PennState World Campus

DAAN862: Analytics Programming in Python

Lesson 4: Data Cleaning, Processing and Manipulating with Pandas

Lesson 4: Data Cleaning, Processing and Manipulating with Pandas

An important task in data mining is data preparation, i.e. cleaning, transforming, and rearranging. Pandas provides various simple and effective preparation methods.

At the end of this lesson, students will be able to:

- Identify and handle missing data by using pandas
- Perform data transformation tasks by using pandas
- Manipulate strings by utilizing string methods and pandas

By the end of this lesson, please complete all readings and assignments found in the Lesson 4 Course Schedule.

Lesson 4.1: Handling Missing Data

Pandas adopts a convention used in R for missing value NA (Not Available). *pandas* built-in methods make handling missing data very easy.

See Table 4.1.1 for a list of methods related to missing data handling.

Table 4.1.1: NA Handling Methods

Argument	Description

Argument	Description
dropna	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
fillna	Fill in missing data with some value or using an interpolation method such as 'ffill' or 'bfill'.
isnull	Return a like-type object containing boolean values indicating which values are missing / NA.
notnull	Negation of isnull

(click the titles to learn more)

Identifying Missing Values

In Figure 4.1 we created artifical data. The data was created by a normal distribution which randomly selects three numbers as NA and one as None.

```
In [1]: import os
   ...: import pandas as pd
   ...: import numpy as np
   ...:
   ...:
   ...: # Create a random data and randomly assign 3 nan
   ...: np.random.seed(123)
   ...: data = np.random.normal(2, 2, 20)
   ...: data[2] = None
   ...: data[np.random.randint(3, 20, 3)] = np.nan
   ...: data = data.reshape(4, 5)
   ...: df = pd.DataFrame(data, columns = list('abcde'))
   ...: df
Out[1]:
          a         b         c         d         e
0 -0.171261  3.994691       NaN -1.012589  0.842799
1  5.302873       NaN  1.142175  4.531873       NaN
2  0.642228  1.810582       NaN  0.722196  1.112036
3  1.131297  6.411860  6.373572  4.008108  2.772373
```

Fig 4.1(click to enlarge)

As shown in Fig 4.1, the None data type will also be considered as NaNs.

Isnull is the method to identify missing data, which returns a DataFrame with Boolean values. By adding statistic function such as *sum*, you can get the total count of Na's in each coloum.

```
In [2]: df.isnull()
Out[2]:
       a      b      c      d      e
0  False  False   True  False  False
1  False   True  False  False   True
2  False  False   True  False  False
3  False  False  False  False  False

In [3]: df.isnull().sum(axis = 0)   # count nans in each column
Out[3]:
a    0
b    1
c    2
d    0
e    1
dtype: int64

In [4]: df.isnull().values.sum()      # count total nans
Out[4]: 4
```

Fig 4.2 (click to enlarge)

Filtering Out Missing Data

There is a number of options when using *dropna* method to filter out missing data. In Fig 4.3 you can find an example of the *dropna* method:

```
In [5]: df.dropna()             # drop rows contain nans
Out[5]:
          a        b         c         d         e
3  1.131297  6.41186  6.373572  4.008108  2.772373

In [6]: df.dropna(axis = 1)     # drop columns contain nans
Out[6]:
          a         d
0 -0.171261 -1.012589
1  5.302873  4.531873
2  0.642228  0.722196
3  1.131297  4.008108

In [7]: df.dropna(how = 'all')  # drop rows whose all of values are nan
Out[7]:
          a         b         c         d         e
0 -0.171261  3.994691       NaN -1.012589  0.842799
1  5.302873       NaN  1.142175  4.531873       NaN
2  0.642228  1.810582       NaN  0.722196  1.112036
3  1.131297  6.411860  6.373572  4.008108  2.772373
```

Fig 4.3 (click to enlarge)

Filling In Missing Data

If you would like to fill in missing data instead of filtering out the missing data, fillna is the correct method to use. You are able to fill in missing values with a constant like 0 in Fig 4.4:

```
In [8]: df.fillna(0)
Out[8]:
          a         b         c         d         e
0 -0.171261  3.994691  0.000000 -1.012589  0.842799
1  5.302873  0.000000  1.142175  4.531873  0.000000
2  0.642228  1.810582  0.000000  0.722196  1.112036
3  1.131297  6.411860  6.373572  4.008108  2.772373
```

Fig 4.4 (click to enlarge)

When usinga dict in fillna, a different fill value will be applied to each column. Please refer to Fig 4.5 for the differnt fill values that were applied to each column:

```
In [11]: df.fillna({'b': 0.5, 'c': 0, 'e': 2})
Out[11]:
          a         b         c         d         e
0 -0.171261  3.994691  0.000000 -1.012589  0.842799
1  5.302873  0.500000  1.142175  4.531873  2.000000
2  0.642228  1.810582  0.000000  0.722196  1.112036
3  1.131297  6.411860  6.373572  4.008108  2.772373
```

Fig 4.5 (click to enlarge)

As shown in Fig 4.5, all missing values in b,c, and e were replaced with 0.5, 0 and 2, respectively.

You can also use the value at the same column but the previous row is is used to fill NA's.

```
In [12]: df.fillna(method = 'ffill')
Out[12]:
          a         b         c         d         e
0 -0.171261  3.994691       NaN -1.012589  0.842799
1  5.302873  3.994691  1.142175  4.531873  0.842799
2  0.642228  1.810582  1.142175  0.722196  1.112036
3  1.131297  6.411860  6.373572  4.008108  2.772373
```

Fig 4.6 (click to enlarge)

As shown in Fig 4.6, there is still a missing value in column c since there is not previous row for it.

You can also fill the missing columns with column means as shown in Fig 4.7:

```
In [13]: df.fillna(df.mean())
Out[13]:
          a         b         c         d         e
0 -0.171261  3.994691  3.757873 -1.012589  0.842799
1  5.302873  4.072378  1.142175  4.531873  1.575736
2  0.642228  1.810582  3.757873  0.722196  1.112036
3  1.131297  6.411860  6.373572  4.008108  2.772373
```

Fig 4.7 (click to enlarge)

See table 4.1.2 for a reference on fillna.

Table 4.1.2: Fillna Function Arguments

Argument	Description
value	Scalar value or dict-like object to use to fill missing values
method	Interpolation, by default 'ffill' if a function called with no other arguments
axis	Axis to fill on, default axis=0
inplace	Modify the calling object without producing a copy
limit	For forward and backward filling, the maximum number of consecutive periods to fill

Transcript

In this video, I will introduce how to handle missing values.

let's import numpy and pandas packages.

First, let's create a dataframe which contains missing values.

Line 14 is used to set the random seed

Line 15 is used to generate a 20x1 array from a normal distribution with the mean of 2 and the standard deviation of 2.

In line 16, we assign None to the third element.

In Line 18, we randomly select three items and assign np.nan to them.

In line 19, we reshaped it into a 4x5 2-D array.

In Line 20, we convert the 2D array to a DataFrame with column names of a, b, c, d, e.

Here is how Dataframe looks like, we can see that it contains NAs. Now, we have a DataFrame ready for analysis.

Isnull is the method to check if elements in DataFrame are null or not. We can see that both None and np.nan will be considered as missing values. From the output, we can see that it returns a boolean dataFrame.

If you want to count missing values in each column, you need to add .sum(axis = 0)

If you want to find the total missing values, You can .values which will convert the boolean dataframe to an array, you can use sum method of array. Since the sum method of numpy array will calculate the total sum by default.

Dropna is the method to remove missing values. Since row 0, 1, 2 contains NAs. It returns row 3 here.

By default, it will drop row with missing values. Since column b, c, e contains NAs. Only column a, d have been kept.

By adding the argument of axis = 1, it will drop columns with missing values.

If you want to only drop rows whose elements are all NAs, you need to use argument how equals to "all". Here we don't have such rows, it returns the same DataFrame as df.

Fillna is the method to replace missing values with something else.

If you can fill all NAs with the same value, Like Line 32, fill all Nas with 0.

Or you can filling different values for different columns by using a dictionary with column names as keys and something you want to replace NAs as values. Like Like 34 here, it will fill NAs in column b with 0.5, column c with 0 and column e with 2.

Fillna has an argument method, you can choose fillna with different methods. Here in Line 35, we use ffill which means to fill nans with previous rows in the same column.

You can also fillna with a vector like df.mean() here. since df.means returns a dictionary-like object with column names as index and means as values.

Lesson 4.2: Data Transformation I

In this section, you will learn data transformations such as removing duplicates and replacing values.

(click the titles to learn more)

Removing Duplicates

Duplicate rows are common issues in data cleaning. Refer to Figure 4.8 for data with duplicates:

```
In [21]: data = pd.DataFrame({'k1': ['one', 'two'] * 2 + ['two'],
    ...:                      'k2': [1, 1, 3, 4, 4]})

In [22]: data
Out[22]:
    k1  k2
0  one   1
1  two   1
2  one   3
3  two   4
4  two   4
```

Fig 4.8 (click to enlarge)

duplicated method of DataFrame returns a Boolean Series which indicates if each row is a duplicate or not as shown in Figure 4.9:

```
In [23]: data.duplicated()
Out[23]:
0    False
1    False
2    False
3    False
4     True
```

Fig 4.9 (click to enlarge)

drop duplicates returns a DataFrame after removing the duplicates as shown in Figure 4.10:

```
In [24]: data.drop_duplicates()
Out[24]:
    k1  k2
0  one   1
1  two   1
2  one   3
3  two   4
```

Fig 4.10 (click to enlarge)

Figures 4.9 and 4.10 are two examples that consider all columns.

You can specify a column or several columns to detect duplicates. For example, in Figure 4.11 you can filter duplicates based on the 'k1' column:

```
In [25]: data['k3'] = range(5)

In [26]: data.drop_duplicates(['k1'])
Out[26]:
    k1  k2  k3
0  one   1   0
1  two   1   1
```

Fig 4.11 (click to enlarge)

Transforming Data Using A Function or Mapping

For most data you may need to perform some value transformation for a column in a DataFrame. *map* method can perform value transformation either by a *dict* or an anonymous function *lambda.* Figure 4.12 shows hypothetic data:

```
In [32]: data = pd.DataFrame({'City': ['New York', 'Chichago', 'Berlin',
    ...:                              'London', 'Toronto', 'Manchester'],
    ...:                     'Variable': [4, 3, 12, 6, 7.5, 8]})

In [33]: data
Out[33]:
         City  Variable
0    New York       4.0
1    Chichago       3.0
2      Berlin      12.0
3      London       6.0
4     Toronto       7.5
5  Manchester       8.0
```

Fig 4.12 (click to enlarge)

Below in Figure 4.13, you have a *dict* to show you which country each city belongs to:

```
In [34]: city_country = {'New York': 'USA',
    ...:                 'Chichago': 'USA',
    ...:                 'Berlin': 'Germany',
    ...:                 'London': 'UK',
    ...:                 'Toronto': 'Canada',
    ...:                 'Manchester': 'UK'}
```

Fig 4.13 (click to enlarge)

If you want to add a column that indicates the country of each city, you can use the *map* function as shown in Figure 4.14:

```
In [35]: data['Country'] = data.City.map(city_country)

In [36]: data
Out[36]:
         City  Variable  Country
0    New York       4.0      USA
1    Chichago       3.0      USA
2      Berlin      12.0  Germany
3      London       6.0       UK
4     Toronto       7.5   Canada
5  Manchester       8.0       UK
```

Fig 4.14 (click to enlarge)

Figure 4.15 shows how you can also use an anonymous function lambda that does all the work:

```
In [37]: data['City'].map(lambda x: city_country[x])
Out[37]:
0        USA
1        USA
2    Germany
3         UK
4     Canada
5         UK
Name: City, dtype: object
```

Fig 4.15 (click to enlarge)

The lambda function in Figure 4.15 will use the values in column city as keys and return corresponding values in the dict *city_country*.

Replacing Values

Filling in missing data is a special case of general value replacement. As shown before, map function can modify values in a subset, while replace method is simpler and more flexible. In Figure 4.16 you can see data that was created to show the replace method:

```
In [55]: np.random.seed(189)
    ...: data = np.random.randint(-20, 20, 12)
    ...: data[np.random.randint(0, 11, 3)] = -100
    ...: data = data.reshape(4, 3)
    ...: df = pd.DataFrame(data, columns = list('ABC'))
    ...: df
Out[55]:
    A    B    C
0 -20   -1    8
1  19 -100   15
2   5   -1 -100
3   5 -100   -4
```

Fig 4.16 (click to enlarge)

The -100 values in Figure 4.17 looks like suspicious values for missing data. You can replace them with NA values and handle them later together with missing values.

```
In [57]: df.replace(-100, np.NAN)
Out[57]:
    A    B     C
0 -20 -1.0   8.0
1  19  NaN  15.0
2   5 -1.0   NaN
3   5  NaN  -4.0
```

Fig 4.17 (click to enlarge)

By default, it will produce a new DataFrame unless you use the argument *inplace*=True in the function.

Transcript

In this video, I will introduce how to use Pandas for Data transform tasks.

First part is to handle duplicates.

Let’s create a DataFrame which contains duplicates. Here is how data looks like. We can see that row 3 and 4 are exactly the same.

Duplicated is the method to check if rows are duplicated or not. It will return a Series of Boolean. The result shows that Row 4 is duplicated.

Drop_duplicates is the method to remove duplicates. We can see that it removed Row 4.

You can also remove duplicated based the values in one or more column.

Let’s add another column K3 in Data.

If we put the a list of column name in drop_duplicates method. It will remove rows with duplicates in this column. Here we, can see that, Only Row 0 and 1 are remained.

The second part we show how to transform a column by using map method.

Here we have a DataFrame which has two columns: City column contains city names, and a variable with some values. If you want to find the country these cities belong to, you can create a dictionary with city names as keys and countries it belongs to as values, Like the city_country dictionary here.

In line 65, we create a new column Country and its values equals to data.city.map(city_country), it will apply the dictionary to all elements in City column.

You can also use an anonymous function lambda in the map method. Lambda is the key work to create anonymous function, x is the variable. Things after the colon is the return of the function. Here it returns city_country[x], the value for the key equals to x.

The last part is to replace values.

Here we sample 12 values from integers between -20 to 20.

In Line 73 we randomly select three elements and assign -100 to it.

In Line 74 reshape the 1-D array to a 4x3 2-D array.

In Line 75 we convert it to a DataFrame with column names of A, B, C.

Here is how df looks like.

Replace is the method to replace existing values with new values.

Line 79 will replace elements with values of -100 to np.nan

Lesson 4.3: Data Transformation II

(click the titles to learn more)

Detecting and Filtering Outliers

Filtering or transforming outliers is another important data process. Figure 4.18 uses a DatatFrame with normally distributed values:

```
In [20]: np.random.seed(123)
    ...: data = pd.DataFrame(np.random.randn(1000, 4))
    ...: data.describe()
Out[20]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean     -0.007502     0.039160    -0.010286     0.024285
std       0.977024     0.973484     1.012230     0.970421
min      -3.167055    -2.920029    -3.801378    -3.231055
25%      -0.662012    -0.636160    -0.687717    -0.599195
50%      -0.024843     0.062549     0.007035     0.038718
75%       0.613950     0.672448     0.664586     0.683228
max       3.050755     2.850708     2.766603     3.571579
```

Fig 4.18 (click to enlarge)

If you consider the values beyond three standard deviations are outliers, you can identify them in column '2' in Figure 4.19:

```
In [21]: col2= data[2]
    ...: col2[np.abs(col2) > 3] # larger than 3 standard deviation
Out[21]:
409   -3.801378
423   -3.587494
510   -3.066988
```

Fig 4.19 (click to enlarge)

Figure 4.20 shows how to find out all rows in the data which contain all outliers, In addtion, you can use any method on a Boolean DataFrame:

```
In [22]: data[(np.abs(data) > 3).any(1)]
Out[22]:
            0         1         2         3
48   0.199582 -0.126118  0.197019 -3.231055
182  0.272735  0.425336 -0.230904  3.571579
235 -3.167055 -0.713989 -1.112364 -1.254184
409  0.151037  0.069403 -3.801378 -1.127172
423  0.231228  1.076113 -3.587494  1.148869
510  0.506533  0.644099 -3.066988 -1.349275
910  3.050755  0.296552 -0.481843  0.930787
```

Fig 4.20 (click to enlarge)

You can reset the values of the outliers as shown in Figure 4.21:

```
In [23]: data[(np.abs(data) > 3)] = np.sign(data) * 3

In [24]: data.describe()
Out[24]:
                 0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean     -0.007386     0.039160    -0.008831     0.023944
std       0.976340     0.973484     1.007422     0.967746
min      -3.000000    -2.920029    -3.000000    -3.000000
25%      -0.662012    -0.636160    -0.687717    -0.599195
50%      -0.024843     0.062549     0.007035     0.038718
75%       0.613950     0.672448     0.664586     0.683228
max       3.000000     2.850708     2.766603     3.000000
```

Fig 4.21 (click to enlarge)

*The np.sign(data) calculates the signs (1 and –1 values) of the data.

After resets the outliers to either 3 or -3, you can see the range [min, max] of all columns is within [-3, 3].

Discretization and Binning

For data processing, sometimes you have to discretize continuous data into "bins". Figure 4.21 shows an example of an array with the range from 18 to 4:

```
In [25]: values = np.random.randint(18, 54, 10)

In [26]: values
Out[26]: array([51, 32, 28, 36, 19, 48, 46, 52, 32, 32])
```

Fig 4.22 (click to enlarge)

You want to divide the array into bins of 0-10, 10-20, 20-30, 30-40, and 40+. Lets create the sepearation first as shown in Fiugre 4.23:

```
In [27]: bins = np.array([0] + list(range(1, 5))+ [np.inf]) * 10

In [28]: bins
Out[28]: array([ 0., 10., 20., 30., 40., inf])

In [29]: cats = pd.cut(values, bins)

In [30]: cats
Out[30]:
[(40.0, inf], (30.0, 40.0], (20.0, 30.0], (30.0, 40.0], (10.0, 20.0], (40.0, inf],
(40.0, inf], (40.0, inf], (30.0, 40.0], (30.0, 40.0]]
Categories (5, interval[float64]): [(0.0, 10.0] < (10.0, 20.0] < (20.0, 30.0] <
(30.0, 40.0] < (40.0, inf]]
```

Fig 4.23 (click to enlarge)

Now. you are able to use the *cut* function to the array value as shown in Figure 4.24:

```
In [27]: bins = np.array([0] + list(range(1, 5))+ [np.inf]) * 10

In [28]: bins
Out[28]: array([ 0., 10., 20., 30., 40., inf])

In [29]: cats = pd.cut(values, bins)

In [30]: cats
Out[30]:
[(40.0, inf], (30.0, 40.0], (20.0, 30.0], (30.0, 40.0], (10.0, 20.0], (40.0, inf],
(40.0, inf], (40.0, inf], (30.0, 40.0], (30.0, 40.0]]
Categories (5, interval[float64]): [(0.0, 10.0] < (10.0, 20.0] < (20.0, 30.0] <
(30.0, 40.0] < (40.0, inf]]
```

Fig 4.24 (click to enlarge)

The cut function in Figure 4.24 returns a special Categorical data. The original values of the data have been converted to the intervals it belongs to.

This returns a special catergorical data. You can consider it like string array with bin ranges as names. Figure 4.25 shows how you can access the codes (the level of intervals the data points belongs to) and categories by:

```
In [31]: cats.codes
Out[31]: array([4, 3, 2, 3, 1, 4, 4, 4, 3, 3], dtype=int8)

In [32]: cats.categories
Out[32]:
IntervalIndex([(0.0, 10.0], (10.0, 20.0], (20.0, 30.0], (30.0, 40.0], (40.0, inf]]
              closed='right',
              dtype='interval[float64]')

In [33]: pd.value_counts(cats)
Out[33]:
(40.0, inf]     4
(30.0, 40.0]    4
(20.0, 30.0]    1
(10.0, 20.0]    1
(0.0, 10.0]     0
dtype: int64
```

Fig 4.25 (click to enlarge)

value counts can be used to count the frequency of each category as shown in Figure 4.26:


```
In [31]: cats.codes
Out[31]: array([4, 3, 2, 3, 1, 4, 4, 4, 3, 3], dtype=int8)

In [32]: cats.categories
Out[32]:
IntervalIndex([(0.0, 10.0], (10.0, 20.0], (20.0, 30.0], (30.0, 40.0], (40.0, inf]]
              closed='right',
              dtype='interval[float64]')

In [33]: pd.value_counts(cats)
Out[33]:
(40.0, inf]     4
(30.0, 40.0]    4
(20.0, 30.0]    1
(10.0, 20.0]    1
(0.0, 10.0]     0
dtype: int64
```


Fig 4.26 (click to enlarge)

You can also specify how many bins you would like to use which will create equal-length bins based on the range of data as shown in Figure 4.27:

```
In [34]: pd.cut(values, 5, precision = 2)
Out[34]:
[(45.4, 52.0], (25.6, 32.2], (25.6, 32.2], (32.2, 38.8], (18.97, 25.6], (45.4,
52.0], (45.4, 52.0], (45.4, 52.0], (25.6, 32.2], (25.6, 32.2]]
Categories (5, interval[float64]): [(18.97, 25.6] < (25.6, 32.2] < (32.2, 38.8] <
(38.8, 45.4] < (45.4, 52.0]]
```

Fig 4.27 (click to enlarge)

*The precision=2 is used to set the decimal precision to two digits.

There is another discretization function called *qcut*, which is shown in Figure 4.28. Qcut cuts data based on its quanities. Typically this will result in roughly equal sized bins since it uses sample quantile:

```
In [33]: values2 = np.random.randn(100)

In [34]: cats2 = pd.qcut(values2, 4)

In [35]: cats2
Out[35]:
[(-0.8, 0.0319], (-2.8, -0.8], (-0.8, 0.0319], (-2.8, -0.8], (-2.8, -0.8], ...,
(0.0319, 0.809], (0.809, 2.598], (-0.8, 0.0319], (0.809, 2.598], (0.0319, 0.809]]
Length: 100
Categories (4, interval[float64]): [(-2.8, -0.8] < (-0.8, 0.0319] < (0.0319,
0.809] < (0.809, 2.598]]

In [36]: pd.value_counts(cats2)
Out[36]:
(0.809, 2.598]     25
(0.0319, 0.809]    25
(-0.8, 0.0319]     25
(-2.8, -0.8]       25
dtype: int64
```

Fig 4.28 (click to enlarge)

Similar to *cut* function, you can choose your own quantiles (values between 0 and 1).

Dummy Variables

For some machine learning algothrims such as linear regression, logistic regression, neural networks, etc, they require the input variables to be numeric. You will have to convert categorical variables to dummy variables. If a categorical column contains k distinct values, it will use k columns containing values of 1's and 0's to represent it.

Get dummies is the correct function to use for this task. Let's use a previous example and convert the 'key' column in the DataFrame to dummy variables as shown in Figure 4.29:

```
In [37]: df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'],
    ...:                    'data1': range(6)})

In [38]: pd.get_dummies(df['key'])
Out[38]:
   a  b  c
0  0  1  0
1  0  1  0
2  1  0  0
3  0  0  1
4  1  0  0
5  0  1  0
```

Fig 4.29 (click to enlarge)

You can see that each value in key column becomes a column, while all the values of new columns are either 1's or 0's.

If you want to add a prefix to the new column names, you can use the prefix argument in the *get_dummies* function. You can join the new columns with the rest of the columns of the original data *df* as seen in Figure 4.30:

```
In [39]: dummies = pd.get_dummies(df['key'], prefix='key')

In [40]: df_with_dummy = df[['data1']].join(dummies)

In [41]: df_with_dummy
Out[41]:
   data1  key_a  key_b  key_c
0      0      0      1      0
1      1      0      1      0
2      2      1      0      0
3      3      0      0      1
4      4      1      0      0
5      5      0      1      0
```

Fig 4.30 (click to enlarge)

Transcript

In this video, I will continue to introduce how to use python for Data transformation tasks.

The first one is to detect and filter outliers.

Let draw a 1000x 4 2-D array from the standard normal distribution.

We can use the describe method to check the statistical information.

Let’s select the third column and call it col2.

If we consider value larger than 3 standard deviations as outliers. We can detect them by using Lime 89. Np.abs(col2) > 3 is to compare if the absolute value of col2 is larger than 3 or not. Line 89 will returns the outliers.

You can find rows by using any method of Boolean dataframe. 1 means rows. Line 91 returns all rows with outliers.

You can assign values to outliers like Line 94. It will assign the sign of the data times 3 to outliers.

You can use describe again, we can see that min, max are -3 or 3 now.

The next part is for discretization and binning.

We create a random data by sample 10 numbers between 18 to 54.

Here is how values look like:

In order to discretize the data, you can create a bin first which contains the separation values.

Here we want cut the data into 0, 10, 20, 30, 40,and infinite.

Here is how bins look like:

The pd.cut function is to cut continues values. Line 102 will cut values according to the bins

The output is the intervals the values belong to. It also show all categories: we have an interval. And they are 0-10， 10-20, 20-30, 30-40 and 40 +

If you want to get the rank of the categories, you can use codes attribute, here we can see that it return its rank of the intervals.

Categories attribute returns all intervals.

You can use value_counts to count the frequency for each interval or category.

If you assign an integer instead of a vector, like here we use 5. The data will be cut into 5 intervals with equal length.

Now we create another random data from standard normal distribution. Qcut will cut the data based on its quantiles. 4 is how many intervals you want to use.

Qcut will create intervals with equal frequency.

You can also specify the quantiles by using a list here.

The last part shows how to create dummy variables.

If you have a multilevel category variables, some time you have to convert it to several binary variables.

Get_dummies is the function for this task. You can put the variables inside of the method. Each category will become a column.

You can add prefix to column names by using prefix argument.

Now you can add the dummies to the original data by using join method.

Here is how df_with_dummy looks like.

Lesson 4.4: String Manipulation

Python is a poplular language to manipulate raw text data. Most text operations are built-in string methods. pandas also provides vectorized string operations.

String Object Methods:

For common text operation, built-in string methods are sufficient. See Table 4.4.1 for main Python's string methods.

Table 4.1.1: Python Built-in String Methods

Method	Descroption

Method	Description
count	Return the number of non-overlapping occurrences of substring in the string.
endswith	Returns True if string ends with suffix.
startswith	Returns True if string starts with prefix.
join	Use string as delimiter for concatenating a sequence of other strings.
index	Return position of first character in substring if found in the string; raises ValueError if not found.
find	Return position of first character of *first* occurrence of substring in the string; like index, but returns –1 if not found.
rfind	Return position of first character of *last* occurrence of substring in the string; returns –1 if not found.
strip, rstrip, lstrip	Trim whitespace, including newlines; equivalent to x.strip() (and rstrip, lstrip, respectively) for each element.
split	Break string into list of substrings using passed delimiter.
lower	Convert alphabet characters to lowercase.
upper	Convert alphabet characters to uppercase.
casefold	Convert characters to lowercase, and convert any region-specific variable character combinations to a common comparable form.
ljust, rjust	Left justify or right justify, respectively; pad opposite side of string with spaces (or some other fill character) to return a string with a minimum width.

Examples

Below are examples of how to use the methods associated with Table 4.4.1:

(click the tabs to learn more)

Split a comma-seperated string into a list of words:

```
In [41]: s = 'Python is a great tool for data analysis'

In [42]: words = s.split(' ')
```

Fig 4.31 (click to enlarge)

Make Upper Case:

```
In [43]: [w.upper() for w in words]
Out[43]: ['PYTHON', 'IS', 'A', 'GREAT', 'TOOL', 'FOR', 'DATA', 'AN
```

Fig 4.32 (click to enlarge)

Convert a List:

```
In [44]: ' '.join(words)
Out[44]: 'Python is a great tool for data analysis'
```

Fig 4.33 (click to enlarge)

Membership and locating a character or a word:

```
In [45]: 'data' in s
Out[45]: True

In [46]: s.index('a')
Out[46]: 10

In [47]: s.find('new')
Out[47]: -1
```

Fig 4.34 (click to enlarge)

There is a difference between *find* and *index*: *index* gives an exception error if a substring is not found, while *find* returns -1.

Frequency:

count returns the number of occurences of a substring as shown in Figure 4.35:

```
In [48]: s.count('for')
Out[48]: 1
```

Fig 4.35 (click to enlarge)

Replace Word:

replace will substitute one pattern with another. For deleting patterns, you can pass an empty string as shown in Figure 4.36:

```
In [49]: s.replace('data', 'Data')
Out[49]: 'Python is a great tool for Data analysis'
```

Fig 4.36 (click to enlarge)

Vectorized String Functions in pandas:

pandas provides very simple vectorized string functions. See Table 4.4.2 for common string methods in pandas.

Table 4.4.2: Common Vectorized String Methods in Pandas

Method	Description
cat	Concatenate strings element-wise with optional delimiter
contains	Return boolean array if each string contains pattern/regex
count	Count occurrences of pattern
extract	Use a regular expression with groups to extract one or more strings from a Series of strings; the result
will	Be a DataFrame with one column per group
endswith	Equivalent to x.startswith(pattern) for each element
startswith	Equivalent to x.startswith(pattern) for each element
findall	Compute list of all occurrences of pattern/regex for each string
get	Index into each element (retrieve i-th element)

Method	Description
isalnum	Equivalent to built-in str.alnum
isalpha	Equivalent to built-in str.isalpha
isdecimal	Equivalent to built-in str.isdecimal
isdigit	Equivalent to built-in str.isdigit
islower	Equivalent to built-in str.islower
isnumeric	Equivalent to built-in str.isnumeric
isupper	Equivalent to built-in str.isupper
join	Join strings in each element of the Series with passed separator
len	Compute length of each string
lower	Upper Convert cases; equivalent to x.lower() or x.upper() for each element
match	Use re.match with the passed regular expression on each element, returning matched groups as list
pad	Add whitespace to left, right, or both sides of strings
center	Equivalent to pad(side='both')
repeat	Duplicate values (e.g., s.str.repeat(3) is equivalent to x * 3 for each string)
replace	Replace occurrences of pattern/regex with some other string
slice	Slice each string in the Series
split	Split strings on delimiter or regular expression
strip	Trim whitespace from both sides, including newlines
rstrip	Trim whitespace on right side
lstrip	Trim whitespace on left side

Examples

Below are examples of how to use the methods listed in Table 4.2.2:

A subset of mtcars datasets will be used as an example as shown in Figure 4.37:

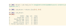

Fig
4.37

(click the tabs to learn more)

Split

You can split the 'model' column and assign the first element to a new column 'Compnay' as shown in Figure 4.38:

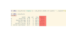

Fig
4.38

Contains

Refer to Figure 4.39 to check if the strings in the 'model' column contain a character or a word:

Fig
4.39

startswith

Refer to Figure 4.40 to check if values in the 'model' column start with a character or a word:

Fig
4.40

Slice

You can also slice strings using the syntax shown in Figure 4.41:

Fig
4.41

Transcript

In this video, I will introduce string operations.

String data types have built-in method which you can use. Here we give a few examples.

S is a string " Python is a great tool for data analysis".

The split is the method to split the string into a list, here we use space to split s.

Upper is the method to convert all character to upper case.

Line 132 is a simple version of for loop. Using square brackets outside means to return a list. The w in words for will be converted to uppercase.

space. Join(words) to concate a list to a string with space.

In is to find out if a word in the string. It returns Ture or False.

The index is a method to return is the index for a word.

Find is similar to index, the difference is that it will return -1 if the word cannot be found. Here since we cannot find new in S, it returns -1, otherwise it returns the index.

Count is to count the frequency of the word. Line 143 count how many times word for appears.

Replace is to replace a word by another word. Line 143 replace data with Data D in Capital.

Pandas support vectorized string operation， which means you can apply a string method to a column.

Here we select partial data of mtcars as an example. Here is how sub_matcars looks like:

It has four column and five rows.

All string method can be find under str. You need to use .str first and then select string method.

Line 152 create a company column by using the first element of the list by splitting the model column by space. Expand = True means Expand the splitted strings into separate columns.

This is how sub_mtcars looks like now.

Contains is the method to check if all element in the column contains a word. Here we check if the column model contains word “Mazda”.

Line 156 check if elements in model column start with ‘M’ or not.

Line 158 select first 4 letters of model column.

Lesson 4 References (6 of 6)

Lesson 4 References

- https://pandas.pydata.org/pandas-docs/stable/tutorials.html (https://pandas.pydata.org/pandas-docs/stable/tutorials.html)