4.5.2

Selecting Rows

We have multiple ways to select rows by using `loc` and `iloc`. Specifically, we can select rows by integer-based location; by index; by filtering; or by combining filters. Let's discuss when we would use each option and how to do so.

Select Rows by Integer-Based Location

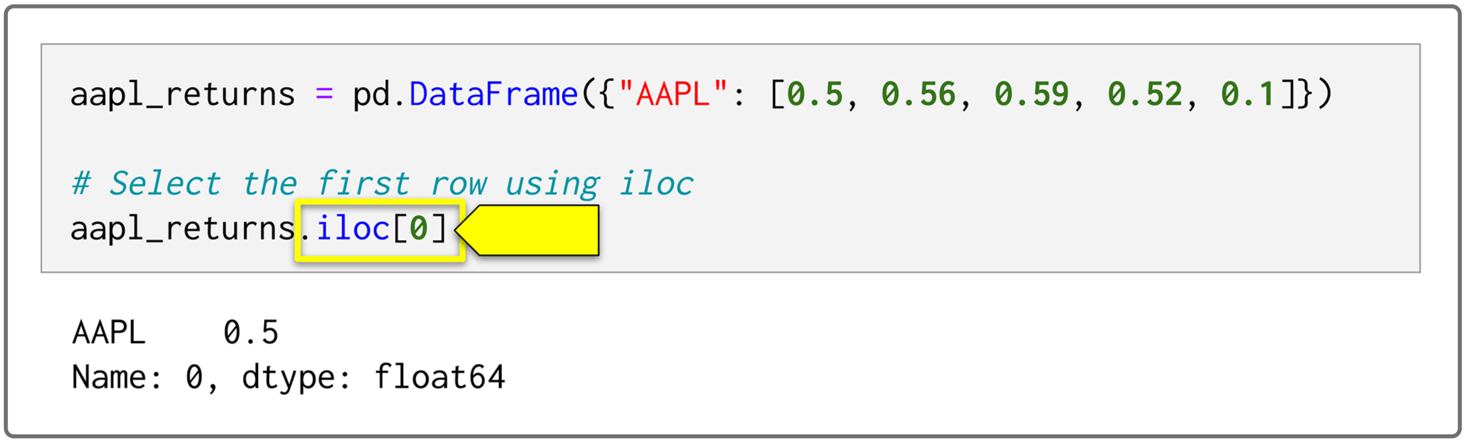
Say that we want to more closely examine the first row in a DataFrame. The easiest way to select that row is to use integer-based location. With **integer-based location**, we select a row by using its index number in the `iloc` function.

The following code uses the `iloc` function to select the first row of a DataFrame:

```
# Create a DataFrame
aapl_returns = pd.DataFrame({"AAPL": [0.5, 0.56, 0.59, 0.52, 0.1]})

# Select the first row using iloc
aapl_returns.iloc[0]
```

Running the preceding code produces the output that the following image shows:



```
aapl_returns = pd.DataFrame({"AAPL": [0.5, 0.56, 0.59, 0.52, 0.1]})

# Select the first row using iloc
aapl_returns.iloc[0]
```

```
AAPL    0.5
Name: 0, dtype: float64
```

Did you notice that the `iloc` function uses brackets instead of parentheses? This is a Pandas design choice, because the brackets mimic Python list indexing.

NOTE

If the data existed in a Python list instead of a Pandas DataFrame, you could use list indexing to select the first item, as the following code shows:

```
aapl_returns = [0.5, 0.56, 0.59, 0.52, 0.1]
first_item = aapl_returns[0]
first_item
```

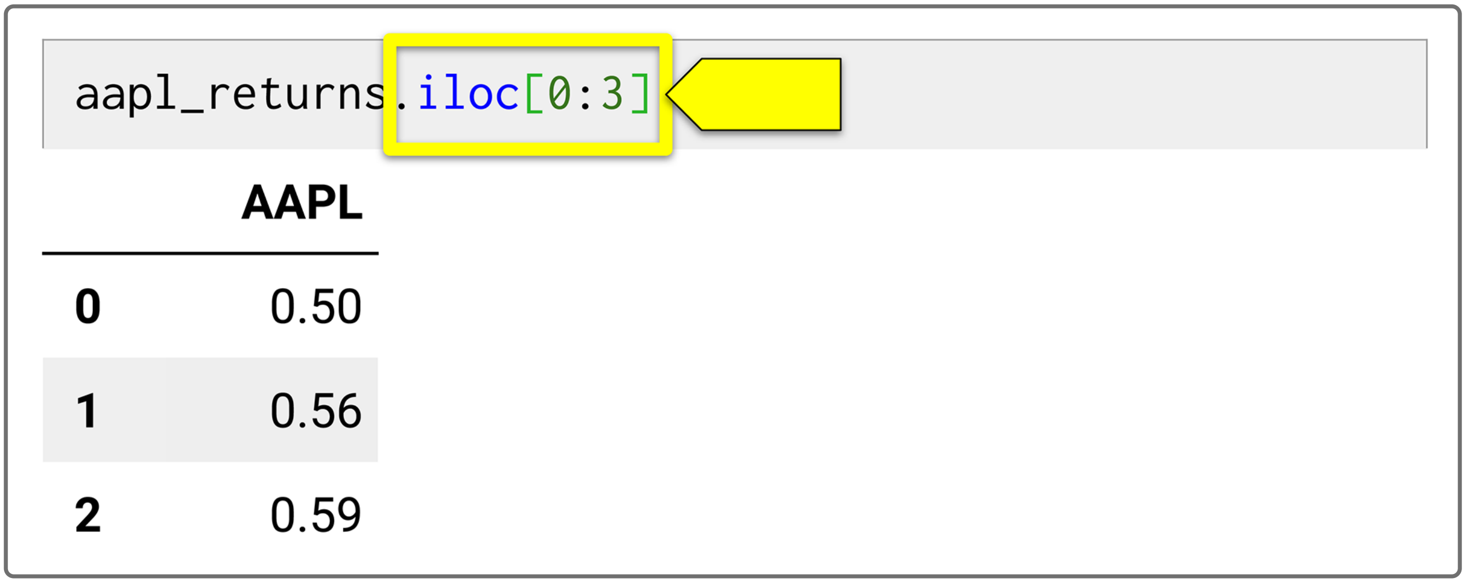
We can also use the Python list syntax with the Pandas `iloc` function to return a range of rows that's based on a numerical index range. We use the same `start:end` range syntax that Python lists use, as the following code shows:

```
aapl_returns.iloc[0:3]
```

REWIND

The Python range syntax selects elements that include the first number in the range but not the second. For example, a range of `[0:3]` includes Elements 0, 1, and 2 but not Element 3.

Running the preceding code produces the output that the following image shows:



```
aapl_returns.iloc[0:3]
```

	AAPL
0	0.50
1	0.56
2	0.59

In the preceding image, notice that output consists of the resulting subset of the DataFrame. This subset contains the index numbers 0, 1, and 2 and the elements 0.50, 0.56, and 0.59.

Now, practice using the `iloc` function with the range syntax yourself in the following Skill Drill:

SKILL DRILL

Use the following code to complete the task that follows:

```
aapl_returns = pd.DataFrame({"AAPL": [0.5, 0.56, 0.59, 0.52, 0.1, 0.75, 0.47]})
```

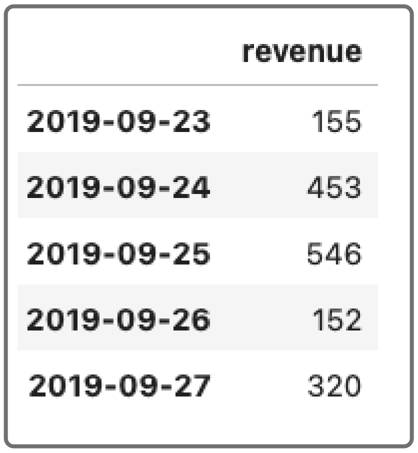
Use the `iloc` function with the range syntax to return the last three elements of the `aapl_returns` DataFrame. Try to do this task by using at least two forms of the range syntax. (Here's a hint: you can access the last element in a Python list by using −1.)

Select Rows by Index

You can use an integer-based location if you know the indexes that you need. But, what if the area of interest is a date range in the middle of a huge DataFrame? The following code shows an example that uses 1,000 days of data from January 1, 2017 to September 27, 2019:

```
import numpy as np
revenue_df = pd.DataFrame(
    np.random.randint(low=1, high=1000, size=1000),
    index=pd.date_range('1/1/2017', periods=1000),
    columns=["revenue"])
revenue_df.tail()
```

Running the preceding code produces the output that the following image shows:



	revenue
2019-09-23	155
2019-09-24	453
2019-09-25	546
2019-09-26	152
2019-09-27	320

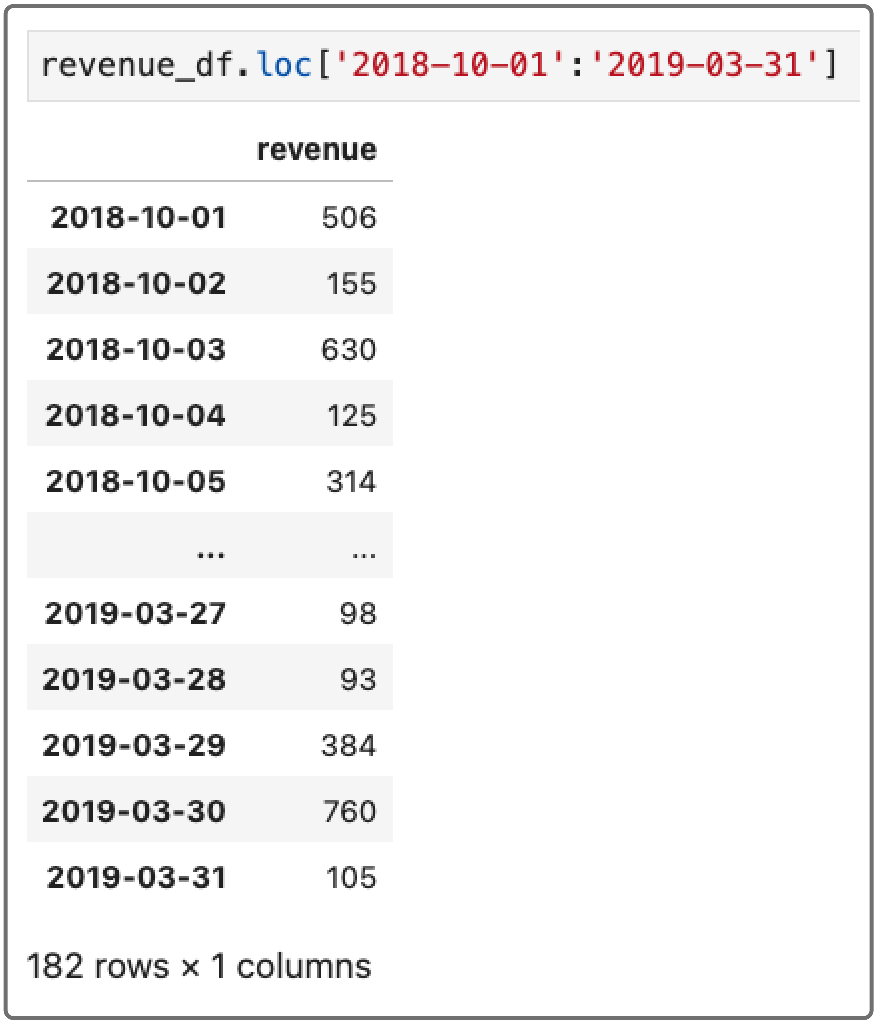
In the preceding image, notice that the output consists of the last five rows of the DataFrame. These have index values that range from `2019-09-23` to `2019-09-27` (meaning September 23, 2019 to September 27, 2019).

Say that we want to analyze a subset of this data—from the last quarter of 2018 to the first quarter of 2019—to assess the business during that period. It would be challenging to do this with the `iloc` function, because the data that we want exists in the middle of the DataFrame. We'd need to determine the exact starting and ending index positions through trial and error. And, that might take a while with a DataFrame of this size! Instead, it would be ideal to select only the dates that we want. This is where the `loc` function comes in.

The `loc` function resembles `iloc`, but we supply the index labels of just the dates that we need. For example, the following code selects the range of dates from October 1, 2018 to March 31, 2019—the last quarter of 2018 through the first quarter of 2019:

```
# Select the last quarter of 2018 to the first quarter of 2019
revenue_df.loc['2018-10-01':'2019-03-31']
```

Running the preceding code produces the output that the following image shows:



```
revenue_df.loc['2018-10-01':'2019-03-31']
```

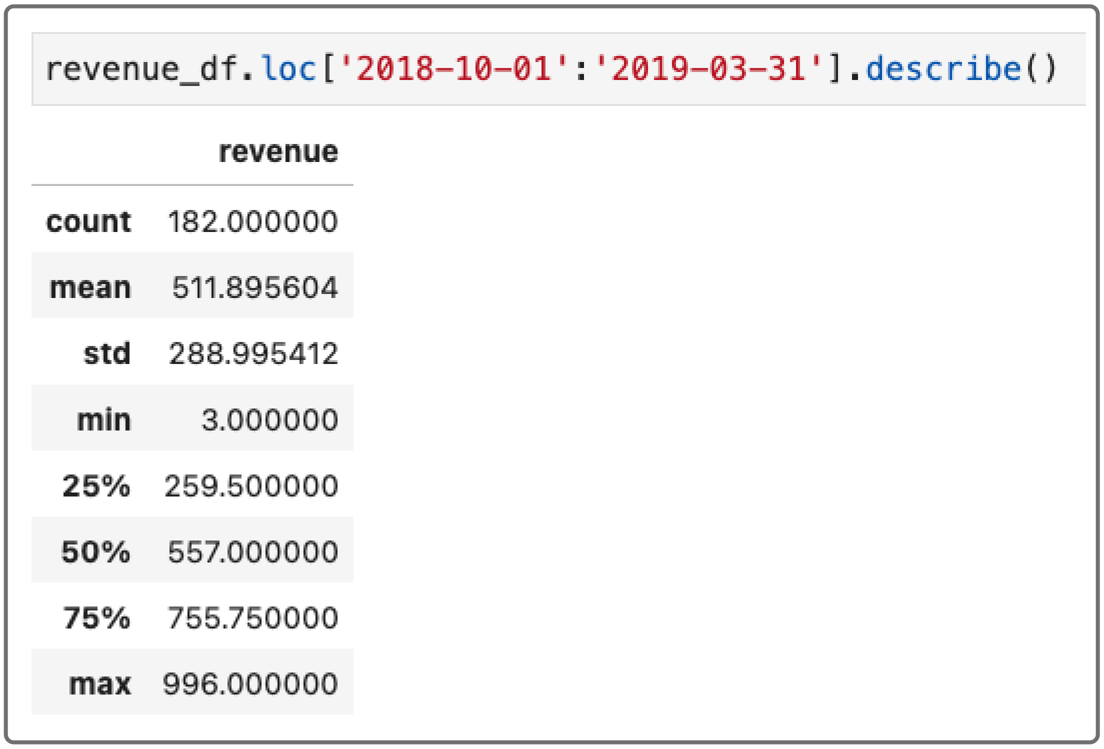
	revenue
2018-10-01	506
2018-10-02	155
2018-10-03	630
2018-10-04	125
2018-10-05	314
...	...
2019-03-27	98
2019-03-28	93
2019-03-29	384
2019-03-30	760
2019-03-31	105

182 rows × 1 columns

Now, we can analyze this range of dates by using `describe`. First, we call the `describe` function, as the following code shows:

```
# Describe the data by using summary statistics
revenue_df.loc['2018-10-01':'2019-03-31'].describe()
```

Running the preceding code produces the output that the following image shows:



```
revenue_df.loc['2018-10-01':'2019-03-31'].describe()
```

	revenue
count	182.000000
mean	511.895604
std	288.995412
min	3.000000
25%	259.500000
50%	557.000000
75%	755.750000
max	996.000000

In the preceding image, notice that we've now zoomed in on the summary statistics for a subset of the original DataFrame. We can confirm this by the `count` value, which is down to 182 from the original 1,000 observations. We can now do an analysis on just this subset of the data.

Select Rows by Filtering

Often, we want to select rows by values that don't exist in the index. For example, consider the produce sales DataFrame from an earlier lesson, as the following code shows:

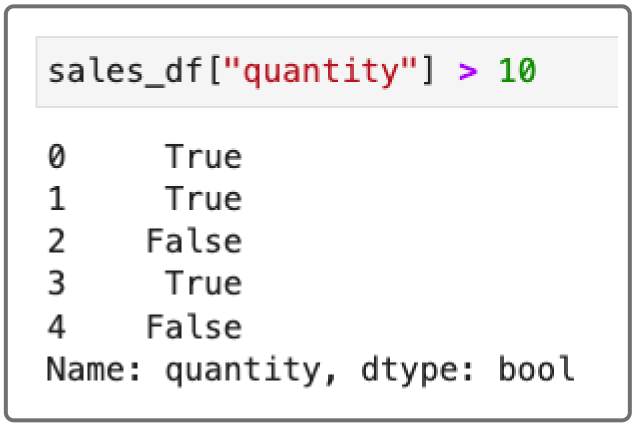
```
sales_df = pd.DataFrame({"item": ["tomatoes", "onions", "potatoes", "tomatoes", "mushrooms"],
                "quantity": [15, 26, 10, 12, 2],
                "total_price": [50.25, 39.52, 50.01, 40.20, 8.46]})
sales_df.head()
```

Because we didn't specify an index when creating that DataFrame, Pandas created a sequential integer index for us. But, what if we want to filter by item or quantity? We can do this by using a conditional statement that includes the column we want to filter on.

The following code shows a conditional statement that includes the "quantity" column:

```
sales_df["quantity"] > 10
```

The preceding statement will individually check the quantity value in every row. If a particular row has a quantity value that's greater than 10, the statement will mark that row with `True`. Otherwise, it will mark that row with `False`. The output will consist of a Pandas Series that has the same number of rows as the original DataFrame, as the following image shows.



```
sales_df["quantity"] > 10

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

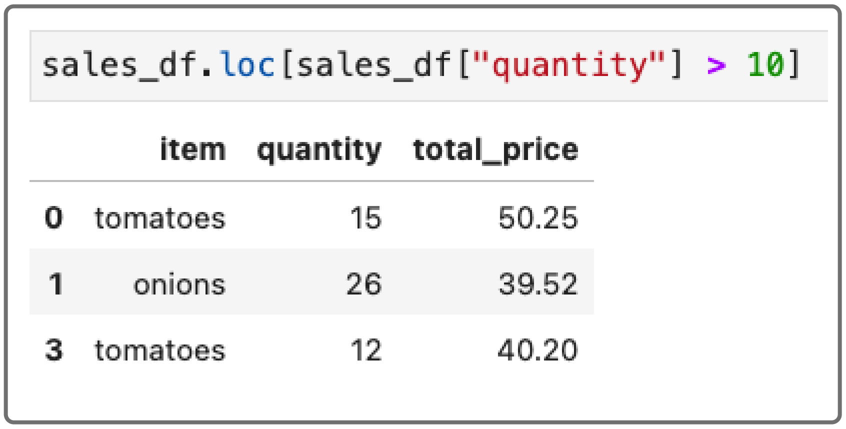
In the preceding image, notice that the values at Indexes 0, 1, and 3 are all `True`. And, the values at Indexes 2 and 4 are both `False`.

Next, we pass these Boolean values to the `loc` function, as the following code shows:

```
filter = sales_df["quantity"] > 10
sales_df.loc[filter]
```

In the preceding code, the `loc` function returns the rows for which the value is `True`.

Running the preceding code products the output that the following image shows:



```
sales_df.loc[sales_df["quantity"] > 10]
```

	item	quantity	total_price
0	tomatoes	15	50.25
1	onions	26	39.52
3	tomatoes	12	40.20

In the preceding image, notice that the output consists of a DataFrame that has three rows—specifically, the rows that have Indexes 0, 1, and 3! Recall that those are the rows that have a quantity greater than 10.

Note that we can also write the code in a single line, as the following code shows:

```
sales_df.loc[sales_df["quantity"] > 10]
```

Select Rows by Combining Filters

Often, we want to filter data based on multiple conditions. Luckily, the `loc` function makes this straightforward. To select the rows where two conditions are both true, we use the AND operator, which is the ampersand (`&`). To select the rows where either of two conditions are true, we use the OR operator, which is the pipe (`|`).

NOTE

You can type the pipe (`|`) on a keyboard by pressing Shift+backslash (`\`).

For example, consider the same sales DataFrame, as the following code shows:

```
sales_df = pd.DataFrame({"item": ["tomatoes", "onions", "potatoes", "tomatoes", "mushrooms"],
                "quantity": [15, 26, 10, 12, 2],
                "total_price": [50.25, 39.52, 50.01, 40.20, 8.46]})

The following code selects the rows where the quantity is greater than ten and the total price is less than 4
```

{.language-python} sales_df.loc[(sales_df["quantity"] > 10) & (sales_df["total_price"] < 45)]

```
Running the preceding code products the output that the following image shows:

![A screenshot depicts Rows 1 and 3.](assets//data-4-5-multiple-filter.png)

In the preceding image, notice that the output is a DataFrame that consists of all the rows where both
statements are true.

> **Important** When using multiple conditional statements, we must wrap each conditional statement in
parentheses. Otherwise, the code will return an error.

If we change the ampersand (`&`) to a pipe (`|`), the code will select all the rows where either the
quantity is greater than ten, the price is less than 45, or both statements are true. Here’s the code:
```

{.language-python} sales_df.loc[(sales_df["quantity"] > 10) | (sales_df["total_price"] < 45)]

```
Running the preceding code produces the output that the following image shows:

![A screenshot depicts Rows 1 and 3.](assets//data-4-5-filter-pipe.png)

In the preceding image, notice that the output is a DataFrame that consists of all the rows where either
one or both of the statements are true..

### Select Rows with mean, min, or max

Recall that the `min` and `max` functions show the minimum and maximum values, respectively, for every
column separately. That makes it difficult to investigate the data. For example, what if we want to know
which item has the lowest total price? By using `loc`, we now have a way to find that out!

The following code determines the item that has the lowest total price:
```

{.language-python} min_price = sales_df["total_price"].min() min_price_row = sales_df.loc[sales_df["total_price"] == min_price] min_price_row

```
In the preceding code, notice that we first set `min_price` to a DataFrame consisting of all the rows where
the total price matches the minimum total price from the whole dataset. We then set `min_price_row` to XXX.
We thus quickly identify that the item with the lowest total price is mushrooms, as the following image
shows:

![A screenshot depicts Row 4.](assets//data-4-5-show-min-price-row.png)

In the preceding image, notice that the output consists of Row 4, which has “mushrooms" in the "item"
column, 2 in the "quantity" column, and 8.46 in the "total_price" column.

Now, practice selecting a row with the `max` function yourself in the following Skill Drill:

> **Skill Drill** Use the following code to complete the task that follows:
>
> ```{.language-python}
```

```
> min_price = sales_df["total_price"].min()
> min_price_row = sales_df.loc[sales_df["total_price"] == min_price]
> min_price_row
> ```
>
> Modify the code to select the row that has the maximum quantity.

We can also select rows with the `mean` function&mdash;thus performing a statistical function on the rows.
For example, we can find the mean price of tomatoes, as the following code shows:
```

{.language-python} sales_df.loc[sales_df["item"] == "tomatoes", ["total_price"]].mean()

```
The output of the preceding code is 45.225, which is the average of the total prices in the two rows that
include "tomatoes".

Now that you’ve learned how to select rows, you’ll next learn how to select columns.

## Selecting Columns

While selecting rows from your data, you might want to also select specific columns.

The good news is that you use the same functions to select columns as you use to select rows. You just add
a second parameter, as the following syntax shows: `iloc[rows, columns]` and `loc[rows, columns]`.

Specifically, we specify the subset of rows, add a comma, and then specify the subset of columns, as the
following code shows:
```

{.language-python} sales_df.iloc[0:3, [0, 2]]

```
Running the preceding code produces the output that the following image shows:

![A screenshot depicts the output.](assets//data-4-5-show-iloc-rows-columns.png)

In the preceding image, notice that the output consists of the first and third columns for the first three
rows of data in the DataFrame.

If we want to use the `loc` function instead of `iloc`, we use similar syntax. But, we can filter rows by
using column values, as the following code shows:
```

{.language-python} sales_df.loc[sales_df["item"] == "tomatoes", ["quantity", "total_price"]]

```
Running the preceding code produces the output that the following image shows:

![A screenshot depicts the output.](assets//data-4-5-show-loc-columns.png)

In the preceding image, notice that the output consists of the "quantity" and "total_price" columns for any
rows that include "tomatoes" in the "items" column.
```

```
To select columns without filtering for rows, we use a colon (`:`) for the rows argument, as the following
code shows:
```

{.language-python} sales_df.loc[:, ["item", "total_price"]]

```
Running the preceding code produces the output that the following image shows:

![A screenshot depicts the output.](assets//data-4-5-show-loc-colon.png)

In the preceding image, notice that the output consists of the "item" and "total_price" columns for all the
rows.

Now that you’ve learned how to select columns, you’ll next learn how to sort values by columns.

## Sorting Values by Columns

To sort a DataFrame, we use the `sort_values` function along with the column or list of columns that we
want to sort by, as the following code shows:
```

{.language-python} sales_df.sort_values("total_price")

```
Running the preceding code produces the output that the following image shows:

![A screenshot depicts the output.](assets//data-4-5-sorted-dataframe.png)

In the preceding image, notice that the DataFrame is sorted by the "total_price" column. That is, the rows
appear in ascending order according to the values in the "total_price" column.

To sort in descending order, we specify `ascending=False`, as the following code shows:
```

{.language-python} sales_df.sort_values("total_price", ascending=False)

```
![“An image shows the result of running the code 'sales_df.sort_values("total_price", ascending=False)'”]
(assets//data-4-5-sorted-dataframe-descending.png)

To sort by multiple columns, you can pass a list of column names, as the following code shows:
```

{.language-python} sales_df.sort_values([“total_price”, “item”], ascending=False) ```

Now that you’ve learned how to sort values by columns, you’ll next have the opportunity to put your new skills together in the following activity.

© 2020 - 2022 Trilogy Education Services, a 2U, Inc. brand. All Rights Reserved.