4.3.2

Handling Missing Data

Analysts commonly encounter the problem of missing data. It can result from a human or computer error or even from a real-world event.

Before you start any analysis, you need to handle missing data values to avoid erroneous visualizations and incorrect calculations.

Pandas typically represents missing values by using a data type that known as **NaN**, which stands for **Not a Number**. You can think of a `NaN` value as a placeholder for missing data.

DEEP DIVE ▲

DEEP DIVE

Every programming language has ways to represent missing values, depending on what you want to do with the data. Two of the main ways are `null` and `NaN`, and `NaN` is a more-sophisticated representation that's specific to Pandas. A `NaN` value has many useful properties, which you can read about in **Working with missing data (https://pandas.pydata.org/pandas-docs/stable/user_guide/missing_data.html)** in the official Pandas documentation.

The most important benefit of the `NaN` data type is that we can use it with particular Pandas functions. With these functions, we can identify the missing values and then either remove or replace them.

Let's start with an example CSV file that has a missing value, as the following image shows:

```
items_csv = Path("items.csv")
item_df = pd.read_csv(items_csv)
item_df
```

	Item Number	Price	Quantity
0	2468	2.99	250
1	1357	NaN	300
2	9753	3.47	350

In the preceding image, notice that the DataFrame represents the missing value with the "NaN" text. For small datasets like this one, we can identify a missing value by the "NaN" text in the DataFrame output. But for large datasets, we need tools to identify missing data so that we can determine how much exists.

Determine How Much Data Is Missing

One of the best tools for finding missing, or **null**, values is the `isnull` function. This function returns `True` if it finds a `NaN` value in the DataFrame and `False` otherwise.

Let's again consider the DataFrame from before. The following image shows that the cell in the second row and second column of the DataFrame contains a `NaN` value:

```
items_csv = Path("items.csv")
item_df = pd.read_csv(items_csv)
item_df
```

	Item Number	Price	Quantity
0	2468	2.99	250
1	1357	NaN	300
2	9753	3.47	350

We run the `isnull` function on the DataFrame, as the following code shows:

```
item_df.isnull()
```

Running the preceding code produces `True` in the cell that had the `NaN` value and `False` in all the other cells, as the following image shows:

```
item_df.isnull()
```

	Item Number	Price	Quantity
0	False	False	False
1	False	True	False
2	False	False	False

You might be wondering why `True` and `False` values prove more useful than the `NaN` values in the original data. Well, after we convert each `NaN` value to `True`, we can then add up all the `True` values in each column. This results in a count of the `NaN` values that each column had. For this, we use the following code:

```
item_df.isnull().sum()
```

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

```
item_df.isnull().sum()
```

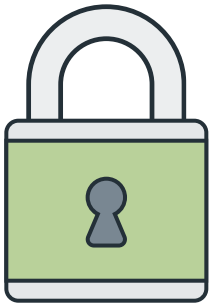
```
Item Number     0
Price           1
Quantity        0
dtype: int64
```

In the preceding image, notice that the number 1 appears for the “Price” column and that the number 0 appears for the other columns. This output means that the “Price” column has one missing, or `NaN`, value and that the “Item Number” and “Quantity” columns have none.

With a count of the missing data values in each column, we’ve now determined how much data is missing. We can then use this information to decide how to handle the missing data.

SHOW PRO TIP

Before moving on, check your knowledge in the following assessment:



Access Denied

You don't have access to view this resource.

By identifying the missing data and determining how much data is missing, we took an important first step in the data preparation process. Now, we need to decide how to handle the missing data.

Decide How to Handle the Missing Data

We can handle missing data values in one of two ways: by removing, or **dropping**, them or by replacing, or **filling**, them.

If we have a large dataset with just a few missing values, we might want to drop the rows that contain the missing values. Dropping a small number of rows from a large dataset shouldn't change the overall structure of the data. In this case, the results of the analysis should remain the same.

But if we have a smaller dataset or many `NaN` values, we don't want to drop the rows. Doing so might drastically change the dataset—and thus the resulting analysis. Instead, we want to fill as many of the `NaN` values as possible with a logical alternative value.

ON THE JOB

If you remove missing values, it's important to be aware of how much data you'll need to perform your analysis. That is, you want to make sure not to delete too much data if doing so might affect your calculations or overall analysis. Statisticians have set guidelines, but most analysts build an intuition for this type of decision making. If you have lots of missing data, you'll probably need to go back to the data collection phase to correct the issue there.

Next, we'll learn how to drop missing data, and then we'll learn how to fill missing data.

Drop the Missing Data

To drop the rows of data that have missing values, we use the `dropna` function. Let's try this on an example dataset.

NOTE

In this example, we'll manually insert a `NaN` value in the data by using the `np.nan` data type from the NumPy library. In the real world, Pandas creates `NaN` values when it reads a CSV file that contains missing values.

The following code imports the Pandas and NumPy libraries and then creates a `daily_returns` DataFrame containing Apple and Google stock data:

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

```
daily_returns = pd.DataFrame({
    "AAPL": [0.5, np.nan, 0.62],
    "GOOG": [0.45, 0.63, 0.55]
})
```

In the preceding code, notice that the DataFrame has three rows, which the code has in brackets and separates by commas.

Running the preceding code results in the output that the following image shows:

```
import pandas as pd
import numpy as np

daily_returns = pd.DataFrame({
    "AAPL": [0.5, np.nan,0.62],
    "GOOG": [0.45, 0.63, 0.55]
})

daily_returns
```

	AAPL	GOOG
0	0.50	0.45
1	NaN	0.63
2	0.62	0.55

In the preceding image, notice that one of the values for Apple is denoted by the "NaN" text, which means that it's missing.

We now call the `dropna` function on the `daily_returns` DataFrame, as the following code shows:

```
daily_returns.dropna()
```

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

```
daily_returns.dropna()
```

	AAPL	GOOG
0	0.50	0.45
2	0.62	0.55

Did you notice what happened? In the preceding image, the `daily_returns` DataFrame appears, but the `NaN` value is gone along with its whole row of data. Now, only two rows of data remain. As you can observe, the `dropna` function offers a quick and powerful way to handle missing data.

Still, we can't ignore the fact that using `dropna` just threw away one third of our data. This is fine for an example. But, it's hard to imagine a scenario where eliminating that much data wouldn't skew some aspect of our analysis. This especially applies in a professional setting, where every data point is thought to have value. So, we have another technique for handling missing data—namely, filling it.

Fill the Missing Data

When dropping the missing data isn't an option, data analysts sometimes fill the missing data with specific values that they can account for in the analysis. The three most-common replacement values are the following:

- The string "Unknown"
- The number 0
- The value that the `mean` function returns

While pros and cons exist for each of those values, you can choose the best replacement value for your particular problem. Maybe, you want to flag `NaN` values with "Unknown". Or, you might want to replace a missing number with either a zero or the average value of that column. This choice is up to you as the analyst. And regardless of your choice, your friend Pandas will make the replacement easy with a function named `fillna`.

To use the `fillna` function, you just supply the function with a replacement value for the `NaN` values—and Pandas handles the rest. The following code shows all three options:

```
# Replace the NaNs with "Unknown"
daily_returns = daily_returns.fillna("Unknown")

# Replace the NaNs with 0
daily_returns = daily_returns.fillna(0)

# Replace the NaNs with the mean of the column
daily_returns = daily_returns.fillna(daily_returns.mean())
```

In the preceding code, the first two options—using "Unknown" and using 0—are straightforward. But, let's more closely examine the third option—using the `mean` function. We calculate the mean, or average, of each column in the DataFrame by using `daily_returns.mean()`. Then with the `fillna` function, we replace the `NaN` values in each column with the mean value of that column.

If we apply this function to the DataFrame of Apple and Google stock by using the preceding code, we'll get the result that the following image shows:

```
daily_returns
```

	AAPL	GOOG
0	0.50	0.45
1	NaN	0.63
2	0.62	0.55

```
daily_returns = daily_returns.fillna(daily_returns.mean())
daily_returns
```

	AAPL	GOOG
0	0.50	0.45
1	0.56	0.63
2	0.62	0.55

The `fillna` function replaced the `NaN` value in the "AAPL" column with the mean value of 0.56, which is the mean of 0.50 and 0.62. So, you can think of the `mean` function as a shortcut for filling in the missing values with the mean value of each column.