pivot_tables_and_graphs

May 26, 2021

1 Using Python to Create Pivot Tables, Output Files, and Graphs:

1.1 A Python Tutorial Program

By Kenneth Burchfiel

In this program, I will demonstrate how Python can be used to turn raw data into pivot tables and graphs. The program uses a fictional scenario and simulated data, but the concepts it presents should prove useful in a number of basic data analysis applications.

For a more detailed introduction, please see my related blog post. [Link to come].

1.2 Scenario:

(Note: both the scenario and the data presented in this program are fictional.)

Orange Valley School District (OVSD) is piloting an after-school math program called Code the Concepts in each of its 8 high schools. This program aims to teach students relevant math skills through programming and computer science applications, and is available in grades 8-12.

Although the district believes that the programming knowledge taught in this course is highly valuable in itself, it also wants to evaluate the impact, if any, this program has on students' math skills. The means of evaluating the math skills will be the Higher Grades Math Assessment (HGMA), a test administered at the beginning and end of each school year.

Let's say that I am an analyst tasked with evaluating this HGMA data. In this notebook, I will show how I can use Python for this type of analytics project.

I will first import a number of libraries:

[1]:
```
import time

start_time = time.time()

import pandas as pd
import numpy as np
import csv
import matplotlib.pyplot as plt
from adjustText import adjust_text
```

1.3 Part 1: Creating simulated data for analysis

I will first need to create the fictional data that I will 'analyze' in the following program. Let's say that OVSD has 8 high schools serving grades 9-12, with around 120 students in each grade. About half of these students will be enrolled in Code the Concepts and the other half will not. Therefore, to store HGMA data for each of these students, I will need to create a Pandas DataFrame with around 3840 rows (8 high schools * 4 grades * 120 students per grade) for the beginning-of-year HGMA score results, and another equally sized DataFrame for the end-of-year results.

One relatively efficient strategy for building this table is to randomly assign integers representing schools, grades, and enrollment values to each student in the table. These integers can then be replaced with string values. Meanwhile, the test score variable will be a combination of random chance and effects from the school, grade, and Code the Concepts enrollment variables.

I will first create an empty DataFrame for the start-of-year data:

[2]:
```
df_scores_start = pd.DataFrame(index=np.
 ↪arange(1,3841),columns=['School','Grade','Enrolled','Time','Score'])
df_scores_start.index.rename('Student_ID',inplace=True)
df_scores_start
```

[2]:
```
           School Grade Enrolled Time Score
Student_ID
1             NaN   NaN      NaN  NaN   NaN
2             NaN   NaN      NaN  NaN   NaN
3             NaN   NaN      NaN  NaN   NaN
4             NaN   NaN      NaN  NaN   NaN
5             NaN   NaN      NaN  NaN   NaN
…           …   …       …   …    …
3836          NaN   NaN      NaN  NaN   NaN
3837          NaN   NaN      NaN  NaN   NaN
3838          NaN   NaN      NaN  NaN   NaN
3839          NaN   NaN      NaN  NaN   NaN
3840          NaN   NaN      NaN  NaN   NaN

[3840 rows x 5 columns]
```

Next, I will use both random numbers and direct assignment to populate the values in this DataFrame.

[3]:
```
rng = np.random.default_rng(1337) # ;)
```

[4]:
```
for i in range (0, len(df_scores_start)):
    df_scores_start.loc[df_scores_start.index[i],'School'] = rng.integers(0,8)␣
 ↪# Returns integers between 1 and 8 to represent each of the 8 schools in the␣
 ↪district. See https://numpy.org/doc/stable/reference/random/generated/numpy.
 ↪random.Generator.integers.html
    df_scores_start.loc[df_scores_start.index[i],'Grade'] = rng.integers(9,13)
    df_scores_start.loc[df_scores_start.index[i],'Enrolled'] = rng.integers(0,2)
```

```
    df_scores_start.loc[df_scores_start.index[i],'Score']= rng.integers(30,80)

df_scores_start['Time'] = '2018_9' # 2018_9 = September 2018, the start of the␣
↪academic year; 2019_5 = May 2019, the end of the academic year.
df_scores_start['Count'] = 1 # Creating a row with the value 1 for every␣
↪student will make it easier to use np.sum to return record counts within␣
↪pivot tables.
school_list = ['Westwood', 'Fair Lake', 'Eagle', 'East River', 'Bayville',␣
↪'Cardinal', 'Olive', 'Central'] # The names of each high school in the␣
↪(fictional) district.

for i in range(len(df_scores_start)):
    df_scores_start.loc[df_scores_start.index[i],'School'] =␣
↪school_list[df_scores_start.loc[df_scores_start.index[i],'School']]# This␣
↪line replaces the integer corresponding to each school with the school in␣
↪school_list whose index equals that integer.
    if df_scores_start.iloc[i,2] == 1: # Converting the Enrolled column from␣
↪numerical values to string values. Using .iloc produces shorter lines of␣
↪code than the above format, but it is also less intuitive (as numbers are␣
↪used in place of column labels).
        df_scores_start.iloc[i,2] = 'Yes'
    else:
        df_scores_start.iloc[i,2] = 'No'
```

[5]:
```
df_scores_start
```

[5]:
```
               School Grade Enrolled    Time Score  Count
Student_ID
1           Bayville    12      Yes  2018_9    39      1
2           Bayville    12       No  2018_9    77      1
3              Eagle    12       No  2018_9    35      1
4           Westwood     9       No  2018_9    47      1
5              Eagle    10      Yes  2018_9    74      1
…                …    …        …       …     …      …
3836           Olive    10      Yes  2018_9    34      1
3837        Bayville    10      Yes  2018_9    30      1
3838        Cardinal    11       No  2018_9    59      1
3839           Eagle    12      Yes  2018_9    33      1
3840        Westwood    10      Yes  2018_9    66      1

[3840 rows x 6 columns]
```

Next, I will create a DataFrame for the end-of-year data. This table will be a copy of the start-of-year DataFrame, except that the Score column will have new values from the HGMA end-of-year test results. These results will be based on the beginning-of-year data, but will also reflect positive effects of the Code the Concepts program for certain schools and classes.

```
[6]: df_scores_end = df_scores_start.copy() # copy() is required here in order to
↪prevent changes to df_scores_end from affecting df_scores_start.
school_list = ['Fair Lake', 'Eagle', 'Bayville', 'Central']
grade_list = [10, 12]
for i in range(len(df_scores_end)):
    if df_scores_end.loc[df_scores_end.index[i],'Enrolled'] == 'Yes':
        if df_scores_end.loc[df_scores_end.index[i],'School'] in school_list:
            df_scores_end.loc[df_scores_end.index[i],'Score'] += 3+rng.
↪integers(0,8) # As a result of this line, Code the Concepts students who
↪attend one of the schools in school_list will earn a higher HGMA score than
↪will other students.
        if df_scores_end.loc[df_scores_end.index[i],'Grade'] in grade_list:
            df_scores_end.loc[df_scores_end.index[i],'Score'] += 5+rng.
↪integers(0,10) # Similarly, Code the Concepts students who are currently in
↪one of the grades in grade_list will earn a higher HGMA score than will
↪other students.
    df_scores_end.loc[df_scores_end.index[i],'Score'] += rng.integers(-5,10)  #
↪This line creates allows for some random (but generally positive) change in
↪each student's HGMA score.
    df_scores_end.loc[df_scores_end.index[i],'Score'] = min(df_scores_end.
↪loc[df_scores_end.index[i],'Score'],100) # Keeps all scores below 100
df_scores_end['Time'] = '2019_5'
df_scores_end
```

```
[6]:               School Grade Enrolled    Time Score  Count
Student_ID
1           Bayville    12      Yes  2019_5    54      1
2           Bayville    12       No  2019_5    75      1
3              Eagle    12       No  2019_5    39      1
4           Westwood     9       No  2019_5    49      1
5              Eagle    10      Yes  2019_5    92      1
…                …     …        …       …     …      …
3836           Olive    10      Yes  2019_5    45      1
3837        Bayville    10      Yes  2019_5    54      1
3838        Cardinal    11       No  2019_5    68      1
3839           Eagle    12      Yes  2019_5    52      1
3840        Westwood    10      Yes  2019_5    81      1

[3840 rows x 6 columns]
```

These start-of-year and end-of-year DataFrames can then be combined into a single DataFrame using pd.concat:

```
[7]: df_scores = pd.concat([df_scores_start.copy(),df_scores_end.copy()])
df_scores.sort_index(inplace=True)
df_scores
```

[7]:
```
                  School Grade Enrolled    Time Score  Count
Student_ID
1            Bayville    12      Yes  2018_9    39      1
1            Bayville    12      Yes  2019_5    54      1
2            Bayville    12       No  2018_9    77      1
2            Bayville    12       No  2019_5    75      1
3               Eagle    12       No  2018_9    35      1
…               …   …     …       …    …     …
3838         Cardinal    11       No  2018_9    59      1
3839            Eagle    12      Yes  2019_5    52      1
3839            Eagle    12      Yes  2018_9    33      1
3840         Westwood    10      Yes  2018_9    66      1
3840         Westwood    10      Yes  2019_5    81      1

[7680 rows x 6 columns]
```

[8]:
```
df_scores['Grade'] = pd.to_numeric(df_scores['Grade']) # These two lines␣
↪convert grade and school data to numerical values.
df_scores['Score'] = pd.to_numeric(df_scores['Score'])
df_scores.dtypes
```

[8]:
```
School      object
Grade        int64
Enrolled    object
Time        object
Score        int64
Count        int64
dtype: object
```

[9]:
```
df_scores.to_excel('scores_by_program_enrollment.xlsx')
```

1.4 Part 2: Analysis function definitions and data import

With the simulated data created and exported to Excel, I can now start writing code for my analysis program.

The following function appends tables to an output CSV file, and also allows a title to be placed above each table. If the output file (specified by the file_path variable) does not yet exist, the function will create it automatically.

[10]:
```
def append_csv(file_path,title_string,table): # Table = dataframe to append to␣
↪the CSV; title_string = header to place above table
    with open(file_path,'a',newline='') as file_to_write:
        writer = csv.writer(file_to_write)
        writer.writerows(['\n','\n']) # Inserts two spaces in the CSV file␣
↪after the above table for improved readability
        writer.writerow([title_string]) # The list notation is necessary;␣
↪otherwise the text will be input one letter per cell!
```

```
    table.to_csv(file_path, mode='a')

output_file_path = 'output_tables.csv'
```

The above function is designed to append, rather than overwrite, output_tables.csv. As a result, if the program is run multiple times, duplicate tables will be appended to the program. Therefore, to avoid this issue, the line below overwrites any pre-existing output_tables.csv with a blank file, and also appends a title to that file.

[11]:
```
with open(output_file_path,'w',newline='') as file_to_write: # Will create this␣
↪file if it does not already exist (or will overwrite it if it does exist)
    writer = csv.writer(file_to_write)
    writer.writerow(['HGMA Score Data for OVSD High Schoolers'])
```

The following function builds off Pandas' pivot_table function to create a pivot table with subtotal rows. This function will make it easier to evaluate both individual grade data and grouped data for schools and enrollment conditions.

[12]:
```
def pivot_with_subtotals(df, values, index, aggfunc, levels): # 'values',␣
↪'index', and 'aggfunc' will be fed directly into pivot_table below. 'levels'␣
↪refers to the number of levels to add into the pivot table. The order of the␣
↪elements in the 'index' list is important, since each set of subtotal rows␣
↪will be created by deleting the rightmost element of index.

    # For example, suppose that 'index' equals ['Enrolled', 'Time', 'School',␣
↪'Grade'], and 'levels' equals 3. This means that the first pivot table will␣
↪use 'Enrolled', 'Time', 'School', and 'Grade' as its index values; the␣
↪second pivot table will use 'Enrolled', 'Time', and 'School' (thus grouping␣
↪the values by grade); and the third table will use 'Enrolled' and 'Time'␣
↪(thus grouping by school and grade and allowing totals to be calculated for␣
↪all enrolled and non-enrolled students). The function concatenates each of␣
↪these tables together, producing one unified pivot table.

    modified_index = index.copy() # modified_index will store the index values␣
↪used in each iteration of the pivot_table function below.
    pivot_combined = pd.DataFrame() # This empty DataFrame will be populated␣
↪with output from the pivot table.
    for i in range(levels): # The higher the levels value, the more subtotal␣
↪levels will be added into the pivot table.
        pivot_individual = pd.pivot_table(data = df, values=values,␣
↪index=modified_index, aggfunc=aggfunc) # See https://pandas.pydata.org/
↪pandas-docs/stable/reference/api/pandas.pivot_table.html
        pivot_individual.reset_index(inplace=True) # Resetting the index makes␣
↪it easier to concatenate different pivot tables on top of each other.
        pivot_combined = pd.concat([pivot_combined,pivot_individual])
        del(modified_index[-1]) # Deletes the final element in modified_index,␣
↪allowing the next run of pivot_table to create a subtotal row.
```

```
    pivot_combined.columns = pivot_combined.columns.to_flat_index() #␣
↪pivot_table() may output its results as a multindex. For the purposes of␣
↪this program, I prefered to work with simple indices, so I used␣
↪to_flat_index() to convert the multindex-formatted columns to a flat index.
    for column in pivot_combined.columns: # When columns are converted from a␣
↪multindex to a flat index, the result is stored as a tuple. The following␣
↪lines of code convert each tuple to a string. For instance, the code will␣
↪convert the tuple ('count', 'Score') to count_Score.
        #print(type(column))
        if isinstance(column, tuple) == True:
            if len(column[1]) > 0:
                pivot_combined.rename(columns={column:
↪column[0]+'_'+column[1]},inplace=True) # [0] and [1] represent the two␣
↪elements of the tuple
            else: # Some tuples are in the format ('Column_name',''). In this␣
↪case, there's no need to add an underscore after the first tuple element, so␣
↪the following line of code simply replaces the tuple with its first element.
                pivot_combined.rename(columns={column:column[0]},inplace=True)
    # Currently, the combined DataFrame has NaN values in cells within columns␣
↪that were not included in the subtotal/total rows. The following for loop␣
↪changes these cells to 'Total'. The for loop only covers colunms included in␣
↪index_i in order not to alter values columns.
    for i in range(len(index)):
        print(pivot_combined.columns[i])
        pivot_combined.fillna(value={pivot_combined.columns[i]:
↪'Total'},axis=0,inplace=True) # See https://pandas.pydata.org/pandas-docs/
↪stable/reference/api/pandas.DataFrame.fillna.html . Axis=0 may not be␣
↪necessary.
    pivot_combined.reset_index(drop=True,inplace=True)
    return pivot_combined
```

In the real world, the data generated earlier and stored in df_scores might exist in the form of an Excel or CSV file. The following line imports df_scores from Excel into a DataFrame.

[13]:
```
df_scores = pd.read_excel('scores_by_program_enrollment.
 ↪xlsx',index_col='Student_ID')
df_scores
```

[13]:
```
                School  Grade Enrolled    Time  Score  Count
Student_ID
1            Bayville     12      Yes  2018_9     39      1
1            Bayville     12      Yes  2019_5     54      1
2            Bayville     12       No  2018_9     77      1
2            Bayville     12       No  2019_5     75      1
3               Eagle     12       No  2018_9     35      1
…                   …    …        …       …      …
3838         Cardinal     11       No  2018_9     59      1
```

```
3839          Eagle      12       Yes   2019_5      52        1
3839          Eagle      12       Yes   2018_9      33        1
3840       Westwood      10       Yes   2018_9      66        1
3840       Westwood      10       Yes   2019_5      81        1

[7680 rows x 6 columns]
```

1.5 Part 3: Creating a pivot table using pivot_with_subtotals formula

[14]:
```
pd.set_option('display.max_rows',200)
```

The following code block creates a pivot table (based off df_scores) with both individual and subtotal rows. The comments in the function definition above explain how the levels and index arguments work together to generate subtotal values.

[15]:
```
df_scores_pivot_combined = pivot_with_subtotals(df = df_scores, values =␣
 ↪'Score', index = ['Enrolled', 'Time', 'School', 'Grade'], aggfunc = ['mean',␣
 ↪'count'], levels=3)

df_scores_pivot_combined.
 ↪sort_values(['School','Grade','Enrolled','Time'],inplace=True) # This sort␣
 ↪is important for correctly drawing each set of graphs later in the program.
df_scores_pivot_combined.reset_index(drop=True,inplace=True) # Resets the index␣
 ↪so that it corresponds with the newly sorted rows.
append_csv('output_tables.csv','Pivot Table for Line␣
 ↪Graphs',df_scores_pivot_combined) # Outputting this table to a .csv file␣
 ↪will make it easier to share the results with individuals who don't have␣
 ↪access to a code editor.

df_scores_pivot_combined
```

```
Enrolled
Grade
School
Time
```

[15]:

	Enrolled	Grade	School	Time	count_Score	mean_Score
0	No	9.0	Bayville	2018_9	50	54.660000
1	No	9.0	Bayville	2019_5	50	58.040000
2	Yes	9.0	Bayville	2018_9	52	56.096154
3	Yes	9.0	Bayville	2019_5	52	64.038462
4	No	10.0	Bayville	2018_9	59	52.050847
5	No	10.0	Bayville	2019_5	59	53.542373
6	Yes	10.0	Bayville	2018_9	56	53.232143
7	Yes	10.0	Bayville	2019_5	56	70.125000
8	No	11.0	Bayville	2018_9	56	56.785714
9	No	11.0	Bayville	2019_5	56	58.500000
10	Yes	11.0	Bayville	2018_9	57	55.157895

11	Yes	11.0	Bayville	2019_5	57	63.982456
12	No	12.0	Bayville	2018_9	60	52.950000
13	No	12.0	Bayville	2019_5	60	53.916667
14	Yes	12.0	Bayville	2018_9	62	54.693548
15	Yes	12.0	Bayville	2019_5	62	70.967742
16	No	Total	Bayville	2018_9	225	54.048889
17	No	Total	Bayville	2019_5	225	55.875556
18	Yes	Total	Bayville	2018_9	227	54.770925
19	Yes	Total	Bayville	2019_5	227	67.418502
20	No	9.0	Cardinal	2018_9	54	53.851852
21	No	9.0	Cardinal	2019_5	54	54.981481
22	Yes	9.0	Cardinal	2018_9	57	51.982456
23	Yes	9.0	Cardinal	2019_5	57	54.280702
24	No	10.0	Cardinal	2018_9	59	54.932203
25	No	10.0	Cardinal	2019_5	59	56.050847
26	Yes	10.0	Cardinal	2018_9	64	50.796875
27	Yes	10.0	Cardinal	2019_5	64	62.640625
28	No	11.0	Cardinal	2018_9	55	53.218182
29	No	11.0	Cardinal	2019_5	55	55.327273
30	Yes	11.0	Cardinal	2018_9	61	55.590164
31	Yes	11.0	Cardinal	2019_5	61	57.245902
32	No	12.0	Cardinal	2018_9	55	51.527273
33	No	12.0	Cardinal	2019_5	55	54.527273
34	Yes	12.0	Cardinal	2018_9	55	50.418182
35	Yes	12.0	Cardinal	2019_5	55	62.072727
36	No	Total	Cardinal	2018_9	223	53.408072
37	No	Total	Cardinal	2019_5	223	55.237668
38	Yes	Total	Cardinal	2018_9	237	52.227848
39	Yes	Total	Cardinal	2019_5	237	59.109705
40	No	9.0	Central	2018_9	63	52.190476
41	No	9.0	Central	2019_5	63	53.698413
42	Yes	9.0	Central	2018_9	62	52.129032
43	Yes	9.0	Central	2019_5	62	61.403226
44	No	10.0	Central	2018_9	56	53.071429
45	No	10.0	Central	2019_5	56	55.178571
46	Yes	10.0	Central	2018_9	56	56.910714
47	Yes	10.0	Central	2019_5	56	74.821429
48	No	11.0	Central	2018_9	67	53.626866
49	No	11.0	Central	2019_5	67	55.880597
50	Yes	11.0	Central	2018_9	55	53.345455
51	Yes	11.0	Central	2019_5	55	61.763636
52	No	12.0	Central	2018_9	61	55.196721
53	No	12.0	Central	2019_5	61	57.409836
54	Yes	12.0	Central	2018_9	55	53.618182
55	Yes	12.0	Central	2019_5	55	71.636364
56	No	Total	Central	2018_9	247	53.522267
57	No	Total	Central	2019_5	247	55.542510

58	Yes	Total	Central	2018_9	228	53.956140
59	Yes	Total	Central	2019_5	228	67.254386
60	No	9.0	Eagle	2018_9	72	54.888889
61	No	9.0	Eagle	2019_5	72	56.486111
62	Yes	9.0	Eagle	2018_9	58	53.224138
63	Yes	9.0	Eagle	2019_5	58	61.534483
64	No	10.0	Eagle	2018_9	68	54.897059
65	No	10.0	Eagle	2019_5	68	56.970588
66	Yes	10.0	Eagle	2018_9	63	54.619048
67	Yes	10.0	Eagle	2019_5	63	72.857143
68	No	11.0	Eagle	2018_9	61	54.426230
69	No	11.0	Eagle	2019_5	61	56.885246
70	Yes	11.0	Eagle	2018_9	68	50.764706
71	Yes	11.0	Eagle	2019_5	68	59.823529
72	No	12.0	Eagle	2018_9	83	54.337349
73	No	12.0	Eagle	2019_5	83	56.301205
74	Yes	12.0	Eagle	2018_9	59	53.135593
75	Yes	12.0	Eagle	2019_5	59	70.372881
76	No	Total	Eagle	2018_9	284	54.630282
77	No	Total	Eagle	2019_5	284	56.633803
78	Yes	Total	Eagle	2018_9	248	52.883065
79	Yes	Total	Eagle	2019_5	248	66.044355
80	No	9.0	East River	2018_9	45	48.777778
81	No	9.0	East River	2019_5	45	50.733333
82	Yes	9.0	East River	2018_9	55	59.145455
83	Yes	9.0	East River	2019_5	55	61.509091
84	No	10.0	East River	2018_9	55	55.181818
85	No	10.0	East River	2019_5	55	58.000000
86	Yes	10.0	East River	2018_9	59	56.779661
87	Yes	10.0	East River	2019_5	59	67.864407
88	No	11.0	East River	2018_9	66	55.136364
89	No	11.0	East River	2019_5	66	58.106061
90	Yes	11.0	East River	2018_9	57	56.368421
91	Yes	11.0	East River	2019_5	57	59.315789
92	No	12.0	East River	2018_9	55	54.636364
93	No	12.0	East River	2019_5	55	56.472727
94	Yes	12.0	East River	2018_9	66	52.409091
95	Yes	12.0	East River	2019_5	66	63.757576
96	No	Total	East River	2018_9	221	53.728507
97	No	Total	East River	2019_5	221	56.171946
98	Yes	Total	East River	2018_9	237	56.012658
99	Yes	Total	East River	2019_5	237	63.189873
100	No	9.0	Fair Lake	2018_9	64	54.718750
101	No	9.0	Fair Lake	2019_5	64	56.812500
102	Yes	9.0	Fair Lake	2018_9	64	51.171875
103	Yes	9.0	Fair Lake	2019_5	64	59.750000
104	No	10.0	Fair Lake	2018_9	47	54.510638

105	No	10.0	Fair Lake	2019_5	47	56.340426
106	Yes	10.0	Fair Lake	2018_9	61	55.639344
107	Yes	10.0	Fair Lake	2019_5	61	74.836066
108	No	11.0	Fair Lake	2018_9	61	54.918033
109	No	11.0	Fair Lake	2019_5	61	56.442623
110	Yes	11.0	Fair Lake	2018_9	69	54.420290
111	Yes	11.0	Fair Lake	2019_5	69	63.797101
112	No	12.0	Fair Lake	2018_9	61	52.803279
113	No	12.0	Fair Lake	2019_5	61	54.393443
114	Yes	12.0	Fair Lake	2018_9	65	52.200000
115	Yes	12.0	Fair Lake	2019_5	65	69.323077
116	No	Total	Fair Lake	2018_9	233	54.227468
117	No	Total	Fair Lake	2019_5	233	55.987124
118	Yes	Total	Fair Lake	2018_9	259	53.347490
119	Yes	Total	Fair Lake	2019_5	259	66.783784
120	No	9.0	Olive	2018_9	59	53.101695
121	No	9.0	Olive	2019_5	59	56.118644
122	Yes	9.0	Olive	2018_9	51	54.411765
123	Yes	9.0	Olive	2019_5	51	55.470588
124	No	10.0	Olive	2018_9	63	53.539683
125	No	10.0	Olive	2019_5	63	56.523810
126	Yes	10.0	Olive	2018_9	57	54.105263
127	Yes	10.0	Olive	2019_5	57	66.719298
128	No	11.0	Olive	2018_9	51	52.960784
129	No	11.0	Olive	2019_5	51	54.333333
130	Yes	11.0	Olive	2018_9	57	56.000000
131	Yes	11.0	Olive	2019_5	57	56.842105
132	No	12.0	Olive	2018_9	54	55.314815
133	No	12.0	Olive	2019_5	54	57.833333
134	Yes	12.0	Olive	2018_9	71	55.690141
135	Yes	12.0	Olive	2019_5	71	66.845070
136	No	Total	Olive	2018_9	227	53.718062
137	No	Total	Olive	2019_5	227	56.237885
138	Yes	Total	Olive	2018_9	236	55.105932
139	Yes	Total	Olive	2019_5	236	61.940678
140	No	Total	Total	2018_9	1924	53.959459
141	No	Total	Total	2019_5	1924	56.038981
142	Yes	Total	Total	2018_9	1916	54.085595
143	Yes	Total	Total	2019_5	1916	64.078810
144	No	9.0	Westwood	2018_9	73	56.739726
145	No	9.0	Westwood	2019_5	73	58.876712
146	Yes	9.0	Westwood	2018_9	53	56.301887
147	Yes	9.0	Westwood	2019_5	53	57.264151
148	No	10.0	Westwood	2018_9	73	52.493151
149	No	10.0	Westwood	2019_5	73	55.054795
150	Yes	10.0	Westwood	2018_9	61	53.475410
151	Yes	10.0	Westwood	2019_5	61	64.606557

152	No	11.0	Westwood	2018_9	60	54.450000
153	No	11.0	Westwood	2019_5	60	56.383333
154	Yes	11.0	Westwood	2018_9	65	56.323077
155	Yes	11.0	Westwood	2019_5	65	58.246154
156	No	12.0	Westwood	2018_9	58	52.896552
157	No	12.0	Westwood	2019_5	58	55.189655
158	Yes	12.0	Westwood	2018_9	65	52.246154
159	Yes	12.0	Westwood	2019_5	65	63.015385
160	No	Total	Westwood	2018_9	264	54.200758
161	No	Total	Westwood	2019_5	264	56.443182
162	Yes	Total	Westwood	2018_9	244	54.520492
163	Yes	Total	Westwood	2019_5	244	60.893443

The above table shows the mean scores for each grade in each school as a function of their Code the Concepts enrollment. However, it also includes subtotal rows that show the mean scores for each school, still grouped by enrollment status, along with rows showing the mean scores for each enrollment status. These subtotal rows provide better insight into how various groups of students performed.

1.6 Part 4: Adding data from this pivot table to a dashboard with an alternate format

The above table will prove useful for the grouped line charts to be created below. However, suppose that there is also an online dashboard that uses a different format than this table. It would be cumbersome and time-consuming to copy and paste the individual cells into the new dashboard; however, using Pandas' query() function, I can automate the reformatting process.

In the following blocks of code, I will create a copy of this online dashboard in DataFrame form (df_dashboard), then run a for loop that copies the relevant data from df_scores_pivot_combined into df_dashboard.

[16]:
```python
# Step 1: initialize the new DataFrame based on the format of the online dashboard

df_dashboard = pd.DataFrame(columns=['School','Time','E_9','E_10','E_11','E_12','E_Total', 'NE_9', 'NE_10', 'NE_11', 'NE_12', 'NE_Total'])
df_dashboard['School'] = ['Westwood', 'Westwood', 'Fair Lake', 'Fair Lake', 'Eagle', 'Eagle', 'East River', 'East River', 'Bayville', 'Bayville', 'Cardinal', 'Cardinal', 'Olive', 'Olive', 'Central', 'Central', 'Total', 'Total'] # There will be two rows for each school, one for each time period

for i in range(0,len(df_dashboard),2): # Adds the time values into each pair of school rows.
    df_dashboard.iloc[i,1] = '2018_9'
    df_dashboard.iloc[i+1,1] = '2019_5'
```

```
df_dashboard
```

[16]:

```
        School    Time  E_9  E_10  E_11  E_12  E_Total  NE_9  NE_10  NE_11  NE_12  \
0     Westwood  2018_9  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
1     Westwood  2019_5  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
2    Fair Lake  2018_9  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
3    Fair Lake  2019_5  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
4        Eagle  2018_9  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
5        Eagle  2019_5  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
6   East River  2018_9  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
7   East River  2019_5  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
8     Bayville  2018_9  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
9     Bayville  2019_5  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
10    Cardinal  2018_9  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
11    Cardinal  2019_5  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
12       Olive  2018_9  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
13       Olive  2019_5  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
14     Central  2018_9  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
15     Central  2019_5  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
16       Total  2018_9  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN
17       Total  2019_5  NaN   NaN   NaN   NaN      NaN   NaN    NaN    NaN    NaN

   NE_Total
0       NaN
1       NaN
2       NaN
3       NaN
4       NaN
5       NaN
6       NaN
7       NaN
8       NaN
9       NaN
10      NaN
11      NaN
12      NaN
13      NaN
14      NaN
15      NaN
16      NaN
17      NaN
```

This format stores grades and enrollment statuses together within columns, rather than within rows. To convert the data in df_scores_pivot_combined into this new format, I will create a for loop that makes heavy use of Pandas' df.query() function.

For each row in df_dashboard, the following for loop first stores that row's school and time values as variables. It then goes through each column of df_dashboard where mean values are to be stored, and, using the df.query() function, finds the correct mean value within df_scores_pivot_combined to enter into the row-column pair within df_dashboard. It then assigns this mean value to the row-column pair.

Because the Total rows have less information available, the for loop only looks for certain data points in order to avoid errors.

[17]:
```
if df_scores_pivot_combined.columns[5] == 'mean_Score': # if column 5 of␣
↪df_scores_pivot_combined doesn't contain the mean values, then the for loop␣
↪won't retrieve the incorrect values; an exception is raised to prevent this.
    for i in range(len(df_dashboard)):
        # print('Now on row',i) # for debugging
        curr_school = df_dashboard.loc[df_dashboard.index[i],'School']
        curr_time = df_dashboard.iloc[i,1] # Used .loc for curr_school and .
↪iloc for curr_time to demonstrate two ways for accessing these values
        if curr_school != 'Total':
            df_dashboard.loc[df_dashboard.index[i],'E_9'] =␣
↪df_scores_pivot_combined.query("Time == @curr_time & School == @curr_school␣
↪& Grade == 9 & Enrolled == 'Yes'").iloc[0,5]
            df_dashboard.loc[df_dashboard.index[i],'E_10'] =␣
↪df_scores_pivot_combined.query("Time == @curr_time & School == @curr_school␣
↪& Grade == 10 & Enrolled == 'Yes'").iloc[0,5]
            df_dashboard.loc[df_dashboard.index[i],'E_11'] =␣
↪df_scores_pivot_combined.query("Time == @curr_time & School == @curr_school␣
↪& Grade == 11 & Enrolled == 'Yes'").iloc[0,5]
            df_dashboard.loc[df_dashboard.index[i],'E_12'] =␣
↪df_scores_pivot_combined.query("Time == @curr_time & School == @curr_school␣
↪& Grade == 12 & Enrolled == 'Yes'").iloc[0,5]
            df_dashboard.loc[df_dashboard.index[i],'E_Total'] =␣
↪df_scores_pivot_combined.query("Time == @curr_time & School == @curr_school␣
↪& Grade == 'Total' & Enrolled == 'Yes'").iloc[0,5]
            df_dashboard.loc[df_dashboard.index[i],'NE_9'] =␣
↪df_scores_pivot_combined.query("Time == @curr_time & School == @curr_school␣
↪& Grade == 9 & Enrolled == 'No'").iloc[0,5]
            df_dashboard.loc[df_dashboard.index[i],'NE_10'] =␣
↪df_scores_pivot_combined.query("Time == @curr_time & School == @curr_school␣
↪& Grade == 10 & Enrolled == 'No'").iloc[0,5]
            df_dashboard.loc[df_dashboard.index[i],'NE_11'] =␣
↪df_scores_pivot_combined.query("Time == @curr_time & School == @curr_school␣
↪& Grade == 11 & Enrolled == 'No'").iloc[0,5]
            df_dashboard.loc[df_dashboard.index[i],'NE_12'] =␣
↪df_scores_pivot_combined.query("Time == @curr_time & School == @curr_school␣
↪& Grade == 12 & Enrolled == 'No'").iloc[0,5]
            df_dashboard.loc[df_dashboard.index[i],'NE_Total'] =␣
↪df_scores_pivot_combined.query("Time == @curr_time & School == @curr_school␣
↪& Grade == 'Total' & Enrolled == 'No'").iloc[0,5]
        else:
```

```
            df_dashboard.loc[df_dashboard.index[i],'E_Total'] =␣
↪df_scores_pivot_combined.query("Time == @curr_time & School == @curr_school␣
↪& Grade == 'Total' & Enrolled == 'Yes'").iloc[0,5]
            df_dashboard.loc[df_dashboard.index[i],'NE_Total'] =␣
↪df_scores_pivot_combined.query("Time == @curr_time & School == @curr_school␣
↪& Grade == 'Total' & Enrolled == 'No'").iloc[0,5]

else:
    raise RuntimeError("Columns not oriented correctly") # https://docs.python.
↪org/3/library/exceptions.html

append_csv('output_tables.csv','Table for Online Dashboard',df_dashboard) #␣
↪Append_csv stores this table below the earlier export of␣
↪df_scores_pivot_combined.

df_dashboard
```

[17]:

	School	Time	E_9	E_10	E_11	E_12	E_Total
0	Westwood	2018_9	56.301887	53.47541	56.323077	52.246154	54.520492
1	Westwood	2019_5	57.264151	64.606557	58.246154	63.015385	60.893443
2	Fair Lake	2018_9	51.171875	55.639344	54.42029	52.2	53.34749
3	Fair Lake	2019_5	59.75	74.836066	63.797101	69.323077	66.783784
4	Eagle	2018_9	53.224138	54.619048	50.764706	53.135593	52.883065
5	Eagle	2019_5	61.534483	72.857143	59.823529	70.372881	66.044355
6	East River	2018_9	59.145455	56.779661	56.368421	52.409091	56.012658
7	East River	2019_5	61.509091	67.864407	59.315789	63.757576	63.189873
8	Bayville	2018_9	56.096154	53.232143	55.157895	54.693548	54.770925
9	Bayville	2019_5	64.038462	70.125	63.982456	70.967742	67.418502
10	Cardinal	2018_9	51.982456	50.796875	55.590164	50.418182	52.227848
11	Cardinal	2019_5	54.280702	62.640625	57.245902	62.072727	59.109705
12	Olive	2018_9	54.411765	54.105263	56.0	55.690141	55.105932
13	Olive	2019_5	55.470588	66.719298	56.842105	66.84507	61.940678
14	Central	2018_9	52.129032	56.910714	53.345455	53.618182	53.95614
15	Central	2019_5	61.403226	74.821429	61.763636	71.636364	67.254386
16	Total	2018_9	NaN	NaN	NaN	NaN	54.085595
17	Total	2019_5	NaN	NaN	NaN	NaN	64.07881

\

	NE_9	NE_10	NE_11	NE_12	NE_Total
0	56.739726	52.493151	54.45	52.896552	54.200758
1	58.876712	55.054795	56.383333	55.189655	56.443182
2	54.71875	54.510638	54.918033	52.803279	54.227468
3	56.8125	56.340426	56.442623	54.393443	55.987124
4	54.888889	54.897059	54.42623	54.337349	54.630282
5	56.486111	56.970588	56.885246	56.301205	56.633803
6	48.777778	55.181818	55.136364	54.636364	53.728507
7	50.733333	58.0	58.106061	56.472727	56.171946
8	54.66	52.050847	56.785714	52.95	54.048889

```
9        58.04  53.542373       58.5  53.916667  55.875556
10   53.851852  54.932203  53.218182  51.527273  53.408072
11   54.981481  56.050847  55.327273  54.527273  55.237668
12   53.101695  53.539683  52.960784  55.314815  53.718062
13   56.118644   56.52381  54.333333  57.833333  56.237885
14   52.190476  53.071429  53.626866  55.196721  53.522267
15   53.698413  55.178571  55.880597  57.409836   55.54251
16         NaN        NaN        NaN        NaN  53.959459
17         NaN        NaN        NaN        NaN  56.038981
```

The values above, having been exported to the output_tables.csv file, could then easily be copied and pasted into the online dashboard, although some sorting would be necessary. This saves a great deal of time over copying and pasting in individual values or trying to get formula references to line up.

1.7 Part 5: Creating grouped line charts for each school

The data is now in two useful formats. However, it is hard to evaluate the Code the Concepts program by looking at the data tables alone. As a result, I will visualize these numbers using a series of line graphs.

First, I will make a set of line graphs for each of the 8 schools. Each of these graphs will show the changes in HGMA scores for students in all four grades and in both enrollment conditions (enrolled and non-enrolled), displaying 16 lines in total. Next, I will create a graph showing just the changes for all enrolled students and all non-enrolled students.

The for loops that create each of these graphs are quite long, and there are surely opportunities to refactor the code to make it leaner. However, I will try to make the code more understandable through comments.

First, I will create a function for plotting (1) a line and (2) separately-colored markers at the end of each line.

[18]:
```python
def plot_line(xpoints, ypoints, color, line_label='_', point_label_1 = '_', point_label_2 = '_' ): # X and Y points need to be in a list format; if they derive from a DataFrame, apply list() to them beforehand. Each list should have only two points.
    # Matplotlib documentation notes that "Specific lines can be excluded from the automatic legend element selection by defining a label starting with an underscore" (https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.legend.html), which is why '_' is used as the default label.
    # print("x values:",xpoints) # for debugging
    # print("y values:",ypoints)
    plt.plot(xpoints, ypoints, linestyle='solid',label=line_label, linewidth=2,color=color)
    plt.plot(xpoints[0], ypoints[0],'og',label=point_label_1) # Plots a data point on the left of the line. See https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html
```

```
    plt.plot(xpoints[1], ypoints[1],'ob',label=point_label_2) # Plots a data␣
↪point on the right of the line.
```

[20]:
```
df_scores_pivot_combined.head() # When creating graphs based on a DataFrame, I␣
↪find it useful to keep a copy of the first few rows of the DataFrame above␣
↪the graph code for reference.
```

[20]:

	Enrolled	Grade	School	Time	count_Score	mean_Score
0	No	9.0	Bayville	2018_9	50	54.660000
1	No	9.0	Bayville	2019_5	50	58.040000
2	Yes	9.0	Bayville	2018_9	52	56.096154
3	Yes	9.0	Bayville	2019_5	52	64.038462
4	No	10.0	Bayville	2018_9	59	52.050847

[21]:
```
if df_scores_pivot_combined.columns[5] != 'mean_Score':
    print("Wut")
```

[22]:
```
data_source = df_scores_pivot_combined # Giving this data_table a shorter␣
↪variable name makes the code more readable and, more importantly, makes it␣
↪easy to replace this DataFrame with a different one.

start_list = [0, 20, 40, 60, 80, 100, 120, 144] # These numbers list the␣
↪zero-indexed row numbers where each set of high school data begins. For␣
↪instance, the data for Bayville starts on row 0, whereas the data for␣
↪Cardinal starts on row 20.

# The following lists will be used for data checking purposes; their purpose␣
↪will become clearer soon.
expected_enrollment_order = []
for a in range(5): # I like to avoid using "i" in more than one for loop within␣
↪a code block, even if the for loops are not nested.
    expected_enrollment_order.append("No")
    expected_enrollment_order.append("No")
    expected_enrollment_order.append("Yes")
    expected_enrollment_order.append("Yes")
    # Returns a list with 20 elements in the following pattern: ['No', 'No',␣
↪'Yes', 'Yes', 'No', 'No', . . . ]. This is the expected pattern of the␣
↪Enrolled values in the data source.

expected_time_order = []
for b in range (10):
    expected_time_order.append("2018_9")
    expected_time_order.append("2019_5")
    # Returns a list with 20 elements in the following pattern: ['2018_9',␣
↪'2019_5', '2018_9', '2019_5', . . . ] This is the expected order of the Time␣
↪values in the data source.
```

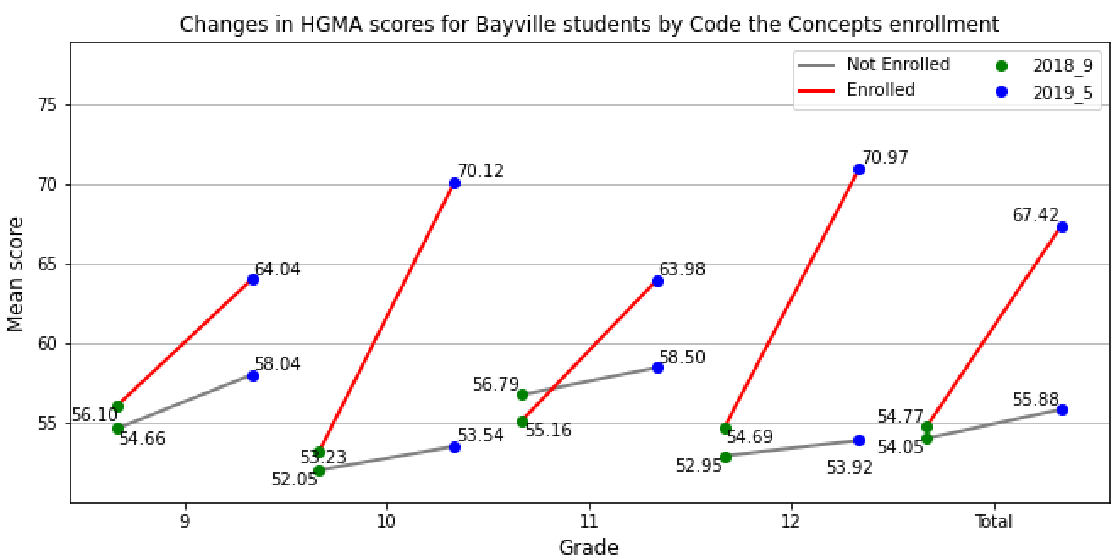
```
expected_grade_order = []
for c in range(9,14,1):
    for d in range(4):
        if c == 13:
            expected_grade_order.append('Total')
        else:
            expected_grade_order.append(c)
        # Returns the following list: [9, 9, 9, 9, 10, 10, 10, 10, 11, 11, 11,
 ↪11, 12, 12, 12, 12, 'Total', 'Total', 'Total', 'Total'].

if data_source.columns[5] != 'mean_Score':
    raise RuntimeError("score data in incorrect column") # Raises an error if
 ↪the 6th column does not contain mean score data, which would result in
 ↪incorrect graphs.

for x in range (0,len(start_list),1): # This for loop iterates over each value
 ↪in start_list, then uses those values as reference points for drawing each
 ↪graph.
    i = start_list[x] # i represents the first of all rows in the DataFrame
 ↪that refer to the current school. Other row values will be stated with
 ↪reference to this row (e.g. i+3 for the 4th row). Each school is represented
 ↪by 20 rows of data, so the code will utilize rows i to i+19 for each school.

    # Data checking:
    # Before running the graphs, it is important to ensure that the data is
 ↪sorted appropriately. Otherwise, the graphs will display the wrong data, and
 ↪this may not be immediately apparent. Therefore, the following lines of code
 ↪employ for loops to verify that the data is in the expected order. If the
 ↪data is not in the proper order, a value error will be raised to prevent the
 ↪function from returning incorrect charts.
    if list(data_source.iloc[i:i+20,0]) != expected_enrollment_order:
        raise RuntimeError("Enrollment data in incorrect order") # See https://
 ↪docs.python.org/3/tutorial/errors.html
    if list(data_source.iloc[i:i+20,3]) != expected_time_order:
        raise RuntimeError("Time data in incorrect order")
    if list(data_source.iloc[i:i+20,1]) != expected_grade_order:
        raise RuntimeError("Grade data in incorrect order")
    school_name = data_source.iloc[i,2] # This variable will also be used after
 ↪data checking is complete.
    for e in range (20):
        if data_source.iloc[i+e,2] != school_name:
            raise RuntimeError("Not all rows to be graphed have the same school
 ↪value")
```

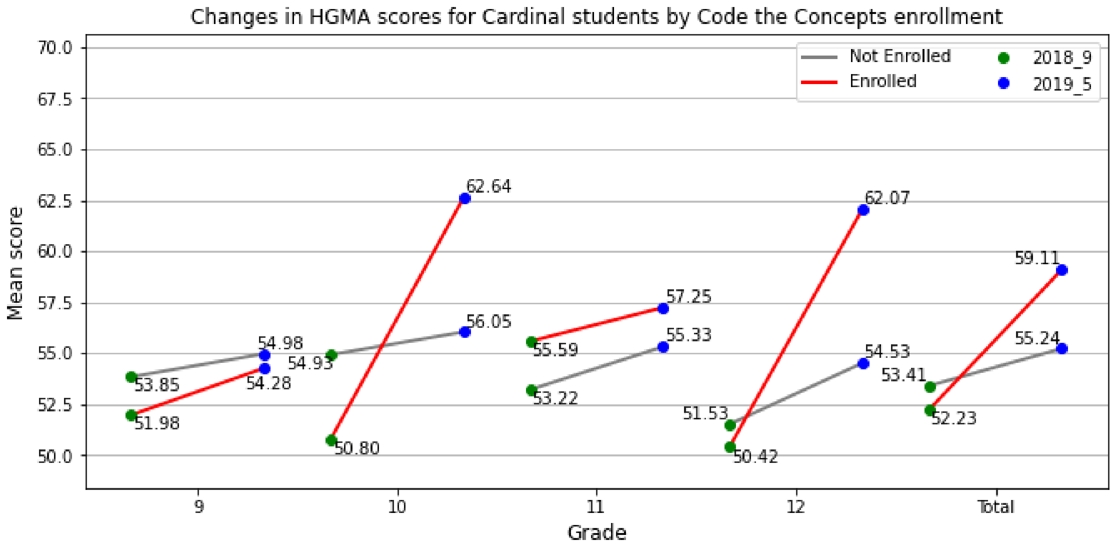
```python
    # If no error is raised, I can conclude that the data is sorted correctly, allowing me to begin the graphing process.

    # The following lines of code calculate the lowest and highest y axis value for the school. These values are then used to determine the y axis range of all graphs that use data exclusively from that school.
    min_val = min(data_source.iloc[i:i+20,5])
    max_val = max(data_source.iloc[i:i+20,5])
    # print("min_val:",min_val) # Was useful for debugging
    # print("max_val:",max_val)

    fig, ax = plt.subplots(figsize=[11,5]) # The figure is made wider to accommodate all the lines each graph will show. The line comes from a very helpful Matplotlib tutorial: https://matplotlib.org/stable/tutorials/introductory/usage.html#sphx-glr-tutorials-introductory-usage-py

    fig.set_facecolor('white')
    plt.ylim(min_val-2,max_val+8) # The extra space provides more room for the legend (to be created below). Without this space, the legend may overlap with line graphs and data labels.
    fig.set_facecolor('white') # Makes area outside chart white as well
    plt.grid(axis = 'y')

    # The following 10 lines of code retrieve mean score data for each school from the DataFrame and store it in list form. "yes" and "no" refers to students who are and are not enrolled in Code the Concepts, respectively. 9, 10, 11, 12, and total refer to the student group (9th grade, 10th grade, etc.--with 'total' referring to all grades for that school as a whole.)
    # Each list stores only 2 data points: the start-of-year and end-of-year mean HGMA scores for a particular grade and enrollment group at a certain school.
    no_9 = list(data_source.iloc[[i,i+1],5])
    yes_9 = list(data_source.iloc[[i+2,i+3],5])
    no_10 = list(data_source.iloc[[i+4,i+5],5])
    yes_10 = list(data_source.iloc[[i+6,i+7],5])
    no_11 = list(data_source.iloc[[i+8,i+9],5])
    yes_11 = list(data_source.iloc[[i+10,i+11],5])
    no_12 = list(data_source.iloc[[i+12,i+13],5])
    yes_12 = list(data_source.iloc[[i+14,i+15],5])
    no_total = list(data_source.iloc[[i+16,i+17],5])
    yes_total = list(data_source.iloc[[i+18,i+19],5])

    dl_1 = [0, 1.5, 3, 4.5, 6] # dl = data_labels. These points, along with those in dl_2, help provide correct x axis spacing for the graphs and axis labels. Each line has a length of 1, and there is a gap of 0.5 in between each line.
```

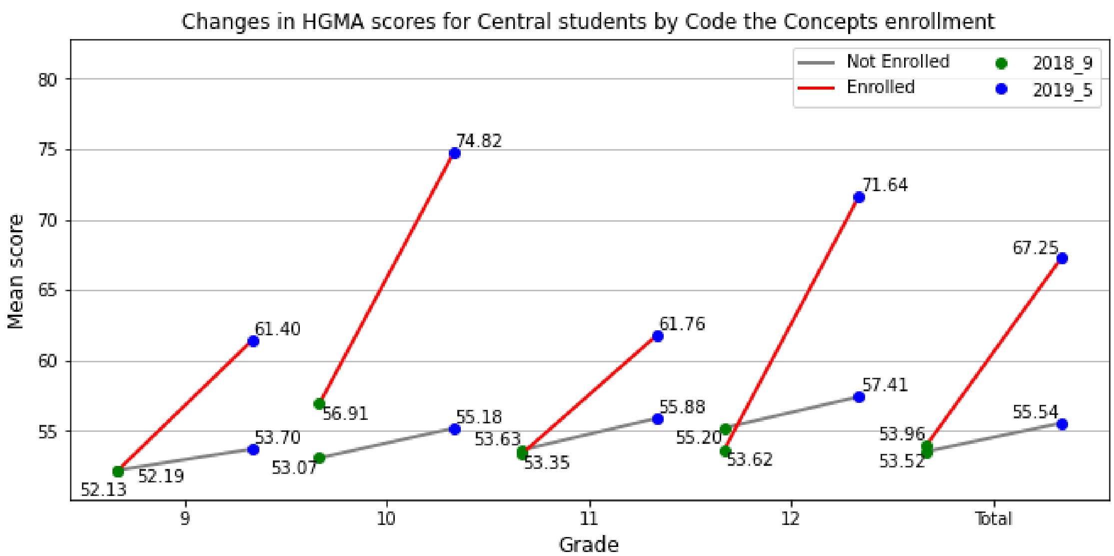
```
    # dl_1 stores the x axis start points for each graph, whereas dl_2 stores the x axis end points.
    dl_2 = [1, 2.5, 4, 5.5, 7]

    # The following lines plot the data points from the 10 lists above.
    plot_line([dl_1[0],dl_2[0]],no_9,'gray',line_label='Not Enrolled')
    plot_line([dl_1[0],dl_2[0]],yes_9,'red',line_label='Enrolled') # In order to make the line labels appear together in the legend, I entered them into the plot_line function before the point labels. Otherwise, the point labels would appear in the middle of the line labels. Only one pair of line and point labels is needed in the legend, as all the other points and lines with the same colors have the same meaning.
    plot_line([dl_1[1],dl_2[1]],no_10,'gray',point_label_1=data_source.iloc[i,3],point_label_2 = data_source.iloc[i+1,3])
    plot_line([dl_1[1],dl_2[1]],yes_10,'red')
    plot_line([dl_1[2],dl_2[2]],no_11,'gray')
    plot_line([dl_1[2],dl_2[2]],yes_11,'red')
    plot_line([dl_1[3],dl_2[3]],no_12,'gray')
    plot_line([dl_1[3],dl_2[3]],yes_12,'red')
    plot_line([dl_1[4],dl_2[4]],no_total,'gray')
    plot_line([dl_1[4],dl_2[4]],yes_total,'red')

    # The following set of code generates data labels for each line. The code also uses the adjust_text library (https://github.com/Phlya/adjustText) to prevent data labels from overlapping with the line graphs and with each other.

    yoffset = 0 # Controls how far below the points the labels are by default. Not needed now that adjust_text has been added.
    x_avoid_list = [] # This list, along with y_avoid_list, will store a series of coordinates that adjust_text will move data labels away from.
    y_avoid_list = []
    data_label_list = []
    for j in range (0,5,1): # Covers each point in dl_1 and dl_2. Each value of j will cover two lines in the graph with the same x coordinates.
        # There are 20 data points for each set of schools, so to access each of these points, I will multiply j by 4, then add another value to that product.
        # I could have added in another for loop to reduce the number of individual plt.text functions to 2, but decided that the reduction in code length would not be worth the increase in complexity.
        data_label_1 = plt.text(dl_1[j],data_source.iloc[i+j*4,5]-yoffset,'{:.2f}'.format(data_source.iloc[i+j*4,5]),ha='center') # Plots a percentage label next to the left end of the line using the points in dl_1 for x values and the mean scores from the DataFrame for y values.
```

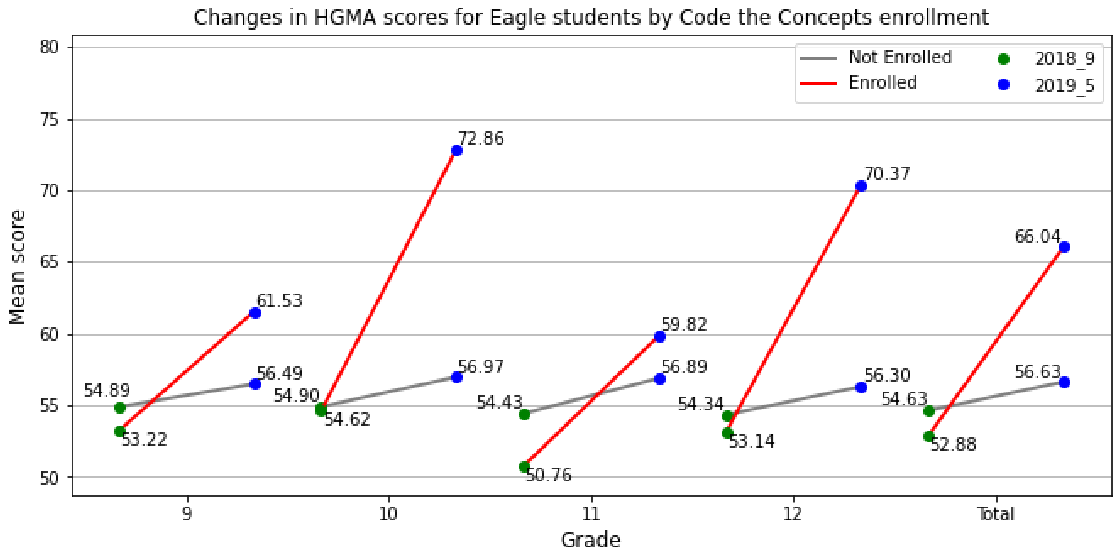
```
        data_label_list.append(data_label_1) # Appending each data point to a list will allow adjust_text to successfully adjust those points' positions.
        data_label_2 = plt.text(dl_2[j],data_source.iloc[i+j*4+1,5]-yoffset,'{:.2f}'.format(data_source.iloc[i+j*4+1,5]),ha='center') # This data label is plotted near the right end of the line (using dl_2 instead of dl_1].
        data_label_list.append(data_label_2)
        data_label_3 = plt.text(dl_1[j],data_source.iloc[i+j*4+2,5]-yoffset,'{:.2f}'.format(data_source.iloc[i+j*4+2,5]),ha='center')
        data_label_list.append(data_label_3)
        data_label_4 = plt.text(dl_2[j],data_source.iloc[i+j*4+3,5]-yoffset,'{:.2f}'.format(data_source.iloc[i+j*4+3,5]),ha='center')
        data_label_list.append(data_label_4)

        # adjust_text knows how to move data labels away from one another. However, in order to keep the data labels from overlapping with the line graphs, adjust_text needs to know where those lines exist. Therefore, the following lines of code create a series of points that overlap with those line graphs. In avoiding those points, adjust_text will keep data labels from overlapping with the lines. This code is based on https://adjusttext.readthedocs.io/en/latest/Examples.html, but the implementation is different
        x_avoid = np.linspace(0,1,21) # Creates 20 values between (and including) 0 and 1, as each line has an x length of 1
        y_avoid_1 = data_source.iloc[i+j*4,5] + x_avoid*(data_source.iloc[i+j*4+1,5] - data_source.iloc[i+j*4,5]) # This line creates 20 y values that match the y values on one of the line graphs. The code equates to (list of y points = leftmost y value + (distance between 0 and 1)*(rightmost y value - leftmost y value). It is similar to the slope-intercept equation y = mx + b.
        y_avoid_2 = data_source.iloc[i+j*4+2,5] + x_avoid*(data_source.iloc[i+j*4+3,5] - data_source.iloc[i+j*4+2,5]) # This line creates 20 y values for the other line graph.
        x_avoid += dl_1[j] # Shifts the points in x_avoid so that they line up with their corresponding line. (Only the first line actually starts at 0; the other lines are offset by dl_1[j].
        # plt.plot(x_avoid, y_avoid_1,color='blue') # These plot() functions are very useful for ensuring that y_avoid_1 and y_avoid_2 line up with the corresponding data lines, thus ensuring that the data labels and actual data lines will not overlap.
        # plt.plot(x_avoid,y_avoid_2,color='black')
        x_avoid_list.extend(x_avoid) # The following lines store the x_avoid and y_avoid points in corresponding lists. The adjust_text function will use these lists to determine which coordinates to move text away from.
        y_avoid_list.extend(y_avoid_1)
        x_avoid_list.extend(x_avoid)
        y_avoid_list.extend(y_avoid_2)
```

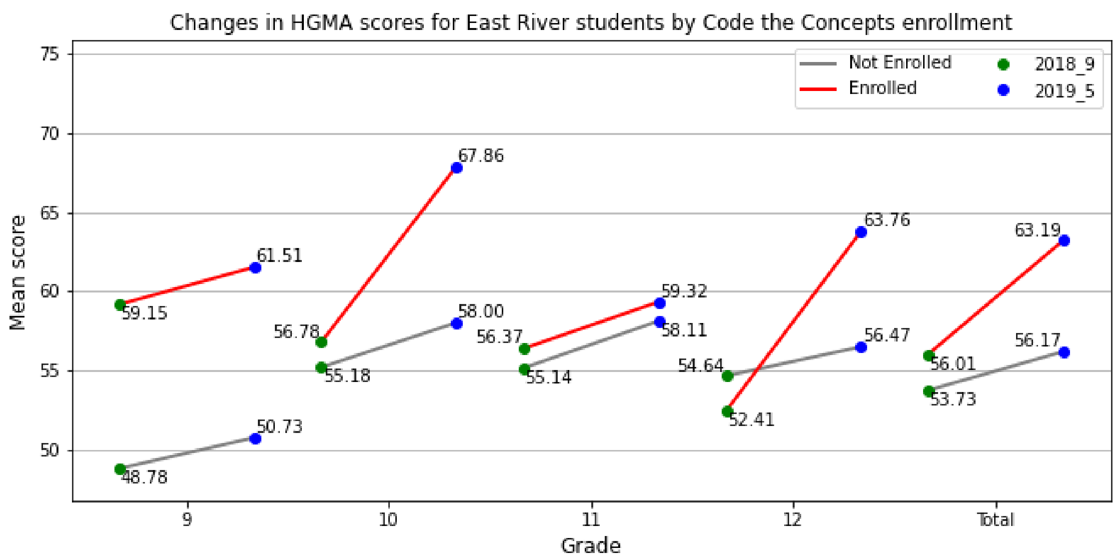
```
    adjust_text(data_label_list, x=x_avoid_list, y=y_avoid_list) # This function shifts the data points away from the line graphs while also keeping them away from each other. It is called once all data labels and x/y coordinates to avoid have been entered.


    title_string = "Changes in HGMA scores for "+school_name+" students by Code the Concepts enrollment"
    plt.title(title_string)


    xtick_list = []
    for m in range(len(dl_1)):
        xtick_list.append(np.mean([dl_1[m], dl_2[m]])) # These ticks are placed in the middle of each set of line graphs in order to make the graph labels align correctly.
    ax.set_xticks(xtick_list)
    ax.set_xticklabels(['9', '10', '11', '12', 'Total'])

    ax.set_ylabel('Mean score',fontsize=12)
    ax.set_xlabel('Grade',fontsize=12)
    plt.legend(loc='best',ncol=2) # ncol=2 creates two columns for the legend, which limits the vertical space it takes up. This helps prevent the legend from overlapping with graph data.

    file_string = 'graphs\\'+school_name+'.png' # Each graph is saved in the same dedicated folder.
    plt.savefig(file_string,dpi=400) # Increasing the DPI increases the visual quality of each graph, which will be helpful for presentation settings.
    plt.show()
```

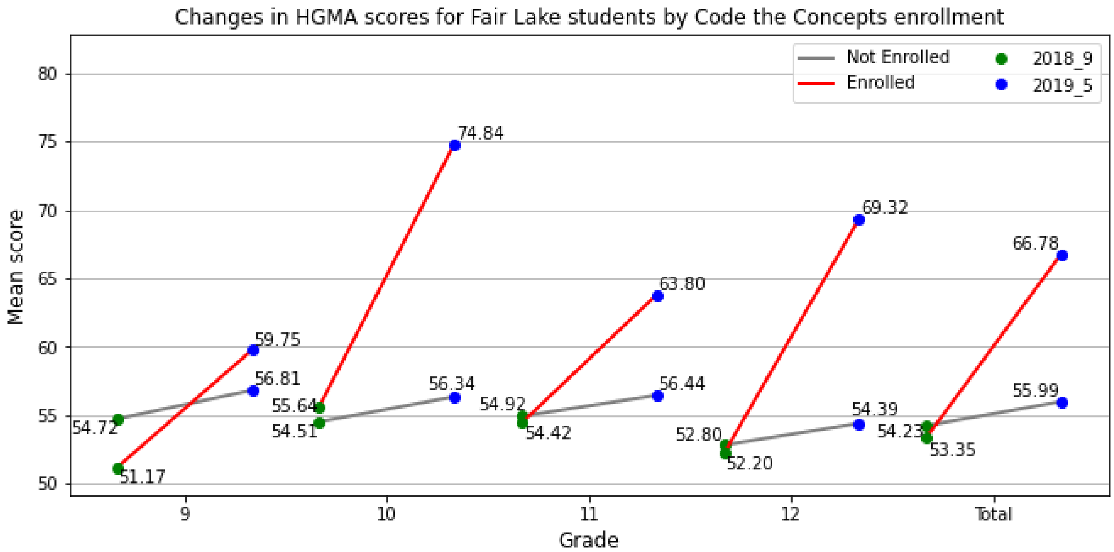


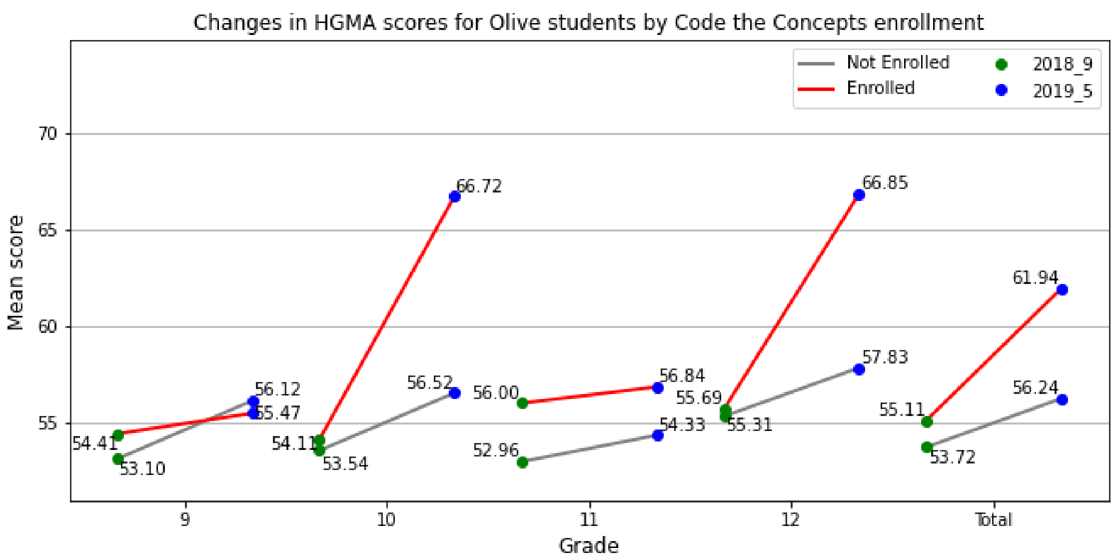

Changes in HGMA scores for Cardinal students by Code the Concepts enrollment
Not Enrolled
Enrolled
2018_9
2019_5
Mean score
Grade
9
10
11
12
Total
70.0
67.5
65.0
62.5
60.0
57.5
55.0
52.5
50.0
53.85
54.98
51.98
54.28
54.93
56.05
50.80
62.64
55.59
57.25
53.22
55.33
51.53
54.53
50.42
62.07
53.41
55.24
52.23
59.11

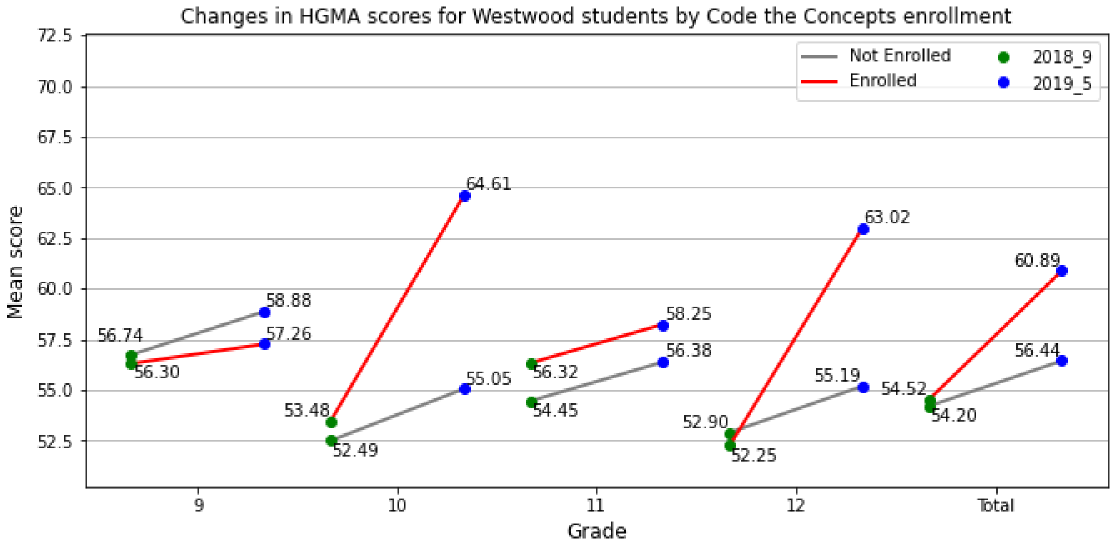

Changes in HGMA scores for Central students by Code the Concepts enrollment
Not Enrolled
Enrolled
2018_9
2019_5
Mean score
Grade
9
10
11
12
Total
80
75
70
65
60
55
52.13
52.19
53.70
61.40
53.07
55.18
56.91
74.82
53.63
53.35
55.88
61.76
55.20
53.62
57.41
71.64
53.96
53.52
55.54
67.25


Changes in HGMA scores for Eagle students by Code the Concepts enrollment
Mean score
Grade
Not Enrolled
Enrolled
2018_9
2019_5
9
10
11
12
Total
54.89
53.22
61.53
56.49
54.90
54.62
72.86
56.97
54.43
50.76
59.82
56.89
54.34
53.14
70.37
56.30
54.63
52.88
66.04
56.63


Changes in HGMA scores for East River students by Code the Concepts enrollment
Mean score
Grade
Not Enrolled
Enrolled
2018_9
2019_5
9
10
11
12
Total
59.15
61.51
48.78
50.73
56.78
55.18
67.86
58.00
56.37
55.14
59.32
58.11
54.64
52.41
63.76
56.47
56.01
53.73
63.19
56.17


Changes in HGMA scores for Fair Lake students by Code the Concepts enrollment
Not Enrolled
Enrolled
2018_9
2019_5
Mean score
Grade
9
10
11
12
Total
54.72
51.17
59.75
56.81
55.64
54.51
74.84
56.34
54.92
54.42
63.80
56.44
52.80
52.20
69.32
54.39
54.23
53.35
66.78
55.99


Changes in HGMA scores for Olive students by Code the Concepts enrollment
Not Enrolled
Enrolled
2018_9
2019_5
Mean score
Grade
9
10
11
12
Total
54.41
53.10
56.12
55.47
54.11
53.54
66.72
56.52
56.00
52.96
56.84
54.33
55.69
55.31
66.85
57.83
55.11
53.72
61.94
56.24



These charts demonstrate that students enrolled in Code the Concepts experienced higher gains in HGMA scores than their peers in certain grades (e.g. 10 and 12) and certain schools (Fair Lake, Eagle, Central, and Bayville). A regression model and/or T-tests would provide more insight into the magnitude of these differences and whether they are statistically significant.

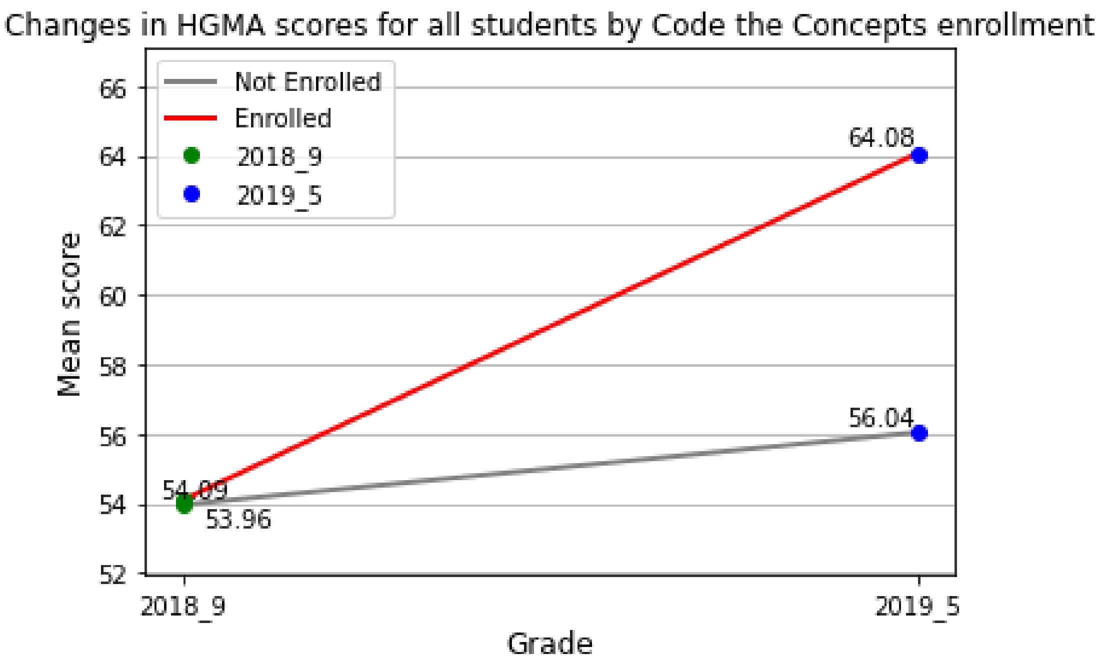
1.8 Creating similar line charts for the Total rows

Since the Total section of the DataFrame comprises only 4 rows (rather than 20), the above code block will not work correctly. Therefore, a similar but separate code block was created to graph the Total rows.

[23]:
```
data_source = df_scores_pivot_combined

i = 140 # Row 140 marks the start of the 4 'Total' rows in the DataFrame.
school_name = data_source.iloc[i,2]
fig, ax = plt.subplots() # No need to make these plots wider
fig.set_facecolor('white')
fig.set_facecolor('white')
plt.grid(axis = 'y')
xlabels = ['2018-09', '2019-05']

min_val = min(data_source.iloc[i:i+4,5])
max_val = max(data_source.iloc[i:i+4,5])
# print("min_val:",min_val)
# print("max_val:",max_val)
plt.ylim(min_val-2,max_val+3)

no_line = list(data_source.iloc[[i,i+1],5])
yes_line = list(data_source.iloc[[i+2,i+3],5])
```

```
dl_1 = 0
dl_2 = 1

plot_line([dl_1,dl_2],no_line,'gray',line_label='Not Enrolled')
plot_line([dl_1,dl_2],yes_line,'red',line_label='Enrolled',point_label_1=data_source.
↪iloc[i,3],point_label_2 = data_source.iloc[i+1,3])

yoffset = 0
x_avoid_list = []
y_avoid_list = []
data_label_list = []
data_label_1 = plt.text(dl_1,data_source.iloc[i,5]-yoffset,'{:.2f}'.
↪format(data_source.iloc[i,5]),ha='center')
data_label_list.append(data_label_1)
data_label_2 = plt.text(dl_2,data_source.iloc[i+1,5]-yoffset,'{:.2f}'.
↪format(data_source.iloc[i+1,5]),ha='center')
data_label_list.append(data_label_2)
data_label_3 = plt.text(dl_1,data_source.iloc[i+2,5]-yoffset,'{:.2f}'.
↪format(data_source.iloc[i+2,5]),ha='center')
data_label_list.append(data_label_3)
data_label_4 = plt.text(dl_2,data_source.iloc[i+3,5]-yoffset,'{:.2f}'.
↪format(data_source.iloc[i+3,5]),ha='center')
data_label_list.append(data_label_4)
x_avoid = np.linspace(0,1,21)
y_avoid_1 = data_source.iloc[i,5] + x_avoid*(data_source.iloc[i+1,5] -␣
↪data_source.iloc[i,5])
y_avoid_2 = data_source.iloc[i+2,5] + x_avoid*(data_source.iloc[i+3,5] -␣
↪data_source.iloc[i+2,5])
x_avoid += dl_1 #
# plt.plot(x_avoid, y_avoid_1,color='blue')
# plt.plot(x_avoid,y_avoid_2,color='black')
x_avoid_list.extend(x_avoid)
y_avoid_list.extend(y_avoid_1)
x_avoid_list.extend(x_avoid)
y_avoid_list.extend(y_avoid_2)
adjust_text(data_label_list, x=x_avoid_list, y=y_avoid_list)
# print("x_avoid_list",x_avoid_list)
# print("y_avoid_list",y_avoid_list)
# plt.plot(x_avoid_list,y_avoid_list,color='black') # Another way of confirming␣
↪that adjust_text is shifting each text block away from the correct points
plt.title("Changes in HGMA scores for all students by Code the Concepts␣
↪enrollment")
xtick_list = [dl_1, dl_2]
ax.set_xticks(xtick_list)
ax.set_xticklabels([data_source.iloc[i,3],data_source.iloc[i+1,3]])
```

```
ax.set_ylabel('Mean score',fontsize=12)
ax.set_xlabel('Grade',fontsize=12)

plt.legend()

file_string = 'graphs\\'+school_name+'.png'
plt.savefig(file_string,dpi=400)
plt.show()
```



[24]:
```
end_time = time.time()

run_time = end_time - start_time
run_minutes = int(run_time // 60)
run_seconds = run_time % 60
print("Total run time:",'{:.2f}'.format(run_time),"second(s)␣
↪("+str(run_minutes),"minute(s) and",'{:.2f}'.
↪format(run_seconds),"second(s))") # Only produces an accurate result when␣
↪the program is run nonstop from start to finish
```

```
Total run time: 146.36 second(s) (2 minute(s) and 26.36 second(s))
```

2 Conclusion

I hope this tutorial program will prove helpful for you in your own journeys. This code covered basic descriptive statistics, but being able to quickly and accurately visualize these statistics is a valuable skill–and one that Python is well suited for. Happy coding!

```
[ ]:
```