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

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
```

'pd' is the standard alias for the Pandas library

```
sales = [0, 5, 155, 0, 518, 0, 1827, 616, 317, 325]

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

sales_series
```

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

The name argument lets you specify a name

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
```

The series name and data type are stored as well

SERIES PROPERTIES

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 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
Name: Sales, dtype: int64
```

These are integers

```
sales_series.astype("float")
```

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

This converts them to floats

```
sales_series.astype("bool")
```

```
0    False
1     True
2     True
3    False
4     True
Name: Sales, dtype: bool
```

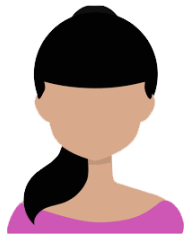
This converts them to Booleans (0 is False, others are True)

```
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!

section02_Series.ipynb

Reply Forward

Results Preview

```
oil_series
```

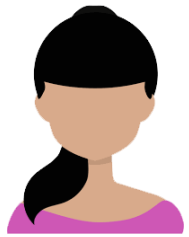
```
0     52.22
1     51.44
2     51.98
3     52.01
4     52.82
      ...
95    45.84
96    47.28
97    47.81
98    47.83
99    48.86

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!

section02_Series.ipynb

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
```

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

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

```
2    155
3      0
Name: Sales, dtype: int64
```

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

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

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

Examples:

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

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

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

THE ILOC METHOD

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

```
sales_series
```

Note that this Series has a custom index

```
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 3rd 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 3rd to the 4th position (stop is non-inclusive)

THE LOC METHOD

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

Examples:

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

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

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

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
```

Note that 'coffee' is used as an index value twice

```
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

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

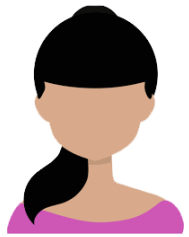
This returns a DataFrame by default, with the previous index values stored as a new column

```
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
```

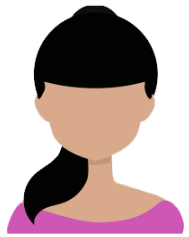
```
date
2016-12-20    52.22
2016-12-21    51.44
2016-12-22    51.98
2016-12-23    52.01
2016-12-27    52.82
              ...
2017-05-09    45.84
2017-05-10    47.28
2017-05-11    47.81
2017-05-12    47.83
2017-05-15    48.86
Name: oil_prices, Length: 100, dtype: float64
```

```
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

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
```

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

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

sales_series.loc[mask]
```

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

*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

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

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()**

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

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

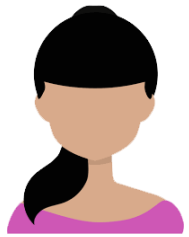
```
sales_series.sort_index()
```

```
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

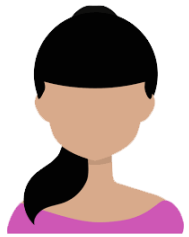
Reply Forward

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!

section02_Series.ipynb

Reply Forward

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

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
```

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

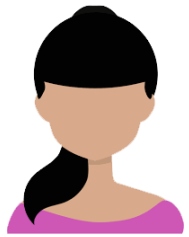
```
clean = prices.str.strip("$").astype("float")

clean
```

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

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!

section02_Series.ipynb

Reply Forward

Results Preview

```
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
```

```
max_price
```

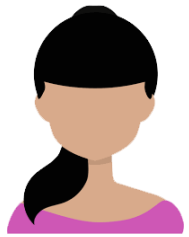
```
54.48
```

```
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
```

```
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
```

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!

section02_Series.ipynb

Reply Forward

Solution Code

```
oil_series * 1.1 + 2
```

```
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
```

```
max_price = oil_series.max()

max_price
```

```
54.48
```

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

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
```

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

```
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
```

NUMERIC SERIES AGGREGATION

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

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*

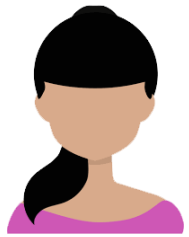
```
items.unique()
```

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

```
items.nunique()
```

```
4
```

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 data? Normalize the results to a percentage.

Thanks!

section02_Series.ipynb

Reply Forward

Results Preview

```
1134.54
```

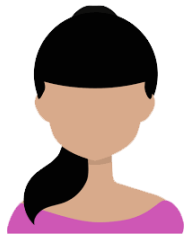
```
49.32782608695651
```

```
47
```

```
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!

section02_Series.ipynb

Reply Forward

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[6:7].isin(["1", "2"])].count()
```

```
47
```

```
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

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
```

np.nan creates a NaN value

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

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

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 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

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
```

```
0    COMPLETE
1         NaN
2         NaN
3         NaN
4    COMPLETE
dtype: object
```

```
checklist.isna()
```

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

You can use this as a Boolean mask!

```
checklist.isna().sum()
```

```
3
```

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

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

```
checklist.value_counts()
```

```
COMPLETE    2
dtype: int64
```

```
checklist.value_counts(dropna=False)
```

```
NaN         3
COMPLETE    2
dtype: int64
```

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

HANDLING MISSING DATA

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
```

```
0    COMPLETE
1         NaN
2         NaN
3         NaN
4    COMPLETE
dtype: object
```

```
checklist.dropna()
```

```
0    COMPLETE
4    COMPLETE
dtype: object
```

Note that the index has gaps, so you can use .reset_index() to restore the range of integers

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

```
checklist
```

```
0    COMPLETE
1         NaN
2         NaN
3         NaN
4    COMPLETE
dtype: object
```

```
checklist.fillna("INCOMPLETE")
```

```
0      COMPLETE
1    INCOMPLETE
2    INCOMPLETE
3    INCOMPLETE
4      COMPLETE
dtype: object
```

HANDLING MISSING DATA

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
```

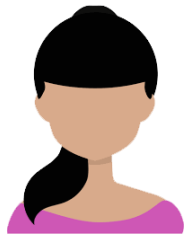
*Do you **impute** them with the mean?*

```
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

```
[redacted]
```

```
2
```

```
[redacted]
```

```
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

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

- This applies the function to every row in the Series, 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 90% 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)
```

A logical expression that evaluates to True or False

Value to return when the expression is False

Series or DataFrame to evaluate data from

Whether to perform the operation in place (default 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 false 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

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

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

1 **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

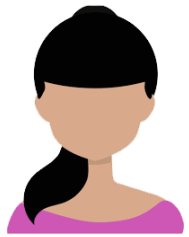
Reply Forward

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

Reply Forward

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*