Homework 2

(Due: March 10, 2025)

The purposes of this homework are to give you experience in working with Pandas and Geopandas, and Python's support for working with classes and objects. Submit to gradescope a pdf writeup containing the problem description for each problem, your Python code, and approprite textual and graphical output.

Question 1: 10 points. A laboratory experiment is conducted on 1030 specimens to determine the compressive strength (MPa) of concrete as a function of age (days) and various ingredients (e.g., cement, blast furnace slag, fly ash, water, superplasticizer, coarse aggregate, and fine aggregate).

The experimental data (see python-code.d/data/materials/concrete-strength-data.csv) comprises eight (quantitative) input parameters:

Parameter	Description
Cement	kg in a m3 mixture.
Blast Furnace Slag	kg in a m3 mixture.
Fly Ash	kg in a m3 mixture.
Water	kg in a m3 mixture.
Superplasticizer	kg in a m3 mixture.
Coarse Aggregate	kg in a m3 mixture.
Fine Aggregate	kg in a m3 mixture.
Age	day (1 -- 365).

and one output:

Parameter	Description
Concrete strength	Concrete compressive strength (MPa).

What to do? Write a Python program that will:

1. Read the experimental test results from a file `concrete-strength-data.csv` into a Pandas dataframe.
2. Extract numpy arrays from the dataframe for age (days) and concrete compressive strength (MPa).

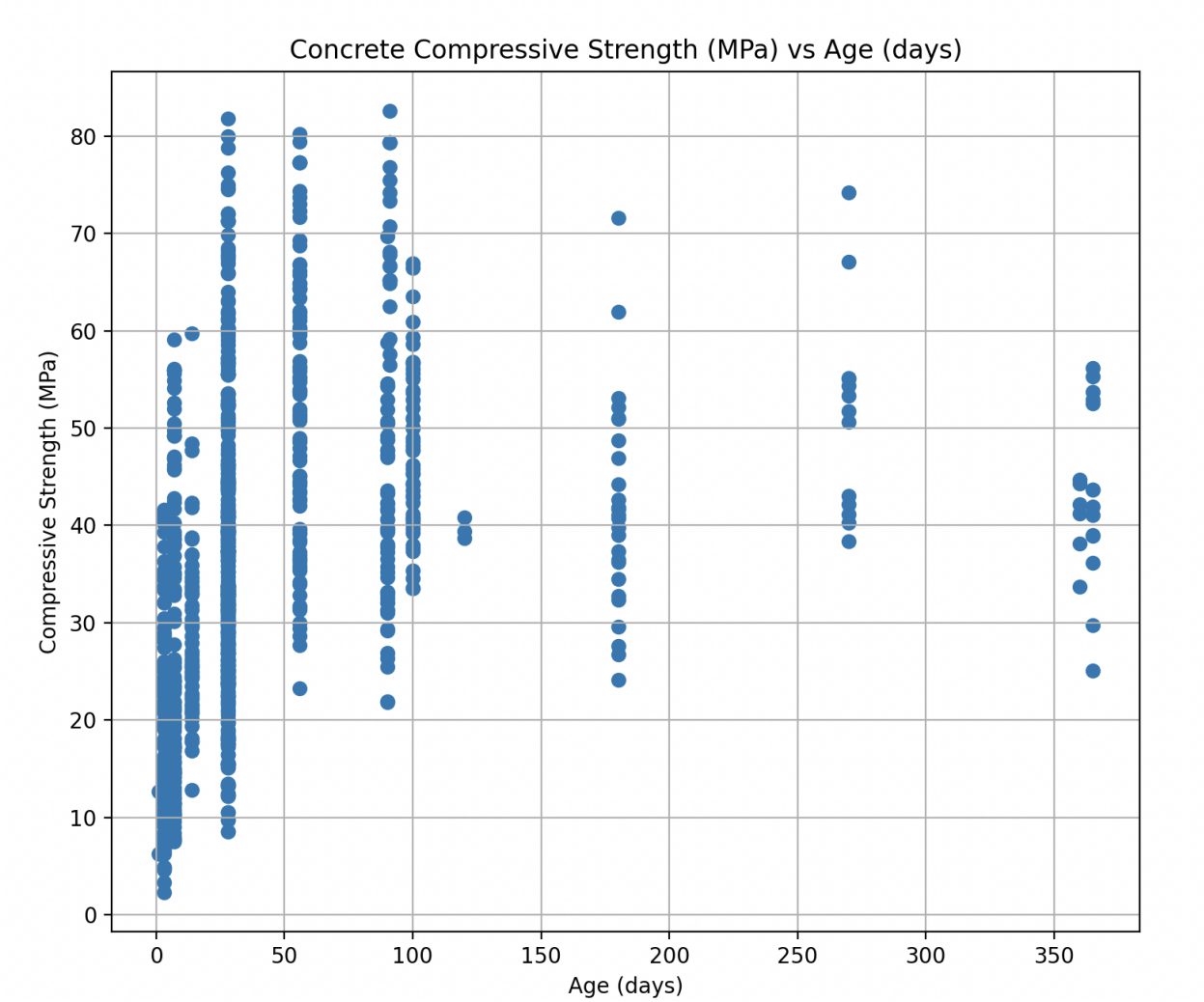

Figure 1: Scatter chart of concrete compressive strength (MPa) vs age (days).

3. Compute and print the range of parameter values for each input (e.g., Cement content, Blast Furnace Slag, Fly Ash, Water, Superplasticizer, Coarse Aggregate, Fine Aggregate and Age.

4. Compute and print the maximum, minimum, and average concrete compressive strengths.

5. Create a scatter chart of concrete strength (MPa) vs age (days). See Figure 1.

6. Use the Pandas cut function (i.e., google `pd.cut()`) to organize the strength data into intervals: 0–25, 25–50, 50–75, and 75–100.

7. Generate a histogram of concrete compressive strength (MPa).

8. Generate the cumulative probability distribution for concrete compressive strength, and then create a stair-step graph of "cumulative frequency" versus "compressive strength."

Note 1. The average value of the experimental results can be computed using Python's buildin functions. The "cumulative frequency" versus "compressive strength" is given by

$$\text{Cumulative frequency(y)} = \int_0^y p(x)dx \tag{1}$$

where $p(x)$ is the probability distribution of concrete compressive strengths. The matplotlib functions `plt.hist(.)` and `plt.step(.)` create histogram and stair-step graphs.

Note 2. See python-code.d/pandas/TestMaterialsA36Steel.py for a very similar problem setup and solution.

Question 2: 20 points.

With only 24 counties, Maryland is a relatively small state. And yet, according to the American Road and Transportation Builders Association, there are 5,484 highway bridges in Maryland, and 250 of them (or 4.6 percent) are classified as structurally deficient.

Highway bridges in Maryland are given one of three ratings: GOOD, FAIR and POOR (or worse). The term structurally deficient means that one or more of the bridges key elements – superstructure, substructure, foundation – are in poor condition (or worse). This question takes a first step toward understanding: (1) how many bridges there are in each county?, (2) which counties contain bridges that have a POOR condition rating?, and (3) who is responsible for the maintenance of these bridges? To get a handle on these concerns we will need a map of Maryland counties and a test program to read and process highway bridge inventory, including condition ratings, location, ownership and maintenance.

Test Program 01: Maryland Counties. The test program:

python-code.d/io/TestReadMarylandCountiesGEOJSON01.py,

reads data from:

python-code.d/data/geography/maryland/Maryland-County-Boundaries-Generalized.geojson

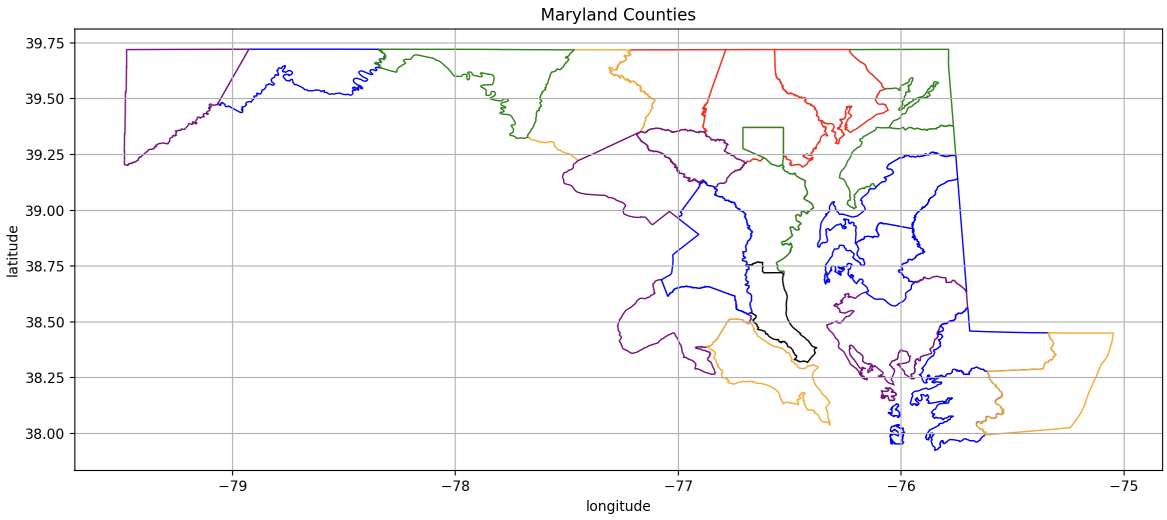

Figure 2: 24 counties in Maryland, including Baltimore City.

and then creates a dictionary of key, value pairs, connecting names of the counties (keys) to WKT (well known text) representations for geometry of the county boundary (values). Individual counties are loaded into GeoPandas, assigned colors, and visualized as shown Figure 2.

US counties are referenced by FIPS (Federal Information Processing Standard), a 5-digit code to represent a a state (first two digits) followed by a county (next three digits). For Maryland the details are as follows:

```
Maryland State Code Prefix:  24

COUNTY FIPS Codes
     001 Allegany County            025 Harford County
     003 Anne Arundel County        027 Howard County
     510 Baltimore City             029 Kent County
     005 Baltimore County           031 Montgomery County
     009 Calvert County             033 Prince George's County
     011 Caroline County            035 Queen Anne's County
     013 Carroll County             037 St.Mary's County
     015 Cecil County               039 Somerset County
     017 Charles County             041 Talbot County
     019 Dorchester County          043 Washington County
     021 Frederick County           045 Wicomico County
     023 Garrett County             047 Worcester County
```

Test Program 02: Highway Bridge Inventory. The test program:

python-code.d/io/TestReadHighwayBridgeSHP01.py

reads data from:

python-code.d/data/bridges/maryland/Bridge_Condition_NHS_2017.shp ...

a collection of 1,932 highway bridges in Maryland. Data for each bridge is organized into 28 columns:

```
Column      Header   Description:
-------------------------------------------------------------------------
     0  Item_8_Str   Structure number.
     1  Item_5_D_R   Route number of the inventory route.
     2  Item_3_Cou   County code.
     3  Item_6a_Fe   Features Intersected.
     4  Item_7_Fac   Facility Carried by Structure.
     5  Item_9_Loc   Location
     6  Item_11_Mi   Kilometerpoint (location of bridge on base highway network).
     7  Item_12_Ba   Base highway network.
     8  Item_16_La   Latitude (xx degrees xx minutes xx.xx seconds).
     9  Item_17_Lo   Longitude (xx degrees xx minutes xx.xx seconds).
    10  Item_21_Ma   Maintenance Responsibility
    11  Item_22_Ow   Owner
    12  Item_26_Fu   Functional Classification of Inventory Route (2 digits).
    13  Item_48_Sp   Length of Maximum Span (xxx meters).
    14  Item_49_St   Structure Length (xxx meters).
    15  Item_51_Wi   Bridge Roadway Width, Curb-to-Curb (m)
    16  Item_52_Wi   Bridge Deck Width (m)
```

```
    17  Item_58_Co   Bridge Deck Condition.
    18  Item_59_Co   Superstructure Condition.
    19  Item_60_Co   Substructure Condition.
    20  Item_61_Co   Channel and Channel Protection Condition.
    21  Item_62_Co   Culverts Conditioon.
    22  Item_104_H   Highway System of Inventory Route.
    23  Item_105_F   Federal Lands Highways
    24  MinimumCon   Minumum Condition.
    25  ConditionR   Condition Rating
    26  DeckArea_S   Deck Area (m^2).
    27  geometry     Location of bridge: POINT (lat, long) ...
```

A sample of output is:

```
--- Bridge 199 ...
---   Structure Number: 100000160162015 ...
---   Route Number of Inventory Route: 00095 ...
---   County Code: 033 ...
---   Features Intersected: SUITLAND ROAD ...
---   Facility Carried by Structure: IS 95 IL ...
---   Location: 1.71 MILES SOUTH OF MD 4 ...
---   Kilometerpoint: 0014576 ...
---   Base highway network: 1 ...
---   Latitude (degree mins secs): 38491861 ...
---   Longitude (degree mins secs): 076531591 ...
---   Maintenance Responsibility: 01 ...
---   Owner: 01 ...
---   Functional Classification of Inventory Route: 11 ...
---   Length of Maximum Span (m): 00125 ...
---   Structure Length (m): 000448 ...
---   Bridge Roadway Width, Curb-to-Curb (m): 0207 ...
---   Bridge Deck Width (m): 0213 ...
---   Bridge Deck Condition: 4 ...
---   Superstructure Condition: 5 ...
---   Substructure Condition: 5 ...
---   Channel and Channel Protection Condition: N ...
---   Culverts Conditioon: N ...
---   Highway System of Inventory Route: 1 ...
---   Federal Lands Highways: 0 ...
---   Minimum Condition: 4 ...
---   Bridge Rating: POOR ...
---   Deck Area (m^2): 954.24 ...
---   Geometry POINT (-76.88776469841079 38.82183509305251) ...
```

Additional information on the codes (e.g., county code: 033; maintenance responsibility: 66) are given by FIPS, and tables within Items 22 and 23 of FHWA-PD-96-001, Recording and Coding Guide for the Structure Inventory and Appraisal of the Nation's Bridges.

What to do? Write a Python program that will:

1. Create dictionaries linking: (1) county codes to county names, and (2) maintenance agency codes to agency names responsible for maintenance, e.g.,

```
County code:      033   --->  Prince George's County.
Maintenance code:  01   --->  State Highway Agency.
```

2. Load the contents of Bridge_Condition_NHS_2017.shp. Create a plot of the bridges within PG County – each bridge can be drawn as a single point (e.g., POINT (-76.88 38.82)) in GeoPandas.

3. Identify the number of bridges within each county having GOOD, FAIR and POOR ratings. Output these numbers in a tidy table.

4. Generate a list of bridges having a POOR rating, along with details of ownership and and maintenance responsibility.

Note 1: For item 2, I originally had in mind a plot for all of the bridges in Maryland, but I think that the result will just be a sea of blue dots. Focusing on a single county might give a picture that is more reasonable.

Note 2: For item 3, there are a number of ways of approaching the problem. Perhaps the simplest approach is to create three dictionaries – poor condition bridges, fair condition bridges, and good condition bridges, with the key being the county and the value being the number of bridges in the county.

A better approach might be to create a class `CountyBridgeConditions` that keeps track of the good, fair, and poor bridges in each county, and produces a string representation summarizing the overall condition of bridges in each geographical region. A single dictionary can link the county name to an object containing all of the details of the bridge conditions, which may simplify the generation of a tidy table summarizing the bridge conditions.

Question 3: 10 points. As shown in Figure 3 below, rectangles may be defined by the (x,y) coordinates of corner points that are diagonally opposite.

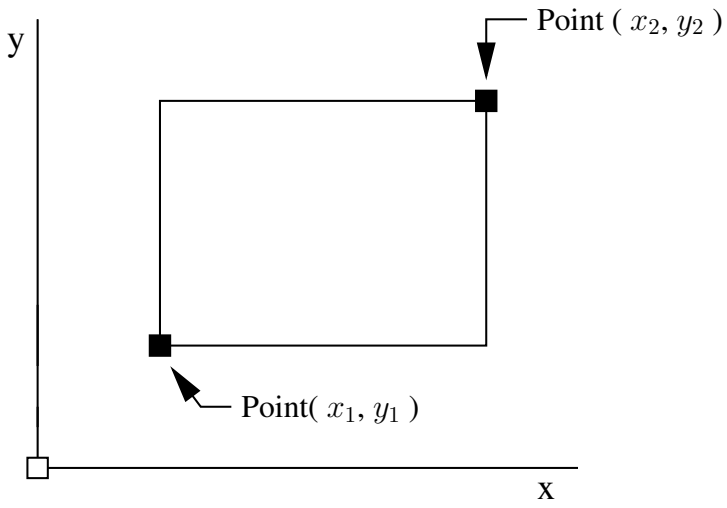

Figure 3: Definition of a rectangle via diagonally opposite corner points.

With this definition in place, the following script of code is a basic implementation of a class for creating and working with rectangle objects.

```
# ========================================================================
# Rectangle01.py: Very basic implementation of rectangle objects, where
# corner points are defined by varibles (x1, y1) and (x2, y2).
#
# Written by: Mark Austin                                    September, 2024
# ========================================================================

class Rectangle:

  def __init__(self, x1, y1, x2, y2 ):
    self.x1 = x1;
    self.y1 = y1;
    self.x2 = x2;
    self.y2 = y2;
    self.name = "";

  # Set rectangle name ...

  def setName(self, name ):
    self.name = name;

  # Compute perimeter of rectangle ..

  def getPerimeter (self):
     perimeter = 2*(abs(self.x2-self.x1) + abs(self.y2-self.y1))
     return perimeter
```

```
  # Compute area of rectangle ..

  def getArea (self):
     area = abs(self.x2-self.x1)*abs(self.y2-self.y1);
     return area

  # String represention of rectangle ...

  def __str__(self):
     rectangleinfo = [];

     ... details removed ...

     return "".join(rectangleinfo);
```

The Rectangle class uses variables x1, y1, x2, and y2 to define the corner points, and has a method to create rectangle objects (i.e, __init__), convert the details of a rectangle object into a string format (i.e., __str__), and compute the rectangle area and perimeter (i.e., getArea() and getPerimeter()).

A script of test program usage is as follows:

```
prompt >>
prompt >> python3 TestRectangle01.py
--- Enter TestRectangle01.main()    ...
--- ============================== ...
--- Part 1: Create and print rectangle A ...

--- Rectangle: A ...
--- ------------------------------------------------
---   Corner Point (x1,y1) = (  1.00,   2.00) ...
---   Corner Point (x2,y2) = (  3.00,   4.00) ...
---   Perimeter =   8.00 ...
---   Area      =   4.00 ...
--- ------------------------------------------------

--- Part 2: Create and print rectangle B ...

--- Rectangle: B ...
--- ------------------------------------------------
---   Corner Point (x1,y1) = (  0.00,   0.00) ...
---   Corner Point (x2,y2) = (  6.00,   5.00) ...
---   Perimeter =  22.00 ...
---   Area      =  30.00 ...
--- ------------------------------------------------

--- ============================== ...
--- Finished TestRectangle01.main()    ...
prompt >> exit
```

Source Code: Full details of the rectangle code and test program can be found in: python-code.d/objects/classes/

...

Question: Now suppose that instead of using the variables x_1, y_1, x_2 and y_2 to define the corner points, we use the class Point:

```
import math

class Point:

    def __init__(self, xCoord=0, yCoord=0):
        self.__xCoord = xCoord
        self.__yCoord = yCoord

    # get x coordinate

    def get_xCoord(self):
        return self.__xCoord

    ..... details of other functions removed ...
```

The appropriate modification for Rectangle is:

```
from Point import Point

class Rectangle:

   def __init__(self, x1, y1, x2, y2 ):
       self.pt1 = Point(x1,y1)     # <-- create lower corner point ...
       self.pt2 = Point(x2,y2)     # <-- create upper corner point ...

   ..... details rectangle removed ....
}
```

The arrangement of Rectangle and Point classes can be visualized as follows:

Figure 4: Test program script and classes in a rectangle system.

What to do? Fill in the missing details (i.e., constructors and __str__ method) of class Point. Modify the code in Rectangle to use the Point class. The resulting program should have essentially the same functionality as the original implementation (v1) of Rectangle.

Question 4: 20 points. The left-hand side of Figure 5 shows the essential details of a domain familiar to many children. One by one, rectangular blocks are stacked as high as possible until they come tumbling down – the goal, afterall, is to create a spectacular crash!!

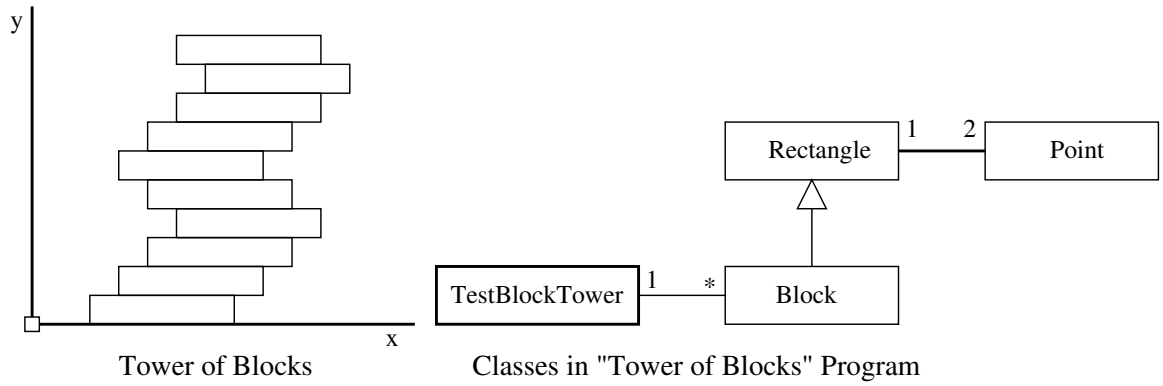

Figure 5: Schematic and classes for tower of blocks problem.

Suppose that we wanted to model this process and use engineering principles to predict incipient instability of the block tower. Consider the following observations:

1. Rather than start from scratch, it would make sense to create a Block class that inherits the properties of Rectangle (previous question), and adds details relevant to engineering analysis (e.g., the density of the block).

2. Then we could develop a BlockTower class that systematically assembles the tower, starting at the base and working upwards. At each step of the tower assembly, analysis procedures should make sure that the tower is still stable.

The right-hand side of Figure 5 shows the relatioship among the classes. One TestBlockTower program (1) will employ many blocks, as indicated by the asterik (*).

Develop a Python program that builds upon the Rectangle class written in the previous questions. The class Block should store the depth and density of the block – this will be important in determining the mass and centroid of each block. The TestBlockTower class will use block objects to build the tower. A straight forward way of modeling the block tower is with a List. After each block is added, the program should conduct a stability check. If the system is still stable, then add another block should be added. The simulation should cease when the tower of blocks eventually becomes unstable.

Note. To simplify the analysis, assume that adjacent blocks are firmly connected.

Stability Considerations. If the blocks are stacked perfectly on top of each other, then from a mathematical standpoint the tower will never become unstable. In practice, this never happens. There is always a small offset and, eventually, it's the accumulation of offsets that leads to spectacular disaster.

For the purposes of this question, assume that blocks are five units wide, one unit high, and depth of one unit. When a new block is added, the block offset should be one unit. To make the question interesting, assume that four blocks are stacked with an offset to the right, then three blocks are added with an offset to the left, then four to the right, three to the left, and so forth. This sequence can be accomplished with the looping construct:

```
# Compute incremental offset for i-th block .....

offset = math.floor((BlockNo - 1)/5.0) + (BlockNo-1)%5;
if ((BlockNo-1)%5 == 4 ):
   offset = offset - 2;
```

The tower will become unstable when the center of gravity of blocks above a particular level falls outside the edge of the supporting block.

What to do? Write a Python program that will:

1. Determine how many blocks can be added to the stack before it crashes.
2. Create a figure of the block configuration and centroid position immediately before collapse.