4.7.8 Format Columns

Maria is impressed by your summary DataFrame but wants to add some formatting to make the DataFrame look more professional. You'll need to format the budget to two decimal places; format the grade averages to one decimal place and grade percentages to the nearest whole number percent; and add a thousands separator for numbers greater than 1,000.

To clean up the `district_summary_df` DataFrame, we will format dollar amounts to two decimal places, and format the grade averages to one decimal place and percentages to the nearest whole number percent.

NOTE

> Grade averages are usually formatted to one decimal place because averages taken to the hundredths (range: 0.01–0.09) or lower do not impact an average as much as tenths of a grade point, (range: 0.1–0.9). When tenths of a grade point equal or are greater than 0.5, you should usually round up to the next whole number.

> Grade percentages are formatted to the whole percentage because the tenths of a percent is equivalent to thousandths of a value, or grade. Like averages, thousandths of a grade don't have an impact on the overall percentage because they are so small.

This type of formatting can be done with the built-in Pandas `map()` function. The `map()` **function** is used for substituting each value in a Series with another value. Where the new value is generated from a function, a dictionary, or a Series.

In this module, as well as future modules, we'll be using and writing functions. So before we use the `map()` function for formatting, let's gain a better understanding of how functions work.

Overview of Functions

When writing algorithms that perform a repetitive task, like getting grade averages over and over again, it's a best practice to convert the algorithm to a function. A **function** is a smaller, more manageable piece of code.

Long algorithms can increase the chance of syntax errors and repeated variable assignments, which can lead to programming headaches—especially if your code is more than 200 lines. Breaking your code into manageable pieces has many advantages. You can:

- Debug and fix errors faster.
- Make your code more readable.
- Reuse code by importing functions into other algorithms.
- Speed up programming development. Multiple team members can work on separate functions of a complex algorithm to speed up the development of the project.

A hallmark of professional programmers is that they can take long blocks of code and write smaller pieces of that code as functions.

There are four basic parts to a function:

1. The name, which is what we call the function
2. The parameters, which are values we send to the function
3. The code block, which are the statements under the function that perform the task
4. The return value, which is what the function gives back, or "returns," to use when the task is complete

---

Naming Functions

The name of a function can be almost anything you want, but it should be descriptive enough so that anyone reading your code can reasonably guess what the function does. It's generally thought of as a best coding practice to write the name of the function as a verb, as functions perform actions. However, Python requires that you follow the same rules that you follow when naming variables.

REWIND

Variable names in Python can be any length and can consist of the following:

- Uppercase letters (A–Z)
- Lowercase letters (a–z)
- Digits (0–9)
- Underscores (_). Also, the first character of a variable name cannot be a digit.

When it comes to naming functions, there are two additional rules and two caveats to the variable naming rules.

- The name cannot be a Python keyword.
- The name cannot contain spaces.
- The first character of the name must be an uppercase or lowercase letter or an underscore.
- After the first character, you can use uppercase and lowercase letters, digits 0 through 9, or an underscore.

Parts of a Function

Let's review the parts of a function. We'll use the simple function below as an example.

```
# Define the function "say_hello" so it prints "Hello!" when called.
def say_hello():
    print("Hello!")
```

Let's go over what is happening in this function.

- The `def` keyword, which is short for "define," and is used to create the function.
- The function name, `say_hello(),` is always written after `def`.
- There is a colon after the function name, which tells Python that a code block or statement will follow. This is similar to conditional and repetition statements. All code within the function should be indented.
- The indented statement that follows is `print("Hello!")`.

Calling a Function

Now we'll practice calling a function. Create a new Jupyter Notebook file and name it `function.ipynb`. Copy the following code for the `say_hello()` function into the file.

```
# Define the function "say_hello" so it prints "Hello!" when called.
def say_hello():
    print("Hello!")
```

When you run the code, nothing appears to happen because we don't see an output error. To get the function to print "Hello!" in the output cell, we have to **call** the function.

To call the `say_hello()` function, type `say_hello()` in a new cell and run the cell.

```
# Call the function.
say_hello()
```

This will print "Hello!" to the output cell:

```
# Define the function "say_hello" so it prints "Hello!" when called.
def say_hello():
    print("Hello!")

# Call the function.
say_hello()
```

```
Hello!
```

Now we'll build on our knowledge of functions and add a parameter inside the parentheses of a new function, `say_something()`. A **parameter** is a value that we send to the function.

Add the following code to a new cell and run the cell.

```
# Define the function "say_something" so it prints whatever is passed as the
def say_something(something):
    print(something)
```

Inside the parentheses, we added the variable `something`. When this function gets called, it will print whatever we have added inside the parentheses of the function. This doesn't have to be the word "something"; it can be whatever you want it to be.

In the next cell, add "Hello World" inside the parentheses of the function, `say_something()`.

```
# Call the function.
say_something("Hello World")
```

When we call the function, i.e., run the cell, "Hello World" is printed to the output.

Like the string "Hello World," we can also pass a variable inside the function and have that variable printed to the output. In the following code, Jane introduces herself with the string "Hi, my name is Jane. I'm learning Python!" We can print her introduction by adding the variable `Jane_says` inside the `say_something()` function:

```
Jane_says = "Hi, my name is Jane. I'm learning Python!"
say_something(Jane_says)
```

When we run this cell, we get "Hi, my name is Jane. I'm learning Python!" in the output.

```
Jane_says = "Hi, my name is Jane. I'm learning Python!"
say_something(Jane_says)
```

```
Hi, my name is Jane. I'm learning Python!
```

Writing Functions for the School District Data

We've reviewed some basics about functions. Now how do we apply functions to our data analysis? For example, consider if you were in charge of calculating passing percentages for not just the school district we've been focusing on, but many school districts. It wouldn't be efficient to perform the steps of the calculation for each school district. A better method would be to create functions for this calculation and pass variables into the function.

Let's say you need to write a function to get the percentage of students who passed math when we know the number of students that passed math, `pass_math_count`, and the total number of students for a school district, `student_count`. We can define this function as `passing_math_percent` and pass the two values inside the function. Here's what this might look like:

```
# Define a function that calculates the percentage of students that passed b
# math and reading and returns the passing percentage when the function is c

def passing_math_percent(pass_math_count, student_count):
    return pass_math_count / float(student_count) * 100
```

Let's go over what is happening in this function.

1. We added two values to the `passing_math_percent` function: `pass_math_count` and `student_count`.
2. We added `return` in front of the calculation for the passing percentage.

The `return` statement has a unique purpose. It causes the function to end and literally "returns" what is in front of the statement back to the caller, which is the function. Let's run through an example of what happens to better illustrate the process.

In a new cell, assign the `passing_math_count` and the `total_student_count` variables 29,730 and 39,170, respectively. When we run this cell, that nothing will happen until we call the function.

```
passing_math_count = 29370
total_student_count = 39170
```

Now we can call the function in a new cell with the two values, like this:

```
# Call the function.
passing_math_percent(passing_math_count, total_student_count)
```

When we run this cell, the calculated percentage is printed to the output window.

```
# Call the function.
passing_math_percent(passing_math_count,total_student_count)
```

```
74.9808526933878
```

IMPORTANT

> If you don't add the number of parameters that are assigned to the function when calling the function, you'll get a `TypeError:`. This means that you need to add one or more of the parameters when calling the function.

Now that you have a good handle on how to write and use functions, let's apply this knowledge to the `map()` function, which we'll use to format our data.

The Pandas map() Function

The `map()` function is used for substituting each value in a Series with another value. Where the new value is generated from a function, a dictionary, or a Series. Let's look at an example of the `map()` function.

First, in `functions.ipynb`, add and run the following code:

```
# A list of my grades.
my_grades = ['B', 'C', 'B' , 'D']
```

Next, we'll import Pandas and convert the list to a Series.

```
# Import pandas.
import pandas as pd
# Convert the my_grades to a Series
my_grades = pd.Series(my_grades)
my_grades
```

When we execute this code, we get the following Series.

```
0    B
1    C
2    B
3    D
dtype: object
```

Using the `map()` function we are going to change B to A, C to B, and D to C. To change each string value in this Series to another value using the following format.

```
map("current_value_1" : "new_value_1",  "current_value_2" : "new_value_2", e
```

Using the format above we can write the following in a new cell.

```
# Change the grades by one letter grade.
my_grades.map({'B': 'A', 'C': 'B', 'D': 'C'})
```

When we run this cell, the output shows that all "B" grades were changed to "A", "C" was changed to "B", and "D" was changed to "C".

```
0    A
1    B
2    A
3    C
dtype: object
```

Since we had two grades that were a "B", we didn't need to map the second "B" to an "A". This change was handled with the first substitution in the `map()` function.

Now, let's look at another Pandas function. The `format()` function.

The Python format() Function

The Python `format()` **function** is used to format a value to a specific format, such as one or two decimal places, or adding a thousands separator. This function follows this syntax:

```
"{value:format specification}".format(value)
```

The "value" within the curly braces represents the value that we want to format. Usually, the value in the `format(value)` is a variable. After the colon, the `format specification` will designate how the value should be formatted.

Let's look at an example of the `format()` function.

In this example, we are going to iterate through a list of grades and format each grade to the nearest whole number percent. Copy the following code, add to a new cell in the `functions.ipynb` file, and run it.

```
# Using the format() function.
my_grades = [92.34, 84.56, 86.78, 98.32]

for grade in my_grades:
    print("{:.0f}".format(grade))
```

In this code, there is a list of grades formatted to two decimal places. To format these grades to the nearest whole number percent, we will do the following:

- Iterate through the grades.
- Pass the `grade` variable inside the `format()` function.
- Specify the format for the `grade` variable we would like by using `{:.0f}`.
    - In this format, the `grade` variable is referenced in front of the colon, so there is no need to add the `grade` variable.
    - After the colon, the `.0f` means to format the grade with no decimal place, where the "period" is for the decimal place, the "0" is for "no" decimal place, and the "f" means floating-point decimal. If we wanted to format to one decimal place, we would use 1 instead of 0, and 2 for two decimal places, and so on.

When we run the example code, we get the grades formatted to the nearest whole number percent:

```
for grade in my_grades:
    print("{:.0f}".format(grade))
```

```
92
85
87
98
```

NOTE

> For more information about formatting, read the following Python documentation:
>
> - **Format Examples (https://docs.python.org/3.4/library/string.html#format-examples)**
> - **Format Specification (https://docs.python.org/3.4/library/string.html#format-specification-mini-language)**

Chaining the map() and format() Functions

We also apply the `format()` function to a Series or column in a DataFrame by chaining the `map()` and `format()` functions.

First let's convert the numerical grades array to a Series with the following code:

```
# Convert the numerical grades to a Series.
my_grades = pd.Series([92.34, 84.56, 86.78, 78.32])
my_grades
```

Next, we can format the numerical grades to the nearest whole number percent as shown in the output after running the following line of code:

```
# Format the grades to the nearest whole number percent.
my_grades.map("{:.0f}".format)
```

```
0    92
1    85
2    87
3    78
dtype: object
```

Using what we just learned, we can format the dollar amounts in the `district_summary_df` columns to two decimal places with a U.S. dollar sign, and numbers larger than 999 with a thousands separator.

Let's apply this formatting technique to format the Total Students column with a thousands separator using the basic syntax:

`df["column"] = df["column"].map("{:,}".format)`

Now let's replace `df["column"]` with `district_summary_df["Total Students"]`, so our code looks like the following. Add this code to you`PyCitySchools.ipynb` file below the output of your `district_summary_df` DataFrame, and run the cell.

```
# Format the "Total Students" to have the comma for a thousands separator.
district_summary_df["Total Students"] = district_summary_df["Total Students"
```

```
district_summary_df["Total Students"]
```

When we execute this code, we get the following output:

```
1  district_summary_df["Total Students"]
```

```
0     39,170
Name: Total Students, dtype: object
```

One benefit of using the `format()` function is that we can add other format specifications, like a U.S. dollar sign or other characters. For example, we will format the Total Budget column in the `district_summary_df` DataFrame with a U.S. dollar sign, and then format the numbers with a thousands separator and to two decimal places using the following syntax: `"${:,.2f}".format`.

Add the following code to a new cell in our `PyCitySchools.ipynb` file and run the cell.

```
# Format "Total Budget" to have the comma for a thousands separator, a decim

district_summary_df["Total Budget"] = district_summary_df["Total Budget"].ma

district_summary_df["Total Budget"]
```

When we execute this code, we get the following output:

We need to format the remaining columns accordingly:

- The "Average Reading Score" column will be formatted to one decimal place.
- The "Average Math Score" column will be formatted to one decimal place.
- The "% Passing Reading" column will be formatted to the nearest whole number percentage.
- The "% Passing Math" column will be formatted to the nearest whole number percentage
- The "% Overall Passing" column will be formatted to the nearest whole number percentage.

Here's what the code for these columns will look like:

```
# Format the columns.
district_summary_df["Average Math Score"] = district_summary_df["Average Mat

district_summary_df["Average Reading Score"] = district_summary_df["Average

district_summary_df["% Passing Math"] = district_summary_df["% Passing Math"

district_summary_df["% Passing Reading"] = district_summary_df["% Passing Re

district_summary_df["% Overall Passing"] = district_summary_df["% Overall Pa
```

When you run the code, the `district_summary_df` DataFrame should look like this:

NOTE

> If the columns in your `district_summary_df` DataFrame are not in the order as shown, then you will have to put them in the correct order.