Guide to Using Python to Accompany Business Analytics by Jaggia et al.

Chapter 2

Prepared by David G. Dobolyi at the University of Notre Dame

Table of Contents

Example 2.1

BalanceGig is a company that matches independent workers for short-term engagements with businesses in the construction, automative, and high-tech industries. The 'gig' employees work only for a short period of time, often on a particular project or a specific task. A manager at BalanceGig extracts the employee data from their most recent work engagement, including the hourly wage (HourlyWage), the client's industry (Industry), and the employee's job classification (Job). A portion of the ***Gig*** data set is shown in Table 2.3 below.

Table 2.3 Gig Employee Data

EmployeeID	HourlyWage	Industry	Job
1	32.81	Construction	Analyst
2	46	Automotive	Engineer
...	...	...	...
604	26.09	Construction	Other

The manager suspects that data about the gig employees are sometimes incomplete, perhaps due to the short engagement and the transient nature of the employees. She would like to find the number of missing observations for the HourlyWage, Industry, and Job variables. In addition, she would like information on the number of employees who (1) worked in the automative industry, (2) earned more than \$30 per hour, and (3) worked in the automotive industry and earned more than \\$30 per hour. Finally, the manager would like to know the hourly wage of the lowest- and the highest-paid employees at the company as a whole and the hourly wage of the lowest- and highest-paid accountants who worked in teh automotive and the tech industries.

Use counting and sorting functions in Python to find the relevant information requested by the manager, and then summarize the results.

Solution

Before following all Python instructions, make sure that you have read Supplement X ("Getting Started with Python"). We assume that you have downloaded Python and Jupyter Lab and that you know how to import an Excel file (including installing necessary packages such as `pandas` and `xlrd`). Throughout the text, our goal is to provide the simplest way to obtain the relevant output. We denote all function, method, and property names in **boldface** and all options within functions in *italics*.

a. Import the ***Gig*** data file into a Pandas DataFrame (table) and label it myData. Keep in mind that the Python language is case sensitive.

In [1]:

```
import pandas as pd

myData = pd.read_excel('jaggia_ba_1e_ch02_Data_Files.xlsx', sheet_name = 'Gi
g')
```

b. We use the **shape** property of the Pandas DataFrame to count the number of observations and variables. Verify that the Python output shows 604 observations and four variables:

In [2]:

```
myData.shape
```

Out[2]:

```
(604, 4)
```

c. Three common functions to display a portion of data are **head**, **tail**, and **sample**. The head function displays the first few observations in the data set, the tail function displays the last few, and the sample function displays a random sample. For example, to verify that the first employee in the data set is an analyst who worked in the construction industry and made \$32.81 per hour, use the following command:

In [3]:

```
myData.head()
```

Out[3]:

	EmployeeID	HourlyWage	Industry	Job
0	1	32.81	Construction	Analyst
1	2	46.00	Automotive	Engineer
2	3	43.13	Construction	Sales Rep
3	4	48.09	Automotive	Other
4	5	43.62	Automotive	Accountant

It is important to note that Pandas lables rows using an index where the first row starts at 0 (i.e., see the table above).

d. Pandas stores missing values as NaN, and we use the **isna** function to identify the observations with missing values. Observations with missing values will return `True` whereas observations without missing values will return `False`:

In [4]:

```
pd.isna(myData.Industry)
```

Out[4]:

```
0      False
1      False
2      False
3      False
4      False
       ...
599    False
600    False
601    False
602    False
603    False
Name: Industry, Length: 604, dtype: bool
```

e. For a large data set, having to look through all observations is inconvenient. To find out how many missing observations are missing, we can subsequently use the **sum** function to add up the True and False values and return a count (i.e., since True is numerically equal to 1 and False is equal to 0):

In [5]:

```
pd.isna(myData.Industry).sum()
```

Out[5]:

```
10
```

Alternatively, we can use the NumPy **where** function together with the **isna** function to identify which particular observations contain missing values. The following command identifies the 10 missing observations in the Industry variable by index. Note that the that first observation with a missing Industry value is in index 23 (i.e., row 24 since Python indexes start with 0, as noted earlier):

In [6]:

```
import numpy as np

np.where(pd.isna(myData.Industry))
```

Out[6]:

```
(array([ 23, 138, 360, 377, 440, 445, 478, 499, 530, 564], dtype=
int64),)
```

f. To inspect the 24th observation, we specify index 23 in the myData DataFrame using the **iloc** property:

In [7]:

```
myData.iloc[23,]
```

Out[7]:

```
EmployeeID           24
HourlyWage        42.58
Industry            NaN
Job           Sales Rep
Name: 23, dtype: object
```

Note that there are two elements within the square bracket, separated by a comma. The first element identifies a row index, and the second element after the comma identifies a column index. Leaving the second element blank will display all columns. To inspect an observation in row 24 and column 3, use:

In [8]:

```
myData.iloc[23, 2]
```

Out[8]:

```
nan
```

g. To identify and count the number of employees using specific selection criteria, we can use the double equal sign (==), also called the equal operator, to check whether the industry is automotive. Subsequently, we can again use the **sum** function to add up the results and return a count. Notice that in Python, text sequences such as 'Automotive' are enclosed in quotation marks:

In [9]:

```
(myData.Industry == 'Automotive').sum()
```

Out[9]:

```
190
```

We can also use the >, <=, <, <=, and != (not equal) operators in the selection criteria. For example, using the following command, we can determine the number of employees who earn more than \$30 per hour:

In [10]:

```
(myData.HourlyWage > 30).sum()
```

Out[10]:

```
536
```

Note that there are 190 employees in the automotive industry and there are 536 employees who earn more than \$30 per hour.

h. To count how many employees worked in a particular industry and earned more than a particular wage, we use the \& (bitwise and) operator. The following command shows that 181 employees worked in the automotive industry and earned more than \$30 per hour:

In [11]:

```
((myData.Industry == 'Automotive') & (myData.HourlyWage > 30)).sum()
```

Out[11]:

```
181
```

Note that the way the parentheses are arranged above is important for the command to work correctly.

i. We use the Pandas **sort_values** function to sort the observations of a variable. The following command sorts myData based on the HourlyWage variable and stores the reordered data set in a new DataFrame called sortedData1:

In [12]:

```
sortedData1 = myData.sort_values('HourlyWage')
```

Calling up sortedData1 directly shows that the lowest and highest hourly wages are \$24.28 and \\$51.00, respectively:

In [13]:

```
sortedData1
```

Out[13]:

	EmployeeID	HourlyWage	Industry	Job
466	467	24.28	Construction	Engineer
546	547	24.28	Construction	Sales Rep
579	580	24.28	Construction	Accountant
558	559	24.42	Construction	Engineer
220	221	24.76	Automotive	Programmer
...	...	...	...	...
598	599	49.84	Automotive	Engineer
347	348	49.91	Construction	Accountant
372	373	49.91	Construction	Accountant
78	79	50.00	Automotive	Engineer
109	110	51.00	Construction	Other

604 rows × 4 columns

By default, the sorting is performed in ascending order. To sort in descending order, change the *ascending* argument to False:

In [14]:

```
sortedData1 = myData.sort_values('HourlyWage', ascending = False)

sortedData1
```

Out[14]:

	EmployeeID	HourlyWage	Industry	Job
109	110	51.00	Construction	Other
78	79	50.00	Automotive	Engineer
347	348	49.91	Construction	Accountant
372	373	49.91	Construction	Accountant
598	599	49.84	Automotive	Engineer
...	...	...	...	...
15	16	24.76	Automotive	Programmer
558	559	24.42	Construction	Engineer
546	547	24.28	Construction	Sales Rep
579	580	24.28	Construction	Accountant
466	467	24.28	Construction	Engineer

604 rows × 4 columns

j. To sort the data by multiple variables, we supply a list of values (i.e., comma separated inside square brackets). The following command sorts the data by industry, job classification, and hourly wage. all in ascending order, and stores the ordered data in a DataFrame called sortedData2:

In [15]:

```
sortedData2 = myData.sort_values(['Industry', 'Job', 'HourlyWage'])
```

Calling up the **head** of sortedData2 shows that the lowest-paid accountant who worked in the automotive industry made $28.74 per hour:

In [16]:

```
sortedData2.head()
```

Out[16]:

	EmployeeID	HourlyWage	Industry	Job
566	567	28.74	Automotive	Accountant
76	77	29.00	Automotive	Accountant
573	574	32.10	Automotive	Accountant
35	36	38.28	Automotive	Accountant
475	476	39.67	Automotive	Accountant

k. To sort the data by industry and job classification in ascending order and then by hourly wage in descending order, we can supply a respective list of True/False values to the *ascending* argument. Verify that the highest-paid accountant in the automotive industry made \$49.32 per hour:

In [17]:

```
sortedData3 = myData.sort_values(['Industry', 'Job', 'HourlyWage'], ascending
 = [True, True, False])

sortedData3.head()
```

Out[17]:

	EmployeeID	HourlyWage	Industry	Job
476	477	49.32	Automotive	Accountant
234	235	48.97	Automotive	Accountant
135	136	48.56	Automotive	Accountant
321	322	48.00	Automotive	Accountant
392	393	48.00	Automotive	Accountant

l. To sort the data by industry in descending order and then by job classification and hourly wage in ascending order, we can use the following command:

In [18]:

```
sortedData4 = myData.sort_values(['Industry', 'Job', 'HourlyWage'], ascending
 = [False, True, True])

sortedData4.head(n = 12)
```

Out[18]:

	EmployeeID	HourlyWage	Industry	Job
527	528	36.13	Tech	Accountant

550	551	40.48	Tech	Accountant
559	560	40.48	Tech	Accountant
511	512	41.11	Tech	Accountant
42	43	41.26	Tech	Accountant
237	238	41.26	Tech	Accountant
502	503	42.21	Tech	Accountant
539	540	47.13	Tech	Accountant
291	292	48.87	Tech	Accountant
140	141	49.49	Tech	Accountant
570	571	35.43	Tech	Consultant
301	302	40.48	Tech	Consultant

As shown in the result, the highest-paid accountants in the technology industry made \$36.13 and \\$49.49 per hour, respectively.

m. To sort the data by industry and hourly wage (in ascending order) for only accountants who worked in the automotive and and the tech industries, we can use the Pandas **query** function in conjunction with selection criteria. In the code below, we select observations in which the job is accountant and the industry is automotive, sorting the result by industry and hourly wage:

In [19]:

```
sortedData4 = myData.query("Job == 'Accountant' & Industry == 'Automotive'").
sort_values(['Industry', 'HourlyWage'])
```

Note that we needed to place single quotes within double quotes when defining our query for it to execute succesfully.

Using **head** with the *n* argument set to one, we can see the lowest-paid accountant in the automotive industry made \$28.74 per hour:

In [20]:

```
sortedData4.head(n = 1)
```

Out[20]:

	EmployeeID	HourlyWage	Industry	Job
566	567	28.74	Automotive	Accountant

Similarly, using **tail** we can see the highest-paid accountant in the automotive industry made \$49.32 per hour:

In [21]:

```
sortedData4.tail(n = 1)
```

Out[21]:

	EmployeeID	HourlyWage	Industry	Job
476	477	49.32	Automotive	Accountant

By adjusting the command to select accountants in the tech industry, we can pull up similar results for those employees:

In [22]:

```
sortedData5 = myData.query("Job == 'Accountant' & Industry == 'Tech'").sort_values(['Industry', 'HourlyWage'])

sortedData5
```

Out[22]:

	EmployeeID	HourlyWage	Industry	Job
527	528	36.13	Tech	Accountant
550	551	40.48	Tech	Accountant
559	560	40.48	Tech	Accountant
511	512	41.11	Tech	Accountant
42	43	41.26	Tech	Accountant
237	238	41.26	Tech	Accountant
502	503	42.21	Tech	Accountant
539	540	47.13	Tech	Accountant
291	292	48.87	Tech	Accountant
140	141	49.49	Tech	Accountant

As shown in the result, the lowest- and highest-paid accountants in the tech industry made \$36.13 and \\$49.49 per hour, respectively.

Finally, to see automative and tech accountants in a single set of results, we can adjust our query by supplying a list of industry types to select using the in operator:

In [23]:

```
sortedData6 = myData.query("Job == 'Accountant' & Industry in ['Automotive', 'Tech']").sort_values(['Industry', 'HourlyWage'])

sortedData6
```

Out[23]:

	EmployeeID	HourlyWage	Industry	Job
566	567	28.74	Automotive	Accountant
76	77	29.00	Automotive	Accountant
573	574	32.10	Automotive	Accountant
35	36	38.28	Automotive	Accountant
475	476	39.67	Automotive	Accountant
552	553	39.67	Automotive	Accountant
535	536	41.06	Automotive	Accountant
324	325	41.26	Automotive	Accountant
599	600	41.26	Automotive	Accountant
313	314	41.37	Automotive	Accountant
66	67	41.50	Automotive	Accountant
58	59	42.92	Automotive	Accountant

58	59	42.92	Automotive	Accountant
74	75	42.92	Automotive	Accountant
83	84	42.92	Automotive	Accountant
429	430	42.92	Automotive	Accountant
547	548	42.92	Automotive	Accountant
4	5	43.62	Automotive	Accountant
235	236	44.54	Automotive	Accountant
229	230	46.10	Automotive	Accountant
374	375	47.00	Automotive	Accountant
321	322	48.00	Automotive	Accountant
392	393	48.00	Automotive	Accountant
135	136	48.56	Automotive	Accountant
234	235	48.97	Automotive	Accountant
476	477	49.32	Automotive	Accountant
527	528	36.13	Tech	Accountant
550	551	40.48	Tech	Accountant
559	560	40.48	Tech	Accountant
511	512	41.11	Tech	Accountant
42	43	41.26	Tech	Accountant
237	238	41.26	Tech	Accountant
502	503	42.21	Tech	Accountant
539	540	47.13	Tech	Accountant
291	292	48.87	Tech	Accountant
140	141	49.49	Tech	Accountant

Alternatively, we could have used the | (binary or) operator to show the same result:

In [24]:

```
sortedData7 = myData.query("Job == 'Accountant' & (Industry == 'Automotive' | Industry == 'Tech')").sort_values(['Industry', 'HourlyWage'])

sortedData7
```

Out[24]:

	EmployeeID	HourlyWage	Industry	Job
566	567	28.74	Automotive	Accountant
76	77	29.00	Automotive	Accountant
573	574	32.10	Automotive	Accountant
35	36	38.28	Automotive	Accountant
475	476	39.67	Automotive	Accountant
552	553	39.67	Automotive	Accountant
535	536	41.06	Automotive	Accountant
324	325	41.26	Automotive	Accountant
599	600	41.26	Automotive	Accountant
313	314	41.37	Automotive	Accountant

66	67	41.50	Automotive	Accountant
58	59	42.92	Automotive	Accountant
74	75	42.92	Automotive	Accountant
83	84	42.92	Automotive	Accountant
429	430	42.92	Automotive	Accountant
547	548	42.92	Automotive	Accountant
4	5	43.62	Automotive	Accountant
235	236	44.54	Automotive	Accountant
229	230	46.10	Automotive	Accountant
374	375	47.00	Automotive	Accountant
321	322	48.00	Automotive	Accountant
392	393	48.00	Automotive	Accountant
135	136	48.56	Automotive	Accountant
234	235	48.97	Automotive	Accountant
476	477	49.32	Automotive	Accountant
527	528	36.13	Tech	Accountant
550	551	40.48	Tech	Accountant
559	560	40.48	Tech	Accountant
511	512	41.11	Tech	Accountant
42	43	41.26	Tech	Accountant
237	238	41.26	Tech	Accountant
502	503	42.21	Tech	Accountant
539	540	47.13	Tech	Accountant
291	292	48.87	Tech	Accountant
140	141	49.49	Tech	Accountant

n. To export the sorted data from the final example in step n as a comma-separated value file, use the Pandas **to_csv** function. You will need to supply an appropriate file name/path and you will likely want to set the *index* argument to False to omit it from the output:

In [25]:

```
sortedData7.to_csv('sortedData7.csv', index = False)
```

Example 2.2

Sarah Johnson the manager of a local restaurant, has conducted a survey to gauge customers' perception about the eatery. Each customer rated the restaurant on its ambience, cleanliness, service, and food using a scale of 1 (lowest) to 7 (highest). Table 2.4 below displays a portion of the survey data.

Table 2.4 Restaurant Reviews

RecordNum	Ambience	Cleanliness	Service	Food
1	4	5	6	4
2	6	6		6
...	...	...	...	...

150	3	5	6	7

Sarah notices that there are a number of missing values in the survey. Use the ***Restaurant_Reviews*** data to first detect the missing values. Then use both ommission and imputation strategies to handle the missing values.

a. Import the ***Restaurant_Reviews*** data file into a Pandas DataFrame (table) and label it myData.

In [26]:

```
import pandas as pd

myData = pd.read_excel('jaggia_ba_1e_ch02_Data_Files.xlsx', sheet_name = 'Res
taurant_Reviews')

myData.columns = myData.columns.str.strip() # note: fixes trailing space in S
ervice column in Excel data file

myData.head()
```

Out[26]:

	RecordNum	Ambience	Cleanliness	Service	Food
0	1	4.0	5.0	6.0	4.0
1	2	6.0	6.0	NaN	6.0
2	3	5.0	6.0	6.0	7.0
3	4	4.0	7.0	5.0	4.0
4	5	4.0	7.0	7.0	4.0

b. The Pandas **isna** function can be used to display missing values across the entire DataFrame. Missing (NaN) data will appear as True whereas non-missing values will appear as False:

In [27]:

```
pd.isna(myData)
```

Out[27]:

	RecordNum	Ambience	Cleanliness	Service	Food
0	False	False	False	False	False
1	False	False	False	True	False
2	False	False	False	False	False
3	False	False	False	False	False
4	False	False	False	False	False
...	...	...	...	...	...
145	False	False	False	False	False
146	False	False	False	False	False
147	False	False	False	False	False
148	False	False	False	False	False
149	False	False	False	False	False

150 rows × 5 columns

To detect and sum missing values in a specific column, used the **sum** function in conjunctio with Pandas **isna** as introduced in Example 2.1 part d above:

In [28]:

```
pd.isna(myData.Service).sum()
```

Out[28]:

```
3
```

c. To implement the omission strategy and keep only those rows that contain no missing values, use the Pandas **dropna** function:

In [29]:

```
omissionData = myData.dropna()

omissionData
```

Out[29]:

	RecordNum	Ambience	Cleanliness	Service	Food
0	1	4.0	5.0	6.0	4.0
2	3	5.0	6.0	6.0	7.0
3	4	4.0	7.0	5.0	4.0
4	5	4.0	7.0	7.0	4.0
5	6	3.0	5.0	6.0	7.0
...	...	...	...	...	...
145	146	5.0	6.0	6.0	3.0
146	147	4.0	6.0	5.0	6.0
147	148	5.0	6.0	7.0	5.0
148	149	5.0	5.0	5.0	4.0
149	150	3.0	5.0	6.0	7.0

145 rows × 5 columns

d. As shown above, only five rows contained missing data (i.e., 145 after dropping missing data as compared to 150 in the original). To see the rows with missing data, incorporate the **any** function with the *axis* argument set to 1 (i.e., where 1 refers columns):

In [30]:

```
myData[myData.isna().any(axis = 1)]
```

Out[30]:

	RecordNum	Ambience	Cleanliness	Service	Food
1	2	6.0	6.0	NaN	6.0
12	13	6.0	NaN	7.0	5.0
25	26	6.0	7.0	5.0	NaN
99	100	6.0	6.0	NaN	3.0
133	134	NaN	5.0	NaN	6.0

e. To implement a basic mean imputation strategy for each column, use the Pandas **fillna** and **mean** functions:

In [31]:

```
imputationData = myData.fillna(myData.mean())

imputationData
```

Out[31]:

	RecordNum	Ambience	Cleanliness	Service	Food
0	1	4.0	5.0	6.000000	4.0
1	2	6.0	6.0	5.965986	6.0
2	3	5.0	6.0	6.000000	7.0
3	4	4.0	7.0	5.000000	4.0
4	5	4.0	7.0	7.000000	4.0
...	...	...	...	...	...
145	146	5.0	6.0	6.000000	3.0
146	147	4.0	6.0	5.000000	6.0
147	148	5.0	6.0	7.000000	5.0
148	149	5.0	5.0	5.000000	4.0
149	150	3.0	5.0	6.000000	7.0

150 rows × 5 columns

Alternatively, you could impute using another function such as **median**:

In [32]:

```
myData.fillna(myData.median())
```

Out[32]:

	RecordNum	Ambience	Cleanliness	Service	Food
0	1	4.0	5.0	6.0	4.0
1	2	6.0	6.0	6.0	6.0
2	3	5.0	6.0	6.0	7.0
3	4	4.0	7.0	5.0	4.0
4	5	4.0	7.0	7.0	4.0
...	...	...	...	...	...
145	146	5.0	6.0	6.0	3.0
146	147	4.0	6.0	5.0	6.0
147	148	5.0	6.0	7.0	5.0
148	149	5.0	5.0	5.0	4.0
149	150	3.0	5.0	6.0	7.0

150 rows × 5 columns

Example 2.3

In the introductory case, Catherine Hill wants to gain a better understanding of Organic Food Superstore's customers who are college-educated millenials, born between 1982 and 2000. She feels that sex, household size, income, total spending in 2018, total number of orders in the past 24 months, and channel through which the customer was acquired are useful for her to create a profile of these customers. Use Python to first identify college-educated millenial customers in the ***Customers*** data file. Then create subsets of female and male college-educated millenial customers. The synopsis that follows this example provides a summary of the results.

a. Import the ***Customers*** data file into a Pandas DataFrame (table) and label it myData.

In [33]:

```
import pandas as pd

myData = pd.read_excel('jaggia_ba_1e_ch02_Data_Files.xlsx', sheet_name = 'Cus
tomers')
```

b. Pull up the data to get a basic overview, including dimensions:

In [34]:

```
myData
```

Out[34]:

	CustID	Sex	Race	BirthDate	College	HouseholdSize	ZipCode	Income
0	1530016	Female	Black	1986-12-16	Yes	5	90047	53000
1	1531136	Male	White	1993-05-09	Yes	5	90026	94000
2	1532160	Male	Black	1966-05-22	Yes	2	90027	64000
3	1532307	Male	White	1964-09-16	Yes	4	90029	60000
4	1532356	Female	Hispanic	1964-07-15	No	5	90017	47000
...	...	...	...	...	...	...	...	...
195	1578525	Male	Hispanic	1963-12-12	Yes	1	90005	82000
196	1579349	Male	Asian	1980-12-19	Yes	1	90010	49000
197	1579389	Female	American Indian	2000-05-21	Yes	1	90009	50000
198	1579857	Female	White	1991-01-26	Yes	1	90055	52000
199	1579979	Male	White	1999-07-05	Yes	5	90043	102000

200 rows × 14 columns

c. To select college age millennials, we use the **query** approach introduced in Example 2.1 part m above. Specifically, we filter the data to return only those rows where the College value is "Yes" and the BirthDate is between January 1, 1982 and December 31, 1999 (inclusive using the >= and <=

operators). Note that dates in Python are formated as YEAR-MONTH-DAY by default (e.g., January 1, 1982 would be written as '1982-01-01'):

In [35]:

```
collegeAgeMillenials1 = myData.query("College == 'Yes' & BirthDate >= '1982-01-01' & BirthDate <= '1999-12-31'")

collegeAgeMillenials1.head()
```

Out[35]:

	CustID	Sex	Race	BirthDate	College	HouseholdSize	ZipCode	Income	Sp
0	1530016	Female	Black	1986-12-16	Yes	5	90047	53000	
1	1531136	Male	White	1993-05-09	Yes	5	90026	94000	
6	1533017	Female	Hispanic	1985-05-14	Yes	3	90063	84000	
10	1533791	Male	White	1999-10-27	Yes	1	90060	97000	
11	1533917	Female	Black	1993-03-03	Yes	3	90045	64000	

d. To check the number of filtered rows and verify the result, use the **shape** property to see the number of college-educated millinnials in the data set (i.e., 59):

In [36]:

```
collegeAgeMillenials1.shape
```

Out[36]:

```
(59, 14)
```

e. To include only the Sex, HouseholdSize, Income, Spending2018, NumOfOrders, and Channel variables in this new DataFrame, we can extract only those columns by referencing a list of column names:

In [37]:

```
collegeAgeMillenials2 = collegeAgeMillenials1[['Sex', 'HouseholdSize', 'Income', 'Spending2018', 'NumOfOrders', 'Channel']]

collegeAgeMillenials2.head()
```

Out[37]:

	Sex	HouseholdSize	Income	Spending2018	NumOfOrders	Channel
0	Female	5	53000	241	3	SM
1	Male	5	94000	843	12	TV
6	Female	3	84000	153	2	Web
10	Male	1	97000	1028	17	Web
11	Female	3	64000	915	15	Referral

f. It can be useful to use the Pandas **dtypes** property to check the data type of each column in the DataFrame:

In [38]:

```
collegeAgeMillenials2.dtypes
```

Out[38]:

```
Sex                object
HouseholdSize       int64
Income              int64
Spending2018        int64
NumOfOrders         int64
Channel            object
dtype: object
```

As shown above, by default, Pandas will assume a generic `object` data type for non-numerical variables such as Sex and Channel, which contain text sequences. To convert these to categorical data, use the Pandas **astype** function and the following command:

In [39]:

```
collegeAgeMillenials3 = collegeAgeMillenials2.astype({'Sex': 'category', 'Cha
nnel': 'category'})

collegeAgeMillenials3.dtypes
```

Out[39]:

```
Sex                category
HouseholdSize         int64
Income                int64
Spending2018          int64
NumOfOrders           int64
Channel            category
dtype: object
```

To verify the Sex variable has been converted, pull up a random sample using the Pandas **sample** function (in this case, with 5 instances specified):

In [40]:

```
collegeAgeMillenials3.Sex.sample(5)
```

Out[40]:

```
12        Male
85      Female
62        Male
11      Female
120       Male
Name: Sex, dtype: category
Categories (2, object): [Female, Male]
```

g. To split the data based on Sex, one option is to use **query** to filter the data using the categories shown above:

In [41]:

```
sexFemale = collegeAgeMillenials3.query("Sex == 'Female'")
sexMale = collegeAgeMillenials3.query("Sex == 'Male'")
```

The result is 21 females:

In [42]:

```
sexFemale.head(5)
```

Out[42]:

	Sex	HouseholdSize	Income	Spending2018	NumOfOrders	Channel
0	Female	5	53000	241	3	SM
6	Female	3	84000	153	2	Web
11	Female	3	64000	915	15	Referral
14	Female	3	42000	313	4	TV
50	Female	5	97000	911	16	Web

In [43]:

```
sexFemale.shape
```

Out[43]:

```
(21, 6)
```

And 38 males:

In [44]:

```
sexMale.head(5)
```

Out[44]:

	Sex	HouseholdSize	Income	Spending2018	NumOfOrders	Channel
1	Male	5	94000	843	12	TV
10	Male	1	97000	1028	17	Web
12	Male	2	114000	665	7	TV
22	Male	2	94000	524	7	Referral
28	Male	1	91000	800	10	Web

In [45]:

```
sexMale.shape
```

Out[45]:

```
(38, 6)
```

For a more complex, programmatic alternative, you could also use Python list comprehension to achieve the same split for each category based on groups using Pandas **groupby** function:

In [46]:

```
splitData = [group for _, group in collegeAgeMillenials3.groupby('Sex')]
```

Using a for loop, we can pull up a sample of each data set using the following command:

In [47]:

```
for eachSplit in splitData:
    display(eachSplit.head())
```

	Sex	HouseholdSize	Income	Spending2018	NumOfOrders	Channel
0	Female	5	53000	241	3	SM
6	Female	3	84000	153	2	Web
11	Female	3	64000	915	15	Referral
14	Female	3	42000	313	4	TV
50	Female	5	97000	911	16	Web

	Sex	HouseholdSize	Income	Spending2018	NumOfOrders	Channel
1	Male	5	94000	843	12	TV
10	Male	1	97000	1028	17	Web
12	Male	2	114000	665	7	TV
22	Male	2	94000	524	7	Referral
28	Male	1	91000	800	10	Web

We could also assign the splits to data sets directly using a modified command:

In [48]:

```
sexFemale, sexMale = [group for _, group in collegeAgeMillenials3.groupby('Se
x')]

sexFemale.head()
```

Out[48]:

	Sex	HouseholdSize	Income	Spending2018	NumOfOrders	Channel
0	Female	5	53000	241	3	SM
6	Female	3	84000	153	2	Web
11	Female	3	64000	915	15	Referral
14	Female	3	42000	313	4	TV
50	Female	5	97000	911	16	Web

In [49]:

```
sexMale.head()
```

Out[49]:

	Sex	HouseholdSize	Income	Spending2018	NumOfOrders	Channel
1	Male	5	94000	843	12	TV
10	Male	1	97000	1028	17	Web
12	Male	2	114000	665	7	TV
22	Male	2	94000	524	7	Referral
28	Male	1	91000	800	10	Web

As with most tasks, there are many ways to split data in Python.

h. In some situations, we might simply want to subset data based on data ranges. For example, we can use NumPy's **r_** method (i.e., **np.r_**) to subset data to include observations 1 to 50 and observations 101 to 200. Enter:

In [50]:

```
import numpy as np

myData.iloc[np.r_[0:50, 100:200]]
```

Out[50]:

	CustID	Sex	Race	BirthDate	College	HouseholdSize	ZipCode	Income
0	1530016	Female	Black	1986-12-16	Yes	5	90047	53000
1	1531136	Male	White	1993-05-09	Yes	5	90026	94000
2	1532160	Male	Black	1966-05-22	Yes	2	90027	64000
3	1532307	Male	White	1964-09-16	Yes	4	90029	60000
4	1532356	Female	Hispanic	1964-07-15	No	5	90017	47000
...	...	...	...	...	...	...	...	...
195	1578525	Male	Hispanic	1963-12-12	Yes	1	90005	82000
196	1579349	Male	Asian	1980-12-19	Yes	1	90010	49000
197	1579389	Female	American Indian	2000-05-21	Yes	1	90009	50000
198	1579857	Female	White	1991-01-26	Yes	1	90055	52000
199	1579979	Male	White	1999-07-05	Yes	5	90043	102000

150 rows × 14 columns

To reference rows 1 to 50, we provide the slice `0:50`, where the start index (0) is inclusive and the stop index (50) is exclusive; for rows 101 to 200 we use the slice `100:200`. In terms of why these start index values, keep in mind the row index starts at 0 in Python (e.g., MyData contains 200 rows with index values 0 to 199 as shown in part b above).

Example 2.4

In order to better understand her customers, Catherine Hill would like to perform the RFM analysis, a popular marketing technique used to identify high-value customers. RFM stands for **r**ecency, **f**requency, and **m**onetary. The RFM ratings can be created from the DaysSinceLast (recency), NumOfOrders (frequency), and Spending2018 (monetary) variables.

Following the 80/20 business rule (i.e., 80% of your business comes from 20% of your best customers), for each of the three RFM variables, Catherine would like to bin customers into five equal-size groups, with 20% of the customers included in each group. Each group is also assigned a score from 1 to 5, with 5 being the highest. Customers with the RFM rating of 555 are considered the most valuable customers to the company.

In addition to the RFM binning, Catherine would like to bin the Income variable into five equal intervals. Finally, she would like to start a tiered membership status where different services and rewards are offered to customers depending on how much they spent in 2018. She would like to assign the bronze membership status to customers who spent less than \$250, silver membership status to those who spent \\$250 or more but less than \$1,000, and the gold membership status to those who spent \\$1,000 or more.

Use Python to bin variables according to Catherine's specifications. Summarize the results.

a. Import the ***Customers*** data file into a Pandas DataFrame (table) and label it myData. Pull up the data to get a basic overview, including dimensions:

In [51]:

```
import pandas as pd

myData = pd.read_excel('jaggia_ba_1e_ch02_Data_Files.xlsx', sheet_name = 'Cus
tomers')

myData
```

Out[51]:

	CustID	Sex	Race	BirthDate	College	HouseholdSize	ZipCode	Income	!
0	1530016	Female	Black	1986-12-16	Yes	5	90047	53000	
1	1531136	Male	White	1993-05-09	Yes	5	90026	94000	
2	1532160	Male	Black	1966-05-22	Yes	2	90027	64000	
3	1532307	Male	White	1964-09-16	Yes	4	90029	60000	
4	1532356	Female	Hispanic	1964-07-15	No	5	90017	47000	
...	...	...	...	...	...	...	...	...	
195	1578525	Male	Hispanic	1963-12-12	Yes	1	90005	82000	
196	1579349	Male	Asian	1980-12-19	Yes	1	90010	49000	
197	1579389	Female	American Indian	2000-05-21	Yes	1	90009	50000	
198	1579857	Female	White	1991-01-26	Yes	1	90055	52000	
199	1579979	Male	White	1999-07-05	Yes	5	90043	102000	

200 rows × 14 columns

b. To create the recency score, we first transform the variable DaysSinceLast to reverse the order of the data because the fewer the number of days since the last purchase, the greater the recency score. Create a new variable called DaysSinceLastReverse by multiplying DaysSinceLast by -1:

In [52]:

```
myData['DaysSinceLastReverse'] = myData.DaysSinceLast * -1
```

c. We now need to create five equal-sized bins for DaysSinceLastReverse (recency), NumOfOrders (frequency), and Spending2018 (monetary). The Pandas **qcut** function can directly create five equal size buckets based on quantile cutoffs (i.e., at 20%, 40%, 60%, 80%, and 100%) for each variable using the *q* argument. For example, consider the following binning of DaysSinceLastReverse (recency):

In [53]:

```
pd.qcut(myData.DaysSinceLastReverse, q = 5)
```

Out[53]:

```
0          (-146.8, -76.0]
1         (-294.2, -218.4]
2          (-146.8, -76.0]
3          (-146.8, -76.0]
4          (-146.8, -76.0]
                ...
195     (-360.001, -294.2]
196       (-218.4, -146.8]
197       (-294.2, -218.4]
198       (-294.2, -218.4]
199        (-146.8, -76.0]
Name: DaysSinceLastReverse, Length: 200, dtype: category
Categories (5, interval[float64]): [(-360.001, -294.2] < (-294.2,
-218.4] < (-218.4, -146.8] < (-146.8, -76.0] < (-76.0, -6.0]]
```

The output shows the respective cutoffs for each of the five quantile bins based on the data (i.e., see Categories). Using the **qcut** approach, we can bin each of the three variables for RFM and also label them as needed using the **range** function within the *labels* argument to **qcut**:

In [54]:

```
myData['Recency'] = pd.qcut(myData.DaysSinceLastReverse, q = 5, labels = rang
e(1, 6))
myData['Frequency'] = pd.qcut(myData.NumOfOrders, q = 5, labels = range(1, 6
))
myData['Monetary'] = pd.qcut(myData.Spending2018, q = 5, labels = range(1, 6
))
```

Note that we use `range(1, 6)` to generate the list of integers from 1 to 5 (i.e., `[1, 2, 3, 4, 5]`) -- the second argument is 6 because range is not inclusive.

d. To create the RFM score, we concatenate the three RFM variables we just created using the + operator. However, note that to concatenate, we must convert the data type to character strings (i.e., `str`) temporarily to make it work using the Pandas **astype** function introduced in Example 2.3 part e above:

In [55]:

```
myData['RFM'] = myData.Recency.astype('str') + myData.Frequency.astype('str')
 + myData.Frequency.astype('str')
```

To verify the result, we can pull up the head of the RFM variable:

In [56]:

```
myData.RFM.head()
```

Out[56]:

```
0    411
1    244
2    433
3    444
4    422
Name: RFM, dtype: object
```

e. We now bin the Income variable into five groups with equal intervals using the Pandas **cut** function with the *bins* argument set to 5. First, let us run the command in isolation to verify that the cuts meet our specifications:

In [57]:

```
pd.cut(myData.Income, bins = 5)
```

Out[57]:

```
0        (30864.0, 58200.0]
1       (85400.0, 112600.0]
2        (58200.0, 85400.0]
3        (58200.0, 85400.0]
4        (30864.0, 58200.0]
               ...
195      (58200.0, 85400.0]
196      (30864.0, 58200.0]
197      (30864.0, 58200.0]
198      (30864.0, 58200.0]
199     (85400.0, 112600.0]
Name: Income, Length: 200, dtype: category
Categories (5, interval[float64]): [(30864.0, 58200.0] < (58200.
0, 85400.0] < (85400.0, 112600.0] < (112600.0, 139800.0] < (13980
0.0, 167000.0]]
```

Similar to Pandas **qcut** above the Categories in the output show the bin cutoffs.

To store the binned data back into the data set using labels from 1 to 5, we can again use the **range** method from earlier:

In [58]:

```
myData['BinnedIncome'] = pd.cut(myData.Income, bins = 5, labels = range(1, 6
))
```

f. To size the size of each bin, use the Pandas **size** function in conjunction with **groupby**:

In [59]:

```
myData.groupby('BinnedIncome').size()
```

Out[59]:

```
BinnedIncome
1     67
2     72
3     52
4      6
5      3
dtype: int64
```

Note that the first bin has 67 customers.

g. To create the membership tiers or user-defined bins proposed by Catherine, we can use the Pandas **cut** function again while specifying our own custom bin cutoffs via a list, along with labels for the respective tiers:

In [60]:

```
myData['MembershipTier'] = pd.cut(myData.Spending2018, bins = [0, 250, 1000,
```

```
float('inf')], labels = ['Bronze', 'Silver', 'Gold'])
```

Recall that Catherine suggested the following breakdown for the tiers:

- Bronze: less than \$250
- Silver: \$250 or more but less than \\$1,000
- Gold: \$1,000 or more

Based on the *bins* argument, the Bronze data will fall between the first two values based on spending (i.e., between 0 and 250), whereas the Gold data will fall between the last two values (i.e., 1000 and `float('inf')`, which represents infinity).

To verify the result, use the Pandas **head** function:

In [61]:

```
myData.MembershipTier.head()
```

Out[61]:

```
0    Bronze
1    Silver
2    Silver
3    Silver
4    Silver
Name: MembershipTier, dtype: category
Categories (3, object): [Bronze < Silver < Gold]
```

Example 2.5

After a closer review of her customers, Catherine Hill feels that the difference and the percentage difference between a customer's 2017 and 2018 spending may be more useful to understanding the customer's spending patterns than the yearly spending values. Therefore, Catherine wants to generate two new variables that capture the year-to-year difference and the percentage difference in spending. She also notices that the income variable is highly skewed, with most customers' incomes falling between \$40,000 and \\$100,000, with only a few very-high-income earners. She has been advised to transform the income variable into natural logarithms, which will reduce the skewness of the data.

Catherine would also like to convert customer birthdates into ages as of January 1, 2019, for exploring differences in purchase behaviors of customers across age groups. Finally she would like to create a new variable that captures the birth month of the customers so that seasonal products can be marketed to these customers during their birth month.

Use Python to transform variables according to Catherine's specifications.

a. Import the ***Customers*** data file into a Pandas DataFrame (table) and label it myData. Pull up the data to get a basic overview, including dimensions:

In [62]:

```
import pandas as pd

myData = pd.read_excel('jaggia_ba_1e_ch02_Data_Files.xlsx', sheet_name = 'Cus
tomers')

myData
```

Out[62]:

	CustID	Sex	Race	BirthDate	College	HouseholdSize	ZipCode	Income

0	1530016	Female	Black	1986-12-16	Yes		5	90047	53000
1	1531136	Male	White	1993-05-09	Yes		5	90026	94000
2	1532160	Male	Black	1966-05-22	Yes		2	90027	64000
3	1532307	Male	White	1964-09-16	Yes		4	90029	60000
4	1532356	Female	Hispanic	1964-07-15	No		5	90017	47000
...	...	...	...	...	...		...	...	...
195	1578525	Male	Hispanic	1963-12-12	Yes		1	90005	82000
196	1579349	Male	Asian	1980-12-19	Yes		1	90010	49000
197	1579389	Female	American Indian	2000-05-21	Yes		1	90009	50000
198	1579857	Female	White	1991-01-26	Yes		1	90055	52000
199	1579979	Male	White	1999-07-05	Yes		5	90043	102000

200 rows × 14 columns

b. Using some basic arithmetic, we can find the spending difference and view the **head** of the observations using Pandas:

In [63]:

```
myData['SpendingDiff'] = myData.Spending2018 - myData.Spending2017

myData.SpendingDiff.head()
```

Out[63]:

```
0     -46
1    -384
2     196
3      66
4     290
Name: SpendingDiff, dtype: int64
```

Note the first customer has a SpendingDiff of -46.

c. To create a variable with the SpendingDiff as a percentage, we again use basic arithmetic and subsequently round the result using the **round** function:

In [64]:

```
myData['PctSpendingDiff'] = round((myData.SpendingDiff / myData.Spending2017)
 * 100)

myData.PctSpendingDiff.head()
```

Out[64]:

```
0    -16.0
1    -31.0
2     37.0
3     13.0
```

```
4     52.0
Name: PctSpendingDiff, dtype: float64
```

Alternatively, to include a % symbol in the new column, we can adjust our code above to include string concatenation after we convert the data type to `str` using the Pandas **astype** function introduced in Example 2.3 part e above:

In [65]:

```
myData['PctSpendingDiff'] = round((myData.SpendingDiff / myData.Spending2017)
 * 100).astype('str') + '%'

myData.PctSpendingDiff.head()
```

Out[65]:

```
0    -16.0%
1    -31.0%
2     37.0%
3     13.0%
4     52.0%
Name: PctSpendingDiff, dtype: object
```

In either case, the first observation of PctSpendingDiff is -16.0%.

d. To take the natural logarithm of Income, we use the NumPy **log**:

In [66]:

```
import numpy as np

myData['IncomeLn'] = np.log(myData.Income)

myData.IncomeLn.head()
```

Out[66]:

```
0    10.878047
1    11.451050
2    11.066638
3    11.002100
4    10.757903
Name: IncomeLn, dtype: float64
```

e. To calculate a customer's age as of January 1, 2019, we can use:

- Pandas **to_datetime** function to define the data to calculate from
- NumPy's **timedelta64** function to define the time difference as 1 year
- NumPy's **floor** function to remove decimal places in the result

The following code will perform the calculation:

In [67]:

```
myData['Age'] = np.floor((pd.to_datetime('2019-01-01') - myData.BirthDate)/np
.timedelta64(1, 'Y'))

myData.Age.head()
```

Out[67]:

```
0    32.0
1    25.0
```

```
2     52.0
3     54.0
4     54.0
Name: Age, dtype: float64
```

Note that in the **to_datetime** function, the argument representing the date is specified in Python's default YEAR-MONTH-DATE format (i.e., YYYY-MM-DD). In addition, the time difference calculated using **timedelta64** is set to calculate relative to 1 year, using the arguments 1 and 'Y', respectively.

Moreover, to supply a date using a different format, consider using the *format* argument:

In [68]:

```
myData['Age'] = np.floor((pd.to_datetime('01/01/2019', format = '%m/%d/%Y') -
 myData.BirthDate)/np.timedelta64(1, 'Y'))

myData.Age.head()
```

Out[68]:

```
0     32.0
1     25.0
2     52.0
3     54.0
4     54.0
Name: Age, dtype: float64
```

In either case, the result is the same, and the first customer's age as of January 1, 2019 is 32 years.

f. To extract the month number from BirthDate, we can use the **months** property for dates:

In [69]:

```
myData['BirthMonth'] = pd.DatetimeIndex(myData.BirthDate).month

myData.BirthMonth.head()
```

Out[69]:

```
0     12
1      5
2      5
3      9
4      7
Name: BirthMonth, dtype: int64
```

The first customer was born in month 12 (i.e., December).

g. The Python datetime module contains many useful functions for working with dates. For example, using the **strftime** function, you can also extract the name of the month if needed:

In [70]:

```
import datetime as dt

myData.BirthDate.dt.strftime('%b').head()
```

Out[70]:

```
0     Dec
1     May
2     May
3     Sep
```

```
4    Jul
Name: BirthDate, dtype: object
```

Alternatively, to get the date, month number, or year, use '%d', '%m', or '%Y', respectively:

In [71]:

```
myData.BirthDate.dt.strftime('%d').head()  # date
```

Out[71]:

```
0    16
1    09
2    22
3    16
4    15
Name: BirthDate, dtype: object
```

In [72]:

```
myData.BirthDate.dt.strftime('%m').head()  # month
```

Out[72]:

```
0    12
1    05
2    05
3    09
4    07
Name: BirthDate, dtype: object
```

In [73]:

```
myData.BirthDate.dt.strftime('%Y').head()  # year
```

Out[73]:

```
0    1986
1    1993
2    1966
3    1964
4    1964
Name: BirthDate, dtype: object
```

To see the current date and time or date, you one option is to use the Pandas **to_datetime** function:

In [74]:

```
pd.to_datetime('today')  # datetime
```

Out[74]:

```
Timestamp('2020-07-15 17:06:54.960117')
```

In [75]:

```
pd.to_datetime('today').strftime('%m/%d/%Y')  # extract the date as a string
```

Out[75]:

```
'07/15/2020'
```

Plenty of other options are also available Python datetime module.

Example 2.6

After gaining some insights from the ***Customers*** data set, Catherine would like to analyze race. However, in its current form, the data set would limit her ability to do a meaningful analysis given the large number of categories of the race variable; plus some categories have very few observations. As a result, she needs to perform a series of data transformations to prepare the data for subsequent analysis. Use Python to create a new category called Other that represents the two least-frequent categories.

a. Import the ***Customers*** data file into a Pandas DataFrame (table) and label it myData. Pull up the data to get a basic overview, including dimensions:

In [76]:

```
import pandas as pd

myData = pd.read_excel('jaggia_ba_1e_ch02_Data_Files.xlsx', sheet_name = 'Cus
tomers')

myData
```

Out[76]:

	CustID	Sex	Race	BirthDate	College	HouseholdSize	ZipCode	Income	S
0	1530016	Female	Black	1986-12-16	Yes	5	90047	53000	
1	1531136	Male	White	1993-05-09	Yes	5	90026	94000	
2	1532160	Male	Black	1966-05-22	Yes	2	90027	64000	
3	1532307	Male	White	1964-09-16	Yes	4	90029	60000	
4	1532356	Female	Hispanic	1964-07-15	No	5	90017	47000	
...	...	...	...	...	...	...	...	...	
195	1578525	Male	Hispanic	1963-12-12	Yes	1	90005	82000	
196	1579349	Male	Asian	1980-12-19	Yes	1	90010	49000	
197	1579389	Female	American Indian	2000-05-21	Yes	1	90009	50000	
198	1579857	Female	White	1991-01-26	Yes	1	90055	52000	
199	1579979	Male	White	1999-07-05	Yes	5	90043	102000	

200 rows × 14 columns

b. Use the Pandas **groupby** and **size** functions introduced in Example 2.4 part f to inspect the frequency of each Race category to identify the two least frequent categories:

In [77]:

```
myData.groupby('Race').size()
```

Out[77]:

```
Race
```

```
American Indian       5
Asian                15
Black                57
Hispanic             41
Pacific Islander      3
White                79
dtype: int64
```

The output shows that American Indians and Pacific Islanders are the two least-frequent categories with only five and three observations, respectively.

c. Using the NumPy **where** function introduced in [Example 2.1] part e and the Pandas **isin** function, we can recode the Other category to represent the two least-frequent categories identified above:

In [78]:

```
import numpy as np

myData['NewRace'] = np.where(myData.Race.isin(['American Indian', 'Pacific Is
lander']), 'Other', myData.Race)
```

The **isin** function in the code above is used to check if each Race value is in the list of values provided (i.e., American Indian and Pacific Islander in this example). This condition is subsequently used with the **where** function to replace values with 'Other' when the check returns True and with the original data from myData.race when the check returns False.

d. To verify the recoding of the data was succesful, we can create a new frequency table:

In [79]:

```
myData.groupby('NewRace').size()
```

Out[79]:

```
NewRace
Asian       15
Black       57
Hispanic    41
Other        8
White       79
dtype: int64
```

We can also query the data to verify the transformation was correct:

In [80]:

```
myData.query("Race in ['American Indian', 'Pacific Islander']")
```

Out[80]:

	CustID	Sex	Race	BirthDate	College	HouseholdSize	ZipCode	Income
18	1536475	Male	Pacific Islander	1958-05-17	Yes	2	90019	73000
22	1538886	Male	American Indian	1996-09-15	Yes	2	90014	94000
27	1540076	Male	American Indian	1978-01-24	Yes	5	90012	60000
40	1543734	Male	Pacific Islander	1990-03-06	Yes	5	90042	69000
116	1559734	Male	American	1968-06-	Yes	1	90004	75000

	CustID	Sex	Race	BirthDate	College	HouseholdSize	ZipCode	Income
			Indian	26				
132	1563548	Male	American Indian	1988-10-02	Yes	5	90043	167000
156	1568380	Female	Pacific Islander	1956-08-19	Yes	2	90020	62000
197	1579389	Female	American Indian	2000-05-21	Yes	1	90009	50000

The 19th row (i.e., row index 18) is the first customer in the Other category.

e. Alternatively to steps b through d above, we could have also recoded the data for the two least-frequent categories of Race programmatically, combining steps from above:

In [81]:

```
myData['NewRaceAlt'] = np.where(myData.Race.isin(myData.groupby('Race').size
().sort_values().index[0:2]), 'Other', myData.Race)
```

In the code above, we extract the first two indexes (i.e., a slice of 0:2, which is equal to the list `[0, 1]`) of the sorted frequencies of the original Race variable and use them to set up the NumPy **where** function descrbied earlier. One advantage of this approach is that it is not necessary to hard code the specific categories that need to be recoded; however, regardless of the approach ultimately used, the result is identical:

In [82]:

```
myData.query("NewRaceAlt == 'Other'")
```

Out[82]:

	CustID	Sex	Race	BirthDate	College	HouseholdSize	ZipCode	Income
18	1536475	Male	Pacific Islander	1958-05-17	Yes	2	90019	73000
22	1538886	Male	American Indian	1996-09-15	Yes	2	90014	94000
27	1540076	Male	American Indian	1978-01-24	Yes	5	90012	60000
40	1543734	Male	Pacific Islander	1990-03-06	Yes	5	90042	69000
116	1559734	Male	American Indian	1968-06-26	Yes	1	90004	75000
132	1563548	Male	American Indian	1988-10-02	Yes	5	90043	167000
156	1568380	Female	Pacific Islander	1956-08-19	Yes	2	90020	62000
197	1579389	Female	American Indian	2000-05-21	Yes	1	90009	50000

Example 2.7

For the new Asian-inspired meal kits, Catherine feels that understanding the channels through which customers were acquired is important to predict customers' future behaviors. In order to include the Channel variable in her predictive model, Catherine needs to convert the Channel categories into dummy variables. Because web banner ads are probably the most common marketing tools used by Organic Food Superstore, she plans to use the Web channel as the reference category and assess the effects of other channels in relation to the Web channel. Use Python to create the relevant dummy variables for the Channel variable.

a. Import the ***Customers*** data file into a Pandas DataFrame (table) and label it myData. Pull up the data to get a basic overview, including dimensions:

In [83]:

```
import pandas as pd

myData = pd.read_excel('jaggia_ba_1e_ch02_Data_Files.xlsx', sheet_name = 'Cus
tomers')

myData
```

Out[83]:

	CustID	Sex	Race	BirthDate	College	HouseholdSize	ZipCode	Income
0	1530016	Female	Black	1986-12-16	Yes	5	90047	53000
1	1531136	Male	White	1993-05-09	Yes	5	90026	94000
2	1532160	Male	Black	1966-05-22	Yes	2	90027	64000
3	1532307	Male	White	1964-09-16	Yes	4	90029	60000
4	1532356	Female	Hispanic	1964-07-15	No	5	90017	47000
...	...	...	...	...	...	...	...	...
195	1578525	Male	Hispanic	1963-12-12	Yes	1	90005	82000
196	1579349	Male	Asian	1980-12-19	Yes	1	90010	49000
197	1579389	Female	American Indian	2000-05-21	Yes	1	90009	50000
198	1579857	Female	White	1991-01-26	Yes	1	90055	52000
199	1579979	Male	White	1999-07-05	Yes	5	90043	102000

200 rows × 14 columns

b. To create a dummy variable for the individual categories in the Channel variable, we use the Pandas **get_dummies** function with a customized *prefix* argument to indicate the name of the variable. Note that because we are treating Web as our reference category, we also drop it via the pandas **drop** function:

In [84]:

```
channelDummies = pd.get_dummies(myData.Channel, prefix = 'Channel').drop(colu
mns = 'Channel_Web')

channelDummies.head()
```

Out[84]:

	Channel_Referral	Channel_SM	Channel_TV
0	0	1	0

1	0	0	1
2	0	0	1
3	0	1	0
4	0	0	0

To add these to our data set, we can supply a list of DataFrames to concatenate (i.e., bind together) using the Pandas **concat** function:

In [85]:

```
myDataCombined = pd.concat([myData, channelDummies], axis = 1)

myDataCombined.head()
```

Out[85]:

	CustID	Sex	Race	BirthDate	College	HouseholdSize	ZipCode	Income	Spe
0	1530016	Female	Black	1986-12-16	Yes	5	90047	53000	
1	1531136	Male	White	1993-05-09	Yes	5	90026	94000	
2	1532160	Male	Black	1966-05-22	Yes	2	90027	64000	
3	1532307	Male	White	1964-09-16	Yes	4	90029	60000	
4	1532356	Female	Hispanic	1964-07-15	No	5	90017	47000	

Note that the *axis* argument must be set to 1 to indicate a column bind (i.e., for a row bind, use axis = 0).

Example 2.8

For the new Asian-inspired meal kits, Catherine wants to pay attention to customer satisfaction. As the customer satisfaction ratings represent ordinal data, she wants to convert them to category scores ranging from 1 (Very Dissatisfied) to 5 (Very Satisfied) to make the variable more readily usable in predictive models. Use Python to create category scores for the Satisfaction variable.

a. Import the ***Customers*** data file into a Pandas DataFrame (table) and label it myData. Pull up the data to get a basic overview, including dimensions:

In [86]:

```
import pandas as pd

myData = pd.read_excel('jaggia_ba_1e_ch02_Data_Files.xlsx', sheet_name = 'Cus
tomers')

myData
```

Out[86]:

	CustID	Sex	Race	BirthDate	College	HouseholdSize	ZipCode	Income	S
0	1530016	Female	Black	1986-12-16	Yes	5	90047	53000	
1	1531136	Male	White	1993-05-09	Yes	5	90026	94000	

2	1532160	Male	Black	1966-05-22	Yes		2	90027	64000
3	1532307	Male	White	1964-09-16	Yes		4	90029	60000
4	1532356	Female	Hispanic	1964-07-15	No		5	90017	47000
...	...	...	...	...	...		...	...	...
195	1578525	Male	Hispanic	1963-12-12	Yes		1	90005	82000
196	1579349	Male	Asian	1980-12-19	Yes		1	90010	49000
197	1579389	Female	American Indian	2000-05-21	Yes		1	90009	50000
198	1579857	Female	White	1991-01-26	Yes		1	90055	52000
199	1579979	Male	White	1999-07-05	Yes		5	90043	102000

200 rows × 14 columns

b. To recode the variables in Satisfaction, we use the Pandas **replace** function with a dictionary of original values and replacements:

In [87]:

```
myData['SatisfactionScore'] = myData.Satisfaction.replace({
    'Very Dissatisfied': 1,
    'Somewhat Dissatisfied': 2,
    'Neutral': 3,
    'Somewhat Satisfied': 4,
    'Very Satisfied': 5})
```

c. To verify the result, we can pull up the first four rows of data using the Pandas **head** method:

In [88]:

```
myData.head(n = 4)
```

Out[88]:

	CustID	Sex	Race	BirthDate	College	HouseholdSize	ZipCode	Income	Spend
0	1530016	Female	Black	1986-12-16	Yes	5	90047	53000	
1	1531136	Male	White	1993-05-09	Yes	5	90026	94000	
2	1532160	Male	Black	1966-05-22	Yes	2	90027	64000	
3	1532307	Male	White	1964-09-16	Yes	4	90029	60000	

As shown in the output above the first four rows (i.e., using the argument *n* = 4) have satisfaction scores of 1, 3, 5, and 1, respectively.