21CSS303T
DATA SCIENCE
UNIT-2 Part-2
DATA WRANGLING, DATA CLEANING AND PREPARATION

- **Reshaping**
- **Pivoting**
- **Data Cleaning and Preparation**
- **Handling Missing Data**
- **Data Transformation**
- **String Manipulation**
- **Summarizing**
- **Binning**
- **Classing and Standardization**
- **Outlier/Noise & Anomalies**

RESHAPE AND PIVOTING

There are a number of basic operations for rearranging tabular data. These are alternatingly referred to as reshape or pivot operations.

a) **Reshaping with Hierarchical Indexing:** Hierarchical indexing provides a consistent way to rearrange data in a DataFrame.
There are two primary actions:
Stack: This "rotates" or pivots from the columns in the data to the rows
Unstuck: This **pivots** from the rows into the columns.

```
data = pd.DataFrame(np.arange(6).reshape((2, 3)), index=pd.Index(['Ohio', 'Colorado'], name='state'),
columns=pd.Index(['one', 'two', 'three'], name='number'))
data
```

Output:

```
number      one    two    three
state
Ohio        0      1      2
Colorado    3      4      5
```

Using the **stack** method on this data pivots the columns into the rows, producing a Series:

```
result = data.stack()
result
```

Output:

```
state       number
Ohio        one       0
            two       1
            three     2
Colorado    one       3
            two       4
            three     5
```

From a hierarchically indexed Series, you can rearrange the data back into a DataFrame with unstack:

```
result.unstack()
```

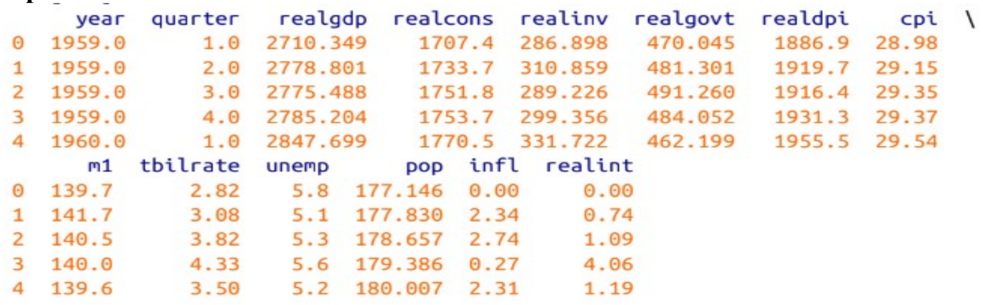
Output:

```
number      one    two    three
state
Ohio        0      1      2
Colorado    3      4      5
```

b) **Pivoting "Long" to "Wide" Format:** A common way to store multiple time series in databases and CSV is in so-called **long or stacked format**.

```
data = pd.read_csv('examples/macrodata.csv')
data.head()
```

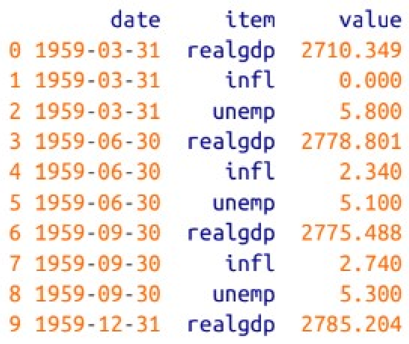
Output:



```
     year  quarter   realgdp  realcons  realinv  realgovt  realdpi    cpi  \
0  1959.0      1.0  2710.349    1707.4  286.898   470.045   1886.9  28.98
1  1959.0      2.0  2778.801    1733.7  310.859   481.301   1919.7  29.15
2  1959.0      3.0  2775.488    1751.8  289.226   491.260   1916.4  29.35
3  1959.0      4.0  2785.204    1753.7  299.356   484.052   1931.3  29.37
4  1960.0      1.0  2847.699    1770.5  331.722   462.199   1955.5  29.54
      m1  tbilrate  unemp      pop  infl  realint
0  139.7      2.82    5.8  177.146  0.00     0.00
1  141.7      3.08    5.1  177.830  2.34     0.74
2  140.5      3.82    5.3  178.657  2.74     1.09
3  140.0      4.33    5.6  179.386  0.27     4.06
4  139.6      3.50    5.2  180.007  2.31     1.19
```

```
periods = pd.PeriodIndex(year=data.year, quarter=data.quarter, name='date')
columns = pd.Index(['realgdp', 'infl', 'unemp'], name='item')
data = data.reindex(columns=columns)
data.index = periods.to_timestamp('D', 'end')
ldata = data.stack().reset_index().rename(columns={0: 'value'})
ldata[:10]
```

Output:

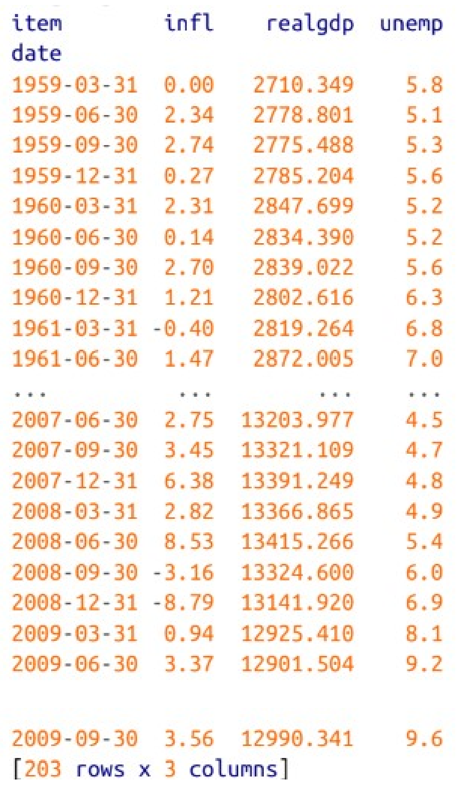


```
        date     item     value
0 1959-03-31  realgdp  2710.349
1 1959-03-31     infl     0.000
2 1959-03-31    unemp     5.800
3 1959-06-30  realgdp  2778.801
4 1959-06-30     infl     2.340
5 1959-06-30    unemp     5.100
6 1959-09-30  realgdp  2775.488
7 1959-09-30     infl     2.740
8 1959-09-30    unemp     5.300
9 1959-12-31  realgdp  2785.204
```

This is the so-called **long format** for multiple time series, or other observational data with two or more keys (here, our keys are date and item). Each row in the table represents a single observation. Data is frequently stored this way in relational databases like MySQL, as a fixed schema (column names and data types) allows the number of distinct values in the item column to change as data is added to the table. In the previous example, date and item would usually be the primary keys (in relational database parlance), offering both relational integrity and easier joins. In some cases, the data may be more difficult to work with in this format; you might prefer to have a DataFrame containing one column per distinct item value indexed by timestamps in the date column. Data-Frame's pivot method performs exactly this transformation:

```
pivoted = ldata.pivot('date', 'item', 'value')
pivoted
```

Output:



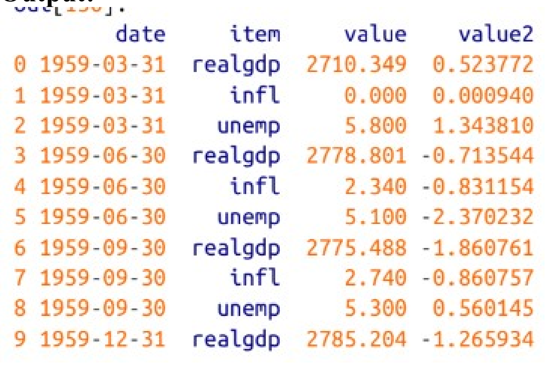
```
item        infl    realgdp  unemp
date
1959-03-31  0.00   2710.349    5.8
1959-06-30  2.34   2778.801    5.1
1959-09-30  2.74   2775.488    5.3
1959-12-31  0.27   2785.204    5.6
1960-03-31  2.31   2847.699    5.2
1960-06-30  0.14   2834.390    5.2
1960-09-30  2.70   2839.022    5.6
1960-12-31  1.21   2802.616    6.3
1961-03-31 -0.40   2819.264    6.8
1961-06-30  1.47   2872.005    7.0
...          ...        ...    ...
2007-06-30  2.75  13203.977    4.5
2007-09-30  3.45  13321.109    4.7
2007-12-31  6.38  13391.249    4.8
2008-03-31  2.82  13366.865    4.9
2008-06-30  8.53  13415.266    5.4
2008-09-30 -3.16  13324.600    6.0
2008-12-31 -8.79  13141.920    6.9
2009-03-31  0.94  12925.410    8.1
2009-06-30  3.37  12901.504    9.2


2009-09-30  3.56  12990.341    9.6
[203 rows x 3 columns]
```

The first two values passed are the columns to be used respectively as the row and column index, then finally an optional value column to fill the DataFrame. Suppose you had two value columns that you wanted to reshape simultaneously:

```
ldata['value2'] = np.random.randn(len(ldata))
ldata[:10]
```

Output:



```
        date     item     value    value2
0 1959-03-31  realgdp  2710.349  0.523772
1 1959-03-31     infl     0.000  0.000940
2 1959-03-31    unemp     5.800  1.343810
3 1959-06-30  realgdp  2778.801 -0.713544
4 1959-06-30     infl     2.340 -0.831154
5 1959-06-30    unemp     5.100 -2.370232
6 1959-09-30  realgdp  2775.488 -1.860761
7 1959-09-30     infl     2.740 -0.860757
8 1959-09-30    unemp     5.300  0.560145
9 1959-12-31  realgdp  2785.204 -1.265934
```

By omitting the last argument, you obtain a DataFrame with hierarchical columns:

```
pivoted = ldata.pivot('date', 'item')
pivoted[:5]
```

Output:

```
           value                      value2
item        infl   realgdp unemp      infl   realgdp     unemp
date
1959-03-31  0.00  2710.349   5.8  0.000940  0.523772  1.343810
1959-06-30  2.34  2778.801   5.1 -0.831154 -0.713544 -2.370232
1959-09-30  2.74  2775.488   5.3 -0.860757 -1.860761  0.560145
1959-12-31  0.27  2785.204   5.6  0.119827 -1.265934 -1.063512
1960-03-31  2.31  2847.699   5.2 -2.359419  0.332883 -0.199543
```

```
pivoted['value'][:5]
```

Output:

```
item        infl   realgdp  unemp
date
1959-03-31  0.00  2710.349    5.8
1959-06-30  2.34  2778.801    5.1
1959-09-30  2.74  2775.488    5.3
1959-12-31  0.27  2785.204    5.6
1960-03-31  2.31  2847.699    5.2
```

```
unstacked = ldata.set_index(['date', 'item']).unstack('item')
unstacked[:7]
```

Output:

```
           value                      value2
item        infl   realgdp unemp      infl   realgdp     unemp
date
1959-03-31  0.00  2710.349   5.8  0.000940  0.523772  1.343810
1959-06-30  2.34  2778.801   5.1 -0.831154 -0.713544 -2.370232
1959-09-30  2.74  2775.488   5.3 -0.860757 -1.860761  0.560145
1959-12-31  0.27  2785.204   5.6  0.119827 -1.265934 -1.063512
1960-03-31  2.31  2847.699   5.2 -2.359419  0.332883 -0.199543
1960-06-30  0.14  2834.390   5.2 -0.970736 -1.541996 -1.307030
1960-09-30  2.70  2839.022   5.6  0.377984  0.286350 -0.753887
```

c) **Pivoting "Wide" to "Long" Format:** An inverse operation to pivot for DataFrames is ***pandas.melt***. Rather than transforming one column into many in a new DataFrame, it merges multiple columns into one, producing a DataFrame that is longer than the input.

```
df = pd.DataFrame({'key': ['foo', 'bar', 'baz'], 'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]})
df
```

Output:

```
   A  B  C  key
0  1  4  7  foo
1  2  5  8  bar
2  3  6  9  baz
```

The **'key'** column may be a group indicator, and the other columns are data values. When using **pandas.melt**, we must indicate which columns (if any) are group indicators.

```
melted = pd.melt(df, ['key'])
melted
```

Output:

```
   key variable  value
0  foo        A      1
1  bar        A      2
2  baz        A      3
3  foo        B      4
4  bar        B      5
5  baz        B      6
6  foo        C      7
7  bar        C      8
8  baz        C      9
```

Using pivot, we can reshape back to the original layout:

```
reshaped = melted.pivot('key', 'variable', 'value')
reshaped
```

Output:

```
variable  A  B  C
key
bar       2  5  8
baz       3  6  9
foo       1  4  7
```

Since the result of pivot creates an index from the column used as the row labels, we may want to use reset_index to move the data back into a column:

```
reshaped.reset_index()
```

Output:

```
variable  key  A  B  C
0         bar  2  5  8
1         baz  3  6  9
2         foo  1  4  7
```

DATA CLEANING AND PREPARATION

During the course of doing data analysis and modeling, a significant amount of time is spent on data preparation: loading, cleaning, transforming, and rearranging. Such tasks are often reported to take up 80% or more of an analyst's time. Sometimes the way that data is stored in files or databases is not in the right format for a particular task. Pandas, along with the built-in Python language features, provides you with a high-level, flexible, and fast set of tools to enable you to manipulate data into the right form.

1. **Handling Missing Data:** For numeric data, pandas use the floating-point value NaN (Not a Number) to represent missing data. We call this a sentinel value that can be easily detected.

a) Create NULL values

```
string_data = pd.Series(['aardvark', 'artichoke', np.nan, 'avocado'])
string_data
```

Output:

```
0 aardvark
1 artichoke
2 NaN
3 avocado
```

b) Check NULL values

```
string_data.isnull()
```

Output:

```
0 False
1 False
```

```
2 True
3 False
```

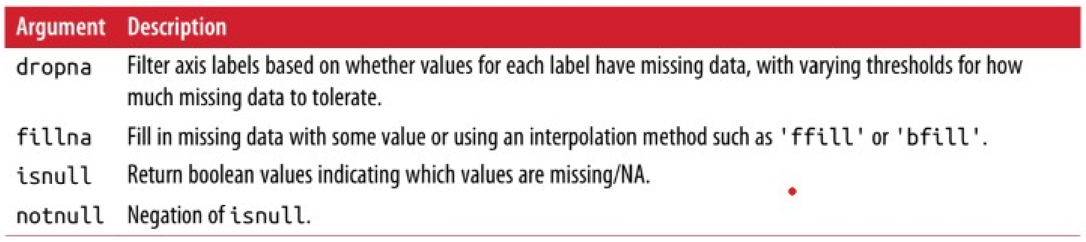
c) NA in Object Arrays

```
string_data[0] = None
string_data.isnull()
```

Output:

```
0    True
1   False
2    True
3   False
```

Table 7-1. NA handling methods



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 boolean values indicating which values are missing/NA.
`notnull`	Negation of `isnull`.

d) Filter out Missing Data

```
from numpy import nan as NA
data = pd.Series([1, NA, 3.5, NA, 7])
data.dropna()
```

Output:

```
0    1.0
2    3.5
4    7.0
```

```
data[data.notnull()]
```

Output:

```
0    1.0
2    3.5
4    7.0
```

With DataFrame objects, things are a bit more complex. You may want to drop rows or columns that are all NA or only those containing any NAs. dropna by default drops any row containing a missing value:

```
data = pd.DataFrame([[1., 6.5, 3.], [1., NA, NA],[NA, NA, NA], [NA, 6.5, 3.]])
cleaned = data.dropna()
data
```

Output:

```
       0     1     2
0    1.0   6.5   3.0
1    1.0   NaN   NaN
2    NaN   NaN   NaN
3    NaN   6.5   3.0
```

```
cleaned
```

Output:

```
     0     1     2
0  1.0   6.5   3.0
```

Passing how='all' will only drop rows that are all NA:

```
data.dropna(how='all')
```

Output:

```
     0      1      2
0   1.0    6.5    3.0
1   1.0    NaN   NaN
3   NaN   6.5    3.0
```

To drop columns in the same way, pass axis=1:

```
data[4] = NA
data
```

Output:

```
        0       1       2       4
0       1.0     6.5     3.0     NaN
1       1.0     NaN     NaN     NaN
2       NaN     NaN     NaN     NaN
3       NaN     6.5     3.0     NaN
```

```
data.dropna(axis=1, how='all')
```

Output:

```
        0       1       2
0       1.0     6.5     3.0
1       1.0     NaN     NaN
2       NaN     NaN     NaN
3       NaN     6.5     3.0
```

A related way to filter out DataFrame rows tends to concern time series data. Suppose you want to keep only rows containing a certain number of observations. You can indicate this with the thresh argument:

```
df = pd.DataFrame(np.random.randn(7, 3))
df.iloc[:4, 1] = NA
df.iloc[:2, 2] = NA
df
```

Output:

```
        0          1          2
0   -0.204708 NaN      NaN
1   -0.555730  NaN      NaN
2   0.092908   NaN      0.769023
3  1.246435    NaN      -1.296221
4  0.274992   0.228913 1.352917
5  0.886429  -2.001637 -0.371843
6  1.669025 -0.438570 -0.539741
```

```
df.dropna()
```

Output:

```
       0         1           2
4 0.274992 0.228913 1.352917
5 0.886429 -2.001637 -0.371843
6 1.669025 -0.438570 -0.539741
```

```
df.dropna(thresh=2)
```

Output:

```
      0           1                 2
2 0.092908     NaN              0.769023
3 1.246435     NaN             -1.296221
4 0.274992    0.228913          1.352917
```

```
5 0.886429   -2.001637       -0.371843
6 1.669025   -0.438570       -0.539741
```

2. Filling In Missing Data:

Calling fillna with a constant replaces missing values with that value:

```
df.fillna(0)
```

Output:

```
          0          1          2
0 -0.204708 0.000000 0.000000
1 -0.555730 0.000000 0.000000
2 0.092908 0.000000 0.769023
3 1.246435 0.000000 -1.296221
4 0.274992 0.228913 1.352917
5 0.886429 -2.001637 -0.371843
6 1.669025 -0.438570 -0.539741
```

Calling fillna with a dict, you can use a different fill value for each column:

```
df.fillna({1: 0.5, 2: 0})
```

Output:

```
          0          1          2
0 -0.204708 0.500000 0.000000
1 -0.555730 0.500000 0.000000
2 0.092908 0.500000 0.769023
3 1.246435 0.500000 -1.296221
4 0.274992 0.228913 1.352917
5 0.886429 -2.001637 -0.371843
6 1.669025 -0.438570 -0.539741
```

fillna returns a new object, but you can modify the existing object in-place:

```
_ = df.fillna(0, inplace=True)
df
```

Output:

```
          0          1          2
0 -0.204708 0.000000 0.000000
1 -0.555730 0.000000 0.000000
2 0.092908 0.000000 0.769023
3 1.246435 0.000000 -1.296221
4 0.274992 0.228913 1.352917
5 0.886429 -2.001637 -0.371843
6 1.669025 -0.438570 -0.539741
```

The same interpolation methods available for reindexing can be used with fillna:

```
df = pd.DataFrame(np.random.randn(6, 3))
df.iloc[2:, 1] = NA
df.iloc[4:, 2] = NA
df
```

Output:

```
      0          1          2
0 0.476985   3.248944 -1.021228
1 -0.577087 0.124121 0.302614
2 0.523772  NaN       1.343810
3 -0.713544 NaN       -2.370232
```

```
4 -1.860761 NaN      NaN
5 -1.265934 NaN     NaN
```

```
                    df.fillna(method='ffill')
```

Output:

```
         0            1           2
0 0.476985     3.248944 -1.021228
1 -0.577087   0.124121 0.302614
2 0.523772    0.124121 1.343810
3 -0.713544   0.124121 -2.370232
4 -1.860761   0.124121 -2.370232
5 -1.265934 0.124121 -2.370232
```

```
                    df.fillna(method='ffill', limit=2)
```

Output:

```
                0       1         2
0 0.476985 3.248944 -1.021228
1 -0.577087 0.124121 0.302614
2 0.523772 0.124121 1.343810
3 -0.713544 0.124121 -2.370232
4 -1.860761 NaN -2.370232
5 -1.265934 NaN -2.370232
```

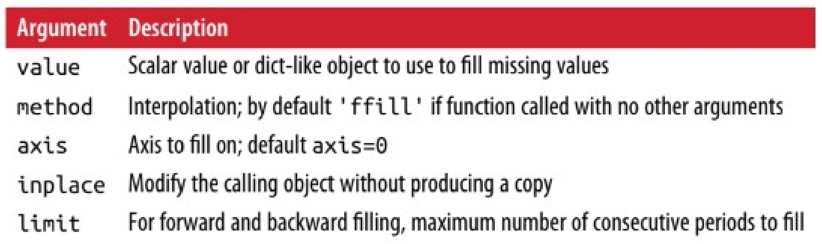
With fillna you can do lots of other things with a little creativity. For example, you might pass the mean or median value of a Series:

```
                    data = pd.Series([1., NA, 3.5, NA, 7])
                    data.fillna(data.mean())
```

Output:

```
0    1.000000
1    3.833333
2    3.500000
3    3.833333
4    7.000000
```

Table 7-2. fillna function arguments



Argument	Description
`value`	Scalar value or dict-like object to use to fill missing values
`method`	Interpolation; by default `'ffill'` if 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, maximum number of consecutive periods to fill

DATA TRANSFORMATION

a) **Removing Duplicates:** Duplicate rows may be found in a DataFrame for any number of reasons.

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

Output:

```
   k1 k2
0 one 1
```

```
1 two 1
2 one 2
3 two 3
4 one 3
5 two 4
6 two 4
```

The DataFrame method duplicated returns a boolean Series indicating whether each row is a duplicate (has been observed in a previous row) or not:

```
data.duplicated()
```

Output:

```
0 False
1 False
2 False
3 False
4 False
5 False
6 True
```

Relatedly, drop_duplicates returns a DataFrame where the duplicated array is False:

```
data.drop_duplicates()
```

Output:

```
  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, you can specify any subset of them to detect duplicates. Suppose we had an additional column of values and wanted to filter duplicates only based on the 'k1' column:

```
data['v1'] = range(7)
data.drop_duplicates(['k1'])
```

Output:

```
    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:

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

Output:

```
    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
```

b) **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. Consider the following hypothetical data collected about various kinds of meat:

```
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
```

Output:
```
 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
```
Suppose you wanted to add a column indicating the type of animal that each food came from. Let's write down a mapping of each distinct meat type to the kind of animal:
```
meat_to_animal = {
'bacon': 'pig',
'pulled pork': 'pig',
'pastrami': 'cow',
'corned beef': 'cow',
'honey ham': 'pig',
'nova lox': 'salmon'
}
```
The map method on a Series accepts a function or dict-like object containing a mapping, but here we have a small problem in that some of the meats are capitalized and others are not. Thus, we need to convert each value to lowercase using the str.lower Series method:
```
        lowercased = data['food'].str.lower()
        lowercased
```
Output:
```
0 bacon
1 pulled pork
2 bacon
3 pastrami
4 corned beef
5 bacon
6 pastrami
7 honey ham
8 nova lox
```

```
      data['animal'] = lowercased.map(meat_to_animal)
      data
```
Output:
```
 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
```
We could also have passed a function that does all the work:
```
        data['food'].map(lambda x: meat_to_animal[x.lower()])
```
Output:

```
0 pig
1 pig
2 pig
3 cow
4 cow
5 pig
6 cow
7 pig
8 salmon
```
Using map is a convenient way to perform element-wise transformations and other data cleaning–related operations.

c) Replacing Values:

```
data = pd.Series([1., -999., 2., -999., -1000., 3.])
data
```
Output:
```
   0 1.0
1 -999.0
2 2.0
3 -999.0
4 -1000.0
5 3.0
```

The -999 values might be sentinel values for missing data. To replace these with NA values that pandas understands, we can use replace, producing a new Series (unless you pass inplace=True):

```
data.replace(-999, np.nan)
```
Output:
```
0 1.0
1 NaN
2 2.0
3 NaN
4 -1000.0
5   3.0
```

d) Renaming Axis Indexes:

```
data = pd.DataFrame(np.arange(12).reshape((3, 4)), index=['Ohio', 'Colorado', 'New York'],
columns=['one', 'two', 'three', 'four'])
```
Like a Series, the axis indexes have a map method:
```
transform = lambda x: x[:4].upper()
data.index.map(transform)
Index(['OHIO', 'COLO', 'NEW '], dtype='object')
```
You can assign to index, modifying the DataFrame in-place:
```
data.index = data.index.map(transform)
data
```
Output:
```
        one   two  three  four
OHIO    0     1      2      3
COLO    4     5      6      7
NEW     8     9     10     11
```
If you want to create a transformed version of a dataset without modifying the original, a useful method is rename:
```
data.rename(index=str.title, columns=str.upper)
```
Output:

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

Notably, rename can be used in conjunction with a dict-like object providing new values for a subset of the axis labels:

```
data.rename(index={'OHIO': 'INDIANA'}, columns={'three': 'peekaboo'})
```

Output:

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

e) Discretization and Binning: Continuous data is often discretized or otherwise separated into "bins" for analysis. Suppose you have data about a group of people in a study, and you want to group them into discrete age buckets:

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

Let's divide these into bins of 18 to 25, 26 to 35, 36 to 60, and finally 61 and older. To do so, you have to use cut, a function in pandas:

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

Output:

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

f) **Detecting and Filtering Outliers:** Filtering or transforming outliers is largely a matter of applying array operations.Consider a DataFrame with some normally distributed data:

```
data = pd.DataFrame(np.random.randn(1000, 4))
data.describe()
```

Output:

```
          0                1                          2         3
count 1000.000000 1000.000000 1000.000000 1000.000000
mean 0.049091 0.026112 -0.002544 -0.051827
std 0.996947 1.007458 0.995232 0.998311
min -3.645860 -3.184377 -3.745356 -3.428254
25% -0.599807 -0.612162 -0.687373 -0.747478
50% 0.047101 -0.013609 -0.022158 -0.088274
75% 0.756646 0.695298 0.699046 0.623331
max 2.653656 3.525865 2.735527 3.366626
```

Suppose you wanted to find values in one of the columns exceeding 3 in absolute value:

```
col = data[2]
col[np.abs(col) > 3]
```

Output:

```
41           -3.399312
136         -3.745356
```

g) **Permutation and Random Sampling:** Permuting (randomly reordering) a Series or the rows in a DataFrame is easy to do using the numpy.random.permutation function. Calling permutation with the length of the axis you want to permute produces an array of integers indicating the new ordering:

```
df = pd.DataFrame(np.arange(5 * 4).reshape((5, 4)))
sampler = np.random.permutation(5)
sampler
```
Output: array([3, 1, 4, 2, 0])

That array can then be used in iloc-based indexing or the equivalent take function:
```
df
```
Output:
```
   0  1  2  3
0  0  1  2  3
1  4  5  6  7
2  8  9 10 11
3 12 13 14 15
4 16 17 18 19
```
```
df.take(sampler)
```
Output:
```
    0  1  2  3
3 12 13 14 15
1  4  5  6  7
4 16 17 18 19
2   8  9 10 11
0   0  1  2  3
```

h) **Computing Indicator/Dummy Variables:** Another type of transformation for statistical modeling or machine learning applications is converting a categorical variable into a "dummy" or "indicator" matrix. If a column in a DataFrame has k distinct values, you would derive a matrix or Data-Frame with k columns containing all 1s and 0s. pandas has a get_dummies function for doing this, though devising one yourself is not difficult.
```
df = pd.DataFrame({'key': ['b', 'b', 'a', 'c', 'a', 'b'], 'data1': range(6)})
pd.get_dummies(df['key'])
```
Output:
```
  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
```

STRING MANIPULATION

Python has long been a popular raw data manipulation language in part due to its ease of use for string and text processing. Most text operations are made simple with the string object's built-in methods. For more complex pattern matching and text manipulations, regular expressions may be needed. pandas add to the mix by enabling you to apply string and regular expressions concisely on whole arrays of data, additionally handling the annoyance of missing data.

a) String Object Methods

In many string munging and scripting applications, built-in string methods are sufficient. As an example, a comma-separated string can be broken into pieces with split:
```
val = 'a,b, guido'
val.split(',')
```
Output: ['a', 'b', ' guido']

split is often combined with *strip* to trim whitespace (including line breaks):

```
                pieces = [x.strip() for x in val.split(',')]
                pieces
```
Output: ['a', 'b', 'guido']

These substrings could be concatenated together with a two-colon delimiter using addition:
```
                first, second, third = pieces
                first + '::' + second + '::' + third
```
Output: 'a::b::guido'

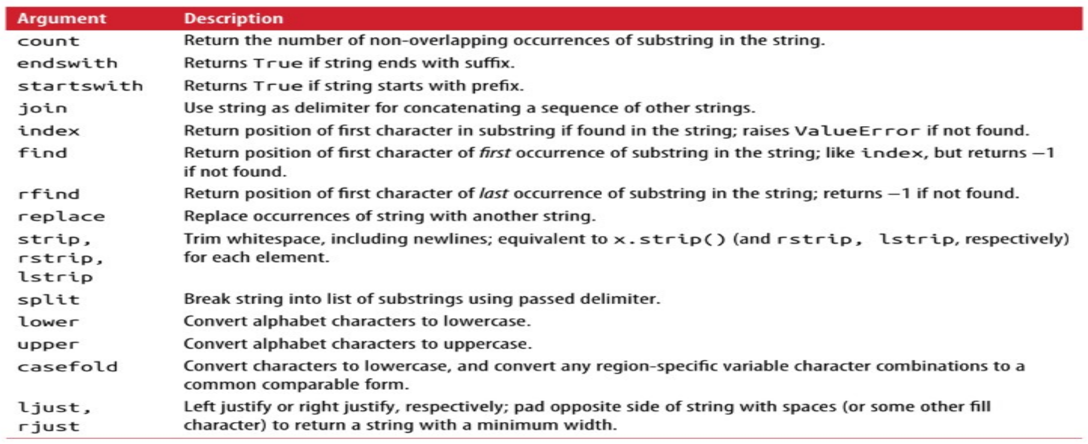
But this isn’t a practical generic method. A faster and more Pythonic way is to pass a list or tuple to the join method on the string '::':
```
                '::'.join(pieces)
                'a::b::guido'
```
Other methods are concerned with locating substrings. Using Python’s in keyword is the best way to detect a substring, though index and find can also be used:
```
                'guido' in val
```
Output: True
```
                val.index(',')
```
Output: 1
```
                val.find(':')
```
Output: -1

Note the difference between find and index is that index raises an exception if the string isn’t found (versus returning –1):
```
                val.index(':')
```
Output:
```
---------------------------------------------------------------------------
ValueError Traceback (most recent call last)
<ipython-input-144-280f8b2856ce> in <module>()
----> 1 val.index(':')
ValueError: substring not found
```

Relatedly, count returns the number of occurrences of a particular substring:
```
                val.count(',')
```
Output: 2

replace will substitute occurrences of one pattern for another. It is commonly used to delete patterns, too, by passing an empty string:
```
                val.replace(',', '::')
```
Output: 'a::b:: guido'
```
                val.replace(',', '')
```
Output: 'ab guido'

Table 7-3. Python built-in string methods



Argument	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.
`replace`	Replace occurrences of string with another string.
`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.

b) Regular Expressions: Regular expressions provide a flexible way to search or match (often more complex) string patterns in text. A single expression, commonly called a ***regex***, is a string formed according to the regular expression language. Python's built-in **re** module is responsible for applying ***regular expressions to strings***. The re-module functions fall into three categories**: pattern matching, substitution, and splitting.**

```
import re
text = "foo bar\t baz \tqux"
re.split('\s+', text)
```
Output: ['foo', 'bar', 'baz', 'qux']

```
regex = re.compile('\s+')
regex.split(text)
```
Output: ['foo', 'bar', 'baz', 'qux']

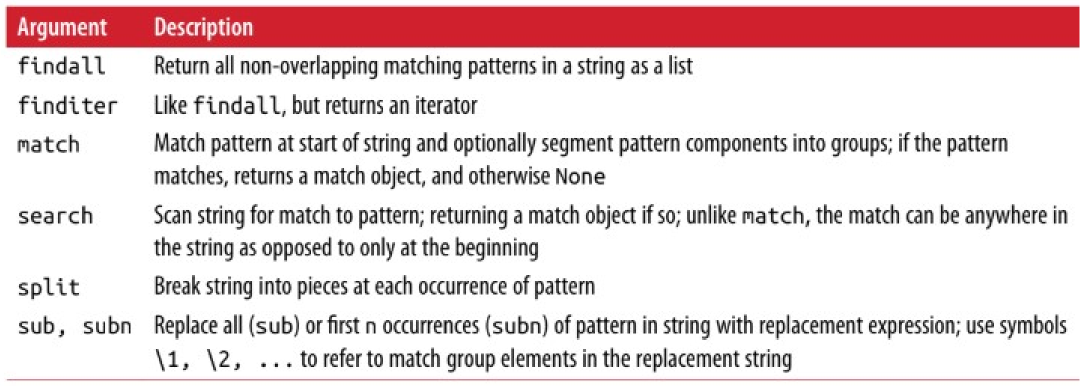
```
regex.findall(text)
```
Output: [' ', '\t ', ' \t']

```
text = """Dave dave@google.com
Steve steve@gmail.com
Rob rob@gmail.com
Ryan ryan@yahoo.com
"""
pattern = r'[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}'
regex = re.compile(pattern, flags=re.IGNORECASE)
regex.findall(text)
```
Output:
```
['dave@google.com',
'steve@gmail.com',
'rob@gmail.com',
'ryan@yahoo.com']
```

```
m = regex.search(text)
m
```
Output:<_sre.SRE_Match object; span=(5, 20), match='dave@google.com'>

```
print(regex.match(text))
```
Output: None

Table 7-4. Regular expression methods



Argument	Description
`findall`	Return all non-overlapping matching patterns in a string as a list
`finditer`	Like `findall`, but returns an iterator
`match`	Match pattern at start of string and optionally segment pattern components into groups; if the pattern matches, returns a match object, and otherwise `None`
`search`	Scan string for match to pattern; returning a match object if so; unlike `match`, the match can be anywhere in the string as opposed to only at the beginning
`split`	Break string into pieces at each occurrence of pattern
`sub, subn`	Replace all (`sub`) or first `n` occurrences (`subn`) of pattern in string with replacement expression; use symbols `\1, \2, ...` to refer to match group elements in the replacement string

c) Vectorized String Functions in pandas: Cleaning up a messy dataset for analysis often requires a lot of string munging and regularization. To complicate matters, a column containing strings will sometimes have missing data:

```
data = {'Dave': 'dave@google.com', 'Steve': 'steve@gmail.com',  'Rob': 'rob@gmail.com', 'Wes': np.nan}
data = pd.Series(data)
data
```

Output:

```
Dave     dave@google.com
Rob      rob@gmail.com
Steve    steve@gmail.com
Wes      NaN
```

```
         data.isnull()
```

Output:

```
Dave     False
Rob      False
Steve    False
Wes      True
```

You can apply string and regular expression methods can be applied (passing a lambda or other function) to each value using ***data.map***, but it will fail on the NA (null) values. To cope with this, Series has array-oriented methods for string operations that skip NA values. These are accessed through Series's str attribute; for example, we could check whether each email address has **'gmail'** in it with **str.contains**:

```
                 data.str.contains('gmail')
```

Output:

```
Dave     False
Rob      True
Steve    True
Wes      NaN
```

Regular expressions can be used, too, along with any re options like IGNORECASE:

```
                 pattern
```

Output: '([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\\.([A-Z]{2,4})'

```
                 data.str.findall(pattern, flags=re.IGNORECASE)
```

Output:

```
Dave [(dave, google, com)]
```

```
Rob [(rob, gmail, com)]
Steve [(steve, gmail, com)]
Wes NaN
```

There are a couple of ways to do vectorized element retrieval. Either use **str.get** or **index** into the str attribute:

```
matches = data.str.match(pattern, flags=re.IGNORECASE)
matches
```

Output:

```
Dave    True
Rob     True
Steve   True
Wes     NaN
```

To access elements in the embedded lists, we can pass an index to either of these functions:

```
matches.str.get(1)
```

Output:

```
Dave    NaN
Rob     NaN
Steve   NaN
Wes     NaN
```

```
matches.str[0]
```

Output:

```
Dave    NaN
Rob     NaN
Steve   NaN
Wes     NaN
```

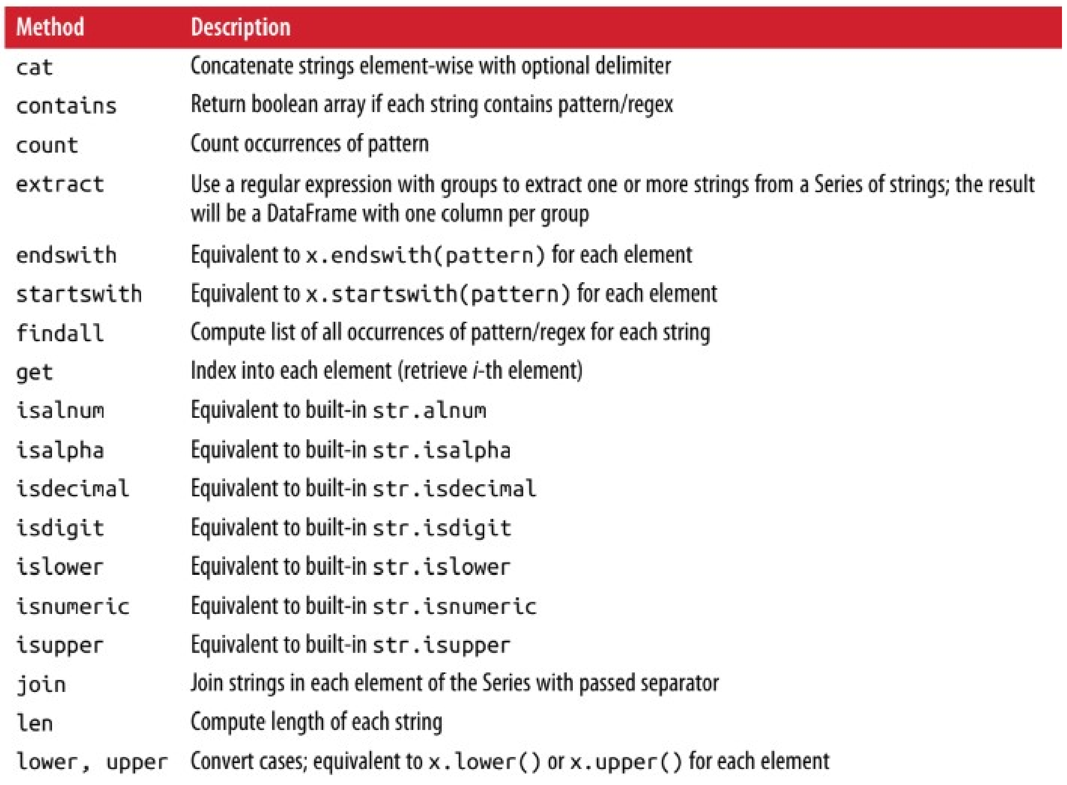
You can similarly slice strings using this syntax:

```
data.str[:5]
```

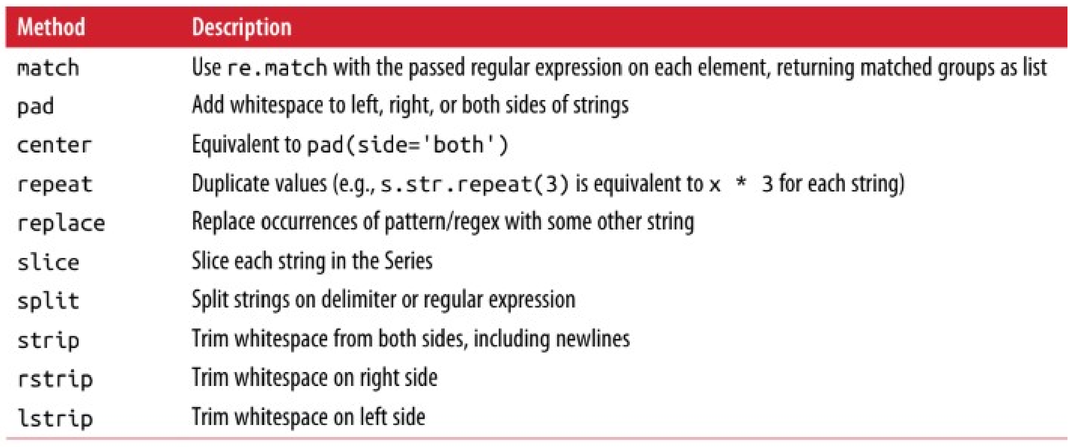
Output:

```
Dave    dave@
Rob     rob@g
Steve   steve
Wes     NaN
```

Table 7-5. Partial listing of vectorized string methods



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.endswith(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)
`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



Method	Description
`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

SUMMARIZING

```
import pandas as pd
import numpy as np

# Creating a DataFrame with various types of data
data = {
'Date': pd.date_range(start='2024-01-01', periods=7),
'Temperature': [78, 85, 74, 84, 79, 73, 77],
'Sales': [234, 190, 302, 280, 310, 215, 275],
'CustomerSatisfaction': [4.5, 3.8, 4.2, 4.0, 5.0, 3.5, 4.1]
}
```

```
df = pd.DataFrame(data)
df.head()
```

Output:

```
     Date Temperature     Sales      Customer      Satisfaction
0         2024-01-01          78           234              4.5
1         2024-01-02          85           190              3.8
2         2024-01-03         74            302              4.2
3         2024-01-04         84            280              4.0
4         2024-01-05         79            310              5.0
5         2024-01-06         73            215              3.5
6         2024-01-07         77             275             4.1
```

BINNING

Binning data is an essential technique in data analysis that enables the transformation of continuous data into discrete intervals, providing a clearer picture of the underlying trends and distributions. In the Python ecosystem, the combination of numpy and scipy libraries offers robust tools for effective data binning.

Why Binning Data is Important?

Binning data is a critical step in data preprocessing that holds significant importance across various analytical domains. By grouping continuous numerical values into discrete bins or intervals, binning simplifies complex datasets, making them more interpretable and accessible.

- Binning captures non-linear patterns, improving understanding of variable relationships.
- It's effective for handling outliers by aggregating extreme values, preventing undue influence on analyses or models.
- Addresses challenges with skewed distributions, aids statistical tests on categorical assumptions.
- Useful where data deviates from normal, providing balanced representation in each bin.

Binning Data using Numpy

Binning data is a common technique in data analysis where you group continuous data into discrete intervals, or bins, to gain insights into the distribution or trends within the data.

1. **Equal Width Binning**
   Bin data into equal-width intervals using numpy's histogram function. This approach divides the data into a specified number of bins (num_bins) of equal width.

Example:

```
import numpy as np
data = np.random.rand(100)
num_bins = 10
hist, bins = np.histogram(data, bins=num_bins)
print("Bin Edges: ", bins)
print("Histogram Counts: ", hist)
```

Output:

```
Bin Edges: [0.01337762 0.11171836 0.21005911 0.30839985 0.4067406 0.50508135
0.60342209 0.70176284 0.80010358 0.89844433 0.99678508]
Histogram Counts: [10 14 10 12 9 8 7 10 11 9]
```

Bin Edges, are the boundaries that define the intervals (bins) into which the data is divided. Each bin includes values up to, but not including, the next bin edge. Histogram Counts are the frequencies or counts of data points that fall within each bin. For example, in the first bin [0.01337762, 0.11171836), there are 10 data points. In the second bin [0.11171836, 0.21005911), there are 14 data points, and so on.

Set our own Bin Edges

- The **numpy.linspace** function creates evenly spaced bin edges, resulting in bins of equal width.
- The **numpy.digitize** function is then used to assign data points to their respective bins based on these equal-width intervals.

Example:

```
import numpy as np
data = np.random.rand(100)
bin_edges = np.linspace(0, 1, 6)
bin_indices = np.digitize(data, bin_edges)

hist = np.bincount(bin_indices)
print("Bin Edges: ", bin_edges)
print("Histogram Counts: ", hist)
```

Output:

```
Bin Edges: [0. 0.2 0.4 0.6 0.8 1. ]
Histogram Counts: [ 0 18 13 24 24 21]
```

Set Custom Binning Intervals with Numpy

Bin data into custom intervals using numpy's **np.histogram** function. Here, we define custom bin edges (bin_edges) to group the data points according to specific intervals.

Example:

```
import numpy as np
data = np.random.rand(100)
bin_edges = [0, 0.2, 0.4, 0.6, 0.8, 1.0]

hist, bins = np.histogram(data, bins=bin_edges)
print("Bin Edges: ", bins)
print("Histogram Counts: ", hist)
```

Output:

```
Bin Edges: [0. 0.2 0.4 0.6 0.8 1. ]
Histogram Counts: [27 20 15 19 19]
```

2. Binning Categorical Data with Numpy

Count occurrences of categories using numpy's unique function. When dealing with categorical data, this approach counts occurrences of each unique category. The code example generates example categorical data and then uses NumPy's unique function to find the unique categories and their corresponding counts in the dataset. This array contains the unique categories present in the categories array. In this case, the unique categories are 'A', 'B', 'C', and 'D'. counts array,contains the corresponding counts for each unique category.

Example:

```
import numpy as np
categories = np.random.choice(['A', 'B', 'C', 'D'], size=100)
unique_categories, counts = np.unique(categories, return_counts=True)
print("Unique Categories: ", unique_categories)
print("Category Counts: ", counts)
```

Output:

```
Unique Categories: ['A' 'B' 'C' 'D']
Category Counts: [29 16 25 30]
```

3. **Binned Mean with Scipy**

Calculate the mean within each bin using scipy's binned_statistic function. This approach demonstrates how to use binned_statistic to calculate the mean of data points within specified bins.

Example:

```
import random
import statistics
from scipy.stats import binned_statistic
data = [random.random() for _ in range(100)]
num_bins = 10

result = binned_statistic(data, data, bins=num_bins, statistic='mean')
bin_edges = result.bin_edges
bin_means = result.statistic
print("Bin Edges: ", bin_edges)
print("Binned Mean: ", bin_means)
```

Output:

```
Bin Edges: [0.0337853 0.12594314 0.21810098 0.31025882 0.40241666 0.4945745
0.58673234 0.67889019 0.77104803 0.86320587 0.95536371]
Binned Mean: [0.07024781 0.15714129 0.26879363 0.36394539 0.44062907 0.54527985
0.63046277 0.72201578 0.84474723 0.91074019]
```

4. **Binned Sum with Scipy**

Calculate the sum within each bin using scipy's binned_statistic function. Similar to the mean Approach, this calculates the sum within each bin, providing a different perspective on aggregating data.

Example:

```
from scipy.stats import binned_statistic
data = np.random.rand(100)
num_bins = 10
result = binned_statistic(data, data, bins=num_bins, statistic='sum')
print("Bin Edges: ", result.bin_edges)
print("Binned Sum: ", result.statistic)
```

Output:

```
Bin Edges: [0.00222855 0.1014526 0.20067665 0.29990071 0.39912476 0.49834881
0.59757286 0.69679692 0.79602097 0.89524502 0.99446907]
Binned Sum: [ 0.60435816 1.60018494 2.47764912 3.49905238 2.73274596 6.07700391
3.15241481 8.89573616 7.75076402 11.36858964]
```

Binned Quantiles with Scipy

Calculate quantiles (75th percentile) within each bin using scipy's binned_statistic function. This demonstrates how to calculate a specific quantile (75th percentile) within each bin, useful for analyzing the spread of data.

Example:

```
from scipy.stats import binned_statistic
data = np.random.randn(1000)
num_bins = 20
result = binned_statistic(data, data, bins=num_bins, statistic=lambda x: np.percentile(x, q=75))
print("Bin Edges: " result.bin_edges)
print("75th Percentile within Each Bin: ", result.statistic)
```

Output:
```
Bin Edges: [-3.8162536 -3.46986707 -3.12348054 -2.777094 -2.43070747 -2.08432094
-1.73793441 -1.39154788 -1.04516135 -0.69877482 -0.35238828 -0.00600175
0.34038478 0.68677131 1.03315784 1.37954437 1.72593091 2.07231744
2.41870397 2.7650905 3.11147703]
75th Percentile within Each Bin: [-3.8162536 nan nan -2.53157311 -2.14902013 -1.82057818
-1.43829609 -1.10931775 -0.76699539 -0.43874444 -0.09672504 0.25824355
0.61470027 0.95566003 1.27059392 1.58331292 1.98752497 2.34089378
2.55623431 3.07407641]
```

CLASSING AND STANDARDIZATION

In Python, "classing" refers to creating classes, which are blueprints for creating objects, while "standardization" in the context of data science involves transforming data to have a mean of 0 and a standard deviation of 1, often using the StandardScaler from scikit-learn.

1. Classing (Creating Classes):

What it is: In Python, a class is a user-defined blueprint or template for creating objects, allowing you to group related data and functions (methods) together.

Example:
```
class Dog: # Define a class named Dog
def __init__(self, name, breed): #Constructor
self.name = name # Instance attribute
self.breed = breed # Instance attribute
def bark(self): # Method
print(f"{self.name} says Woof!")

my_dog = Dog("Buddy", "Golden Retriever") # Create an object (instance) of the Dog class
print(my_dog.name) # Access the instance attribute
my_dog.bark() # Call the method
```

2. Standardization (Data Transformation):

Standardization, also known as Z-score normalization, is a data preprocessing technique used to transform data so that it has a mean of 0 and a standard deviation of 1.

Formula: $z = (x - \mu) / \sigma$ where:
z is the standardized value
x is the original value
μ is the mean of the original data
σ is the standard deviation of the original data

Using StandardScaler (scikit-learn):
```
import pandas as pd
from sklearn.preprocessing import StandardScaler
data = {'feature1': [10, 20, 30, 40, 50], 'feature2': [1, 2, 3, 4, 5]}
df = pd.DataFrame(data)
scaler = StandardScaler()
scaler.fit(df)
standardized_data = scaler.transform(df)
print(standardized_data)
```

OUTLIER/NOISE & ANOMALIES

In Python, identifying and handling outliers and anomalies (often used interchangeably) involves using various techniques and libraries to detect data points that deviate significantly from the norm.

Understanding Outliers and Anomalies

- **Outliers**: Data points that deviate significantly from the majority of the data.
- **Anomalies**: Similar to outliers, they are rare events that deviate significantly from expected behavior.
- **Noise**: Random errors or variations in the data that don't represent meaningful patterns.

Python Libraries for Outlier/Anomaly Detection

Scikit-learn: Offers various algorithms for outlier detection, including LocalOutlierFactor, IsolationForest, and OneClassSVM.
PyOD (Python Outlier Detection): A dedicated library for outlier detection, providing a wide range of algorithms.
NumPy and Pandas: Used for data manipulation, analysis, and visualization.
Matplotlib and Seaborn: Used for data visualization to identify potential outliers.

Common Techniques for Outlier/Anomaly Detection

Statistical Methods:

- *Z-score*: Measures how many standard deviations a data point is from the mean.
- *Interquartile Range (IQR):* Identifies outliers based on the spread of the middle 50% of the data.

Machine Learning Methods:

- *Clustering*: Algorithms like DBSCAN can identify outliers as points that don't belong to any cluster.
- *Isolation Forest*: An ensemble method that isolates outliers by randomly partitioning the data.
- *One-Class SVM*: A machine learning model that learns the characteristics of normal data and flags outliers as deviations.
- *Local Outlier Factor (LOF)*: Measures the local density deviation of a given point with respect to its neighbors.

Time Series Anomaly Detection:

- *Statistical Methods:* Z-score, moving average, and other statistical methods can be used to detect anomalies in time series data.
- *Machine Learning Methods*: Algorithms like ARIMA, LSTM, and autoencoders can be used to model time series data and detect anomalies

Example:

```
from sklearn.neighbors import LocalOutlierFactor
import numpy as np
import pandas as pd
data = np.array([[1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6], [99, 2], [95, 95]])
clf = LocalOutlierFactor(n_neighbors=2)
y_pred = clf.fit_predict(data)
outliers = data[y_pred == -1]
print(outliers)
from pyod.models.iforest import IForest
import numpy as np
data = np.array([[1, 2], [1.5, 1.8], [5, 8], [8, 8], [1, 0.6], [99, 2], [95, 95]])
clf = IForest(n_estimators=100)
clf.fit(data)
y_pred = clf.decision_function(data)
outliers = data[y_pred < -0.5]
print(outliers)
```