s “None” data type
- Pandas treats NaN values as a float, which allows them to be used in vectorized operations

```
sales = [0, 5, 155, np.nan, 518]
sales_series = pd.Series(sales, name="Sales")
sales_series
```

```
0      0.0
1      5.0
2    155.0
3      NaN
4    518.0
Name: Sales, dtype: float64
```

**np.nan** creates a NaN value

These are rarely created by hand,  
and typically appear when reading  
in data from external sources

If NaN was not present here,  
the data type would be int64

```
sales_series + 2
```

```
0      2.0
1      7.0
2    157.0
3      NaN
4    520.0
Name: Sales, dtype: float64
```

Arithmetic operations performed  
on NaN values will return NaN

```
sales_series.add(2, fill_value=0)
```

```
0      2.0
1      7.0
2    157.0
3      2.0
4    520.0
Name: Sales, dtype: float64
```

Most operation methods include a  
'fill\_value' argument that lets you  
pass a value instead of NaN



# MISSING DATA

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

Pandas released its own **missing data type**, NA, in December 2020

- This allows missing values to be stored as integers, instead of needing to convert to float
- This is still a new feature, but most bugs end up converting the data to NumPy's NaN

```
sales = [0, 5, 155, pd.NA, 518]  
  
sales_series = pd.Series(sales, name="Sales", dtype="Int16")  
  
sales_series
```

```
0      0  
1      5  
2    155  
3    <NA>  
4    518  
Name: Sales, dtype: Int16
```

} **pd.NA** creates an NA value

Note that if **dtype="Int16"**  
wasn't specified, the values  
would be stored as objects



At this time, **neither np.NaN nor pd.NA are perfect**,  
but pd.NA functionality should continue to improve,  
and having a nullable integer is usually worth it  
(more on that in the next section!)



# IDENTIFYING MISSING DATA

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

The `.isna()` and `.value_counts()` methods let you **identify missing data** in a Series

- The `.isna()` method returns True if a value is missing, and False otherwise

checklist		checklist.isna()		checklist.isna().sum()	
0	COMPLETE	0	False		
1	NaN	1	True		
2	NaN	2	True		
3	NaN	3	True	3	
4	COMPLETE	4	False		
	<code>dtype: object</code>		<code>dtype: bool</code>		

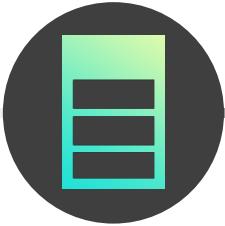
*You can use this as a Boolean mask!*

`.isna().sum()` returns the count of NaN values

- The `.value_counts()` method returns unique values and their frequency

checklist.value_counts()		checklist.value_counts(dropna=False)	
COMPLETE	2	NaN	3
		COMPLETE	2
		<code>dtype: int64</code>	<code>dtype: int64</code>

Most methods ignore NaN values, so you need to specify `dropna=False` to return the count of NaN values



# HANDLING MISSING DATA

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

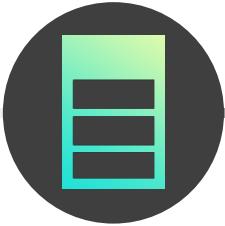
The `.dropna()` and `.fillna()` methods let you **handle missing data** in a Series

- The `.dropna()` method removes NaN values from your Series or DataFrame

checklist		checklist.dropna()	Note that the index has gaps, so you can use <code>.reset_index()</code> to restore the range of integers
0	COMPLETE	0	COMPLETE
1	NaN	4	COMPLETE
2	NaN		
3	NaN		
4	COMPLETE		
		dtype: object	

- The `.fillna(value)` method replaces NaN values with a specified value

checklist		checklist.fillna("INCOMPLETE")	
0	COMPLETE	0	COMPLETE
1	NaN	1	INCOMPLETE
2	NaN	2	INCOMPLETE
3	NaN	3	INCOMPLETE
4	COMPLETE	4	COMPLETE
		dtype: object	



# HANDLING MISSING DATA

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

It's important to **be thoughtful and deliberate** in how you handle missing data

## EXAMPLE

*Handling missing values from product sales*

Do you **keep** them?

`sales_series`

```
coffee      0.0
coffee      5.0
tea       155.0
coconut     NaN
sugar      518.0
Name: Sales, dtype: float64
```

Do you **remove** them?

`sales_series.dropna()`

```
coffee      0.0
coffee      5.0
tea       155.0
sugar      518.0
Name: Sales, dtype: float64
```

Do you **replace** them with zeros?

`sales_series.fillna(0)`

```
coffee      0.0
coffee      5.0
tea       155.0
coconut     0.0
sugar      518.0
Name: Sales, dtype: float64
```

Do you **impute** them with the mean?

`sales_series.fillna(sales_series.mean())`

```
coffee      0.0
coffee      5.0
tea       155.0
coconut    169.5
sugar      518.0
Name: Sales, dtype: float64
```



**PRO TIP:** These operations can dramatically impact the results of an analysis, so make sure you understand these impacts and talk to a data SME to understand *why* data is missing

# ASSIGNMENT: MISSING DATA

 **NEW MESSAGE**  
June 27, 2022

**From:** **Rachel Revenue** (Financial Analyst)  
**Subject:** Erroneous Data

Hey,

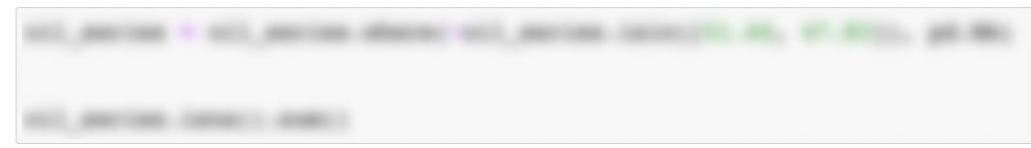
I just got a promotion thanks to the analysis you helped me with. I owe you lunch!

I noticed that two prices (51.44, 47.83), were incorrect, so I had them filled in with missing values. I'm not sure if I did this correctly. Can you confirm the number of missing values in the price column? Once you've done that, fill the prices in with the median of the oil price series.

Thanks!

 section02\_Series.ipynb       Reply       Forward

## Results Preview



2



date

```
2016-12-20    52.220
2016-12-21    52.205
2016-12-22    51.980
2016-12-23    52.010
2016-12-27    52.820
...
2017-05-09    45.840
2017-05-10    47.280
2017-05-11    47.810
2017-05-12    52.205
2017-05-15    48.860
```

Name: oil\_prices, Length: 100, dtype: float64

# SOLUTION: MISSING DATA

 NEW MESSAGE  
June 27, 2022

**From:** Rachel Revenue (Financial Analyst)  
**Subject:** Erroneous Data

Hey,

I just got a promotion thanks to the analysis you helped me with. I owe you lunch!

I noticed that two prices (51.44, 47.83), were incorrect, so I had them filled in with missing values. I'm not sure if I did this correctly. Can you confirm the number of missing values in the price column? Once you've done that, fill the prices in with the median of the oil price series.

Thanks!

 section02\_Series.ipynb     Reply     Forward

## Solution Code

```
oil_series = oil_series.where(~oil_series.isin([51.44, 47.83]), pd.NA)

oil_series.isna().sum()

2

oil_series.fillna(oil_series.median())

date
2016-12-20      52.220
2016-12-21      52.205
2016-12-22      51.980
2016-12-23      52.010
2016-12-27      52.820
...
2017-05-09      45.840
2017-05-10      47.280
2017-05-11      47.810
2017-05-12      52.205
2017-05-15      48.860
Name: oil_prices, Length: 100, dtype: float64
```



# THE APPLY METHOD

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

The `.apply()` method lets you apply custom functions to Pandas Series

- This function will not be vectorized, so it's not as efficient as native functions

```
def discount(price):
    if price > 20:
        return round(price * 0.9, 2)
    return price
```

This function applies a 10% discount to prices over 20

```
clean_wholesale
```

```
0    3.99
1    5.99
2   22.99
3    7.99
4   33.99
dtype: float64
```

```
clean_wholesale.apply(discount)
```

```
0    3.99
1    5.99
2  20.69
3    7.99
4  30.59
dtype: float64
```

Discount applied!

```
clean_wholesale.apply(lambda x: round(x * 0.9, 2) if x > 20 else x)
```

```
0    3.99
1    5.99
2  20.69
3    7.99
4  30.59
dtype: float64
```

You can also use **Lambda**  
functions for one-off tasks!



# THE WHERE METHOD

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

Pandas' **.where()** method lets you manipulate data based on a logical condition

```
df.where(logical test,  
         value if False,  
         inplace=False)
```

*Series or DataFrame  
to evaluate data from*

*Whether to perform  
the operation in place  
(default is False)*

*A logical expression that  
evaluates to True or False*

*Value to return when  
the expression is False*



Heads up! This is different  
from NumPy's where function



# THE WHERE METHOD

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

Pandas' **.where()** method lets you manipulate data based on a logical condition

```
clean_wholesale
```

```
0    3.99
1    5.99
2   22.99
3    7.99
4   33.99
dtype: float64
```

```
clean_wholesale.where(clean_wholesale <= 20, round(clean_wholesale * 0.9, 2))
```

```
0    3.99
1    5.99
2  20.69
3    7.99
4  30.59
dtype: float64
```



This expression returns `False` if the price is greater than 20,  
and the value if `True` statement for the discount is applied



# THE WHERE METHOD

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

Pandas' **.where()** method lets you manipulate data based on a logical condition

```
clean_wholesale
```

```
0    3.99
1    5.99
2   22.99
3    7.99
4   33.99
dtype: float64
```

```
clean_wholesale.where(~(clean_wholesale > 20), round(clean_wholesale * 0.9, 2))
```

```
0    3.99
1    5.99
2   20.69
3    7.99
4   30.59
dtype: float64
```

You can use a tilde '`~`' to invert the Boolean values and turn this into a "value if True"





# CHAINING WHERE

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

You can **chain .where() methods** to combine logical expressions

```
clean_wholesale
```

```
0      3.99
1      5.99
2     22.99
3      7.99
4     33.99
dtype: float64
```

```
(clean_wholesale
    .where(~(clean_wholesale > 20), round(clean_wholesale * 0.9, 2))
    .where(clean_wholesale > 10, 0)
)
```

```
0      0.00
1      0.00
2     20.69
3      0.00
4     30.59
dtype: float64
```

The first where method applies a 90% discount if a price is greater than 20

The second applies a value of 0 when a price is NOT greater than 10



# NUMPY VS. PANDAS WHERE

Pandas Series  
Basics

Series Indexing

Sorting &  
Filtering

Operations &  
Aggregations

Handling  
Missing Data

Applying Custom  
Functions

NumPy's where function is often more convenient & useful than Pandas' method

```
clean_wholesale
```

```
0      3.99
1      5.99
2     22.99
3      7.99
4     33.99
dtype: float64
```

```
np.where(clean_wholesale > 20, "Discounted", "Normal Price")
```

```
array(['Normal Price', 'Normal Price', 'Discounted', 'Normal Price',
       'Discounted'], dtype='<U12')
```



Note that this returns a NumPy array that  
you'd need to convert into a Pandas Series

# ASSIGNMENT: APPLY & WHERE



**NEW MESSAGE**

June 29, 2022

**From:** **Rachel Revenue** (Financial Analyst)

**Subject:** Additional Metrics

Hey, our 'well' of oil analysis is almost dried up!

Write a function that outputs 'buy' if price is less than the 90th percentile and 'wait' if it's not. Apply it to the oil series.

Then, I need to fix two final prices. Create a series that multiplies price by .9 if the date is '2016-12-23' or '2017-05-10', and 1.1 for all other dates.

Thanks!

 section02\_Series.ipynb

 Reply     Forward

## Results Preview

```
date
2016-12-20    Buy
2016-12-21    Wait
2016-12-22    Buy
2016-12-23    Buy
2016-12-27    Buy
...
2017-05-09    Buy
2017-05-10    Buy
2017-05-11    Buy
2017-05-12    Wait
2017-05-15    Buy
Name: dcoilwtico, Length: 100, dtype: object
```

```
date
2016-12-20    57.442
2016-12-21    NaN
2016-12-22    57.178
2016-12-23    46.809
2016-12-27    58.102
...
2017-05-09    50.424
2017-05-10    42.552
2017-05-11    52.591
2017-05-12    NaN
2017-05-15    53.746
Length: 100, dtype: float64
```

# SOLUTION: APPLY & WHERE



**NEW MESSAGE**

June 29, 2022

**From:** Rachel Revenue (Financial Analyst)

**Subject:** Additional Metrics

Hey, our 'well' of oil analysis is almost dried up!

Write a function that outputs 'buy' if price is less than the 90th percentile and 'wait' if it's not. Apply it to the oil series.

Then, I need to fix two final prices. Create a series that multiplies price by .9 if the date is '2016-12-23' or '2017-05-10', and 1.1 for all other dates.

Thanks!

 section02\_Series.ipynb

## Solution Code

```
oil_series.apply(lambda x: 'Buy' if x < oil_series.quantile(.9) else 'Wait')
```

```
date
2016-12-20    Buy
2016-12-21    Wait
2016-12-22    Buy
2016-12-23    Buy
2016-12-27    Buy
...
2017-05-09    Buy
2017-05-10    Buy
2017-05-11    Buy
2017-05-12    Wait
2017-05-15    Buy
Name: dcoilwtico, Length: 100, dtype: object
```

```
pd.Series(
    np.where(
        oil_series.index.isin(["2016-12-23", "2017-05-10"]),
        oil_series * 0.9,
        oil_series * 1.1,
    ),
    index=dates,
)
```

```
date
2016-12-20    57.442
2016-12-21    NaN
2016-12-22    57.178
2016-12-23    46.809
2016-12-27    58.102
...
2017-05-09    50.424
2017-05-10    42.552
2017-05-11    52.591
2017-05-12    NaN
2017-05-15    53.746
Length: 100, dtype: float64
```

# KEY TAKEAWAYS

---



Pandas Series add an **index** & **title** to NumPy arrays

- *Pandas Series form the columns for DataFrames, which we will cover in the next section*



The **.loc()** & **.iloc()** methods are key in working with Pandas data structures

- *These methods allow you to access rows in Series (and later columns in DataFrames), either by their positional index or by their labels*



Pandas & NumPy have **similar operations** for filtering, sorting & aggregating

- *Use built-in Pandas and NumPy functions and methods to take advantage of vectorization, which is much more efficient than writing for loops in base Python*

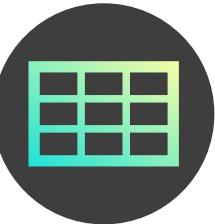


Pandas lets you easily **handle missing data**

- *It's important to understand the impact dropping or imputing might have on your analysis, so make sure you consult an expert about the root cause of missing data*

# 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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

**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)

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

Each column is a Pandas Series

The column index  
points to each  
individual Series  
(axis = 1)



# DATAFRAME PROPERTIES

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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.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')]`

`df.dtypes`

<code>id</code>	<code>int64</code>
<code>date</code>	<code>object</code>
<code>store_nbr</code>	<code>int64</code>
<code>family</code>	<code>object</code>
<code>sales</code>	<code>float64</code>
<code>onpromotion</code>	<code>int64</code>
<code>dtype: object</code>	

Pandas will try to guess the data types when creating a DataFrame (we'll modify them later!)



# CREATING A DATAFRAME

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

	<b>id</b>	<b>store_nbr</b>	<b>family</b>
<b>0</b>	1	1	POULTRY
<b>1</b>	2	2	PRODUCE



# CREATING A DATAFRAME

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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
```

	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>
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 x 6 columns

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!



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



## Alert

This data set is too large for the Excel grid. If you save this workbook, you'll lose data that wasn't loaded.

# 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 transactions.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

(83488, 3)

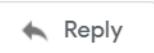
date  
store\_nbr  
transactions  
dtype: object

# 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 transactions.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.shape  
(83488, 3)
```

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



# EXPLORING A DATAFRAME

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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()
```

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

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

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

	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>
<b>3000885</b>	3000885	2017-08-15	9	PRODUCE	2419.729	148
<b>3000886</b>	3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.0	8
<b>3000887</b>	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



DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

```
retail_df.sample()
```

	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>
	<b>1476628</b>	1476628	2015-04-11	40	EGGS	77.0

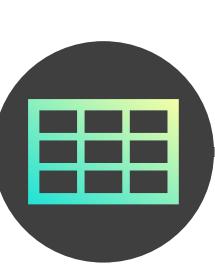
} This returns **1** row by default

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

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

} 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



DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

```
retail_df.info()
```

Rows & columns  
in the DataFrame

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3000888 entries, 0 to 3000887
Data columns (total 6 columns):
```

Position, name,  
and data type for  
each column

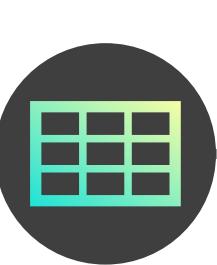
#	Column	Dtype
0	<code>id</code>	<code>int64</code>
1	<code>date</code>	<code>object</code>
2	<code>store_nbr</code>	<code>int64</code>
3	<code>family</code>	<code>object</code>
4	<code>sales</code>	<code>float64</code>
5	<code>onpromotion</code>	<code>int64</code>

```
dtypes: float64(1), int64(3), object(2)
```

Memory usage

```
{ memory usage: 137.4+ MB}
```

# INFO



DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

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

#	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

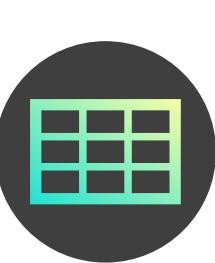
dtypes: float64(1), int64(3), object(2)  
memory usage: 137.4+ MB

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



DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

`retail_df.describe()`

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

Only numeric columns by default

# DESCRIBE



DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

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

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

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

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

# ASSIGNMENT: EXPLORING DATAFRAMES



**1 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



**1 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

## 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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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



# ACCESSING DATAFRAME COLUMNS

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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().iloc[: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



# ACCESSING DATAFRAME COLUMNS

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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"]].iloc[: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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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, :]
```

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



# ACCESING DATA WITH ILOC

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

Wrap single columns  
in brackets to return  
a DataFrame

3000888 rows × 1 columns



# ACCESING DATA WITH LOC

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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,

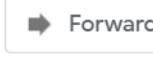
I noticed that the first row is the only one from 2013-01-01.

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!

## 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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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`

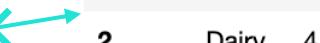
	product	price
0	Dairy	2.56
1	Dairy	2.56
2	Dairy	4.55
3	Vegetables	2.74
4	Fruits	5.44

`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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

## Results Preview



### 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

transactions [REDACTED]

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 [REDACTED].head()

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

transactions [REDACTED].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



**1 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

## 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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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*

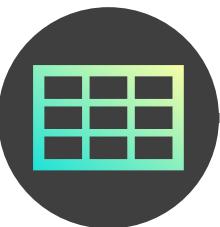
The `.isna()` method returns a DataFrame with Boolean values (True for NAs, False for others)

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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})`

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

Use a dictionary  
to specify a value  
for each column

`product_df.dropna()`

	product	price
2	Dairy	4.55

This drops any row  
with missing values

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

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

Use subset to drop rows  
with missing values in  
specified columns

# ASSIGNMENT: MISSING DATA



**1 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

## 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!

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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"]
```

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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()
```

	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

Row filter

Column filter

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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]
```

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

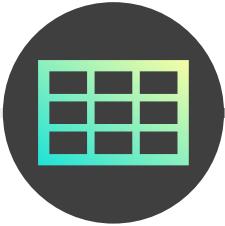
- You can specify any number of filtering conditions by using the “and” & “or” keywords

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

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

## 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
```



section03\_DataFrames.ipynb

Reply

Forward



# SORTING DATAFRAMES BY INDICES

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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
```

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

	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>
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(ascending=False)
```

This sorts the sample DataFrame in descending order by its row index  
(it sorts in ascending order by default)

+	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>	
-	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



# SORTING DATAFRAMES BY INDICES

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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
```

	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>
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
```

	<b>date</b>	<b>family</b>	<b>id</b>	<b>onpromotion</b>	<b>sales</b>	<b>store_nbr</b>
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



Remember that DataFrame methods **don't sort in place by default**, allowing you to chain multiple methods together

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



# SORTING DATAFRAMES BY VALUES

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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



**1 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

## 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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

**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
```

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

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

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

product\_df

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

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



# RENAMING COLUMNS

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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'})

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

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



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

product\_df.rename(columns=lambda x: x.upper())

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

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



# REORDERING COLUMNS

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

**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



**1 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

## Results Preview

transactions.head()

1	2013-01-02	1
2	2013-01-02	2
3	2013-01-02	3
4	2013-01-02	4
5	2013-01-02	5

	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.tail()

	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



**1 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

## 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  
)
```

	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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>
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

	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>	<b>tax_amount</b>	<b>total</b>
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

This creates a new **tax\_amount** column equal to **sales \* 0.05**

This creates a new **total** column equal to **sales + tax\_amount**

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



# BOOLEAN COLUMN CREATION

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

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

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

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

```
baby_books
```

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

This creates a new **taxable\_category** column with Boolean values – True if the family is not “BABY CARE”, and False if it is

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 ( $\text{sales} * 0.05 * 1$ ), otherwise return zero ( $\text{sales} * 0.05 * 0$ )

# 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!

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



section03\_DataFrames.ipynb

# 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!

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>	<b>tax_amount</b>	<b>total</b>	<b>taxable_category</b>	<b>Sale_Name</b>
	2390356	2390356	2016-09-06	29	BABY CARE	3.0	0	0.0	3.0	False
	2691613	2691613	2017-02-23	31	BABY CARE	1.0	0	0.0	1.0	False
	2538925	2538925	2016-11-28	47	BOOKS	6.0	0	0.3	6.3	True
	2022637	2022637	2016-02-13	11	BABY CARE	1.0	0	0.0	1.0	False
	2585356	2585356	2016-12-24	5	BOOKS	2.0	0	0.1	2.1	True

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

product_id	product	price	Vegan?
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

Keys

Values

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



# MAPPING VALUES TO COLUMNS

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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
1	2	Dairy	\$2.56
2	3	Dairy	\$4.55
3	4	Vegetables	\$2.74
4	5	Fruits	\$5.44

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



# PRO TIP: COLUMN CREATION WITH ASSIGN

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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)
```

	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>	<b>tax_amount</b>
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'")
```

	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>	<b>tax_amount</b>	<b>on_promotion_flag</b>	<b>year</b>
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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization



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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization



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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization



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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization



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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization



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!

## 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

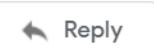
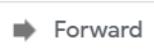
7409700

# 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!

## 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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

**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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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"})
```

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

<b>family</b>
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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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
```

	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>	<b>sales_int</b>	
	<b>2782266</b>	2782266	2017-04-15	25	BEVERAGES	5516.0	24	5516
	<b>2356175</b>	2356175	2016-08-18	2	DAIRY	714.0	7	714
	<b>1691390</b>	1691390	2015-08-10	17	DAIRY	613.0	0	613
	<b>1435443</b>	1435443	2015-03-19	35	DELI	134.0	24	134
	<b>2939747</b>	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
```

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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"))`

	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

But applying cleaning steps  
to the data can make it work



# PRO TIP: MEMORY OPTIMIZATION

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

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
```

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

1467

The total memory usage is 1,472 bytes

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



**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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

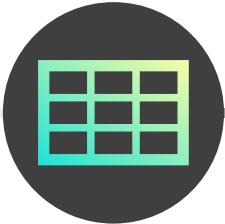
```
class_data.drop("id", axis=1, inplace=True)
```

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

1427

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

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



# STEP 2: CONVERTING OBJECT DATA TYPES

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

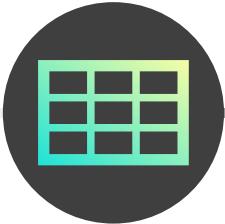
Use **memory\_usage="deep"** with the .info() method to get total memory usage along with the column data types

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

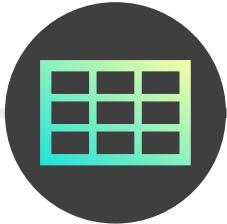
```
class_data = class_data.astype({"start_date": "Datetime64", "title": "string"})
```

```
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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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")
```

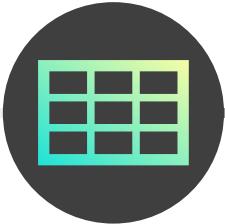
```
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 5 entries, 0 to 4  
Data columns (total 5 columns):  
 #   Column           Non-Null Count  Dtype     
 ---  --     
 0   start_date      5 non-null       datetime64[ns]  
 1   title           5 non-null       string  
 2   class_level     4 non-null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

Dtype
datetime64[ns]
string
float64
float64
float64

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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

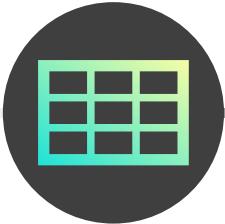
```
class_data = class_data.astype(  
    {  
        "class_level": "Int8",  
        "price": "Int16",  
        "students_enrolled": "Int16"  
    }  
  
    class_data.info(memory_usage="deep")  
  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 5 entries, 0 to 4  
Data columns (total 5 columns):  
 #   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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

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
1	2022-02-22	Algebra 2		9 <NA>	43
2	2022-03-03	Geometry	<NA>	399	16
3	2022-04-04	Trigonometry		11	599
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	102
1	2022-02-22	Algebra 2	9	<NA>	43	43
2	2022-03-03	Geometry	<NA>	399	16	16
3	2022-04-04	Trigonometry	11	599	8	8
4	2022-05-05	Calculus	12	<NA>	<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

DataFrame  
Basics

Exploring  
DataFrames

Accessing &  
Dropping Data

Blank & Duplicate  
Values

Sorting &  
Filtering

Modifying  
Columns

Pandas Data  
Types

Memory  
Optimization

***“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   date             83487 non-null    datetime64[ns]  
 1   time             83487 non-null    datetime64[ns]  
 2   transaction_id  83487 non-null    int64  
 3   product_id      83487 non-null    int64  
 4   quantity        83487 non-null    int64  
 5   unit_price      83487 non-null    float64  
 6   total           83487 non-null    float64  
 7   customer_id     83487 non-null    int64  
 8   store_id        83487 non-null    int64
```

memory usage: 3.3 MB

# SOLUTION: MEMORY OPTIMIZATION

## 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
```



### 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

# 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()*

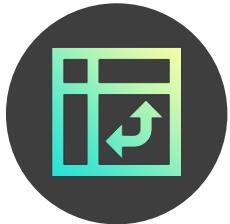


**Memory optimization** is critical in working with large datasets in Pandas

- *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

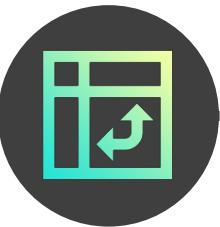
Aggregating Groups

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.sum()`)

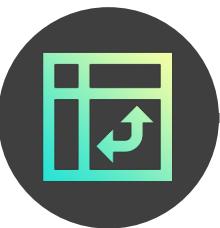
```
retail.loc[:, ['sales', 'onpromotion']].sample(100).sum().round(2)
```

```
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*

	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>
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

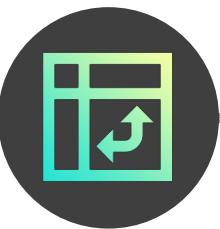
*Raw sales data by transaction*

*Grouped by Family*

	<b>family</b>	<b>sales</b>
0	<b>AUTOMOTIVE</b>	264.0
1	<b>BABY CARE</b>	0.0
2	<b>BEAUTY</b>	121.0
3	<b>BEVERAGES</b>	29817.0
4	<b>BOOKS</b>	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

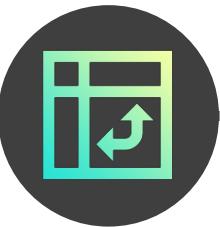
```
small_retail.groupby('family')[ 'sales' ].sum().head()
```

```
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:

- Specify the column for the calculations
- 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

	sales
family	
<b>AUTOMOTIVE</b>	264.0
<b>BABY CARE</b>	0.0
<b>BEAUTY</b>	121.0
<b>BEVERAGES</b>	29817.0
<b>BOOKS</b>	0.0

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

- Specify the column for the calculations
- 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

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

# SOLUTION: GROUPBY

 **NEW MESSAGE**  
August 3, 2022

**From:** Phoebe Product (Merchandising)  
**Subject:** Top Stores by Transactions

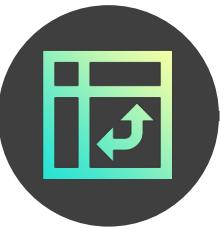
Hi there, it's Phoebe!  
I want to create some custom displays for our busiest stores.  
Can you return a table containing the top 10 stores by total transactions in the data?  
Make sure they're sorted from highest to lowest.  
Thanks!  
P.S. Let me know if you want to hear my music!

 section04\_aggregations.ipynb     Reply     Forward

## 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

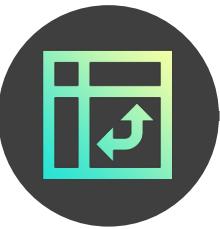
```
small_retail.groupby(['family', 'store_nbr'])[['sales']].sum()
```

		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!

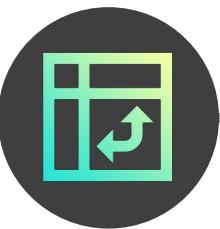
 section04\_aggregations.ipynb     Reply     Forward

## 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

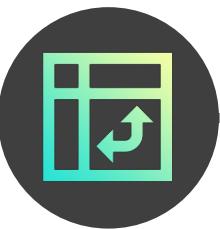
sales_sums		
sales		
family	store_nbr	
AUTOMOTIVE	11	8.000000
	12	9.000000
	13	7.000000
	16	5.000000
	19	5.000000
SEAFOOD	...	...
	47	48.239998
	49	78.718000
	50	18.583000
	53	2.000000
	8	57.757000

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

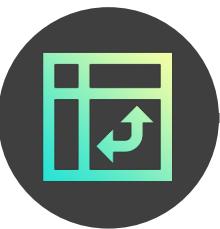
All rows for "AUTOMOTIVE"  
(note that 'family' is dropped)

`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)

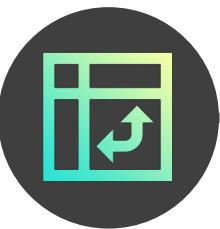
`sales_sums.loc['AUTOMOTIVE', '11':'BEAUTY', '11'], :]`

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()
```

	store_nbr	family	sales
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')
```

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



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



Be careful! You may lose important information

# 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

	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)
```

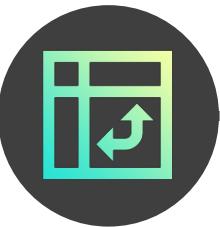
```
store_nbr    month
44           1        4246.202703
Name: (transactions, mean), dtype: float64
```

```
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

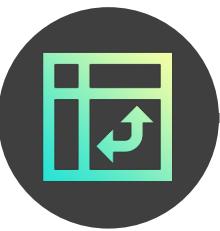


```
small_retail.groupby(['store_nbr', 'family']).agg('sum')
```

store_nbr	family	sales		onpromotion
		1	BEAUTY	7.000
	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

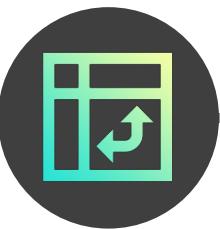
You can perform **multiple aggregations** by passing a list of aggregation functions

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

family	store_nbr	sales		onpromotion	
		sum	mean	sum	mean
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

Grouping  
Columns

Multi-index  
DataFrames

Aggregating  
Groups

Pivot Tables

Melting  
DataFrames

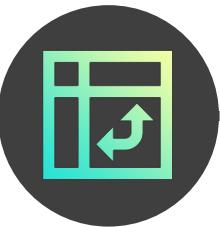
You can perform **specific aggregations by column** by passing a dictionary with column names as keys, and lists of aggregation functions as values

```
(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'

family	store_nbr	sales		onpromotion	
		sum	mean	min	max
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!

		sales_sum	sales_avg	on_promotion_max
family	store_nbr			
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



**1 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

Grouping  
Columns

Multi-index  
DataFrames

Aggregating  
Groups

Pivot Tables

Melting  
DataFrames

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

The value for rows with store 48 is the same!

1000 rows x 6 columns

# 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
1	2013-01-02	1	2111	0.8444	False		0	1	2	1523.844272
2	2013-01-02	2	2358	0.9432	False		0	1	2	1920.036374
3	2013-01-02	3	3487	1.3948	True		100	1	2	3201.879475
4	2013-01-02	4	1922	0.7688	False		0	1	2	1502.987470
										419.012530



# PIVOT TABLES

Grouping  
Columns

Multi-index  
DataFrames

Aggregating  
Groups

Pivot Tables

Melting  
DataFrames

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

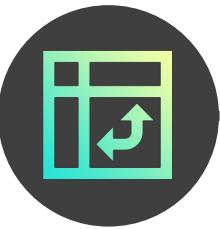
store_nbr	1	2	3	4
family				
AUTOMOTIVE	5475.0	9100.0	15647.0	6767.0
BABY CARE	0.0	84.0	672.0	24.0
BEAUTY	4056.0	7936.0	16189.0	6890.0

The screenshot shows the 'PivotTable Fields' ribbon in Excel. The 'Columns' section has 'store\_nbr' selected. The 'Rows' section has 'family' selected. The 'Values' section has 'Sum of sales' selected.

Sum of sales	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



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

Grouping  
Columns

Multi-index  
DataFrames

Aggregating  
Groups

Pivot Tables

Melting  
DataFrames

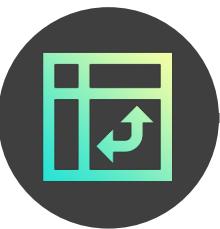
The `.pivot_table()` method has these arguments:

- **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')
```

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

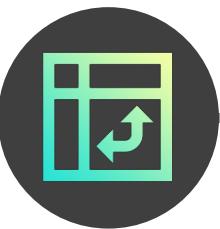
The `.pivot_table()` method has these arguments:

- **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

Grouping  
Columns

Multi-index  
DataFrames

Aggregating  
Groups

Pivot Tables

Melting  
DataFrames

Multiple aggregation functions can be passed to the “aggfunc” argument

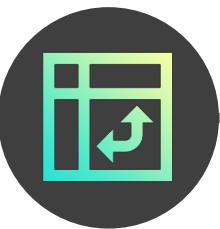
- The new values are added as additional columns

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

The functions are passed as a tuple

store_nbr	max				min			
	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

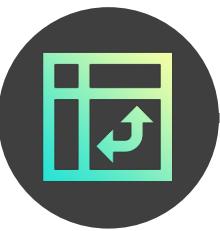
**Multiple aggregation functions** can be passed to the “aggfunc” argument

- 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

store_nbr	onpromotion		sales						
			max	mean					
	1	2	3	4	1	2	3	4	
family									
AUTOMOTIVE	1	1	1	1	3.251188	5.403800	9.291568	4.018409	5475.0
BABY CARE	0	0	1	0	0.000000	0.049881	0.399050	0.014252	9100.0
BEAUTY	2	2	2	2	2.408551	4.712589	9.613420	4.091449	15647.0
									6767.0
									672.0
									24.0
									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'])
```

family	store_nbr	onpromotion		sales	
		mean	sum	mean	sum
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"]})
```

family	store_nbr	onpromotion		sales	
		mean	sum	mean	sum
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')
```

family	store_nbr	onpromotion		sales	
		mean	sum	mean	sum
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

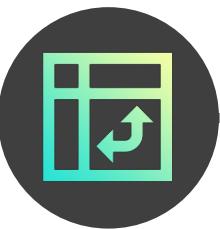


```
smaller_retail.groupby(['family', 'store_nbr']).agg(
    on_promo_avg="onpromotion", "mean",
    on_promo_sum="onpromotion", "sum",
    sales_avg="sales", "mean"),
    sales_sum="sales", "sum",
)
```

		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

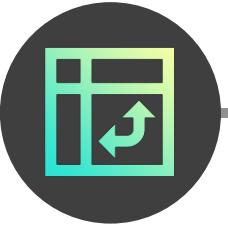
```
(smaller_retail.pivot_table(  
    index="family",  
    columns="store_nbr",  
    values="sales",  
    aggfunc="sum"  
).style.background_gradient(cmap="RdYlGn", axis=None)  
)
```

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

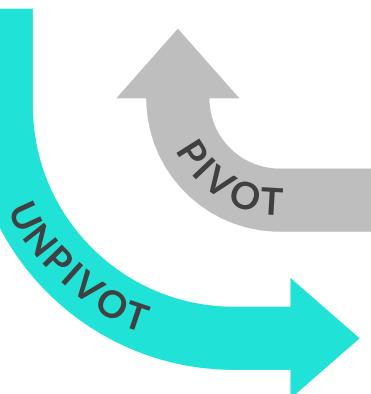
Aggregating  
Groups

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

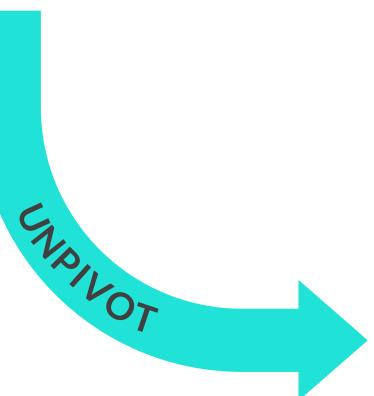
Aggregating  
Groups

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



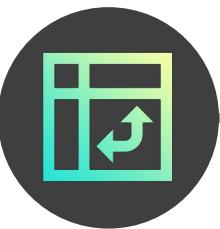
country_revenue.melt(id_vars="country")			
	country	variable	value
0	Algeria	2000	151
1	Egypt	2000	204
2	Nigeria	2000	277
3	South Africa	2000	190
4	Algeria	2001	171
5	Egypt	2001	214
6	Nigeria	2001	301
7	South Africa	2001	211
8	Algeria	2002	176
9	Egypt	2002	227
10	Nigeria	2002	311
11	South Africa	2002	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



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

Grouping  
Columns

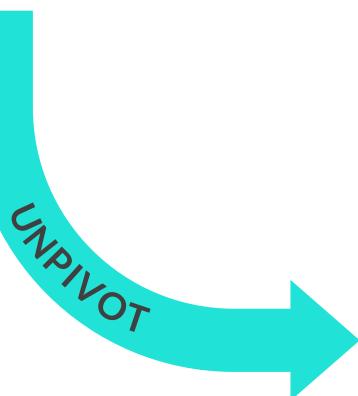
Multi-index  
DataFrames

Aggregating  
Groups

Pivot Tables

Melting  
DataFrames

country_revenue		2000	2001	2002	2003
	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	200.0
1	2	300.0
2	3	24000.0
3	4	200.0
4	5	200.0

# SOLUTION: PIVOT & MELT



**1 NEW MESSAGE**

August 7, 2022

From: Chandler Capital (Accounting)

Subject: Store Transactions Vs. Average

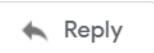
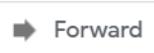
Hey again,

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

## 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	200.0
1	2	300.0
2	3	24000.0
3	4	200.0
4	5	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

# DATA VISUALIZATION

# DATA VISUALIZATION



In this section we'll cover basic **data visualization** in Pandas, and use the plot method to create & customize line charts, bar charts, pie charts, scatterplots, and histograms

## TOPICS WE'LL COVER:

The Plot Method

Chart Formatting

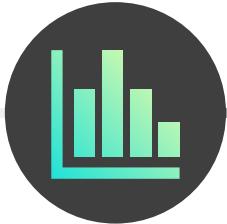
Chart Types

Saving Charts

Additional Libraries

## GOALS FOR THIS SECTION:

- Use the Matplotlib API and the .plot() method to create visualizations from Pandas DataFrames
- Create different chart types for individual and multiple series of data
- Modify the chart elements and formatting to tell a clear story with data



# THE MATPLOTLIB API

The Plot Method

Chart Formatting

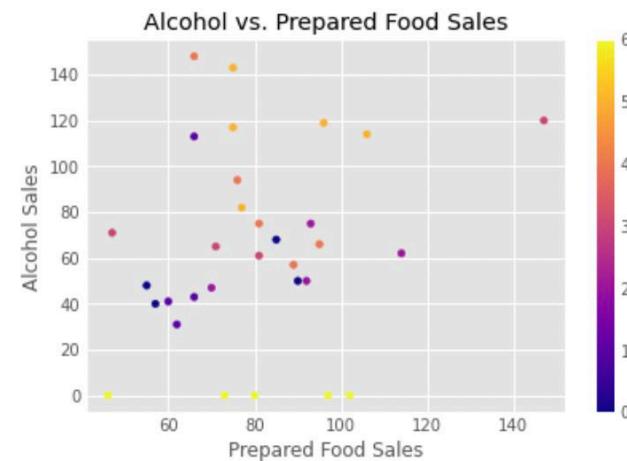
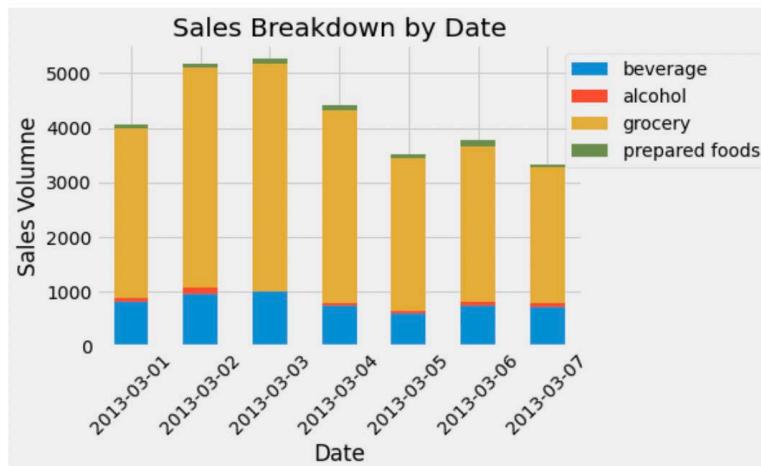
Chart Types

Saving Charts

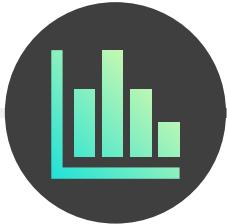
Additional Libraries

Pandas uses the **Matplotlib API** to create charts and visualize data

- This is an integration with the main Matplotlib library



This course only covers Pandas' limited data visualization capabilities, but other full Python libraries like **Matplotlib**, **Seaborn**, and **Plotly** offer more chart types, options and capabilities



# THE PLOT METHOD

The Plot Method

Chart Formatting

Chart Types

Saving Charts

Additional Libraries

You can visualize a DataFrame by using the `.plot()` method

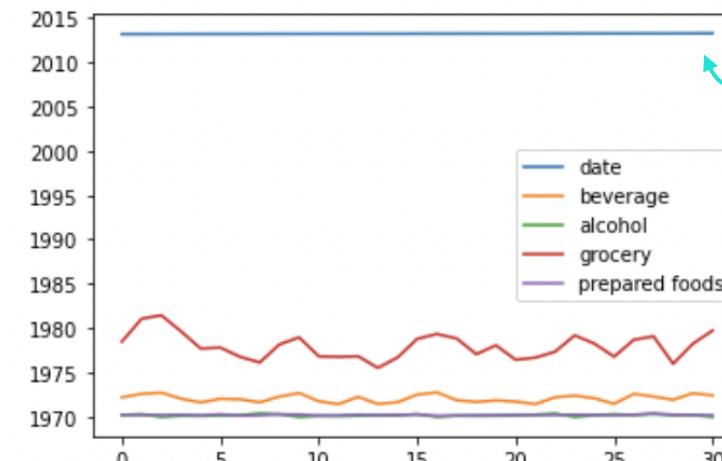
- This creates a **line chart** by default, using the row index as the x-axis and plotting each numerical column as a separate series on the y-axis

`sales_df.head()`

	date	beverage	alcohol	grocery	prepared foods
0	2013-03-01	809.0	75.0	3105.0	81.0
1	2013-03-02	950.0	117.0	4044.0	75.0
2	2013-03-03	1000.0	0.0	4186.0	80.0
3	2013-03-04	750.0	50.0	3525.0	90.0
4	2013-03-05	595.0	41.0	2815.0	60.0

`sales_df.plot()`

`<AxesSubplot:>`



Note that 'date' was plotted as a numeric series

The index is used as the x-axis by default



# THE PLOT METHOD

The Plot Method

Chart Formatting

Chart Types

Saving Charts

Additional Libraries

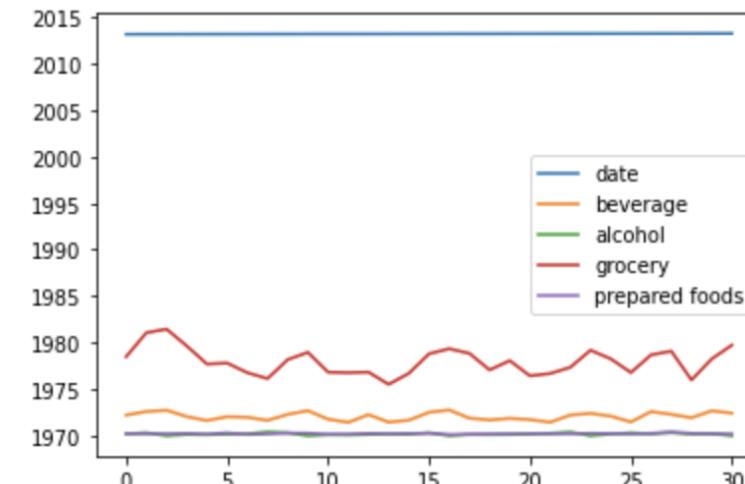
You can visualize a DataFrame by using the `.plot()` method

- This creates a **line chart** by default, using the row index as the x-axis and plotting each numerical column as a separate series on the y-axis

```
sales_df.head()
```

	date	beverage	alcohol	grocery	prepared foods
0	2013-03-01	809.0	75.0	3105.0	81.0
1	2013-03-02	950.0	117.0	4044.0	75.0
2	2013-03-03	1000.0	0.0	4186.0	80.0
3	2013-03-04	750.0	50.0	3525.0	90.0
4	2013-03-05	595.0	41.0	2815.0	60.0

```
sales_df.plot();
```



**PRO TIP:** Add ";" at the end of the code to remove "<AxesSubplot:>" when using Jupyter Notebooks



# THE PLOT METHOD

The Plot Method

Chart Formatting

Chart Types

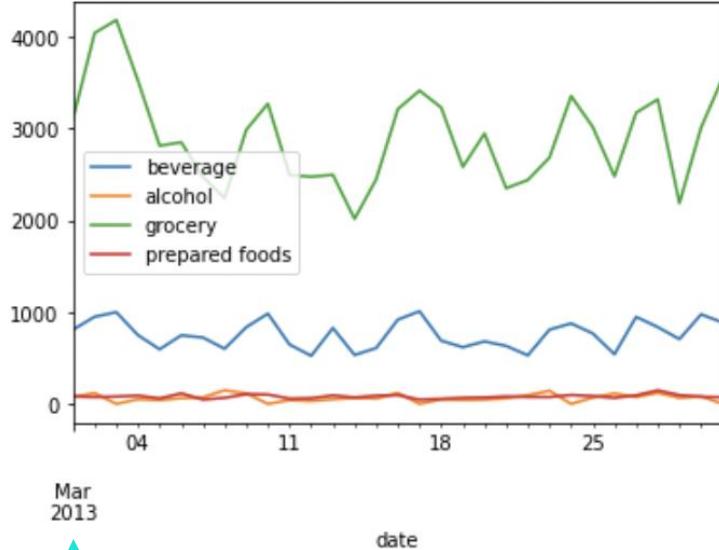
Saving Charts

Additional Libraries

You can **change the x-axis** by setting a different index or using the “x” argument

This sets ‘date’ as the index before plotting the chart

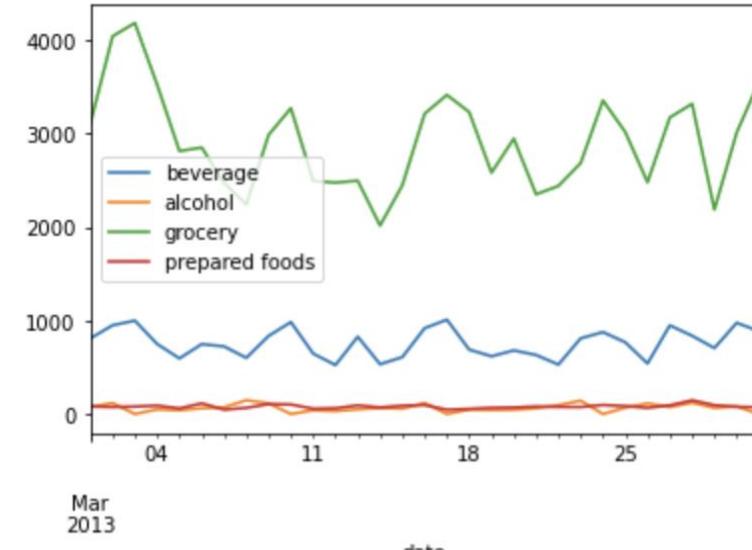
```
sales_df.set_index("date").plot()  
  
<AxesSubplot:xlabel='date'>
```

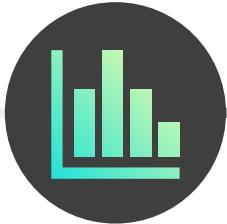


It automatically splits  
the date components!

This sets ‘date’ as the x-axis in the .plot() parameters

```
sales_df.plot(x="date")  
  
<AxesSubplot:xlabel='date'>
```





# THE PLOT METHOD

The Plot Method

Chart Formatting

Chart Types

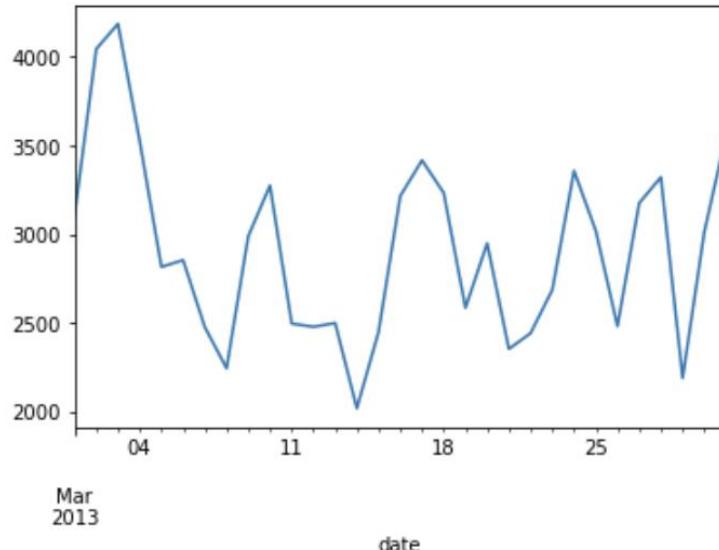
Saving Charts

Additional Libraries

You can **select series** to plot with the `.loc[]` accessor or using the “y” argument

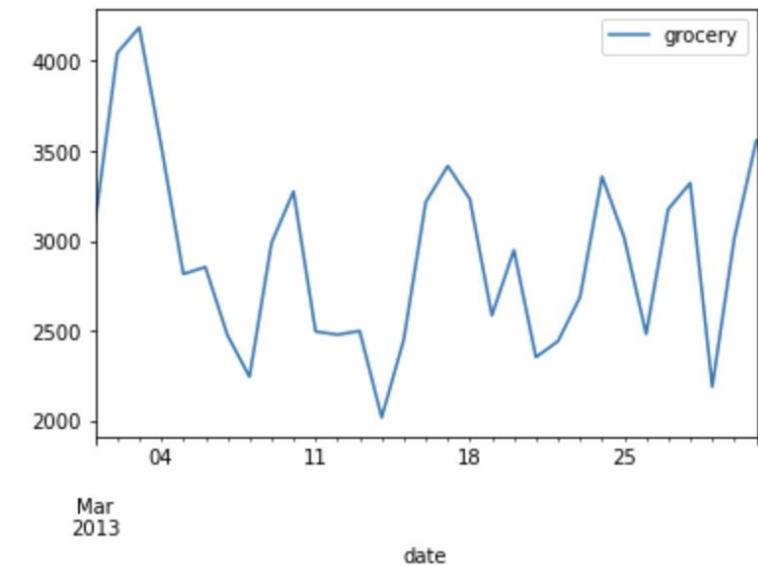
This filters the ‘grocery’ column before plotting the chart

```
sales_df.set_index("date").loc[:, "grocery"].plot()  
<AxesSubplot:xlabel='date'>
```



This plots ‘grocery’ values along the y-axis

```
sales_df.plot(x="date", y="grocery")  
<AxesSubplot:xlabel='date'>
```



# ASSIGNMENT: THE PLOT METHOD

 NEW MESSAGE  
August 14, 2022

**From:** Rachel Revenue (Sr. Financial Analyst)  
**Subject:** Oil Price Trends

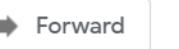
Hi there,

I don't have a great grasp on the trends in oil prices, there are way too many values to understand from table data.

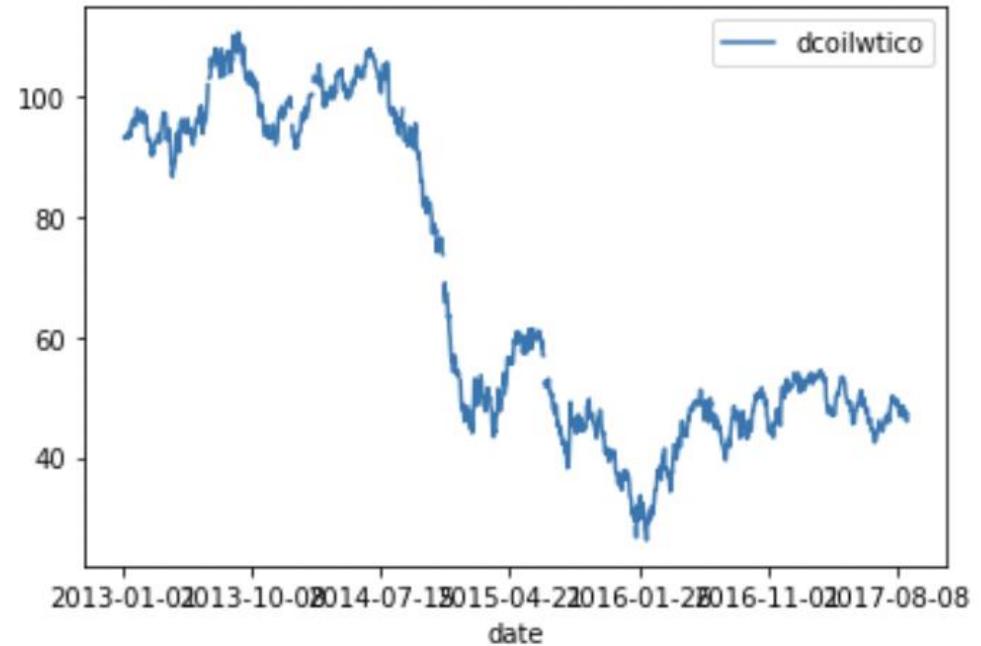
Can you plot the oil prices with a simple line chart?

Can you Google 'oil price decline 2014' to help understand the big drop in prices?

Thanks!

## Results Preview



# SOLUTION: THE PLOT METHOD

 NEW MESSAGE  
August 14, 2022

From: **Rachel Revenue** (Sr. Financial Analyst)  
Subject: Oil Price Trends

Hi there,

I don't have a great grasp on the trends in oil prices, there are way too many values to understand from table data.

Can you plot the oil prices with a simple line chart?

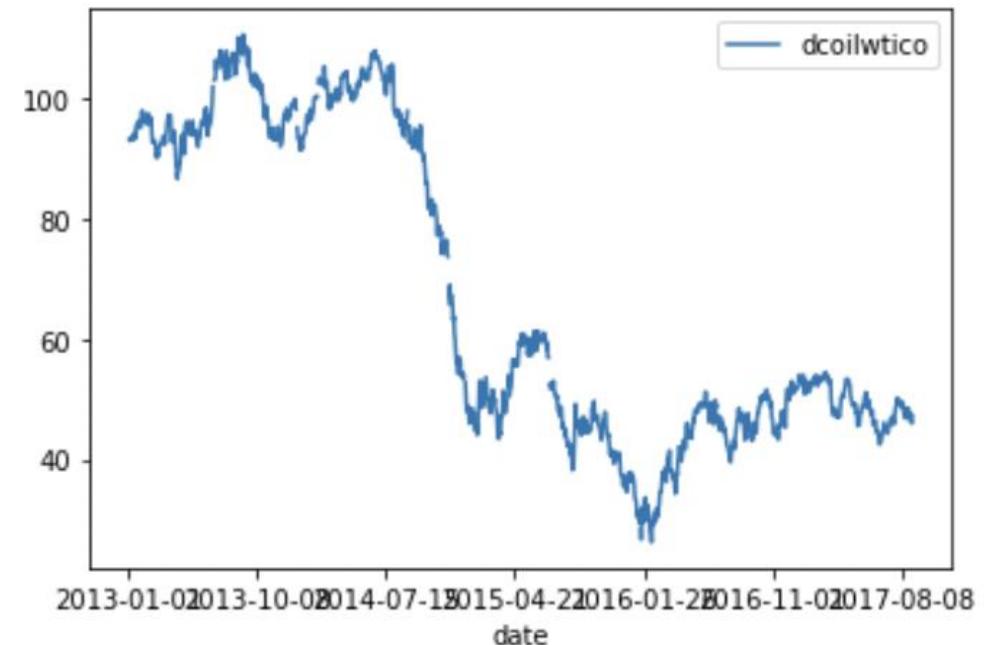
Can you Google 'oil price decline 2014' to help understand the big drop in prices?

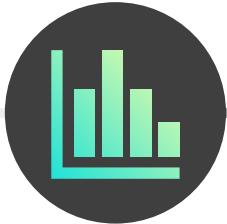
Thanks!

 section05\_data\_viz.ipynb     Reply     Forward

## Solution Code

```
oil.set_axis(oil['date']).plot()
```





# CHART FORMATTING

The Plot Method

Chart Formatting

Chart Types

Saving Charts

Additional Libraries

You can modify the **chart formatting** by using `.plot()` method arguments

- **title = “title”** – title to use for the chart
- **xlabel = “title”** – name to use for the x-axis
- **ylabel = “title”** – name to use for the y-axis
- **color = “color” or “#hexacode”** – color(s) to use for the data series
- **cmap = “color palette”** – preset color palette to apply to the chart
- **style = “symbol”** – style for the line (*dashed, dotted, etc.*)
- **legend = True/False** – adds or removes the legend from the chart
- **rot = degrees** – degrees to rotate the x-axis labels for readability
- **figsize = (width, height)** – size for the chart in inches
- **grid = True/False** – adds gridlines to the chart when True
- **subplots = True/False** – creates a separate chart for each series when True
  - **sharex = True/False** – each series in the subplot shares the x-axis when True
  - **sharey = True/False** – each series in the subplot shares the y-axis when True
  - **layout = (rows, columns)** – the rows and columns to break the subplots into



# CHART TITLES

The Plot Method

Chart Formatting

Chart Types

Saving Charts

Additional Libraries

You can add a custom **chart title** and **axis labels** to increase understanding

```
sales_df.set_index("date").loc[:, "grocery"].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales"  
);
```

Break each .plot() argument into separate rows to make the code easier to read





# SERIES COLORS

The Plot Method

Chart Formatting

Chart Types

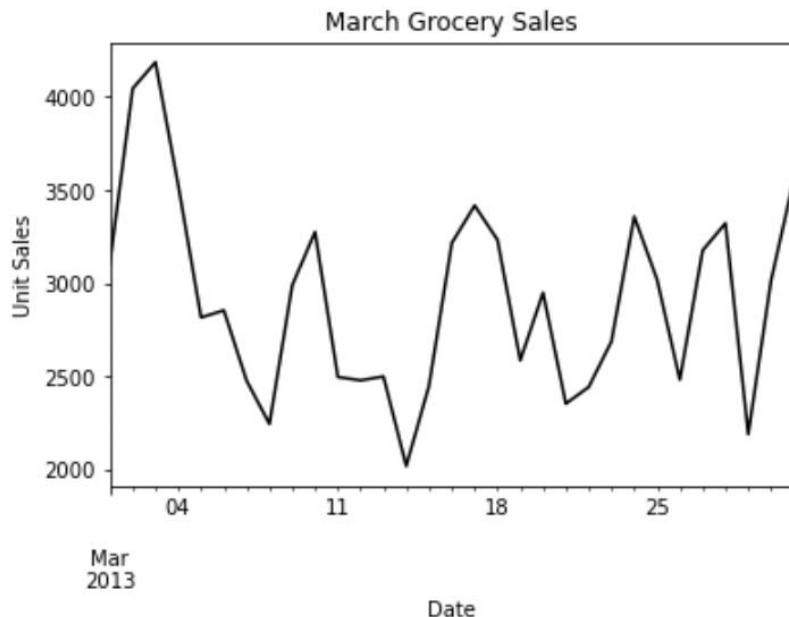
Saving Charts

Additional Libraries

You can modify the **series colors** by using common color names or hex codes

```
sales_df.set_index("date")["grocery"].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color="Black")
```

Note that Python understands basic colors like "Black", "Red", "Green", etc.





# SERIES COLORS

The Plot Method

Chart Formatting

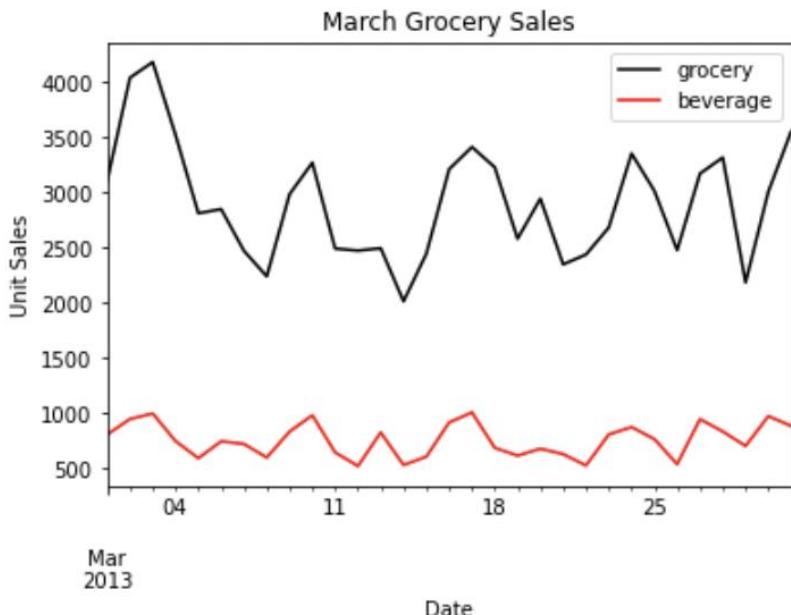
Chart Types

Saving Charts

Additional Libraries

You can modify the **series colors** by using common color names or hex codes

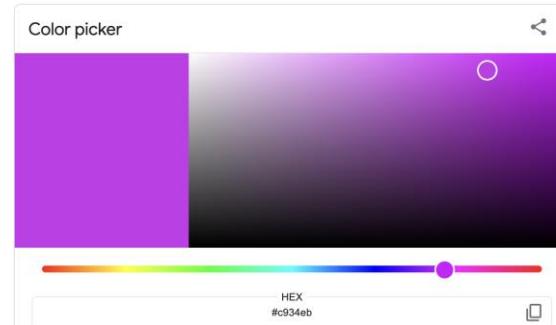
```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color=["#000000", "#FF0000"], Hexadecimal codes work too!  
)
```



If you have more than one series, you can pass the colors in a list



**PRO TIP:** Sites like Google have helpful hexadecimal color pickers





# COLOR PALETTES

The Plot Method

Chart Formatting

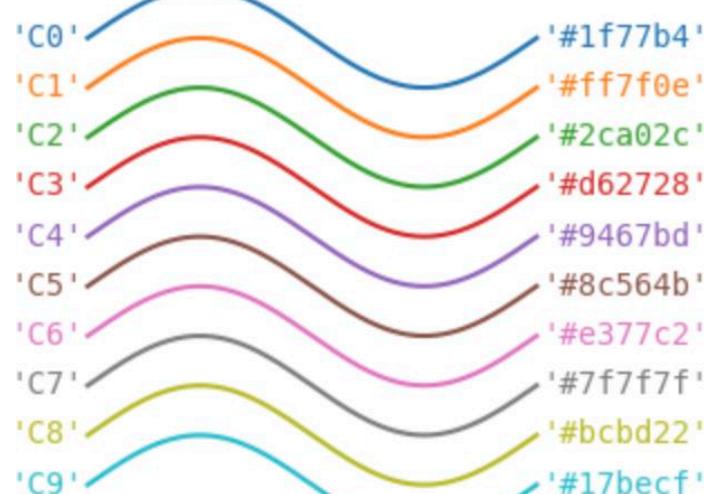
Chart Types

Saving Charts

Additional Libraries

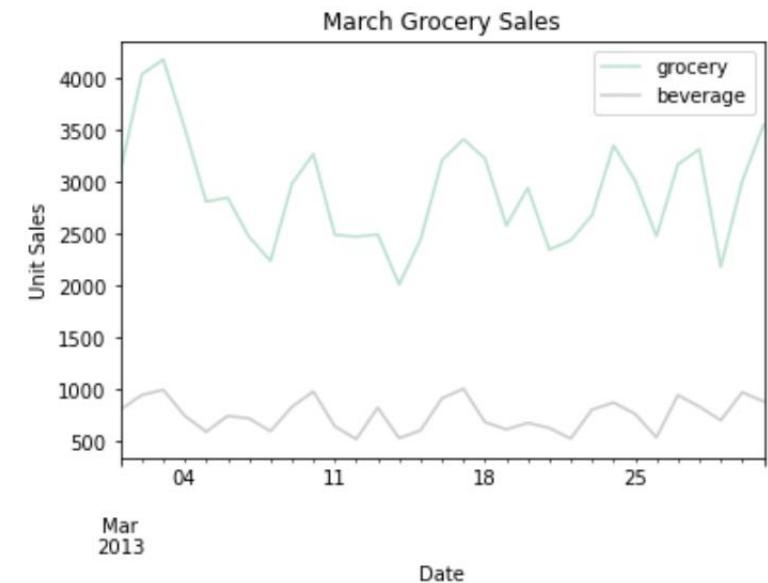
You can also modify the entire **color palette** for the series in the chart

## Default Color Palette:



Series colors are applied  
in this sequential order

```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    cmap="Pastel2",  
)
```



The "Pastel2" color palette is applied here



# LINE STYLE

The Plot Method

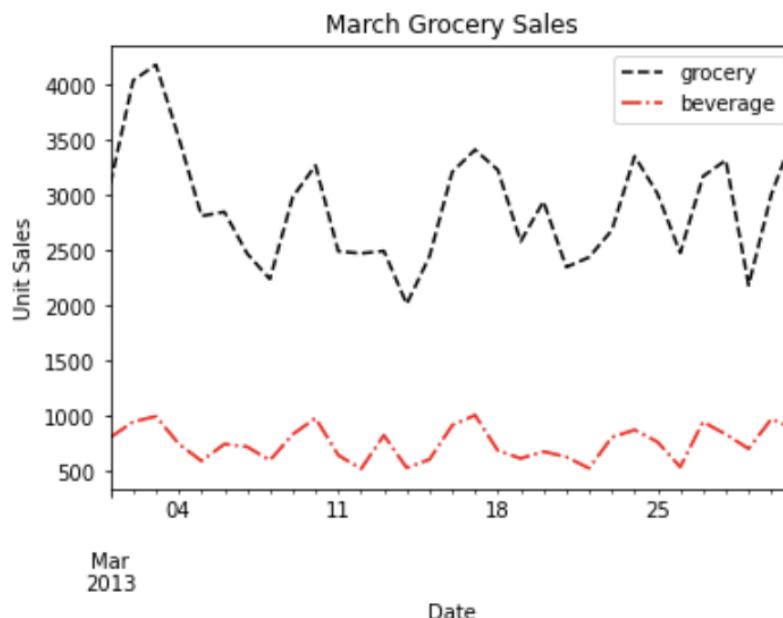
Chart Formatting

Chart Types

Saving Charts

Additional Libraries

```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color=["Black", "Red"],  
    style=[ "--", "-."],  
);
```



This sets the first series as a dashed line and the second as dot-dashed

Symbol	Line Style
-	Solid (default)
--	Dashed
-.	Dash-dot
..	Dotted



# CHART LEGEND

The Plot Method

Chart Formatting

Chart Types

Saving Charts

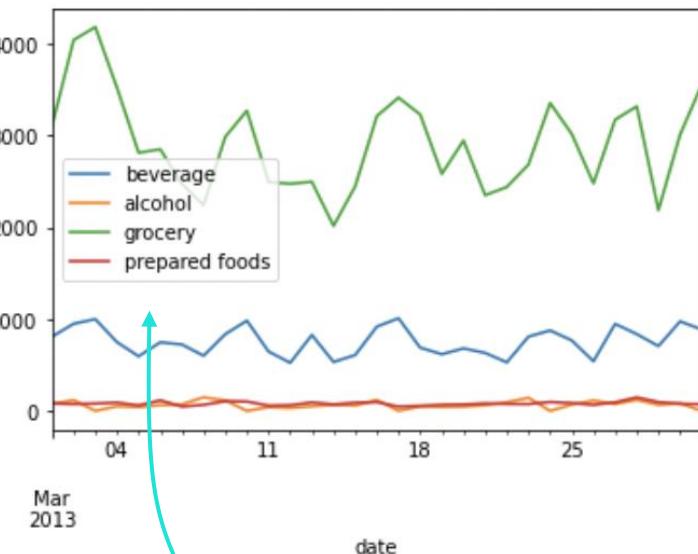
Additional Libraries

The “legend” .plot() argument lets you **add or remove the legend**

- This can be useful in some scenarios, but in others you’ll want to *reposition* the legend

```
sales_df.set_index("date").plot()
```

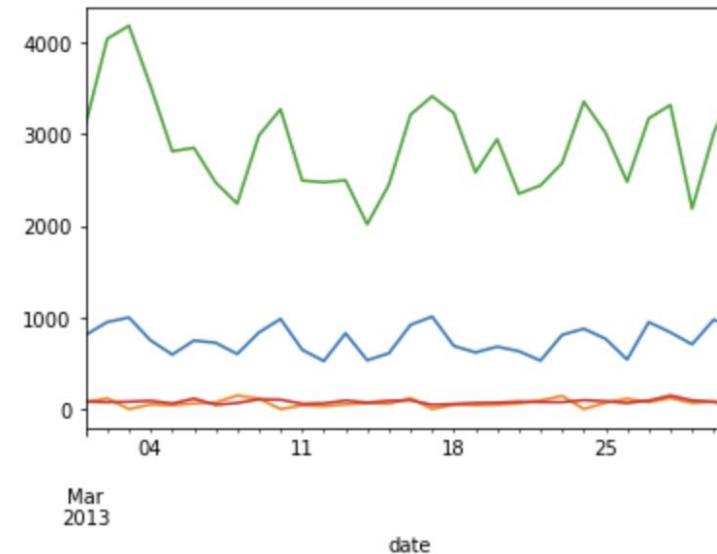
```
<AxesSubplot:xlabel='date'>
```



Pandas tries to place  
it in the “best” spot

```
sales_df.set_index("date").plot(legend=False)
```

```
<AxesSubplot:xlabel='date'>
```





# CHART LEGEND

The Plot Method

Chart Formatting

Chart Types

Saving Charts

Additional Libraries

## Location Options

best (default)

upper right

upper left

upper center

lower right

lower left

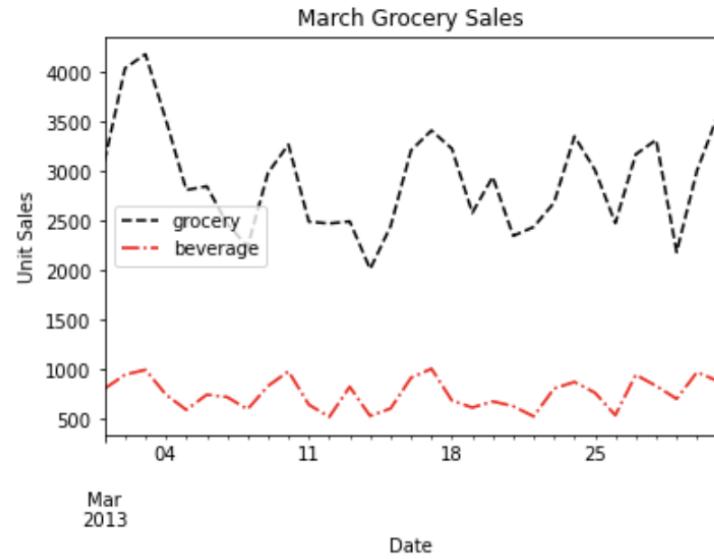
lower center

center right

center left

center

```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color=["Black", "Red"],  
    style=[ "--", "-."],  
    ).legend(loc="center left");
```





# CHART LEGEND

The Plot Method

Chart Formatting

Chart Types

Saving Charts

Additional Libraries

## Location Options

best (default)

upper right

upper left

upper center

lower right

lower left

lower center

center right

center left

center

```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color=["Black", "Red"],  
    style=[ "--", "-."],  
    .legend(bbox_to_anchor=(1.3, 1));
```



“bbox\_to\_anchor” lets you specify  
the coordinates for the legend  
(it takes some trial and error!)



# GRIDLINES

The Plot Method

Chart Formatting

Chart Types

Saving Charts

Additional Libraries

You can add **gridlines** to charts for visual reference

```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color=["Black", "Red"],  
    style=[ "--", "-."],  
    grid=True,  
).legend(bbox_to_anchor=(1.3, 1))
```



This adds gridlines on the y-axis for line charts, but other chart types get x and y gridlines



# PRO TIP: CHART STYLES

The Plot Method

Chart Formatting

Chart Types

Saving Charts

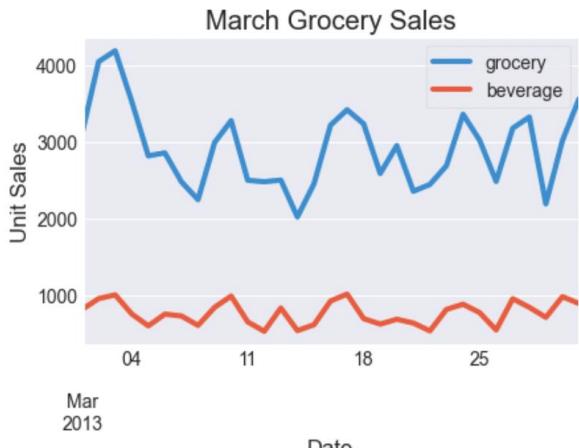
Additional Libraries

Matplotlib & Seaborn have premade **style templates** that can be applied to charts

- Once a style is set, it will automatically be applied to all charts

```
import seaborn as sns
sns.set_style("darkgrid") ← The style is set in advance

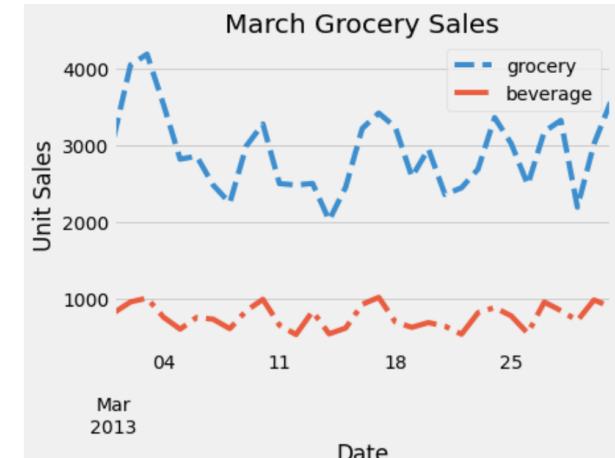
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(
    title="March Grocery Sales",
    xlabel="Date",
    ylabel="Unit Sales",
);
```



This is Seaborn's **darkgrid** style, which has better font sizing, and a background color

```
import matplotlib
matplotlib.style.use("fivethirtyeight")

sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(
    title="March Grocery Sales",
    xlabel="Date",
    ylabel="Unit Sales",
    # color=["Black", "Red"],
    style=[ "--", "-." ], ← You can still customize each chart
);
```



This is Matplotlib's **fivethirtyeight** style, but with dashed lines instead of solid

# ASSIGNMENT: FORMATTED CHART

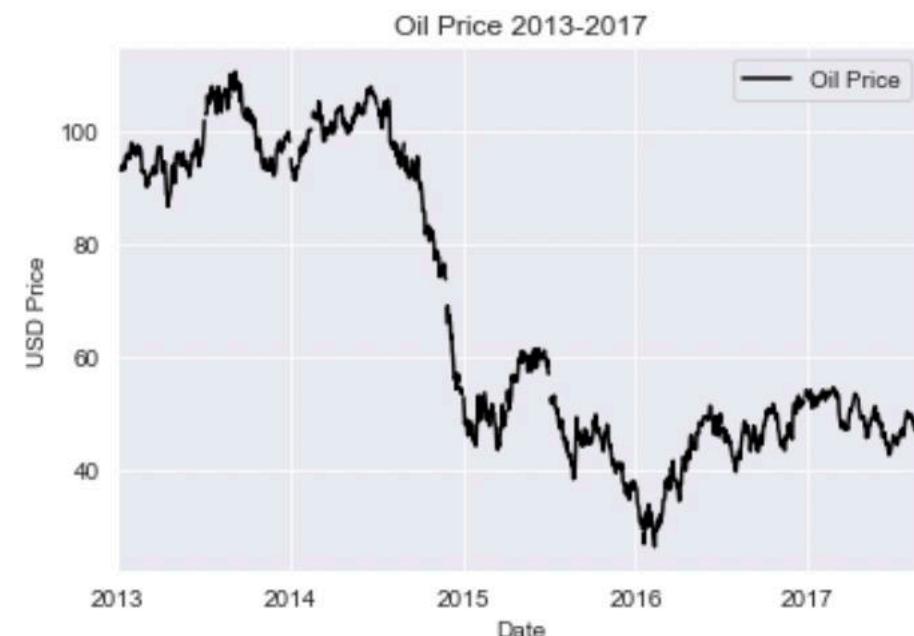
 **NEW MESSAGE**  
August 15, 2022

**From:** **Rachel Revenue** (Sr. Financial Analyst)  
**Subject:** Oil Price Presentation

Hi there, thanks for helping me visualize the price trends.  
Can you format it so it's more suitable for management?  
Our company tends to use "darkgrid" for the style, but I'd also like a chart title, axis labels, the line to be in black, and to rename the price column to a more intuitive name.  
Finally, consider converting the 'date' column to datetime64 – I've heard this will create better labels on the x-axis.  
Thanks!

 section05\_data\_viz.ipynb       Reply       Forward

## Results Preview



# SOLUTION: FORMATTED CHART

 **NEW MESSAGE**  
August 15, 2022

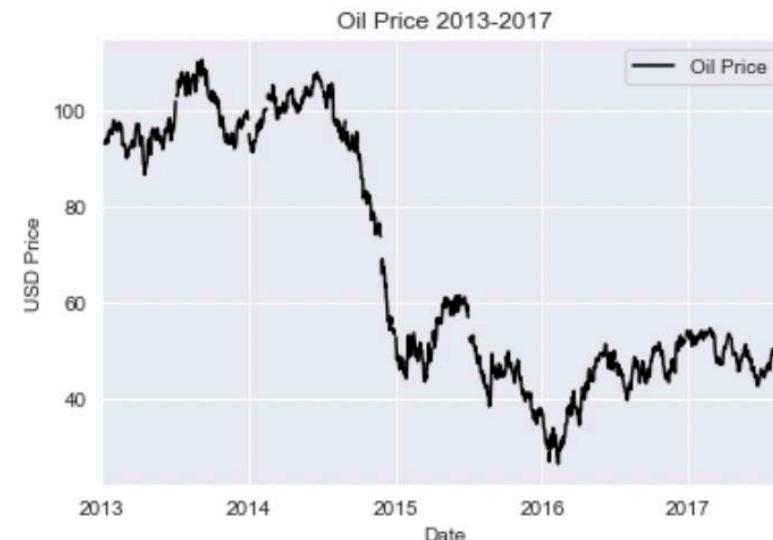
**From:** **Rachel Revenue** (Sr. Financial Analyst)  
**Subject:** Oil Price Presentation

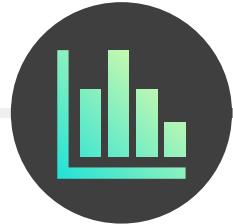
Hi there, thanks for helping me visualize the price trends.  
Can you format it so it's more suitable for management?  
Our company tends to use "darkgrid" for the style, but I'd also like a chart title, axis labels, the line to be in black, and to rename the price column to a more intuitive name.  
Finally, consider converting the 'date' column to datetime64 – I've heard this will create better labels on the x-axis.  
Thanks!

## Solution Code

```
oil.set_axis(oil[ "date" ].astype("Datetime64")).plot(  
    title="Oil Price 2013-2017",  
    xlabel="Date",  
    ylabel="USD Price",  
    c="Black"  
)
```





# SUBPLOTS

The Plot Method

Chart Formatting

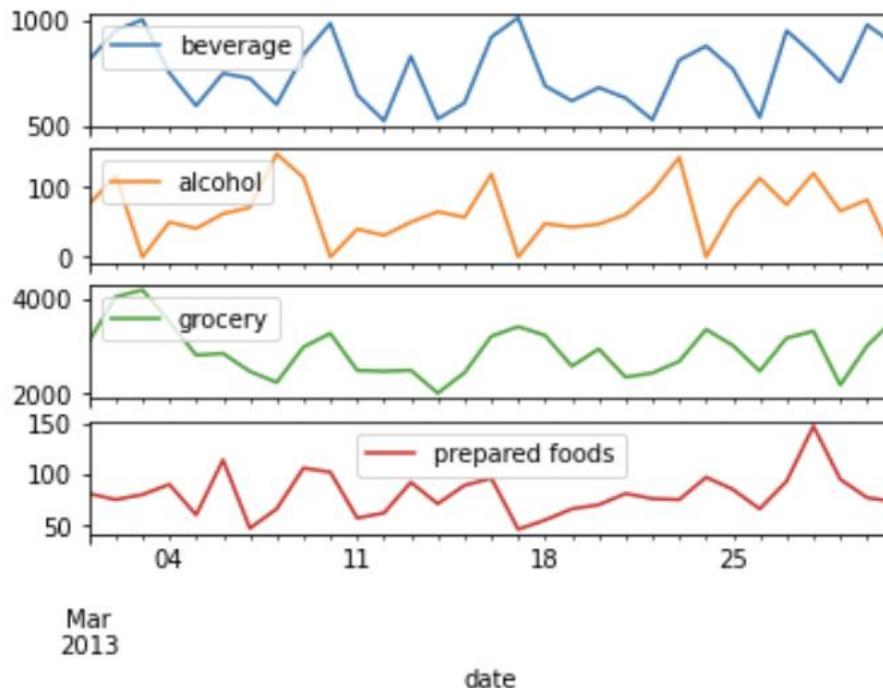
Chart Types

Saving Charts

Additional Libraries

You can leverage **subplots** to create a separate chart for each series

```
sales_df.set_index("date").plot(subplots=True);
```



The subplots are stacked in a single column by default



# SUBPLOTS

The Plot Method

Chart Formatting

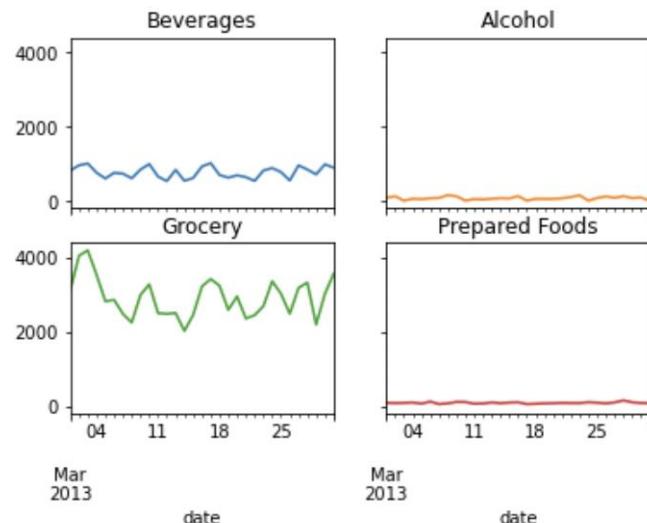
Chart Types

Saving Charts

Additional Libraries

You can leverage **subplots** to create a separate chart for each series

```
sales_df.set_index("date").plot(  
    subplots=True,  
    layout=(2, 2),  
    sharey=True,  
    legend=False,  
    title=["Beverages", "Alcohol", "Grocery", "Prepared Foods"],  
)
```



There are several subplot options you can use:

- **layout** lets you specify the rows and columns for the subplots
- **sharey** & **sharex** let you set consistent axes across the subplots
- **title** can map a list of specified titles to each subplot



# CHART SIZE

The Plot Method

Chart Formatting

Chart Types

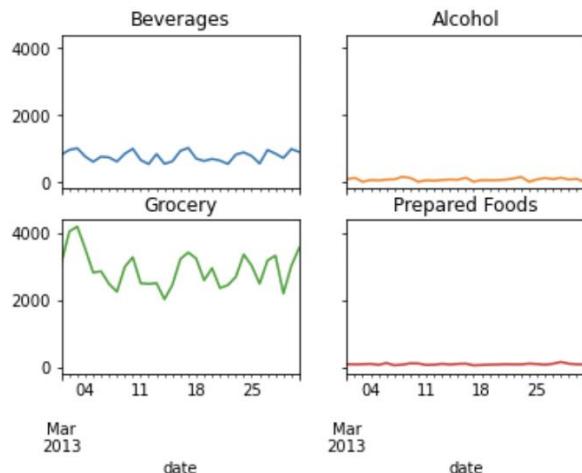
Saving Charts

Additional Libraries

You can adjust the **size** of the plot or subplots (in inches) using “`figsize`”

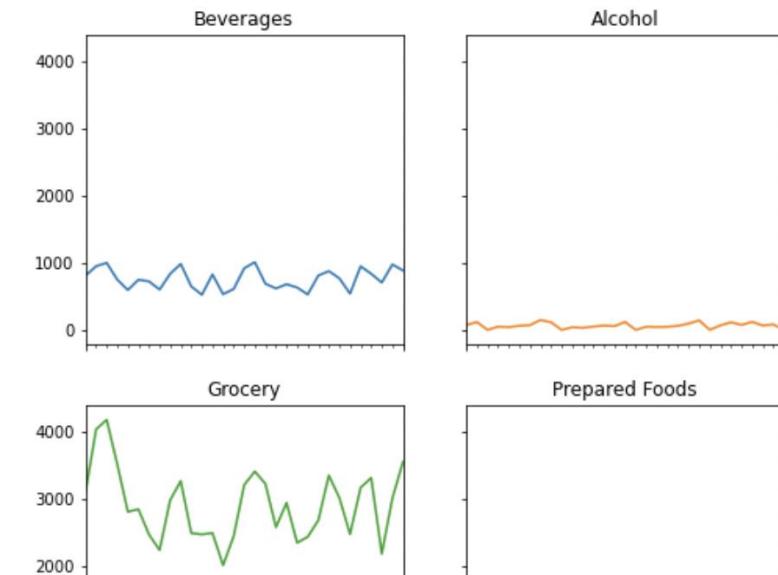
The default size is 6.4 x 4.8 inches

```
sales_df.set_index("date").plot(  
    subplots=True,  
    layout=(2, 2),  
    sharey=True,  
    legend=False,  
    title=["Beverages", "Alcohol", "Grocery", "Prepared Foods"],  
)
```



This has been resized to 8 x 8 inches

```
sales_df.set_index("date").plot(  
    figsize=(8, 8),  
    subplots=True,  
    layout=(2, 2),  
    sharey=True,  
    legend=False,  
    title=["Beverages", "Alcohol", "Grocery", "Prepared Foods"],  
)
```



# ASSIGNMENT: SUBPLOTS

  NEW MESSAGE  
August 16, 2022

**From:** Joey Justin Time (Logistics Clerk)  
**Subject:** Transactions by Store

Hi there, nice to meet you!

I need some help assessing the volumes moving through each store, so I've provided a pivot table with store transactions.

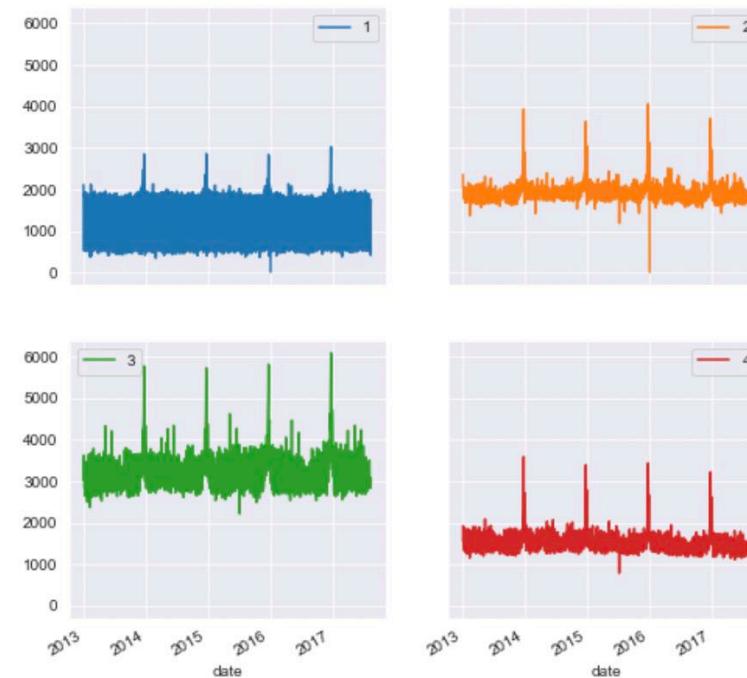
Can you create a 2 x 2 plot the transactions for each of our stores in a line chart? Make sure they share an axis and consider making the figure a bit larger.

Which stores have the most transactions? Do they spike at similar times?

Thanks again!

## Results Preview



# SOLUTION: SUBPLOTS

  NEW MESSAGE  
August 16, 2022

**From:** Joey Justin Time (Logistics Clerk)  
**Subject:** Transactions by Store

Hi there, nice to meet you!

I need some help assessing the volumes moving through each store, so I've provided a pivot table with store transactions.

Can you create a 2 x 2 plot the transactions for each of our stores in a line chart? Make sure they share an axis and consider making the figure a bit larger.

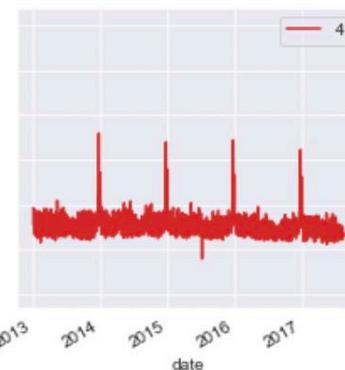
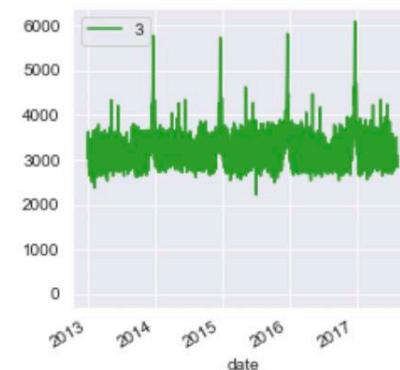
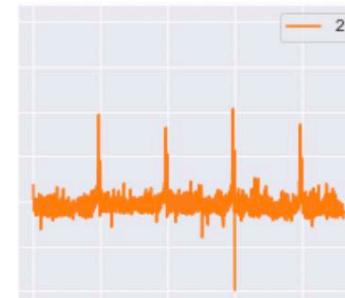
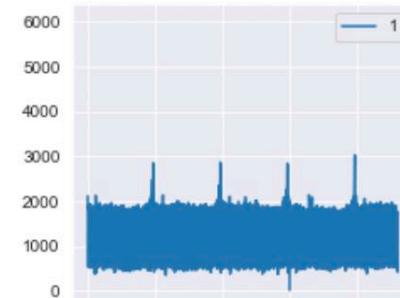
Which stores have the most transactions? Do they spike at similar times?

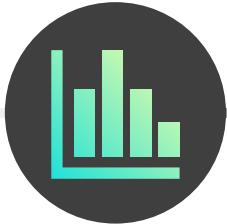
Thanks again!

## Solution Code

```
stores_1234.plot(subplots=True,  
                  layout=(2, 2),  
                  figsize=(8, 8),  
                  sharey=True)
```





# CHANGING CHART TYPES

The Plot Method

Chart Formatting

Chart Types

Saving Charts

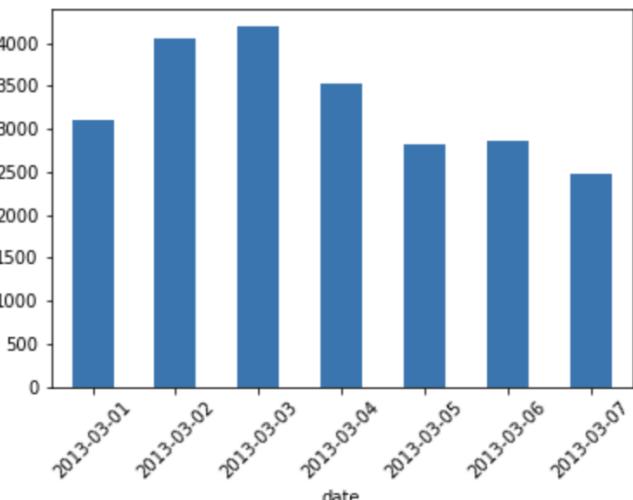
Additional Libraries

You can **change chart types** with the “kind” argument or the attribute for each chart

This selects the chart type within the `.plot()` parameters

```
(sales_df.set_index(sales_df["date"].dt.date).iloc[:7, 3]
 .plot(kind='bar',
       y="grocery",
       rot=45))
```

`<AxesSubplot:xlabel='date'>`

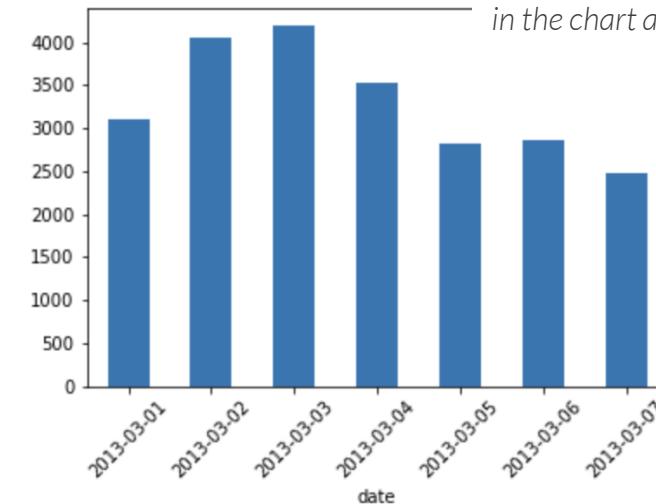


This chains the chart type as an attribute to `.plot()`

```
(sales_df.set_index(sales_df["date"].dt.date).iloc[:7, 3]
 .plot.bar(y="grocery",
           rot=45))
```

`<AxesSubplot:xlabel='date'>`

Note that the arguments live  
in the chart attribute now



PREFERRED



# COMMON CHART TYPES

The Plot Method

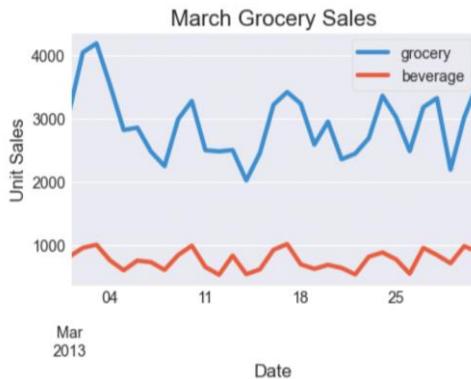
Chart Formatting

Chart Types

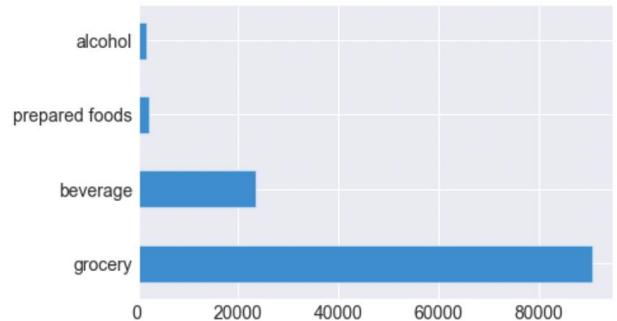
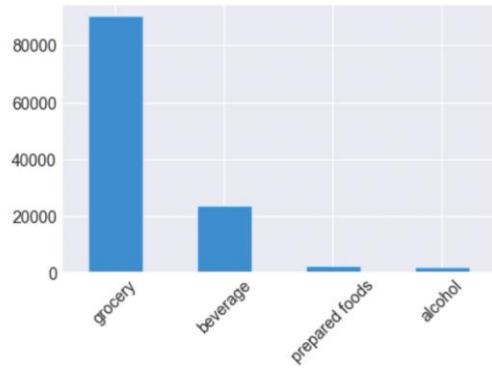
Saving Charts

Additional Libraries

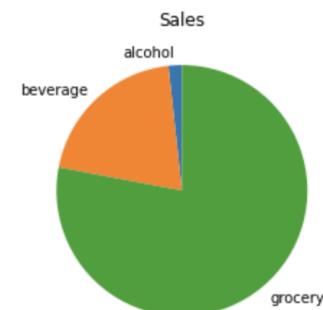
## LINE CHART (*default*)



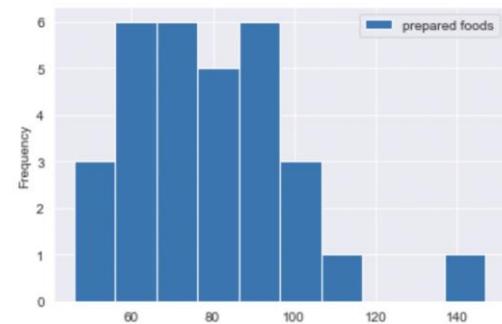
## BAR CHART (bar or barh)



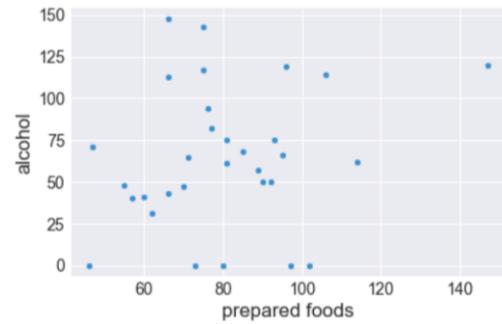
## PIE CHART (pie)



## HISTOGRAM (hist)



## SCATTERPLOT (scatter)





# LINE CHARTS

The Plot Method

Chart Formatting

Chart Types

Saving Charts

Additional Libraries

```
sales_df.set_index("date").loc[:, ["grocery", "beverage"]].plot(  
    title="March Grocery Sales",  
    xlabel="Date",  
    ylabel="Unit Sales",  
    color=["Black", "Red"],  
    style=[ "--", "-." ],  
    grid=True,  
    rot=45,  
);
```

They are the default chart type when using .plot()



The x-axis should be continuous!



# BAR CHARTS

The Plot Method

Chart Formatting

Chart Types

Saving Charts

Additional Libraries

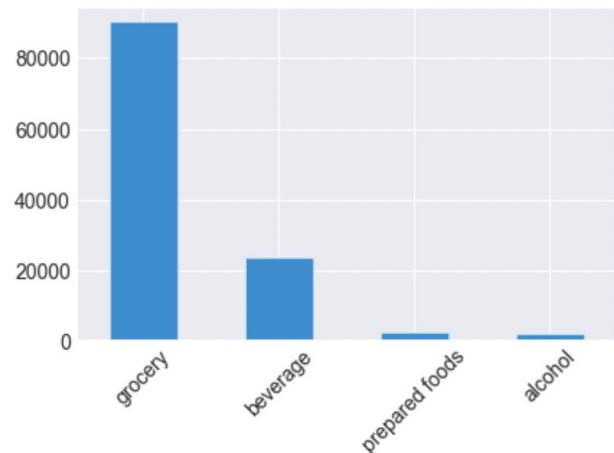
```
sales_summary = (
    sales_df[["beverage", "alcohol", "grocery", "prepared foods"]]
    .sum()
    .sort_values(ascending=False)
)

sales_summary
```

grocery	90429.0
beverage	23567.0
prepared foods	2490.0
alcohol	2000.0

You'll typically aggregate  
the data by category first

```
sales_summary.plot.bar(rot=45)
```



"bar" will return a vertical  
bar chart (or column chart),  
and "rot" rotates the labels



# BAR CHARTS

The Plot Method

Chart Formatting

Chart Types

Saving Charts

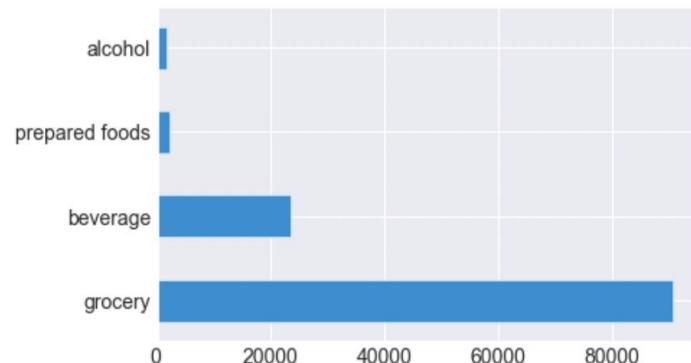
Additional Libraries

```
sales_summary = (
    sales_df[["beverage", "alcohol", "grocery", "prepared foods"]]
    .sum()
    .sort_values(ascending=False)
)

sales_summary
```

grocery	90429.0
beverage	23567.0
prepared foods	2490.0
alcohol	2000.0

```
sales_summary.plot.bart()
```



You'll typically aggregate the data by category first

"bart" will return a horizontal bar chart



Note that horizontal bar charts are sorted in reverse order



# GROUPED BAR CHARTS

The Plot Method

Chart Formatting

Chart Types

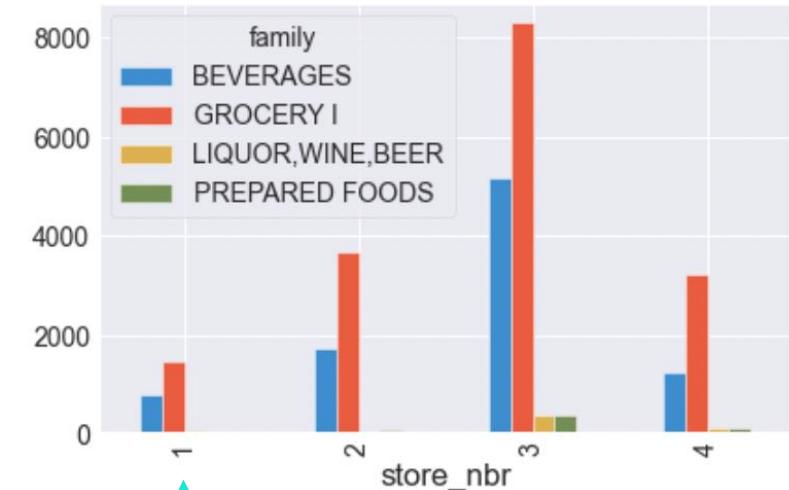
Saving Charts

Additional Libraries

store\_sales\_by\_family

family	BEVERAGES	GROCERY I	LIQUOR,WINE,BEER	PREPARED FOODS
store_nbr				
1	783.5	1469.5	79.5	45.5000
2	1740.5	3672.0	52.5	100.7355
3	5180.0	8312.5	382.0	386.4820
4	1247.5	3209.5	116.0	116.7000

store\_sales\_by\_family.plot.bar()



The bars are grouped by  
the DataFrame index



# STACKED BAR CHARTS

The Plot Method

Chart Formatting

Chart Types

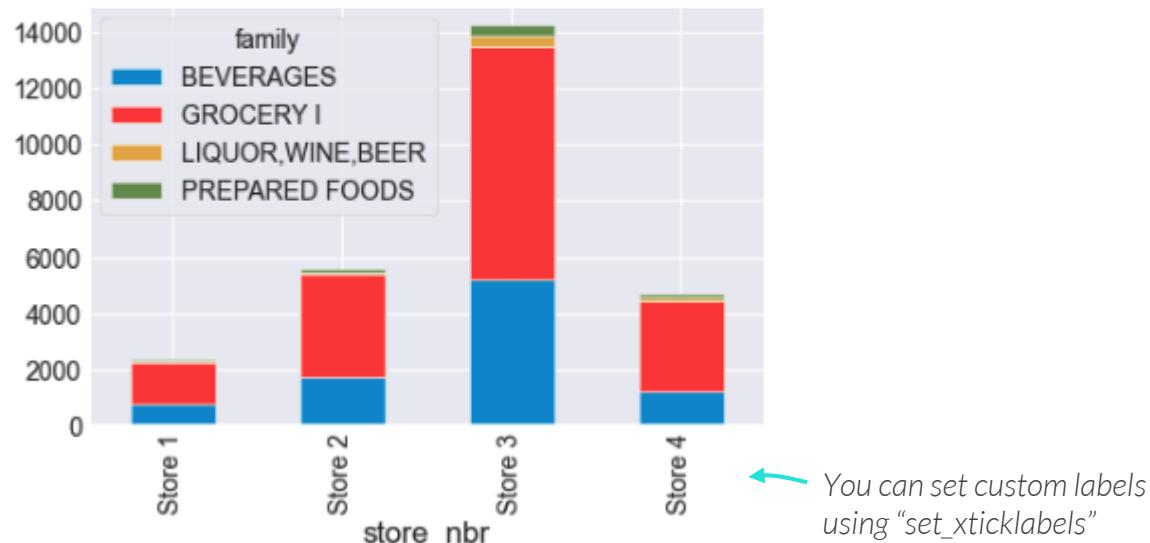
Saving Charts

Additional Libraries

Specify “stacked=True” when plotting multiple series to create a **stacked bar chart**

- This still lets you compare the categories, but also shows the composition of each category

```
(store_sales_by_family.plot.bar(stacked=True).set_xticklabels(  
    labels=["Store 1", "Store 2", "Store 3", "Store 4"]  
))
```





# 100% STACKED BAR CHARTS

The Plot Method

Chart Formatting

Chart Types

Saving Charts

Additional Libraries

Calculate percent of total values by index to create a **100% stacked bar chart**

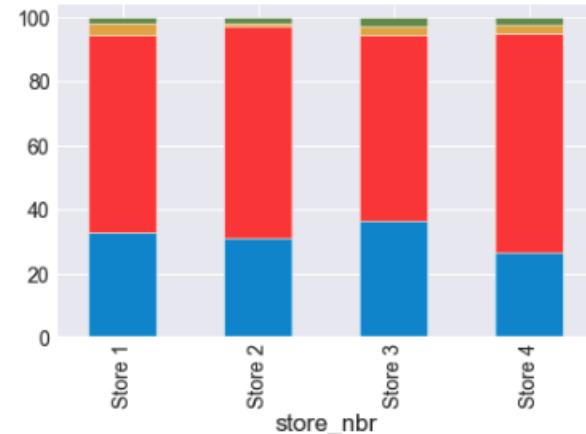
- This emphasizes the difference in composition between categories (*instead of absolute values*)

```
stacked_data = store_sales_by_family.apply(lambda x: x * 100 / sum(x), axis=1)  
stacked_data
```

	family	BEVERAGES	GROCERY I	LIQUOR,WINE,BEER	PREPARED FOODS
store_nbr					
1	family	32.947855	61.795627	3.343146	1.913373
2	store_nbr	31.271698	65.975108	0.943272	1.809923
3	family	36.322884	58.288412	2.678637	2.710066
4	store_nbr	26.600849	68.437213	2.473506	2.488432

The values in each row add up to 100

```
(stacked_data.plot.bar(stacked=True, legend=False).set_xticklabels(  
    labels=["Store 1", "Store 2", "Store 3", "Store 4"]  
)  
)
```



# ASSIGNMENT: BAR CHARTS

 **1 NEW MESSAGE**  
August 17, 2022

**From:** Joey Justin Time (Logistics Clerk)  
**Subject:** Transactions by Store

Hi there, thanks for your help!

Can we summarize total transactions by sales in a bar chart?

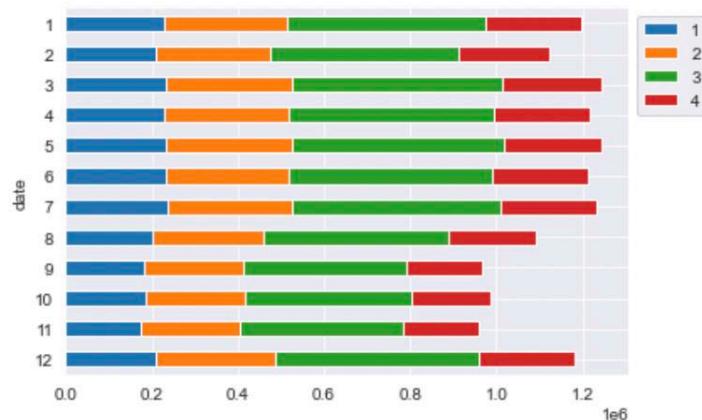
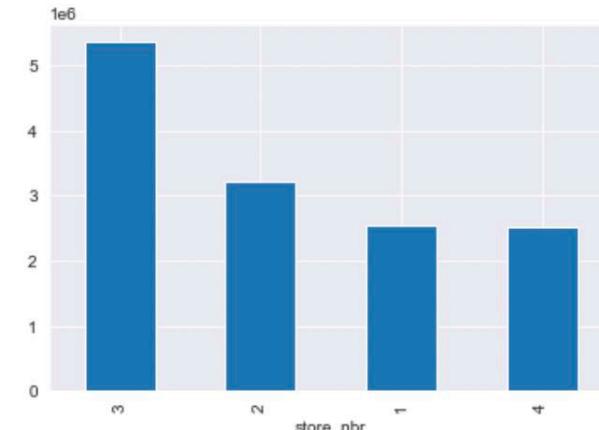
Then, create a horizontal bar chart with the y axis as month, and the x axis as sales, sorted from January to February.

Some code has been provided in the notebook attached to create the table for the stacked bar plot.

Thanks!

 section05\_data\_viz.ipynb Reply Forward

## Results Preview



# SOLUTION: BAR CHARTS

 **1 NEW MESSAGE**  
August 17, 2022

**From:** Joey Justin Time (Logistics Clerk)  
**Subject:** Transactions by Store

Hi there, thanks for your help!

Can we summarize total transactions by sales in a bar chart?

Then, create a horizontal bar chart with the y axis as month, and the x axis as sales, sorted from January to February.

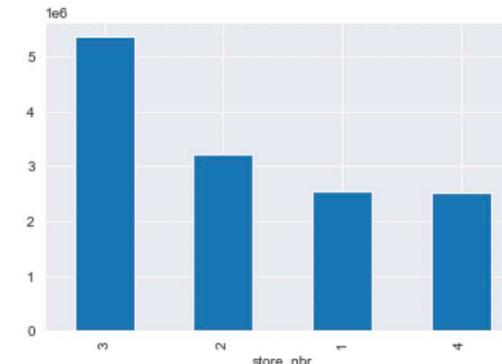
Some code has been provided in the notebook attached to create the table for the stacked bar plot.

Thanks!

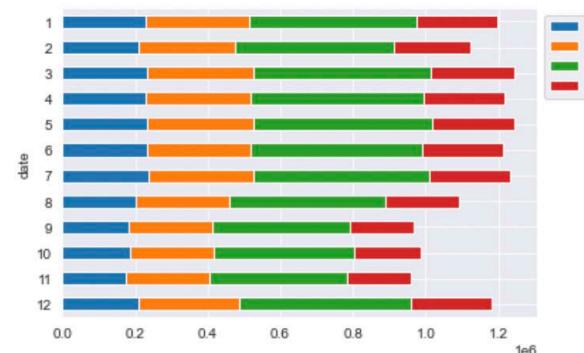
 section05\_data\_viz.ipynb Reply Forward

## Solution Code

```
stores_1234.sum().sort_values(ascending=False).plot.bar()
```



```
stores_1234_monthly.sort_index(ascending=False).plot.banh(stacked=True).legend(  
    bbox_to_anchor=(1, 1)  
)
```





# PIE CHARTS

The Plot Method

Chart Formatting

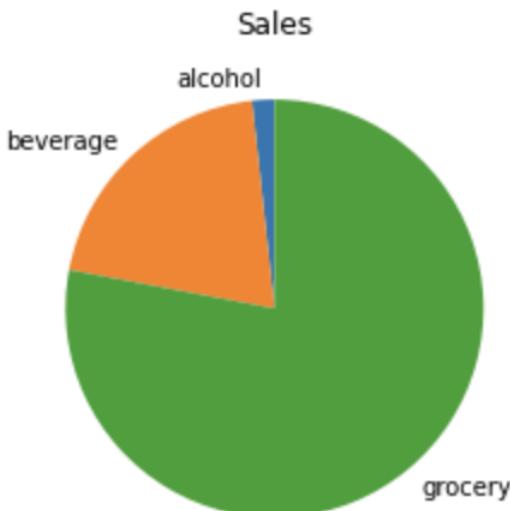
Chart Types

Saving Charts

Additional Libraries

**Pie charts** are used for showing composition with categorical data

```
(sales_df.loc[:, ["beverage", "alcohol", "grocery"]]  
    .sum()  
    .sort_values()  
    .plot.pie(title="Sales", ylabel="", startangle=90)  
)
```



**PRO TIP:** Pie charts get a bad rep, but they can be effective as long as:

- # of categories  $\leq 5$
- Slices are sorted
- First slice starts at  $0^\circ$



# SCATTERPLOTS

The Plot Method

Chart Formatting

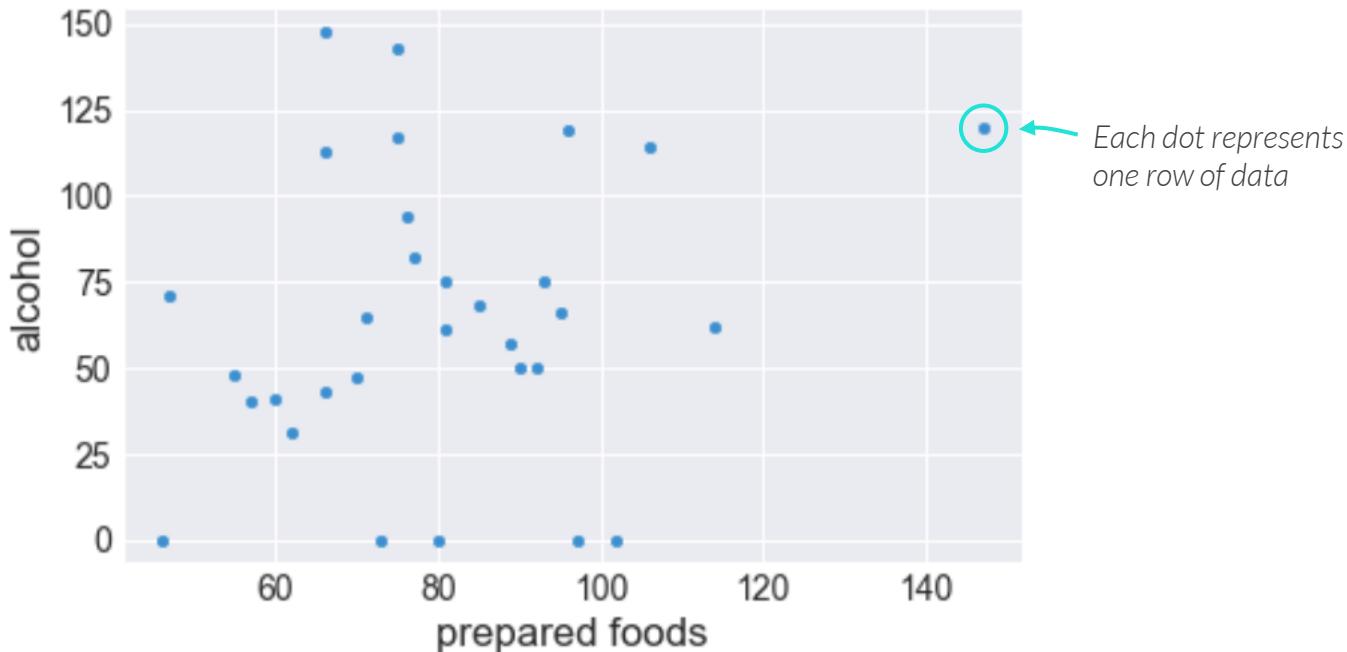
Chart Types

Saving Charts

Additional Libraries

```
sales_df.plot.scatter(x="prepared foods", y="alcohol");
```

Just specify the x and y values!





# SCATTERPLOTS

The Plot Method

Chart Formatting

Chart Types

Saving Charts

Additional Libraries

```
sales_df.plot.scatter(  
    x="prepared foods",  
    y="alcohol",  
    title="Alcohol vs. Prepared Food Sales",  
    xlabel="Prepared Food Sales",  
    ylabel="Alcohol Sales",  
    c=sales_df["date"].dt.dayofweek,  
    colormap="Set2",  
)
```



This sets a different color based on "dayofweek"

# ASSIGNMENT: SCATTERPLOTS

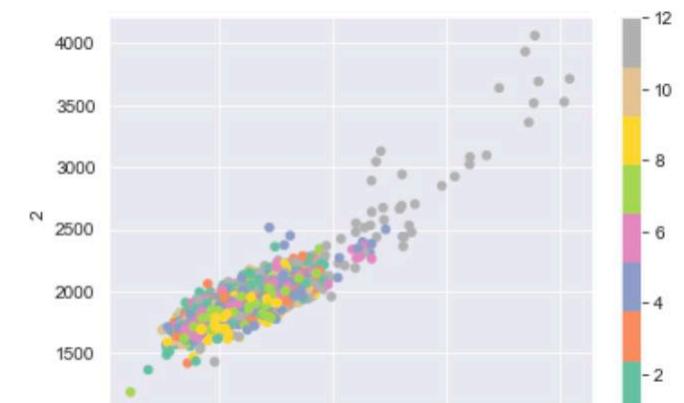
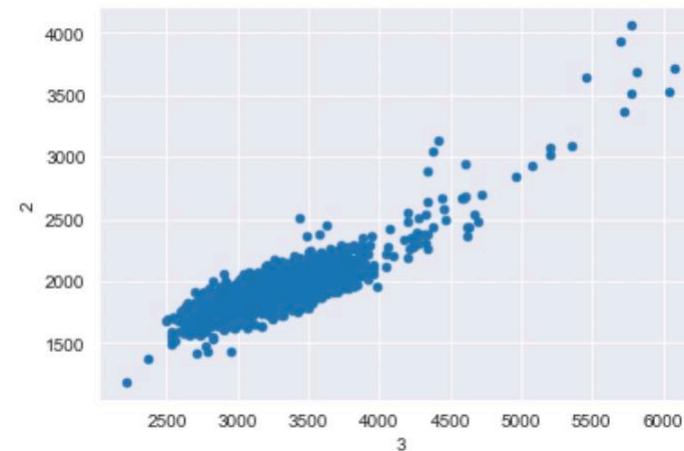
  NEW MESSAGE  
August 18, 2022

**From:** Joey Justin Time (Logistics Clerk)  
**Subject:** Relationship Between Store Sales

Hi there,  
I was curious to see whether stores tend to have high sales at the same time.  
Can you make a scatterplot of the relationship between store 2 and 3 sales?  
Then, color the dots by month to check if the relationship is different across time.  
Thanks!

## Results Preview



# SOLUTION: SCATTERPLOTS

  NEW MESSAGE  
August 18, 2022

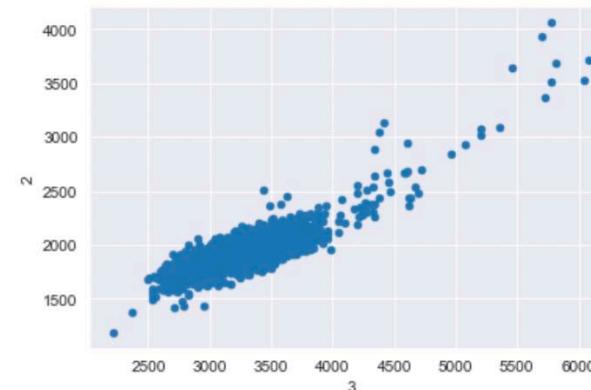
**From:** Joey Justin Time (Logistics Clerk)  
**Subject:** Relationship Between Store Sales

Hi there,  
I was curious to see whether stores tend to have high sales at the same time.  
Can you make a scatterplot of the relationship between store 2 and 3 sales?  
Then, color the dots by month to check if the relationship is different across time.  
Thanks!

 section05\_data\_viz.ipynb Reply Forward

## Solution Code

```
stores_1234.plot.scatter(x=3, y=2)
```



```
stores_1234.plot.scatter(x=3, y=2, c=stores_1234.index.month, colormap="Set2")
```





# HISTOGRAMS

The Plot Method

Chart Formatting

Chart Types

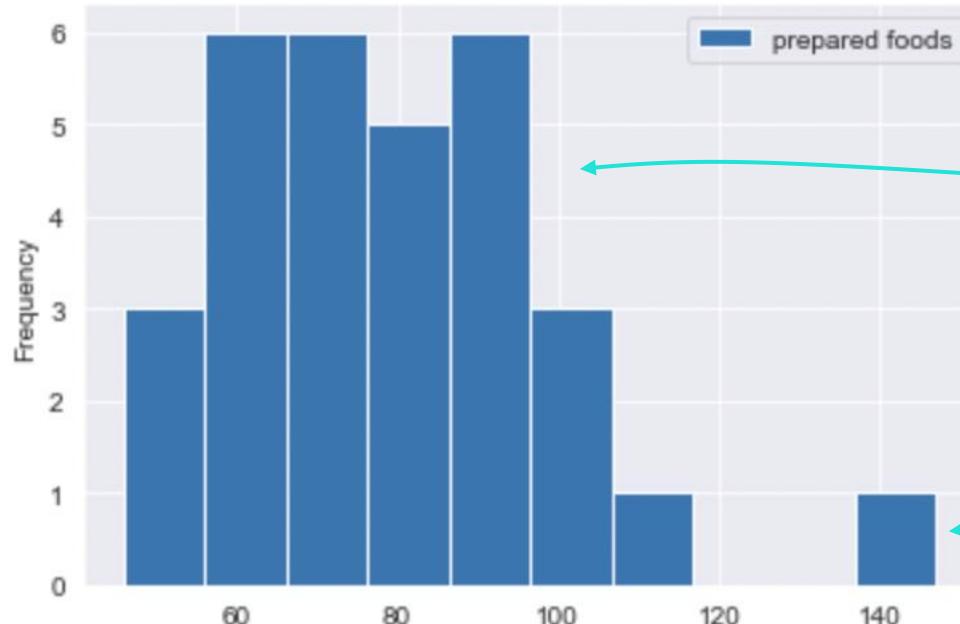
Saving Charts

Additional Libraries

**Histograms** are used for showing the distribution of numerical series

- It divides the data in “bins” and plots the frequency of values that fall into each bin as bars

```
sales_df[["prepared foods"]].plot.hist()
```



This is great for seeing the “shape” of the data, in this case centered around 70

And identifying outliers!



# HISTOGRAMS

The Plot Method

Chart Formatting

Chart Types

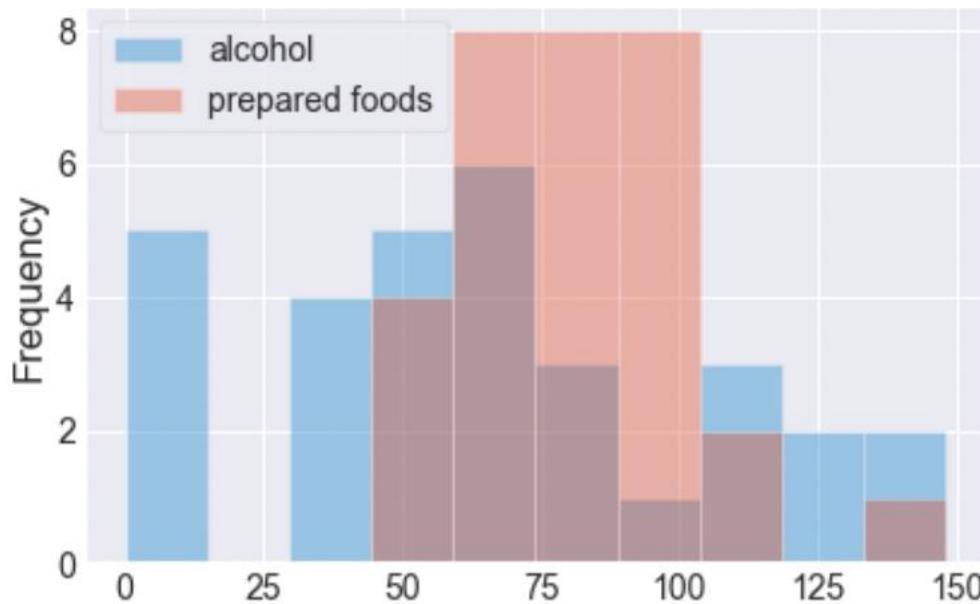
Saving Charts

Additional Libraries

**Histograms** are used for showing the distribution of numerical series

- It divides the data in “bins” and plots the frequency of values that fall into each bin as bars

```
sales_df[["alcohol", "prepared foods"]].plot.hist(alpha=0.4)
```



“alpha” sets the transparency  
(0 is invisible, 1 is solid)

Plotting multiple series lets you compare their distributions and understand drivers of statistics like mean, min, max and standard deviation

# ASSIGNMENT: HISTOGRAMS

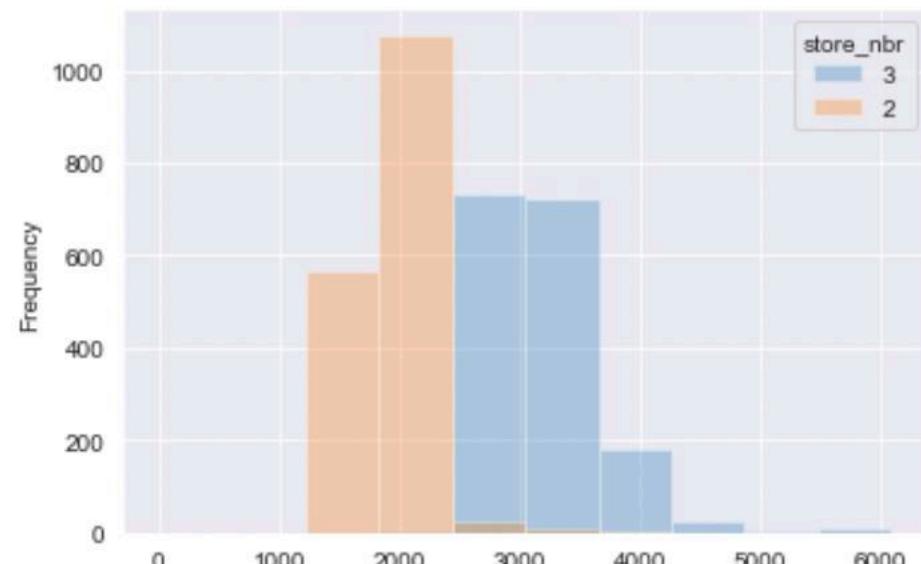
  NEW MESSAGE  
August 20, 2022

From: **Joey Justin Time** (Logistics Clerk)  
Subject: Transaction Distributions

Hi there,  
Can you plot the distributions of sales for stores 2 and 3?  
I want to see how often store 2 has sales similar to store 3, as well as if there are any outlier type days.  
Thanks!

## Results Preview



# SOLUTION: HISTOGRAMS

 NEW MESSAGE  
August 20, 2022

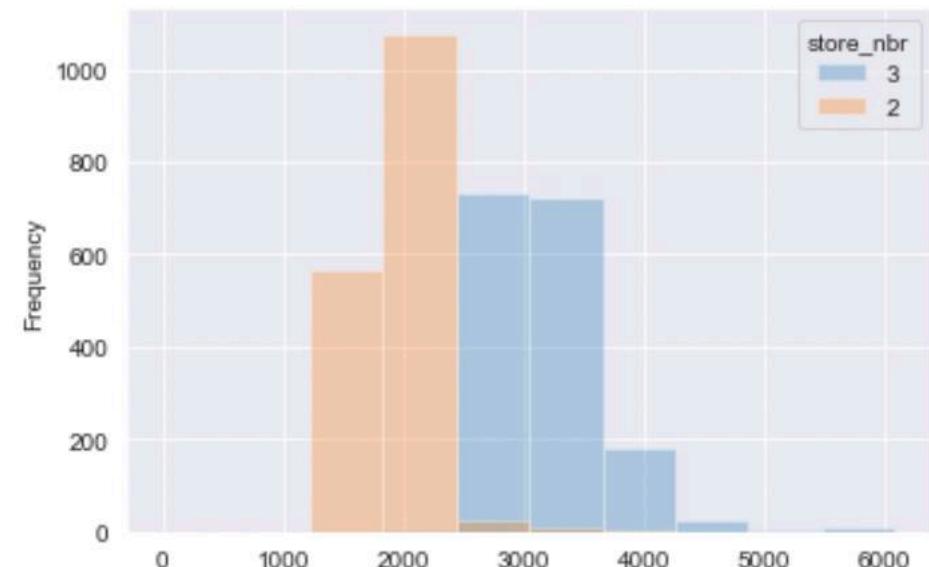
From: **Joey Justin Time** (Logistics Clerk)  
Subject: Transaction Distributions

Hi there,  
Can you plot the distributions of sales for stores 2 and 3?  
I want to see how often store 2 has sales similar to store 3, as well as if there are any outlier type days.  
Thanks!

## Solution Code

```
stores_1234.loc[:, [3, 2]].plot.hist(alpha=0.3)
```





# SAVING PLOTS

The Plot Method

Chart Formatting

Chart Types

Saving Charts

Additional Libraries

```
plot = (
    sales_df.set_index("date")
    .loc[:, ["grocery", "beverage"]]
    .plot(
        title="March Grocery Sales",
        xlabel="Date",
        ylabel="Unit Sales",
        color=["Black", "Red"],
        style=[ "--", "-." ],
        grid=True,
    )
    .legend(bbox_to_anchor=(1.05, 1))
)

plot.figure.savefig("March_Grocery_Sales.png", bbox_inches="tight")
```

Just specify the file name & extension  
(bbox\_inches="tight" pads the image  
to include the legend)



**PRO TIP:** You can also take a screenshot of the plot and save it





# MORE DATA VISUALIZATION

The Plot Method

Chart Formatting

Chart Types

Saving Charts

Additional  
Libraries

There is **much more to data visualization** in Python, and most libraries are built to be compatible with Pandas Series and DataFrames

For more chart types, customization, and interactivity, consider these libraries:



**Matplotlib** is what we've used to visualize data so far, but there are more chart types and customization options available (*which can make it challenging to work with*)



**Seaborn** is an extension of Matplotlib that is easier to work with and produces better looking visuals – it also introduces new chart types!



**Plotly** is used for creating beautiful, interactive visuals, and has a sister app (Dash) that allows you to easily create dashboards as web applications



**Folium** is great for creating geospatial maps, especially since it contains features that are not available in the rest of the libraries

# KEY TAKEAWAYS

---



The **.plot()** method lets you visualize the data in a DataFrame

- *This creates a line chart by default, using the row index as the x-axis and plotting each numerical column as a separate series on the y-axis, but can easily be modified using .plot() arguments*



Use .plot() arguments to modify the **chart formatting** options

- *You can customize the chart titles, axis labels, colors, styles, legend, and more*



Consider the **chart's purpose** before plotting to communicate effectively

- *The charts we covered have a specific purpose – line charts show trends over time, bar charts show comparison across categories, scatterplots show relationships between series, and histograms show the data's distribution*



There is **more to data visualization** with Python and Pandas

- *Once you're comfortable with the basics covered in this course, take a look at the other visualization libraries if you need new chart types, want to add interactivity, or wish to create online dashboards*

# MID-COURSE PROJECT

# PROJECT DATA: OVERVIEW

## Transactions

```
transactions.head(5)
```

	household_key	BASKET_ID	DAY	PRODUCT_ID	QUANTITY	SALES_VALUE	STORE_ID	RETAIL_DISC	WEEK_NO	COUPON_DISC	COUPON_MATCH_DISC
0	1364	26984896261	1	842930	1	2.19	31742	0.00	1	0.0	0.0
1	1364	26984896261	1	897044	1	2.99	31742	-0.40	1	0.0	0.0
2	1364	26984896261	1	920955	1	3.09	31742	0.00	1	0.0	0.0
3	1364	26984896261	1	937406	1	2.50	31742	-0.99	1	0.0	0.0
4	1364	26984896261	1	981760	1	0.60	31742	-0.79	1	0.0	0.0

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2146311 entries, 0 to 2146310
Data columns (total 11 columns):
 #   Column           Dtype  
 --- 
 0   household_key   int64  
 1   BASKET_ID       int64  
 2   DAY              int64  
 3   PRODUCT_ID      int64  
 4   QUANTITY         int64  
 5   SALES_VALUE     float64
 6   STORE_ID         int64  
 7   RETAIL_DISC     float64
 8   WEEK_NO          int64  
 9   COUPON_DISC     float64
 10  COUPON_MATCH_DISC float64
dtypes: float64(4), int64(7)
memory usage: 180.1 MB
```

```
cols = [ "QUANTITY", "SALES_VALUE", "RETAIL_DISC", "COUPON_DISC", "COUPON_MATCH_DISC"]
```

```
transactions.loc[:, cols].describe().round()
```

	QUANTITY	SALES_VALUE	RETAIL_DISC	COUPON_DISC	COUPON_MATCH_DISC
count	2146311.0	2146311.0	2146311.0	2146311.0	2146311.0
mean	101.0	3.0	-1.0	-0.0	-0.0
std	1152.0	4.0	1.0	0.0	0.0
min	0.0	0.0	-130.0	-56.0	-8.0
25%	1.0	1.0	-1.0	0.0	0.0
50%	1.0	2.0	0.0	0.0	0.0
75%	1.0	3.0	0.0	0.0	0.0
max	89638.0	840.0	4.0	0.0	0.0



**MAVEN**  
MEGA MART

# PROJECT DATA: OVERVIEW

## Products

PRODUCT_ID	MANUFACTURER	DEPARTMENT	BRAND	COMMODITY_DESC	SUB_COMMODITY_DESC	CURR_SIZE_OF_PRODUCT
0	25671	2	GROCERY	National	FRZN ICE	ICE - CRUSHED/CUBED
1	26081	2	MISC. TRANS.	National	NO COMMODITY DESCRIPTION	NO SUBCOMMODITY DESCRIPTION
2	26093	69	PASTRY	Private	BREAD	BREAD:ITALIAN/FRENCH
3	26190	69	GROCERY	Private	FRUIT - SHELF STABLE	APPLE SAUCE
4	26355	69	GROCERY	Private	COOKIES/CONES	SPECIALTY COOKIES

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 92353 entries, 0 to 92352
Data columns (total 7 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   PRODUCT_ID      92353 non-null   int64  
 1   MANUFACTURER    92353 non-null   int64  
 2   DEPARTMENT      92353 non-null   object  
 3   BRAND            92353 non-null   object  
 4   COMMODITY_DESC   92353 non-null   object  
 5   SUB_COMMODITY_DESC 92353 non-null   object  
 6   CURR_SIZE_OF_PRODUCT 92353 non-null   object  
dtypes: int64(2), object(5)
memory usage: 31.1 MB
```



**MAVEN**  
MEGA MART

# ASSIGNMENT: MID-COURSE PROJECT



## NEW MESSAGE

August 23, 2022

**From:** Ross Retail (Head of Analytics)  
**Subject:** Maven Acquisition Target Data

Hi there,

This is going to be a deep dive analysis presented to senior management at our company. Maven MegaMart is planning to acquire another retailer to expand our market share. As part of the due diligence process, they've sent us over several tables relating to their customers and sales.

I've taken a quick look, but given your great work so far I want you to lead on this. There are some more detailed questions in the attached notebook.

Thanks!

section06\_midcourse\_project.ipynb

Reply

Forward

## Key Objectives

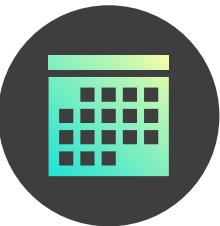
1. Read in data from multiple csv files
2. Explore the data (*millions of rows!*)
3. Create new columns to aid in analysis
4. Filter, sort, and aggregate the data to pinpoint and summarize important information
5. Build plots to communicate key insights



**MAVEN**  
MEGA MART

# TIME SERIES

# TIME SERIES



In this section we'll cover **time series** in Pandas, which leverages the datetime data type to extract date components, group by dates, and perform calculations like moving averages

## TOPICS WE'LL COVER:

The Datetime Data Type

Formatting & Parting

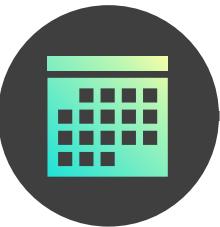
Time Deltas

Datetime Indices

Shifting & Aggregating

## GOALS FOR THIS SECTION:

- Understand the nuances of the datetime data types in base Python and Pandas
- Apply custom date formats and extract date and time components from datetime data
- Access portions of time series data and offset time series for period-by-period comparison
- Create custom time periods and reshape your data to fit time periods of interest



# THE DATETIME DATA TYPE

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

Base Python's **datetime** data type lets you work with time series data

- Datetimes include both a date and time portion by default

```
from datetime import datetime  
  
now = datetime.now()  
  
now
```

```
datetime.datetime(2022, 6, 3, 13, 44, 35, 495577)
```

Date components

Time components

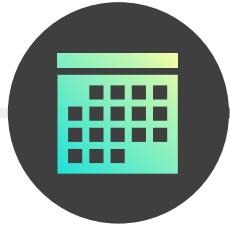
The **datetime.now()** function returns the current date and time, which is great for comparing time deltas from the dates in the data to the present

```
type(now)
```

```
datetime.datetime
```



Datetime data type!



# DATETIME64

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

NumPy's **datetime64** data type lets you work with datetimes in DataFrames

- Pandas will treat dates as "object" datatypes until converted to "datetime64"

sales

	date	sales
0	2022-01-01	10
1	2022-01-02	15
2	2022-02-01	<NA>
3	2022-03-01	20
4	2023-01-01	30
5	2023-01-02	35

```
sales = sales.astype({"date": "datetime64", "sales": "Int16"})
```

sales

	date	sales
0	2022-01-01	10
1	2022-01-02	15
2	2022-02-01	<NA>
3	2022-03-01	20
4	2023-01-01	30
5	2023-01-02	35

sales.dtypes

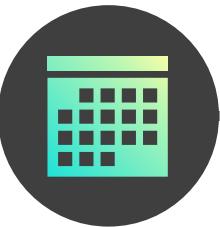
```
date      object  
sales     object  
dtype: object
```



sales.dtypes

```
date      datetime64[ns]  
sales     Int16  
dtype: object
```

Pandas correctly converted the strings to dates!



# CONVERSION LIMITATIONS

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

The .astype() method can convert strings to datetimes, but has some **limitations**:

1. Any values that Pandas can't correctly identify as dates will return an error

	order_date	delivery_date
0	2021-04-04	2021-04-07
1	2021-05-23	N/A

```
shipping_dates.astype({"order_date": "datetime64", "delivery_date": "datetime64"})
```

ParserError: Unknown string format: N/A

The missing value here is formatted as a string

2. Ignoring the errors will return the Series as an “object” data type, not datetime64

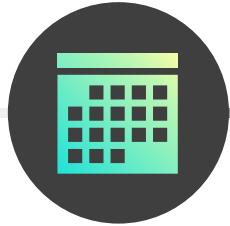
```
shipping_dates.astype(  
    {"order_date": "datetime64", "delivery_date": "datetime64"}, errors="ignore")
```

	order_date	delivery_date
0	2021-04-04	2021-04-07
1	2021-05-23	N/A

```
shipping_dates.dtypes
```

order_date	datetime64[ns]
delivery_date	object
dtype:	object

You can specify **errors="ignore"** to return the DataFrame, but the conversion to datetime won’t work



# TO\_DATETIME

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

The `.to_datetime()` method is a more flexible option for converting datetimes

```
shipping_dates["delivery_date"] = pd.to_datetime(  
    shipping_dates["delivery_date"],  
    errors="coerce",  
    infer_datetime_format=True,  
    # format="%Y-%M-%D",  
)
```

	order_date	delivery_date
0	2021-04-04	2021-04-07
1	2021-05-23	NaT

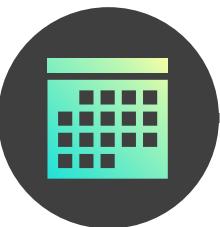
← This stands for “not a time”

```
shipping_dates.dtypes
```

```
order_date      datetime64[ns]  
delivery_date   datetime64[ns] ← This was converted correctly!  
dtype: object
```

To convert using `.to_datetime()`:

- Pass the column(s) to convert
- Specify `errors="coerce"` to convert errors to NaT
- Use `infer_datetime_format=True` to let Pandas intelligently guess the date format of the text strings
- Or use `format=` to specify the datetime format yourself, especially if Pandas can't guess correctly



# DATETIME CODES

You can use these **datetime codes** to format dates or extract date components:

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

## Date:

Code	Example	Description
%D	07/14/2022	Zero-padded date ( <i>based on OS settings</i> )
%Y	2022	Four-digit year
%m	07	Month of the year
%B	July	Full month name ( <i>based on OS settings</i> )
%d	14	Day of the month
%	4	Weekday number (0-6, Sunday-Saturday)
%A	Thursday	Full weekday name ( <i>based on OS settings</i> )
%U	28	Week of the year ( <i>starting on the first Sunday</i> )
%j	195	Day of the year

## Time:

Code	Example	Description
%T	16:36:30	Zero-padded time (24-hour format)
%H	16	Hour (24-hour format)
%I	04	Hour (12-hour format)
%p	PM	AM or PM ( <i>based on OS settings</i> )
%M	36	Minute
%S	30	Second



# FORMATTING DATETIMES

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

Use datetime codes with the `.strftime()` method to customize the **datetime format**

now

```
datetime.datetime(2022, 6, 3, 13, 44, 35, 495577)
```

```
now.strftime("%B %d, %Y")
```

'June 03, 2022'

This is a string now

The **.dt accessor** lets you use `.strftime()` on datetimes in Pandas DataFrames

sales

	date	sales
0	2022-01-01	10
1	2022-01-02	15
2	2022-02-01	<NA>
3	2022-03-01	20
4	2023-01-01	30
5	2023-01-02	35



```
sales.assign(date=sales["date"].dt.strftime("%Y-%B-%a"))
```

	date	sales
0	2022-January-Sat	10
1	2022-January-Sun	15
2	2022-February-Tue	<NA>
3	2022-March-Tue	20
4	2023-January-Sun	30
5	2023-January-Mon	35

You can specify multiple datetime codes and punctuation to create custom formats



Note that `.strftime()` returns a string, so assigning the formatted datetime values to a DataFrame column **reverts the data type to an object**



# EXTRACTING DATETIME COMPONENTS

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

`dates.dt.date`

```
0    2022-01-01
1    2022-01-02
2    2022-02-01
3    2022-03-01
4    2023-01-01
5    2023-01-02
dtype: object
```

`dates.dt.year`

```
0    2022
1    2022
2    2022
3    2022
4    2023
5    2023
dtype: int64
```

`dates.dt.month`

```
0    1
1    1
2    2
3    3
4    1
5    1
dtype: int64
```

`dates.dt.dayofweek`

```
0    5
1    6
2    1
3    1
4    6
5    0
dtype: int64
```

`dates.dt.time`

```
0    00:00:00
1    00:00:00
2    00:00:00
3    00:00:00
4    00:00:00
5    00:00:00
dtype: object
```

`dates.dt.hour`

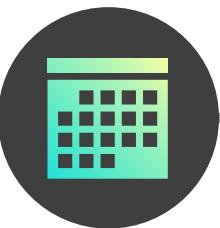
```
0    0
1    0
2    0
3    0
4    0
5    0
dtype: int64
```

`dates.dt.minute`

```
0    0
1    0
2    0
3    0
4    0
5    0
dtype: int64
```

`dates.dt.second`

```
0    0
1    0
2    0
3    0
4    0
5    0
dtype: int64
```



# EXTRACTING DATETIME COMPONENTS

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

While you can also use the `.strftime()` method to **extract datetime components**, it's preferred to use the dedicated `.dt` accessors

```
sales.head()
```

	date	sales	year	quarter	day_of_week
0	2022-01-01	10	2022	1	5
1	2022-02-01	15	2022	1	1
2	2022-01-02	<NA>	2022	1	6
3	2022-01-03	20	2022	1	0
4	2023-01-01	30	2023	1	6

```
sales = sales.assign(  
    year=sales["date"].dt.year,  
    quarter=sales["date"].dt.quarter,  
    day_of_week=sales["date"].dt.dayofweek,  
)  
  
sales.head()
```

	date	sales	year	quarter	day_of_week
0	2022-01-01	10	2022	1	5
1	2022-02-01	15	2022	1	1
2	2022-01-02	<NA>	2022	1	6
3	2022-01-03	20	2022	1	0
4	2023-01-01	30	2023	1	6



# ASSIGNMENT: DATEPARTS & TO\_DATETIME

 NEW MESSAGE  
August 29, 2022

**From:** Chandler Capital (Accounting)  
**Subject:** Time to Last Date

Hi,  
I need some additional transactions data.  
Can you create columns for year, month, and day of week?  
Then create a column, days\_to\_acquisition, which represents the difference between the planned acquisition date of new stores, '2022-10-01', and the date in the row.  
Finally, create a column representing weeks to acquisition.

Thanks

 section07\_time\_series.ipynb     Reply     Forward

## Results Preview

	date	store_nbr	transactions	year	month	day_of_week	days_to_acquisition	weeks_to_acquisition
0	2013-01-01	25	770	2013	1	1	3560 days	508
1	2013-01-02	1	2111	2013	1	2	3559 days	508
2	2013-01-02	2	2358	2013	1	2	3559 days	508
3	2013-01-02	3	3487	2013	1	2	3559 days	508
4	2013-01-02	4	1922	2013	1	2	3559 days	508

# SOLUTION: DATEPARTS & TO\_DATETIME



## NEW MESSAGE

August 29, 2022

From: Chandler Capital (Accounting)  
Subject: Time to Last Date

Hi,

I need some additional transactions data.

Can you create columns for year, month, and day of week?

Then create a column, days\_to\_acquisition, which represents the difference between the planned acquisition date of new stores, '2022-10-01', and the date in the row.

Finally, create a column representing weeks to acquisition.

Thanks

section07\_time\_series.ipynb

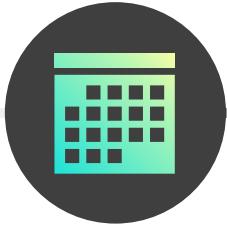
Reply

Forward

## Solution Code

```
transactions.assign(  
    year = transactions["date"].dt.year,  
    month = transactions["date"].dt.month,  
    day_of_week = transactions["date"].dt.dayofweek,  
    days_to_acquisition = pd.to_datetime("2022-10-01", infer_datetime_format=True)  
        - transactions["date"],  
    weeks_to_acquisition = lambda x: (  
        (x["days_to_acquisition"].dt.days / 7).astype("int"))  
    ),  
).head()
```

	date	store_nbr	transactions	year	month	day_of_week	days_to_acquisition	weeks_to_acquisition	
0	2013-01-01	25	770	2013	1	1	3560 days	508	
1	2013-01-02	1	2111	2013	1	2	3559 days	508	
2	2013-01-02	2	2358	2013	1	2	3559 days	508	
3	2013-01-02	3	3487	2013	1	2	3559 days	508	
4	2013-01-02	4	1922	2013	1	2	3559 days	508	



# TIME DELTAS

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

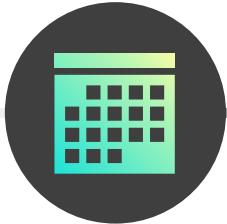
**Time deltas** represent the amount or time, or difference, between two datetimes

- A time delta is returned when subtracting two datetime values

	order_date	delivery_date
0	2021-04-04	2021-04-07
1	2021-05-23	NaT

```
shipping_dates.assign(  
    shipping_days=shipping_dates["delivery_date"] - shipping_dates["order_date"]  
)
```

	order_date	delivery_date	shipping_days
0	2021-04-04	2021-04-07	3 days
1	2021-05-23	NaT	NaT



# TIME DELTAS

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

- A time delta is returned when subtracting two datetime values

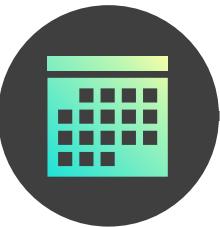
	order_date	delivery_date
0	2021-04-04	2021-04-07
1	2021-05-23	NaT

```
shipping_dates.loc[:, "delivery_date"] - shipping_dates.loc[:, "order_date"]  
0    3 days  
1      NaT  
dtype: timedelta64[ns]
```

This returns a Series with the timedelta64 data type!

```
(shipping_dates.loc[:, "delivery_date"] - shipping_dates.loc[:, "order_date"]).dt.days  
0    3.0  
1    NaN  
dtype: float64
```

You can use .dt accessors on time deltas to convert them to integers



# TO\_TIMEDELTA & FREQUENCIES

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

Frequency	Unit
"D"	Day
"W"	Week
"H"	Hour
"T"	Minute
"S"	Second

```
pd.to_timedelta(5, unit="D")
```

```
Timedelta('5 days 00:00:00')
```

```
pd.to_timedelta(5, unit="W")
```

```
Timedelta('35 days 00:00:00')
```

```
pd.to_timedelta(5, unit="Y")
```

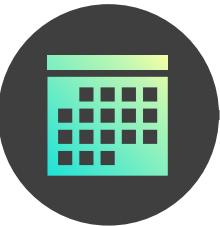
-----  
ValueError

Simply specify a number and the frequency as unit

"M"	Month end
"Q"	Quarter end
"Y"	Year end

Because months, quarters, and years have differing lengths, they are sometimes known as "anchored" frequencies, since they are anchored to the end of calendar periods

They can't be used in time delta arithmetic, but are useful for things like resampling



# TIME DELTA ARITHMETIC

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

You can use **time delta arithmetic** to offset dates by a specified period of time

	order_date	delivery_date
0	2021-04-04	2021-04-07
1	2021-05-23	NaT



```
shipping_dates + pd.to_timedelta(1, unit="D")
```

	order_date	delivery_date
0	2021-04-05	2021-04-08
1	2021-05-24	NaT

```
shipping_dates + pd.to_timedelta(2, unit="W")
```

	order_date	delivery_date
0	2021-04-18	2021-04-21
1	2021-06-06	NaT

# ASSIGNMENT: TIME DELTA ARITHMETIC

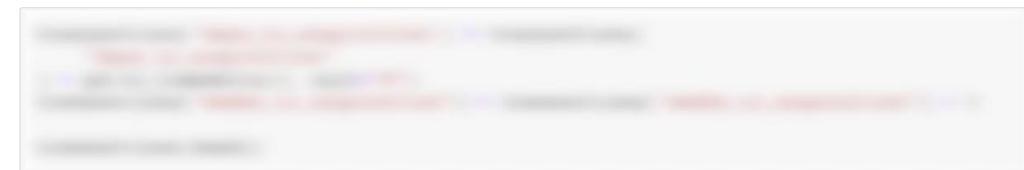
 **NEW MESSAGE**  
August 29, 2022

**From:** Chandler Capital (Accounting)  
**Subject:** Time to Last Date

Hi again,  
I just got word the acquisition date is going to be pushed back by three weeks.  
Can you add three weeks to the days to acquisition column?  
Just add 3 to the weeks to acquisition column.  
Thanks

 section07\_time\_series.ipynb     

## Results Preview



	date	store_nbr	transactions	year	month	day_of_week	days_to_acquisition	weeks_to_acquisition	
0	2013-01-01	25	770	2013	1	1	1967 days	511	
1	2013-01-02	1	2111	2013	1	2	1966 days	511	
2	2013-01-02	2	2358	2013	1	2	1966 days	511	
3	2013-01-02	3	3487	2013	1	2	1966 days	511	
4	2013-01-02	4	1922	2013	1	2	1966 days	511	

# SOLUTION: TIME DELTA ARITHMETIC



NEW MESSAGE

August 29, 2022

From: **Chandler Capital** (Accounting)  
Subject: Time to Last Date

Hi again,

I just got word the acquisition date is going to be pushed back by three weeks.

Can you add three weeks to the days to acquisition column?  
Just add 3 to the weeks to acquisition column.

Thanks



section07\_time\_series.ipynb

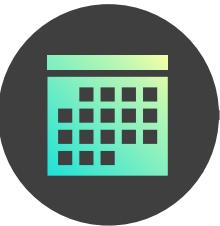
Reply

Forward

## Solution Code

```
transactions["days_to_acquisition"] = transactions[
    "days_to_acquisition"
] + pd.to_timedelta(3, unit="W")
transactions["weeks_to_acquisition"] = transactions["weeks_to_acquisition"] + 3
transactions.head()
```

	date	store_nbr	transactions	year	month	day_of_week	days_to_acquisition	weeks_to_acquisition	
0	2013-01-01	25	770	2013	1	1	1967 days	511	
1	2013-01-02	1	2111	2013	1	2	1966 days	511	
2	2013-01-02	2	2358	2013	1	2	1966 days	511	
3	2013-01-02	3	3487	2013	1	2	1966 days	511	
4	2013-01-02	4	1922	2013	1	2	1966 days	511	



# TIME SERIES INDICES

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

Use **datetimes as the index** to allow for intuitive slicing of your DataFrame

- Like with integers, make sure your time index is sorted or you will get very odd results!

```
sales.head()
```

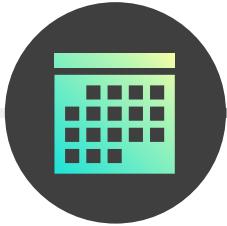
	date	sales
0	2022-01-01	10
1	2022-01-02	15
2	2022-02-01	<NA>
3	2022-03-01	20
4	2023-01-01	30

```
sales.loc[sales["date"].dt.year == 2023]
```

	date	sales
4	2023-01-01	30
5	2023-01-02	35

```
sales.set_index("date").loc["2023"]
```

	date	sales
	2023-01-01	30
	2023-01-02	35



# TIME SERIES INDICES

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

Use **datetimes as the index** to allow for intuitive slicing of your DataFrame

- Like with integers, make sure your time index is sorted or you will get very odd results!

`dates`

```
2022-01-01    10  
2022-01-02    15  
2022-02-01    <NA>  
2022-03-01    20  
2023-01-01    30  
2023-01-02    35  
dtype: Int16
```

`dates.loc["2022"]`

```
2022-01-01    10  
2022-01-02    15  
2022-02-01    <NA>  
2022-03-01    20  
dtype: Int16
```

`dates.loc["2022":"2023"]`

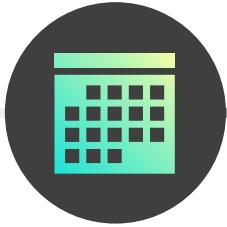
```
2022-01-01    10  
2022-01-02    15  
2022-02-01    <NA>  
2022-03-01    20  
2023-01-01    30  
2023-01-02    35  
dtype: Int16
```

`dates.loc["2022-01":"2022-02"]`

```
2022-01-01    10  
2022-01-02    15  
2022-02-01    <NA>  
dtype: Int16
```

`dates.loc["2022-01-01":"2023-01-01"]`

```
2022-01-01    10  
2022-01-02    15  
2022-02-01    <NA>  
2022-03-01    20  
2023-01-01    30  
dtype: Int16
```



# MISSING TIME SERIES DATA

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

`dates`

2022-01-01	10
2022-01-02	15
2022-02-01	<NA>
2022-03-01	20
2023-01-01	30
2023-01-02	35

`dtype: Int16`

`dates.ffill()`

2022-01-01	10
2022-01-02	15
2022-02-01	15
2022-03-01	20
2023-01-01	30
2023-01-02	35

`dtype: Int16`

`dates.bfill()`

2022-01-01	10
2022-01-02	15
2022-02-01	20
2022-03-01	20
2023-01-01	30
2023-01-02	35

`dtype: Int16`

**Forward-filling** and **back-filling** data replaces missing values with either the value from the previous available date, or the next available date

You need to convert to a float first

```
dates = dates.astype("float64")  
dates.interpolate()
```

2022-01-01	10.0
2022-01-02	15.0
2022-02-01	17.5
2022-03-01	20.0
2023-01-01	30.0
2023-01-02	35.0

`dtype: float64`

$(15 + 20) / 2 = 17.5$

**Interpolating** data replaces missing values with a linear approximation based on the values from the previous and next available dates

# ASSIGNMENT: MISSING TIME SERIES DATA



## NEW MESSAGE

August 30, 2022

**From:** **Rachel Revenue** (Sr. Financial Analyst)  
**Subject:** Missing Oil Prices

Hey rockstar,

Management mostly enjoyed the presentation on oil prices, but they were concerned about the gaps in the chart. I explained these were holidays, but it didn't matter. Can you fill in the missing values?

Take a look at the mean value for the oil price using forward fill, backfill, and interpolation. Are they very different?

Then, plot the series with forward fill for the year 2014, the month of December 2014, and the days from December 1<sup>st</sup> to December 15<sup>th</sup>, 2014.

 section07\_time\_series.ipynb

 Reply     Forward

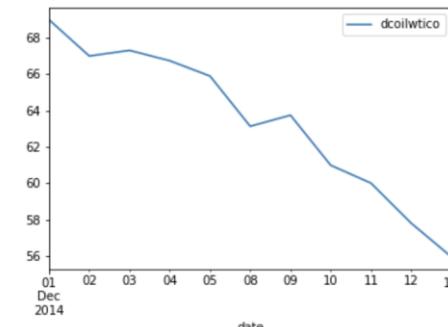
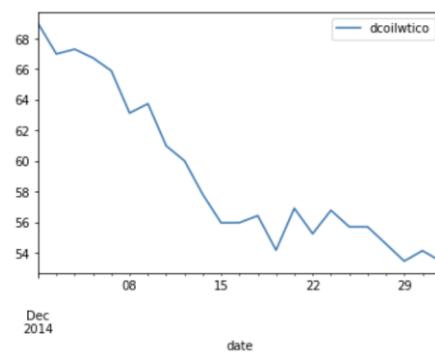
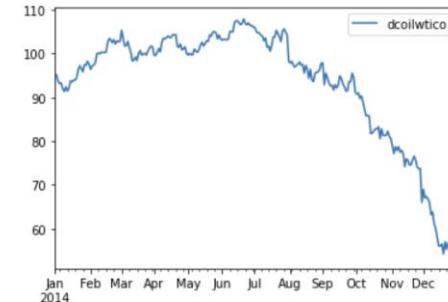
## Results Preview

Mean with Missing Values: **67.71**

Mean with Forward Fill: **67.67**

Mean with Backfill: **67.67**

Mean with Interpolation: **67.66**



# SOLUTION: MISSING TIME SERIES DATA



## NEW MESSAGE

August 30, 2022

From: **Rachel Revenue** (Sr. Financial Analyst)  
Subject: Missing Oil Prices

Hey rockstar,

Management mostly enjoyed the presentation on oil prices, but they were concerned about the gaps in the chart. I explained these were holidays, but it didn't matter. Can you fill in the missing values?

Take a look at the mean value for the oil price using forward fill, backfill, and interpolation. Are they very different?

Then, plot the series with forward fill for the year 2014, the month of December 2014, and the days from December 1<sup>st</sup> to December 15<sup>th</sup>, 2014.

section07\_time\_series.ipynb

Reply

Forward

## Solution Code

```
oil.mean()
```

```
dcoilwtico    67.714366  
dtype: float64
```

```
oil.bfill().mean()
```

```
dcoilwtico    67.673325  
dtype: float64
```

```
oil.ffill().mean()
```

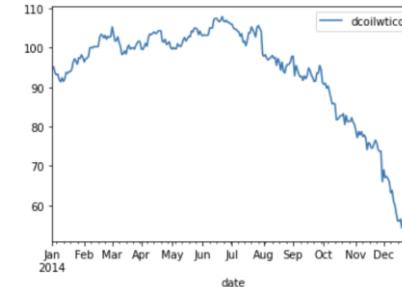
```
dcoilwtico    67.671249  
dtype: float64
```

```
oil.interpolate().mean()
```

```
dcoilwtico    67.661824  
dtype: float64
```

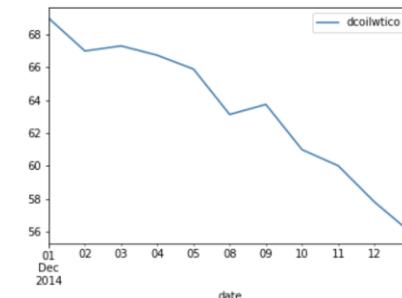
```
oil.loc["2014"].ffill().plot()
```

```
<AxesSubplot:xlabel='date'>
```



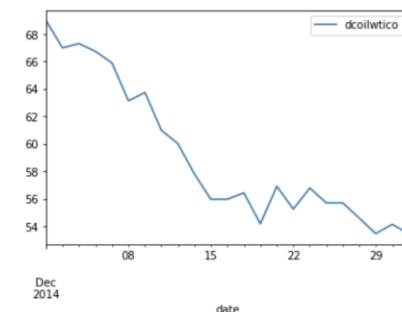
```
oil.loc["2014-12-01":"2014-12-15"].ffill().plot()
```

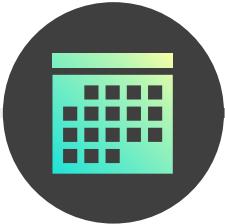
```
<AxesSubplot:xlabel='date'>
```



```
oil.loc["2014-12"].ffill().plot()
```

```
<AxesSubplot:xlabel='date'>
```





# SHIFTING SERIES

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

You can **shift a Series** by a specified number of rows using the `.shift()` method

- This is helpful when working with time series to compare values against previous periods

dates	
2022-01-01	10
2022-01-02	15
2022-02-01	15
2022-03-01	20
2023-01-01	30
2023-01-02	35

`dtype: Int16`

dates.shift()	
2022-01-01	<NA>
2022-01-02	10
2022-02-01	15
2022-03-01	15
2023-01-01	20
2023-01-02	30

`dtype: Int16`

dates.shift(-1)	
2022-01-01	15
2022-01-02	15
2022-02-01	20
2022-03-01	30
2023-01-01	35
2023-01-02	<NA>

`dtype: Int16`

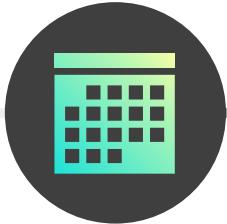
Series shift forward one row by default, but you can specify any positive or negative number to shift the Series by

(dates / dates.shift(1)).sub(1).mul(100).round(2)	
2022-01-01	<NA>
2022-01-02	50.0
2022-02-01	0.0
2022-03-01	33.33
2023-01-01	50.0
2023-01-02	16.67

`dtype: Float64`

This is calculating period-over-period growth by leveraging a date shift:

$$\% \text{ Growth} = \frac{\text{Current Period}}{\text{Previous Period}} - 1$$



# PRO TIP: DIFF

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

The `.diff()` method calculates the difference between the values in a Series and those same values shifted a specified number of periods

- This is useful in measuring absolute changes over time

dates		dates.diff()	=	dates - dates.shift(1)	
2022-01-01	10	2022-01-01	<NA>	2022-01-01	<NA>
2022-01-02	15	2022-01-02	5	2022-01-02	5
2022-02-01	15	2022-02-01	0	2022-02-01	0
2022-03-01	20	2022-03-01	5	2022-03-01	5
2023-01-01	30	2023-01-01	10	2023-01-01	10
2023-01-02	35	2023-01-02	5	2023-01-02	5
dtype: Int16		dtype: Int16		dtype: Int16	

Both methods calculate the difference between the current row and the previous one

dates.diff(2)

2022-01-01	<NA>
2022-01-02	<NA>
2022-02-01	5
2022-03-01	5
2023-01-01	15
2023-01-02	15
dtype: Int16	

`.diff()` calculates the difference from the previous row by default, but you can specify the number of rows as well

# SOLUTION: SHIFT & DIFF



**1 NEW MESSAGE**

August 31, 2022

**From:** Chandler Capital (Accounting)  
**Subject:** Monthly and Annual Charts

Hello,

I'm looking into a few different year over year trends related to year over year changes for store number 47.

Can you plot the monthly average transactions in year 2015 vs the monthly average in the year prior?

Then, can you plot year over year growth percent in average monthly transactions?

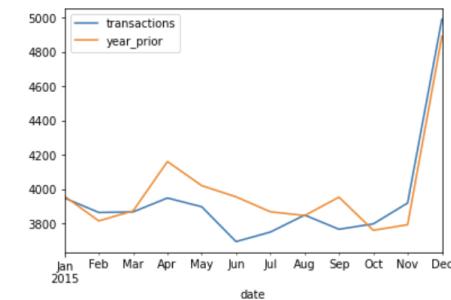
Thanks

 section07\_time\_series.ipynb

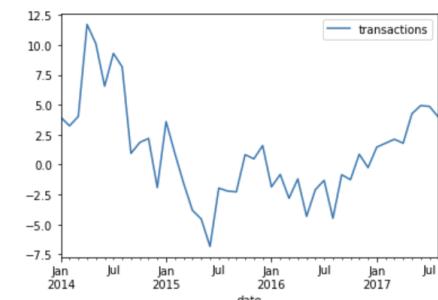
 

## Solution Code

```
transactions_47 = transactions.loc[  
    transactions["store_nbr"] == 47, ["date", "transactions"]]  
transactions_47.set_index("date")  
  
transactions_47["year_prior"] = transactions_47.shift(365)  
  
transactions_47.loc["2015"].resample("M").mean().plot()
```



```
transactions_47.drop(["year_prior"], axis=1, inplace=True)  
  
transactions_47_monthly = transactions_47.resample("M").mean()  
  
(transactions_47_monthly.diff(12) / transactions_47_monthly).mul(100).loc[  
    "2014":].plot()
```



# ASSIGNMENT: SHIFT & DIFF

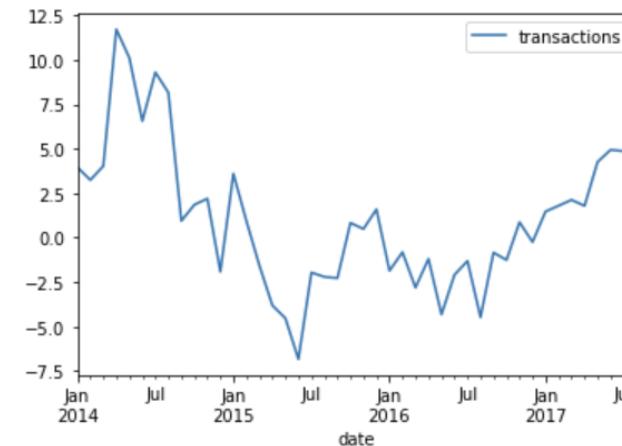
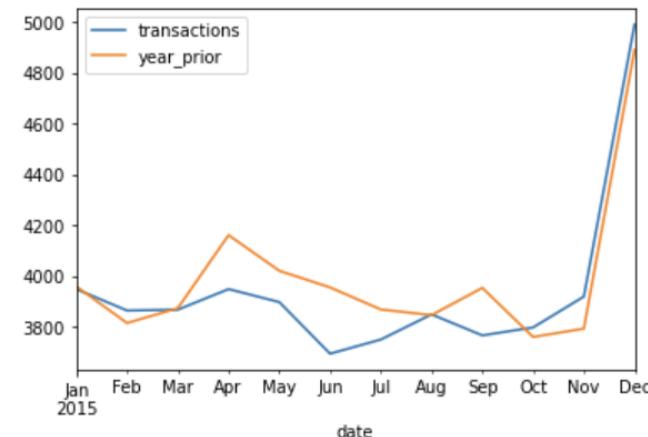
 NEW MESSAGE  
August 31, 2022

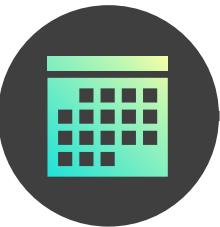
**From:** Chandler Capital (Accounting)  
**Subject:** Monthly and Annual Charts

Hello,  
I'm looking into a few different year over year trends related to year over year changes for store number 47.  
Can you plot the monthly average transactions in year 2015 vs the monthly average in the year prior?  
Then, can you plot year over year growth percent in average monthly transactions?  
Thanks

 section07\_time\_series.ipynb     Reply     Forward

## Results Preview





# AGGREGATING TIME SERIES

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

You can **aggregate time series** using the `.groupby()` method

`sales`



```
sales.groupby(sales[ "date" ].dt.month).agg({ "sales" : "sum" })
```

	date	sales
0	2022-01-01	10
1	2022-01-02	15
2	2022-02-01	<NA>
3	2022-03-01	20
4	2023-01-01	30
5	2023-01-02	35

`sales`

`date`

1

2

3

`sales`

Calculates the total sales by month, regardless of the year

```
sales.groupby(sales[ "date" ].dt.year).agg({ "sales" : "sum" })
```

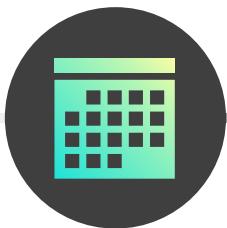
`sales`

`date`

2022

2023

Calculates the total sales by year



# RESAMPLING TIME SERIES

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

**Resampling** is a special form of time series aggregation that modifies date indices and fills any gaps to create continuous date values

- Frequencies are used to define the granularity (you can use years, quarters, and months now!)

dates	dates.resample("M").sum()
2022-01-01	2022-01-31 25
2022-01-02	2022-02-28 0
2022-02-01	2022-03-31 20
2022-03-01	2022-04-30 0
2023-01-01	2022-05-31 0
2023-01-02	2022-06-30 0
dtype: Int16	2022-07-31 0
	2022-08-31 0
	2022-09-30 0
	2022-10-31 0
	2022-11-30 0
	2022-12-31 0
	2023-01-31 65
	Freq: M, dtype: Int64

This modified the index and created consecutive month values from Jan 2022 to Jan 2023, even though only 4 months have data, and summed the values for each month using .sum()

dates.resample("M").transform("sum")	
2022-01-01	25.0
2022-01-02	25.0
2022-02-01	0.0
2022-03-01	20.0
2023-01-01	65.0
2023-01-02	65.0
	dtype: float64

Use .transform() to keep the original index and apply the group-level aggregations!

# ASSIGNMENT: RESAMPLING TIME SERIES

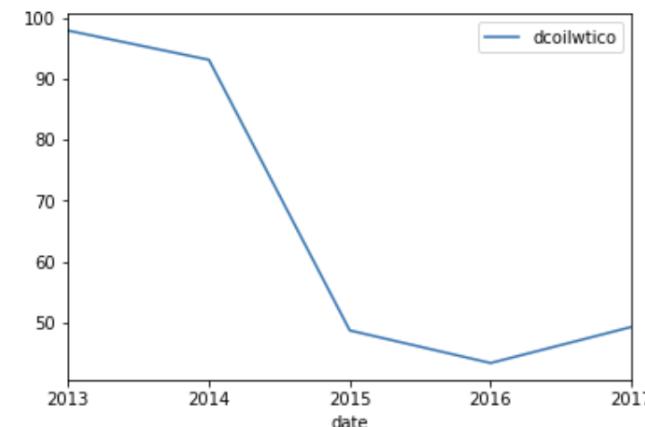
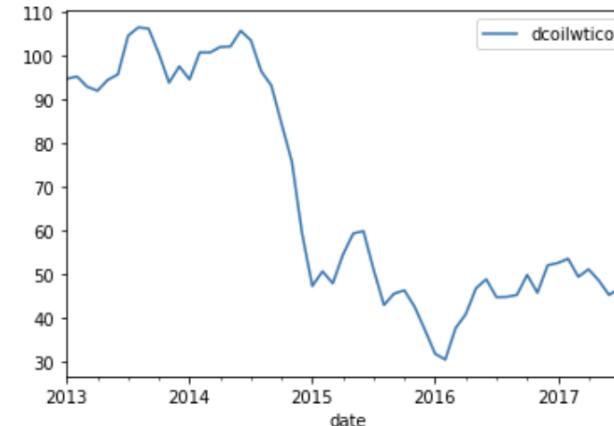
 **NEW MESSAGE**  
August 31, 2022

**From:** **Rachel Revenue** (Sr. Financial Analyst)  
**Subject:** Monthly and Annual Charts

Hey again,  
Really quick, can you also get me charts of the average oil price by month and year!  
It will help complete my new presentation to leadership.  
Thank you!!!!

 section07\_time\_series.ipynb     

## Results Preview

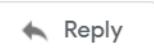
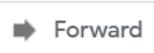


# SOLUTION: RESAMPLING TIME SERIES

 NEW MESSAGE  
August 31, 2022

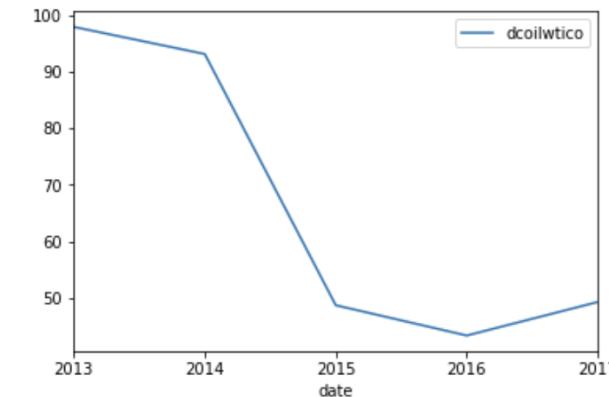
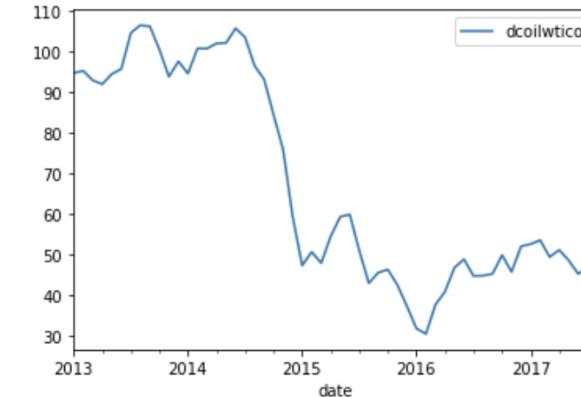
**From:** Rachel Revenue (Sr. Financial Analyst)  
**Subject:** Monthly and Annual Charts

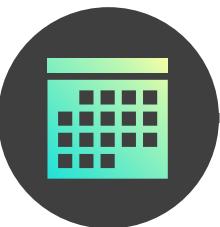
Hey again,  
Really quick, can you also get me charts of the average oil price by month and year!  
It will help complete my new presentation to leadership.  
Thank you!!!!

## Solution Code

```
for period in ["M", "Y"]:  
    oil.resample(period).mean().plot()
```





# ROLLING AGGREGATIONS

The Datetime Data Type

Formatting & Parting

Time Deltas

Datetime Indices

Shifting & Aggregating

**Rolling aggregations** let you perform calculations on shifting periods of time

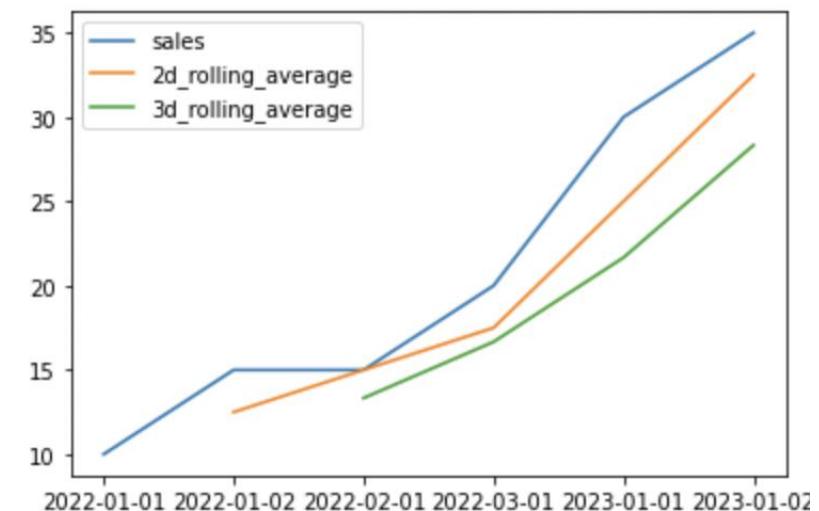
- This can be used to calculate things like moving averages, which are helpful in reducing noise when plotting trends

```
dates  
2022-01-01    10  
2022-01-02    15  
2022-02-01    15  
2022-03-01    20  
2023-01-01    30  
2023-01-02    35  
dtype: Int16
```



```
dates.rolling(2).mean()  
2022-01-01      NaN  
2022-01-02     12.5  
2022-02-01     15.0  
2022-03-01     17.5  
2023-01-01     25.0  
2023-01-02     32.5  
dtype: float64
```

```
dates.rolling(3).mean()  
2022-01-01      NaN  
2022-01-02      NaN  
2022-02-01    13.333333  
2022-03-01    16.666667  
2023-01-01    21.666667  
2023-01-02    28.333333  
dtype: float64
```



# ASSIGNMENT: ROLLING AGGREGATIONS

 NEW MESSAGE  
August 31, 2022

From: **Chandler Capital** (Accounting)  
Subject: Monthly and Annual Charts

Hey,  
Thanks for your help on those year over year figures.  
Can you plot the 90-day moving average for transactions for store 47?  
I want to present them with less noise than the daily figures.  
Thanks!

## Results Preview



# SOLUTION: ROLLING AGGREGATIONS



NEW MESSAGE

August 31, 2022

From: **Chandler Capital** (Accounting)  
Subject: Monthly and Annual Charts

Hey,  
Thanks for your help on those year over year figures.  
Can you plot the 90-day moving average for transactions for store 47?  
I want to present them with less noise than the daily figures.  
Thanks!

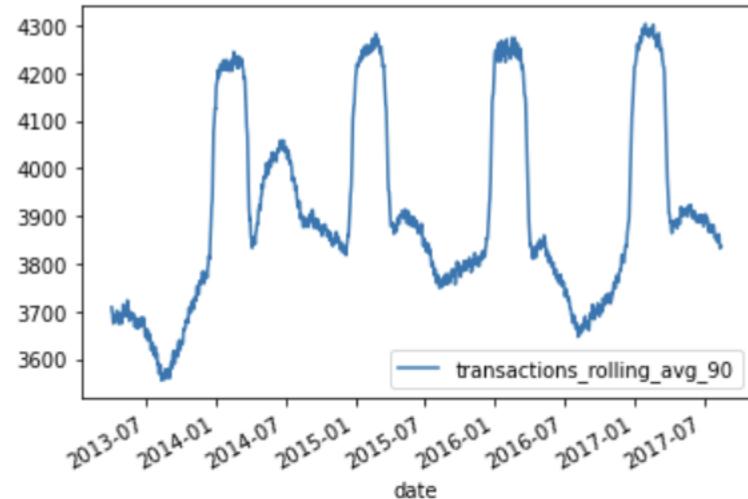
📎 section07\_time\_series.ipynb

Reply

Forward

## Solution Code

```
transactions_47.assign(  
    transactions_rolling_avg_90=transactions_47.rolling(90).mean(),  
).drop(["transactions"], axis=1).plot()
```



# KEY TAKEAWAYS

---



The **datetime64** data type lets you work with time series in Pandas

- *Conversion can be deceptively complicated, so use the `.to_datetime()` method to manage errors, or explicitly state the datetime format for Pandas to interpret it correctly*



Use datetime **codes** & **accessors** to format dates and extract date components

- *There are dozens of options, but you only need to memorize the common parts and formats for most business analysis scenarios – you can always reference the documentation if needed!*



Leverage **datetime indices** to slice time series data intuitively & efficiently

- *This also provides alternative methods for fixing missing data, like forward-filling, back-filling, and interpolation*



**Shifting** & **aggregation** methods let you perform time intelligence calculations

- *Shifting and differencing let you calculate period-over-period changes, grouping and resampling let you perform unique time series summaries, while rolling aggregations let you calculate things like moving averages*

# IMPORTING & EXPORTING DATA

# IMPORTING & EXPORTING DATA



In this section we'll cover **importing & exporting data** in Pandas, including reading in data from flat files and SQL tables, applying processing steps on import, and writing back out

## TOPICS WE'LL COVER:

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

## GOALS FOR THIS SECTION:

- Apply data processing steps like converting data types, setting an index, and handling missing data during the import process
- Read & write data from different flat files and multiple Excel worksheets
- Connect to SQL Databases and create DataFrames from custom SQL queries



# READ\_CSV REVISITED

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

The **read\_csv()** function only needs a file path to read in a file, but it also has many capabilities for preprocessing the data using other arguments:

- **file = “path/name.csv”** – file path & name to read in (*can also be a URL*)
- **sep = “/”** – character used to separate each column of values (*default is comma*)
- **header = 0** – row number to use as the column names (*default is “infer”*)
- **names = [“date”, “sales”]** – list of column names to override the existing ones
- **index\_col = “date”** – column name to be used as the DataFrame index
- **usecols = [“date”, “sales”]** – list of columns to keep in the DataFrame
- **dtype = {“date”: “datetime64”, “sales” : “Int32”}** – dictionary with column names and data types
- **parse\_dates = True** – converts date strings into datetimes when True
- **infer\_datetime\_format = True** – makes parsing dates more efficient
- **na\_values = [“, “#N/A!”]** – strings to recognize as NaN values
- **nrows = 100** – number of rows to read in
- **skiprows = [0, 2]** – line numbers to skip (*accepts lambda functions*)
- **converters = {“sales”: lambda x: f"\${x}"}** – dictionary with column names and functions to apply



# COLUMN NAMES

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

Pandas will try to infer the **column names** in your file by default, but there are several options to override this behavior:

- Specify **header=None** to keep all the rows in the file as data, and use integers as column names
- Specify **header=0** and use the **names** argument to pass a list of desired column names

```
pd.read_csv("monthly_sales.csv").head(3)
```

	Date	Sales	Sales.1	Useless
0	10/10/21	5	1	NaN
1	10/11/21	.	2	NaN
2	10/12/21	15	3	NaN

```
pd.read_csv("monthly_sales.csv", header=None).head(3)
```

	0	1	2	3
0	Date	Sales	Sales	Useless
1	10/10/21	5	1	NaN
2	10/11/21	.	2	NaN

The first row is kept in the DataFrame

```
cols = ["Date", "Grocery Sales", "Beverage Sales", "to drop"]  
pd.read_csv("monthly_sales.csv", header=0, names=cols).head(3)
```

	Date	Grocery Sales	Beverage Sales	to drop
0	10/10/21	5	1	NaN
1	10/11/21	.	2	NaN
2	10/12/21	15	3	NaN

The new column names are used

The first row is used for the column headers by default, and a suffix of ".1" was added to avoid duplicates



# SETTING AN INDEX

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

You can **set the index** for the DataFrame with the “index\_col” argument

- Pass a list of column names to create a multi-index DataFrame (*not recommended!*)
- Specify **parse\_dates=True** to convert index date column to a datetime data type

```
pd.read_csv("monthly_sales.csv", index_col="Date")
```

	Sales	Sales.1	Useless
Date			
10/10/21	5	1	NaN
10/11/21	.	2	NaN
10/12/21	15	3	NaN
10/13/21	20	4	NaN
10/14/21	MISSING	5	NaN
10/15/21	30	6	NaN
10/16/21	35	7	NaN
10/17/21	40	8	NaN

```
pd.read_csv("monthly_sales.csv",
            index_col="Date",
            parse_dates=True).index.dtype
```

```
dtype('datetime64[ns]')
```

This is a synonym for `datetime64`



# SELECTING COLUMNS

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

You can **select the columns** to read in with the “use\_cols” argument

- This can save a lot of processing time and memory

```
pd.read_csv("monthly_sales.csv").head(3)
```

	Date	Sales	Sales.1	Useless
0	10/10/21	5	1	NaN
1	10/11/21	.	2	NaN
2	10/12/21	15	3	NaN

```
pd.read_csv("monthly_sales.csv", usecols=[ "Date", "Sales"]).head(3)
```

	Date	Sales
0	10/10/21	5
1	10/11/21	.
2	10/12/21	15



# SELECTING ROWS

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

You can **select the rows** to read in from the top with the “nrows” argument, and specify any rows to skip with “skiprows”

```
pd.read_csv("monthly_sales.csv", nrows=5)
```

	Date	Sales	Sales.1	Useless
0	10/10/21	5	1	NaN
1	10/11/21	.	2	NaN
2	10/12/21	15	3	NaN
3	10/13/21	20	4	NaN
4	10/14/21	MISSING	5	NaN

Reading in the first few rows  
is great for peeking at big  
files before reading them in  
completely

```
pd.read_csv("monthly_sales.csv", skiprows=[1, 3, 5, 7])
```

	Date	Sales	Sales.1	Useless
0	10/11/21	.	2	NaN
1	10/13/21	20	4	NaN
2	10/15/21	30	6	NaN
3	10/17/21	40	8	NaN

This skips these four  
odd-numbered rows

```
pd.read_csv("monthly_sales.csv", skiprows=lambda x: x % 2 == 1)
```

	Date	Sales	Sales.1	Useless
0	10/11/21	.	2	NaN
1	10/13/21	20	4	NaN
2	10/15/21	30	6	NaN
3	10/17/21	40	8	NaN

This lambda function skips  
all odd-numbered rows



# MISSING VALUES

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

You can specify strings (or other values) to treat as **missing values** with the “na\_values” argument

- They are replaced with NumPy NaN values

```
pd.read_csv("monthly_sales.csv")
```

	Date	Sales	Sales.1	Useless
0	10/10/21	5	1	NaN
1	10/11/21	.	2	NaN
2	10/12/21	15	3	NaN
3	10/13/21	20	4	NaN
4	10/14/21	MISSING	5	NaN
5	10/15/21	30	6	NaN
6	10/16/21	35	7	NaN
7	10/17/21	40	8	NaN

```
pd.read_csv("monthly_sales.csv", na_values=[ ".", "MISSING" ])
```

	Date	Sales	Sales.1	Useless
0	10/10/21	5.0	1	NaN
1	10/11/21	NaN	2	NaN
2	10/12/21	15.0	3	NaN
3	10/13/21	20.0	4	NaN
4	10/14/21	NaN	5	NaN
5	10/15/21	30.0	6	NaN
6	10/16/21	35.0	7	NaN
7	10/17/21	40.0	8	NaN





# PARSING DATES

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

Dates are read in as object data types by default, but you can **parse dates** with the “parse\_dates” argument to convert them to datetime64

- Specifying **infer\_datetime\_format=True** will speed up the date parsing

```
pd.read_csv("monthly_sales.csv").dtypes
```

```
Date      object
Sales     object
Sales.1    int64
Useless   float64
dtype: object
```



```
pd.read_csv(
    "monthly_sales.csv",
    parse_dates=["Date"],
    infer_datetime_format=True,
    na_values=["MISSING", ". "],
).dtypes
```

```
Date      datetime64[ns]
Sales    float64
Sales.1    int64
Useless   float64
dtype: object
```



# DATA TYPES

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

You can set the **data type** for each column with the “dtype” argument by passing in a dictionary with column names as keys and desired data types as values

- Get your data into its most efficient format from the start!

```
pd.read_csv("monthly_sales.csv").dtypes
```

```
Date      object  
Sales     object  
Sales.1    int64  
Useless   float64  
dtype: object
```



```
pd.read_csv(  
    "monthly_sales.csv",  
    parse_dates=["Date"],  
    infer_datetime_format=True,  
    na_values=["MISSING", "."],  
    dtype={"Sales": "Int32", "Sales.1": "Int8", "Useless": "string"}),  
).dtypes
```

```
Date      datetime64[ns]  
Sales     Int32  
Sales.1    Int8  
Useless   string  
dtype: object
```



# PRO TIP: CONVERTERS

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

You can apply functions to columns of data by using the “**converters**” argument

- Pass a dictionary with the column names as keys and the functions as values

```
pd.read_csv("monthly_sales.csv")
```

	Date	Sales	Sales.1	Useless
0	10/10/21	5	1	NaN
1	10/11/21	.	2	NaN
2	10/12/21	15	3	NaN
3	10/13/21	20	4	NaN
4	10/14/21	MISSING	5	NaN
5	10/15/21	30	6	NaN
6	10/16/21	35	7	NaN
7	10/17/21	40	8	NaN



```
missing = lambda x: x if x not in ["MISSING", "."] else 0  
currency = lambda x: f"${x}.00"
```

```
pd.read_csv(  
    "monthly_sales.csv",  
    converters={"Sales": missing,  
               "Sales.1": currency  
    }  
)
```

	Date	Sales	Sales.1	Useless
0	10/10/21	5	\$1.00	NaN
1	10/11/21	0	\$2.00	NaN
2	10/12/21	15	\$3.00	NaN
3	10/13/21	20	\$4.00	NaN
4	10/14/21	0	\$5.00	NaN
5	10/15/21	30	\$6.00	NaN
6	10/16/21	35	\$7.00	NaN
7	10/17/21	40	\$8.00	NaN

Assign the functions to variables  
to make the code easier to read

# ASSIGNMENT: IMPORTING DATA

 **NEW MESSAGE**  
September 2, 2022

**From:** Monica Market (Marketing Analytics)  
**Subject:** Streamlined Transaction Workflow

Hey there!

I saw some of the work you did with Chandler on the transactions dataset.

I need to use the same data for some marketing analysis.

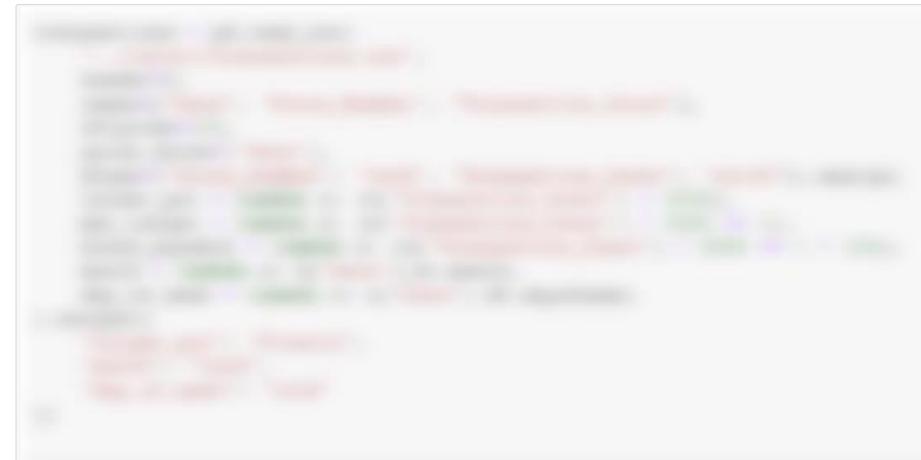
Can you streamline the code and send it over for analysis?

Thanks!

 section08\_import\_export.ipynb

## Results Preview



```
transactions.info(memory_usage="deep")  
  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 83487 entries, 0 to 83486  
Data columns (total 8 columns):  
 #   Column           Non-Null Count  Dtype     
 ---  --  
 0   Date             83487 non-null   datetime64[ns]  
 1   Store_Number     83487 non-null   Int8  
 2   Transaction_Count 83487 non-null   Int16  
 3   target_pct       83487 non-null   Float32  
 4   met_target       83487 non-null   boolean  
 5   bonus_payable    83487 non-null   boolean  
 6   month            83487 non-null   Int8  
 7   day_of_week      83487 non-null   Int8  
dtypes: Float32(1), Int16(1), Int8(3), boolean(2), datetime64[ns](1)  
memory usage: 2.1 MB
```

# SOLUTION: IMPORTING DATA

 NEW MESSAGE

September 2, 2022

From: Monica Market (Marketing Analytics)  
Subject: Streamlined Transaction Workflow

Hey there!

I saw some of the work you did with Chandler on the transactions dataset.

I need to use the same data for some marketing analysis.

Can you streamline the code and send it over for analysis?

Thanks!

 section08\_import\_export.ipynb

## Solution Code

```
transactions = pd.read_csv(  
    "../retail/transactions.csv",  
    header=0,  
    names=["Date", "Store_Number", "Transaction_Count"],  
    skiprows=[0],  
    parse_dates=["Date"],  
    dtype={"Store_Number": "Int8", "Transaction_Count": "Int16"}).assign(  
    target_pct = lambda x: (x["Transaction_Count"] / 2500),  
    met_target = lambda x: (x["Transaction_Count"] / 2500 >= 1),  
    bonus_payable = lambda x: (x["Transaction_Count"] / 2500 >= 1 * 100),  
    month = lambda x: x["Date"].dt.month,  
    day_of_week = lambda x: x["Date"].dt.dayofweek,  
).astype({  
    "target_pct": "Float32",  
    "month": "Int8",  
    "day_of_week": "Int8"  
})
```

```
transactions.info(memory_usage="deep")  
  
<class 'pandas.core.frame.DataFrame'>  
RangeIndex: 83487 entries, 0 to 83486  
Data columns (total 8 columns):  
 #   Column           Non-Null Count  Dtype     
 ---  --  
 0   Date             83487 non-null   datetime64[ns]  
 1   Store_Number     83487 non-null   Int8  
 2   Transaction_Count 83487 non-null   Int16  
 3   target_pct       83487 non-null   Float32  
 4   met_target       83487 non-null   boolean  
 5   bonus_payable    83487 non-null   boolean  
 6   month            83487 non-null   Int8  
 7   day_of_week      83487 non-null   Int8  
 dtypes: Float32(1), Int16(1), Int8(3), boolean(2), datetime64[ns](1)  
 memory usage: 2.1 MB
```



# READING TXT FILES

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

The `read_csv()` function can also **read in .txt files**, and other types of flat files

- Simply use the “sep” argument to specify the delimiter
- You can also read in .tsv (*tab separated values*) files and URLs pointing to text files

```
pd.read_csv('tab_separated.txt')
```

Date\tSales

0	2021-10-10\t5
1	2021-10-11\t10
2	2021-10-12\t15
3	2021-10-13\t20
4	2021-10-14\t25
5	2021-10-15\t30
6	2021-10-16\t35
7	2021-10-17\t40

Pandas is looking for  
comma separators

```
pd.read_csv('tab_separated.txt', sep='\t')
```

Date Sales

0	2021-10-10	5
1	2021-10-11	10
2	2021-10-12	15
3	2021-10-13	20
4	2021-10-14	25
5	2021-10-15	30
6	2021-10-16	35
7	2021-10-17	40

This represents a tab



# READING EXCEL FILES

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

The `read_excel()` function is used to **read in Excel files** in Pandas

- You can specify worksheets by passing the sheet name or position to the “sheet\_name” argument

```
pd.read_excel('monthly_sales.xlsx')
```

	Date	Sales
0	2021-10-10	5
1	2021-10-11	10
2	2021-10-12	15
3	2021-10-13	20
4	2021-10-14	25
5	2021-10-15	30
6	2021-10-16	35
7	2021-10-17	40

```
pd.read_excel('monthly_sales.xlsx', sheet_name=1)
```

	Date	Sales
0	2021-11-10	105
1	2021-11-11	110
2	2021-11-12	115
3	2021-11-13	120
4	2021-11-14	125
5	2021-11-15	130
6	2021-11-16	135
7	2021-11-17	140

0-indexed!





# PRO TIP: APPENDING SHEETS

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

You can use Pandas' concat() function to **append data from multiple sheets**

- We'll cover combining DataFrames in the next section, but this is a sneak peek!

```
all_sales = pd.concat(  
    pd.read_excel("monthly_sales.xlsx", sheet_name=None),  
    ignore_index=True  
)
```

all\_sales.head()

all\_sales.tail()

	Date	Sales
0	2021-10-10	5
1	2021-10-11	10
2	2021-10-12	15
3	2021-10-13	20
4	2021-10-14	25

	Date	Sales
11	2021-11-13	120
12	2021-11-14	125
13	2021-11-15	130
14	2021-11-16	135
15	2021-11-17	140

`sheet_name=None` makes Pandas read every sheet and store them as a dictionary, which is appended using .concat()  
`ignore_index=True` makes sure each row has a unique index

The DataFrame has the data from both sheets!



# EXPORTING TO FLAT FILES

Preprocessing  
Options

Flat Files

SQL Databases

Additional  
Formats

The `to_csv()` and `to_excel()` functions let you **export DataFrames** to flat files

```
my_df.to_csv("cleaned_data.csv")
```

cleaned\_data.csv  
 cleaned\_data.txt

```
my_df.to_csv("cleaned_data.txt", sep="\t")
```

```
my_df.to_excel("cleaned_data.xlsx", sheet_name="October_Sales")
```

cleaned\_data.xlsx

## PRO TIP:

```
with pd.ExcelWriter("cleaned_data.xlsx") as writer:  
    my_df.to_excel(writer, sheet_name="October_Sales")  
    my_df.to_excel(writer, sheet_name="November_Sales")
```

Multiple tabs!

	A	B	C	D
1		Date	Sales	Sales.1
2	0	10/10/21	5	1
3	1	10/11/21	.	2
4	2	10/12/21	15	3
5	3	10/13/21	20	4
6	4	10/14/21	MISSING	5
7	5	10/15/21	30	6
8	6	10/16/21	35	7
9	7	10/17/21	40	8
10		October_Sales	November_Sales	+

# ASSIGNMENT: EXPORTING DATA

  NEW MESSAGE  
September 2, 2022

**From:** Chandler Capital (Accounting)  
**Subject:** Export Transactions To Excel

Hey again!

As you know, my department works primarily with Excel.

I'd like a copy of the modified transactions data stored in an Excel workbook, with one sheet for each year in the data.

If you don't have Excel, then a csv file for each year is ok too.

Thanks!

**Results Preview**



The screenshot shows the Microsoft Excel ribbon with the tab "Data" selected. Below the ribbon, a preview of a CSV file is displayed. The CSV file contains five rows of transaction data with columns: Date, core\_Numbers, transaction\_Co, target\_pct, met\_target, bonus\_payable, month, and day\_of\_week. The first row is a header. The data starts from row 2 to 5. Row 2: Date 2013-01-02 00:00:00, core\_Numbers 1, transaction\_Co 2111, target\_pct 0.8444, met\_target FALSE, bonus\_payable FALSE, month 1, day\_of\_week 2. Row 3: Date 2013-01-02 00:00:00, core\_Numbers 2, transaction\_Co 2358, target\_pct 0.9432, met\_target FALSE, bonus\_payable FALSE, month 1, day\_of\_week 2. Row 4: Date 2013-01-02 00:00:00, core\_Numbers 3, transaction\_Co 3487, target\_pct 1.3948, met\_target TRUE, bonus\_payable FALSE, month 1, day\_of\_week 2. Row 5: Date 2013-01-02 00:00:00, core\_Numbers 4, transaction\_Co 1922, target\_pct 0.7688, met\_target FALSE, bonus\_payable FALSE, month 1, day\_of\_week 2. At the bottom of the preview, there are tabs for 2013, 2014, 2015, 2016, 2017, and a plus sign for more years.

# SOLUTION: EXPORTING DATA



## NEW MESSAGE

September 2, 2022

From: **Chandler Capital** (Accounting)  
Subject: Export Transactions To Excel

Hey again!

As you know, my department works primarily with Excel.

I'd like a copy of the modified transactions data stored in an Excel workbook, with one sheet for each year in the data.

If you don't have Excel, then a csv file for each year is ok too.

Thanks!

Reply

Forward

## Solution Code

```
with pd.ExcelWriter("DataForChandler.xlsx") as writer:  
    for year in range(2013, 2018):  
        transactions.loc[transactions["Date"].dt.year == year].to_excel(  
            writer, sheet_name=str(year))
```

	A	B	C	D	E	F	G	H	I	J	K	L
1		Date	core_Numbers	transaction_Co	target_pct	met_target	onus_payab	month	day_of_week			
2	0	2013-01-02 00:00:00	1	2111	0.8444	FALSE	FALSE	1	2			
3	1	2013-01-02 00:00:00	2	2358	0.9432	FALSE	FALSE	1	2			
4	2	2013-01-02 00:00:00	3	3487	1.3948	TRUE	FALSE	1	2			
5	3	2013-01-02 00:00:00	4	1922	0.7688	FALSE	FALSE	1	2			



# CONNECTING TO SQL DATABASES

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

The SQLAlchemy library lets you **connect to SQL databases**

- All major SQL implementations can be accessed (MySQL, Oracle, MS-SQL, etc.)

```
from sqlalchemy import create_engine, inspect
engine = create_engine("sqlite:///.../soccer_database.sqlite")
inspect(engine).get_table_names()
['Country',
 'League',
 'Match',
 'Player',
 'Player_Attributes',
 'Team',
 'Team_Attributes',
 'sqlite_sequence']
```

The `create_engine()` function creates the database connection, in this case to a local database, and `inspect().get_table_names()` is used to view the database contents

```
# Generic Database Connection String
dialect+driver://username:password@host:port/database

# Oracle Example
engine = create_engine("oracle://chris:password@127.0.0.1:1521/sidname")
```

This example shows a connection to an Oracle database by providing a username and password, as well as the server location



# QUERYING SQL DATABASES

Preprocessing  
Options

Flat Files

SQL Databases

Additional  
Formats

```
league_df = pd.read_sql("SELECT * FROM League", engine)  
league_df.head()
```

	<b>id</b>	<b>country_id</b>	<b>name</b>
0	1	1	Belgium Jupiler League
1	1729	1729	England Premier League
2	4769	4769	France Ligue 1
3	7809	7809	Germany 1. Bundesliga
4	10257	10257	Italy Serie A

```
query = """  
SELECT match.id,  
       League.name AS league_name,  
       season,  
       home_team_goal AS HomeGoals,  
       away_team_goal AS AwayGoals  
FROM Match  
      JOIN League on League.id = Match.league_id  
      WHERE league_id=1729 AND season='2015/2016'  
"""  
premier_league_games = pd.read_sql(query, engine)
```

This query is being performed on the SQL database connection set up previously, and assigned to a DataFrame named league\_df



**PRO TIP:** In most cases, it's more efficient to perform tasks like joins and filters in your database – databases are designed for these tasks and tend to have more resources than your local machine



# WRITING TO SQL DATABASES

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

The `to_sql()` function lets you **create a SQL table** from your DataFrame

- This lets you clean data using Pandas before storing it in a SQL Database
- You will likely need permission from your database administrator (DBA) to do this

```
from sqlalchemy.types import Integer  
  
premier_league_games.to_sql(  
    name="pl_games",  
    con=engine,  
    if_exists="append",  
    index=False,  
    dtype={"HomeGoals": Integer()},  
)
```

This creates a new table called "pl\_games" on the existing database connection

- `if_exists="append"` will append the data to the table if it already exists
- `index=False` is leaving the default index in the SQL table, but you can select a column from the DataFrame to use instead

```
pd.read_sql("SELECT * FROM pl_games", engine).head()
```

Now you can query the table you created!

	<b>id</b>	<b>league_name</b>	<b>season</b>	<b>HomeGoals</b>	<b>AwayGoals</b>
0	4389	England Premier League	2015/2016	0	2
1	4390	England Premier League	2015/2016	0	1
2	4391	England Premier League	2015/2016	2	2
3	4392	England Premier League	2015/2016	2	2
4	4393	England Premier League	2015/2016	4	2



# ADDITIONAL FORMATS

Preprocessing Options

Flat Files

SQL Databases

Additional Formats

Pandas has functions to read & write these **additional formats**

- These functions work similarly to `read_csv()` and `to_csv()`

Format	Read Function	Write Function	Format Description
JSON	<code>read_json</code>	<code>to_json</code>	Short for Java Script Open Notation, a common format returned by APIs (similar to a nested dictionary in Python)
Feather	<code>read_feather</code>	<code>to_feather</code>	Feather is a relatively new file format - designed to read, write, and store DataFrames as efficiently as possible
HTML	<code>read_html</code>	<code>to_html</code>	Can be used to read and write data in webpage formats, which makes it nice for scraping tables on sites like Wikipedia
Pickle	<code>read_pickle</code>	<code>to_pickle</code>	A serialized storage format that allows for quick reproduction of DataFrames. Mostly used in Machine Learning workflows
Python Dictionary	<code>pd.DataFrame</code>	<code>to_dict</code>	You can convert many Python data types to DataFrames, and can convert them to Python dictionaries with <code>to_dict()</code>

# KEY TAKEAWAYS

---



The `read_csv()` function is capable of significant **data preprocessing**

- Once you've cleaned your data, take the time to incorporate those steps into the import phase to save time and memory the next time you work with it



Pandas lets you easily **read in & write to** flat files like CSV and Excel

- Just make sure to specify the correct column delimiter for `.tsv` and `.txt` files, and the desired Excel worksheet



You can create a DataFrame from any **SQL query**

- SQLAlchemy lets you create a connection to any SQL database, and the `read_sql()` function lets you pass a query into the database to create a DataFrame – this is what real-world Pandas workflows are built on!

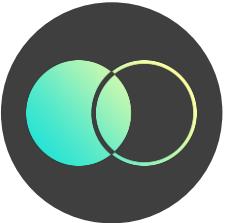


Many **additional formats** can be read in using Pandas

- Check the documentation or Google some examples when encountering a format you haven't worked with

# COMBINING DATAFRAMES

# COMBINING DATAFRAMES



In this section we'll cover **combining multiple DataFrames**, including joining data from related fields to add new columns, and appending data with the same fields to add new rows

## TOPICS WE'LL COVER:

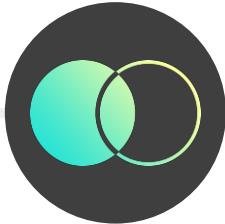
Combining Basics

Appending

Joining

## GOALS FOR THIS SECTION:

- Append the rows for similar DataFrames
- Learn the different join types and their behavior
- Join DataFrames with single & multiple related fields



# WHY MULTIPLE DATAFRAMES?

Combining Basics

Appending

Joining

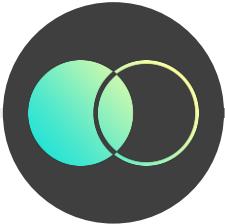
There are many scenarios where working with **multiple DataFrames** is necessary:

- Relational Databases save a lot of space by not repeating redundant data (*normalization*)
- Data may come from multiple sources (*like complementing company numbers with external data*)
- Multiple files can be used to store the same data split by different time periods or groups

item_sales.head()						
	<b>id</b>	<b>date</b>	<b>store_nbr</b>	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

store_transactions.head()			
	<b>date</b>	<b>store_nbr</b>	<b>transactions</b>
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

These two tables come from the same database, so they share the “date” and “store\_nbr” columns  
If you want to calculate transactions and sales by store, you need to join the tables by their related columns



# APPENDING & JOINING

## Combining Basics

### Appending

### Joining

There are two ways to combine DataFrames: **appending** & **joining**

- **Appending** stacks the rows from multiple DataFrames with the same column structure
- **Joining** adds related columns from one DataFrame to another, based on common values

date	store	sales
2022-05-01	1	341
2022-05-01	2	291
2022-05-01	3	493
2022-05-01	4	428
2022-05-01	5	152

date	store	sales
2022-06-01	1	67
2022-06-01	2	144
2022-06-01	3	226
2022-06-01	4	397
2022-06-01	5	163

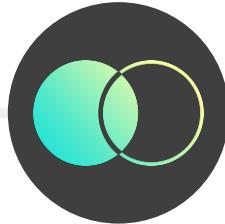
date	store	sales	region
2022-05-01	1	341	North
2022-05-01	2	291	North
2022-05-01	3	493	East
2022-05-01	4	428	East
2022-05-01	5	152	West

date	store	sales
2022-05-01	1	341
2022-05-01	2	291
2022-05-01	3	493
2022-05-01	4	428
2022-05-01	5	152
2022-06-01	1	67
2022-06-01	2	144
2022-06-01	3	226
2022-06-01	4	397
2022-06-01	5	163

Appending these two tables with the same columns added the rows from one to the other

date	store	sales	region
2022-05-01	1	341	North
2022-05-01	2	291	North
2022-05-01	3	493	East
2022-05-01	4	428	East
2022-05-01	5	152	West

Joining these two tables added the region column based on the matching store values



# APPENDING DATAFRAMES

Combining Basics

Appending

Joining

You can **append DataFrames** with the concat() function

- The columns for the DataFrames must be identical
- pd.concat([df\_1, df\_2]) will stack the rows from df\_2 at the bottom of df\_1

`first_5_rows.head()`

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

`next_5_rows.head()`

	date	id	store_nbr	family	sales	onpromotion
5	2013-01-01	5	1	BREAD/BAKERY	0.0	0
6	2013-01-01	6	1	CELEBRATION	0.0	0
7	2013-01-01	7	1	CLEANING	0.0	0
8	2013-01-01	8	1	DAIRY	0.0	0
9	2013-01-01	9	1	DELI	0.0	0

`pd.concat([df1, df2, df3, ...])`

`pd.concat([first_5_rows, next_5_rows])`

	date	id	store_nbr	family	sales	onpromotion
0	2013-01-01	0	1	AUTOMOTIVE	0.0	0
1	2013-01-01	1	1	BABY CARE	0.0	0
2	2013-01-01	2	1	BEAUTY	0.0	0
3	2013-01-01	3	1	BEVERAGES	0.0	0
4	2013-01-01	4	1	BOOKS	0.0	0
5	2013-01-01	5	1	BREAD/BAKERY	0.0	0
6	2013-01-01	6	1	CELEBRATION	0.0	0
7	2013-01-01	7	1	CLEANING	0.0	0
8	2013-01-01	8	1	DAIRY	0.0	0
9	2013-01-01	9	1	DELI	0.0	0



# ASSIGNMENT: APPENDING DATAFRAMES



 NEW MESSAGE

September 5, 2022

**From:** Ross Retail (Head of Analytics)  
**Subject:** Concatenating DataFrames?

Hey again,  
I've been looking at the data you sent to Chandler.  
Looks great, but I am working on an analysis for 2014-2015.  
Can you send me a code snippet to combine these tables?  
Thanks!

## Results Preview



	Date	Store_Number	Transaction_Count	target_pct	met_target	bonus_payable	month	day_of_week
0	2014-01-01	25	840	0.3360	False	False	1	2
1	2014-01-01	36	487	0.1948	False	False	1	2
2	2014-01-02	1	1875	0.7500	False	False	1	3
3	2014-01-02	2	2122	0.8488	False	False	1	3
4	2014-01-02	3	3350	1.3400	True	False	1	3

`transactions.tail()`

	Date	Store_Number	Transaction_Count	target_pct	met_target	bonus_payable	month	day_of_week
18341	2015-12-31	49	3828	1.5312	True	False	12	3
18342	2015-12-31	50	2948	1.1792	True	False	12	3
18343	2015-12-31	51	2892	1.1568	True	False	12	3
18344	2015-12-31	53	2300	0.9200	False	False	12	3
18345	2015-12-31	54	1572	0.6288	False	False	12	3

# SOLUTION: APPENDING DATAFRAMES



NEW MESSAGE

September 5, 2022

From: Ross Retail (Head of Analytics)  
Subject: Concatenating DataFrames?

Hey again,

I've been looking at the data you sent to Chandler.

Looks great, but I am working on an analysis for 2014-2015.

Can you send me a code snippet to combine these tables?

Thanks!

## Solution Code

```
transactions = pd.concat(  
    (pd.read_csv("transactions_2014.csv"), pd.read_csv("transactions_2015.csv"))  
).drop(["Unnamed: 0"], axis=1)  
  
transactions.head()
```

	Date	Store_Number	Transaction_Count	target_pct	met_target	bonus_payable	month	day_of_week
0	2014-01-01	25	840	0.3360	False	False	1	2
1	2014-01-01	36	487	0.1948	False	False	1	2
2	2014-01-02	1	1875	0.7500	False	False	1	3
3	2014-01-02	2	2122	0.8488	False	False	1	3
4	2014-01-02	3	3350	1.3400	True	False	1	3

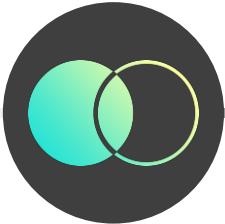
```
transactions.tail()
```

	Date	Store_Number	Transaction_Count	target_pct	met_target	bonus_payable	month	day_of_week
18341	2015-12-31	49	3828	1.5312	True	False	12	3
18342	2015-12-31	50	2948	1.1792	True	False	12	3
18343	2015-12-31	51	2892	1.1568	True	False	12	3
18344	2015-12-31	53	2300	0.9200	False	False	12	3
18345	2015-12-31	54	1572	0.6288	False	False	12	3

section09\_joining\_dfs.ipynb

Reply

Forward



# JOINING DATAFRAMES

You can **join DataFrames** with the `merge()` function

- The DataFrames must share at least one column to match the values between them

Combining Basics

Appending

Joining

```
left_df.merge(right_df,  
             how,  
             left_on,  
             right_on)
```

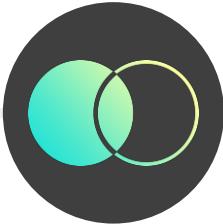
“Left” DataFrame to join with “Right”

“Right” DataFrame to join with “Left”

Type of join

Column(s) on “Left” DataFrame to join by

Column(s) on “Right” DataFrame to join by



# JOINING DATAFRAMES

Combining Basics

Appending

Joining

You can **join DataFrames** with the `merge()` function

- The DataFrames must share at least one column to match the values between them

## EXAMPLE

Adding the “transactions” by date and store number to the family-level sales table

`item_sales.head()`

	<b>id</b>	<b>date</b>	<b>store_nbr</b>	<b>family</b>	<b>sales</b>	<b>onpromotion</b>
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

`store_transactions.head()`

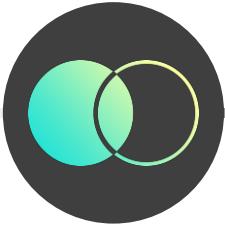
	<b>date</b>	<b>store_nbr</b>	<b>transactions</b>
0	2013-01-01	25	770
1	2013-01-02	1	2111
2	2013-01-02	2	2358
3	2013-01-02	3	3487
4	2013-01-02	4	1922

What would be the `merge()` arguments to join these two DataFrames?

- The **left DataFrame** is “item\_sales”
- The **right DataFrame** is “store\_transactions”
- The **left columns to join by** are “date” and “store\_nbr”
- The **right columns to join by** are “date” and “store\_nbr”
- What about the **join type**?



Joining these tables only on “date” or “store\_nbr” **will lead to incorrect matches** if either “date” or “store\_nbr” has more than 1 unique value



# TYPES OF JOINS

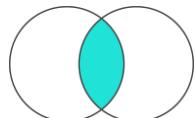
There are 4 primary **types of joins**: Inner, Left, Right, and Outer

- Inner and Left joins are the most common, while Right is rarely used

Combining Basics

Appending

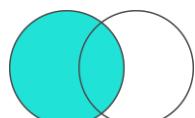
Joining



**Inner**

Returns records that exist in BOTH tables, and excludes unmatched records from either table

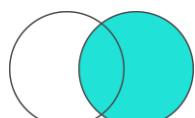
**how="inner"**



**Left**

Returns ALL records from the LEFT table, and any matching records from the RIGHT table

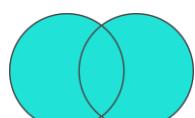
**how="left"**



**Right**

Returns ALL records from the RIGHT table, and any matching records from the LEFT table

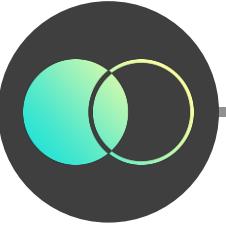
**how="right"**



**Outer**

Returns ALL records from BOTH tables, including non-matching records

**how="outer"**



# TYPES OF JOINS

Combining Basics

Appending

Joining

*Left Table*

date	store	offer	sales
2022-05-01	1	1	341
2022-05-01	2		291
2022-05-01	3	1	493
2022-05-01	4	2	428
2022-05-01	5		152
2022-06-01	1	3	67
2022-06-01	2		144
2022-06-01	3	1	226
2022-06-01	4		397
2022-06-01	5	4	163

*n=10*

*Right Table*

offer	discount
1	5%
2	10%
3	15%
4	20%
5	50%

*n=5*

*Left Join*

date	store	offer	sales	discount
2022-05-01	1	1	341	5%
2022-05-01	2		291	
2022-05-01	3	1	493	5%
2022-05-01	4	2	428	10%
2022-05-01	5		152	
2022-06-01	1	3	67	15%
2022-06-01	2		144	
2022-06-01	3	1	226	5%
2022-06-01	4		397	
2022-06-01	5	4	163	20%

*n=10*

*Inner Join*

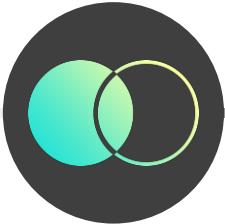
date	store	offer	sales	discount
2022-05-01	1	1	341	5%
2022-05-01	3	1	493	5%
2022-05-01	4	2	428	10%
2022-06-01	1	3	67	15%
2022-06-01	3	1	226	5%
2022-06-01	5	4	163	20%

*n=6*

*Outer Join*

date	store	offer	sales	discount
2022-05-01	1	1	341	5%
2022-05-01	2		291	
2022-05-01	3	1	493	5%
2022-05-01	4	2	428	10%
2022-05-01	5		152	
2022-06-01	1	3	67	15%
2022-06-01	2		144	
2022-06-01	3	1	226	5%
2022-06-01	4		397	
2022-06-01	5	4	163	20%
				50%

*n=11*



# EXAMPLE: INNER JOIN

Combining Basics

Appending

Joining

Left Table

item_sales						
	id	date	store_nbr	family	sales	onpromotion
0	1945944	2016-01-01	1	AUTOMOTIVE	0.000	0
1	1945945	2016-01-01	1	BABY CARE	0.000	0
2	1945946	2016-01-01	1	BEAUTY	0.000	0
3	1945947	2016-01-01	1	BEVERAGES	0.000	0
4	1945948	2016-01-01	1	BOOKS	0.000	0
...	...	...	...	...	...	...
1054939	3000883	2017-08-15	9	POULTRY	438.133	0
1054940	3000884	2017-08-15	9	PREPARED FOODS	154.553	1
1054941	3000885	2017-08-15	9	PRODUCE	2419.729	148
1054942	3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.000	8
1054943	3000887	2017-08-15	9	SEAFOOD	16.000	0

1054944 rows x 6 columns

Right Table

store_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
...	...	...	...
83483	2017-08-15	50	2804
83484	2017-08-15	51	1573
83485	2017-08-15	52	2255
83486	2017-08-15	53	932
83487	2017-08-15	54	802

83488 rows x 3 columns

Inner Join

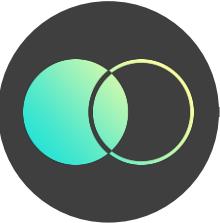
```
item_sales.merge(  
    store_transactions,  
    how="inner",  
    left_on=["store_nbr", "date"],  
    right_on=["store_nbr", "date"],  
)
```

	id	date	store_nbr	family	sales	onpromotion	transactions
0	1947957	2016-01-02	16	AUTOMOTIVE	8.000	0	373
1	1947958	2016-01-02	16	BABY CARE	0.000	0	373
2	1947959	2016-01-02	16	BEAUTY	0.000	0	373
3	1947960	2016-01-02	16	BEVERAGES	1533.000	0	373
4	1947961	2016-01-02	16	BOOKS	0.000	0	373
...	...	...	...	...	...	...	...
1026163	3000883	2017-08-15	9	POULTRY	438.133	0	2155
1026164	3000884	2017-08-15	9	PREPARED FOODS	154.553	1	2155
1026165	3000885	2017-08-15	9	PRODUCE	2419.729	148	2155
1026166	3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.000	8	2155
1026167	3000887	2017-08-15	9	SEAFOOD	16.000	0	2155

1026168 rows x 7 columns

An inner join between these two DataFrames discards all rows where "store\_nbr" and "date" don't match

Because '2016-01-01' was NOT in transactions, it is NOT included in the joined table, which leads to a reduced row count



# EXAMPLE: LEFT JOIN

Combining Basics

Appending

Joining

Left Table

item_sales					
	id	date	store_nbr	family	sales
					onpromotion
0	1945944	2016-01-01	1	AUTOMOTIVE	0.000
1	1945945	2016-01-01	1	BABY CARE	0.000
2	1945946	2016-01-01	1	BEAUTY	0.000
3	1945947	2016-01-01	1	BEVERAGES	0.000
4	1945948	2016-01-01	1	BOOKS	0.000
...	...	...	...	...	...
1054939	3000883	2017-08-15	9	POULTRY	438.133
1054940	3000884	2017-08-15	9	PREPARED FOODS	154.553
1054941	3000885	2017-08-15	9	PRODUCE	2419.729
1054942	3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.000
1054943	3000887	2017-08-15	9	SEAFOOD	16.000

1054944 rows x 6 columns

Right Table

store_transactions		
	date	store_nbr
		transactions
0	2013-01-01	25
1	2013-01-02	1
2	2013-01-02	2
3	2013-01-02	3
4	2013-01-02	4
...	...	...
83483	2017-08-15	50
83484	2017-08-15	51
83485	2017-08-15	52
83486	2017-08-15	53
83487	2017-08-15	54

83488 rows x 3 columns

Left Join

```
item_sales.merge(  
    store_transactions,  
    how="left",  
    left_on=["store_nbr", "date"],  
    right_on=["store_nbr", "date"],  
)
```

	id	date	store_nbr	family	sales	onpromotion	transactions
0	1945944	2016-01-01	1	AUTOMOTIVE	0.000	0	NaN
1	1945945	2016-01-01	1	BABY CARE	0.000	0	NaN
2	1945946	2016-01-01	1	BEAUTY	0.000	0	NaN
3	1945947	2016-01-01	1	BEVERAGES	0.000	0	NaN
4	1945948	2016-01-01	1	BOOKS	0.000	0	NaN
...	...	...	...	...	...	...	...
1054939	3000883	2017-08-15	9	POULTRY	438.133	0	2155.0
1054940	3000884	2017-08-15	9	PREPARED FOODS	154.553	1	2155.0
1054941	3000885	2017-08-15	9	PRODUCE	2419.729	148	2155.0
1054942	3000886	2017-08-15	9	SCHOOL AND OFFICE SUPPLIES	121.000	8	2155.0
1054943	3000887	2017-08-15	9	SEAFOOD	16.000	0	2155.0

1054944 rows x 7 columns

A left join will preserve all rows in the left table, and simply includes NaN values in “transactions” when no match is found

**PRO TIP:** If you’re performing a join between two tables for the first time, start with a left join to grasp the potential data loss from an inner join and decide which works best for your analysis



# ASSIGNMENT: JOINING DATAFRAMES



## NEW MESSAGE

September 10, 2022

From: **Joey Justin Time** (Logistics)

Subject: Sales Patterns

Hey there!

I'm trying to get a better handle on sales patterns at our stores, as we're looking at opening more regional distribution centers for faster delivery.

Can you join retail.csv with stores.csv?

Once you have that, plot total average sales by city, the sum of sales by "type" over time, and a stacked bar chart with sales by month, with "type" as the "stacks".

Thanks!

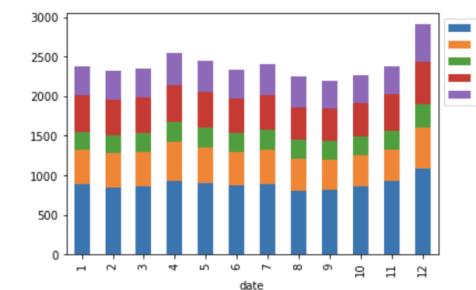
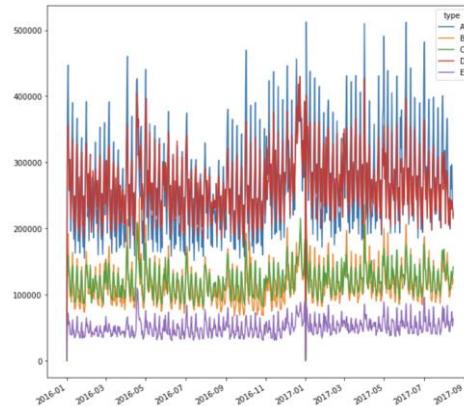
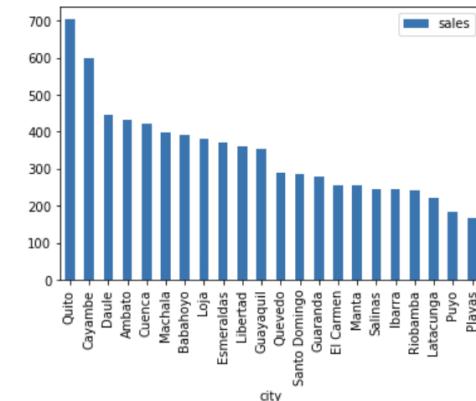


section07\_time\_series.ipynb

Reply

Forward

## Results Preview



# SOLUTION: JOINING DATAFRAMES

 **1 NEW MESSAGE**  
September 10, 2022

**From:** Joey Justin Time (Logistics)  
**Subject:** Sales Patterns

Hey there!

I'm trying to get a better handle on sales patterns at our stores, as we're looking at opening more regional distribution centers for faster delivery.

Can you join retail.csv with stores.csv?

Once you have that, plot total average sales by city, the sum of sales by "type" over time, and a stacked bar chart with sales by month, with "type" as the "stacks".

Thanks!

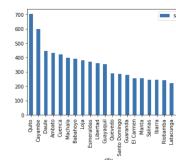
 

## Solution Code

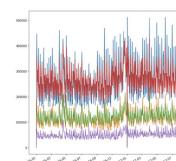
```
retail = pd.read_csv("../retail/retail_2016_2017.csv", parse_dates=[ "date" ])
stores = pd.read_csv("../retail/stores.csv")
```

```
retail_stores = retail.merge(stores, how="left", on="store_nbr")
```

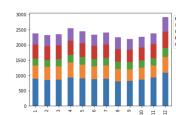
```
retail_stores.groupby([ "city" ]).agg({ "sales": "mean" }).sort_values(
    by="sales", ascending=False
).plot.bar()
```

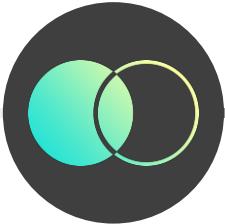


```
retail_stores.pivot_table(
    index=[ "date" ], columns="type", values="sales", aggfunc="sum"
).plot(figsize=(10, 10))
```



```
retail_stores.pivot_table(
    index="type", columns=retail_stores[ "date" ].dt.month, values="sales", aggfunc="mean"
).T.plot.bar(stacked=True).legend(bbox_to_anchor=(1, 1))
```





# THE JOIN METHOD

The `.join()` method can join two DataFrames based on their index

Combining Basics

Appending

Joining

item\_sales\_short

	date	store_nbr	family	sales
0	2013-01-01	1	PRODUCE	10
1	2013-01-01	2	PRODUCE	20
2	2013-01-02	1	PRODUCE	30
3	2013-01-02	2	PRODUCE	40
4	2013-01-03	1	PRODUCE	50

transactions\_short

	date	store_nbr	transactions
0	2012-12-31	1	100
1	2013-01-01	1	200
2	2013-01-01	2	300
3	2013-01-02	1	400
4	2013-01-02	2	500

item\_sales\_short.join(transactions\_short, rsuffix="2")

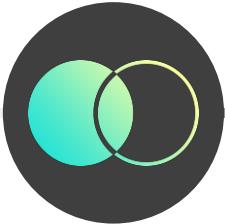
	date	store_nbr	family	sales	date2	store_nbr2	transactions
0	2013-01-01	1	PRODUCE	10	2012-12-31	1	100
1	2013-01-01	2	PRODUCE	20	2013-01-01	1	200
2	2013-01-02	1	PRODUCE	30	2013-01-01	2	300
3	2013-01-02	2	PRODUCE	40	2013-01-02	1	400
4	2013-01-03	1	PRODUCE	50	2013-01-02	2	500



The "transactions" column was joined based on the index here, not on matches for "date" and "store\_nbr"

(`rsuffix="2"` lets Pandas know not to include the first two columns from the right DataFrame – they are duplicates)





# THE JOIN METHOD

Combining Basics

Appending

Joining

You can also join based on indices with .merge()

- Specify **left\_index=True** and **right\_index=True**

item\_sales\_short

	date	store_nbr	family	sales
0	2013-01-01	1	PRODUCE	10
1	2013-01-01	2	PRODUCE	20
2	2013-01-02	1	PRODUCE	30
3	2013-01-02	2	PRODUCE	40
4	2013-01-03	1	PRODUCE	50

item\_sales\_short.merge(transactions\_short, left\_index=True, right\_index=True)

	date_x	store_nbr_x	family	sales	date_y	store_nbr_y	transactions
0	2013-01-01	1	PRODUCE	10	2012-12-31	1	100
1	2013-01-01	2	PRODUCE	20	2013-01-01	1	200
2	2013-01-02	1	PRODUCE	30	2013-01-01	2	300
3	2013-01-02	2	PRODUCE	40	2013-01-02	1	400
4	2013-01-03	1	PRODUCE	50	2013-01-02	2	500

transactions\_short

	date	store_nbr	transactions
0	2012-12-31	1	100
1	2013-01-01	1	200
2	2013-01-01	2	300
3	2013-01-02	1	400
4	2013-01-02	2	500



**PRO TIP:** You will see examples of joining tables with the join method, but it's rare to have perfect alignment of DataFrame indices in practice; since merge() can join on indices as well as columns, it is the only method you need to know

# KEY TAKEAWAYS

---



## Combining DataFrames is a key step in Pandas' data analysis workflow

- *Being able to connect multiple database tables and/or external data sources together is necessary to answer many questions and drive your analysis further*



## Use **concat()** to stack DataFrames with the same column structure

- *This is equivalent to a SQL union, and is used to add new rows to a DataFrame*



## Use **merge()** to join DataFrames by their related fields

- *This is equivalent to a SQL join, and is used to add new columns to a DataFrame*
- *You can perform left, inner, right, and outer joins depending on the records you'd like to return*



## Keep an eye on the **shape of your data** as you join tables

- *Is your joined table smaller than one or both of your starting tables? Is it bigger? Is this behavior expected?*

# FINAL PROJECT

# PROJECT DATA: OVERVIEW

## Transactions

```
transactions.head(5)
```

	household_key	BASKET_ID	DAY	PRODUCT_ID	QUANTITY	SALES_VALUE	STORE_ID	RETAIL_DISC	WEEK_NO	COUPON_DISC	COUPON_MATCH_DISC
0	1364	26984896261	1	842930	1	2.19	31742	0.00	1	0.0	0.0
1	1364	26984896261	1	897044	1	2.99	31742	-0.40	1	0.0	0.0
2	1364	26984896261	1	920955	1	3.09	31742	0.00	1	0.0	0.0
3	1364	26984896261	1	937406	1	2.50	31742	-0.99	1	0.0	0.0
4	1364	26984896261	1	981760	1	0.60	31742	-0.79	1	0.0	0.0

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2146311 entries, 0 to 2146310
Data columns (total 11 columns):
 #   Column           Dtype  
 --- 
 0   household_key   int64  
 1   BASKET_ID       int64  
 2   DAY              int64  
 3   PRODUCT_ID      int64  
 4   QUANTITY         int64  
 5   SALES_VALUE     float64
 6   STORE_ID         int64  
 7   RETAIL_DISC     float64
 8   WEEK_NO          int64  
 9   COUPON_DISC     float64
 10  COUPON_MATCH_DISC float64
dtypes: float64(4), int64(7)
memory usage: 180.1 MB
```

```
cols = [ "QUANTITY", "SALES_VALUE", "RETAIL_DISC", "COUPON_DISC", "COUPON_MATCH_DISC"]
```

```
transactions.loc[:, cols].describe().round()
```

	QUANTITY	SALES_VALUE	RETAIL_DISC	COUPON_DISC	COUPON_MATCH_DISC
count	2146311.0	2146311.0	2146311.0	2146311.0	2146311.0
mean	101.0	3.0	-1.0	-0.0	-0.0
std	1152.0	4.0	1.0	0.0	0.0
min	0.0	0.0	-130.0	-56.0	-8.0
25%	1.0	1.0	-1.0	0.0	0.0
50%	1.0	2.0	0.0	0.0	0.0
75%	1.0	3.0	0.0	0.0	0.0
max	89638.0	840.0	4.0	0.0	0.0



**MAVEN**  
MEGA MART

# PROJECT DATA: OVERVIEW

## Products

PRODUCT_ID	MANUFACTURER	DEPARTMENT	BRAND	COMMODITY_DESC	SUB_COMMODITY_DESC	CURR_SIZE_OF_PRODUCT
0	25671	2	GROCERY	National	FRZN ICE	ICE - CRUSHED/CUBED
1	26081	2	MISC. TRANS.	National	NO COMMODITY DESCRIPTION	NO SUBCOMMODITY DESCRIPTION
2	26093	69	PASTRY	Private	BREAD	BREAD:ITALIAN/FRENCH
3	26190	69	GROCERY	Private	FRUIT - SHELF STABLE	APPLE SAUCE
4	26355	69	GROCERY	Private	COOKIES/CONES	SPECIALTY COOKIES

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 92353 entries, 0 to 92352
Data columns (total 7 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   PRODUCT_ID      92353 non-null   int64  
 1   MANUFACTURER    92353 non-null   int64  
 2   DEPARTMENT      92353 non-null   object  
 3   BRAND            92353 non-null   object  
 4   COMMODITY_DESC   92353 non-null   object  
 5   SUB_COMMODITY_DESC 92353 non-null   object  
 6   CURR_SIZE_OF_PRODUCT 92353 non-null   object  
dtypes: int64(2), object(5)
memory usage: 31.1 MB
```



**MAVEN**  
MEGA MART

# PROJECT DATA: OVERVIEW

## Households

```
demographics.head()
```

	AGE_DESC	MARITAL_STATUS_CODE	INCOME_DESC	HOMEOWNER_DESC	HH_COMP_DESC	HOUSEHOLD_SIZE_DESC	KID_CATEGORY_DESC	household_key
0	65+	A	35-49K	Homeowner	2 Adults No Kids	2	None/Unknown	1
1	45-54	A	50-74K	Homeowner	2 Adults No Kids	2	None/Unknown	7
2	25-34	U	25-34K	Unknown	2 Adults Kids	3	1	8
3	25-34	U	75-99K	Homeowner	2 Adults Kids	4	2	13
4	45-54	B	50-74K	Homeowner	Single Female	1	None/Unknown	16

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 801 entries, 0 to 800
Data columns (total 8 columns):
 #   Column           Non-Null Count  Dtype  
 --- 
 0   AGE_DESC         801 non-null    object 
 1   MARITAL_STATUS_CODE 801 non-null    object 
 2   INCOME_DESC       801 non-null    object 
 3   HOMEOWNER_DESC    801 non-null    object 
 4   HH_COMP_DESC      801 non-null    object 
 5   HOUSEHOLD_SIZE_DESC 801 non-null    object 
 6   KID_CATEGORY_DESC 801 non-null    object 
 7   household_key     801 non-null    int64  
dtypes: int64(1), object(7)
memory usage: 352.7 KB
```



**MAVEN**  
MEGA MART

# ASSIGNMENT: FINAL PROJECT



## NEW MESSAGE

September 20, 2022

**From:** Ross Retail (Head of Analytics)  
**Subject:** Maven Acquisition Target Data

Hey again,

We're getting closer to a final decision on the acquisition.

Management was impressed by your findings, and we need to discuss a promotion for you soon.

They did have a few more questions we need to answer before they feel comfortable moving forward.

Those questions are in the attached notebook.

Thanks!

section10\_final\_project.ipynb

Reply

Forward

## Key Objectives

1. Read in data from multiple csv files
2. Explore the data (*millions of rows!*)
3. Join multiple DataFrames
4. Create new columns to aid in analysis
5. Filter, sort, and aggregate the data to pinpoint and summarize important information
6. Work with datetime fields to analyze time series
7. Build plots to communicate key insights
8. Optimize the import workflow
9. Write out summary tables for stakeholders