APSC-5984 Lab 5: Dataframe manipulation

Due: 2023-02-20 (Monday) 23:59:59

0. Overview

We will introduce the concept of `DataFrame` in this lab. You will be intstructed to use the Python library `pandas` to manipulate dataframes. First, let's import the library. Conventionally, we import it as `pd`.

```
import pandas as pd
```

1. Data Loading and Saving

We will work on the several files in the `lab_05` folder to practice how to load and save files

1.1 CSV and tab-delimited files

1.1.1 Separators

The basic function to load data in `pandas` is `pd.read_csv()`. It can read data from a CSV file or a tab-delimited file. The default delimiter is comma "`,`", but it also allows you to specify other delimiters, such as tab "`\t`".

The file `file_A.csv` is a CSV file with comma as the delimiter:

```
!cat file_A.csv
```

```
id,A,B,C
a1,1,1,1
a2,0,1,0
a3,1,0,1
```

```
pd.read_csv('file_A.csv')
```

```
   id  A  B  C
0  a1  1  1  1
1  a2  0  1  0
2  a3  1  0  1
```

The file `file_A.csv` was correctly loaded into Python. The dataframe has 3 rows and 4 columns. What if we use the same way to load the file `file_B.txt` that is tab-delimited?

```
!cat file_B.txt
```

```
id  A   B   C
a1  1   1   1
a2  0   1   0
a3  1   0   1
```

```
pd.read_csv('file_B.txt')
```

```
         id\tA\tB\tC
0        a1\t1\t1\t1
1        a2\t0\t1\t0
2  a3\t1\t0\t1\t\t\t
```

The result was not what we expected. The reason is that the default delimiter is comma, but the file is tab-delimited. We can specify the delimiter as tab "`\t`" to fix the problem.

```
pd.read_csv('file_B.txt', sep='\t')
```

```
   id  A  B  C
0  a1  1  1  1
1  a2  0  1  0
2  a3  1  0  1
```

Great! Noted that `sep` can be any character, such as "`|`", "`;`", "`:`", etc. So, always check the delimiter before loading the file.

1.1.2 Header

In some cases, the first row of the file is not the header. We can use the argument `header` to specify the row number of the header.

This example shows what would happen if we do not specify the header wiht a non-header file `file_A_nh.csv`.

```
!cat file_A_nh.csv
```

```
a1,1,1,1
a2,0,1,0
a3,1,0,1
```

```
pd.read_csv('file_A_nh.csv')
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	a1	1	1.1	1.2
0	a2	0	1	0
1	a3	1	0	1

The first row was loaded as the header. Here is the fix.

```
pd.read_csv('file_A_nh.csv', header=None)
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	0	1	2	3
0	a1	1	1	1
1	a2	0	1	0
2	a3	1	0	1

Some files may be coded with two headers:

```
!cat file_A_2h.csv
```

```
id,A,B,C
a1,1,1,1
a2,0,1,0
a3,1,0,1
id,D,E,F
a4,1,1,1
a5,0,1,0
a6,1,0,1
```

If we want the 5th row to be the header, we can use `header=4` (again, it is 0-based).

```
pd.read_csv('file_A_2h.csv', header=4)
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	id	D	E	F
0	a4	1	1	1
1	a5	0	1	0
2	a6	1	0	1

1.2 Excel spreadsheet (.xlsx)

Excel spreadsheet is a common format for data storage. However, given it is a format that contains multiple sheets, it is not straightforward to load it into a tabular format.

1.2.1 Load a single sheet

Here is an example of using `pd.read_excel()` to load the spreadsheet `file_C.xlsx`:

```
pd.read_excel('file_C.xlsx')
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	id	A	B	C
0	a1	1	1	1
1	a2	0	1	0
2	a3	1	0	1

By default, it only loads the first sheet. We can specify the sheet name or the sheet number to load other sheets.

```
pd.read_excel('file_C.xlsx', sheet_name='Sheet2')
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	id	D	E	F
0	a4	0	1	0
1	a5	0	0	0
2	a6	1	1	0
3	a7	2	2	0
4	a8	3	3	0
5	a9	4	4	0

```
pd.read_excel('file_C.xlsx', sheet_name='Sheet3')
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	A	B
0	0.631007	0.034287
1	0.114071	0.370723
2	0.156949	0.851093
3	0.051913	0.089328
4	0.089216	0.861941

	A	B
5	0.572473	0.364972
6	0.452546	0.152391
7	0.052752	0.024641

1.2.2 Dictionary of dataframes

In `pandas`, Excel spreadsheet is loaded as a dictionary of dataframes. The keys are the sheet names, and the values are the dataframes.

To load the entire spreadsheet taht contains all sheets, we can use `pd.read_excel()` with `sheet_name=None`:

```
data = pd.read_excel('file_C.xlsx', sheet_name=None)
print(data)
```

```
{'Sheet1':    id  A  B  C
0  a1  1  1  1
1  a2  0  1  0
2  a3  1  0  1, 'Sheet2':    id  D  E  F
0  a4  0  1  0
1  a5  0  0  0
2  a6  1  1  0
3  a7  2  2  0
4  a8  3  3  0
5  a9  4  4  0, 'Sheet3':          A         B
0  0.631007  0.034287
1  0.114071  0.370723
2  0.156949  0.851093
3  0.051913  0.089328
4  0.089216  0.861941
5  0.572473  0.364972
6  0.452546  0.152391
7  0.052752  0.024641}
```

The sheets might not be displayed well aligned, but you can still see the keys as each sheet name and its corresponding dataframe. You can use the 'lookup' function we learned in the previous lecture to find the dataframe of a specific sheet:

```
data["Sheet3"]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	A	B
0	0.631007	0.034287
1	0.114071	0.370723
2	0.156949	0.851093
3	0.051913	0.089328
4	0.089216	0.861941
5	0.572473	0.364972
6	0.452546	0.152391
7	0.052752	0.024641

1.3 Save data

1.3.1 Save as CSV

We can use `df.to_csv()` to save a dataframe as a CSV file. Here are parameters that we can use:

- `sep`: the delimiter. Default is comma "`,`".
- `index`: whether to save the index column. Default is `True`.
- `header`: whether to save the header. Default is `True`.
- `columns`: the columns to save. Default is `None` (all columns).
- `mode`: the mode to open the file. Default is `"w"` (write). Other options are `"a"` (append) and `"r"` (read).

```
data["Sheet1"].to_csv('out_A.csv')
!cat out_A.csv
```

```
,id,A,B,C
0,a1,1,1,1
1,a2,0,1,0
2,a3,1,0,1
```

```
data["Sheet1"].to_csv('out_A.csv', index=False)
!cat out_A.csv
```

```
id,A,B,C
a1,1,1,1
a2,0,1,0
a3,1,0,1
```

```
data["Sheet1"].to_csv('out_A.csv', index=False, header=None)
!cat out_A.csv
```

```
a1,1,1,1
a2,0,1,0
a3,1,0,1
```

```
data["Sheet1"].to_csv('out_A.csv', index=False, header=None, sep='\t')
!cat out_A.csv
```

```
a1	1	1	1
a2	0	1	0
a3	1	0	1
```

```
data["Sheet1"].to_csv('out_A.csv', index=False, columns=['A', 'B'])
!cat out_A.csv
```

```
A,B
1,1
0,1
1,0
```

1.3.2 Save as Excel spreadsheet

Pandas also allows us to save a dataframe as an Excel spreadsheet. It is highly recommended to interact with Excel spreadsheet using `with` statement when you want to work with multiple sheets. Here is an example:

```
with pd.ExcelWriter('out_C2.xlsx') as writer:
    data["Sheet1"].to_excel(writer, sheet_name='Sheet1')
    data["Sheet2"].to_excel(writer, sheet_name='Sheet2')
    data["Sheet3"].to_excel(writer, sheet_name='Sheet3')
```

An example to append a new sheet to an existing spreadsheet:

```
with pd.ExcelWriter('out_C2.xlsx', mode="a") as writer:
    data["Sheet1"].to_excel(writer, sheet_name='Sheet4', index=False)
    data["Sheet2"].to_excel(writer, sheet_name='Sheet5', index=False)
    data["Sheet3"].to_excel(writer, sheet_name='Sheet6', index=False)
```

2. Construct a dataframe

We can also construct a dataframe from scratch. We can start with a dictionary of lists to define our dataframe:

```
data = dict()
data["id"] = ["id1", "id2", "id3", "id4"]
data["factor"] = ["A", "B", "A", "B"]
data["value"] = [1, 2, 3, 4]
print(data)
```

```
{'id': ['id1', 'id2', 'id3', 'id4'], 'factor': ['A', 'B', 'A', 'B'],
'value': [1, 2, 3, 4]}
```

And we can put the dictionary into a dataframe using `pd.DataFrame()`:

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

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	id	factor	value
0	id1	A	1
1	id2	B	2
2	id3	A	3
3	id4	B	4

3. Dataframe manipulation

3.1 Index location (.iloc)

We can use `.iloc()` method to access the data by numeric index location. The indexing rule is the same as what we have learned in the sections of `list` and `numpy`. In `.iloc()`, the first argument is the row index, and the second argument is the column index.

Here is an example dataframe:

```
data = pd.read_excel('file_C.xlsx', sheet_name="Sheet1")
data
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	id	A	B	C
0	a1	1	1	1
1	a2	0	1	0
2	a3	1	0	1

Get the second and third row:

```
data.iloc[[1, 2], :]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}
```

```
.dataframe thead th {
    text-align: right;
}
```

	id	A	B	C
1	a2	0	1	0
2	a3	1	0	1

Get multiple (first and second) columns. (Note we use `:` to specify all rows.)

```
data.iloc[:, [0, 1]]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	id	A
0	a1	1
1	a2	0
2	a3	1

It is equivalent to using slicing:

```
data.iloc[:, :2]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	id	A
0	a1	1
1	a2	0
2	a3	1

3.2 Label-based indexing (.loc)

The `.loc()` method is another way to access the data. It works with either column/index names or boolean arrays.

```
data.loc[[0, 1], :]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	id	A	B	C
0	a1	1	1	1
1	a2	0	1	0

```
data.loc[:, ['id', 'B']]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	id	B
0	a1	1
1	a2	1

	id	B
2	a3	0

Use boolean to select column containing a letter "B". (We can use `df.columns` to list all column names)

```
colnames = data.columns
bol_B = ["B" in col for col in colnames]
print(bol_B)
```

```
[False, False, True, False]
```

```
data.loc[:, bol_B]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	B
0	1
1	1
2	0

```
data
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	id	A	B	C
0	a1	1	1	1
1	a2	0	1	0
2	a3	1	0	1

3.3 Create a new column

The `.loc()` method is also a recommended way (compared to `df["new_column"]`) to create a new column. Simply put a desired column name in the second argument, and assign a value to it.

```
data.loc[:, "new_col"] = ["new"] * 3
# or
data.loc[:, "new_col"] = "new"
data
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	id	A	B	C	new_col
0	a1	1	1	1	new
1	a2	0	1	0	new
2	a3	1	0	1	new

3.4 Miscellaneous

3.4.1 Drop a column

```
data.drop(columns=["B"])
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
```

```
    text-align: right;
}
```

	id	A	C	new_col
0	a1	1	1	new
1	a2	0	0	new
2	a3	1	1	new

3.4.2 Drop a row

```
data.drop(index=[0, 1])
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	id	A	B	C	new_col
2	a3	1	0	1	new

3.4.3 inspect the dimension and summary

`df.shape` returns the dimension of the dataframe. This tells us that the dataframe has 3 rows and 5 columns.

```
data.shape
```

```
(3, 5)
```

df.info() is another way to inspect the dataframe of its dimension and data types of each column.

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 5 columns):
 #   Column   Non-Null Count  Dtype 
---  ------   --------------  ----- 
 0   id       3 non-null      object
 1   A        3 non-null      int64 
 2   B        3 non-null      int64 
 3   C        3 non-null      int64 
 4   new_col  3 non-null      object
dtypes: int64(3), object(2)
memory usage: 248.0+ bytes
```

`df.describe()` returns the summary statistics of the dataframe. Only numeric columns are included in the summary statistics.

```
data.describe()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	A	B	C
count	3.000000	3.000000	3.000000
mean	0.666667	0.666667	0.666667
std	0.577350	0.577350	0.577350
min	0.000000	0.000000	0.000000
25%	0.500000	0.500000	0.500000
50%	1.000000	1.000000	1.000000
75%	1.000000	1.000000	1.000000
max	1.000000	1.000000	1.000000

`df["column"].value_counts()` returns the counts of unique values in that specified column. Below the example tells us that there are two rows with value 1 and one row with value 0.

```
data["B"].value_counts()
```

```
1    2
0    1
Name: B, dtype: int64
```

4.Querying with an example dataframe

Let's create a mock dataframe for this section:

```
import numpy as np
import pandas as pd

factors = [i for _ in range(30) for i in ["A", "B", "C", "D"]]
# random sample from id {1, 2, 3, 4, 5, 6}
ids = np.random.choice(["id_%d" % (i + 1) for i in range(6)], 120)
envs = [i for _ in range(60) for i in ["env_1", "env_2"]]
obs = np.random.normal(0, 1, 120)
data = pd.DataFrame({"factor": factors, "id": ids, "env": envs, "obs":
obs})
data.to_csv("file_D.csv", index=False)
```

```
data = pd.read_csv("file_D.csv")
data.info()
data
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 120 entries, 0 to 119
Data columns (total 4 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   factor  120 non-null    object 
 1   id      120 non-null    object 
 2   env     120 non-null    object 
 3   obs     120 non-null    float64
dtypes: float64(1), object(3)
memory usage: 3.9+ KB
```

```
.dataframe tbody tr th {
    vertical-align: top;
```

```
}

.dataframe thead th {
    text-align: right;
}
```

	factor	id	env	obs
0	A	id_4	env_1	-0.096375
1	B	id_6	env_2	-0.501217
2	C	id_2	env_1	0.653886
3	D	id_1	env_2	0.592581
4	A	id_2	env_1	-0.573104
...	...	...	...	...
115	D	id_3	env_2	-0.144350
116	A	id_1	env_1	1.098145
117	B	id_2	env_2	1.206627
118	C	id_2	env_1	-0.565825
119	D	id_5	env_2	0.837069

120 rows × 4 columns

4.1 Check the distribution of each column

```
data["factor"].value_counts()
```

```
A    30
B    30
C    30
D    30
Name: factor, dtype: int64
```

```
data["id"].value_counts()
```

```
id_2    27
id_1    22
```

```
id_4    20
id_5    19
id_6    16
id_3    16
Name: id, dtype: int64
```

```
data["env"].value_counts()
```

```
env_1    60
env_2    60
Name: env, dtype: int64
```

```
data["obs"].value_counts()
```

```
-0.096375    1
-0.501217    1
-1.738500    1
 0.885649    1
 0.046750    1
            ..
-2.412101    1
-0.225374    1
-0.001173    1
-1.442376    1
 0.837069    1
Name: obs, Length: 120, dtype: int64
```

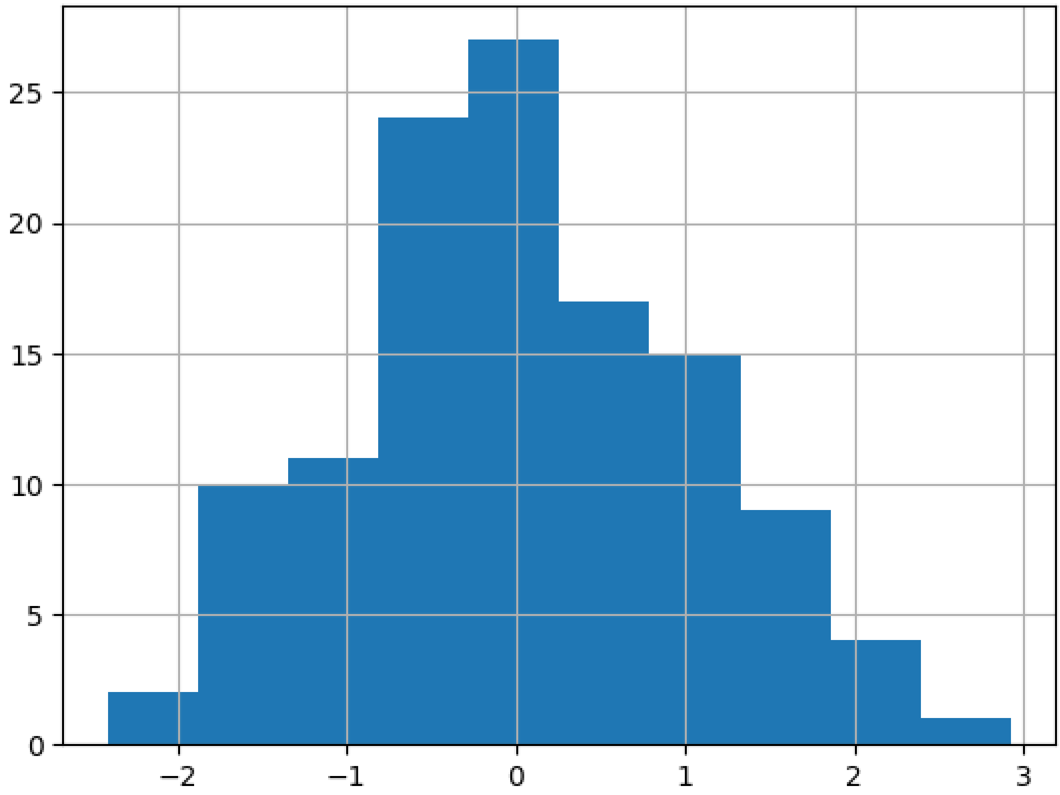
```
data["obs"].describe()
```

```
count    120.000000
mean       0.036494
std        1.002144
min       -2.412101
25%       -0.667615
50%        0.008102
75%        0.762718
max        2.922766
Name: obs, dtype: float64
```

For better visualization, we can use `df.hist()` to plot the histogram of each column.

```
data["obs"].hist()
```

```
<AxesSubplot: >
```



4.2 Subset the dataframe (query)

```
data_sub = data.query("obs > 0")
data_sub[:5]
```

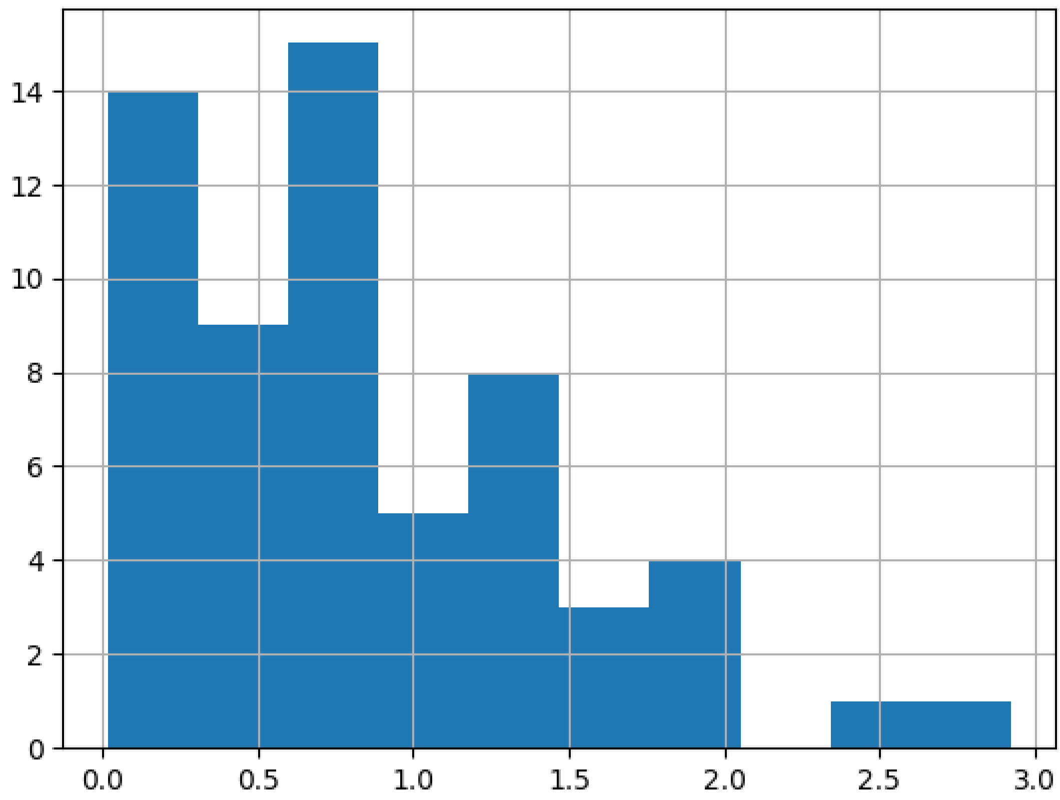
```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	factor	id	env	obs

	factor	id	env	obs
2	C	id_2	env_1	0.653886
3	D	id_1	env_2	0.592581
5	B	id_5	env_2	1.583229
6	C	id_4	env_1	0.875182
7	D	id_5	env_2	1.416503

```
data_sub["obs"].hist()
```

```
<AxesSubplot: >
```




```
data_id1 = data.query("id == 'id_1'")
data_id1[:5]
```

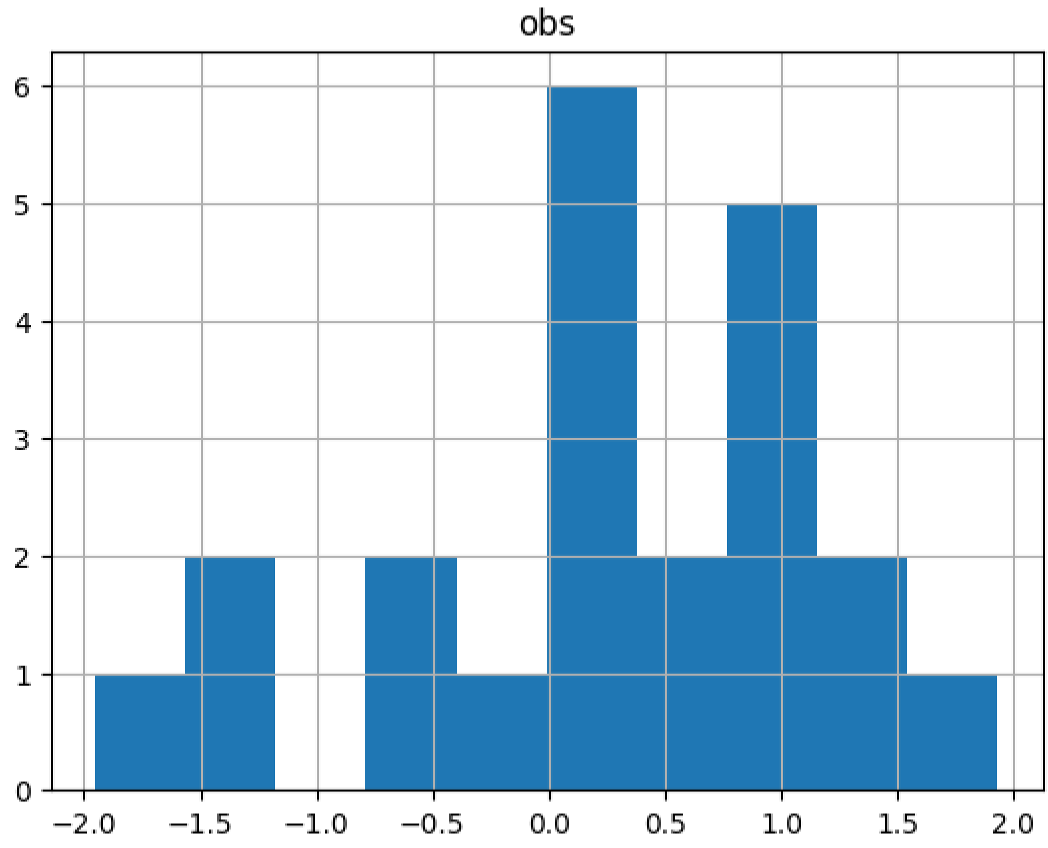
```
.dataframe tbody tr th {
    vertical-align: top;
}
```

```
.dataframe thead th {
    text-align: right;
}
```

	factor	id	env	obs
3	D	id_1	env_2	0.592581
8	A	id_1	env_1	0.043624
10	C	id_1	env_1	0.154333
18	C	id_1	env_1	0.355309
20	A	id_1	env_1	0.879280

```
data_id1.hist()
```

```
array([[<AxesSubplot: title={'center': 'obs'}>]], dtype=object)
```



Multiple conditions can be combined using `&` (and) and `|` (or).

```
data.query("id == 'id_1' and (obs > 1 or obs < -1)")
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	factor	id	env	obs
21	B	id_1	env_2	-1.950799
23	D	id_1	env_2	1.524525
32	A	id_1	env_1	-1.442376
43	D	id_1	env_2	-1.418744
68	A	id_1	env_1	1.929814
81	B	id_1	env_2	1.399891
116	A	id_1	env_1	1.098145

4.3 Grouping

```
data.groupby("id")["obs"].mean()
```

```
id
id_1    0.227263
id_2    0.121979
id_3   -0.229144
id_4   -0.161469
id_5    0.312349
id_6   -0.184553
Name: obs, dtype: float64
```

```
data.groupby(["id", "factor"])["obs"].mean()
```

```
id    factor
id_1  A          0.450216
      B         -0.125937
      C          0.488375
```

```
       D          0.122448
id_2   A         -0.030652
       B          0.129420
       C          0.475717
       D         -0.475298
id_3   A          0.056030
       B         -0.654136
       C         -0.165900
       D         -0.370053
id_4   A         -0.523897
       B         -0.246820
       C         -0.083987
       D          0.160026
id_5   A          0.258584
       B          0.823033
       C         -0.208428
       D          0.331506
id_6   A          0.218340
       B         -0.417258
       C         -0.275666
       D         -0.721520
Name: obs, dtype: float64
```

```
# multiple calculation
cus_fun = lambda x: x.max() - x.min()
pivot = data.groupby(["id", "factor"])["obs"].agg(["mean", "std", "count",
cus_fun])
pivot
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

		mean	std	count	<lambda_0>
id	**factor**				
id_1	**A**	0.450216	1.149487	6	3.372190
	B	-0.125937	1.196017	6	3.350690
	C	0.488375	0.444752	5	0.945838

		mean	std	count	<lambda_0>
id	**factor**				
	D	0.122448	1.130064	5	2.943270
id_2	**A**	-0.030652	0.525032	4	1.072389
	B	0.129420	0.842530	8	2.520642
	C	0.475717	0.950265	10	2.940279
	D	-0.475298	0.956688	5	2.488064
id_3	**A**	0.056030	0.926571	7	2.849754
	B	-0.654136	0.668831	4	1.575225
	C	-0.165900	1.353717	2	1.914444
	D	-0.370053	0.300505	3	0.566801
id_4	**A**	-0.523897	0.786162	5	2.097706
	B	-0.246820	1.020748	5	2.157371
	C	-0.083987	0.864944	4	2.064482
	D	0.160026	0.762164	6	1.860967
id_5	**A**	0.258584	0.846841	3	1.663688
	B	0.823033	1.107697	3	2.031118
	C	-0.208428	0.766466	3	1.490895
	D	0.331506	1.716422	10	5.334867
id_6	**A**	0.218340	1.552812	5	3.078880
	B	-0.417258	0.882586	4	2.123702
	C	-0.275666	0.778769	6	1.978419
	D	-0.721520	NaN	1	0.000000

```
pivot.loc["id_5"]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	mean	std	count	<lambda_0>
factor				
A	0.258584	0.846841	3	1.663688
B	0.823033	1.107697	3	2.031118
C	-0.208428	0.766466	3	1.490895
D	0.331506	1.716422	10	5.334867

```
pivot.loc["id_3"].loc["A"]
```

```
mean           0.056030
std            0.926571
count          7.000000
<lambda_0>     2.849754
Name: A, dtype: float64
```

```
data_pivot = pivot.reset_index()
data_pivot
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	id	factor	mean	std	count	<lambda_0>
0	id_1	A	0.450216	1.149487	6	3.372190
1	id_1	B	-0.125937	1.196017	6	3.350690
2	id_1	C	0.488375	0.444752	5	0.945838
3	id_1	D	0.122448	1.130064	5	2.943270
4	id_2	A	-0.030652	0.525032	4	1.072389
5	id_2	B	0.129420	0.842530	8	2.520642

	id	factor	mean	std	count	<lambda_0>
6	id_2	C	0.475717	0.950265	10	2.940279
7	id_2	D	-0.475298	0.956688	5	2.488064
8	id_3	A	0.056030	0.926571	7	2.849754
9	id_3	B	-0.654136	0.668831	4	1.575225
10	id_3	C	-0.165900	1.353717	2	1.914444
11	id_3	D	-0.370053	0.300505	3	0.566801
12	id_4	A	-0.523897	0.786162	5	2.097706
13	id_4	B	-0.246820	1.020748	5	2.157371
14	id_4	C	-0.083987	0.864944	4	2.064482
15	id_4	D	0.160026	0.762164	6	1.860967
16	id_5	A	0.258584	0.846841	3	1.663688
17	id_5	B	0.823033	1.107697	3	2.031118
18	id_5	C	-0.208428	0.766466	3	1.490895
19	id_5	D	0.331506	1.716422	10	5.334867
20	id_6	A	0.218340	1.552812	5	3.078880
21	id_6	B	-0.417258	0.882586	4	2.123702
22	id_6	C	-0.275666	0.778769	6	1.978419
23	id_6	D	-0.721520	NaN	1	0.000000

```python
data_pivot.to_csv("out_pivot.csv", index=False)
```

```
!cat out_pivot.csv
```

```
id,factor,mean,std,count,<lambda_0>
id_1,A,0.4502155942942296,1.1494867060236997,6,3.3721897757162207
id_1,B,-0.1259366610516431,1.196016985460374,6,3.350689989092595
id_1,C,0.4883747585534007,0.44475233854395824,5,0.9458380117674557
id_1,D,0.12244768621716347,1.1300641404870861,5,2.943269690440943
id_2,A,-0.030652242441548905,0.5250324077373204,4,1.072389338186159
id_2,B,0.12942035707074961,0.8425298886853323,8,2.520641650849491
id_2,C,0.4757170648811825,0.9502646143283857,10,2.940279011945107
id_2,D,-0.47529762313270607,0.956688486181213,5,2.4880643354394296
id_3,A,0.05602950865985269,0.9265710782024227,7,2.849753629637062
```

```
id_3,B,-0.6541364904986321,0.668831320727894,4,1.5752249399766416
id_3,C,-0.16589986071968604,1.3537165137416785,2,1.9144442533419062
id_3,D,-0.37005307748887545,0.3005047663244578,3,0.5668013533114964
id_4,A,-0.523897437744935,0.7861618270334397,5,2.0977061799076733
id_4,B,-0.24681997904527445,1.0207480310530137,5,2.1573714859726527
id_4,C,-0.08398675310215067,0.8649444477299162,4,2.0644821538776297
id_4,D,0.16002643891394427,0.7621635992642647,6,1.8609673554616017
id_5,A,0.2585839127391286,0.8468407992687575,3,1.6636883541767142
id_5,B,0.8230330925737949,1.1076966694366093,3,2.031118209894996
id_5,C,-0.20842750162321177,0.7664656533758334,3,1.4908953311812227
id_5,D,0.3315064539246367,1.716422386830676,10,5.334867463079514
id_6,A,0.2183398832539288,1.5528119273700163,5,3.0788800254758226
id_6,B,-0.41725782592547744,0.8825863995871458,4,2.1237017745237234
id_6,C,-0.275666141606862,0.7787689601953202,6,1.9784193244503814
id_6,D,-0.7215196634577331,,1,0.0
```