Pandas基础 C08

信息科学技术学院

胡俊峰

Pandas —— Panel data analysis

- 序列：indexed list
- 数据表处理
- 表间计算
- 聚合运算
- Pandas的统计功能与应用

The Pandas Series Object —— 序列

A Pandas `Series` is a one-dimensional array of indexed data. It can be created from a list or array as

缺省情况类似excel的表格，自动维护标号索引

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
print(data)

data.index
```

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64

RangeIndex(start=0, stop=4, step=1)
```

```
data = pd.Series(i*i for i in [0.25, 0.5, 0.75, 1.0])
print(data)
data.index
```

```
0    0.0625
1    0.2500
2    0.5625
3    1.0000
```

与数组类似，支持下标切片访问操作

```
data.values
```

```
array([ 0.25,  0.5 ,  0.75,  1.  ])
```

The `index` is an array-like object of type `pd.Index`

```
data[1]
```

```
0.5
```

```
data[1:3]
```

```
1    0.50
2    0.75
dtype: float64
```

也可以指定可哈希的索引项，类似dict

```python
data = pd.Series([0.5, 0.25, 1.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
print(data)
print(data.sort_values())
data['b']
```

```
a    0.50
b    0.25
c    1.75
d    1.00
dtype: float64
b    0.25
a    0.50
d    1.00
c    1.75
dtype: float64
```

```
0.25
```

非连续的索引项也可以，但一般建议避免非连续数字

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=[2, 5, 3, 7])
data[5]
```

```
0.5
```

```
data[data>0.7] * 2
```

类似numpy，可以通过布尔条件生成下标访问序列

```
3    1.5
7    2.0
dtype: float64
```

Indexers: loc, iloc, and ix

按位置索引

These slicing and indexing conventions can be a source of confusion. For example, if your `Series` has an explicit integer index, an indexing operation such as `data[1]` will use the explicit indices, while a slicing operation like `data[1:3]` will use the implicit Python-style index.

```
data = pd.Series(['a', 'b', 'c'], index=[1, 3, 5])
data
```

```
1    a
3    b
5    c
dtype: object
```

```
# explicit index when indexing
data[1]
```

```
'a'
```

```
print(data.loc[1])
print(data.iloc[1])

```

```
a
b
```

```
# implicit index when slicing
data[1:3]
```

```
3    b
5    c
dtype: object
```

```python
print(0.75 in data)
0.75 in data.values
```

False

True

```python
for i in data.values:
    print(i)
```

迭代器，也要指明具体字段

0.25
0.5
0.75
1.0

True

```python
for k in data.index:
    print(data[k])
```

0.5
0.25
1.75
1.0

```python
a = pd.Series([2, 4, 6])
b = pd.Series({2:'a', 1:'b', 3:'c'})
print(b[1])
2 in b
```

词典数据初始化序列

```
b

True
```

```python
for i in b:
    print (i)
```

```
a
b
c
```

The Pandas DataFrame Object

- 视角1：多个对齐的序列（series）的组合（record）
- 视角2：支持多层索引的二维数据表

初始化一个dataframe： 字段名+数据序列

```
df2 = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20240102"),
        "C": pd.Series([ i for i in range(1,11,2)]),
        "D": np.array([3] * 5, dtype="float"),
        "E": pd.Categorical(["test", "train", "test", "train","break"]),
        "F": "流水账",
    }
)

df2
```

	A	B	C	D	E	F
0	1.0	2024-01-02	1	3.0	test	流水账
1	1.0	2024-01-02	3	3.0	train	流水账
2	1.0	2024-01-02	5	3.0	test	流水账
3	1.0	2024-01-02	7	3.0	train	流水账
4	1.0	2024-01-02	9	3.0	break	流水账

```
df2.dtypes
```

```
A            float64
B      datetime64[s]
C              int64
D            float64
E           category
F             object
dtype: object
```

表5-1：可以输入给DataFrame构造器的数据

类型	说明
二维ndarray	数据矩阵，还可以传入行标和列标
由数组、列表或元组组成的字典	每个序列会变成DataFrame的一列。所有序列的长度必须相同
NumPy的结构化/记录数组	类似于“由数组组成的字典”
由Series组成的字典	每个Series会成为一列。如果没有显式指定索引，则各Series的索引会被合并成结果的行索引
由字典组成的字典	各内层字典会成为一列。键会被合并成结果的行索引，跟“由Series组成的字典”的情况一样
字典或Series的列表	各项将会成为DataFrame的一行。字典键或Series索引的并集将会成为DataFrame的列标
由列表或元组组成的列表	类似于“二维ndarray”
另一个DataFrame	该DataFrame的索引将会被沿用，除非显式指定了其他索引
NumPy的MaskedArray	类似于“二维ndarray”的情况，只是掩码值在结果DataFrame会变成NA/缺失值

词典的列表生成dataframe：

If some keys in the dictionary are missing, Pandas will fill them in with `NaN` (i.e., "not a number") values:

```
data = [{'a': i, 'b': 2 * i}
        for i in range(3)]

print(data)
pd.DataFrame(data)
```

索引key + 列表生成式

```
[{'a': 0, 'b': 0}, {'a': 1, 'b': 2}, {'a': 2, 'b': 4}]
```

	a	b
0	0	0
1	1	2
2	2	4

From a two-dimensional NumPy array

Given a two-dimensional array of data, we can create a `DataFrame` with any specified column and index names. If omitted, an integer index will be used for each:

```
pd.DataFrame(np.random.rand(3, 2),
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])
```

	foo	bar
a	0.865257	0.213169
b	0.442759	0.108267
c	0.047110	0.905718

时间序列索引

```
dates = pd.date_range("20240101", periods=6)
dates
```

```
DatetimeIndex(['2024-01-01', '2024-01-02', '2024-01-03', '2024-01-04',
               '2024-01-05', '2024-01-06'],
              dtype='datetime64[ns]', freq='D')
```

```
df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))

df
```

	A	B	C	D
2024-01-01	-0.420908	0.561723	1.029381	-0.547718
2024-01-02	1.107427	1.996233	-0.542999	0.532342
2024-01-03	-0.570062	0.150368	-0.121603	0.224366
2024-01-04	0.338397	0.019086	-0.236876	-1.211000
2024-01-05	0.458176	0.488300	0.099074	-2.192264
2024-01-06	0.463506	0.348168	0.154023	0.393635

排序:

```python
df.sort_values(by="B")
```

	A	B	C	D
2024-01-04	1.181177	-1.011614	0.082741	0.277036
2024-01-01	1.272659	0.160868	-0.955028	0.111053
2024-01-02	-0.580428	0.381052	0.043766	1.561938
2024-01-06	-0.698205	0.604560	-0.960241	0.732983
2024-01-03	-0.655237	0.841888	-0.331852	1.454877
2024-01-05	-0.735982	2.021000	1.199130	-0.332525

```python
df.sort_index(ascending=False)
```

	A	B	C	D
2024-01-06	-0.698205	0.604560	-0.960241	0.732983
2024-01-05	-0.735982	2.021000	1.199130	-0.332525
2024-01-04	1.181177	-1.011614	0.082741	0.277036
2024-01-03	-0.655237	0.841888	-0.331852	1.454877
2024-01-02	-0.580428	0.381052	0.043766	1.561938
2024-01-01	1.272659	0.160868	-0.955028	0.111053

下标访问：

```
df["A"]
```

```
2024-01-01   -0.420908
2024-01-02    1.107427
2024-01-03   -0.570062
2024-01-04    0.338397
2024-01-05    0.458176
2024-01-06    0.463506
Freq: D, Name: A, dtype: float64
```

```
df[0:3]
```

	A	B	C	D
2024-01-01	-0.420908	0.561723	1.029381	-0.547718
2024-01-02	1.107427	1.996233	-0.542999	0.532342
2024-01-03	-0.570062	0.150368	-0.121603	0.224366

```
df.loc[dates[1]]
```

```
A    1.107427
B    1.996233
C   -0.542999
D    0.532342
Name: 2024-01-02 00:00:00, dtype: float64
```

下标切片：

```python
df.loc[:, ["A", "B"]]
```

	A	B
2024-01-01	1.272659	0.160868
2024-01-02	-0.580428	0.381052
2024-01-03	-0.655237	0.841888
2024-01-04	1.181177	-1.011614
2024-01-05	-0.735982	2.021000
2024-01-06	-0.698205	0.604560

```python
df.loc["20240102":"20240104", ["A", "B"]]
```

	A	B
2024-01-02	-0.580428	0.381052
2024-01-03	-0.655237	0.841888
2024-01-04	1.181177	-1.011614

```python
a = df.loc[dates[0], "A"]
b = df.at[dates[0], "A"]
a == b
```

True

生成视图副本：

```
df2 = df.iloc[1:4, :2].copy()
df2
```

	A	B
2024-01-02	-0.580428	0.381052
2024-01-03	-0.655237	0.841888
2024-01-04	1.181177	-1.011614

```
df[df["B"] > 0] # 生成布尔下标
```

	A	B	C	D
2024-01-01	1.272659	0.160868	-0.955028	0.111053
2024-01-02	-0.580428	0.381052	0.043766	1.561938
2024-01-03	-0.655237	0.841888	-0.331852	1.454877
2024-01-05	-0.735982	2.021000	1.199130	-0.332525
2024-01-06	-0.698205	0.604560	-0.960241	0.732983

新加列:

```python
new_col = pd.Series([1, 2, 3, 4, 5, 6], index=pd.date_range("20240102", periods=6))
new_col
```

```
2024-01-02    1
2024-01-03    2
2024-01-04    3
2024-01-05    4
2024-01-06    5
2024-01-07    6
Freq: D, dtype: int64
```

```python
df["F"] = new_col
df
```

	A	B	C	D	F
2024-01-01	1.272659	0.160868	-0.955028	0.111053	NaN
2024-01-02	-0.580428	0.381052	0.043766	1.561938	1.0
2024-01-03	-0.655237	0.841888	-0.331852	1.454877	2.0
2024-01-04	1.181177	-1.011614	0.082741	0.277036	3.0
2024-01-05	-0.735982	2.021000	1.199130	-0.332525	4.0
2024-01-06	-0.698205	0.604560	-0.960241	0.732983	5.0

算术运算-广播：

```
df["D"] = 3 # 广播机制
df
```

	A	B	C	D	F
2024-01-01	1.272659	0.160868	-0.955028	3	NaN
2024-01-02	-0.580428	0.381052	0.043766	3	1.0
2024-01-03	-0.655237	0.841888	-0.331852	3	2.0
2024-01-04	1.181177	-1.011614	0.082741	3	3.0
2024-01-05	-0.735982	2.021000	1.199130	3	4.0
2024-01-06	-0.698205	0.604560	-0.960241	3	5.0

```
df2 = df.copy()
df2[df2 > 0] = df2*2
df2
```

	A	B	C	D	F
2024-01-01	2.545319	0.321737	-0.955028	6	NaN
2024-01-02	-0.580428	0.762104	0.087531	6	2.0
2024-01-03	-0.655237	1.683777	-0.331852	6	4.0
2024-01-04	2.362354	-1.011614	0.165483	6	6.0
2024-01-05	-0.735982	4.042000	2.398260	6	8.0
2024-01-06	-0.698205	1.209119	-0.960241	6	10.0

Working with NumPy ufunc

```
df = pd.DataFrame(rng.randint(0, 10, (3,4)),
                  columns=['A', 'B', 'C', 'D'])
df
```

	A	B	C	D
0	9	2	6	7
1	4	3	7	7
2	2	5	4	1

采用Numpy的广播机制，逐元素计算

```
np.sin(df * np.pi / 4)
```

	A	B	C	D
0	7.071068e-01	1.000000	-1.000000e+00	-0.707107
1	1.224647e-16	0.707107	-7.071068e-01	-0.707107
2	1.000000e+00	-0.707107	1.224647e-16	0.707107

算术运算-赋值：

```
df["D"] = 3 # 广播机制
df
```

	A	B	C	D	F
2024-01-01	1.272659	0.160868	-0.955028	3	NaN
2024-01-02	-0.580428	0.381052	0.043766	3	1.0
2024-01-03	-0.655237	0.841888	-0.331852	3	2.0
2024-01-04	1.181177	-1.011614	0.082741	3	3.0
2024-01-05	-0.735982	2.021000	1.199130	3	4.0
2024-01-06	-0.698205	0.604560	-0.960241	3	5.0

```
df2 = df.copy()
df2[df2 > 0] = df2*2
df2
```

	A	B	C	D	F
2024-01-01	2.545319	0.321737	-0.955028	6	NaN
2024-01-02	-0.580428	0.762104	0.087531	6	2.0
2024-01-03	-0.655237	1.683777	-0.331852	6	4.0
2024-01-04	2.362354	-1.011614	0.165483	6	6.0
2024-01-05	-0.735982	4.042000	2.398260	6	8.0
2024-01-06	-0.698205	1.209119	-0.960241	6	10.0

筛选+赋值：

```
data.loc[data.density > 100, ['pop', 'density']]
```

	pop	density
Florida	19552860	114.806121
New York	19651127	139.076746

Any of these indexing conventions may also be used to set or modify values; this is done in the standard way that you might be accustomed to from working with NumPy:

```
data.iloc[0, 2] = 90
data
```

Iloc支持多重索引

	area	pop	density
California	423967	38332521	90.000000
Texas	695662	26448193	38.018740
New York	141297	19651127	139.076746
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763

Dataframe列间的运算自动进行索引键对齐/补缺

Out[22]:

	A	B
0	2	4
1	18	6

In [23]:

```
B = pd.DataFrame(rng.randint(0, 10, (3, 3)),
                 columns=list('BAC'))
B
```

Out[23]:

	B	A	C
0	4	8	6
1	1	3	8
2	1	9	8

In [24]:

```
A + B
```

Out[24]:

	A	B	C
0	10.0	8.0	NaN
1	21.0	7.0	NaN
2	NaN	NaN	NaN

行列对象间支持的运算符：

The following table lists Python operators and their equivalent Pandas object methods:

Python Operator	Pandas Method(s)
`+`	`add()`
`-`	`sub()`, `subtract()`
`*`	`mul()`, `multiply()`
`/`	`truediv()`, `div()`, `divide()`
`//`	`floordiv()`
`%`	`mod()`
`**`	`pow()`

Frame，按行broadcasting

```
A = rng.randint(10, size=(3, 4))
A
```

```
array([[9, 4, 1, 3],
       [6, 7, 2, 0],
       [3, 1, 7, 3]])
```

```
df = pd.DataFrame(A, columns=list('QRST'))
df - df.iloc[0]
```

	Q	R	S	T
0	0	0	0	0
1	-3	3	1	-3
2	-6	-3	6	0

```
df.subtract(df['R'], axis=0)
```

	Q	R	S	T
0	5	0	-3	-1
1	-1	0	-5	-7
2	2	0	6	2

运算过程中类型自适应转换

The following table lists the upcasting conventions in Pandas when NA values are introduced:

Typeclass	Conversion When Storing NAs	NA Sentinel Value
`floating`	No change	`np.nan`
`object`	No change	`None` or `np.nan`
`integer`	Cast to `float64`	`np.nan`
`boolean`	Cast to `object`	`None` or `np.nan`

Keep in mind that in Pandas, string data is always stored with an `object` dtype.

Detecting null values

Pandas data structures have two useful methods for detecting null data: `isnull()` and `notnull()`. Either one will return a Boolean mask over the data. For example:

```
data = pd.Series([1, np.nan, 'hello', None])
```

```
data.isnull()
```

```
0    False
1     True
2    False
3     True
dtype: bool
```

As mentioned in Data Indexing and Selection, Boolean masks can be used directly as a `Series` or `DataFrame` index:

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

```
0        1
2    hello
dtype: object
```

We can fill NA entries with a single value, such as zero:

```
data.fillna(0)
```

```
a    1.0
b    0.0
c    2.0
d    0.0
e    3.0
dtype: float64
```

We can specify a forward-fill to propagate the previous value forward:

```
# forward-fill
data.fillna(method='ffill')
```

```
a    1.0
b    1.0
c    2.0
d    2.0
e    3.0
dtype: float64
```

例子

```
df3 = df.reindex(index=dates[0:4], columns=list(df.columns) + ["E"])
df3.loc[dates[0] : dates[2], "E"] = 1
df3
```

	A	B	C	D	F	E
2024-01-01	-0.554219	-0.405164	-0.277979	3	NaN	1.0
2024-01-02	-2.216250	-0.139490	0.164979	3	1.0	1.0
2024-01-03	-1.134139	-1.330454	1.132189	3	2.0	1.0
2024-01-04	1.122301	0.386015	-0.942354	3	3.0	NaN

```
print(df3.fillna(value=0))   # df3.dropna() # 去掉包含na的数据行
#df3
```

```
                   A         B         C  D    F    E
2024-01-01 -0.554219 -0.405164 -0.277979  3  0.0  1.0
2024-01-02 -2.216250 -0.139490  0.164979  3  1.0  1.0
2024-01-03 -1.134139 -1.330454  1.132189  3  2.0  1.0
2024-01-04  1.122301  0.386015 -0.942354  3  3.0  0.0
```

```python
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)

area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
```

```python
states = pd.DataFrame({'population': population,'area': area})
states
```

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

Step 1、词典到序列数据
Step 2、多序列合并生成dataframe

```python
population_dict = {'California': 38332521, 'Texas': 26448193,
                   'New York': 19651127, 'W.DC': 11000000}
population = pd.Series(population_dict)

area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)
```

```python
states = pd.DataFrame({'population': population,'area': area})
states
```

	population	area
California	38332521.0	423967.0
Florida	NaN	170312.0
Illinois	NaN	149995.0
New York	19651127.0	141297.0
Texas	26448193.0	695662.0
W.DC	11000000.0	NaN

索引-数据 与 索引合并（非对齐情况）：
索引扩展，数据用NaN填充

```python
print(states.index)
print(states.columns)
for i in states.columns:
    print(states[i])
```

行列的表头都是索引

```
Index(['California', 'Florida', 'Illinois', 'New York', 'Texas', 'W.DC'], dtype='ob
ject')
Index(['population', 'area'], dtype='object')
California    38332521.0
Florida              NaN
Illinois             NaN
New York      19651127.0
Texas         26448193.0
W.DC          11000000.0
Name: population, dtype: float64
California    423967.0
Florida       170312.0
Illinois      149995.0
New York      141297.0
Texas         695662.0
W.DC               NaN
Name: area, dtype: float64
```

都是索引对象，逻辑上对等（可互换）

行列互换操作

```
s = data.T
print(s)
s['California']
```

```
         California         Texas      New York       Florida      Illinois
area     4.239670e+05  6.956620e+05  1.412970e+05  1.703120e+05  1.499950e+05
pop      3.833252e+07  2.644819e+07  1.965113e+07  1.955286e+07  1.288214e+07
density  9.041393e+01  3.801874e+01  1.390767e+02  1.148061e+02  8.588376e+01

area       4.239670e+05
pop        3.833252e+07
density    9.041393e+01
Name: California, dtype: float64
```

计算生成新列:

```
data['density'] = data['pop'] / data['area']
data
```

	area	pop	density
California	423967	38332521	90.413926
Florida	170312	19552860	114.806121
Illinois	149995	12882135	85.883763
New York	141297	19651127	139.076746
Texas	695662	26448193	38.018740

多维表操作

- 本质上是多重索引支持的2维数表

```
# hierarchical indices and columns
index = pd.MultiIndex.from_product([[2013, 2014], [1, 2, 3, 4]],
                                    names=['year', 'season'])  # 行、列都是2维索引
columns = pd.MultiIndex.from_product([['Bob1', 'Bob2', 'Bob3'], ['WR', 'HD']],
                                      names=['name', 'type'])

# mock some data
data = np.random.randint(1,91, 48)  # 1-91之间的整数, 48个
data = data.reshape(8,6)
# create the DataFrame
wh_data = pd.DataFrame(data, index=index, columns=columns)
wh_data
```

	name	Bob1		Bob2		Bob3	
	type	WR	HD	WR	HD	WR	HD
year	season						
2013	1	55	29	27	74	75	48
	2	9	86	54	5	11	14
	3	18	22	79	42	39	73
	4	77	77	52	77	65	17
2014	1	45	30	9	43	16	18

多维索引表操作

直接定位行列的索引:

```
wh_data.iloc[:3, :3]   # 可以按二维表的iloc直接切片
```

	name	Bob		Guido
	type	WR	HD	WR
year	season			
2013	1	39	14	2
	2	46	20	21
	3	50	1	74

.loc可以实现最外层索引的切片：

```
wh_data.loc[2014:,:'Bob2']    # 右面也是闭区间
# wh_data[2014:,:'Bob2']  # TypeError: unhashable type: 'slice'
```

	name	Bob1		Bob2	
	type	WR	HD	WR	HD
year	season				
2014	1	27	85	69	78
	2	2	26	27	16
	3	17	9	42	41
	4	71	33	43	66

使用indexSlice对象（了解）

```
# wh_data.loc[2014:, (:'WR')]  # invalid syntax
idx = pd.IndexSlice
wh_data.loc[idx[2014:],idx[ :,'WR']]   # 加入index切片对象辅助实现
```

	name	Bob1	Bob2	Bob3
	type	WR	WR	WR
year	season			
2014	1	27	69	50
	2	2	27	56
	3	17	42	82
	4	71	43	59

层次-组合 索引（Hierarchical-Indexing）

```
index = [('California', 2000), ('California', 2010),
         ('New York', 2000), ('New York', 2010),
         ('Texas', 2000), ('Texas', 2010)]
populations = [33871648, 37253956,
               18976457, 19378102,
               20851820, 25145561]
pop = pd.Series(populations, index=index)
pop
```

```
(California, 2000)    33871648
(California, 2010)    37253956
(New York, 2000)      18976457
(New York, 2010)      19378102
(Texas, 2000)         20851820
(Texas, 2010)         25145561
dtype: int64
```

类似二维表切片

```
pop[:, 2010]
```

```
California    37253956
New York      19378102
Texas         25145561
dtype: int64
```

多键值词典索引初始化：自动识别为多层索引

Similarly, if you pass a dictionary with appropriate tuples as keys, Pandas will automatically recognize this and use a `MultiIndex` by default:

```
data = {('California', 2000): 33871648,
        ('California', 2010): 37253956,
        ('Texas', 2000): 20851820,
        ('Texas', 2010): 25145561,
        ('New York', 2000): 18976457,
        ('New York', 2010): 19378102}
pd.Series(data)
```

```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

MultiIndex VS extra dimension

```
#unstack() method will quickly convert a multiply indexed Series
#into a conventionally indexed DataFrame:
pop_df = pop.unstack()
pop_df
```

	2000	2010
California	33871648	37253956
New York	18976457	19378102
Texas	20851820	25145561

```
#unstack() method will quickly convert a multiply indexed Series into a conventi
pop_df.stack()
```

```
California  2000    33871648
            2010    37253956
New York    2000    18976457
            2010    19378102
Texas       2000    20851820
            2010    25145561
dtype: int64
```

```
pop_df = pd.DataFrame({'total': pop,
                       'under18': [9267089, 9284094,
                                   4687374, 4318033,
                                   5906301, 6879014]})
pop_df
```

		total	under18
California	2000	33871648	9267089
	2010	37253956	9284094
New York	2000	18976457	4687374
	2010	19378102	4318033
Texas	2000	20851820	5906301
	2010	25145561	6879014

```
f_u18 = pop_df['under18'] / pop_df['total']
f_u18.unstack()
```

	2000	2010
California	0.273594	0.249211
New York	0.247010	0.222831
Texas	0.283251	0.273568

多层索引的生成方案:

```
pd.MultiIndex.from_arrays([['a', 'a', 'b', 'b'], [1, 2, 1, 2]])
```

```
MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can construct it from a list of tuples giving the multiple index values of each point:

```
pd.MultiIndex.from_tuples([('a', 1), ('a', 2), ('b', 1), ('b', 2)])
```

```
MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

You can even construct it from a Cartesian product of single indices:

```
pd.MultiIndex.from_product([['a', 'b'], [1, 2]])
```

```
MultiIndex(levels=[['a', 'b'], [1, 2]],
           labels=[[0, 0, 1, 1], [0, 1, 0, 1]])
```

多个键值直接组合为多层索引：

Methods of MultiIndex Creation

The most straightforward way to construct a multiply indexed `Series` or `DataFrame` is to simply pass a list of two or more index arrays to the constructor. For example:

2]:

```
df = pd.DataFrame(np.random.rand(4, 2),
                  index=[['a', 'a', 'b', 'b'], [1, 2, 1, 2]],
                  columns=['data1', 'data2'])
df
```

]:

		data1	data2
a	1	0.554233	0.356072
	2	0.925244	0.219474
b	1	0.441759	0.610054
	2	0.171495	0.886688

Data Aggregations

Group by certain Key：

- Splitting the data into groups based on some criteria.
- Applying a function to each group independently.

- Pandas has built-in data aggregation methods,
- such as mean(), sum(), and max().

```python
df = pd.DataFrame({'A':['bob','john','bob','jeff','bob','jeff','bob','john'],
                   'B':['one','one','two','three','two','two','one','three'],
                   'C':[3,1,4,1,5,9,2,6],
                   'D':[1,2,3,4,5,6,7,8]})  # 给出4栏数据
df
```

	A	B	C	D
0	bob	one	3	1
1	john	one	1	2
2	bob	two	4	3
3	jeff	three	1	4
4	bob	two	5	5
5	jeff	two	9	6
6	bob	one	2	7
7	john	three	6	8

```python
print (df.groupby('A'))
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000001A27FBE41D0>
```

```python
print (df.groupby('A').groups)
```

```
{'bob': [0, 2, 4, 6], 'jeff': [3, 5], 'john': [1, 7]}
```

按属性分组求均值：

```
1  health_data
```

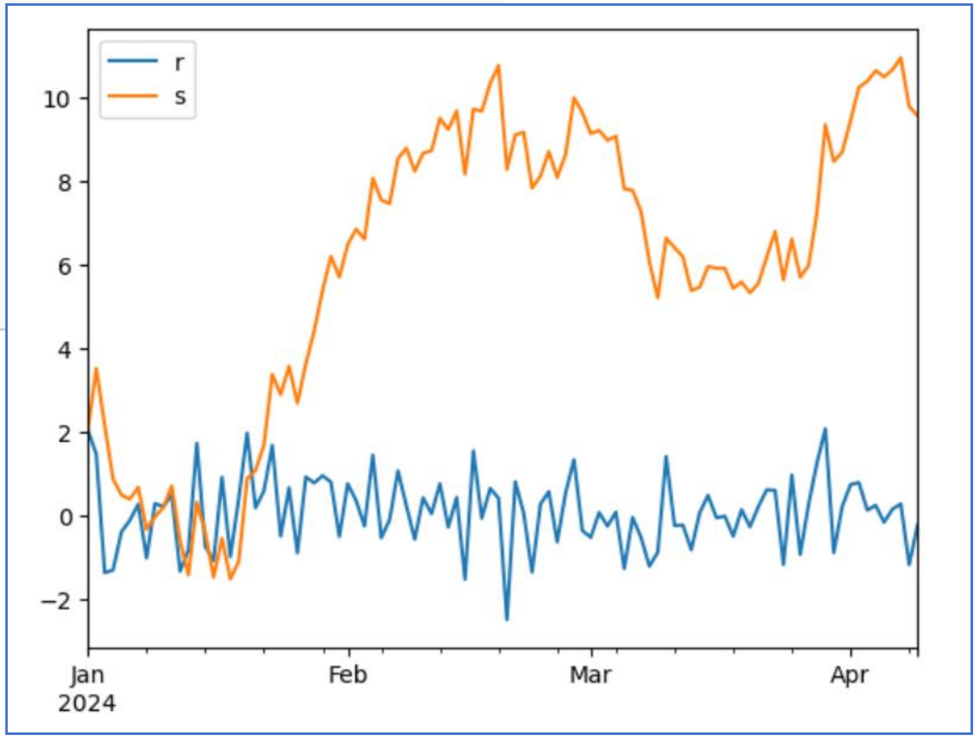
subject		Bob		Guido		Sue	
type		HR	Temp	HR	Temp	HR	Temp
year	visit						
2013	1	31.0	38.7	32.0	36.7	35.0	37.2
	2	44.0	37.7	50.0	35.0	29.0	36.7
2014	1	30.0	37.4	39.0	37.8	61.0	36.9
	2	47.0	37.8	48.0	37.3	51.0	36.5

```
1  data_mean = health_data.mean(level='year')
2  data_mean
```

subject	Bob		Guido		Sue	
type	HR	Temp	HR	Temp	HR	Temp
year						
2013	37.5	38.2	41.0	35.85	32.0	36.95
2014	38.5	37.6	43.5	37.55	56.0	36.70

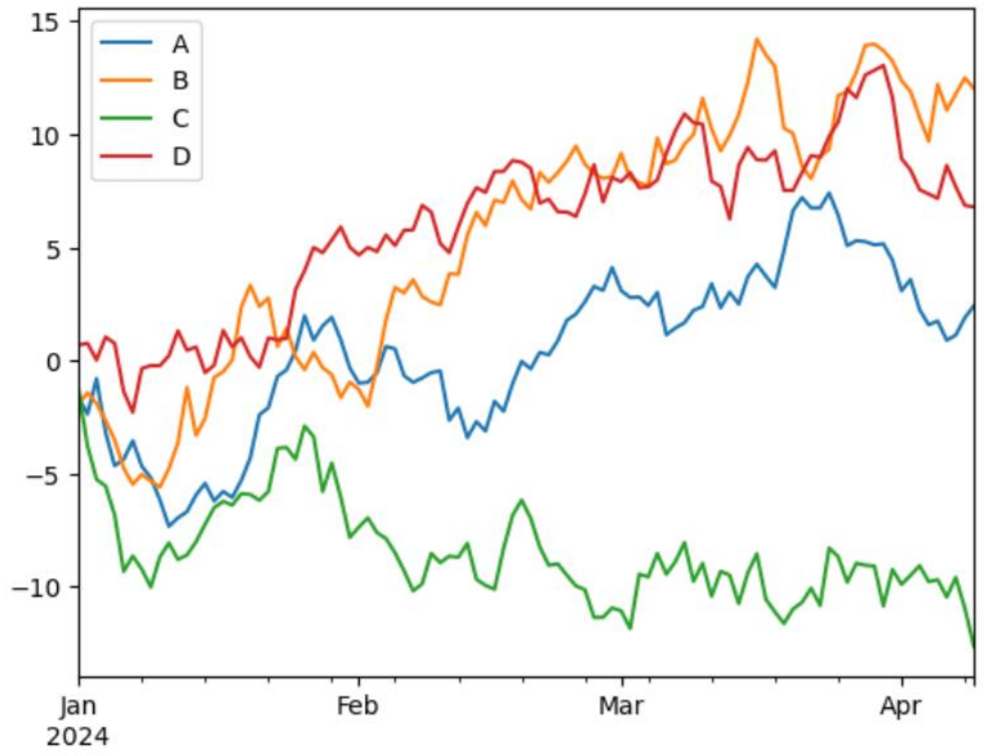
数据分析应用示例：序列可视化

```
import matplotlib.pyplot as plt

ts = pd.Series(np.random.randn(100), index=pd.date_range("1/1/2024",
                                                          periods=100))

ts.plot()

ts = ts.cumsum()
ts.plot()

plt.legend('rs')
```



```
dfr = pd.DataFrame(
    np.random.randn(100, 4), index=ts.index, columns=["A", "B", "C", "D"]
)

dfr2 = dfr.cumsum()
```

```
plt.figure()
dfr2.plot()
plt.legend(loc ='best')
```



数据分析实例：The MovieLens Dataset
电影评分数据集

Neither the University of Minnesota nor any of the researchers involved can guarantee the correctness of the data, its suitability for any particular purpose, or the validity of results based on the use of the data set. The data set may be used for any research purposes under the following conditions:

* The user may not state or imply any endorsement from the University of Minnesota or the GroupLens Research Group.

* The user must acknowledge the use of the data set in publications resulting from the use of the data set (see below for citation information).

* The user may not redistribute the data without separate permission.

* The user may not use this information for any commercial or revenue-bearing purposes without first obtaining permission from a faculty member of the GroupLens Research Project at the University of Minnesota.

If you have any further questions or comments, please contact GroupLens <grouplens-info@cs.umn.edu>.

CITATION

==

To acknowledge use of the dataset in publications, please cite the following paper:

F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4, Article 19 (December 2015), 19 pages. DOI=http://dx.doi.org/10.1145/2827872

名称	修改日期	类型
movies	2018/7/10 20:01	DAT
ratings	2018/7/10 20:01	DAT
README	2018/7/10 20:01	MD 文件
users	2018/7/10 20:01	DAT

加载用户评分数据

```python
# Read the Ratings File
ratings = pd.read_csv(os.path.join(MOVIELENS_DIR, RATING_DATA_FILE),
                      sep='::',
                      engine='python',
                      encoding='latin-1',
                      names=['user_id', 'movie_id', 'rating', 'timestamp'])
```

```python
print(len(ratings), 'ratings loaded')
ratings.head()                        # by default显示前5条
```

```
1000209 ratings loaded
```

	user_id	movie_id	rating	timestamp
0	1	1193	5	978300760
1	1	661	3	978302109
2	1	914	3	978301968
3	1	3408	4	978300275
4	1	2355	5	978824291

fruit.csv | online_shopping_10_cats.csv | douban.dat | movies.dat | ratings.dat

```
1::1193::5::978300760
1::661::3::978302109
1::914::3::978301968
1::3408::4::978300275
1::2355::5::978824291
1::1197::3::978302268
1::1287::5::978302039
1::2804::5::978300719
1::594::4::978302268
1::919::4::978301368
```

pandas.read_csv

```
pandas.read_csv(filepath_or_buffer, *, sep=_NoDefault.no_default, delimiter=None,
header='infer', names=_NoDefault.no_default, index_col=None, usecols=None, squeeze=None,
```

Parameters: **filepath_or_buffer** : *str, path object or file-like object*

Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is expected. A local file could be: file://localhost/path/to/table.csv.

If you want to pass in a path object, pandas accepts any `os.PathLike`.

By file-like object, we refer to objects with a `read()` method, such as a file handle (e.g. via builtin `open` function) or `StringIO`.

sep : *str, default ','*

Delimiter to use. If sep is None, the C engine cannot automatically detect the separator, but the Python parsing engine can, meaning the latter will be used and automatically detect the separator by Python's builtin sniffer tool, `csv.Sniffer`. In addition, separators longer than 1 character and different from `'\s+'` will be interpreted as regular expressions and will also force the use of the Python parsing engine. Note that regex delimiters are prone to ignoring quoted data. Regex example: `'\r\t'`.

delimiter : *str, default* `None`

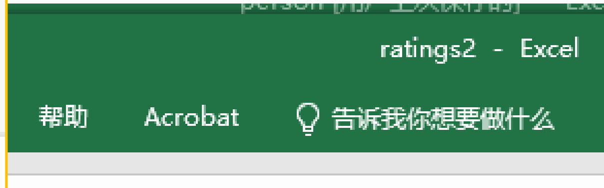
Alias for sep.

header : *int, list of int, None, default 'infer'*

写出数据表文件

```python
# Save into ratings-2.csv
ratings.to_csv('ratings2.csv',
               header = True,
               encoding = 'latin-1',
               index = False
               #columns=['user_id', 'movie_id', 'rating', 'timestamp']
              )
print('Saved to', 'ratings2.csv')
```

```
Saved to ratings2.csv
```



	A	B	C	D	E	F	G	H	I
1	user_id	movie_id	rating	timestamp					
2	1	1193	5	978300760					
3	1	661	3	978302109					
4	1	914	3	978301968					
5	1	3408	4	978300275					
6	1	2355	5	978824291					
7	1	1197	3	978302268					
8	1	1287	5	978302039					
9	1	2804	5	978300719					
10	1	594	4	978302268					
11	1	919	4	978301368					
12	1	595	5	978824268					
13	1	938	4	978301752					
14	1	2398	4	978302281					
15	1	2918	4	978302124					
16	1	1035	5	978301753					
17	1	2791	4	978302188					
18	1	2687	3	978824268					
19	1	2018	4	978301777					

连续值属性量化

```
# Specify User's Age and Occupation Column
AGES = { 1: "Under 18", 18: "18-24", 25: "25-34", 35: "35-44", 45: "45-49", 50: "50-55", 56: "56+" }
OCCUPATIONS = { 0: "other or not specified", 1: "academic/educator", 2: "artist", 3: "clerical/admin",
                4: "college/grad student", 5: "customer service", 6: "doctor/health care",
                7: "executive/managerial", 8: "farmer", 9: "homemaker", 10: "K-12 student", 11: "lawyer",
                12: "programmer", 13: "retired", 14: "sales/marketing", 15: "scientist", 16: "self-employed",
                17: "technician/engineer", 18: "tradesman/craftsman", 19: "unemployed", 20: "writer" }
```

```
# Read the Users File
users = pd.read_csv(os.path.join(MOVIELENS_DIR, USER_DATA_FILE),
                    sep='::',
                    engine='python',
                    encoding='latin-1',
                    names=['user_id', 'gender', 'age', 'occupation', 'zipcode'])

users['age_desc'] = users['age'].apply(lambda x: AGES[x]) # 变换成年龄段

users['occ_desc'] = users['occupation'].apply(lambda x: OCCUPATIONS[x]) # 职业词典
print(len(users), 'descriptions of', max_userid, 'users loaded.')
```

```
6040 descriptions of 6040 users loaded.
```

加工后的数据表：

	user_id	gender	age	occupation	zipcode	age_desc	occ_desc
0	1	F	1	10	48067	Under 18	K-12 student
1	2	M	56	16	70072	56+	self-employed
2	3	M	25	15	55117	25-34	scientist
3	4	M	45	7	2460	45-49	executive/managerial
4	5	M	25	20	55455	25-34	writer
5	6	F	50	9	55117	50-55	homemaker
6	7	M	35	1	6810	35-44	academic/educator
7	8	M	25	12	11413	25-34	programmer
8	9	M	25	17	61614	25-34	technician/engineer
9	10	F	35	1	95370	35-44	academic/educator
10	11	F	25	1	4093	25-34	academic/educator
11	12	M	25	12	32793	25-34	programmer

电影信息表：

```
print(len(movies), 'descriptions of', max_movieid, 'movies loaded.')
movies.head()
```

```
3883 descriptions of 3952 movies loaded.
```

	movie_id	title	genres
0	1	Toy Story (1995)	Animation|Children's|Comedy
1	2	Jumanji (1995)	Adventure|Children's|Fantasy
2	3	Grumpier Old Men (1995)	Comedy|Romance
3	4	Waiting to Exhale (1995)	Comedy|Drama
4	5	Father of the Bride Part II (1995)	Comedy

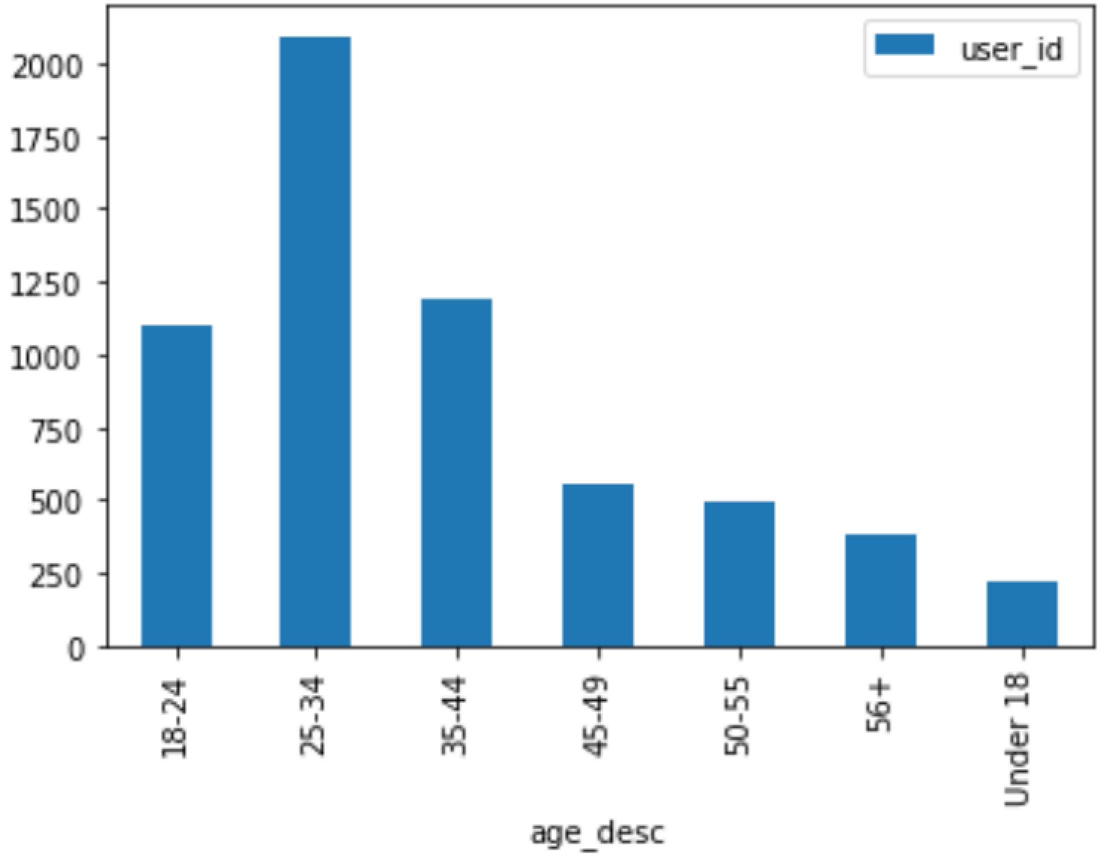
```python
count_age = users.groupby(['age_desc']).count() # 按年龄段统计
count_age
```

	user_id	gender	zipcode	occ_desc
age_desc				
18-24	1103	1103	1103	1103
25-34	2096	2096	2096	2096
35-44	1193	1193	1193	1193
45-49	550	550	550	550
50-55	496	496	496	496
56+	380	380	380	380
Under 18	222	222	222	222

```
count_age = users.groupby(by = ['age_desc'])['user_id'].count().reset_index()
count_age
```

	age_desc	user_id
0	18-24	1103
1	25-34	2096
2	35-44	1193
3	45-49	550
4	50-55	496
5	56+	380
6	Under 18	222

```
ax = count_age.plot.bar(x = 'age_desc', y = 'user_id')
```



pandas.DataFrame.plot

pandas.DataFrame.plot.area

pandas.DataFrame.plot.bar

pandas.DataFrame.plot.barh

pandas.DataFrame.plot.box

pandas.DataFrame.plot.density

pandas.DataFrame.plot.hexbin

pandas.DataFrame.plot.hist

pandas.DataFrame.plot.kde

pandas.DataFrame.plot.line

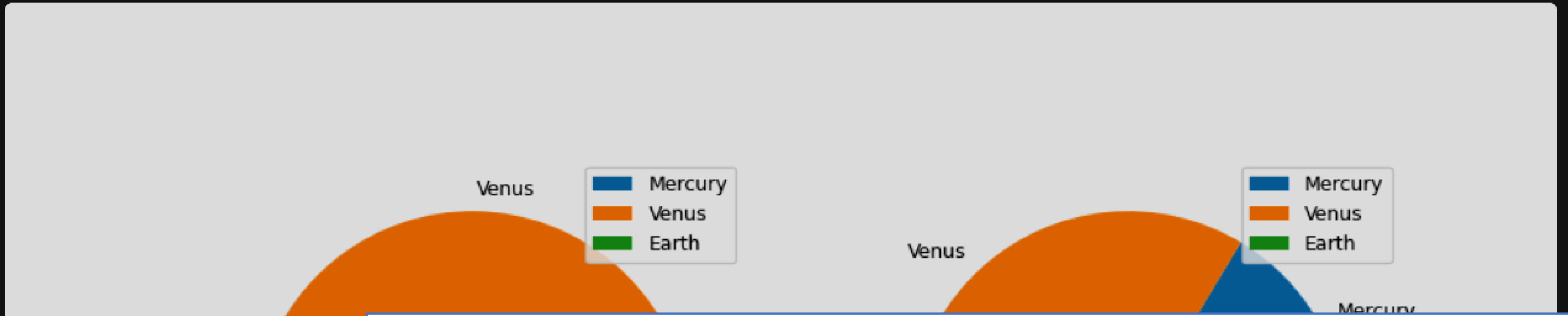
pandas.DataFrame.plot.pie

pandas.DataFrame.plot.scatter

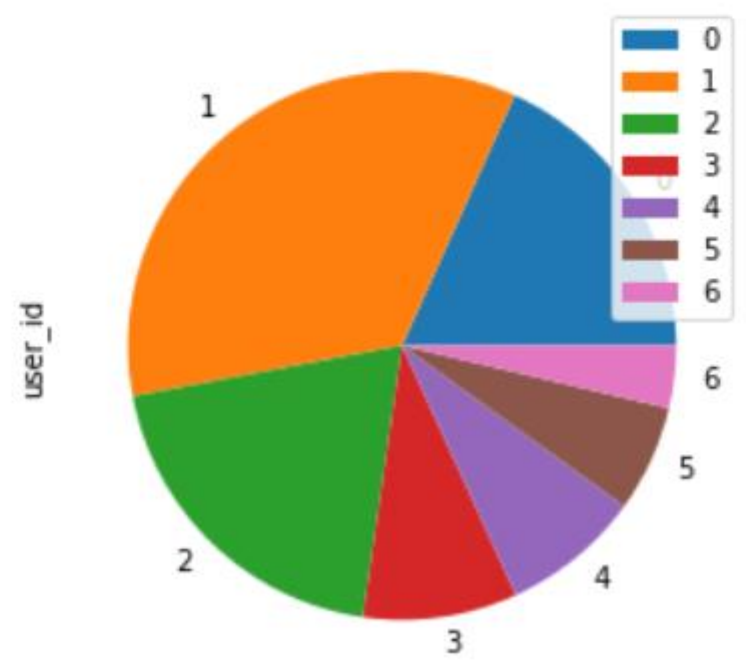
pandas.DataFrame.boxplot

pandas.DataFrame.hist

```
>>> plot = df.plot.pie(subplots=True, figsize=(11, 6))
```




```
1  plot = count_age.plot.pie(y = 'user_id', figsize = (4.5,4.5))
```



```
count_age_gender = users.groupby(by = ['age_desc','gender'])['user_id'].count().reset_index()
count_age_gender
```

	age_desc	gender	user_id
0	18-24	F	298
1	18-24	M	805
2	25-34	F	558
3	25-34	M	1538
4	35-44	F	338
5	35-44	M	855
6	45-49	F	189
7	45-49	M	361
8	50-55	F	146
9	50-55	M	350
10	56+	F	102
11	56+	M	278
12	Under 18	F	78
13	Under 18	M	144

```
print(count_age_gender.describe())
```

```
           user_id
count    14.000000
mean    431.428571
std     399.754100
min      78.000000
25%     156.750000
50%     318.000000
75%     508.750000
max    1538.000000
```