7.2 Data Transformation

Removing Duplicates

In [16]:
```
import numpy as np
import pandas as pd
```

In [17]:
```
data = pd.DataFrame({'k1': ['one', 'two'] * 3 + ['two'],
....: 'k2': [1, 1, 2, 3, 3, 4, 4]})
data
```

Out[17]:

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4
6	two	4

DataFrame method - duplicated()

- returns a boolean Series indicating whether each row is a duplicate (has been observed in a previous row) or not

In [18]:
```
data.duplicated()
```

Out[18]:
```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

drop_duplicates()

- returns a DataFrame where the duplicated array is False:

In [19]:
```
data.drop_duplicates()
```

Out[19]:

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4

- Both of these methods by default consider all of the columns
- alternatively, specify any subset of them to detect duplicates.

```
Example:
There is an additional column of values and wanted to filter duplicates
only based on the 'k1' column
```

In [20]:
```
data
```

Out[20]:

	k1	k2
0	one	1
1	two	1
2	one	2
3	two	3
4	one	3
5	two	4
6	two	4

In [21]:
```
data['v1'] = range(7)
data
```

Out[21]:

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
5	two	4	5
6	two	4	6

In [22]:
```
data.drop_duplicates(['k1'])
```

Out[22]:

	k1	k2	v1
0	one	1	0
1	two	1	1

```
duplicated() and drop_duplicates() -  by default keep the first observed
value combination.

Passing keep='last' will return the last one:
```

In [23]:
```
data.drop_duplicates(['k1'], keep='last')
```

Out[23]:

	k1	k2	v1
4	one	3	4
6	two	4	6

In [24]:
```
data
```

Out[24]:

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
5	two	4	5
6	two	4	6

In [28]:
```
data.drop_duplicates(['k1', 'k2'], keep='last')
```

Out[28]:

	k1	k2	v1
0	one	1	0
1	two	1	1
2	one	2	2
3	two	3	3
4	one	3	4
6	two	4	6

In [ ]:
```

```

Transforming Data Using a Function or Mapping

For many datasets, you may wish to perform some transformation based on the values in an array, Series, or column in a DataFrame.

Example: hypothetical data collected about various kinds of meat

In [29]:
```
data = pd.DataFrame({'food': ['bacon', 'pulled pork', 'bacon',
....: 'Pastrami', 'corned beef', 'Bacon',
....: 'pastrami', 'honey ham', 'nova lox'],
....: 'ounces': [4, 3, 12, 6, 7.5, 8, 3, 5, 6]})
data
```

Out[29]:

	food	ounces
0	bacon	4.0
1	pulled pork	3.0
2	bacon	12.0
3	Pastrami	6.0
4	corned beef	7.5
5	Bacon	8.0
6	pastrami	3.0
7	honey ham	5.0
8	nova lox	6.0

In [30]:
```
meat_to_animal = {
'bacon': 'pig',
'pulled pork': 'pig',
'pastrami': 'cow',
'corned beef': 'cow',
'honey ham': 'pig',
'nova lox': 'salmon'
}
```

In [31]:
```
data['food']
```

Out[31]:
```
0          bacon
1    pulled pork
2          bacon
3       Pastrami
4    corned beef
5          Bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object
```

In [32]:
```
lowercased = data['food'].str.lower()
lowercased
```

Out[32]:
```
0          bacon
1    pulled pork
2          bacon
3       pastrami
4    corned beef
5          bacon
6       pastrami
7      honey ham
8       nova lox
Name: food, dtype: object
```

In [33]:
```
data['animal'] = lowercased.map(meat_to_animal)
data
```

Out[33]:

	food	ounces	animal
0	bacon	4.0	pig
1	pulled pork	3.0	pig
2	bacon	12.0	pig
3	Pastrami	6.0	cow
4	corned beef	7.5	cow
5	Bacon	8.0	pig
6	pastrami	3.0	cow
7	honey ham	5.0	pig
8	nova lox	6.0	salmon

In [34]:
```
data['food'].map(lambda x: meat_to_animal[x.lower()])
```

Out[34]:
```
0       pig
1       pig
2       pig
3       cow
4       cow
5       pig
6       cow
7       pig
8    salmon
Name: food, dtype: object
```

In [ ]:

Replacing Values

In [ ]:

Renaming Axes Indexes

In [35]:
```
data = pd.DataFrame(np.arange(12).reshape((3, 4)),
....: index=['Ohio', 'Colorado', 'New York'],
....: columns=['one', 'two', 'three', 'four'])
data
```

Out[35]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
New York	8	9	10	11

In [36]:
```
transform = lambda x: x[:4].upper()
```

In [37]:
```
data.index.map(transform)
data
```

Out[37]:

	one	two	three	four
Ohio	0	1	2	3
Colorado	4	5	6	7
New York	8	9	10	11

In [38]:
```
data.index = data.index.map(transform)
data
```

Out[38]:

	one	two	three	four
OHIO	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

rename() - Creates a transformed version of a dataset without modifying the original

- [https://www.geeksforgeeks.org/python-pandas-dataframe-rename/ (https://www.geeksforgeeks.org/python-pandas-dataframe-rename/)](https://www.geeksforgeeks.org/python-pandas-dataframe-rename/)

In [39]:
```
data.rename(index=str.title, columns=str.upper)
```

Out[39]:

	ONE	TWO	THREE	FOUR
Ohio	0	1	2	3
Colo	4	5	6	7
New	8	9	10	11

In [40]:
```
data
```

Out[40]:

	one	two	three	four
OHIO	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

In [ ]:
```
```

In [41]:
```
data.rename(index={'Ohio': 'INDIANA'},
....: columns={'three': 'peekaboo'})
```

Out[41]:

	one	two	peekaboo	four
OHIO	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

In [42]:
```
data
```

Out[42]:

	one	two	three	four
OHIO	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

In [43]:
```
data.rename(index={'Ohio': 'INDIANA'}, inplace=True)
data
```

Out[43]:

	one	two	three	four
OHIO	0	1	2	3
COLO	4	5	6	7
NEW	8	9	10	11

Discretization and Binning

Continuous data is often discretized or otherwise separated into “bins” for analysis.

[https://www.geeksforgeeks.org/pandas-cut-method-in-python/ (https://www.geeksforgeeks.org/pandas-cut-method-in-python/)](https://www.geeksforgeeks.org/pandas-cut-method-in-python/)

[https://towardsdatascience.com/all-pandas-cut-you-should-know-for-transforming-numerical-data-into-categorical-data-1370cf7f4c4f (https://towardsdatascience.com/all-pandas-cut-you-should-know-for-transforming-numerical-data-into-categorical-data-1370cf7f4c4f)](https://towardsdatascience.com/all-pandas-cut-you-should-know-for-transforming-numerical-data-into-categorical-data-1370cf7f4c4f)

cut() function

- Pandas cut() function is used to separate the array elements into different bins
- to segment and sort the data values into bins, i.e. to segregate an array of elements into separate bins
- It is used to convert a continuous variable to a categorical variable.
- cut() function is mainly used to perform statistical analysis on scalar data.

Syntax: cut(x, bins, right=True, labels=None, retbins=False, precision=3, include_lowest=False, duplicates="raise",)

Parameters:

x: The input array to be binned. Must be 1-dimensional.

bins: defines the bin edges for the segmentation.

right : (bool, default True) Indicates whether bins includes the rightmost edge or not. If right == True (the default), then the bins [1, 2, 3, 4] indicate (1,2], (2,3], (3,4].

labels : (array or bool, optional) Specifies the labels for the returned bins. Must be the same length as the resulting bins. If False, returns only integer indicators of the bins.

retbins : (bool, default False) Whether to return the bins or not. Useful when bins is provided as a scalar.

Returns:

out: Categorical, Series, or ndarray An array-like object representing the respective bin for each value of x.

bins: numpy.ndarray or IntervalIndex The computed or specified bins.

The object pandas returns is a special Categorical object. it like an array of strings indicating the

- bin name;
- internally it contains a categories array specifying the distinct category names along with a labeling for the ages data in the codes attribute

Example 1: Let's say we have an array of 10 random numbers from 1 to 100 and we wish to separate data into 5 bins of:

(1,20] , (20,40] , (40,60] , (60,80] , (80,100] .

In [44]:
```
df= pd.DataFrame({'number': np.random.randint(1, 100, 10)})
```

In [45]:
```
df
```

Out[45]:

	number
0	21
1	2
2	38
3	43
4	45
5	83
6	46
7	55
8	88
9	2

In [46]:
```
mbins=[1, 20, 40, 60, 80, 100]
pd.cut(df['number'],mbins)
```

Out[46]:
```
0     (20, 40]
1      (1, 20]
2     (20, 40]
3     (40, 60]
4     (40, 60]
5    (80, 100]
6     (40, 60]
7     (40, 60]
8    (80, 100]
9      (1, 20]
Name: number, dtype: category
Categories (5, interval[int64, right]): [(1, 20] < (20, 40] < (40, 60] <
(60, 80] < (80, 100]]
```

In [47]:
```
df['bins'] = pd.cut(x=df['number'], bins=[1, 20, 40, 60, 80, 100])
df
```

Out[47]:

	number	bins
0	21	(20, 40]
1	2	(1, 20]
2	38	(20, 40]
3	43	(40, 60]
4	45	(40, 60]
5	83	(80, 100]
6	46	(40, 60]
7	55	(40, 60]
8	88	(80, 100]
9	2	(1, 20]

In [48]:
```
print(df['bins'].unique())
```

```
[(20, 40], (1, 20], (40, 60], (80, 100]]
Categories (5, interval[int64, right]): [(1, 20] < (20, 40] < (40, 60] <
(60, 80] < (80, 100]]
```

In [49]:
```
# get the frequency of each bin
df.value_counts(df['bins'])
```

Out[49]:
```
bins
(40, 60]     4
(1, 20]      2
(20, 40]     2
(80, 100]    2
(60, 80]     0
dtype: int64
```

Example 2: for example 1 add some labels to bins

In [50]:
```
df = pd.DataFrame({'number': np.random.randint(1, 100, 10)})
df['bins'] = pd.cut(x=df['number'], bins=[1, 20, 40, 60, 80, 100],
                    labels=['1 to 20', '21 to 40', '41 to 60', '61 to 80',
print(df)
```

```
   number       bins
0      41   41 to 60
1      50   41 to 60
2      98  81 to 100
3      72   61 to 80
4      52   41 to 60
5      52   41 to 60
6      74   61 to 80
7       8    1 to 20
8      14    1 to 20
9      63   61 to 80
```

In [51]:
```
print(df['bins'].unique())
```

```
['41 to 60', '81 to 100', '61 to 80', '1 to 20']
Categories (5, object): ['1 to 20' < '21 to 40' < '41 to 60' < '61 to 80'
< '81 to 100']
```

In [52]:
```
# get the frequency of each bin
df.value_counts(df['bins'])
```

Out[52]:
```
bins
41 to 60     4
61 to 80     3
1 to 20      2
81 to 100    1
21 to 40     0
dtype: int64
```

Example: Given data about a group of people in a study, and you want to group them into discrete age buckets

divide these into bins of 18 to 25, 26 to 35, 36 to 60, and 61 and older.

In [53]:
```
ages = [20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
ages
```

Out[53]:
```
[20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

In [54]:
```
bins = [18, 25, 35, 60, 100]
cats = pd.cut(ages, bins)
cats
```

Out[54]:
```
[(18, 25], (18, 25], (18, 25], (25, 35], (18, 25], ..., (25, 35], (60, 10
0], (35, 60], (35, 60], (25, 35]]
Length: 12
Categories (4, interval[int64, right]): [(18, 25] < (25, 35] < (35, 60] <
(60, 100]]
```

In [ ]:

https://pandas.pydata.org/docs/reference/api/pandas.Categorical.codes.html (https://pandas.pydata.org/docs/reference/api/pandas.Categorical.codes.html)

property Categorical.codes

The category codes of this categorical.

Codes are an array of integers which are the positions of the actual values in the categories array.

There is no setter, use the other categorical methods and the normal item setter to change values in the categorical.

Returns: ndarray A non-writable view of the codes array.

In [55]:
```
ages
```

Out[55]:
```
[20, 22, 25, 27, 21, 23, 37, 31, 61, 45, 41, 32]
```

In [56]:
```
cats.codes
```

Out[56]:
```
array([0, 0, 0, 1, 0, 0, 2, 1, 3, 2, 2, 1], dtype=int8)
```

In [57]:
```
cats.categories
```

Out[57]:
```
IntervalIndex([(18, 25], (25, 35], (35, 60], (60, 100]], dtype='interval
[int64, right]')
```

In [58]:
```
# Note: pd.value_counts(cats) are the bin counts for the result of pandas.c
pd.value_counts(cats)
```

Out[58]:
```
(18, 25]     5
(25, 35]     3
(35, 60]     3
(60, 100]    1
dtype: int64
```

Like mathematical notation for intervals:

- a parenthesis means that the side is open,
- square bracket means it is closed (inclusive).
- You can change which side is closed by passing right=False

In [59]:
```
pd.cut(ages, [18, 26, 36, 61, 100], right=False)
```

Out[59]:
```
[[18, 26), [18, 26), [18, 26), [26, 36), [18, 26), ..., [26, 36), [61, 10
0), [36, 61), [36, 61), [26, 36)]
Length: 12
Categories (4, interval[int64, left]): [[18, 26) < [26, 36) < [36, 61) <
[61, 100)]
```

Can also pass your user defined bin names by passing a list or array to the labels option

In [60]:
```
group_names = ['Youth', 'YoungAdult', 'MiddleAged', 'Senior']
pd.cut(ages, bins, labels=group_names)
```

Out[60]:
```
['Youth', 'Youth', 'Youth', 'YoungAdult', 'Youth', ..., 'YoungAdult', 'Se
nior', 'MiddleAged', 'MiddleAged', 'YoungAdult']
Length: 12
Categories (4, object): ['Youth' < 'YoungAdult' < 'MiddleAged' < 'Senio
r']
```

If you pass an integer number of bins to cut instead of explicit bin edges, it will compute equal-length bins based on the minimum and maximum values in the data (uniformly distributed data)

In [61]:
```
data5 = np.random.rand(20)
data5
```

Out[61]:
```
array([0.48670007, 0.53515511, 0.72784372, 0.92051948, 0.740652  ,
       0.03408974, 0.89675927, 0.79973273, 0.31298893, 0.40941487,
       0.74876567, 0.05671149, 0.2462012 , 0.84022727, 0.68625906,
       0.16849273, 0.87502197, 0.39731197, 0.54182095, 0.27215484])
```

In [62]:
```
# uniformly distributed data chopped into fourths
# The precision=2 option limits the decimal precision to two digits.
cats5 = pd.cut(data5, 4, precision=2)
```

In [63]:
```
cats5
```

Out[63]:
```
[(0.48, 0.7], (0.48, 0.7], (0.7, 0.92], (0.7, 0.92], (0.7, 0.92], ...,
(0.033, 0.26], (0.7, 0.92], (0.26, 0.48], (0.48, 0.7], (0.26, 0.48]]
Length: 20
Categories (4, interval[float64, right]): [(0.033, 0.26] < (0.26, 0.48] <
(0.48, 0.7] < (0.7, 0.92]]
```

In [64]:
```
pd.value_counts(cats5)
```

Out[64]:
```
(0.7, 0.92]      8
(0.033, 0.26]    4
(0.26, 0.48]     4
(0.48, 0.7]      4
dtype: int64
```

qcut()

- bins the data based on sample quantiles.
- Depending on the distribution of the data, using cut will not usually result in each bin having the same number of data points.
- Since qcut uses sample quantiles instead, n roughly equal-size bins will be obtained

https://www.geeksforgeeks.org/how-to-use-pandas-cut-and-qcut/
(https://www.geeksforgeeks.org/how-to-use-pandas-cut-and-qcut/)

In [65]:
```
data6 = np.random.randn(1000) # Normally distributed
data6
```

Out[65]:
```
array([ 7.21868920e-01,  2.82615701e+00,  8.13376828e-01,  1.94018680e
-01,
        7.32370777e-01,  7.93000249e-01,  3.40902715e-01, -6.99179204e
-02,
       -1.00884012e+00,  1.25767282e-01,  4.24128304e-01, -6.38047592e
-01,
       -4.90532456e-01, -1.36196699e+00, -1.02323154e-01, -1.78005025e
+00,
       -3.94130137e-02,  9.95276384e-01, -1.48592109e-01, -6.61044273e
-01,
       -3.88255447e-01, -2.14106786e-01, -6.29247765e-01,  1.70572986e
+00,
        2.35193882e-01,  1.36184930e+00, -3.24020473e-01,  6.19760734e
-02,
       -6.63585132e-01, -3.31314847e-03,  3.02715510e-02, -5.60167408e
-01,
       -1.36343093e+00, -5.76740598e-01, -2.27995100e+00, -8.57460796e
-01,
       -8.02048781e-01,  3.51854682e-01, -3.22343748e-01, -2.04539511e
```

In [66]:
```
cats6 = pd.qcut(data6, 4) # Cut into quartiles
cats6
```

Out[66]:
```
[(0.634, 3.084], (0.634, 3.084], (0.634, 3.084], (-0.0942, 0.634], (0.63
4, 3.084], ..., (-0.713, -0.0942], (-0.0942, 0.634], (-3.499, -0.713], (-
0.0942, 0.634], (-0.713, -0.0942]]
Length: 1000
Categories (4, interval[float64, right]): [(-3.499, -0.713] < (-0.713, -
0.0942] < (-0.0942, 0.634] < (0.634, 3.084]]
```

In [67]:
```
pd.value_counts(cats6)
```

Out[67]:
```
(-3.499, -0.713]     250
(-0.713, -0.0942]    250
(-0.0942, 0.634]     250
(0.634, 3.084]       250
dtype: int64
```

In [68]:
```
# Similar to cut you can pass your own quantiles (numbers between 0 and 1,
cuts7 = pd.qcut(data6, [0, 0.1, 0.5, 0.9, 1.])
cuts7
```

Out[68]:
```
[(-0.0942, 1.175], (1.175, 3.084], (-0.0942, 1.175], (-0.0942, 1.175], (-
0.0942, 1.175], ..., (-1.307, -0.0942], (-0.0942, 1.175], (-3.499, -1.30
7], (-0.0942, 1.175], (-1.307, -0.0942]]
Length: 1000
Categories (4, interval[float64, right]): [(-3.499, -1.307] < (-1.307, -
0.0942] < (-0.0942, 1.175] < (1.175, 3.084]]
```

In [69]:
```
pd.value_counts(cuts7)
```

Out[69]:
```
(-1.307, -0.0942]    400
(-0.0942, 1.175]     400
(-3.499, -1.307]     100
(1.175, 3.084]       100
dtype: int64
```

In [ ]:

In [ ]: