5

Apply Functions

Learning about `.apply()` is fundamental in the data cleaning process. It also encapsulates key concepts in programming, mainly writing functions. The `.apply()` method takes a function and applies it (i.e., runs it) across each row or column of a `DataFrame` without having you write the code for each element separately.

If you've programmed before, then the concept of an apply should be familiar. It is similar to writing a `for` loop across each row or column and calling the function, or making a `map()` call to a function. In general, this is the preferred way to apply functions across dataframes, because it typically is much faster than writing a `for` loop in Python.

If you haven't programmed before, then prepare to see how we can easily incorporate custom calculations that can be easily repeated across our data.

Learning Objectives

The concept map for this chapter can be found in **Figure A.1**.

- Create and use functions
- Use the `.apply()` method to iteratively perform a calculation across `Series` and `DataFrames`
- Identify what parts of a `Series` and `DataFrame` are passed into `.apply()`
- Create vectorized functions using Python decorators

Note About This Chapter

This chapter was also moved up from a later chapter for the second edition. This is one of the few parts of the book that relies on a completely toy example to simplify what is going on. Later on, we will be able to build on the skills taught in this chapter.

5.1 Primer on Functions

Functions are core elements of using the `.apply()` method. There's a lot more information about functions in **Appendix O**, but here's a quick introduction.

Functions are a way to group and reuse Python code. If you are ever in a situation where you are copying/pasting code and changing a few parts of the code, then chances are, the copied code can be written into a function. To create a function, we need to define it (with the `def` keyword). The body of a function is indented.

The PEP8 Style Guide for Python Code says to use four spaces for an indentation. This book uses two spaces for an indentation because of horizontal space limitations, but I am a new convert to using tabs for indentation because it creates more accessible code and is friendlier for people using Braille readers.[1]

1. Tabs for accessibility: https://alexandersandberg.com/articles/default-to-tabs-instead-of-spaces-for-an-accessible-first-environment/

The basic function skeleton looks like this:

Click here to view code image

```
def my_function(): # define a new function called my_function
  # indentation for
  # function code
  pass # this statement is here to make a valid empty function
```

Since Pandas is used for data analysis, let's write some more "useful" functions:

- squares a given value
- takes two numbers and calculates their average

Click here to view code image

```
def my_sq(x):
  """Squares a given value
  """
  return x ** 2


def avg_2(x, y):
  """Calculates the average of 2 numbers
  """
  return (x + y) / 2
```

The text within the triple quotes `"""` is a “docstring.” It is the text that appears when you look up the help documentation about a function. You can such docstrings to create your own documentation for functions you write as well.

We’ve been using functions (and methods) throughout this book. If we want to use functions that we’ve created ourselves, we can call them just like functions we’ve loaded from a library.

```
my_calc_1 = my_sq(4)
print(my_calc_1)
```

```
16
```

```
my_calc_2 = avg_2(10, 20)
print(my_calc_2)
```

```
15.0
```

5.2 Apply (Basics)

Now that we know how to write functions, how do we use them in Pandas? When working with `DataFrame`s, it's more likely that you want to use a function across rows or columns of your data.

Here's a mock dataframe of two columns.

Click here to view code image

```
import pandas as pd

df = pd.DataFrame({"a": [10, 20, 30], "b": [20, 30, 40]})
print(df)
```

```
    a   b
0  10  20
1  20  30
2  30  40
```

We can `.apply()` our functions over a `Series` (i.e., an individual column or row).

For didactic purposes, let's use the function we wrote to square the `'a'` column. In this overly-simplified example, we could have directly squared the column.

```
print(df['a'] ** 2)
```

```
0    100
1    400
2    900
Name: a, dtype: int64
```

Of course, that would not allow us to use a function we wrote ourselves.

5.2.1 Apply Over a Series

In our example, if we subset a single column or row using a single pair of square brackets, `[ ]`, the `type()` of the object we get back is a Pandas `Series`.

Click here to view code image

```
# get the first column
print(type(df['a']))
```

```
<class 'pandas.core.series.Series'>
```

```
# get the first row
print(type(df.iloc[0]))
```

```
<class 'pandas.core.series.Series'>
```

The `Series` has a method called `.apply()`.[2] To use the `.apply()` method, we give it the function we want to use across each element in the `Series`.

---

[2]. `Series apply` documentation: https://pandas.pydata.org/docs/reference/api/pandas.Series.apply.html

For example, if we want to square each value in column `a`, we can do the following:

Click here to view code image

```
# apply our square function on the 'a' column
sq = df['a'].apply(my_sq)
print(sq)
```

```
0    100
1    400
```

```
2    900
Name: a, dtype: int64
```

Note

We do not need the round parentheses, `( )`, when we pass the function into `.apply()`, we pass in `my_sq` instead of `my_sq()`.

In more technical terms, this is called a “function factory,” where we are giving `.apply()` a reference to the function we want to use, but we are not invoking the function at this moment.

Let’s build on this example by writing a function that takes two parameters. The first parameter will be a value, and the second parameter will be the exponent to which we’ll raise the value. So far in our `my_sq()` function, we’ve “hard-coded” the exponent, `2`, to raise our value.

```
def my_exp(x, e):
  return x ** e
```

Now, if we want to use our function, we have to provide two parameters to it.

Click here to view code image

```
# pass in the exponent, 3
cubed = my_exp(2, 3)
print(cubed)
```

```
8
```

```
# if we don't pass in all the parameters
my_exp(2)
```

```
TypeError: my_exp() missing 1 required positional argument: 'e'
```

However, if we want to apply the function on our series, we will need to pass in the second parameter. To do this, we pass the second argument as a **keyword argument** into `.apply()`.

Click here to view code image

```
# the exponent, e, to 2
ex = df['a'].apply(my_exp, e=2)
print(ex)
```

```
0    100
1    400
2    900
Name: a, dtype: int64
```

```
# exponent, e, to 3
ex = df['a'].apply(my_exp, e=3)
print(ex)
```

```
0     1000
1     8000
2    27000
Name: a, dtype: int64
```

5.2.2 Apply Over a DataFrame

Now that we've seen how to apply functions over a one-dimensional `Series`, let's see how the syntax changes when we are working with `DataFrames`. Here is the example `DataFrame` from earlier:

Click here to view code image

```
df = pd.DataFrame({"a": [10, 20, 30], "b": [20, 30, 40]})
print(df)
```

```
    a   b
0  10  20
1  20  30
2  30  40
```

`DataFrame`s typically have at least two dimensions. Thus, when we apply a function over a dataframe, we first need to specify which axis to apply the function over–for example, column-by-column or row-by-row.

Let’s first write a function that takes a single value and prints out the given value. The function below does not have a `return` statement, All it is doing is displaying on the screen whatever we pass it.

```
def print_me(x):
  print(x)
```

Let’s `.apply()` this function on our dataframe, The syntax is similar to using the `.apply()` method on a `Series`, but this time we need to specify whether we want the function to be applied column-wise or row-wise.

If we want the function to work column-wise, we can pass the `axis=0` or `axis="index"` parameter into `.apply()`. If we want the function to work row-wise, we can pass the `axis=1` or `axis="columns"` parameter into `.apply()`.[3]

---

3. I find the “index” and “column” text specification for the `axis` parameter counter-intuitive, so I will typically specify using the `0`/`1` notation with a comment. In practice, you will al-

most never set `axis=1` or `axis="columns"` for performance reasons.

5.2.2.1 Column-Wise Operations

Use the `axis=0` parameter (the default value) in `.apply()` when working with functions in a column-wise manner (i.e., for each column).

```
df.apply(print_me, axis=0)
```

```
0    10
1    20
2    30
Name: a, dtype: int64
0    20
1    30
2    40
Name: b, dtype: int64
___________

     0
___________
 a   None
 b   None
___________
```

Compare this output to the following:

```
print(df['a'])
```

```
0    10
1    20
2    30
Name: a, dtype: int64
```

```
print(df['b'])
```

```
0    20
1    30
2    40
Name: b, dtype: int64
```

You can see that the outputs are exactly the same. When you apply a function across a `DataFrame` (in this case, column-wise with `axis=0`), the entire axis (e.g., column) is passed into the first argument of the function. To illustrate this further, let's write a function that calculates the mean (average) of three numbers (each column in our data set contains values).

```
def avg_3(x, y, z):
  return (x + y + z) / 3
```

If we try to apply this function across our columns, we get an error.

Click here to view code image

```
# will cause an error
print(df.apply(avg_3))
```

```
TypeError: avg_3() missing 2 required positional arguments: 'y' and 'z'
```

From the (last line of the) error message, you can see that the function takes three arguments (`x`, `y`, and `z`), but we failed to pass in the `y` and `z` (i.e., the second and third) arguments. Again, when we use `.apply()`, the **entire** column is passed into the **first** argument. For this function to work with the `.apply()` method, we will have to rewrite parts of it.

Click here to view code image

```
def avg_3_apply(col):
  """The avg_3 function but apply compatible
  by taking in all the values as the first argument
  and parsing out the values within the function
  """
  x = col[0]
  y = col[1]
  z = col[2]
  return (x + y + z) / 3
```

```
print(df.apply(avg_3_apply))
```

```
a    20.0
b    30.0
dtype: float64
```

Now that we've rewritten our function to take in all the column values, we get two values back after we apply (one for each column of our `DataFrame`) and each value represents the average of the three values.

5.2.2.2 Row-Wise Operations

Row-wise operations work just like column-wise operations. The part that differs is the axis we use. We will now use `axis=1` in the `.apply()` method. Instead of the entire column being passed into the first argument of the function, the **entire row** is used as the first argument.

Since our example dataframe has two columns and three rows, the `avg_3\apply()` function we just wrote will not work for row-wise operations.

Click here to view code image

```
# will cause an error
print(df.apply(avg_3_apply, axis=1))
```

```
IndexError: index 2 is out of bounds for axis 0 with size 2
```

The main issue here is the `'index out of bounds'`. We passed the row of data in as the first argument, but in our function we begin indexing out of range (i.e., we have only two values in each row, but we tried to get index `2`, which means the third element, and it does not exist). If we wanted to calculate our averages row-wise, we would have to write a new function to work with two values.

Click here to view code image

```
def avg_2_apply(row):
  """Taking the average of row value.
  Assuming that there are only 2 values in a row.
  """
  x = row[0]
  y = row[1]
  return (x + y) / 2

print(df.apply(avg_2_apply, axis=0))
```

```
a    15.0
b    25.0
dtype: float64
```

5.3 Vectorized Functions

When we use `.apply()`, we are able to make a function work on a column-by-column or row-by-row basis. In the previous section, Section 5.2, we had to rewrite our function when we wanted to apply it because the entire column or row was passed into the first parameter of the function. However, there might be times when it is not feasible to rewrite a function in this way. We can leverage the `.vectorize()` function and decorator to vectorize any function. Vectorizing your code can also lead to performance gains (Appendix V).

Here’s our toy dataframe:

Click here to view code image

```
df = pd.DataFrame({"a": [10, 20, 30], "b": [20, 30, 40]})
print(df)
```

```
    a   b
0  10  20
1  20  30
2  30  40
```

And here’s our average function, which we can apply on a row-by-row basis:

```
def avg_2(x, y):
  return (x + y) / 2
```

For a vectorized function, we'd like to be able to pass in a vector of values for `x` and a vector of values for `y`, and the results should be the average of the given `x` and `y` values in the same order. In other words, we want to be able to write `avg_2(df['a'], df['y'])` and get `[15, 25, 35]` as a result.

Click here to view code image

```
print(avg_2(df['a'], df['b']))
```

```
0    15.0
1    25.0
2    35.0
dtype: float64
```

This approach works because the actual calculations within our function are inherently vectorized. That is, if we add two numeric columns together, Pandas (and the NumPy library) will automatically perform element-wise addition. Likewise, when we divide by a scalar, it will "broadcast" the scalar, and divide each element by the scalar.

Let's change our function and perform a non-vectorizable calculation.

Click here to view code image

```
import numpy as np

def avg_2_mod(x, y):
   """Calculate the average, unless x is 20
   If the value is 20, return a missing value
   """
   if (x == 20):
     return(np.NaN)
   else:
     return (x + y) / 2
```

If we run this function, it will cause an error.

Click here to view code image

```
# will cause an error
print(avg_2_mod(df['a'], df['b']))
```

```
ValueError: The truth value of a Series is ambiguous. Use a.empty,
a.bool(), a.item(), a.any() or a.all().
```

However, if we give it individual numbers instead of a vector, it will work as expected.

```
print(avg_2_mod(10, 20))
```

```
15.0
```

```
print(avg_2_mod(20, 30))
```

```
nan
```

5.3.1 Vectorize with NumPy

We want to change our function so that when it is given a vector of values, it will perform the calculations in an element-wise manner. We can do this by using the `vectorize()` function from `numpy`. We pass `np.vectorize()` to the **function** we want to vectorize, to create a new function.

Click here to view code image

```
import numpy as np

# np.vectorize actually creates a new function
avg_2_mod_vec = np.vectorize(avg_2_mod)

# use the newly vectorized function
print(avg_2_mod_vec(df['a'], df['b']))
```

```
[15. nan 35.]
```

This method works well if you do not have the source code for an existing function. However, if you are writing your own function, you can use a Python decorator to automatically vectorize the function without having to create a new function. A decorator is a function that takes another function as input, and modifies how that function's output behaves.

Click here to view code image

```
# to use the vectorize decorator
# we use the @ symbol before our function definition
@np.vectorize
def v_avg_2_mod(x, y):
  """Calculate the average, unless x is 20
  Same as before, but we are using the vectorize decorator
  """
  if (x == 20):
    return(np.NaN)
  else:
    return (x + y) / 2

# we can then directly use the vectorized function
# without having to create a new function
print(v_avg_2_mod(df['a'], df['b']))
```

```
[15. nan 35.]
```

5.3.2 Vectorize with Numba

The `numba` library[4] is designed to optimize Python code, especially calculations on arrays performing mathematical calculations. Just like `numpy`, it also has a `vectorize` decorator.

4. numba : https://numba.pydata.org/

Click here to view code image

```
import numba

@numba.vectorize
def v_avg_2_numba(x, y):
  """Calculate the average, unless x is 20
  Using the numba decorator.
  """
  # we now have to add type information to our function
  if (int(x) == 20):
    return(np.NaN)
  else:
    return (x + y) / 2
```

The numba library is so optimized that it does not understand Pandas objects.

Click here to view code image

```
print(v_avg_2_numba(df['a'], df['b']))
```

```
ValueError: Cannot determine Numba type of
<class 'pandas.core.series.Series'>
```

We actually have to pass in the `numpy` array representation of our data using the `.values` attribute of our `Series` objects (Chapter R).

Click here to view code image

```
# passing in the numpy array
print(v_avg_2_numba(df['a'].values, df['b'].values))
```

```
[15. nan 35.]
```

5.4 Lambda Functions (Anonymous Functions)

Sometimes the function used in the `.apply()` method is simple enough that there is no need to create a separate function.

Let's look at our simple `DataFrame` example and our squaring function again.

Click here to view code image

```
df = pd.DataFrame({'a': [10, 20, 30],
                   'b': [20, 30, 40]})
print(df)
```

```
    a   b
0  10  20
```

```
1  20  30
2  30  40
```

```
def my_sq(x):
  return x ** 2

df['a_sq'] = df['a'].apply(my_sq)
print(df)
```

```
    a   b  a_sq
0  10  20   100
1  20  30   400
2  30  40   900
```

You can see that the actual function is a simple one-liner. Usually when this happens, people will opt to write the one-liner directly in the `apply` method. This method is called using **lambda functions**. We can perform the same operation as shown earlier in the following manner.

Click here to view code image

```
df['a_sq_lamb'] = df['a'].apply(lambda x: x ** 2)
print(df)
```

```
    a   b  a_sq  a_sq_lamb
0  10  20   100        100
1  20  30   400        400
2  30  40   900        900
```

To write the lambda function, we use the `lambda` keyword. Since apply functions will pass the entire axis as the first argument, our `lambda` function example takes only one parameter, `x`. The `x` in `lambda x` is analogous to the `x` in `def my_sq(x)`, each value in the `'a'` column will be individually passed into our `lambda` function. We can then write our function directly, without having to define it. The calculated result is automatically returned.

Although you can write complex multiple-line lambda functions, typically people will use the lambda function approach when small one-liner calculations are needed. The code can become hard to read if the lambda function tries to do too much at once.

Conclusion

This chapter covered an important concept – namely, creating functions that can be used on our data. Not all data cleaning steps or manipulations can be done using built-in functions. There will be many times when you will have to write your own custom functions to process and analyze data.

This chapter uses oversimplified examples to create and use functions, but that means we can go into more complex examples as we learn more about the `pandas` library.