Data Manipulation, Cleaning, and Transformation

Neba Nfonsang

¶

University of Denver

In [1]:
```python
# import necessary packages
import pandas as pd
import numpy as np
```

Data Manipulation

- We will start by creating some data that will be used for data manipulation. In a real world scenario, you would mostly read data from other sources into the Python environment instead of creating data. Later on, we will see how to read csv data in to Python using a pandas feature.
- In this section, you will learn how to check/change column and row labels, drop and add columns/rows, select specific sections of your data.

Create the Data

In [2]:
```python
# let's create restaurant data using a dictionary
# this data is not real, it is intended for educational purposes

restaurant_data = {"res_name": ["Javanut", "Mixers", "Grizzly",
                                "Tiki Taco", "Tarmac", "Zilla",
                                "Homestyle", "Roadhouse"],
                   "rating": ["good", "very good", "good",
                              "excellent", "very good",
                              "good", "good", "excellent"],
                   "city": ["Denver", "Aurora", "Aurora",
                            "Denver", "Lakewood", "Denver",
                            "Denver", "Lakewood"],
                   "price": [18, 22, 20, 38, 33, 28, 25, 30],
                   "wait_time": [10, 6, 15, 13, 6, 8, 8, 10]}

res_data = pd.DataFrame(restaurant_data)
res_data
```

Out[2]:

	res_name	rating	city	price	wait_time
0	Javanut	good	Denver	18	10
1	Mixers	very good	Aurora	22	6
2	Grizzly	good	Aurora	20	15
3	Tiki Taco	excellent	Denver	38	13
4	Tarmac	very good	Lakewood	33	6
5	Zilla	good	Denver	28	8
6	Homestyle	good	Denver	25	8
7	Roadhouse	excellent	Lakewood	30	10

Reorder and Change Column Names

In [3]:
```python
# change the order of the column names
# columns names must be exactly the same as specified in the data

res_data = pd.DataFrame(restaurant_data, columns=["res_name", "rating", "city",
                                                   "price", "wait_time"])
res_data
```

Out[3]:

	res_name	rating	city	price	wait_time
0	Javanut	good	Denver	18	10
1	Mixers	very good	Aurora	22	6
2	Grizzly	good	Aurora	20	15
3	Tiki Taco	excellent	Denver	38	13
4	Tarmac	very good	Lakewood	33	6
5	Zilla	good	Denver	28	8
6	Homestyle	good	Denver	25	8
7	Roadhouse	excellent	Lakewood	30	10

In [4]:
```python
# change column names (change price and rating to meal_price
# and quality_rating respectively)
# set the value of the inplace parameter to "True",
# to permanently change the column names

res_data.rename(columns={"rating":"quality_rating",
                         "price":"meal_price"}, inplace=True)
```

In [5]:
```python
# display data to see changes in column names
res_data
```

Out[5]:

	res_name	quality_rating	city	meal_price	wait_time
0	Javanut	good	Denver	18	10
1	Mixers	very good	Aurora	22	6
2	Grizzly	good	Aurora	20	15
3	Tiki Taco	excellent	Denver	38	13
4	Tarmac	very good	Lakewood	33	6
5	Zilla	good	Denver	28	8
6	Homestyle	good	Denver	25	8
7	Roadhouse	excellent	Lakewood	30	10

In [6]:
```python
# check the column names of your data
res_data.columns
```

Out[6]:
```
Index(['res_name', 'quality_rating', 'city', 'meal_price', 'wait_time'], dtype='objec
t')
```

In [7]:
```python
# creae a list of the column names
list(res_data.columns)
```

Out[7]:
```
['res_name', 'quality_rating', 'city', 'meal_price', 'wait_time']
```

Add and Remove Column Names

Let's add a "delivery" column to the data. The values ("yes" or "no") in this column indicate whether the restaurant offers delivery services or not.

In [8]:
```python
# add a delivery column
res_data["delivery"] = ["yes", "no", "no", "yes",
                        "yes", "yes", "no", "yes"]
res_data
```

Out[8]:

	res_name	quality_rating	city	meal_price	wait_time	delivery
0	Javanut	good	Denver	18	10	yes
1	Mixers	very good	Aurora	22	6	no
2	Grizzly	good	Aurora	20	15	no
3	Tiki Taco	excellent	Denver	38	13	yes
4	Tarmac	very good	Lakewood	33	6	yes
5	Zilla	good	Denver	28	8	yes
6	Homestyle	good	Denver	25	8	no
7	Roadhouse	excellent	Lakewood	30	10	yes

In [9]:
```python
# remove the delivery column
res_data.drop("delivery", axis="columns")
```

Out[9]:

	res_name	quality_rating	city	meal_price	wait_time
0	Javanut	good	Denver	18	10
1	Mixers	very good	Aurora	22	6
2	Grizzly	good	Aurora	20	15
3	Tiki Taco	excellent	Denver	38	13
4	Tarmac	very good	Lakewood	33	6
5	Zilla	good	Denver	28	8
6	Homestyle	good	Denver	25	8
7	Roadhouse	excellent	Lakewood	30	10

In [10]:
```python
# let's view the data
# you would notice that the delivery column
# was not permanently dropped

res_data
```

Out[10]:

	res_name	quality_rating	city	meal_price	wait_time	delivery
0	Javanut	good	Denver	18	10	yes
1	Mixers	very good	Aurora	22	6	no
2	Grizzly	good	Aurora	20	15	no
3	Tiki Taco	excellent	Denver	38	13	yes
4	Tarmac	very good	Lakewood	33	6	yes
5	Zilla	good	Denver	28	8	yes
6	Homestyle	good	Denver	25	8	no
7	Roadhouse	excellent	Lakewood	30	10	yes

In [11]:
```
# set the value of the inplace parameter to "True",
# to permanenlty drop the delivery column

res_data.drop("delivery", axis="columns", inplace=True)
res_data
```

Out[11]:

	res_name	quality_rating	city	meal_price	wait_time
0	Javanut	good	Denver	18	10
1	Mixers	very good	Aurora	22	6
2	Grizzly	good	Aurora	20	15
3	Tiki Taco	excellent	Denver	38	13
4	Tarmac	very good	Lakewood	33	6
5	Zilla	good	Denver	28	8
6	Homestyle	good	Denver	25	8
7	Roadhouse	excellent	Lakewood	30	10

Index Labels

In [12]:
```
# check the index labels of the data
res_data.index
```

Out[12]: `RangeIndex(start=0, stop=8, step=1)`

In [13]:
```
res_data.index.values
```

Out[13]: `array([0, 1, 2, 3, 4, 5, 6, 7], dtype=int64)`

In [14]:
```
# create a code for the restaurant names
code = ['JN','MX', 'GZ', 'TT', 'TM', 'ZL', 'HS', 'RH']

# set the code as the new index
res_data.index = code
res_data
```

Out[14]:

	res_name	quality_rating	city	meal_price	wait_time
JN	Javanut	good	Denver	18	10
MX	Mixers	very good	Aurora	22	6
GZ	Grizzly	good	Aurora	20	15
TT	Tiki Taco	excellent	Denver	38	13
TM	Tarmac	very good	Lakewood	33	6
ZL	Zilla	good	Denver	28	8
HS	Homestyle	good	Denver	25	8
RH	Roadhouse	excellent	Lakewood	30	10

View the Head and Tail of the Data

By default, .head() displays the first five rows while .tail() displays the last five rows of the data.
To display a different number of rows, pass that number as an argument into the .head() or .tail() method

In [15]:
```
# view the first five rows of the data
res_data.head()
```

Out[15]:

	res_name	quality_rating	city	meal_price	wait_time
JN	Javanut	good	Denver	18	10
MX	Mixers	very good	Aurora	22	6
GZ	Grizzly	good	Aurora	20	15
TT	Tiki Taco	excellent	Denver	38	13
TM	Tarmac	very good	Lakewood	33	6

In [16]:
```
# view the first three rows of the data
res_data.head(3)
```

Out[16]:

	res_name	quality_rating	city	meal_price	wait_time
JN	Javanut	good	Denver	18	10
MX	Mixers	very good	Aurora	22	6
GZ	Grizzly	good	Aurora	20	15

In [17]:
```
# view the last five rows of the data
res_data.tail()
```

Out[17]:

	res_name	quality_rating	city	meal_price	wait_time
TT	Tiki Taco	excellent	Denver	38	13
TM	Tarmac	very good	Lakewood	33	6
ZL	Zilla	good	Denver	28	8
HS	Homestyle	good	Denver	25	8
RH	Roadhouse	excellent	Lakewood	30	10

In [18]:
```
# view the last three rows of the data
res_data.tail(3)
```

Out[18]:

	res_name	quality_rating	city	meal_price	wait_time
ZL	Zilla	good	Denver	28	8
HS	Homestyle	good	Denver	25	8
RH	Roadhouse	excellent	Lakewood	30	10

Selecting (Indexing or Slicing) Data

The square bracket operator ([ ]) is used to select a portion or section of data in Python. The .loc() and .iloc() methods of the DataFrame object are also used for indexing and slicing

Select Columns with Square Bracket Operator ([ ]) and Column Names

In [19]:
```python
# use [] to select a single column
res_data["res_name"]
```

Out[19]:
```
JN       Javanut
MX        Mixers
GZ       Grizzly
TT     Tiki Taco
TM        Tarmac
ZL         Zilla
HS     Homestyle
RH     Roadhouse
Name: res_name, dtype: object
```

In [20]:
```python
# use [[]] to select multiple columns
# double square brackets are used because
# multiple columns need to be in a list

res_data[["res_name", "wait_time"]]
```

Out[20]:

	res_name	wait_time
JN	Javanut	10
MX	Mixers	6
GZ	Grizzly	15
TT	Tiki Taco	13
TM	Tarmac	6
ZL	Zilla	8
HS	Homestyle	8
RH	Roadhouse	10

Select a Column Using a Dot(.) Operator

use the dot operator sparingly. Sometimes an error may be generated if Python is interpreting what follows after the dot as an attribute of the object.

In [21]:
```python
res_data.res_name
```

Out[21]:
```
JN       Javanut
MX        Mixers
GZ       Grizzly
TT     Tiki Taco
TM        Tarmac
ZL         Zilla
HS     Homestyle
RH     Roadhouse
Name: res_name, dtype: object
```

Select Rows Using Square Bracket Operator ([ ]) and Row Indexes or Positions

In [22]:
```python
# select the first three rows
# remember that Python starts counting from zero (0)
res_data[0:3]
```

Out[22]:

	res_name	quality_rating	city	meal_price	wait_time
JN	Javanut	good	Denver	18	10
MX	Mixers	very good	Aurora	22	6
GZ	Grizzly	good	Aurora	20	15

In [23]:
```
# select from the second row through the third row
res_data[1:3]
```

Out[23]:

	res_name	quality_rating	city	meal_price	wait_time
MX	Mixers	very good	Aurora	22	6
GZ	Grizzly	good	Aurora	20	15

Select a Row (or Rows) Using the .loc() Method and the Index Label

In [24]:
```
# select the row with index lable (or row name) "JN"
res_data.loc["JN"]
```

Out[24]:
```
res_name          Javanut
quality_rating       good
city               Denver
meal_price             18
wait_time              10
Name: JN, dtype: object
```

In [25]:
```
# select rows starting from row with
# label "JN" to row with label "TM"
res_data.loc["JN":"TM"]
```

Out[25]:

	res_name	quality_rating	city	meal_price	wait_time
JN	Javanut	good	Denver	18	10
MX	Mixers	very good	Aurora	22	6
GZ	Grizzly	good	Aurora	20	15
TT	Tiki Taco	excellent	Denver	38	13
TM	Tarmac	very good	Lakewood	33	6

In [26]:
```
# select the rows with labels "JM" and "TM"
# double square brackets are used because multiple row names should be in a list
# you could include as many row names as you want

res_data.loc[["JN", "TM"]]
```

Out[26]:

	res_name	quality_rating	city	meal_price	wait_time
JN	Javanut	good	Denver	18	10
TM	Tarmac	very good	Lakewood	33	6

Select both Rows and Columns Using the .loc() Method

Note that when we use the .loc() method, the name(s) of the row(s) or/and column(s) are passed into the square bracket operator

In [27]:
```python
# select row with label "JN" to row with label "TT"
# and at the same time, select column with name
# "res_name" to column with name "city"

res_data.loc["JN":"TT", "res_name":"city"]
```

Out[27]:

	res_name	quality_rating	city
JN	Javanut	good	Denver
MX	Mixers	very good	Aurora
GZ	Grizzly	good	Aurora
TT	Tiki Taco	excellent	Denver

In [28]:
```python
# select all rows and at the same time
# select columns from "res_name" to "city"
res_data.loc[ : , "res_name":"city"]
```

Out[28]:

	res_name	quality_rating	city
JN	Javanut	good	Denver
MX	Mixers	very good	Aurora
GZ	Grizzly	good	Aurora
TT	Tiki Taco	excellent	Denver
TM	Tarmac	very good	Lakewood
ZL	Zilla	good	Denver
HS	Homestyle	good	Denver
RH	Roadhouse	excellent	Lakewood

In [29]:
```python
# select rows "GZ" to "HS" and all columns
res_data.loc["GZ":"HS", :]
```

Out[29]:

	res_name	quality_rating	city	meal_price	wait_time
GZ	Grizzly	good	Aurora	20	15
TT	Tiki Taco	excellent	Denver	38	13
TM	Tarmac	very good	Lakewood	33	6
ZL	Zilla	good	Denver	28	8
HS	Homestyle	good	Denver	25	8

Select both Rows and Columns Using the .iloc[ ] Method

The .iloc[ ] method performs the same selection tasks as the .loc[ ] method, however, the .iloc[ ] method uses the position(s) of the row(s) or/and columns(s) to select data while the .loc[ ] method uses row or/and column names

In [30]:
```python
# to select the first row
res_data.iloc[0]
```

Out[30]:
```
res_name          Javanut
quality_rating       good
city               Denver
meal_price             18
wait_time              10
Name: JN, dtype: object
```

In [31]:
```
# select the first five rows
res_data.iloc[0:5]
```

Out[31]:

	res_name	quality_rating	city	meal_price	wait_time
JN	Javanut	good	Denver	18	10
MX	Mixers	very good	Aurora	22	6
GZ	Grizzly	good	Aurora	20	15
TT	Tiki Taco	excellent	Denver	38	13
TM	Tarmac	very good	Lakewood	33	6

In [32]:
```
# select the second and the fourth rows
res_data.iloc[[0, 3]]
```

Out[32]:

	res_name	quality_rating	city	meal_price	wait_time
JN	Javanut	good	Denver	18	10
TT	Tiki Taco	excellent	Denver	38	13

In [33]:
```
# select first to fourth row including
# only the first three columns
res_data.iloc[0:4, 0:3]
```

Out[33]:

	res_name	quality_rating	city
JN	Javanut	good	Denver
MX	Mixers	very good	Aurora
GZ	Grizzly	good	Aurora
TT	Tiki Taco	excellent	Denver

In [34]:
```
# select the value at the intersection
# of the first row, second column
res_data.iloc[0][1]
```

Out[34]: 'good'

In [35]:
```
# alternatively, select the value
# at the intersection of the first row, second column
res_data.iloc[0, 1]
```

Out[35]: 'good'

Boolean Selection

In [36]:
```
# select the entire dataset only for ratings that are "good"
res_data[res_data.quality_rating=="good"]
```

Out[36]:

	res_name	quality_rating	city	meal_price	wait_time
JN	Javanut	good	Denver	18	10
GZ	Grizzly	good	Aurora	20	15
ZL	Zilla	good	Denver	28	8
HS	Homestyle	good	Denver	25	8

In [37]:
```python
# select the entire dataset for ratings
# all other ratings except "good"
res_data[~(res_data.quality_rating=="good")]
```

Out[37]:

	res_name	quality_rating	city	meal_price	wait_time
MX	Mixers	very good	Aurora	22	6
TT	Tiki Taco	excellent	Denver	38	13
TM	Tarmac	very good	Lakewood	33	6
RH	Roadhouse	excellent	Lakewood	30	10

In [38]:
```python
# select the dataset where the city is Denver or Aurora
# this type of selection is useful when you want to select
# specific groups for analysis

res_data[(res_data.city=="Denver")|(res_data.city=="Aurora")]
```

Out[38]:

	res_name	quality_rating	city	meal_price	wait_time
JN	Javanut	good	Denver	18	10
MX	Mixers	very good	Aurora	22	6
GZ	Grizzly	good	Aurora	20	15
TT	Tiki Taco	excellent	Denver	38	13
ZL	Zilla	good	Denver	28	8
HS	Homestyle	good	Denver	25	8

In [39]:
```python
# select meal prices for Denver and Aurora
res_data.meal_price[(res_data.city=="Denver")|
                    (res_data.city=="Aurora")]
```

Out[39]:
```
JN    18
MX    22
GZ    20
TT    38
ZL    28
HS    25
Name: meal_price, dtype: int64
```

In [40]:
```python
# alternative way of selecting meal prices for Denver and Aurora
# you can first do boolean selection for the entire dataset,
# then select the meal price

res_data[(res_data.city=="Denver")|
         (res_data.city=="Aurora")].meal_price
```

Out[40]:
```
JN    18
MX    22
GZ    20
TT    38
ZL    28
HS    25
Name: meal_price, dtype: int64
```

Data Cleaning

- In this section, we will learn how to handle missing data with pandas, delete columns and rows, remove duplicates and transform data
- Significant amount of time in data analysis in used in data cleaning and preparation because the format in which data is collected is not necessarily the format suitable for analysis
- pandas simplies the process of dealing with missing data. Missing data in all descriptive statistics is excluded by default
- **NaN ("Not a Number")** is used to indicate missing data in pandas which is equivalent to Python's **None** type.

Handling Missing Data

Before reading data from a file such as csv into Python, make sure to understand how missing data is represented in the file. For example, SPSS usually represent discrete missing data as 999. Some specific code might be used to represent missing data. Sometimes, missing data is represented as NA or NONE. Again, in pandas, missing values are represented as NaN. Any missing data that is not automatically converted to NaN when the data is read with pandas will need to be replaced with NaN.

Create Data with Missing Values

In real life, we will not need to create the data, you will already have data that was collected with missing values.

In [41]:
```
# let's create data with missing values
# note that the price column contains missing data represented as 999.
# there is also missing value for the city column indicated as "None"

missing = {"res_name": ["Javanut", "Mixers", "Grizzly",
                        "Tiki Taco", "Tarmac", "Zilla",
                        "Homestyle", "Roadhouse"],
                   "rating":["good", "very good", "good",
                             "excellent", "very good", "good",
                             "good", "excellent"],
                   "city":["Denver", "Aurora", "Aurora",
                           "Denver", "Lakewood", "Denver",
                           "Denver", "None"],
                   "meal_price": [18, 22, 999, 38, 999, 28, 999, 999],
                   "wait_time": [10, 999, 15, 13, 6, 8, 999, 10]}

missing = pd.DataFrame(missing, columns=["res_name", "rating",
                                         "city", "meal_price",
                                         "wait_time"])
missing
```

Out[41]:

	res_name	rating	city	meal_price	wait_time
0	Javanut	good	Denver	18	10
1	Mixers	very good	Aurora	22	999
2	Grizzly	good	Aurora	999	15
3	Tiki Taco	excellent	Denver	38	13
4	Tarmac	very good	Lakewood	999	6
5	Zilla	good	Denver	28	8
6	Homestyle	good	Denver	999	999
7	Roadhouse	excellent	None	999	10

Replace Missing Values with NaN

In [42]:
```
# replace all 999 with np.nan
missing.replace(999, np.nan, inplace=True)
```

In [43]:
```python
# view how missing data is replace with "NaN"
missing
```

Out[43]:

	res_name	rating	city	meal_price	wait_time
0	Javanut	good	Denver	18.0	10.0
1	Mixers	very good	Aurora	22.0	NaN
2	Grizzly	good	Aurora	NaN	15.0
3	Tiki Taco	excellent	Denver	38.0	13.0
4	Tarmac	very good	Lakewood	NaN	6.0
5	Zilla	good	Denver	28.0	8.0
6	Homestyle	good	Denver	NaN	NaN
7	Roadhouse	excellent	None	NaN	10.0

In [44]:
```python
# also replace the None in the city column with np.nan
missing["city"].replace("None", np.nan, inplace=True)
```

In [45]:
```python
# now, we have the data with missing
# values ready for cleaning
missing
```

Out[45]:

	res_name	rating	city	meal_price	wait_time
0	Javanut	good	Denver	18.0	10.0
1	Mixers	very good	Aurora	22.0	NaN
2	Grizzly	good	Aurora	NaN	15.0
3	Tiki Taco	excellent	Denver	38.0	13.0
4	Tarmac	very good	Lakewood	NaN	6.0
5	Zilla	good	Denver	28.0	8.0
6	Homestyle	good	Denver	NaN	NaN
7	Roadhouse	excellent	NaN	NaN	10.0

Check and Count Missing Values

In [46]:
```python
# check the missing values in the data
# using the .isnull() method
# "True" indicates missing data

missing.isnull()
```

Out[46]:

	res_name	rating	city	meal_price	wait_time
0	False	False	False	False	False
1	False	False	False	False	True
2	False	False	False	True	False
3	False	False	False	False	False
4	False	False	False	True	False
5	False	False	False	False	False
6	False	False	False	True	True
7	False	False	True	True	False

In [47]:
```python
# check the missing values in the
# data using the .notnull() method
# "False" indicates missing data

missing.notnull()
```

Out[47]:

	res_name	rating	city	meal_price	wait_time
0	True	True	True	True	True
1	True	True	True	True	False
2	True	True	True	False	True
3	True	True	True	True	True
4	True	True	True	False	True
5	True	True	True	True	True
6	True	True	True	False	False
7	True	True	False	False	True

Note: if you have a large dataset, using .isnull() or .notnull() methods will not be helpful in understanding how much data is missing. It would be better to count the number of missing data for each variable. Sometimes, variables with a high percentage of missing data should be excluded in the data analysis.

In [48]:
```python
# compute how much data is missing for each column
missing.isnull().sum()
```

Out[48]:
```
res_name      0
rating        0
city          1
meal_price    4
wait_time     2
dtype: int64
```

Drop Rows or Columns with Missing Data

In [49]:
```python
# drop all rows with any missing data
# to drop permanently, set the value
# of the inplace parameter to "True"

missing.dropna(axis="rows")
```

Out[49]:

	res_name	rating	city	meal_price	wait_time
0	Javanut	good	Denver	18.0	10.0
3	Tiki Taco	excellent	Denver	38.0	13.0
5	Zilla	good	Denver	28.0	8.0

In [50]:
```
# drop all columns with any missing data
# set axis="columns" or axis=1

missing.dropna(axis="columns")
```

Out[50]:

	res_name	rating
0	Javanut	good
1	Mixers	very good
2	Grizzly	good
3	Tiki Taco	excellent
4	Tarmac	very good
5	Zilla	good
6	Homestyle	good
7	Roadhouse	excellent

In [51]:
```
# keep columns if number of non-missing values is equal to
# or greater than 5 (thresh=5)
# note that the meal_price is dropped because
# it's non-missing values are less than 5

missing.dropna(axis="columns", thresh=5)
```

Out[51]:

	res_name	rating	city	wait_time
0	Javanut	good	Denver	10.0
1	Mixers	very good	Aurora	NaN
2	Grizzly	good	Aurora	15.0
3	Tiki Taco	excellent	Denver	13.0
4	Tarmac	very good	Lakewood	6.0
5	Zilla	good	Denver	8.0
6	Homestyle	good	Denver	NaN
7	Roadhouse	excellent	NaN	10.0

Filling in Missing Data

In [52]:
```
# fill in missing data with a scalar
# fill in missing data for the meal_price column
# with a scalar value of 9 (use a dictionary to specify the column)

missing.fillna({"meal_price": 9})
```

Out[52]:

	res_name	rating	city	meal_price	wait_time
0	Javanut	good	Denver	18.0	10.0
1	Mixers	very good	Aurora	22.0	NaN
2	Grizzly	good	Aurora	9.0	15.0
3	Tiki Taco	excellent	Denver	38.0	13.0
4	Tarmac	very good	Lakewood	9.0	6.0
5	Zilla	good	Denver	28.0	8.0
6	Homestyle	good	Denver	9.0	NaN
7	Roadhouse	excellent	NaN	9.0	10.0

In [53]:
```python
# use a dictionary to fill in missing values for various columns
# there should be a good rationale for filling in
# missing values with specific scalar values

missing.fillna({"city": "Denver", "meal_price":9, "wait_time":10})
```

Out[53]:

	res_name	rating	city	meal_price	wait_time
0	Javanut	good	Denver	18.0	10.0
1	Mixers	very good	Aurora	22.0	10.0
2	Grizzly	good	Aurora	9.0	15.0
3	Tiki Taco	excellent	Denver	38.0	13.0
4	Tarmac	very good	Lakewood	9.0	6.0
5	Zilla	good	Denver	28.0	8.0
6	Homestyle	good	Denver	9.0	10.0
7	Roadhouse	excellent	Denver	9.0	10.0

In [54]:
```python
# fill in missing values with mean
# similarly, median values could also be used

missing.fillna({"meal_price":missing.meal_price.mean(),
                "wait_time":missing.wait_time.mean()})
```

Out[54]:

	res_name	rating	city	meal_price	wait_time
0	Javanut	good	Denver	18.0	10.000000
1	Mixers	very good	Aurora	22.0	10.333333
2	Grizzly	good	Aurora	26.5	15.000000
3	Tiki Taco	excellent	Denver	38.0	13.000000
4	Tarmac	very good	Lakewood	26.5	6.000000
5	Zilla	good	Denver	28.0	8.000000
6	Homestyle	good	Denver	26.5	10.333333
7	Roadhouse	excellent	NaN	26.5	10.000000

In [55]:
```python
# use ffill or forward fill to propagage the last valid
# observation to fill the next gap

missing.fillna(method="ffill")
```

Out[55]:

	res_name	rating	city	meal_price	wait_time
0	Javanut	good	Denver	18.0	10.0
1	Mixers	very good	Aurora	22.0	10.0
2	Grizzly	good	Aurora	22.0	15.0
3	Tiki Taco	excellent	Denver	38.0	13.0
4	Tarmac	very good	Lakewood	38.0	6.0
5	Zilla	good	Denver	28.0	8.0
6	Homestyle	good	Denver	28.0	8.0
7	Roadhouse	excellent	Denver	28.0	10.0

In [56]:
```python
# backfill works in an opposite way compared to forward fill
# if missing values are the last values, they will not be filled
# since there is no value below that last value to propagate backward

missing.fillna(method="bfill")
```

Out[56]:

	res_name	rating	city	meal_price	wait_time
0	Javanut	good	Denver	18.0	10.0
1	Mixers	very good	Aurora	22.0	15.0
2	Grizzly	good	Aurora	38.0	15.0
3	Tiki Taco	excellent	Denver	38.0	13.0
4	Tarmac	very good	Lakewood	28.0	6.0
5	Zilla	good	Denver	28.0	8.0
6	Homestyle	good	Denver	NaN	10.0
7	Roadhouse	excellent	NaN	NaN	10.0

Remove Duplicates

We will first create a duplicate, though we will not need to create a duplicate in real life as real data either have a duplicate or not

In [57]:
```python
# create a duplicate for the last row
dup = missing.iloc[-1]
dup
```

Out[57]:
```
res_name       Roadhouse
rating         excellent
city                 NaN
meal_price           NaN
wait_time             10
Name: 7, dtype: object
```

In [58]:
```python
# append the duplicate to the data
missing = missing.append(dup, ignore_index=True)
missing
```

Out[58]:

	res_name	rating	city	meal_price	wait_time
0	Javanut	good	Denver	18.0	10.0
1	Mixers	very good	Aurora	22.0	NaN
2	Grizzly	good	Aurora	NaN	15.0
3	Tiki Taco	excellent	Denver	38.0	13.0
4	Tarmac	very good	Lakewood	NaN	6.0
5	Zilla	good	Denver	28.0	8.0
6	Homestyle	good	Denver	NaN	NaN
7	Roadhouse	excellent	NaN	NaN	10.0
8	Roadhouse	excellent	NaN	NaN	10.0

In [59]:
```
# check if the data has a duplicate
# .duplicated() returns "True" for a duplicate row

missing.duplicated()
```

Out[59]:
```
0    False
1    False
2    False
3    False
4    False
5    False
6    False
7    False
8     True
dtype: bool
```

In [60]:
```
# drop the duplicate row or rows
missing.drop_duplicates(inplace=True)
```

In [61]:
```
# view data again, you would notice the duplicate
# row is no more there

missing
```

Out[61]:

	res_name	rating	city	meal_price	wait_time
0	Javanut	good	Denver	18.0	10.0
1	Mixers	very good	Aurora	22.0	NaN
2	Grizzly	good	Aurora	NaN	15.0
3	Tiki Taco	excellent	Denver	38.0	13.0
4	Tarmac	very good	Lakewood	NaN	6.0
5	Zilla	good	Denver	28.0	8.0
6	Homestyle	good	Denver	NaN	NaN
7	Roadhouse	excellent	NaN	NaN	10.0

Data Transformation

Data transfromation is a part of preprocessing or data preparation. We will take a look at how to transform categorical data into numerical codes, how to transform numerical data into categorical data as well as how to create dummy data

Transform categorical data into numerical code

In [62]:
```
# let's use the restaurant data again
res_data
```

Out[62]:

	res_name	quality_rating	city	meal_price	wait_time
JN	Javanut	good	Denver	18	10
MX	Mixers	very good	Aurora	22	6
GZ	Grizzly	good	Aurora	20	15
TT	Tiki Taco	excellent	Denver	38	13
TM	Tarmac	very good	Lakewood	33	6
ZL	Zilla	good	Denver	28	8
HS	Homestyle	good	Denver	25	8
RH	Roadhouse	excellent	Lakewood	30	10

In [63]:
```python
# let's transform the quality_rating values into numerical codes
# such that good = 1, very good = 2 and excellent = 3

# we can acheive this using the .replace() method
# or by using .map() method
# to use the .map() method, we can create a new column

d = {"good":1, "very good":2, "excellent":3}
res_data["rating_code"] = res_data["quality_rating"].map(d)
res_data
```

Out[63]:

	res_name	quality_rating	city	meal_price	wait_time	rating_code
JN	Javanut	good	Denver	18	10	1
MX	Mixers	very good	Aurora	22	6	2
GZ	Grizzly	good	Aurora	20	15	1
TT	Tiki Taco	excellent	Denver	38	13	3
TM	Tarmac	very good	Lakewood	33	6	2
ZL	Zilla	good	Denver	28	8	1
HS	Homestyle	good	Denver	25	8	1
RH	Roadhouse	excellent	Lakewood	30	10	3

Using a Function to Transform Data

In [64]:
```python
# use a function to tranform values

# let's transform the restaurant names to upper case
upper = lambda value: value.upper()

# the function can then be used as input in the
# .apply() or .map() method.

res_data["res_name"]= res_data["res_name"].apply(upper)
res_data
```

Out[64]:

	res_name	quality_rating	city	meal_price	wait_time	rating_code
JN	JAVANUT	good	Denver	18	10	1
MX	MIXERS	very good	Aurora	22	6	2
GZ	GRIZZLY	good	Aurora	20	15	1
TT	TIKI TACO	excellent	Denver	38	13	3
TM	TARMAC	very good	Lakewood	33	6	2
ZL	ZILLA	good	Denver	28	8	1
HS	HOMESTYLE	good	Denver	25	8	1
RH	ROADHOUSE	excellent	Lakewood	30	10	3

In [65]:
```python
# create a function that increases meal_price by 0.5%

def increase_price(price):
    return price*1.05

res_data["meal_price"]= res_data["meal_price"].apply(increase_price)
res_data
```

Out[65]:

	res_name	quality_rating	city	meal_price	wait_time	rating_code
JN	JAVANUT	good	Denver	18.90	10	1
MX	MIXERS	very good	Aurora	23.10	6	2
GZ	GRIZZLY	good	Aurora	21.00	15	1
TT	TIKI TACO	excellent	Denver	39.90	13	3
TM	TARMAC	very good	Lakewood	34.65	6	2
ZL	ZILLA	good	Denver	29.40	8	1
HS	HOMESTYLE	good	Denver	26.25	8	1
RH	ROADHOUSE	excellent	Lakewood	31.50	10	3

Descretization and Binning

Discretization and binning is a way of transforming or grouping continuous data into categorical data for analysis. The pd.cut() and pd.qcut() functions can be used to transform continuous variable into categorical variables.

- pd.cut() groups the data into bins of equal length. That means, the bins are equally spaced.
- pd.qcut() groups data into into bins such that each bin contains approximately the same number of data points. The bins don't have to be of equal length.

In [66]:
```python
# let's group the price data into three bins
# let's say the three bins represent low, moderate and high prices

cat_price = pd.cut(res_data.meal_price, bins=3)
cat_price
```

Out[66]:
```
JN    (18.879, 25.9]
MX    (18.879, 25.9]
GZ    (18.879, 25.9]
TT      (32.9, 39.9]
TM      (32.9, 39.9]
ZL      (25.9, 32.9]
HS      (25.9, 32.9]
RH      (25.9, 32.9]
Name: meal_price, dtype: category
Categories (3, interval[float64]): [(18.879, 25.9] < (25.9, 32.9] < (32.9, 39.9]]
```

In [67]:
```python
# label the bins or price ranges

cat_price = pd.cut(res_data.meal_price, bins=3,
                   labels=["low", "moderate", "high"])
cat_price
```

Out[67]:
```
JN         low
MX         low
GZ         low
TT        high
TM        high
ZL    moderate
HS    moderate
RH    moderate
Name: meal_price, dtype: category
Categories (3, object): [low < moderate < high]
```

In [68]:
```python
# we could add this categorical data to the restaurant data
res_data["cat_price"] = cat_price
res_data
```

Out[68]:

	res_name	quality_rating	city	meal_price	wait_time	rating_code	cat_price
JN	JAVANUT	good	Denver	18.90	10	1	low
MX	MIXERS	very good	Aurora	23.10	6	2	low
GZ	GRIZZLY	good	Aurora	21.00	15	1	low
TT	TIKI TACO	excellent	Denver	39.90	13	3	high
TM	TARMAC	very good	Lakewood	34.65	6	2	high
ZL	ZILLA	good	Denver	29.40	8	1	moderate
HS	HOMESTYLE	good	Denver	26.25	8	1	moderate
RH	ROADHOUSE	excellent	Lakewood	31.50	10	3	moderate

In [69]:
```python
# let's group the wait_time data into 2 quantiles
# using pd.qcut

cat_time1 = pd.qcut(res_data.wait_time, q=2)
cat_time1
```

Out[69]:
```
JN     (9.0, 15.0]
MX    (5.999, 9.0]
GZ     (9.0, 15.0]
TT     (9.0, 15.0]
TM    (5.999, 9.0]
ZL    (5.999, 9.0]
HS    (5.999, 9.0]
RH     (9.0, 15.0]
Name: wait_time, dtype: category
Categories (2, interval[float64]): [(5.999, 9.0] < (9.0, 15.0]]
```

In [70]:
```python
# compare .qcut() results with .cut()
cat_time2 = pd.cut(res_data.wait_time, bins=2)
cat_time2
```

Out[70]:
```
JN    (5.991, 10.5]
MX    (5.991, 10.5]
GZ     (10.5, 15.0]
TT     (10.5, 15.0]
TM    (5.991, 10.5]
ZL    (5.991, 10.5]
HS    (5.991, 10.5]
RH    (5.991, 10.5]
Name: wait_time, dtype: category
Categories (2, interval[float64]): [(5.991, 10.5] < (10.5, 15.0]]
```

You would notice that, for .qcut(), the data is grouped into quantiles of equal sizes. That is, each quantile has equal number of data points (4 data points in each quantile or range). The quantiles have different lengths. For .cut(), the lengths (intervals) of the bins are approximately the same but the number of data points in each bin are not the same: six data points are in the first bin and two data points in the second bin.

In [71]:
```python
# let's count the data points in the quantiles
# use the .value_counts() method

cat_time1.value_counts()
```

Out[71]:
```
(9.0, 15.0]     4
(5.999, 9.0]    4
Name: wait_time, dtype: int64
```

In [72]:
```
# let's count the data points in the bins
cat_time2.value_counts()
```

Out[72]:
```
(5.991, 10.5]    6
(10.5, 15.0]     2
Name: wait_time, dtype: int64
```

Create Dummies (or Dummy Codes)

In [73]:
```
# create dummy values for the city variable
dummies = pd.get_dummies(res_data.city)
dummies
```

Out[73]:

	Aurora	Denver	Lakewood
JN	0	1	0
MX	1	0	0
GZ	1	0	0
TT	0	1	0
TM	0	0	1
ZL	0	1	0
HS	0	1	0
RH	0	0	1

In [74]:
```
# create dummy values for the city variable
# we can add a city prefix just to indicate that the dummy variables
# were created from the values of the city variable

city_dummies = pd.get_dummies(res_data.city, prefix="city" )
city_dummies
```

Out[74]:

	city_Aurora	city_Denver	city_Lakewood
JN	0	1	0
MX	1	0	0
GZ	1	0	0
TT	0	1	0
TM	0	0	1
ZL	0	1	0
HS	0	1	0
RH	0	0	1

In [75]:
```
# add the dummies values to the restaurant data
res_data.join(city_dummies)
```

Out[75]:

	res_name	quality_rating	city	meal_price	wait_time	rating_code	cat_price	city_Aurora	city_Denver	city_Lal
JN	JAVANUT	good	Denver	18.90	10	1	low	0	1	
MX	MIXERS	very good	Aurora	23.10	6	2	low	1	0	
GZ	GRIZZLY	good	Aurora	21.00	15	1	low	1	0	
TT	TIKI TACO	excellent	Denver	39.90	13	3	high	0	1	
TM	TARMAC	very good	Lakewood	34.65	6	2	high	0	0	
ZL	ZILLA	good	Denver	29.40	8	1	moderate	0	1	
HS	HOMESTYLE	good	Denver	26.25	8	1	moderate	0	1	
RH	ROADHOUSE	excellent	Lakewood	31.50	10	3	moderate	0	0	

Note: Using the .query() Method on Data

Note: Using the .query() Method on Data

In [76]:

```python
res_data
```

Out[76]:

	res_name	quality_rating	city	meal_price	wait_time	rating_code	cat_price
JN	JAVANUT	good	Denver	18.90	10	1	low
MX	MIXERS	very good	Aurora	23.10	6	2	low
GZ	GRIZZLY	good	Aurora	21.00	15	1	low
TT	TIKI TACO	excellent	Denver	39.90	13	3	high
TM	TARMAC	very good	Lakewood	34.65	6	2	high
ZL	ZILLA	good	Denver	29.40	8	1	moderate
HS	HOMESTYLE	good	Denver	26.25	8	1	moderate
RH	ROADHOUSE	excellent	Lakewood	31.50	10	3	moderate

In [77]:

```python
# let's retrieve the data where the wait time is below 10 minutes
res_data.query("wait_time < 10")
```

Out[77]:

	res_name	quality_rating	city	meal_price	wait_time	rating_code	cat_price
MX	MIXERS	very good	Aurora	23.10	6	2	low
TM	TARMAC	very good	Lakewood	34.65	6	2	high
ZL	ZILLA	good	Denver	29.40	8	1	moderate
HS	HOMESTYLE	good	Denver	26.25	8	1	moderate

In [78]:

```python
res_data[["city", "wait_time"]].query("wait_time < 10")
```

Out[78]:

	city	wait_time
MX	Aurora	6
TM	Lakewood	6
ZL	Denver	8
HS	Denver	8

In [79]:

```python
# to select city only, where wait_time < 10
# first return entire data frame on the condition,
# then select the city

res_data.query("wait_time < 10")["city"]
```

Out[79]:

```
MX      Aurora
TM    Lakewood
ZL      Denver
HS      Denver
Name: city, dtype: object
```

In [ ]:

In [ ]: