Case Study 2: Artificial UIUC Course Catalog

We have seen that pandas data frames have a spreadsheet-like structure with the following characteristics:

- **columns** correspond to different variables and have a single type, either numerical (integer, floating point, complex) or categorical (character strings or boolean)
- **rows** are labeled by an **index** that identifies individual elements, which may be subjects, different time points, subject visits at different time points, products, or any other basic unit under study. We call these units observational units.

We have alaso seen that there are functions that operate on data frames to extract their attributes, like pandas.head() or perform operations like counting, summing, averaging, or graphing.

In this notebook, we will continue to work with data frame structures by creating and analyzing the data frame about courses at UIUC. We will investigate:

- how to create simple objects
- how to build a data frame from simpler objects
- how to import and export data files
- how to extract subsets of the data and refer to individual elements in a data frame
- how to add new data
- how to combine data from multiple sources
- how to sort data by specific variables in a data frame
- how missing data are represented, and how we can process them

At the beginning of most notebooks, you will find the necessary packages imported into your workspace. This is similar to place a book onto your desk, so that you can easily reference any information contained in that book.

We'll do this below with the **pandas package**, as the package contains many useful tools for **manipulating** and **cleaning dataframes**, which are pandas objects.

In [1]:
```
import pandas as pd
```

1. Types of Python Objects

There are many types of python objects. In our first Case Study, we used python to print the type of object created. You may review that document to see some of the object types that were created. We'll continue exploring some of the object types here.

First, we'll **create** two **list** objects.

In [2]:
```
courses = ['cs105', 'stat107', 'stat207', 'ansc307', 'hist407']
courses
```

Out[2]:
```
['cs105', 'stat107', 'stat207', 'ansc307', 'hist407']
```

In [3]:
```
type(courses)
```

```
Out[3]: list
```

```
In [4]: enrollment = [479, 343, 226, 55, 22]
        enrollment
```

```
Out[4]: [479, 343, 226, 55, 22]
```

```
In [5]: type(enrollment)
```

```
Out[5]: list
```

By using the **`type()`** function we verified that these are both lists. A list is a one-dimensional sequence of values or objects. A **list** can contain any combination of objects and is always enclosed in square **brackets** when we create one. Remember to use commas to separate the entries in the list.

We can observe what type of objects are contained within each of these two lists.

In order to do this, we need to extract individual entries in python. Note that Python starts counting from 0, so the first entry will have an index of 0, the second of 1, and so forth.

```
In [6]: courses[0]
```

```
Out[6]: 'cs105'
```

```
In [7]: courses[1]
```

```
Out[7]: 'stat107'
```

```
In [8]: courses[2]
```

```
Out[8]: 'stat207'
```

```
In [9]: type(courses[0])
```

```
Out[9]: str
```

We can see that the entries in the courses are listed as **string objects**, shortened as str. This indicates that an entry contains characters.

```
In [10]: type(enrollment[4])
```

```
Out[10]: int
```

We can see that the entries in the enrollment list are **integer objects**.

We can also create a **dictionary object**. A dictionary is a set of **ordered pairs** of **keys** and **values**. For instance,

- the 'course' key in course_dictionary below corresponds to the courses value and
- the 'enrolled' key in course_dictionary below corresponds to the enrollment value.

The structure for creating a dictionary is:

{'key1': value1, 'key2': value2, 'key3': value3, ...}

In [11]:
```
course_dictionary = {'course': courses, 'enrolled': enrollment}
course_dictionary
```

Out[11]:
```
{'course': ['cs105', 'stat107', 'stat207', 'ansc307', 'hist407'],
 'enrolled': [479, 343, 226, 55, 22]}
```

In [12]:
```
type(course_dictionary)
```

Out[12]:
```
dict
```

Pay close attention to the different types of brackets:

- '( )' enclose function arguments
- '[ ]' enclose elements in a *list* or *array*
- '{ }' enclose elements in a *dictionary*

2. Creating a dataframe

We can use the objects that we've created to build a new dataframe. We saw in Case Study 1 that we can also create a dataframe with an existing data file on our computer.

We can use the pandas **`DataFrame()`** function to create a dataframe from a dictionary.

Notice how each of the **keys become a column name** and each of the **values become a column of data**.

In [13]:
```
# Both of the first two lines of code do the same thing.
littledf = pd.DataFrame(course_dictionary)
littledf = pd.DataFrame({'course': courses, 'enrolled': enrollment})
littledf
```

Out[13]:

	course	enrolled
0	cs105	479
1	stat107	343
2	stat207	226
3	ansc307	55
4	hist407	22

3. Select a column from a dataframe

We saw in the previous Case Study that we can use square brackets to isolate a single variable/column from our dataframe. We use the following syntax to do so:

`dataframe_name['column_name']`

In [14]:
```
littledf['enrolled']
```

Out[14]:
```
0    479
1    343
2    226
```

```
3     55
4     22
Name: enrolled, dtype: int64
```

4. Add a column to a dataframe

We can **create a new column in a dataframe** by using brackets and the name of the column we want to create on the left, and then an object with the new data for the column on the right. We use the following syntax to do so:

`dataframe_name['column_name'] = new_data`

In [15]:
```
littledf['college'] = ['ENGR', 'LAS', 'LAS', 'ACES', 'LAS']
littledf
```

Out[15]:

	course	enrolled	college
0	cs105	479	ENGR
1	stat107	343	LAS
2	stat207	226	LAS
3	ansc307	55	ACES
4	hist407	22	LAS

5. Writing a dataframe to a csv

We often start a data analysis by reading in data from an external file source. We saw an example of this in Case Study 1 using `pd.read_csv()` .

After data processing and/or manipulation, we may be interested in the reverse operation: saving our current, internal dataframe to a transportable data file. Here, we'll export the 'littledf' data to an external csv file using the **`pandas.DataFrame:to_csv`** function.

In [16]:
```
littledf.to_csv('courses.csv')
```

If you navigate to the folder that this Jupyter notebook is saved in, you should see a **new csv** file created called 'courses.csv' that contains the data from littledf.

Try It Tutorial!

Suppose that you wanted to add another variable to littledf, indicating whether each course had reached its enrollment capacity.

Of the courses, only ANSC305 has reached its capacity.

To prevent any later issues, we'll make a copy of the dataframe. For this tutorial, use the tryit dataframe created below.

1. Add an appropriate variable titled 'isfull' to the dataframe.
2. Check the object type of the first entry.
3. What type of variable is 'isfull'?

4. Use python tools to calculate the percent of classes in our littledf that are at capacity?

In [17]:
```
tryit = littledf.copy()
```

In [ ]:
```

```

6. Subsetting a Dataframe with Index Numbers

For various reasons, we may be interested in reducing our data. We may look to subset our data to only include specific instances within a dataframe. We'll see examples below.

If you know the specific location of the dataframe that you are targeting, you can use the **`iloc`** attribute. We can use this to refer to specific elements or "slices" of elements in the dataframe.

We can select a **single entry** as below. Here, we are targeting the upper left entry:

In [18]:
```
littledf.iloc[0,0]
```

Out[18]:
```
'cs105'
```

We can extract the element with row index 3, column index 2 as:

In [19]:
```
littledf.iloc[3,2]
```

Out[19]:
```
'ACES'
```

Note: because Python starts counting at 0, you might describe this as the element visually appearing in the 4th row and the 3rd column.

Another way to get this same information is to extract the column (using the name) first, and then select the index of the entry in the row you want.

In [20]:
```
littledf['college'][3]
```

Out[20]:
```
'ACES'
```

In [21]:
```
littledf['college']
```

Out[21]:
```
0    ENGR
1     LAS
2     LAS
3    ACES
4     LAS
Name: college, dtype: object
```

We can also select **ranges of entries** using row and/or column indices.

With a two-dimensional dataframe, our row indices are provided first, followed by a comma and then our column indices.

We can extract a slice of more than one element using the sequence notation i:j:k to refer to indices running from i to j using step-size k. The default k is assumed to be 1. If we do not provide i and j, then it is assumed that we'd like to use the whole range.

Here's an example where we extract the middle three rows of the dataframe. Note that '1:4' results in the inclusion of rows 1, 2, and 3 but not 4!

In [22]:
```
littledf.iloc[1:4,:]
```

Out[22]:

	course	enrolled	college
1	stat107	343	LAS
2	stat207	226	LAS
3	ansc307	55	ACES

If we wanted to include rows 0-3, we can use the sequence ':4', which includes all rows before the row with index=4.

In [23]:
```
littledf.iloc[:4,:]
```

Out[23]:

	course	enrolled	college
0	cs105	479	ENGR
1	stat107	343	LAS
2	stat207	226	LAS
3	ansc307	55	ACES

If, on the other hand, we wished to include all rows after rows 0 and 1, the sequence ':2' will do this.

In [24]:
```
littledf.iloc[2:,:]
```

Out[24]:

	course	enrolled	college
2	stat207	226	LAS
3	ansc307	55	ACES
4	hist407	22	LAS

We can also specify lists of the **specific row and/or column indices** that we want to select, even if those entries are not in subsequent order. For example, we may wish to print only the first, second, and fourth rows, or only the zeroth and second columns.

In [25]:
```
littledf.iloc[[1,2,4],:]
```

Out[25]:

	course	enrolled	college
1	stat107	343	LAS
2	stat207	226	LAS
4	hist407	22	LAS

In [26]:
```
littledf.iloc[:,[0,2]]
```

Out[26]:

	course	college
0	cs105	ENGR
1	stat107	LAS
2	stat207	LAS
3	ansc307	ACES
4	hist407	LAS

In [27]:
```
littledf.iloc[[0,3],[2]]
```

Out[27]:

	college
0	ENGR
3	ACES

7. Subsetting a Dataframe with Conditions

7.1. Filtering rows by a column condition

Three cells above this, we printed all of the rows in our littledf dataframe that are in the college of LAS. We could easily find these indices in our small dataframe, but how would we select just these rows without tediously having to look up the row indices? This would be especially helpful if we had a larger dataframe of all courses on campus.

First, we'll want to create a series that 'checks' whether a *given condition* is true. I will often refer to this as a "logical statement".

In [28]:
```
# Logical Statement: Is the entry = 'LAS'?
littledf['college']=='LAS'
```

Out[28]:
```
0    False
1     True
2     True
3    False
4     True
Name: college, dtype: bool
```

In [29]:
```
type(littledf['college']=='LAS')
```

Out[29]:
```
pandas.core.series.Series
```

Once we have our condition checked, we can then filter our rows based on the column condition. This takes the syntax below

`dataframe_name[column entry condition]`

In [30]:
```
littledf[littledf['college']=='LAS']
```

Out[30]:

	course	enrolled	college
1	stat107	343	LAS

	course	enrolled	college
2	stat207	226	LAS
4	hist407	22	LAS

What if we want only the enrollment of the LAS courses?

In this case, we want to combine filtering based on a condition and column selection. There are many ways to combine our tools to accomplish this. We can do this step-by-step or all at once.

In [31]:
```
# Approach 1
smallerdf = littledf[littledf['college']=='LAS']
smallerdf
```

Out[31]:

	course	enrolled	college
1	stat107	343	LAS
2	stat207	226	LAS
4	hist407	22	LAS

In [32]:
```
smallerdf['enrolled']
```

Out[32]:
```
1    343
2    226
4     22
Name: enrolled, dtype: int64
```

In [33]:
```
# Approach 2
littledf[littledf['college']=='LAS']['enrolled']
```

Out[33]:
```
1    343
2    226
4     22
Name: enrolled, dtype: int64
```

Above, we see that we first extracted the three rows assocaciated with LAS courses. We can then refer to the 'enrolled' column of this smaller dataframe.

We could switch the order, by first extracting the enrollments and then filtering.

In [34]:
```
# Approach 3
enrolled_column = littledf['enrolled']
enrolled_column
```

Out[34]:
```
0    479
1    343
2    226
3     55
4     22
Name: enrolled, dtype: int64
```

In [35]:
```
enrolled_column[littledf['college']=='LAS']
```

Out[35]:
```
1    343
2    226
4     22
Name: enrolled, dtype: int64
```

In [36]:
```
# Approach 4
littledf['enrolled'][littledf['college']=='LAS']
```

Out[36]:
```
1    343
2    226
4     22
Name: enrolled, dtype: int64
```

7.2. Syntax for checking conditions

Notice that when we wanted to test if an entry in the 'college' column was **equal to** 'LAS', we use "==" rather than "=". The distinction between the two is:

- we use "==" when we are setting a condition in Python
- we use "=" when we are assigning a value in Python. That is, when we are defining a variable or parameter.

There are additional **operators** that we can use to define other types of **conditions*:

- equal to: ==
- greater than or equal to: >=
- greater than: >
- less than: <
- less than or equal to: <=

Using this, we can extract all the courses with enrollemnts of at least 50:

In [37]:
```
littledf[littledf['enrolled']>=50]
```

Out[37]:

	course	enrolled	college
0	cs105	479	ENGR
1	stat107	343	LAS
2	stat207	226	LAS
3	ansc307	55	ACES

We can extract the courses with enrollments less than 50:

In [38]:
```
littledf[littledf['enrolled']<50]
```

Out[38]:

	course	enrolled	college
4	hist407	22	LAS

We can extract the record corresponding to a particular course:

In [39]:
```
littledf[littledf['course']=='ansc307']
```

Out[39]:

	course	enrolled	college
3	ansc307	55	ACES

8. Summarizing Columns of a Dataframe

8.1. Aggregating a Column

Suppose, first, we want the total enrollment of all classes in our dataframe. Below is one way to calculate that information, using the **`.sum()`** function.

In [40]:
```
littledf['enrolled'].sum()
```

Out[40]:
```
1125
```

There are many other column aggregation functions like the following:

- **`.min()`**
- **`.max()`**

We'll learn additional functions later.

In [41]:
```
littledf['enrolled'].max()
```

Out[41]:
```
479
```

In [42]:
```
littledf['enrolled'].min()
```

Out[42]:
```
22
```

8.2. Aggregating a Filtered Dataframe

We can combine filtering a data with summarizing a column. Suppose that we want to find the total enrollment of JUST LAS classes in our dataframe. We can do this with the following steps:

1. First create a dataframe filtered with only 'LAS' classes
2. Extract just the 'enrolled' column
3. Take the sum of the column from step 2

In [43]:
```
# In one cell
print('Total LAS Enrollment')
littledf[littledf['college']=='LAS']['enrolled'].sum()
```

```
Total LAS Enrollment
```

Out[43]:
```
591
```

In [44]:
```
# Step 1 completed
littledf[littledf['college']=='LAS']
```

Out[44]:

	course	enrolled	college
1	stat107	343	LAS
2	stat207	226	LAS
4	hist407	22	LAS

In [45]:
```
# Step 2 completed
littledf[littledf['college']=='LAS']['enrolled']
```

Out[45]:
```
1    343
2    226
4     22
Name: enrolled, dtype: int64
```

In [46]:
```
# Step 3 completed
littledf[littledf['college']=='LAS']['enrolled'].sum()
```

Out[46]:
```
591
```

8.3. Using a Summary Measure as a Condition

Suppose now we want to extract the **row** from littledf that has the maximum enrollment. We can first find this largest enrollment size and then use the number directly.

In [47]:
```
print('Maximum Enrollment')
littledf['enrolled'].max()
```

```
Maximum Enrollment
```

Out[47]:
```
479
```

In [48]:
```
littledf[littledf['enrolled']==479]
```

Out[48]:

	course	enrolled	college
0	cs105	479	ENGR

We can also combine these steps directly by using the 'littledf['enrolled'].max()' value directly in the condition. This can be done in one line of code and is more efficient from a coding perspective.

In [49]:
```
littledf[littledf['enrolled']==littledf['enrolled'].max()]
```

Out[49]:

	course	enrolled	college
0	cs105	479	ENGR

9. Concatenate ("stack") Two Dataframes

Suppose we had more enrollment data for additional courses to add to the dataframe. We can use the pandas **concat()** function to combine the original dataframe with a new dataframe containing the additional records/observational units. Here we create a new dataframe with the hypothetical new data.

In [50]:
```
moredf = pd.DataFrame({'course': ['math227', 'is457'],
                       'enrolled': [59, 58],
                       'college': ['LAS', 'IS']})
```

In [51]:
```
display(littledf, moredf)
```

	course	enrolled	college

	course	enrolled	college
0	cs105	479	ENGR
1	stat107	343	LAS
2	stat207	226	LAS
3	ansc307	55	ACES
4	hist407	22	LAS

	course	enrolled	college
0	math227	59	LAS
1	is457	58	IS

Next, we combine them. In this example, we will tell python to ignore the original index values and create a new index for the combined data.

In [52]:
```
fulldf = pd.concat([littledf, moredf], ignore_index=True)
fulldf
```

Out[52]:

	course	enrolled	college
0	cs105	479	ENGR
1	stat107	343	LAS
2	stat207	226	LAS
3	ansc307	55	ACES
4	hist407	22	LAS
5	math227	59	LAS
6	is457	58	IS

What would happen if we didn't ask python to create a new set of row indices?

In [53]:
```
pd.concat([littledf, moredf])
```

Out[53]:

	course	enrolled	college
0	cs105	479	ENGR
1	stat107	343	LAS
2	stat207	226	LAS
3	ansc307	55	ACES
4	hist407	22	LAS
0	math227	59	LAS
1	is457	58	IS

Note that we now have two rows with index 0, and two rows with index 1. This is not preferred and could cause later challenges with analysis.

A quick way to add new records is with the **`append()`** function. Row indices are also handled nicely with this function.

In [54]:

```
newdf = pd.DataFrame({'course': ['badm210'],
                      'enrolled': [388],
                      'college': ['BUSN']})
newdf
```

Out[54]:

```
   course  enrolled  college
0  badm210      388     BUSN
```

In [55]:

```
updateddf = fulldf.append(newdf, ignore_index=True)
updateddf
```

Out[55]:

```
    course  enrolled  college
0    cs105       479     ENGR
1  stat107       343      LAS
2  stat207       226      LAS
3  ansc307        55     ACES
4  hist407        22      LAS
5  math227        59      LAS
6    is457        58       IS
7  badm210       388     BUSN
```

10. Merge ("join") Two Dataframes

Another common scenario is to have more than one source of data with different variables, and we wish to combined data sets for further analysis. As an example, suppose in the previous course list example we had another source with the credit hours for each class. We'd like to add this information.

In [56]:

```
creditdf = pd.DataFrame({'course': ['ansc307', 'cs105', 'stat107', 'stat207',
                                    'hist407', 'math227', 'is457', 'badm210'],
                         'credit': [3.0, 3.0, 4.0, 4.0, 2.0, 3.0, 3.0, 3.0]})
creditdf
```

Out[56]:

```
    course  credit
0  ansc307     3.0
1    cs105     3.0
2  stat107     4.0
3  stat207     4.0
4  hist407     2.0
5  math227     3.0
6    is457     3.0
7  badm210     3.0
```

In this case, we can do a one-to-one join between the two data frames using the python `merge()` function. Notice that the order of courses does not need to be the same; the records are matched based on the shared course name.

In [57]:
```
fullerdf = pd.merge(updateddf, creditdf)
fullerdf
```

Out[57]:

	course	enrolled	college	credit
0	cs105	479	ENGR	3.0
1	stat107	343	LAS	4.0
2	stat207	226	LAS	4.0
3	ansc307	55	ACES	3.0
4	hist407	22	LAS	2.0
5	math227	59	LAS	3.0
6	is457	58	IS	3.0
7	badm210	388	BUSN	3.0

Often, the two data sources will not be in a one-to-one correspondence between their records. Then we might need to perform a "many-to-one" merge.

Example: In one data source, we have courses and section enrollments. In the other data source, we have courses and credit hours. Let's combine them. First, we'll create a dataframe with the section information.

In [58]:
```
courses = ['cs105', 'stat207', 'stat207', 'badm210', 'badm210']
sections = ['A', 'A', 'B', 'A', 'B']
enrollments = [479, 103, 123, 216, 172]
sectdf = pd.DataFrame({'course': courses,
                      'section': sections,
                      'enrolled': enrollments})
sectdf
```

Out[58]:

	course	section	enrolled
0	cs105	A	479
1	stat207	A	103
2	stat207	B	123
3	badm210	A	216
4	badm210	B	172

We'd like to merge this with the credit information.

In [59]:
```
creditdf
```

Out[59]:

	course	credit
0	ansc307	3.0
1	cs105	3.0
2	stat107	4.0
3	stat207	4.0
4	hist407	2.0

	course	credit
5	math227	3.0
6	is457	3.0
7	badm210	3.0

We can try a "default" merge and see what we get:

In [60]:
```
pd.merge(sectdf, creditdf)
```

Out[60]:

	course	section	enrolled	credit
0	cs105	A	479	3.0
1	stat207	A	103	4.0
2	stat207	B	123	4.0
3	badm210	A	216	3.0
4	badm210	B	172	3.0

Did it work? Yes, in the sense that all course sections in the first dataframe have now been assigned credit hours. Any course that appears in both data sources gets matched. The courses missing from one or the other were not included.

In some cases, we need to specify which variable to use as the matching **key** using the **on=** option:

In [61]:
```
pd.merge(sectdf, creditdf, on='course')
```

Out[61]:

	course	section	enrolled	credit
0	cs105	A	479	3.0
1	stat207	A	103	4.0
2	stat207	B	123	4.0
3	badm210	A	216	3.0
4	badm210	B	172	3.0

11. Sorting a Dataframe

In the examples we've been considering, the course names are in no particular order. What if we want the courses to be in alphanumeric order? pandas has a function for that: `.sort_values` . For the syntax, see https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sort_values.html

To select a specific column on which to sort, we use the **by=** option.

In [62]:
```
creditdf.sort_values(by='course')
```

Out[62]:

	course	credit
0	ansc307	3.0
7	badm210	3.0

	course	credit
1	cs105	3.0
4	hist407	2.0
6	is457	3.0
5	math227	3.0
2	stat107	4.0
3	stat207	4.0

In [63]:
```
creditdf
```

Out[63]:

	course	credit
0	ansc307	3.0
1	cs105	3.0
2	stat107	4.0
3	stat207	4.0
4	hist407	2.0
5	math227	3.0
6	is457	3.0
7	badm210	3.0

Notice that while the first dataframe is sorted by course name, it did not permanently sort the creditdf dataframe.

Remarks:

1. We can specify more than one variable for sorting, and we can also select various other options such as "ascending=False" (default is "ascending=True"), where to put NaNs in the ordering ("na_position='last'"), and whether to sort in-place (overwriting the original object).
2. This operation did **not** replace the original data with sorted data. It merely displayed the sorted data. If we wanted to save the sorted results, we can assign them to a new pandas object or we can sort "in place", which is illustrated below.

Sorting "in place" will sort the data and have it remain sorted.

In [64]:
```
creditdf.sort_values(by='course', inplace=True)
creditdf
```

Out[64]:

	course	credit
0	ansc307	3.0
7	badm210	3.0
1	cs105	3.0
4	hist407	2.0
6	is457	3.0
5	math227	3.0

	course	credit
2	stat107	4.0
3	stat207	4.0

In [65]:
```
# overwriting the original data
creditdf = creditdf.sort_values(by='course')
creditdf
```

Out[65]:

	course	credit
0	ansc307	3.0
7	badm210	3.0
1	cs105	3.0
4	hist407	2.0
6	is457	3.0
5	math227	3.0
2	stat107	4.0
3	stat207	4.0

We could also sort the class sections by their enrollments, from lowest to highest.

In [66]:
```
sectdf.sort_values(by='enrolled', ascending=True)
```

Out[66]:

	course	section	enrolled
1	stat207	A	103
2	stat207	B	123
4	badm210	B	172
3	badm210	A	216
0	cs105	A	479

Try It Tutorial!

Suppose that we'd like to analyze our fullerdf dataframe with the following steps.

We've started by making a copy of the fullerdf dataframe as tryit2, which you should use for this tutorial. This preserves the original data, should you need to reference it.

1. Filter the dataset to contain only courses that have small enrollment, defined as having enrollments less than 100.
2. Sort the dataset by the college.
3. What is the sum of the credit hours for the smaller enrollment courses?
4. **Review**: Make a table summarizing the number of courses with smaller enrollments that are in each college type. How many small enrollment courses are from ACES?

Repeat this process with the larger enrollment courses, or those courses that have enrollments of at least 100. How many credit hours are associated with the larger enrollment courses? What proportion of larger enrollment courses are from LAS?

In [67]:
```
tryit2 = fullerdf.copy()
```

In [ ]:
```

```

12. Overwriting a Single Entry

Let's pretend that we are now unsure about the enrollment of section B of badm210. We could overwrite the enrollment entry by replacing 172 with the string 'unknown'. However, we see below that by adding a string to a column comprised of numbers, we receive an error when we try to apply a function that only applied to numbers, like .sum().

In [68]:
```
tmp = sectdf.copy() # copy of data frame
tmp
```

Out[68]:

	course	section	enrolled
0	cs105	A	479
1	stat207	A	103
2	stat207	B	123
3	badm210	A	216
4	badm210	B	172

In [69]:
```
tmp['enrolled'][4] # Access the enrollment for badm210 section B
```

Out[69]:
```
172
```

In [70]:
```
tmp.iloc[4,2] # another option to access
```

Out[70]:
```
172
```

In [71]:
```
tmp.iloc[4,2] = 'unknown' # coding this element as something else
tmp
```

Out[71]:

	course	section	enrolled
0	cs105	A	479
1	stat207	A	103
2	stat207	B	123
3	badm210	A	216
4	badm210	B	unknown

In [72]:
```
tmp['enrolled'].sum()
```

```
---------------------------------------------------------------------------
```

```
TypeError                                 Traceback (most recent call last)
/var/folders/8n/vcl88k4x7qnddbqbz4ljfx780000gn/T/ipykernel_96950/1824853949.py in <module>
----> 1 tmp['enrolled'].sum()

~/miniconda3/lib/python3.8/site-packages/pandas/core/generic.py in sum(self, axis, skipna,
 level, numeric_only, min_count, **kwargs)
  10706             **kwargs,
  10707         ):
> 10708             return NDFrame.sum(
  10709                 self, axis, skipna, level, numeric_only, min_count, **kwargs
  10710             )

~/miniconda3/lib/python3.8/site-packages/pandas/core/generic.py in sum(self, axis, skipna,
 level, numeric_only, min_count, **kwargs)
  10444         **kwargs,
  10445     ):
> 10446         return self._min_count_stat_function(
  10447             "sum", nanops.nansum, axis, skipna, level, numeric_only, min_count, **
kwargs
  10448         )

~/miniconda3/lib/python3.8/site-packages/pandas/core/generic.py in _min_count_stat_functio
n(self, name, func, axis, skipna, level, numeric_only, min_count, **kwargs)
  10426                 numeric_only=numeric_only,
  10427             )
> 10428         return self._reduce(
  10429             func,
  10430             name=name,

~/miniconda3/lib/python3.8/site-packages/pandas/core/series.py in _reduce(self, op, name,
 axis, skipna, numeric_only, filter_type, **kwds)
   4390                 )
   4391             with np.errstate(all="ignore"):
-> 4392                 return op(delegate, skipna=skipna, **kwds)
   4393 
   4394     def _reindex_indexer(

~/miniconda3/lib/python3.8/site-packages/pandas/core/nanops.py in _f(*args, **kwargs)
     92             try:
     93                 with np.errstate(invalid="ignore"):
---> 94                     return f(*args, **kwargs)
     95             except ValueError as e:
     96                 # we want to transform an object array

~/miniconda3/lib/python3.8/site-packages/pandas/core/nanops.py in new_func(values, axis, s
kipna, mask, **kwargs)
    409             mask = isna(values)
    410 
--> 411         result = func(values, axis=axis, skipna=skipna, mask=mask, **kwargs)
    412 
    413         if datetimelike:

~/miniconda3/lib/python3.8/site-packages/pandas/core/nanops.py in nansum(values, axis, ski
pna, min_count, mask)
    589         dtype_sum = np.float64  # type: ignore[assignment]
    590 
--> 591     the_sum = values.sum(axis, dtype=dtype_sum)
    592     the_sum = _maybe_null_out(the_sum, axis, mask, values.shape, min_count=min_cou
nt)
    593 

~/miniconda3/lib/python3.8/site-packages/numpy/core/_methods.py in _sum(a, axis, dtype, ou
t, keepdims, initial, where)
     45 def _sum(a, axis=None, dtype=None, out=None, keepdims=False,
     46          initial=_NoValue, where=True):
---> 47     return umr_sum(a, axis, dtype, out, keepdims, initial, where)
```

```
     48
     49 def _prod(a, axis=None, dtype=None, out=None, keepdims=False,

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In [73]:
```
sectdf['enrolled'].sum()
```

Out[73]:
```
1093
```

13. NaN Objects

We see that we have an error when we try to sum the enrolled variable in our new dataframe, but we don't see the same error when we try to sum from our original dataframe. The only difference is the 'unknown' entry, so that must be the cause of our error!

Missing data are very common in real data applications. How can we handle them at a basic level? What is a better way to indicate to Python that we do not know the enrollment of section B of badm210? We can overwrite the enrollment for this class using another type of filter for unknown or missing values.

In [74]:
```
tmp = sectdf.copy() # copy of data frame
tmp
```

Out[74]:

	course	section	enrolled
0	cs105	A	479
1	stat207	A	103
2	stat207	B	123
3	badm210	A	216
4	badm210	B	172

In [75]:
```
tmp.iloc[4,2] = None # coding this element as NaN
tmp
```

Out[75]:

	course	section	enrolled
0	cs105	A	479.0
1	stat207	A	103.0
2	stat207	B	123.0
3	badm210	A	216.0
4	badm210	B	NaN

In [76]:
```
tmp.iloc[4,2]
```

Out[76]:
```
nan
```

In [77]:
```
tmp['enrolled'].sum()
```

Out[77]:
```
921.0
```

We can now sum the enrollment numbers without an error!

Our missing value is encoded as NaN (not a number). Python knows how to handle NaNs when performing calculations or other data frame manipulations.

By default, many functions will skip data with missing values. This is what happened with the sum function. Often this makes sense, but not always! The question of what this sum now represents is one that we won't answer now.

What if we wanted to sort by enrollment? We need to specify whether missing values should go first or last on the list.

In [78]:
```
tmp.sort_values(by='enrolled', na_position='first')
```

Out[78]:

	course	section	enrolled
4	badm210	B	NaN
1	stat207	A	103.0
2	stat207	B	123.0
3	badm210	A	216.0
0	cs105	A	479.0

14. Finding Missing Values in a Dataframe (Basic)

The **`DataFrame.isna`** function can scan a dataframe for missing values. **`DataFrame.notna`** scans for non-missing values.

In [79]:
```
tmp.isna()
```

Out[79]:

	course	section	enrolled
0	False	False	False
1	False	False	False
2	False	False	False
3	False	False	False
4	False	False	True

We can take the sum of each column above. When we sum a column of boolean values, Python automatically translates a True to a 1 and a False to a 0.

Therefore, the **`.sum()`** function counted 1 True (i.e. a missing value) in the enrolled column and 0 Trues (i.e., no missing values) in the other columns.

In [80]:
```
tmp.isna().sum()
```

Out[80]:
```
course      0
section     0
enrolled    1
dtype: int64
```

15. Dropping Rows with Missing Values

There are many ways that we might want to handle missing data. Deciding how to handle missing data requires care, and may depend on the specific situation.

One method (not always the best, but often the easiest) is to only analyze the observations with complete information. For that analysis, we can use the **`DataFrame.dropna`** function, which extracts complete data for us.

In [81]:

```
tmp.dropna()
```

Out[81]:

```
    course  section  enrolled
0    cs105        A     479.0
1  stat207        A     103.0
2  stat207        B     123.0
3  badm210        A     216.0
```

STAT 207: Julie Deeke, Victoria Ellison, and Douglas Simpson. University of Illinois at Urbana-Champaign