Small Applications Development with Python
Session 8

Robert Palmere

2021

In this session we will take a look at how to write a small application for linear regression in Python. We will first generate a template of the class to be generated for this purpose. This template will serve as a way of organizing the application before we begin writing code. For larger programs, it is often best to write a document outlining the design of the application including the purpose of the program, the relation of classes and functions to one another, and how the program is used among other considerations. The annotated class outline is shown below.

```
class Application(object):
    def __init__(self):
        pass

    def __str__(self):
        '''Returns a report for print()'''
        pass

    @staticmethod
    def register():
        '''Functional decorator to capture which analysis function
            was most recently called'''
        pass

    @property
    def all_ok():
        '''Check if all instance attributes are set properly'''
        pass

    def load(self):
        '''Load the data from a given .dat or .txt file'''
        pass

    def select(self):
        '''Select the x / y columns for analysis'''
```

```
        pass

    def report(self):
        '''Generate a report for the most recently performed
            analysis'''
        pass

    def save(self):
        '''Save a report for the most recently performed analysis'''
        pass

    def linear_regression(self):
        '''Perform linear regression on the loaded data set'''
        pass

    def multi_linear_regression(self):
        '''Perform linear regression on all of the data set'''
        pass

    def plot(self):
        '''Plot the data from the most recently performed analysis'''
        pass
```

The purpose of each function, belonging to our *Application* class, is described in the documentation string under each function prototype. How each function relates to the class will become clear as we add function definitions and instance attributes. Let's first start with the function to load a file containing data to be linearly regressed.

```
import numpy as np

def load(f):
    data_ = []
    try:
        if f[-4:] == '.txt' or f[-4:] == '.dat':
            with open(f) as fp:
                lines = fp.readlines()
                for line in lines:
                    line = [float(i) for i in line.strip() if ' ' not
                        in i]
                    data_.append(line)
        except:
        raise ValueError('Loaded file must have a .txt or .dat file
            extension.')

        return np.asarray(data_)

test_data = load('test.txt')
```

```
print(test_data)
```

The *load()* function expects either a '.txt' or '.dat' extension for input data. A 2D NumPy array is returned of this data if it was accessible. The application user will need to be able to select which data to perform linear regression on. Since we have not yet set up our graphical user interface (GUI) we will write a function which takes in the data returned by *load()* and prompts the user to select x and y data columns of the array using *input()*.

```
def select(data):
   print('--Available Data--\n')
   print(data)
   print('\n')

   selected = []
   for option in ('x', 'y'):
      idx = input(f'Select the {option} column: ') # cin >> var
      idx = int(idx)
      if not isinstance(idx, int):
           raise ValueError(f'{option} selection must be an
               integer.')
      selected.append(data[:, idx])

   return selected[0], selected[1]

x, y = select(test_data)
```

Since the focus of this session is on application development and organization, we will write the functions for linear regression using *scipy.stats*. Separate functions will be written for simple linear regression and multiple linear regression to be selected by the user.

```
from scipy.stats import linregress

def linear_regression(x, y):
   result = linregress(x, y)
   y_ = result.intercept + result.slope*x
   return y_

def multi_linear_regression(x, data):
   rys_ = []
   ys_ = []
   for col_idx in range(data.shape[1]):
      if not np.array_equal(x, data[:, col_idx]): # Get every other
          column aside from selected x column
           result = linregress(x, data[:, col_idx])
```

```
            calculated_y = result.intercept + result.slope*x
            rys_.append(calculated_y)
            ys_.append(data[:, col_idx])
    return np.asarray(ys_), rys_
```

Lastly, *matplotlib* will be used to display linear regression results to the user.

```
import matplotlib.pyplot as plt

def plot(x, y, ry):
   if len(y.shape) > 1:
      fig, ax = plt.subplots(len(y), 1)
      for n, i in enumerate(y):
            ax[n].scatter(x, i, color='b')
            ax[n].plot(x, ry[n], 'r')
      plt.tight_layout()
   else:
      fig, ax = plt.subplots()
      ax.scatter(x, y, color='b')
      ax.plot(x, ry, 'r')
      plt.tight_layout()
```

Now that we have the primary functions of our application written, we can move on to integrating them into our *Application* class. Functions already written will be modified as part of the class.

```
from functools import wraps
from scipy.stats import linregress
import numpy as np
import matplotlib.pyplot as plt

class Application(object):

   def __init__(self):
      self.data = None
      self.registered = None
      self.x, self.y = (None, None)
      self.selected = False
         self.ry = None
      self.rvalue = None

   def __str__(self):
      '''Returns a report for print()'''
      if self.all_ok:
            return self.report()
```

```
def register(func):
   '''Functional decorator to capture which analysis function
       was most recently called'''
   @wraps(func)
   def wrapped(inst, *args, **kwargs):
        inst.registered = func.__name__
        return func(inst, *args, **kwargs)
   return wrapped

@property
def all_ok(self):
   '''Check if instance attributes are set properly before
       permitting analysis'''
   conditions = np.asarray([self.data.all(), self.selected ==
       True, self.registered != None])
   return conditions.all()

@register
def load(self, f):
   ...
   return self.data

def select(self):
   '''Select the x / y columns for analysis'''
   ...
   self.x = selected[0]
   self.y = selected[1]
   self.selected = True

def report(self):
   '''Generate a report for the most recently performed
       analysis'''
   if self.all_ok:
      ...

def save(self):
   '''Save a report for the most recently performed analysis'''
   with open('saved.txt', 'w') as fp:
        fp.write(self.report())

...
```

Instance attributes are written to capture the output of methods of the class instance. If we print the class instance the *str* special method is called to return a report of values from linear regression. A function decorator called *register* is also written so that the latest analysis function called by the user is known to the class as *self.registered* which contains the name of the function last called if the function is decorated with *@register*. To ensure that instance attributes are

set correctly before analysis, the *all_ok()* function checks if *self.data* contains numeric values, that the user selected data, and that a function was called. The *all_ok()* method is decorated with the native *@property* decorator so that we can access its return value as a class attribute in the scope of our other function definitions. The *@property* decorator is not necessary here, but changes how we access this method.

```
# @property decorated
def some_function(self):
   if self.all_ok:
      ...

# Not decorated with @property
def some_function(self):
   if self.all_ok():
      ...
```

We then proceed with testing our class by generating a class instance called *Analyzer*. The final step will be to implement a GUI for this application using Tkinter. We will not review all features available from Tkinter, but will generate a basic window and buttons to perform the analysis. To generate a Tkinter window we only need to access the *Tk()* class after importing and running the *mainloop()* function.

```
import tkinter as tk

root = tk.Tk()
root.mainloop()
```

We will write a small Tkinter window containing buttons for the various functions we have included as part of the *Applications* class. The functions requiring user input will have to change *input()* to use Tkinter related textual inputs instead. First we write the GUI class which inherits from Tkinter.

```
class GUI(tk.Tk):
   def __init__(self):
      super().__init__()
      self.geometry = ('300x200')
      self.title('Regression Analysis')
      tk.Button(self, text='Load', bg='white',
          fg='black').pack(expand=True)
      tk.Button(self, text='Save', bg='white',
          fg='black').pack(expand=True)
      tk.Button(self, text='Linear Regression', bg='white',
```

```
            fg='black').pack(expand=True)
        tk.Button(self, text='Multiple Linear Regression',
            bg='white', fg='black').pack(expand=True)
        tk.Button(self, text='Plot Result', bg='white',
            fg='black').pack(expand=True)

root = GUI()
root.mainloop()
```

To add functionality to each of the buttons we generated in the GUI class using Tkinter, we only need to inherit the *Application* class and set each of the lambda functions as returning the output from their associated methods for the 'command' keyword of each button.

```
class GUI(tk.Tk, Application):
   def __init__(self):
      super().__init__()
      self.geometry = ('300x200')
      self.title('Regression Analysis')

      tk.Button(self, text='Load', bg='white', fg='black',
          command=lambda: self.load('test.txt')).pack(expand=True)
      tk.Button(self, text='Select', bg='white', fg='black',
          command=lambda: self.select()).pack(expand=True)
      tk.Button(self, text='Save', bg='white', fg='black',
          command=lambda: self.save()).pack(expand=True)
      tk.Button(self, text='Linear Regression', bg='white',
          fg='black', command=lambda:
          self.linear_regression()).pack(expand=True)
      tk.Button(self, text='Multiple Linear Regression',
          bg='white', fg='black', command=lambda:
          self.multi_linear_regression()).pack(expand=True)
      tk.Button(self, text='Plot Result', bg='white', fg='black',
          command=lambda: self.plot()).pack(expand=True)

root = GUI()
root.mainloop()
```

This concludes basic development of a small application using Python. There is a lot more that can be done using Tkinter. The hope is that this serves as a review for how we can implement the topics were previously learned about in the context of an application.