Python Classes

Defining a Class

Classes are used to combine data and actions into a single unit. When you define a class, you define a blueprint for a data type.

When working with data, we actually don't define classes too often, but we use classes all the time. For example, we use instances of Pandas DataFrame class to work with data in Python.

One place where we create classes is when we use scikit-learn custom transformers. We create a class that inherits from `BaseEstimator` and `TransformerMixin` classes and implement `fit` and `transform` methods.

```
# A canonical example of a class in Python

class Book:
    '''This is a class that represents a book
    This string is called a docstring and is used to document the class'''

    def __init__(self, title, author, pages):
        '''This is the docstring for the __init__ method'''
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return f"{self.title} by {self.author}"

    def __len__(self):
        return self.pages
```

Creating an Instance

An instance is a specific object created from a particular class. Here's how you create an instance of a class. It is used to store the data and allow for easy access to the methods.

```
>>> book = Book("The Catcher in the Rye", "J.D. Salinger", 234)
>>> print(book)
```

The Catcher in the Rye by J.D. Salinger

Subclassing to Create sklearn Transformers

In order to create a custom transformer, we need to subclass the BaseEstimator and TransformerMixin classes from the sklearn.base module. The BaseEstimator class provides the get_params and set_params methods, while the TransformerMixin class provides the fit_transform method. You need to implement the .fit and .transform methods in your custom transformer class.

The .fit method should return self and the .transform method should return the transformed data. Because it subclasses the TransformerMixin class, the custom transformer can be used in a pipeline and also has other methods like .fit_transform.

```
from sklearn.base import BaseEstimator, TransformerMixin

class DoubleTransformer(BaseEstimator, TransformerMixin):
    def __init__(self, param1=True):
        self.param1 = param1

    def fit(self, X, y=None):
        # Fit logic here
        return self

    def transform(self, X):
        # Transform logic here
        return X * 2

import pandas as pd
df = pd.DataFrame({'A': [1, 2, 3, 4, 5]})
dt = DoubleTransformer()
dt.fit(df)

>>> print(dt.transform(df))
    A
0   2
1   4
2   6
3   8
4  10
```

Python Functions

A function is used to group a set of statements so they can be combined into a logical unit and be executed more than once. Functions are defined using the `def` keyword followed by the function name, a set of parentheses, and a colon. The statements that make up the function body are indented.

Defining a Function

When you use libraries like Pandas, you use functions that are defined in the library. You can also define your own functions. Here's how you define a function in Python.

- Put the keyword `def` in front of the function name.
- Put the function arguments inside the parentheses.
- Put a colon at the end of the function definition.
- Indent the code that makes up the function body.
- You can add an optional docstring to describe what the function does.

```
def my_function(param1, param2):
    '''This is a docstring that explains what the function
    does and how it should be used'''
    return param1 + param2
```

Calling a Function

Once you've defined a function, you can call it by using the function name followed by a set of parentheses. If the function takes arguments, you put the values for those arguments inside the parentheses.

```
>>> result = my_function(5, 3)
>>> print(result)
8
```

Lambdas

Lambdas a syntactic sugar for defining simple functions in a single line. They are also known as anonymous functions because they don't have a name. They are defined using the `lambda` keyword followed by a set of arguments, a colon, and an expression.

I use them all the time when working with Pandas DataFrames. If you want to bring your code up to the next level, you should learn how to use lambdas.

Defining a Lambda Function

To define a lambda function, you use the `lambda` keyword followed by a set of arguments, a colon, and an expression. Here's how you define a lambda function in Python.

- Put the keyword `lambda` in front of the arguments.
- Put a colon at the end of the arguments.
- Put the expression that makes up the function body after the colon. (No need to use `return` keyword)

```
>>> add = lambda x, y: x + y
>>> print(add(2, 3))  # Output: 5
5
```

Unpacking Arguments

Python allows us to use a * in a function definition to specify that the function can take a variable number of arguments. The variable will be a tuple containing all the arguments passed to the function.

You can also use a * in front of a sequence when you call a function to *unpack* the sequence into individual arguments.

Python also allows us to use a ** in a function definition to specify that the function can take a variable number of keyword arguments. The variable will be a dictionary containing all the keyword arguments passed to the function.

You can also use a ** in front of a dictionary when you call a function to *unpack* the dictionary into individual keyword arguments. We use this feature when working with Pandas DataFrames. I use it when passing a dictinoary of parameters to the `.assign` method.

Using *args

```
def func(*args):
    for arg in args:
        print(arg)
```

```
>>> func(1, 2, 3)
1
2
3
```

```
>>> args = (4, 5, 6)
>>> func(*args)
4
```

```
5
6
```

Using **kwargs

```
def func(**kwargs):
    for key, value in kwargs.items():
        print(f"{key} = {value}")


>>> func(a=1, b=2, c=3)
a = 1
b = 2
c = 3


>>> kwargs = {'a': 4, 'b': 5, 'c': 6}
>>> func(**kwargs)
a = 4
b = 5
c = 6
```

Using Lambdas in pandas .assign

You really need to master the `.assign` method in Pandas. It allows you to create new columns in a DataFrame by specifying the column name

- a Series
- a scalar value
- a function

This last one is important. You can use a normal function, but you can also use a lambda function. The function must take the current DataFrame as an argument and return a Series or scalar value. If you have a chain of operations, you can use a lambda function to create a new column based on the current state of the DataFrame rather than using the original data.

I generally name the argument `df_` to make it clear that it is a DataFrame. You can name it whatever you want, but it is a good practice to use a name that makes it clear what the argument is. I like `df_` because it is short and clear and I also only use that convention in the `.assign` method.

```
import pandas as pd
```

```
df = pd.DataFrame({'item': ['apple', 'banana', 'cherry'],
                   'price': [1, 2, 3]})


>>> print(df)
     item  price
0   apple      1
1  banana      2
2  cherry      3


>>> print(df
...       .assign(tax_total=lambda df_: df_['price'] * 1.05)
... )

     item  price  tax_total
0   apple      1       1.05
1  banana      2       2.10
2  cherry      3       3.15
```

Using Lambdas in .loc

Another place where you can use lambdas is in the `.loc` method. You can use a lambda function to filter rows based on the current state of the DataFrame. The lambda function must return a boolean Series. The `.loc` method will keep the rows where the Series is `True` and drop the rows where the Series is `False`.

```
>>> print(df
...       .assign(tax_total=lambda df_: df_['price'] * 1.05)
...       .loc[lambda df_: df_['tax_total'] > 2]
... )

     item  price  tax_total
1  banana      2       2.10
2  cherry      3       3.15
```

Plotting with pandas

It is imperative that you learn how to plot data in Python. The most common library for plotting data is Matplotlib, but Pandas has a built-in plotting library that makes it easy to create plots from DataFrames. I think it is easier to use the Pandas plotting library than the Matplotlib library. I also think that you should focus your efforts on learning four main plots:

- Line Plot
- Scatter Plot
- Histogram
- Bar Plot

Line Plot

A line plot is used to show data that changes over time. It is a good plot to use when you have a time series or when you want to show how a variable changes over time. You can create a line plot by calling the `.plot` method on a DataFrame directly. (Or by calling the `.plot.line` method.)

```
import pandas as pd
import matplotlib.pyplot as plt

fig, axs = plt.subplots(2, 2, figsize=(10, 10))

url = 'https://github.com/mattharrison/datasets/raw/master/data/vehicles.csv.zip'

raw_data = pd.read_csv(url, dtype_backend='pyarrow', engine='pyarrow')


(raw_data
 .query('make in ["Ford", "Chevrolet", "Toyota", "Honda"]')
 .groupby(['year', 'make'])
.city08
.mean()
.unstack()
.plot(ax=axs[0, 0])
)
```

Histogram

A histogram shows the distribution of a single numerical variable. It is a great way to see the spread of the data. We can inspect if the data appears to be normally distributed or if there are any outliers.

You can use the `.plot.hist` method to create a histogram of a single column in a DataFrame.

```
raw_data['city08'].plot.hist(ax=axs[0, 1], title='City MPG')
```

Bar Plot

A bar plot is used to show the relationship between a numerical variable and a categorical variable. You can use the `.plot.bar` method to create a bar plot in Pandas. You can also use the `.plot.barh` method to create a horizontal bar plot.

We generally use a bar plot to show the count of a categorical variable but it can be used to show values like weights in PCA or coefficients in a linear regression model.

```
raw_data['cylinders'].value_counts().plot.bar(ax=axs[1, 0], title='Cylinders')
```

Scatter Plot

A scatter plot is used to show the relationship between two numerical variables. It is a great way to see if there is a relationship between two variables. You can see if there is a linear relationship, a non-linear relationship, or no relationship at all.

You can use the `.plot.scatter` method to create a scatter plot in Pandas. You can also use the `c` parameter to color the points based on a third variable. This variable should be numerical or categorical with a small number of unique values.

```
raw_data.plot.scatter(x='displ', y='highway08',
    c='year', ax=axs[1, 1], title='Engine Displacement vs Highway MPG')

fig.savefig('vehicles.png', dpi=300)
```

Here are the four plots:

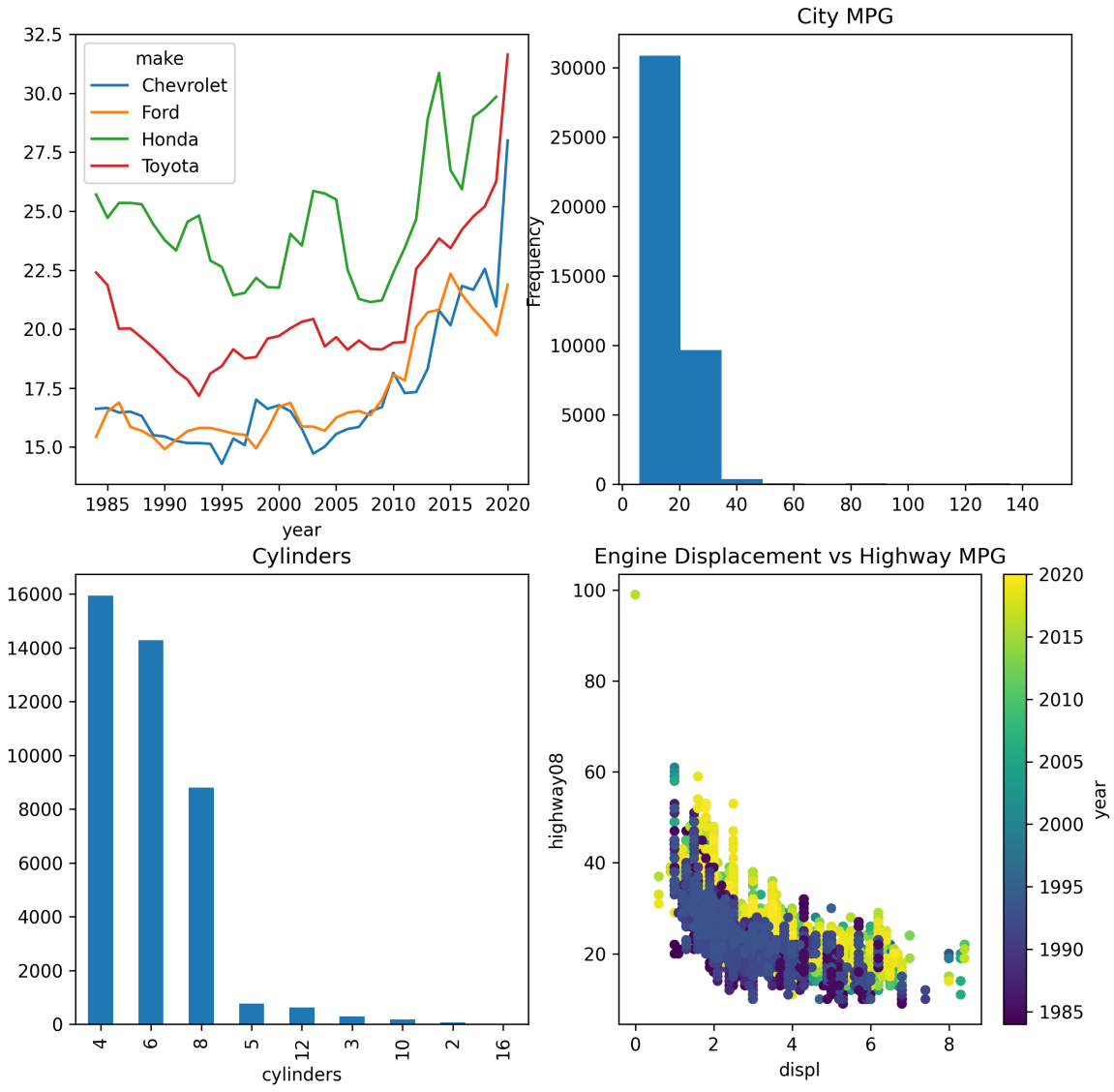

Figure 1: The four main plots you should focus on