Classes and objects in Python

Aksel Hiorth

University of Stavanger

Jan 16, 2024

Contents

1 Why classes?

A class can be a way for you to create a clean interface to your code. You have already used a lot of classes. The `DataFrame()` in Pandas is a class, and we access functions inside this class by using a dot (`.`). Classes also provides encapsulation: By wrapping parts of your code into classes, and particular realizations of classes (objects), you facilitate code re-use, and it can make your code easier to understand and work with, thus reducing the probability of introducing bugs which may be hard to track down.

To get started, there are really only a couple of things you need to know. First, all of your classes should include a special function called `__init__`, in which you declare the variables (attributes) you wish an instance / object of the class to keep track of.

Second, when setting, updating, or fetching attributes stored within the class, you should always use the prefix `self`, followed by a dot. Furthermore, the functions you define inside the class should have `self` as the first function argument (there are exceptions[1]), but we will not consider those here.

[1] `https://realpython.com/python3-object-oriented-programming/`

2 Example: A class for production data

Earlier in this course we have made functions to read data from an Excel file, and plot the data. This can also be done within a class, below is an example

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

class ProdData:
    """
    A class to extract production data from FactPages
    """
    def __init__(self):
        self.df_prod=pd.read_excel('../data/field_production_gross_monthly.xlsx')

    def get_data(self,field):
        """
        Extracts data for a specific field
        """
        df= self.df_prod[(self.df_prod['Field (Discovery)'] == field)]
        return df

    def plot(self,field,cols=[3,4,5,6,7]):
        """
        Plots the different columns in the DataFrame
        """
        df=self.get_data(field)
        xcol=df['Year']+df['Month']/12
        for col in cols:
            plt.plot(xcol,df.iloc[:,col],label=df.columns[col])
        plt.legend(loc='center', bbox_to_anchor=(0.5,-.3),
          ncol=3, fancybox=True, shadow=True)
        plt.title(field)
        plt.xlabel('Years')
        plt.ylabel('mill Sm$^3$')
        plt.grid()

ff=ProdData()
ff.plot('DRAUGEN')
```

The nice thing about the class is that it has a very nice interface, if the user wants to plot data from another field, it is just give the name of that field

```
ff.plot('EKOFISK')
```

3 Example: A class for a mathematical function

A mathematical function should be the perfect example of *when to use a function*, but it turns out that it can be quite convenient to use a class. Here we will consider the mathematical function in the equation below

$$f(x) = \sin(bx) \cdot e^{-ax^2}. \tag{1}$$

If we implement this function in a class, we can also add other functionalities to our function such as an ability to plot itself

```
class WavePacket:
    """
    A class representation of a wave packet-function.
    """
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def f(self, x):
        return np.sin(self.b*x)*np.exp(-self.a*x*x)

    def plot(self, x_min=-10, x_max=10, dx=0.01):
        """
        A simple plotting routine for plotting f(x) in some range.
        """
        x = np.arange(x_min, x_max, dx)
        y = self.f(x)
        fig = plt.figure()
        plt.plot(x, y)
        plt.grid()
```

Besides the initialization method and a function that calculates $f(x)$ from equation (1), the class includes a simple plotting routine. A major difference from before is the following: when our function $f(x)$ is defined inside a class, we do not have to pass around a and b as arguments to the function `f`. Instead, we simply access a and b from inside the class itself (using the `self`-prefix).

Below is an example of how to use the class:

```
# Create two WavePacket objects, having their own parameter values
WP1 = WavePacket(0.1, 2)  # a=0.1, b=2
WP2 = WavePacket(0.1, 10)  # a = 0.1, b=10

# Evaluate the two functions at a specific point
x = 1
print(WP1.f(x))
print(WP2.f(x))

# Plot the two functions
```

```
WP1.plot()
WP2.plot()
```

Although we had to write slightly more code, we hope you appreciate how easy this makes running parallel simulations with different parameters. Actually, Python provides a way for us to simplify even further, by defining the special `__call__`[2] method for the class:

```
class FancyWavePacket:
    """
    A slightly more fancy class representation of a wave packet-function.

    In this version, we define the dunder (double-underscore) method __call__,
    which lets us treat objects of the class as if they were real functions!
    """
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __call__(self, x):
        return np.sin(self.b*x)*np.exp(-self.a*x*x)
```

Compared to the first example of the class, observe that we have replaced the function `f` by `__call__` (with two underscores on both sides of "call"). This way, we we can write our code as if `FancyWavePacket` was a function:

```
WP1 = FancyWavePacket(0.1, 2)  # a=0.1, b=2
WP2 = FancyWavePacket(0.1, 10)  # a = 0.1, b=10

# Evaluate the two functions at a specific point
x = 1
print(WP1(x))  # If WP1 had been a function, the syntax would be the same here!
print(WP2(x))  # Again, we no longer have to type "WP2.f(x)", we can do "WP2(x)".
```

References

[2]https://www.realpythonproject.com/python-magic-oop-dunder/