# spreadsheet_ops

April 11, 2024

## 1 Python for Nonprofits Part x: Spreadsheet Operations

By Kenneth Burchfiel



This notebook provides an interview to executing spreadsheet operations in PythonThe benefit of performing these tasks in Python (rather than Excel, Google Sheets, or another spreadsheet program) is that, once you have these tasks scripted, you can quickly rerun these tasks whenever the original data gets updated*. You can even have your computer run the script on a daily or hourly basis, thus saving you from busywork and freeing up your time for more interesting tasks.

For example, suppose leaders at your school network would like to see an overview of the network's enrollment each day. One way to accomplish this task would be to retrieve data from your database each day, paste it into Excel or Google Sheets, pivot the data, and then share the output with them. However, you could accomplish these same steps much more quickly in Python. This notebook will show you how!

* There are certainly ways to automate Excel tasks as well (e.g. using Visual Basic). I don't have any experience with Visual Basic, so I'm not the best person to compare these two tools; however, I have no doubt tat learning it would take some time, and given Python's versatility and power, I would recommend applying that time to learning Python instead. (You can get an estimate of the world's interest in Python versus Visual Basic by checking out the TIOBE index.)

Importing the libaries we'll need for this project:

[ ]:
```
import time
start_time = time.time() # Allows the program's runtime to be measured
import pandas as pd
pd.set_option('display.max_rows', 30) # This update reduces the number
# of rows that the output will show to a maximum of 30, thus making
# the notebook a bit more readable.
import sqlalchemy
import numpy as np
```

# 2 Part 1: Importing data

## 2.1 Connecting to our SQLite database

This local SQLite database was created using the database_generator.ipnyb code found in supplemental/db_generator. The steps for connecting to an online database are quite similar; for guidance on this process, visit the app_functions_and_variables.py file within my Dash School Dashboard project.

[ ]:
```
engine = sqlalchemy.create_engine(
'sqlite:///'+'../data/network_database.db')
# Based on:
#  https://docs.sqlalchemy.org/en/13/dialects/sqlite.html#connect-strings

engine
```

[ ]:
```
Engine(sqlite:///../data/network_database.db)
```

### 2.1.1 Reviewing a list of all tables within our database

[ ]:
```
pd.read_sql("Select name from sqlite_schema", con = engine)
```

[ ]:
```
              name
0  curr_enrollment
1     test_results
2    grad_outcomes
```

## 2.2 Retrieving all data from the curr_enrollment table and reading it into a Pandas DataFrame

DataFrames are essentially spreadsheets that can be manipulated and summarized within Python. It's easy to convert them to .csv or .xlsx files (or vice versa). Lots of data analysis tasks within Python involve DataFrames, so they will show up very often within Python for Nonprofits.

[ ]:
```
df_curr_enrollment = pd.read_sql(
    "Select * from curr_enrollment", con = engine)

# 'Select * from curr_enrollment' is a SQL command that imports all
# data from the current_enrollment table within our SQLite database.
# If we wanted to import only a few columns, we could replace the *
# with those specific column names (e.g. 'Student_ID', 'School', 'Grade');
# or, if we wanted to import data for just one school, we could enter:
# "Select * from curr_enrollment where School = 'CA'".

# SQL is a language of its own and not the main focus of Python for Nonprofits,
# but Pandas' read_sql and to_sql functions will allow you to perform
# many SQL-related tasks with only an elementary understanding of how it works.
```

```
# Some of the Python code in this notebook could be replaced with SQL
# code, which could actually speed up the program's runtime (since you wouldn't
# need to import as much data within your initial SQL query), but this
# notebook's purpose is to demonstrate how to use Pandas, not SQL.

df_curr_enrollment
```

[ ]:

| | Student_ID | First_Name | Last_Name | Full_School_Name | School | Grade | Gender | \ |
|---|---|---|---|---|---|---|---|---|
| 0 | 42646 | Jeanne | Bell | Chestnut Academy | CA | 1 | Female | |
| 1 | 41632 | Theodore | Brown | Chestnut Academy | CA | 1 | Male | |
| 2 | 42586 | Lynn | Callahan | Chestnut Academy | CA | 1 | Female | |
| 3 | 40108 | Edward | Carrillo | Chestnut Academy | CA | 1 | Male | |
| 4 | 43600 | Sara | Carson | Chestnut Academy | CA | 1 | Female | |
| … | … | … | … | … | … | … | … | |
| 3995 | 42085 | Jessica | Williams | Sycamore Academy | SA | K | Female | |
| 3996 | 42179 | Kim | Williams | Sycamore Academy | SA | K | Female | |
| 3997 | 41677 | Micheal | Williams | Sycamore Academy | SA | K | Male | |
| 3998 | 42238 | Susan | Williams | Sycamore Academy | SA | K | Female | |
| 3999 | 43527 | Jessica | Woods | Sycamore Academy | SA | K | Female | |

| | Race | Ethnicity | Street | \ |
|---|---|---|---|---|
| 0 | African American | Non-Hispanic | 200 N PINECREST LN | |
| 1 | African American | Non-Hispanic | 1301 Whitehead Rd | |
| 2 | African American | Hispanic | 11230 WAPLES MILL RD STE 100 | |
| 3 | White | Hispanic | 901 Rose Hill Drive | |
| 4 | Asian | Hispanic | 1825 Wenonah Avenue | |
| … | … | … | … | |
| 3995 | American Indian | Non-Hispanic | 100 Cedarmeade Ave | |
| 3996 | American Indian | Non-Hispanic | 7719D FULLERTON RD | |
| 3997 | White | Hispanic | 361 Walnut St | |
| 3998 | African American | Non-Hispanic | 5000 W NORFOLK RD | |
| 3999 | African American | Non-Hispanic | 705 Waterloo Rd | |

| | City | State | Zip | Lat | Lon | \ |
|---|---|---|---|---|---|---|
| 0 | BRISTOL | VA | 24201 | 36.615974 | -82.190919 | |
| 1 | Richmond | VA | 23225 | 37.489169 | -77.508759 | |
| 2 | FAIRFAX | VA | 22030 | 38.858060 | -77.334451 | |
| 3 | Charlottesville | VA | 22903 | 38.039900 | -78.486600 | |
| 4 | Pearisburg | VA | 24134 | 37.327800 | -80.704500 | |
| … | … | … | … | … | … | |
| 3995 | Winchester | VA | 22601 | 39.151300 | -78.180800 | |
| 3996 | SPRINGFIELD | VA | 22153 | 38.741592 | -77.211129 | |
| 3997 | Warsaw | VA | 22572 | 37.945528 | -76.744655 | |
| 3998 | PORTSMOUTH | VA | 23703 | 36.869169 | -76.378579 | |
| 3999 | Warrenton | VA | 20186 | 38.720000 | -77.814700 | |

| | Address | Students | \ |
|---|---|---|---|

```
0               200 N PINECREST LN, BRISTOL, VA 24201         1
1               1301 Whitehead Rd, Richmond, VA 23225         1
2     11230 WAPLES MILL RD STE 100, FAIRFAX, VA 22030         1
3      901 Rose Hill Drive, Charlottesville, VA 22903         1
4           1825 Wenonah Avenue, Pearisburg, VA 24134         1
…                                                   …       …
3995         100 Cedarmeade Ave, Winchester, VA 22601         1
3996        7719D FULLERTON RD, SPRINGFIELD, VA 22153         1
3997                  361 Walnut St, Warsaw, VA 22572         1
3998          5000 W NORFOLK RD, PORTSMOUTH, VA 23703         1
3999             705 Waterloo Rd, Warrenton, VA 20186         1

      Grade_for_Sorting
0                     1
1                     1
2                     1
3                     1
4                     1
…                   …
3995                  0
3996                  0
3997                  0
3998                  0
3999                  0

[4000 rows x 18 columns]
```

### 2.3 Part 2: Pivoting DataFrames

Let's say that you want to determine the number of students in each grade at each school. You can do so easily using the pivot_table function within Pandas. In the following code, 'index' represents the pairs of variables for which you want to analyze a given metric; 'values' shows the items that you wish to analyze, and 'aggfunc' shows how you wish to analyze them. In this case, we want to count the number of students belonging to each school-grade pair, so we'll pass ['School', 'Grade_for_Sorting', 'Grade'] to index; 'Students' (a column that contains the value '1' for each student); and 'sum' to aggfunc.

('Grade_for_Sorting' is added before 'Grade' so that the pivot output will sort grades in the correct ascending order (e.g. 'K', '1', '2' . . . '11', '12'). Because the 'Grade' column uses an object data type, its default sort order would be alphabetical (e.g. '1', '11', '12' . . . '8', '9', 'K'), which certainly isn't what we want. Therefore, we'll sort the data by a column that stores all grades as integers *and* sets K equal to 0, thus eliminating the need to attempt an alphabetical sort.)

[ ]:
```
df_school_grade_pivot = df_curr_enrollment.pivot_table(
    index = ['School', 'Grade_for_Sorting', 'Grade'],
    values = 'Students', aggfunc = 'sum')

# Here's what the first 15 rows of the DataFrame look like:
```

```
df_school_grade_pivot.head(15) # .head(15) allows us to view the first
# 15 rows of data; similarly, .tail(5) would let us see the final 5 rows.
```

[ ]:
```
                                Students
School Grade_for_Sorting Grade
CA     0                 K           90
       1                 1           71
       2                 2           76
       3                 3           61
       4                 4           85
       5                 5           66
       6                 6           74
       7                 7           65
       8                 8           75
       9                 9           77
       10                10          79
       11                11          75
       12                12          70
DA     0                 K           93
       1                 1           56
```

Note that 'CA' and 'DA' appear only once (at the start of their respective sections of the pivot table). My preference is to use reset_index() to add a numerical index back into the pivot table:

[ ]:
```
df_school_grade_pivot.reset_index(inplace=True)
df_school_grade_pivot
```

[ ]:
```
   School  Grade_for_Sorting Grade  Students
0      CA                  0     K        90
1      CA                  1     1        71
2      CA                  2     2        76
3      CA                  3     3        61
4      CA                  4     4        85
..    …                  …   …       …
47     SA                  8     8        68
48     SA                  9     9        71
49     SA                 10    10        86
50     SA                 11    11        75
51     SA                 12    12        83

[52 rows x 4 columns]
```

To determine schoolwide student counts, we can pass 'School' as our 'index' argument:

[ ]:
```
# The following pivot table isn't saved to a variable, so its output
# won't be accessible in later parts of the code. This approach works fine
# if you just need to check a set of values or test out a potential change
```

```
# to a DataFrame.

df_curr_enrollment.pivot_table(
    index = 'School',
    values = 'Students', aggfunc = 'sum').reset_index()
```

[ ]:

| | School | Students |
|---|---|---|
| 0 | CA | 964 |
| 1 | DA | 977 |
| 2 | HA | 1038 |
| 3 | SA | 1021 |

We don't need a pivot table in order to determine our network-wide enrollment; instead, we can just use Series.sum():

(Series is the name Pandas uses for a column within a DataFrame. Series can also be standalone objects, but you'll often find them within larger tables.)

[ ]:
```
df_curr_enrollment['Students'].sum()
```

[ ]:
```
4000
```

### 2.3.1 Filtering the DataFrame with query()

To view data for only one school in particular, we can use the query() method within Pandas:

[ ]:
```
df_ca_school_grade_pivot = df_school_grade_pivot.query("School == 'CA'").copy()
# The inclusion of .copy() prevents operations applied to this new
# DataFrame from *also* affecting the source DataFrame.
df_ca_school_grade_pivot
```

[ ]:

| | School | Grade_for_Sorting | Grade | Students |
|---|---|---|---|---|
| 0 | CA | 0 | K | 90 |
| 1 | CA | 1 | 1 | 71 |
| 2 | CA | 2 | 2 | 76 |
| 3 | CA | 3 | 3 | 61 |
| 4 | CA | 4 | 4 | 85 |
| 5 | CA | 5 | 5 | 66 |
| 6 | CA | 6 | 6 | 74 |
| 7 | CA | 7 | 7 | 65 |
| 8 | CA | 8 | 8 | 75 |
| 9 | CA | 9 | 9 | 77 |
| 10 | CA | 10 | 10 | 79 |
| 11 | CA | 11 | 11 | 75 |
| 12 | CA | 12 | 12 | 70 |

We can also use query() to identify classes with enrollment below a particular threshold (such as, say, 65 students):

[ ]:
```
df_school_grade_pivot.query("Students < 65")
```

[ ]:

| | School | Grade_for_Sorting | Grade | Students |
|---|---|---|---|---|
| 3 | CA | 3 | 3 | 61 |
| 14 | DA | 1 | 1 | 56 |
| 22 | DA | 9 | 9 | 63 |

Passing a variable name to a query() expression is very helpful and can be accomplished via the @ symbol:

[ ]:
```
overenrolled_threshold = 80
df_school_grade_pivot.query("Students >= @overenrolled_threshold")
```

[ ]:

| | School | Grade_for_Sorting | Grade | Students |
|---|---|---|---|---|
| 0 | CA | 0 | K | 90 |
| 4 | CA | 4 | 4 | 85 |
| 13 | DA | 0 | K | 93 |
| 18 | DA | 5 | 5 | 89 |
| 23 | DA | 10 | 10 | 90 |
| 27 | HA | 1 | 1 | 84 |
| 28 | HA | 2 | 2 | 93 |
| 33 | HA | 7 | 7 | 93 |
| 34 | HA | 8 | 8 | 87 |
| 35 | HA | 9 | 9 | 84 |
| 36 | HA | 10 | 10 | 82 |
| 40 | SA | 1 | 1 | 85 |
| 41 | SA | 2 | 2 | 80 |
| 42 | SA | 3 | 3 | 97 |
| 43 | SA | 4 | 4 | 89 |
| 49 | SA | 10 | 10 | 86 |
| 51 | SA | 12 | 12 | 83 |

You can also use query() to select rows whose values are found in a given list:

[ ]:
```
hs_grades = ['9', '10', '11', '12']
df_school_grade_pivot.query("Grade in @hs_grades")
# An alternative option would be:
# df_school_grade_pivot.query("Grade in ['9', '10', '11', '12']")
```

[ ]:

| | School | Grade_for_Sorting | Grade | Students |
|---|---|---|---|---|
| 9 | CA | 9 | 9 | 77 |
| 10 | CA | 10 | 10 | 79 |
| 11 | CA | 11 | 11 | 75 |
| 12 | CA | 12 | 12 | 70 |
| 22 | DA | 9 | 9 | 63 |
| 23 | DA | 10 | 10 | 90 |
| 24 | DA | 11 | 11 | 70 |
| 25 | DA | 12 | 12 | 75 |

```
35     HA                   9     9        84
36     HA                  10    10        82
37     HA                  11    11        65
38     HA                  12    12        70
48     SA                   9     9        71
49     SA                  10    10        86
50     SA                  11    11        75
51     SA                  12    12        83
```

query() is a very powerful tool for filtering DataFrames. The official Pandas documentation offers more information on this method.

### 2.4 Adding new columns using np.where(), np.select(), and map()

We'll now try out three powerful tools for analyzing and working with DataFrames: np.where(), np.select(), and map().

First, let's add in an 'Underenrolled' flag that will read 1 if a grade has fewer than 65 students and 0 otherwise. We can accomplish this via np.where():

[ ]:
```
df_school_grade_pivot['Underenrolled'] = np.where(
    df_school_grade_pivot['Students'] < 65, 1, 0)
# The first argument (df_school_grade_pivot['Students'] < 65) is the condition
# being evaluated by np.where(), and the second and third arguments (1 and 0)
# show what values to add to the column if the condition is met or not met,
# respectively.

# Note that we don't need a for loop to accomplish this operation! In general,
# the less you use for loops within DataFrame operations, the faster your code
# will run.

df_school_grade_pivot
```

[ ]:
```
   School  Grade_for_Sorting Grade  Students  Underenrolled
0      CA                  0     K        90              0
1      CA                  1     1        71              0
2      CA                  2     2        76              0
3      CA                  3     3        61              1
4      CA                  4     4        85              0
..    …                  …   …       …            …
47     SA                  8     8        68              0
48     SA                  9     9        71              0
49     SA                 10    10        86              0
50     SA                 11    11        75              0
51     SA                 12    12        83              0

[52 rows x 5 columns]
```

np.where() works great for assigning rows to one of two groups (like overenrolled or not overen-

rolled). However, you'll sometimes need to assign columns to more than one group, in which case np.select() proves very handy.

In the following cell, *condlist* describes the 'overenrolled' and 'underenrolled' conditions, whereas 'choicelist' defines how to designate these conditions within the DataFrame. The final argument within np.select(), 'Normal', explains how to label rows that don't fall into any of the categories in condlist.

You can add as many categories to condlist as you want, but be careful to keep the values in both condlist and choicelist in the correct order. (The first condition in condlist will trigger the first value in choicelist; the second condlist condition will trigger the second choicelist value; and so on.)

[ ]:
```
condlist = [df_school_grade_pivot['Students'] < 65,
            df_school_grade_pivot['Students'] >= overenrolled_threshold]
choicelist = ['Underenrolled', 'Overenrolled']
df_school_grade_pivot['Enrollment_Category'] = np.select(condlist, choicelist,␣
 ↪'Normal')
df_school_grade_pivot
```

[ ]:
```
   School  Grade_for_Sorting Grade  Students  Underenrolled  \
0      CA                  0     K        90              0
1      CA                  1     1        71              0
2      CA                  2     2        76              0
3      CA                  3     3        61              1
4      CA                  4     4        85              0
..    …                  …   …       …            …
47     SA                  8     8        68              0
48     SA                  9     9        71              0
49     SA                 10    10        86              0
50     SA                 11    11        75              0
51     SA                 12    12        83              0

   Enrollment_Category
0         Overenrolled
1               Normal
2               Normal
3        Underenrolled
4         Overenrolled
..                 …
47              Normal
48              Normal
49        Overenrolled
50              Normal
51        Overenrolled

[52 rows x 6 columns]
```

Also note that, once np.select() finds a match for a row within condlist, it will assign that condition's corresponding choicelist item to the row. Therefore, it's OK if your condlist items aren't mutually

exclusive as long as you're careful with how they're ordered.

As an example of this behavior, let's try using np.select() to assign Elementary School (ES), Middle School (MS), and High School (HS) designations to each grade. **The following approach produces the wrong results–can you figure out why?**

[ ]:
```
# In our fictional district, elementary school runs from grades K to 6;
# middle school encompasses grades 7 and 8; and high school includes grades
# 9 to 12.

# Note: this is an example of how *not* to use np.select()!
condlist = [df_school_grade_pivot['Grade_for_Sorting'] >= 7,
            df_school_grade_pivot['Grade_for_Sorting'] >= 9]
choicelist = ['MS', 'HS']
df_school_grade_pivot['Stage'] = np.select(condlist, choicelist, 'ES')
df_school_grade_pivot['Stage'].value_counts()
```

[ ]:
```
Stage
ES    28
MS    24
Name: count, dtype: int64
```

value_counts() allows you to quickly see the distribution of values within a particular column. Note that we have only ES and MS values–HS is nowhere to be seen. Why is that? Because all high school grades met the first condition (a grade greater than or equal to 7) and were thus categorized as MS.

In order to use np.select to accurately assign grades, we can revise our code as follows:

[ ]:
```
# Corrected code:

condlist = [df_school_grade_pivot['Grade_for_Sorting'] <= 6,
            df_school_grade_pivot['Grade_for_Sorting'] <= 8]
choicelist = ['ES', 'MS']
df_school_grade_pivot['Stage'] = np.select(condlist, choicelist, 'HS')
df_school_grade_pivot['Stage'].value_counts()
```

[ ]:
```
Stage
ES    28
HS    16
MS     8
Name: count, dtype: int64
```

This revision ensures that only one grade will meet each criteria at a time, thus preventing multiple grades from getting assigned the same option.

np.select() works great in this case for assigning stages (ES, MS, or HS) to each grade. However, if grade data in some rows were missing, the above line would naively assign 'HS' to those missing rows. Although select() could be rewritten to avoid this error, another workaround would be to

use Series.map() instead. This pandas method uses a dictionary to explicitly assign each value to another value, thus making it easier to handle missing records.

The following code recreates the 'Stage' column using map. Note that, because we're addressing each grade individually, we can use the string-based 'Grade' column without falling into issues related to alphabetical order.

[ ]:
```
grade_to_stage_map = {'K':'ES', '1':'ES', '2':'ES', '3':'ES',
                      '4':'ES', '5':'ES', '6':'ES',
                      '7':'MS', '8':'MS',
                      '9':'HS', '10':'HS', '11':'HS', '12':'HS'}
# Note that we need to enter these numbers in string form because the
# presence of 'K' grades has made 'Grade' a string-formatted column (unlike
# the integer-based 'Grade_for_Sorting' column).

df_school_grade_pivot['Stage'] = df_school_grade_pivot[
    'Grade'].map(grade_to_stage_map) # This function applies
    # grade_to_stage map to map 'K' to 'ES', '7' to 'MS', and so forth.)
print("Number of grades in each stage:",
      df_school_grade_pivot['Stage'].value_counts())
# The output of this line matches that found in the above np.select() pivot.

# Here's what df_school_grade_pivot looks like at this point:
df_school_grade_pivot
```

```
Number of grades in each stage: Stage
ES    28
HS    16
MS     8
Name: count, dtype: int64
```

[ ]:
```
    School  Grade_for_Sorting Grade  Students  Underenrolled  \
0       CA                  0     K        90              0
1       CA                  1     1        71              0
2       CA                  2     2        76              0
3       CA                  3     3        61              1
4       CA                  4     4        85              0
..     …                  …   …       …            …
47      SA                  8     8        68              0
48      SA                  9     9        71              0
49      SA                 10    10        86              0
50      SA                 11    11        75              0
51      SA                 12    12        83              0

   Enrollment_Category Stage
0         Overenrolled    ES
1               Normal    ES
2               Normal    ES
```

```
3        Underenrolled    ES
4         Overenrolled    ES
..                 …   …
47              Normal    MS
48              Normal    HS
49        Overenrolled    HS
50              Normal    HS
51        Overenrolled    HS

[52 rows x 7 columns]
```

### 2.5 Renaming column values:

Two options for renaming values within columns are the Series.replace() method and Series.str.replace(). The former works great for replacing entire values with other ones, whereas the latter is useful for making edits at the character level.

First, we'll use replace() to display full school names in place of their abbreviations:

[ ]:
```
df_school_grade_pivot['School'] = df_school_grade_pivot['School'].replace(
    {'CA':'Chestnut Academy', 'SA':'Sycamore Academy',
    'HA':'Hickory Academy', 'DA':'Dogwood Academy'})
    # .map() could also have been a good candidate for
    # this operation.
df_school_grade_pivot
```

[ ]:
```
              School  Grade_for_Sorting Grade  Students  Underenrolled  \
0   Chestnut Academy                  0     K        90              0
1   Chestnut Academy                  1     1        71              0
2   Chestnut Academy                  2     2        76              0
3   Chestnut Academy                  3     3        61              1
4   Chestnut Academy                  4     4        85              0
..               …                …   …        …  …            …
47  Sycamore Academy                  8     8        68              0
48  Sycamore Academy                  9     9        71              0
49  Sycamore Academy                 10    10        86              0
50  Sycamore Academy                 11    11        75              0
51  Sycamore Academy                 12    12        83              0

   Enrollment_Category Stage
0         Overenrolled    ES
1               Normal    ES
2               Normal    ES
3        Underenrolled    ES
4         Overenrolled    ES
..                 …   …
47              Normal    MS
48              Normal    HS
```

```
49        Overenrolled    HS
50              Normal    HS
51        Overenrolled    HS

[52 rows x 7 columns]
```

Now suppose we had a change of heart and decided that we didn't need the full 'Academy' string within these column names after all. The ideal solution would be to modify the above cell to not include those names, but for demonstration purposes, we'll use str.replace() to remove them:

[ ]:
```
df_school_grade_pivot['School'] = \
df_school_grade_pivot['School'].str.replace(' Academy', '') # '' represents
# an empty string, so we're essentially instructing Pandas to delete
# the word 'Academy' from each 'School' value.
# Note the space before 'Academy' in the replace() call. This ensures that
# the spaces in between the school names and 'Academy' also gets replaced.
# Otherwise, we'd end up with names like 'Chestnut ' and 'Sycamore ', which
# would look fine to the end user but would cause issues with a merge
# operation later in this code.
df_school_grade_pivot
```

[ ]:
```
       School  Grade_for_Sorting Grade  Students  Underenrolled  \
0   Chestnut                   0     K        90              0
1   Chestnut                   1     1        71              0
2   Chestnut                   2     2        76              0
3   Chestnut                   3     3        61              1
4   Chestnut                   4     4        85              0
..       …                   …   …       …             …
47  Sycamore                   8     8        68              0
48  Sycamore                   9     9        71              0
49  Sycamore                  10    10        86              0
50  Sycamore                  11    11        75              0
51  Sycamore                  12    12        83              0

   Enrollment_Category Stage
0         Overenrolled    ES
1               Normal    ES
2               Normal    ES
3        Underenrolled    ES
4         Overenrolled    ES
..                 …   …
47              Normal    MS
48              Normal    HS
49        Overenrolled    HS
50              Normal    HS
51        Overenrolled    HS
```

```
[52 rows x 7 columns]
```

Note that `['School'].replace(' Academy', '')` would not have made any changes to the column, since no value was equal to ' Academy' outright. Thus, both Series.replace() and Series.str.replace() are required for different use cases; they're not interchangeable.

### 2.6 Merging data

Pandas is also a great tool for merging data together. Suppose an administrator wants to see the average fall and spring results on a state test for each school and grade. We can add that data to our existing school/grade pivot table by importing that data from our database; pivoting it by school, grade, and test period; and then merging it in.

[ ]:
```
df_test_results = pd.read_sql("Select * from test_results", con = engine)
df_test_results
```

[ ]:

| | Student_ID | School | Grade | Starting_Year | Period | Score |
|---|---|---|---|---|---|---|
| 0 | 42646 | CA | 1 | 2023 | Fall | 47 |
| 1 | 41632 | CA | 1 | 2023 | Fall | 49 |
| 2 | 42586 | CA | 1 | 2023 | Fall | 57 |
| 3 | 40108 | CA | 1 | 2023 | Fall | 63 |
| 4 | 43600 | CA | 1 | 2023 | Fall | 51 |
| … | … | … | … | … | … | … |
| 7995 | 42085 | SA | K | 2023 | Spring | 58 |
| 7996 | 42179 | SA | K | 2023 | Spring | 50 |
| 7997 | 41677 | SA | K | 2023 | Spring | 60 |
| 7998 | 42238 | SA | K | 2023 | Spring | 52 |
| 7999 | 43527 | SA | K | 2023 | Spring | 57 |

```
[8000 rows x 6 columns]
```

This table contains one row per student/year/period grouping. Therefore, in order to prepare it for our merge, we'll first pivot it by year, school, and grade. The 'columns' argument within pd.pivot_table() will allow us to easily place fall and spring results on the same row, thus allowing us to end up with the same number of rows as df_school_grade_pivot.

[ ]:
```
df_test_results_pivot = df_test_results.pivot_table(
    index = ['Starting_Year', 'School', 'Grade'],
    columns = 'Period',
    values = 'Score', aggfunc = 'mean').reset_index()
# Currently, the fall and spring result columns are labeled
# 'Fall' and 'Spring', so we'll rename them to make
# their meaning within the merged dataset clearer.
df_test_results_pivot.rename(
    columns = {column:column+'_Test_Results'
               for column in ['Fall', 'Spring']},
inplace = True)
# A simpler alternative to this dictionary comprehension
```

```
# would be 'Fall':'Fall_Test_Results',
# 'Spring':'Spring_Test_Results'. However, the
# dictionary comprehension will make the code easier to
# maintain in the event that new periods get added
# in the future.

df_test_results_pivot
```

[ ]:
| Period | Starting_Year | School | Grade | Fall_Test_Results | Spring_Test_Results |
|---|---|---|---|---|---|
| 0 | 2023 | CA | 1 | 49.450704 | 59.084507 |
| 1 | 2023 | CA | 10 | 50.265823 | 61.227848 |
| 2 | 2023 | CA | 11 | 48.986667 | 55.986667 |
| 3 | 2023 | CA | 12 | 51.642857 | 58.757143 |
| 4 | 2023 | CA | 2 | 51.197368 | 56.368421 |
| .. | … | … | … | … | … |
| 47 | 2023 | SA | 6 | 50.671429 | 59.128571 |
| 48 | 2023 | SA | 7 | 47.472222 | 58.069444 |
| 49 | 2023 | SA | 8 | 49.147059 | 59.367647 |
| 50 | 2023 | SA | 9 | 50.042254 | 58.704225 |
| 51 | 2023 | SA | K | 48.861111 | 57.597222 |

[52 rows x 5 columns]

In order for the merge to work successfully, we'll need to make sure that the `School` values within the test results DataFrame match the format of those within our enrollment DataFrame. Currently, the former uses abbreviated names (e.g. 'CA') and the latter uses school names (e.g. 'Chestnut'). We'll update the school names in the test results file to match those in the enrollment table via `.map()`:

[ ]:
```
df_test_results_pivot['School'] = df_test_results_pivot['School'].map(
    {'CA':'Chestnut', 'SA':'Sycamore', 'HA':'Hickory','DA':'Dogwood'})
df_test_results_pivot
```

[ ]:
| Period | Starting_Year | School | Grade | Fall_Test_Results | Spring_Test_Results |
|---|---|---|---|---|---|
| 0 | 2023 | Chestnut | 1 | 49.450704 | 59.084507 |
| 1 | 2023 | Chestnut | 10 | 50.265823 | 61.227848 |
| 2 | 2023 | Chestnut | 11 | 48.986667 | 55.986667 |
| 3 | 2023 | Chestnut | 12 | 51.642857 | 58.757143 |
| 4 | 2023 | Chestnut | 2 | 51.197368 | 56.368421 |
| .. | … | … | … | … | … |
| 47 | 2023 | Sycamore | 6 | 50.671429 | 59.128571 |
| 48 | 2023 | Sycamore | 7 | 47.472222 | 58.069444 |
| 49 | 2023 | Sycamore | 8 | 49.147059 | 59.367647 |
| 50 | 2023 | Sycamore | 9 | 50.042254 | 58.704225 |
| 51 | 2023 | Sycamore | K | 48.861111 | 57.597222 |

[52 rows x 5 columns]

Now that we have a test result table that shows data at the same school/grade level as our enrollment pivot, we can merge the two datasets together via the merge() method.

In the following code, `on` represents the keys on which to merge the data. In this case, the keys have the same name in both datasets, so we can submit the same list for each. If they had different names (e.g. 'School' vs. 'school'), we could submit separate merge key lists via `left_on` and `right_on`, but in many cases the easiest solution would be to rename the columns so that they match.

`how = 'left'` specifies that we want all school/grade pairs in df_school_grade_pivot to get retained, even if a match isn't found within df_test_results_pivot. We could also have used `how = 'inner'` to keep only rows whose school and grade keys were present in both datasets or `how = 'outer'` to retain all rows.

For more information on the merge() method, see the Pandas documentation.

[ ]:
```
df_school_grade_pivot = df_school_grade_pivot.merge(
    df_test_results_pivot.drop('Starting_Year', axis = 1),
    on = ['School', 'Grade'],
    how = 'left')
df_school_grade_pivot
```

[ ]:

| | School | Grade_for_Sorting | Grade | Students | Underenrolled | \ |
|---|---|---|---|---|---|---|
| 0 | Chestnut | 0 | K | 90 | 0 | |
| 1 | Chestnut | 1 | 1 | 71 | 0 | |
| 2 | Chestnut | 2 | 2 | 76 | 0 | |
| 3 | Chestnut | 3 | 3 | 61 | 1 | |
| 4 | Chestnut | 4 | 4 | 85 | 0 | |
| .. | … | … | … | … | … | |
| 47 | Sycamore | 8 | 8 | 68 | 0 | |
| 48 | Sycamore | 9 | 9 | 71 | 0 | |
| 49 | Sycamore | 10 | 10 | 86 | 0 | |
| 50 | Sycamore | 11 | 11 | 75 | 0 | |
| 51 | Sycamore | 12 | 12 | 83 | 0 | |

| | Enrollment_Category | Stage | Fall_Test_Results | Spring_Test_Results |
|---|---|---|---|---|
| 0 | Overenrolled | ES | 49.244444 | 58.033333 |
| 1 | Normal | ES | 49.450704 | 59.084507 |
| 2 | Normal | ES | 51.197368 | 56.368421 |
| 3 | Underenrolled | ES | 47.819672 | 59.475410 |
| 4 | Overenrolled | ES | 49.741176 | 55.988235 |
| .. | … | … | … | … |
| 47 | Normal | MS | 49.147059 | 59.367647 |
| 48 | Normal | HS | 50.042254 | 58.704225 |
| 49 | Overenrolled | HS | 48.290698 | 58.569767 |
| 50 | Normal | HS | 51.120000 | 57.466667 |
| 51 | Overenrolled | HS | 52.108434 | 57.927711 |

[52 rows x 9 columns]

If the merge failed to find a match for a certain school-grade pair, the test data columns for that pair would show up as NaN (e.g. blank). We can confirm that the merge worked as expected by checking for the presence of any NaN values within the test data columns, then raising an error message if we find any.

[ ]:
```
if df_school_grade_pivot[
    ['Fall_Test_Results', 'Spring_Test_Results']].isna().any().sum() > 0:
    raise ValueError("Missing test result data! Check your merge keys \
and revise them as needed.")
else:
    print("Test result data were found for all school/grade pairs.")
```

```
Test result data were found for all school/grade pairs.
```

We could have saved the result of the merge() operation to a new DataFrame, but in this case we overwrote df_school_grade_pivot() with the new data.

Note that running this cell a *second* time would produce unwanted results: the act of merging test result data into a dataset that already has them would cause duplicate test result columns to appear (with '_x' and '_y' added to each set of results for differentiation purposes). Here's what that would look like in practice:

[ ]:
```
# Avoid situations like the following!
double_merge_example = df_school_grade_pivot.merge(
    df_test_results_pivot.drop('Starting_Year', axis = 1),
    on = ['School', 'Grade'],
    how = 'left')
double_merge_example
```

[ ]:

| | School | Grade_for_Sorting | Grade | Students | Underenrolled | \ |
|---|---|---|---|---|---|---|
| 0 | Chestnut | 0 | K | 90 | 0 | |
| 1 | Chestnut | 1 | 1 | 71 | 0 | |
| 2 | Chestnut | 2 | 2 | 76 | 0 | |
| 3 | Chestnut | 3 | 3 | 61 | 1 | |
| 4 | Chestnut | 4 | 4 | 85 | 0 | |
| .. | … | … | … | … | … | |
| 47 | Sycamore | 8 | 8 | 68 | 0 | |
| 48 | Sycamore | 9 | 9 | 71 | 0 | |
| 49 | Sycamore | 10 | 10 | 86 | 0 | |
| 50 | Sycamore | 11 | 11 | 75 | 0 | |
| 51 | Sycamore | 12 | 12 | 83 | 0 | |

| | Enrollment_Category | Stage | Fall_Test_Results_x | Spring_Test_Results_x | \ |
|---|---|---|---|---|---|
| 0 | Overenrolled | ES | 49.244444 | 58.033333 | |
| 1 | Normal | ES | 49.450704 | 59.084507 | |
| 2 | Normal | ES | 51.197368 | 56.368421 | |
| 3 | Underenrolled | ES | 47.819672 | 59.475410 | |
| 4 | Overenrolled | ES | 49.741176 | 55.988235 | |
| .. | … | … | … | … | |

```
47              Normal    MS            49.147059              59.367647
48              Normal    HS            50.042254              58.704225
49        Overenrolled    HS            48.290698              58.569767
50              Normal    HS            51.120000              57.466667
51        Overenrolled    HS            52.108434              57.927711

    Fall_Test_Results_y  Spring_Test_Results_y
0             49.244444              58.033333
1             49.450704              59.084507
2             51.197368              56.368421
3             47.819672              59.475410
4             49.741176              55.988235
..                  …                      …
47            49.147059              59.367647
48            50.042254              58.704225
49            48.290698              58.569767
50            51.120000              57.466667
51            52.108434              57.927711

[52 rows x 11 columns]
```

### 2.7 Performing calculations on columns

Pandas also makes it very easy to perform mathematical operations on columns. This section will show just two examples: (1) adding two string columns together and (2) calculating students' fall-to-spring growth on their state exams.

First, we'll create a column that displays both school and grade data (e.g. 'Chestnut K', 'Sycamore 11'):

[ ]:
```
df_school_grade_pivot['School/Grade'] = \
    df_school_grade_pivot['School'] + ' ' + df_school_grade_pivot['Grade']
    # Note the addition of ' ' in order to provide spacing between the
    # school and grade.
df_school_grade_pivot
```

[ ]:
```
       School  Grade_for_Sorting Grade  Students  Underenrolled  \
0   Chestnut                  0     K        90              0
1   Chestnut                  1     1        71              0
2   Chestnut                  2     2        76              0
3   Chestnut                  3     3        61              1
4   Chestnut                  4     4        85              0
..       …                  …   …       …            …
47  Sycamore                  8     8        68              0
48  Sycamore                  9     9        71              0
49  Sycamore                 10    10        86              0
50  Sycamore                 11    11        75              0
51  Sycamore                 12    12        83              0
```

```
   Enrollment_Category Stage  Fall_Test_Results  Spring_Test_Results  \
0         Overenrolled    ES          49.244444            58.033333
1               Normal    ES          49.450704            59.084507
2               Normal    ES          51.197368            56.368421
3        Underenrolled    ES          47.819672            59.475410
4         Overenrolled    ES          49.741176            55.988235
..                 …   …                 …                    …
47              Normal    MS          49.147059            59.367647
48              Normal    HS          50.042254            58.704225
49        Overenrolled    HS          48.290698            58.569767
50              Normal    HS          51.120000            57.466667
51        Overenrolled    HS          52.108434            57.927711

   School/Grade
0    Chestnut K
1    Chestnut 1
2    Chestnut 2
3    Chestnut 3
4    Chestnut 4
..          …
47   Sycamore 8
48   Sycamore 9
49  Sycamore 10
50  Sycamore 11
51  Sycamore 12

[52 rows x 10 columns]
```

Next, we'll subtract our fall test results from our spring results in order to see how much students' performance grew (or declined) during the school year:

[ ]:
```
df_school_grade_pivot['Fall_to_Spring_Test_Growth'] = \
    (df_school_grade_pivot['Spring_Test_Results']
    - df_school_grade_pivot['Fall_Test_Results'])
df_school_grade_pivot
```

[ ]:
```
      School  Grade_for_Sorting Grade  Students  Underenrolled  \
0   Chestnut                  0     K        90              0
1   Chestnut                  1     1        71              0
2   Chestnut                  2     2        76              0
3   Chestnut                  3     3        61              1
4   Chestnut                  4     4        85              0
..       …                  …   …       …            …
47  Sycamore                  8     8        68              0
48  Sycamore                  9     9        71              0
49  Sycamore                 10    10        86              0
```

```
50  Sycamore                 11    11        75               0
51  Sycamore                 12    12        83               0

   Enrollment_Category Stage  Fall_Test_Results  Spring_Test_Results  \
0         Overenrolled    ES          49.244444            58.033333
1               Normal    ES          49.450704            59.084507
2               Normal    ES          51.197368            56.368421
3        Underenrolled    ES          47.819672            59.475410
4         Overenrolled    ES          49.741176            55.988235
..                 …   …                 …                    …
47              Normal    MS          49.147059            59.367647
48              Normal    HS          50.042254            58.704225
49        Overenrolled    HS          48.290698            58.569767
50              Normal    HS          51.120000            57.466667
51        Overenrolled    HS          52.108434            57.927711

   School/Grade  Fall_to_Spring_Test_Growth
0    Chestnut K                    8.788889
1    Chestnut 1                    9.633803
2    Chestnut 2                    5.171053
3    Chestnut 3                   11.655738
4    Chestnut 4                    6.247059
..          …                          …
47   Sycamore 8                   10.220588
48   Sycamore 9                    8.661972
49  Sycamore 10                   10.279070
50  Sycamore 11                    6.346667
51  Sycamore 12                    5.819277

[52 rows x 11 columns]
```

We can now sort the DataFrame by this new growth column in order to identify which school-grade pairs grew the most (and least):

[ ]:
```
df_school_grade_pivot.sort_values(
    'Fall_to_Spring_Test_Growth', ascending = False, inplace = True)
# Adding "ascending = False" places higher values above lower ones.
```

The DataFrame's original index values were retained within this sort, causing them to appear out of order. Since those values have no real meaning, we can simply replace them with a new set of values using the following code:

[ ]:
```
df_school_grade_pivot.reset_index(drop=True,inplace=True)
# "drop = True" instructs Pandas to get rid of the old index.
df_school_grade_pivot
```

```
[ ]:       School  Grade_for_Sorting Grade  Students  Underenrolled  \
     0   Chestnut                  3     3        61              1
     1   Chestnut                  8     8        75              0
     2   Chestnut                 10    10        79              0
     3   Sycamore                  7     7        72              0
     4   Chestnut                  9     9        77              0
     ..       …                  …   …       …            …
     47   Dogwood                  1     1        56              1
     48   Dogwood                  4     4        74              0
     49   Dogwood                  2     2        75              0
     50   Hickory                 11    11        65              0
     51   Hickory                 12    12        70              0

        Enrollment_Category Stage  Fall_Test_Results  Spring_Test_Results  \
     0        Underenrolled    ES          47.819672            59.475410
     1               Normal    MS          49.066667            60.066667
     2               Normal    HS          50.265823            61.227848
     3               Normal    MS          47.472222            58.069444
     4               Normal    HS          48.961039            59.285714
     ..                 …   …               …                    …
     47       Underenrolled    ES          50.642857            47.803571
     48              Normal    ES          50.513514            47.486486
     49              Normal    ES          51.026667            47.466667
     50              Normal    HS          48.415385            44.446154
     51              Normal    HS          50.828571            46.085714

        School/Grade  Fall_to_Spring_Test_Growth
     0    Chestnut 3                   11.655738
     1    Chestnut 8                   11.000000
     2   Chestnut 10                   10.962025
     3    Sycamore 7                   10.597222
     4    Chestnut 9                   10.324675
     ..          …                          …
     47    Dogwood 1                   -2.839286
     48    Dogwood 4                   -3.027027
     49    Dogwood 2                   -3.560000
     50   Hickory 11                   -3.969231
     51   Hickory 12                   -4.742857

     [52 rows x 11 columns]
```

### 2.8 Saving the DataFrame to a .csv file

We can now save our pivot table to a .csv file for easy access. This is just one way to store the data; you could also export it to Google Sheets (see the Google Sheets Uploads section of Python for Nonprofits to learn how); an .xlsx spreadsheet; or a database table, to name just some examples.

[ ]:
```
df_school_grade_pivot.to_csv(
    '../data/test_results_by_school_and_grade.csv', index = False)
# index = False excludes our numerical index from the .csv results. This
# makes re-importing the data a bit simpler, since we won't have to specify
# an index column in the dataset; instead, Pandas will simply recreate it.
# Of course, if the index contained important data (such as school names),
# we'd want to preserve it during the .csv export process.
```

That's it for this section! We will continue to use Pandas a great deal in future lessons, which will allow you to learn additional operations. My hope is that this lesson has already demonstrated the power and ease of use that this Python library offers.

### 2.9 Appendix: alternative grade sorting method

[ ]:
```
# We could also have ordered our pivot table results by grade in ascending
# chronological order (i.e. 'K', '1' . . . '11', '12') by creating a
# dictionary that stores 'K' grades as 0 and all other grades as integers,
# then sorting the DataFrame by these dictionary values. However, this
# approach works best if we *only* need to sort the pivot table by grade.

# The following cell creates the dictionary that will be used to sort grades
# in their proper ascending order (K to 12). It does so via a dictionary
# comprehension; see https://docs.python.org/3/tutorial/datastructures.
 ↪html#dictionaries
# for more information on this approach.

grade_sorting_map = {
    grade:0 if grade == 'K' else int(grade)
    for grade in df_school_grade_pivot['Grade']}

df_school_grade_pivot_alt_sort = df_curr_enrollment.pivot_table(
    index = ['School', 'Grade'],
    values = 'Students', aggfunc = 'sum').reset_index()

df_school_grade_pivot_alt_sort.sort_values(['School', 'Grade'],
    key = lambda x: x.map(grade_sorting_map), inplace = True)

# In the above code, the addition of 'inplace = True' makes the sort operation
# permanent. If we didn't include that argument, the DataFrame would
# revert back to its original sort after the operation was complete.

# See
# https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.map.
 ↪html and
# https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.
 ↪sort_values.html
# for more information on map() and the 'key' argument
```

```
# within sort_values(), respectively.


df_school_grade_pivot_alt_sort
```

[ ]:

| | School | Grade | Students |
|---|---|---|---|
| 12 | CA | K | 90 |
| 25 | DA | K | 93 |
| 38 | HA | K | 74 |
| 51 | SA | K | 72 |
| 0 | CA | 1 | 71 |
| .. | … | … | … |
| 41 | SA | 11 | 75 |
| 3 | CA | 12 | 70 |
| 16 | DA | 12 | 75 |
| 29 | HA | 12 | 70 |
| 42 | SA | 12 | 83 |

[52 rows x 3 columns]