RapidsPY Documentation

Release 1.0

BorayData

May 12, 2022

CONTENTS:

1 安装 1
  1.1 安装前准备 . . . 1
  1.2 安装流程 . . . 2

2 快速开始 5
  2.1 PostgreSQL backend . . . 5
  2.2 MOXE backend . . . 21

3 API 参考 33
  3.1 RapidsPY . . . 33
  3.2 Input/output . . . 34
  3.3 General functions . . . 38
  3.4 Series . . . 48
  3.5 DataFrame . . . 120
  3.6 GroupBy . . . 206

4 发行说明 209
  4.1 Version 1.0 . . . 209

5 Indices and tables 213

Index 215

CHAPTER

ONE

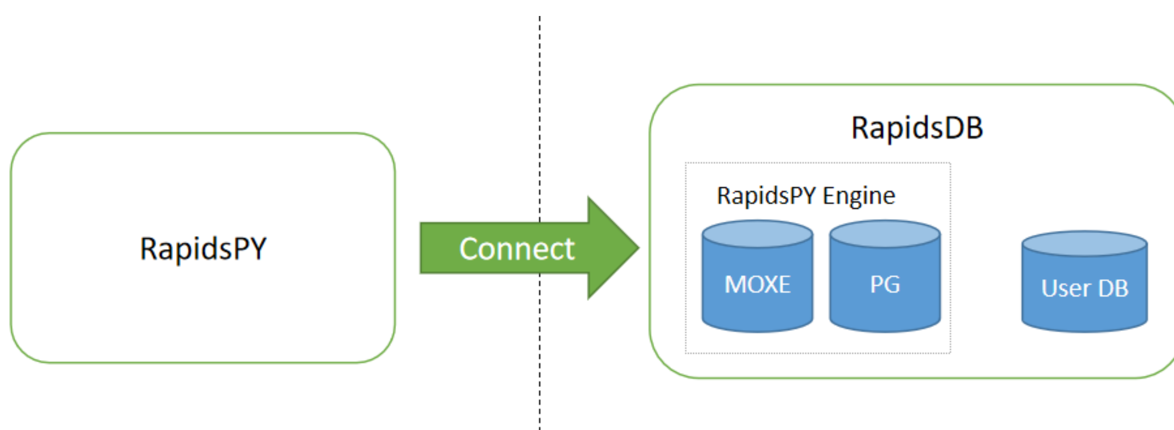
安装

1.1 安装前准备

1.1.1 环境需求

- RapidsDB >= 4.3
- MOXE >= 4.3
- PostgreSQL >= 12.7
- python >= 3.7
- OS 支持 Linux、Windows、Mac OS

1.1.2 部署图



上图是 RapidsPY 的部署图，通过该图可以看出，RapidsPY 需要连接到 RapidsDB，安装 RapidsPY 之前，请先保证你的 RapidsDB 是可用的。

RapidsPY 支持 MOXE 和 PostgreSQL 两种后端引擎，请至少配置一个 MOXE 或者 PostgreSQL 连接器。

1.1.3 配置 RapidsDB 连接器

通过 RapdisDB 语法，创建 MOXE 连接器或者 PostgreSQL 连接器（需要预先安装 PostgreSQL）。

```
rapids > CREATE CONNECTOR MOXE TYPE MOXE WITH PARTITIONS_PER_NODE=2, MEM_
↪PER_NODE='500gb' NODE *
```

（此处示例为创建 MOXE 连接器，如需创建 POSTGRES 连接器，请参考 RapidsDB 文档）

注意：创建 MOXE 连接器之后，需在 MOXE 中创建一张空表。

1.1.4 修改 RapidsDB 的 JVM 参数

打开 startDqx.sh 文件，找到 JVM_SETTINGS，修改-Xss 参数为 512M，修改-XX:MaxMetaspaceSize=512M

```
JVM_SETTINGS="-Xss512m -Xms8G -Xmx8G -XX:NewRatio=1 -XX:+PrintFlagsFinal"
JVM_SETTINGS="${JVM_SETTINGS} -XX:MaxInlineLevel=50 -XX:InlineSmallCode=16000 -
↪XX:ReservedCodeCacheSize=512M -XX:MaxMetaspaceSize=512M -XX:+PrintCodeCache"
```

1.2 安装流程

1.2.1 选择介质

根据操作系统版本和 Python 版本获取相应的介质包。

以 Linux 和 Python3.8 为例，选择介质：RapidsPY-1.0.0-cp38-cp38-linux_x86_64.whl。

其他依赖的介质有：pyRDP-4.0.0-py3-none-any.whl 和 sqlalchemy_RDP-2.0.0-py3-none-any.whl。

1.2.2 安装介质

```
$ pip install pyRDP-4.0.0-py3-none-any.whl

$ pip install sqlalchemy_RDP-2.0.0-py3-none-any.whl

$ pip install RapidsPY-1.0.0-cp38-cp38-linux_x86_64.whl
```

1.2.3 配置 connector.json 文件

找到 RapidsPY 的安装目录并进入

```
$ pip show RapidsPY

$ cd rapidspy
```

根据安装前准备中配置的 RapidsDB 连接器创建并配置 connectors.json 文件

```
{
    ”con_name1”: {
        ”con”: ”RDP://RAPIDS:rapids@HOST:PORT/CATALOG/SCHEMA”,
        ”connector_type”: ”MOXE”,
        ”description”: ”description”
    },
    ”con_name2”: {
        ”con”: ”RDP://RAPIDS:rapids@HOST:PORT/CATALOG/SCHEMA”,
        ”connector_type”: ”POSTGRES”,
        ”description”: ”description”
    }
}
```

举例如下，请根据实际环境进行修改

```
{
    ”rcon1”: {
        ”con”: ”RDP://RAPIDS:rapids@192.168.30.100:4333/MOXE/MOXE”,
        ”connector_type”: ”MOXE”,
        ”description”: ”RDP connector based on MOXE”
    },
    ”rcon2”: {
        ”con”: ”RDP://RAPIDS:rapids@192.168.30.100:4333/rapidspy/public”,
        ”connector_type”: ”POSTGRES”,
        ”description”: ”RDP connector based on PostgreSQL”
    }
}
```

1.2.4 检查是否安装成功

测试 RapidsPY 是否安装成功：

```
In [1]: from rapidspy import RapidsPYSession

In [2]: print(RapidsPYSession().con_list())
    name                      description
0  rcon1        RDP connector based on MOXE
1  rcon2  RDP connector based on PostgreSQL
```

CHAPTER

TWO

快速开始

这个页面介绍 RapidsPY 的简单使用。提供两个示例，分别是以 Postgres 为后端计算引擎和以 MOXE 为后端计算引擎。两个示例中所调用的接口有所不同。

2.1 PostgreSQL backend

以下是 RapidsPY 以 PG 为 backend engine 时常用命令的例子。在开始前，需配置好 RapidsDB 到 PG 的连接，并在 connectors.json 中写好 PG 的 connector 信息。

```
In [1]: import pandas as pd

In [2]: import matplotlib.pyplot as plt

In [3]: import warnings

In [4]: warnings.filterwarnings('ignore')

In [5]: import numpy as np

In [6]: import rapidspy as rdp

In [7]: from rapidspy import RapidsPYSession
```

配置 backend engine：填写 connectors.json 中配置的 PG 连接器的名称，例如 ‘‘rcon2‘‘

```
In [8]: rc = RapidsPYSession().configure('rcon2')
```

2.1.1 Create Object

From pandas

创建一个 pandas Series，注意 Series 需要设置 index 名称

再通过 rc.from_pandas 将 pandas Series 转为 RapidsPY 的 Series。通过 "compute" 生成一个 Series 并查看

```
In [9]: s1 = pd.Series([1, 3, 5, np.nan, 6, 8], name='a')

In [10]: s1.index.name='i'

In [11]: s1
Out[11]:
i
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
Name: a, dtype: float64

In [12]: rs1 = rc.from_pandas(s1, name='rs1',if_exists="replace")

In [13]: rs1.compute()
Out[13]:
i
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
Name: a, dtype: float64
```

创建一个 pandas DataFrame, 注意 DataFrame 也需要设置 index

```
In [14]: dates = pd.date_range("20130101", periods=6)

In [15]: df1 = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("abcd"))
```

(continues on next page)

(continued from previous page)

```
In [16]: df1 = df1.reindex(columns=list(df1.columns) + ["e"])

In [17]: df1.loc[dates[0]:dates[1], "e"] = 1

In [18]: df1.index.name='time'

In [19]: df1
Out[19]:
                   a         b         c         d    e
time
2013-01-01 -1.371337  0.404553  0.347884 -0.489973  1.0
2013-01-02 -0.625644 -0.138117  1.008382  0.628243  1.0
2013-01-03 -0.697443 -0.129793 -0.221877  0.288161  NaN
2013-01-04 -1.205701 -0.809986  0.012796 -1.501649  NaN
2013-01-05 -1.787924 -1.011623  0.707885  0.208272  NaN
2013-01-06 -0.820933 -0.703304 -1.372442  0.469276  NaN
```

再通过 rc.from_pandas 将 pandas DataFrame 转为 RapidsPY 的 DataFrame，此时需要设置写到数据库的表名称，例如 rdf1

```
In [20]: rdf1 = rc.from_pandas(df1, name='rdf1',if_exists="replace")

In [21]: rdf1.compute()
Out[21]:
                   a         b         c         d    e
time
2013-01-01 -1.371337  0.404553  0.347884 -0.489973  1.0
2013-01-02 -0.625644 -0.138117  1.008382  0.628243  1.0
2013-01-03 -0.697443 -0.129793 -0.221877  0.288161  NaN
2013-01-04 -1.205701 -0.809986  0.012796 -1.501649  NaN
2013-01-05 -1.787924 -1.011623  0.707885  0.208272  NaN
2013-01-06 -0.820933 -0.703304 -1.372442  0.469276  NaN
```

创建另外一个 DataFrame，以供后续使用

```
In [22]: df2 = pd.DataFrame(
   ....:     {
   ....:     "a": 1.0,
   ....:     "b": pd.Timestamp("20130102"),
   ....:     "c": pd.Series(1, index=list(range(4)), dtype="float32"),
```

(continues on next page)

(continued from previous page)

```
   ....:         "d": np.array([3] * 4, dtype="int32"),
   ....:         "e": pd.Categorical(["test", "train", "test", "train"]),
   ....:         "f": "foo",
   ....:     }
   ....: )
   ....:

In [23]: df2.index.name='index'

In [24]: df2
Out[24]:
         a          b    c  d      e    f
index
0      1.0 2013-01-02  1.0  3   test  foo
1      1.0 2013-01-02  1.0  3  train  foo
2      1.0 2013-01-02  1.0  3   test  foo
3      1.0 2013-01-02  1.0  3  train  foo

In [25]: rdf2 = rc.from_pandas(df2, name='rdf2', if_exists="replace")

In [26]: rdf2.compute()
Out[26]:
         a          b    c  d      e    f
index
0      1.0 2013-01-02  1.0  3   test  foo
1      1.0 2013-01-02  1.0  3  train  foo
2      1.0 2013-01-02  1.0  3   test  foo
3      1.0 2013-01-02  1.0  3  train  foo
```

可以看到，产生的 DataFrame 的数据有多种类型

```
In [27]: rdf2.dtypes
Out[27]:
a           float64
b    datetime64[ns]
c           float32
d             int32
e          category
f            object
dtype: object
```

将上面的 df1 数据导入到 RapidsDB 中。此例是写在了 RapidsDB 的 MySQL 连接中。用户可以根据实际情况修改连接串信息。

```
In [28]: MYSQL_CON ="RDP://RAPIDS:rapids@192.168.120.253:4333/mysql1/rapidspy"

In [29]: df1.to_sql(name='DF1', con=MYSQL_CON, if_exists="replace")
```

From database

将刚才写入到 MySQL 中的数据读取回来

```
In [30]: rdf11 = rc.read_sql_table("DF1", con=MYSQL_CON, index_col='time')

In [31]: rdf11
Out[31]:
Empty DataFrame
Columns: [a, b, c, d, e]
Index: []
```

2.1.2 Viewing Data

查看数据的前 5 行。如果不输入 compute，则只能看到一个空的 DataFrame

```
In [32]: rdf11.head().compute()
Out[32]:
                   a         b         c         d    e
time
2013-01-01 -1.371337  0.404553  0.347884 -0.489973  1.0
2013-01-02 -0.625644 -0.138117  1.008382  0.628243  1.0
2013-01-03 -0.697443 -0.129793 -0.221877  0.288161  NaN
2013-01-04 -1.205701 -0.809986  0.012796 -1.501649  NaN
2013-01-05 -1.787924 -1.011623  0.707885  0.208272  NaN
```

查看数据的 index

```
In [33]: rdf11.index.to_pandas()
Out[33]:
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', name='time', freq=None)
```

查看数据的列名

```
In [34]: rdf11.columns
Out[34]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

查看数据框的值，此操作暗含了 compute

```
In [35]: rdf11.values
Out[35]:
array([[-1.3713374 ,  0.40455272,  0.3478842 , -0.48997282,  1.        ],
       [-0.62564423, -0.13811734,  1.00838161,  0.62824333,  1.        ],
       [-0.69744318, -0.12979301, -0.22187671,  0.28816141,         nan],
       [-1.2057012 , -0.80998595,  0.012796  , -1.50164924,         nan],
       [-1.78792437, -1.01162295,  0.70788467,  0.20827211,         nan],
       [-0.82093343, -0.70330424, -1.37244232,  0.46927616,         nan]])
```

查看数据的描述

```
In [36]: rdf11.describe().compute()
Out[36]:
              a         b         c         d    e
count  6.000000  6.000000  6.000000  6.000000  2.0
mean  -1.084831 -0.398045  0.080438 -0.066278  1.0
std    0.452018  0.533487  0.840610  0.801248  0.0
min   -1.787924 -1.011623 -1.372442 -1.501649  1.0
25%   -1.371337 -0.809986 -0.221877 -0.489973  1.0
50%   -1.205701 -0.703304  0.012796  0.208272  1.0
75%   -0.697443 -0.129793  0.707885  0.469276  1.0
max   -0.625644  0.404553  1.008382  0.628243  1.0
```

对数据进行排序

```
In [37]: rdf12 = rdf11.sort_values(by='a')

In [38]: rdf12.compute()
Out[38]:
                   a         b         c         d    e
time
2013-01-05 -1.787924 -1.011623  0.707885  0.208272  NaN
2013-01-01 -1.371337  0.404553  0.347884 -0.489973  1.0
2013-01-04 -1.205701 -0.809986  0.012796 -1.501649  NaN
2013-01-06 -0.820933 -0.703304 -1.372442  0.469276  NaN
```

(continues on next page)

(continued from previous page)

```
2013-01-03 -0.697443 -0.129793 -0.221877  0.288161  NaN
2013-01-02 -0.625644 -0.138117  1.008382  0.628243  1.0
```

```
In [39]: rdf12.sort_index().compute()
Out[39]: 
                   a         b         c         d    e
time                                                   
2013-01-01 -1.371337  0.404553  0.347884 -0.489973  1.0
2013-01-02 -0.625644 -0.138117  1.008382  0.628243  1.0
2013-01-03 -0.697443 -0.129793 -0.221877  0.288161  NaN
2013-01-04 -1.205701 -0.809986  0.012796 -1.501649  NaN
2013-01-05 -1.787924 -1.011623  0.707885  0.208272  NaN
2013-01-06 -0.820933 -0.703304 -1.372442  0.469276  NaN
```

2.1.3 Selection

Getting

Selecting a single column, which yields a Series

```
In [40]: rdf11['a']
Out[40]: Series([], Name: a, dtype: float64)
```

Selection by Label

```
In [41]: rdf2.loc[1].compute()
Out[41]: 
         a          b    c  d      e    f
index                                    
1      1.0 2013-01-02  1.0  3  train  foo
```

```
In [42]: rdf2.loc[:,['a', 'b']].compute()
Out[42]: 
         a          b
index                
0      1.0 2013-01-02
1      1.0 2013-01-02
```

(continues on next page)

(continued from previous page)

```
2       1.0 2013-01-02
3       1.0 2013-01-02
```

```
In [43]: rdf2.loc[1:3,'e'].compute()
Out[43]:
index
0     test
1    train
2     test
3    train
Name: e, dtype: category
Categories (2, object): ['test', 'train']
```

Selection by Position

Select via the position of the passed integers:

```
In [44]: rdf11.iloc[3].compute()
Out[44]:
                   a         b         c         d   e
time
2013-01-04 -1.205701 -0.809986  0.012796 -1.501649 NaN
```

By integer slices, acting similar to NumPy/Python:

```
In [45]: rdf11.iloc[3:5, 0:2].compute()
Out[45]:
                   a         b
time
2013-01-04 -1.205701 -0.809986
2013-01-05 -1.787924 -1.011623
```

By lists of integer position locations, similar to the NumPy/Python style:

```
In [46]: rdf11.iloc[[1,2,4], [0,2]].compute()
Out[46]:
                   a         c
time
2013-01-02 -0.625644  1.008382
```

(continues on next page)

(continued from previous page)

```
2013-01-03 -0.697443 -0.221877
2013-01-05 -1.787924  0.707885
```

For slicing rows explicitly:

```
In [47]: rdf11.iloc[1:3, :].compute()
Out[47]:
                   a         b         c         d    e
time
2013-01-02 -0.625644 -0.138117  1.008382  0.628243  1.0
2013-01-03 -0.697443 -0.129793 -0.221877  0.288161  NaN
```

For slicing columns explicitly:

```
In [48]: rdf11.iloc[:, 1:3].compute()
Out[48]:
                   b         c
time
2013-01-01  0.404553  0.347884
2013-01-02 -0.138117  1.008382
2013-01-03 -0.129793 -0.221877
2013-01-04 -0.809986  0.012796
2013-01-05 -1.011623  0.707885
2013-01-06 -0.703304 -1.372442
```

Boolean Indexing

```
In [49]: rdf11[rdf11['a']>0].compute()
Out[49]:
Empty DataFrame
Columns: [a, b, c, d, e]
Index: []
```

Setting

Setting a new column automatically aligns the data by the indexes

```
In [50]: rdf11.at[:,['a','c']].compute()
Out[50]:
                   a         c
time
2013-01-01 -1.371337  0.347884
2013-01-02 -0.625644  1.008382
2013-01-03 -0.697443 -0.221877
2013-01-04 -1.205701  0.012796
2013-01-05 -1.787924  0.707885
2013-01-06 -0.820933 -1.372442
```

```
In [51]: rdf11 = rdf11.assign(d=5)

In [52]: rdf11.compute()
Out[52]:
                   a         b         c  d    e
time
2013-01-01 -1.371337  0.404553  0.347884  5  1.0
2013-01-02 -0.625644 -0.138117  1.008382  5  1.0
2013-01-03 -0.697443 -0.129793 -0.221877  5  NaN
2013-01-04 -1.205701 -0.809986  0.012796  5  NaN
2013-01-05 -1.787924 -1.011623  0.707885  5  NaN
2013-01-06 -0.820933 -0.703304 -1.372442  5  NaN
```

Missing data

```
In [53]: rdp.isna(rdf11).compute()
Out[53]:
                a      b      c      d      e
time
2013-01-01  False  False  False  False  False
2013-01-02  False  False  False  False  False
2013-01-03  False  False  False  False   True
2013-01-04  False  False  False  False   True
2013-01-05  False  False  False  False   True
2013-01-06  False  False  False  False   True
```

```
In [54]: rdf13 = rdf11.dropna()

In [55]: rdf13.compute()
Out[55]:
                   a         b         c  d    e
time
2013-01-01 -1.371337  0.404553  0.347884  5  1.0
2013-01-02 -0.625644 -0.138117  1.008382  5  1.0
```

2.1.4 Operations

Stats

Operations in general exclude missing data.

Performing a descriptive statistic:

```
In [56]: rdf11.mean().compute()
Out[56]:
a   -1.084831
b   -0.398045
c    0.080438
d    5.000000
e    1.000000
dtype: float64
```

```
In [57]: rdf11.sub(1).compute()
Out[57]:
                   a         b         c  d    e
time
2013-01-01 -2.371337 -0.595447 -0.652116  4  0.0
2013-01-02 -1.625644 -1.138117  0.008382  4  0.0
2013-01-03 -1.697443 -1.129793 -1.221877  4  NaN
2013-01-04 -2.205701 -1.809986 -0.987204  4  NaN
2013-01-05 -2.787924 -2.011623 -0.292115  4  NaN
2013-01-06 -1.820933 -1.703304 -2.372442  4  NaN
```

```
In [58]: (rdf11['d']+1).compute()
Out[58]:
time
```

(continues on next page)

(continued from previous page)

```
2013-01-01    6
2013-01-02    6
2013-01-03    6
2013-01-04    6
2013-01-05    6
2013-01-06    6
Name: d, dtype: int64
```

Histogramming

```
In [59]: s2 = pd.Series(np.random.randint(0, 7, size=10), name='a')
```

```
In [60]: s2
Out[60]:
0    1
1    4
2    2
3    0
4    1
5    4
6    2
7    3
8    4
9    4
Name: a, dtype: int64
```

```
In [61]: rs2 = rc.from_pandas(s2, name='rs2',if_exists="replace")

In [62]: rs2.value_counts().compute()
Out[62]:
4    4
2    2
1    2
3    1
0    1
Name: a, dtype: int64
```

2.1.5 Merge

```
In [63]: left = pd.DataFrame({"key":["foo", "foo"], "lval":[1, 2]})

In [64]: right = pd.DataFrame({"key":["foo", "foo"], "rval":[4, 5]})

In [65]: left.index.name='index'

In [66]: right.index.name='index'

In [67]: left
Out[67]:
      key  lval
index
0      foo     1
1      foo     2

In [68]: right
Out[68]:
      key  rval
index
0      foo     4
1      foo     5

In [69]: rleft = rc.from_pandas(left, name='left_t',if_exists="replace")

In [70]: rright = rc.from_pandas(right, name='right_t',if_exists="replace")

In [71]: rleft.merge(rright, left_on='key', right_on='key').compute()
Out[71]:
   lval  key  rval
0     1  foo     4
1     1  foo     5
2     2  foo     4
3     2  foo     5
```

2.1.6 Grouping

```
In [72]: df3 = pd.DataFrame(
   ....:     {
   ....:         "a": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
   ....:         "b": ["one", "one", "two", "three", "two", "two", "one", "three"],
   ....:         "c": np.random.randn(8),
   ....:         "d": np.random.randn(8),
   ....:     }
   ....: )
   ....:
```

```
In [73]: df3.index.name='index'

In [74]: rdf3 = rc.from_pandas(df3, name='rdf3')

In [75]: rdf3.compute()
Out[75]:
         a      b         c         d
index
0      foo    one  0.253474 -0.271280
1      bar    one -1.071936 -0.419491
2      foo    two  0.264619 -0.238317
3      bar  three -0.275381  1.836672
4      foo    two -0.019413  2.338473
5      bar    two  1.396420  1.404539
6      foo    one -0.479722  1.264027
7      foo  three  1.011864  1.051258
```

```
In [76]: rdf3.groupby("a").sum().compute()
Out[76]:
            c        d
a
bar  0.049103  2.82172
foo  1.030821  4.14416
```

```
In [77]: rdf3.groupby(["a", "b"]).sum().compute()
Out[77]:
                  c         d
a   b
```

(continues on next page)

(continued from previous page)

```
bar one   -1.071936 -0.419491
    three -0.275381  1.836672
    two    1.396420  1.404539
foo one   -0.226248  0.992747
    three  1.011864  1.051258
    two    0.245206  2.100155
```

2.1.7 Time Series

```
In [78]: ts = rdf11.reset_index()['time'].compute()

In [79]: ts
Out[79]:
0    2013-01-01 00:00:00
1    2013-01-02 00:00:00
2    2013-01-03 00:00:00
3    2013-01-04 00:00:00
4    2013-01-05 00:00:00
5    2013-01-06 00:00:00
Name: time, dtype: object
```

```
In [80]: ts = ts.to_datetime()

In [81]: ts.dt.day.compute()
Out[81]:
0    1
1    2
2    3
3    4
4    5
5    6
Name: time, dtype: int64
```

```
In [82]: ts.dt.year.compute()
Out[82]:
0    2013
1    2013
2    2013
```

(continues on next page)

(continued from previous page)

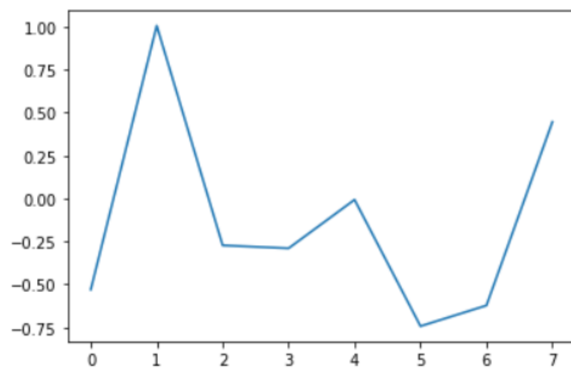
```
3     2013
4     2013
5     2013
Name: time, dtype: int64

In [83]: rc.close()
```

2.1.8 Plotting

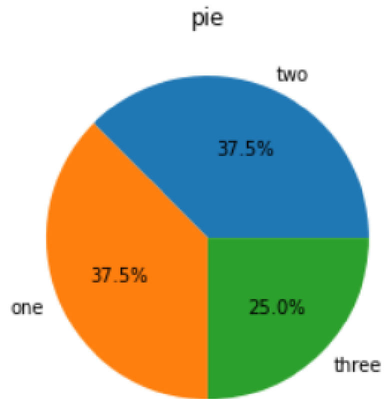
```
In [84]: plt.plot(rdf3['c'].values)

In [85]: plt.show()
```




```
In [86]: pie_rdf = rdf3['b'].value_counts()
In [87]: pie_rdf
Out [87]:
Series([], Name: b, dtype: int64)

In [88]: plt.pie(x=pie_rdf.values, labels=pie_rdf.to_pandas().index, autopct='%1.1f%%')
In [89]: plt.title("pie")
In [90]: plt.show()
```



2.2 MOXE backend

以下是 RapidsPY 以 MOXE 为 backend engine 时常用命令的例子。在开始前，需配置好 RapidsDB 到 MOXE 的连接，并在 connectors.json 中写好 MOXE 的 connector 信息。

```
In [1]: import pandas as pd

In [2]: import matplotlib.pyplot as plt

In [3]: import warnings

In [4]: warnings.filterwarnings('ignore')

In [5]: import numpy as np

In [6]: import rapidspy as rdp

In [7]: from rapidspy import RapidsPYSession
```

配置 backend engine：填写 connectors.json 中配置的 MOXE 连接器的名称，例如 ‘‘rcon1‘‘

```
In [8]: rc = RapidsPYSession().configure('rcon1')
```

2.2.1 Create Object

From pandas

创建一个 pandas Series，注意 Series 需要设置 index 名称

再通过 rc.from_pandas 将 pandas Series 转为 RapidsPY 的 Series。通过 "compute"'生成一个 Series 并查看

```
In [9]: s = pd.Series(np.random.randint(0, 7, size=10), name='A')

In [10]: s.index.name='i'

In [11]: rs = rc.from_pandas(s, name='rs122')

In [12]: rs.compute()
Out[12]:
i
0    6
1    5
2    2
3    2
4    2
5    2
6    3
7    6
8    0
9    3
Name: A, dtype: int64
```

创建一个 pandas DataFrame, 注意 DataFrame 也需要设置 index

```
In [13]: dates = pd.date_range("20130101", periods=6)

In [14]: df = pd.DataFrame(np.random.randn(6, 4), index=dates, columns=list("ABCD"))

In [15]: df.index.name='TIME'
```

```
In [16]: df1 = df.reindex(columns=list(df.columns) + ["E"])

In [17]: df1.loc[dates[0] : dates[1], "E"] = 1
```

(continues on next page)

(continued from previous page)

```
In [18]: df1
Out[18]:
                   A         B         C         D    E
TIME
2013-01-01  0.229164  1.779358 -0.788180 -0.109822  1.0
2013-01-02 -0.449375  0.610758  0.706601  0.212569  1.0
2013-01-03  0.732276 -0.608776 -1.370000 -1.015765  NaN
2013-01-04  0.523726 -0.985897  0.856339 -0.404535  NaN
2013-01-05 -0.795703 -1.686420 -2.197842 -0.788638  NaN
2013-01-06  0.458185 -0.715032 -0.727784  0.941248  NaN

In [19]: rdf1 = rc.from_pandas(df1, name='RDF3')

In [20]: rdf1.compute()
Out[20]:
                   A         B         C         D    E
TIME
2013-01-01  0.229164  1.779358 -0.788180 -0.109822  1.0
2013-01-02 -0.449375  0.610758  0.706601  0.212569  1.0
2013-01-03  0.732276 -0.608776 -1.370000 -1.015765  NaN
2013-01-04  0.523726 -0.985897  0.856339 -0.404535  NaN
2013-01-05 -0.795703 -1.686420 -2.197842 -0.788638  NaN
2013-01-06  0.458185 -0.715032 -0.727784  0.941248  NaN
```

数据准备，将上面的 df 数据导入到 RDP 中

```
In [21]: MYSQL_CON ="RDP://RAPIDS:rapids@192.168.120.253:4333/mysql1/rapidspy"

In [22]: df1.to_sql(name='DF', con=MYSQL_CON,if_exists="replace")
```

From database

将刚才写入到 mysql 中的数据读取回来

```
In [23]: rdf = rc.read_sql_table("DF", con=MYSQL_CON, index_col='TIME')

In [24]: rdf
Out[24]:
Empty DataFrame
```

(continues on next page)

(continued from previous page)

```
Columns: [A, B, C, D, E]
Index: []
```

2.2.2 Viewing Data

```
In [25]: rdf.head().compute()
Out[25]:
                   A         B         C         D    E
TIME
2013-01-01  0.229164  1.779358 -0.788180 -0.109822  1.0
2013-01-02 -0.449375  0.610758  0.706601  0.212569  1.0
2013-01-03  0.732276 -0.608776 -1.370000 -1.015765  NaN
2013-01-04  0.523726 -0.985897  0.856339 -0.404535  NaN
2013-01-05 -0.795703 -1.686420 -2.197842 -0.788638  NaN

In [26]: rdf.index
Out[26]: <rapidspy.index> object, name: TIME

In [27]: rdf.columns
Out[27]: Index(['A', 'B', 'C', 'D', 'E'], dtype='object')

In [28]: rdf.values
Out[28]:
array([[ 0.22916354,  1.77935777, -0.7881798 , -0.10982218,  1.        ],
       [-0.44937543,  0.61075786,  0.70660102,  0.21256934,  1.        ],
       [ 0.73227641, -0.60877565, -1.37000031, -1.01576526,         nan],
       [ 0.52372618, -0.98589664,  0.8563392 , -0.40453452,         nan],
       [-0.79570345, -1.68642019, -2.1978423 , -0.78863768,         nan],
       [ 0.45818531, -0.71503161, -0.72778401,  0.94124769,         nan]])

In [29]: rdf.dtypes
Out[29]:
A    float64
B    float64
C    float64
D    float64
E    float64
dtype: object
```

(continues on next page)

(continued from previous page)

```
In [30]: rdf.sort_values(by='A').compute()
Out[30]:
                   A         B         C         D    E
TIME
2013-01-05 -0.795703 -1.686420 -2.197842 -0.788638  NaN
2013-01-02 -0.449375  0.610758  0.706601  0.212569  1.0
2013-01-01  0.229164  1.779358 -0.788180 -0.109822  1.0
2013-01-06  0.458185 -0.715032 -0.727784  0.941248  NaN
2013-01-04  0.523726 -0.985897  0.856339 -0.404535  NaN
2013-01-03  0.732276 -0.608776 -1.370000 -1.015765  NaN
```

2.2.3 Selection

Getting

Selecting a single column, which yields a Series

```
In [31]: rdf['A']
Out[31]: Series([], Name: A, dtype: float64)
```

Selection by Label

```
In [32]: rdf.loc[:,['A', 'C']].compute()
Out[32]:
                   A         C
TIME
2013-01-01  0.229164 -0.788180
2013-01-02 -0.449375  0.706601
2013-01-03  0.732276 -1.370000
2013-01-04  0.523726  0.856339
2013-01-05 -0.795703 -2.197842
2013-01-06  0.458185 -0.727784
```

Selection by Position

```
In [33]: rdf.iloc[:, 0:3].compute()
Out[33]:
                   A         B         C
TIME
2013-01-01  0.229164  1.779358 -0.788180
2013-01-02 -0.449375  0.610758  0.706601
2013-01-03  0.732276 -0.608776 -1.370000
2013-01-04  0.523726 -0.985897  0.856339
2013-01-05 -0.795703 -1.686420 -2.197842
2013-01-06  0.458185 -0.715032 -0.727784
```

Boolean Indexing

```
In [34]: rdf[rdf['A']>0].compute()
Out[34]:
                   A         B         C         D    E
TIME
2013-01-01  0.229164  1.779358 -0.788180 -0.109822  1.0
2013-01-03  0.732276 -0.608776 -1.370000 -1.015765  NaN
2013-01-04  0.523726 -0.985897  0.856339 -0.404535  NaN
2013-01-06  0.458185 -0.715032 -0.727784  0.941248  NaN
```

Setting

Setting a new column automatically aligns the data by the indexes

```
In [35]: rdf.at[:,'A'].compute()
Out[35]:
TIME
2013-01-01    0.229164
2013-01-02   -0.449375
2013-01-03    0.732276
2013-01-04    0.523726
2013-01-05   -0.795703
2013-01-06    0.458185
Name: A, dtype: float64
```

(continues on next page)

(continued from previous page)

```
In [36]: rdf = rdf.assign(D=5)

In [37]: rdf.compute()
Out[37]:
                   A         B         C  D    E
TIME
2013-01-01  0.229164  1.779358 -0.788180  5  1.0
2013-01-02 -0.449375  0.610758  0.706601  5  1.0
2013-01-03  0.732276 -0.608776 -1.370000  5  NaN
2013-01-04  0.523726 -0.985897  0.856339  5  NaN
2013-01-05 -0.795703 -1.686420 -2.197842  5  NaN
2013-01-06  0.458185 -0.715032 -0.727784  5  NaN
```

Missing data

```
In [38]: rdp.isna(rdf).compute()
Out[38]:
                A      B      C      D      E
TIME
2013-01-01  False  False  False  False  False
2013-01-02  False  False  False  False  False
2013-01-03  False  False  False  False   True
2013-01-04  False  False  False  False   True
2013-01-05  False  False  False  False   True
2013-01-06  False  False  False  False   True

In [39]: rdf1 = rdf.dropna()

In [40]: rdf1.compute()
Out[40]:
                   A         B         C  D    E
TIME
2013-01-01  0.229164  1.779358 -0.788180  5  1.0
2013-01-02 -0.449375  0.610758  0.706601  5  1.0
```

2.2.4 Operations

Stats

```
In [41]: rdf.mean().compute()
Out[41]:
A    0.116379
B   -0.267668
C   -0.586811
D    5.000000
E    1.000000
dtype: float64

In [42]: rdf.sub(1).compute()
Out[42]:
                   A         B         C  D    E
TIME
2013-01-01 -0.770836  0.779358 -1.788180  4  0.0
2013-01-02 -1.449375 -0.389242 -0.293399  4  0.0
2013-01-03 -0.267724 -1.608776 -2.370000  4  NaN
2013-01-04 -0.476274 -1.985897 -0.143661  4  NaN
2013-01-05 -1.795703 -2.686420 -3.197842  4  NaN
2013-01-06 -0.541815 -1.715032 -1.727784  4  NaN

In [43]: rdf.compute()
Out[43]:
                   A         B         C  D    E
TIME
2013-01-01  0.229164  1.779358 -0.788180  5  1.0
2013-01-02 -0.449375  0.610758  0.706601  5  1.0
2013-01-03  0.732276 -0.608776 -1.370000  5  NaN
2013-01-04  0.523726 -0.985897  0.856339  5  NaN
2013-01-05 -0.795703 -1.686420 -2.197842  5  NaN
2013-01-06  0.458185 -0.715032 -0.727784  5  NaN

In [44]: (rdf['A']+1).compute()
Out[44]:
TIME
2013-01-01    1.229164
2013-01-02    0.550625
2013-01-03    1.732276
```

(continues on next page)

(continued from previous page)

```
2013-01-04    1.523726
2013-01-05    0.204297
2013-01-06    1.458185
Name: A, dtype: float64
```

Histogramming

```
In [45]: rs.value_counts().compute()
Out[45]:
2    4
3    2
6    2
0    1
5    1
Name: A, dtype: int64
```

2.2.5 Merge

```
In [46]: left = rdf1.iloc[:,0:3]

In [47]: left.compute()
Out[47]:
                   A         B         C
TIME
2013-01-01  0.229164  1.779358 -0.788180
2013-01-02 -0.449375  0.610758  0.706601

In [48]: right = rdf1.iloc[:,2:5]

In [49]: right.compute()
Out[49]:
                   C  D    E
TIME
2013-01-01 -0.788180  5  1.0
2013-01-02  0.706601  5  1.0

In [50]: merge = left.merge(right,left_index=True, right_index=True)
```

(continues on next page)

(continued from previous page)

```
In [51]: merge.compute()
Out[51]:
                   A         B       C_x       C_y  D    E
TIME
2013-01-01  0.229164  1.779358 -0.788180 -0.788180  5  1.0
2013-01-02 -0.449375  0.610758  0.706601  0.706601  5  1.0
```

2.2.6 Grouping

```
In [52]: df2 = pd.DataFrame(
   ....:     {
   ....:         "A": ["foo", "bar", "foo", "bar", "foo", "bar", "foo", "foo"],
   ....:         "B": ["one", "one", "two", "three", "two", "two", "one", "three"],
   ....:         "C": np.random.randn(8),
   ....:         "D": np.random.randn(8),
   ....:     }
   ....: )
   ....:

In [53]: df2.index.name='INDEX'

In [54]: rdf2 = rc.from_pandas(df2, name='RDF2')

In [55]: rdf2.compute()
Out[55]:
         A      B         C         D
INDEX
0      foo    one  0.630174  0.789647
1      bar    one -1.863595  0.652930
2      foo    two -3.763735 -0.493473
3      bar  three -0.326244 -1.351434
4      foo    two -0.236953  1.465651
5      bar    two -0.155481 -0.193201
6      foo    one -0.614765  0.853663
7      foo  three  0.318818  0.092858

In [56]: rdf2.groupby("A").sum().compute()
Out[56]:
            C         D
```

(continues on next page)

(continued from previous page)

```
A
bar -2.34532 -0.891705
foo -3.66646  2.708345

In [57]: rdf2.groupby(["A", "B"]).sum().compute()
Out[57]:
                  C         D
A   B
bar one   -1.863595  0.652930
    three -0.326244 -1.351434
    two   -0.155481 -0.193201
foo one    0.015409  1.643309
    three  0.318818  0.092858
    two   -4.000688  0.972178

In [58]: rc.close()
```

CHAPTER

THREE

API 参考

这个页面提供了 RapidsPY API 的注释。每个 API 注释介绍了该 API 的功能、参数描述、返回值描述和示例。

API 的注释来源于 Pandas API reference。其中，示例是采用 Pandas API Reference 中的示例，使用时请参考实际的参数描述和返回值描述。

3.1 RapidsPY

3.1.1 Session

RapidsPYSession.con_list()	Return the connnector info in connectors.json file.
RapidsPYSession.configure(connector_name)	Connect to the RapidsPY Backend Engine.
RapidsPYSession.close()	Close the RapisPY Session.

rapidspy.RapidsPYSession.con_list

RapidsPYSession.con_list()

Return the connnector info in connectors.json file.

rapidspy.RapidsPYSession.configure

RapidsPYSession.configure(connector_name)

Connect to the RapidsPY Backend Engine.

Parameters

connector_name [str] Connector name in connectors.json file.

Returns

DataFrame RapidsPY Sessionn.

rapidspy.RapidsPYSession.close

RapidsPYSession.close()

Close the RapisPY Session.

3.2 Input/output

3.2.1 SQL

RapidsPYSession.read_sql_table(table_name, con)	Read SQL database table into a DataFrame.
RapidsPYSession.read_sql_query(sql, con[, ...])	Read SQL query into a DataFrame.
RapidsPYSession.read_sql(sql, con[, ...])	Read SQL query or database table into a DataFrame.
RapidsPYSession.from_pandas(frame, name[, ...])	Converts the existing DataFrame into a RapidsPY DataFrame.

rapidspy.RapidsPYSession.read_sql_table

RapidsPYSession.read_sql_table(table_name, con, index_col=None, columns=None, schema=None)

Read SQL database table into a DataFrame.

Given a table name and a SQLAlchemy connectable, returns a DataFrame. This function does not support DBAPI connections.

Parameters

table_name [str] Name of SQL table in database.

con [str] A database URI could be provided as str, for example "RDP://RAPIDS:rapids@192.168.120.253:4333/MOXE/MOXE" . Format is RDP://username:password@host:port/catalog/schema

index_col [str or list of str, optional(mandatory when RapidsPy] Engine is MOXE). default: None. Column(s) to set as index(MultiIndex).

columns [list, default None] List of column names to select from SQL table.

schema [str, default None] Name of SQL schema in database to query (if database flavor supports this). Uses default schema if None (default).

Returns

DataFrame A SQL table is returned as two-dimensional data structure with labeled axes.

Notes

Any datetime values with time zone information will be converted to UTC.

Examples

```
>>> MYSQL_CON ="RDP://RAPIDS:rapids@192.168.120.253:4333/rapidspy/rapidspy"
>>> rc.read_sql_table('table_name', con=MYSQL_CON, index_col='mta_tax')
```

rapidspy.RapidsPYSession.read_sql_query

RapidsPYSession.read_sql_query(sql, con, index_col=None)

Read SQL query into a DataFrame.

Returns a DataFrame corresponding to the result set of the query string. Optionally provide an index_col parameter to use one of the columns as the index, otherwise default integer index will be used.

Parameters

sql [str SQL query] SQL query to be executed.

con [str] A database URI could be provided as str, for example "RDP://RAPIDS:rapids@192.168.120.253:4333/MOXE/MOXE" . Format is RDP://username:password@host:port/catalog/schema

index_col [str or list of str, optional(mandatory when RapidsPy] Engine is MOXE). default: None. Column(s) to set as index(MultiIndex).

Returns

DataFrame

Notes

Any datetime values with time zone information parsed via the parse_dates parameter will be converted to UTC.

rapidspy.RapidsPYSession.read_sql

RapidsPYSession.read_sql(sql, con, index_col=None, columns=None)

Read SQL query or database table into a DataFrame.

This function is a convenience wrapper around read_sql_table and read_sql_query (for backward compatibility). It will delegate to the specific function depending on the provided input. A SQL query will be routed to read_sql_query, while a database table name will be routed to read_sql_table. Note that the delegated function might have more specific notes about their functionality not listed here.

Parameters

sql [str] SQL query to be executed or a table name.

con [str] A database URI could be provided as str, for example "RDP://RAPIDS:rapids@192.168.120.253:4333/MOXE/MOXE" . Format is RDP://username:password@host:port/catalog/schema

index_col [str or list of str, optional(mandatory when RapidsPy] Engine is MOXE). default: None. Column(s) to set as index(MultiIndex).

columns [list, default: None] List of column names to select from SQL table (only used when reading a table).

Returns

DataFrame

rapidspy.RapidsPYSession.from_pandas

RapidsPYSession.from_pandas(frame, name, schema=None, if_exists='fail', chunksize=None, dtype=None, method=None)

Converts the existing DataFrame into a RapidsPY DataFrame.

Parameters

frame: pandas Object pandas DataFrame or pandas Series.

name: str table name in RapidsDB.

schema: str, optional Specify the schema (if database flavor supports this). If None, use default schema.

schema [str, optional] Name of SQL schema in database to write to (if database flavor supports this). If None, use default schema (default).

if_exists [{ 'fail' , 'replace' , 'append' }, default 'fail']

- fail: If table exists, do nothing.
- replace: If table exists, drop it, recreate it, and insert data.
- append: If table exists, insert data. Create if does not exist.

chunksize [int, optional] Specify the number of rows in each batch to be written at a time. By default, all rows will be written at once.

dtype [dict or scalar, optional] Specifying the datatype for columns. If a dictionary is used, the keys should be the column names and the values should be the SQLAlchemy types or strings for the sqlite3 fallback mode. If a scalar is provided, it will be applied to all columns.

method [{None, 'multi' , callable()}, optional] Controls the SQL insertion clause used:

- None : Uses standard SQL INSERT clause (one per row).
- 'multi' : Pass multiple values in a single INSERT clause.
- callable with signature (pd_table, conn, keys, data_iter).

Details and a sample callable implementation can be found in the section insert method.

Returns

Series or DataFrame

Examples

```
>>> s = pd.Series([1, 3, 5, np.nan, 6, 8], name='a')
>>> s.index.name='i'
>>> s
i
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
Name: a, dtype: float64
```

(continues on next page)

(continued from previous page)

```
>>> rs = rc.from_pandas(s, name='rs')
>>> rs.compute()
i
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
Name: a, dtype: float64
```

3.3 General functions

3.3.1 Data manipulations

cut(x, bins[, labels])	Bin values into discrete intervals.

3.3.2 Top-level missing data

isna(obj)	Detect missing values for an array-like object.
isnull(obj)	Detect missing values for an array-like object.
notna(obj)	Detect non-missing values for an array-like object.
notnull(obj)	Detect non-missing values for an array-like object.

rapidspy.isna

rapidspy.isna(obj)

Detect missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are missing (NaN in numeric arrays, None or NaN in object arrays, NaT in datetimelike).

Parameters

obj [scalar or array_like] Object to check for null or missing values.

Returns

bool or array_like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is missing.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.isna('dog')
False
```

```
>>> pd.isna(pd.NA)
True
```

```
>>> pd.isna(np.nan)
True
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.isna(array)
array([[False,  True, False],
       [False, False,  True]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.isna(index)
array([False, False,  True, False])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
     0     1    2
```

(continues on next page)

(continued from previous page)

```
0  ant   bee  cat
1  dog  None  fly
>>> pd.isna(df)
       0      1      2
0  False  False  False
1  False   True  False
```

```
>>> pd.isna(df[1])
0    False
1     True
Name: 1, dtype: bool
```

rapidspy.isnull

rapidspy.isnull(obj)

Detect missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are missing (NaN in numeric arrays, None or NaN in object arrays, NaT in datetimelike).

Parameters

obj [scalar or array_like] Object to check for null or missing values.

Returns

bool or array_like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is missing.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.isna('dog')
False
```

```
>>> pd.isna(pd.NA)
True
```

```
>>> pd.isna(np.nan)
True
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.isna(array)
array([[False,  True, False],
       [False, False,  True]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.isna(index)
array([False, False,  True, False])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
     0     1    2
0  ant   bee  cat
1  dog  None  fly
>>> pd.isna(df)
       0      1      2
0  False  False  False
1  False   True  False
```

```
>>> pd.isna(df[1])
0    False
1     True
Name: 1, dtype: bool
```

rapidspy.notna

rapidspy.notna(obj)

Detect non-missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are valid (not missing, which is NaN in numeric arrays, None or NaN in object arrays, NaT in datetimelike).

Parameters

obj [array_like or object value] Object to check for not null or non-missing values.

Returns

bool or array_like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is valid.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.notna('dog')
True
```

```
>>> pd.notna(pd.NA)
False
```

```
>>> pd.notna(np.nan)
False
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.notna(array)
array([[ True, False,  True],
       [ True,  True, False]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
```

(continues on next page)

(continued from previous page)

```
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.notna(index)
array([ True,  True, False,  True])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
     0     1    2
0  ant   bee  cat
1  dog  None  fly
>>> pd.notna(df)
      0      1     2
0  True   True  True
1  True  False  True
```

```
>>> pd.notna(df[1])
0     True
1    False
Name: 1, dtype: bool
```

rapidspy.notnull

rapidspy.notnull(obj)

Detect non-missing values for an array-like object.

This function takes a scalar or array-like object and indicates whether values are valid (not missing, which is NaN in numeric arrays, None or NaN in object arrays, NaT in datetimelike).

Parameters

obj [array_like or object value] Object to check for not null or non-missing values.

Returns

bool or array_like of bool For scalar input, returns a scalar boolean. For array input, returns an array of boolean indicating whether each corresponding element is valid.

Examples

Scalar arguments (including strings) result in a scalar boolean.

```
>>> pd.notna('dog')
True
```

```
>>> pd.notna(pd.NA)
False
```

```
>>> pd.notna(np.nan)
False
```

ndarrays result in an ndarray of booleans.

```
>>> array = np.array([[1, np.nan, 3], [4, 5, np.nan]])
>>> array
array([[ 1., nan,  3.],
       [ 4.,  5., nan]])
>>> pd.notna(array)
array([[ True, False,  True],
       [ True,  True, False]])
```

For indexes, an ndarray of booleans is returned.

```
>>> index = pd.DatetimeIndex(["2017-07-05", "2017-07-06", None,
...                           "2017-07-08"])
>>> index
DatetimeIndex(['2017-07-05', '2017-07-06', 'NaT', '2017-07-08'],
              dtype='datetime64[ns]', freq=None)
>>> pd.notna(index)
array([ True,  True, False,  True])
```

For Series and DataFrame, the same type is returned, containing booleans.

```
>>> df = pd.DataFrame([['ant', 'bee', 'cat'], ['dog', None, 'fly']])
>>> df
     0     1    2
0  ant   bee  cat
1  dog  None  fly
>>> pd.notna(df)
      0      1     2
```

(continues on next page)

(continued from previous page)

```
0  True   True  True
1  True  False  True
```

```
>>> pd.notna(df[1])
0     True
1    False
Name: 1, dtype: bool
```

3.3.3 Top-level dealing with datetimelike data

to_datetime(arg)	Convert argument to datetime.

rapidspy.to_datetime

rapidspy.to_datetime(arg)

Convert argument to datetime.

This function converts a scalar, array-like, Series or DataFrame/dict-like to a pandas datetime object.

Parameters

arg [int, float, str, datetime, list, tuple, 1-d array, Series, DataFrame/dict-like] The object to convert to a datetime.If a DataFrame is provided, the method expects minimally the following columns: ”year”, ”month”, ”day”.

Returns

datetime If parsing succeeded. Return type depends on input(types in parenthesis correspond to fallback in case of unsuccessful timezone or out-of-range timestamp parsing):

- scalar: Timestamp (or datetime.datetime)
- array-like: DatetimeIndex (or Series with object dtype containing datetime.datetime)
- Series: Series of datetime64 dtype (or Series of object dtype containing datetime.datetime)
- DataFrame: Series of datetime64 dtype (or Series of object dtype containing datetime.datetime)

Raises

ParserError When parsing a date from string fails.

ValueError When another datetime conversion error happens. For example when one of 'year', 'month', 'day' columns is missing in a DataFrame, or when a Timezone-aware datetime.datetime is found in an array-like of mixed time offsets.

Notes

Many input types are supported, and lead to different output types:

- scalars can be int, float, str, datetime object (from stdlib datetime module or numpy). They are converted to Timestamp when possible, otherwise they are converted to datetime.datetime. None/NaN/null scalars are converted to NaT.
- array-like can contain int, float, str, datetime objects. They are converted to DatetimeIndex when possible, otherwise they are converted to Index with object dtype, containing datetime.datetime. None/NaN/null entries are converted to NaT in both cases.
- Series are converted to Series with datetime64 dtype when possible, otherwise they are converted to Series with object dtype, containing datetime.datetime. None/NaN/null entries are converted to NaT in both cases.
- DataFrame/dict-like are converted to Series with datetime64 dtype. For each row a datetime is created from assembling the various dataframe columns. Column keys can be common abbreviations like ['year', 'month', 'day', 'minute', 'second', 'ms', 'us', 'ns']) or plurals of the same.

The following causes are responsible for datetime.datetime objects being returned (possibly inside an Index or a Series with object dtype) instead of a proper pandas designated type (Timestamp, DatetimeIndex or Series with datetime64 dtype):

- when any input element is before Timestamp.min or after Timestamp.max, see timestamp limitations.

Examples

Assembling a datetime from multiple columns of a DataFrame. The keys can be common abbreviations like ['year', 'month', 'day', 'minute', 'second', 'ms', 'us', 'ns']) or plurals of the same

```
>>> df = pd.DataFrame({'year': [2015, 2016],
...                    'month': [2, 3],
...                    'day': [4, 5]})
>>> pd.to_datetime(df)
```

(continues on next page)

(continued from previous page)

```
0    2015-02-04
1    2016-03-05
dtype: datetime64[ns]
```

Timezones and time offsets

- Timezone-naive inputs are converted to timezone-naive DatetimeIndex:

```
>>> pd.to_datetime(['2018-10-26 12:00', '2018-10-26 13:00:15'])
DatetimeIndex(['2018-10-26 12:00:00', '2018-10-26 13:00:15'],
          dtype='datetime64[ns]', freq=None)
```

- Timezone-aware inputs with constant time offset are converted to timezone-aware DatetimeIndex:

```
>>> pd.to_datetime(['2018-10-26 12:00 -0500', '2018-10-26 13:00 -0500'])
DatetimeIndex(['2018-10-26 12:00:00-05:00', '2018-10-26 13:00:00-05:00'],
          dtype='datetime64[ns, pytz.FixedOffset(-300)]', freq=None)
```

- However, timezone-aware inputs with mixed time offsets (for example issued from a timezone with daylight savings, such as Europe/Paris) are not successfully converted to a DatetimeIndex. Instead a simple Index containing datetime.datetime objects is returned:

```
>>> pd.to_datetime(['2020-10-25 02:00 +0200', '2020-10-25 04:00 +0100'])
Index([2020-10-25 02:00:00+02:00, 2020-10-25 04:00:00+01:00],
  dtype='object')
```

- A mix of timezone-aware and timezone-naive inputs is converted to a timezone-aware DatetimeIndex if the offsets of the timezone-aware are constant:

```
>>> from datetime import datetime
>>> pd.to_datetime(["2020-01-01 01:00 -01:00", datetime(2020, 1, 1, 3, 0)])
DatetimeIndex(['2020-01-01 01:00:00-01:00', '2020-01-01 02:00:00-01:00'],
          dtype='datetime64[ns, pytz.FixedOffset(-60)]', freq=None)
```

- Finally, mixing timezone-aware strings and datetime.datetime always raises an error, even if the elements all have the same time offset.

```
>>> from datetime import datetime, timezone, timedelta
>>> d = datetime(2020, 1, 1, 18, tzinfo=timezone(-timedelta(hours=1)))
>>> pd.to_datetime(["2020-01-01 17:00 -0100", d])
Traceback (most recent call last):
    ...
ValueError: Tz-aware datetime.datetime cannot be converted to datetime64
            unless utc=True
```

3.4 Series

3.4.1 RapidsPY

Series.compute()	Return a computed RapidsPY DataFrame of Series.
Series.to_pandas()	Convert a RapidsPY Series to Pandas Series.

rapidspy.Series.compute

Series.compute()

Return a computed RapidsPY DataFrame of Series.

rapidspy.Series.to_pandas

Series.to_pandas()

Convert a RapidsPY Series to Pandas Series.

Returns

Series

Notes

This method should only be used if the resulting pandas Series is expected to be small, as all the data is loaded into the memory.

Examples

```
>>> s = pd.Series([1, 3, 5, np.nan, 6, 8], name='a')
>>> s.index.name='i'
>>> s
i
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
Name: a, dtype: float64
>>> rs = rc.from_pandas(s, name='rs')
>>> rs.compute()
i
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
Name: a, dtype: float64
>>> rtp=rs.to_pandas()
>>> rtp
i
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
Name: a, dtype: float64
```

3.4.2 Attributes

Axes

Series.index	The index (axis labels) of the Series.
Series.axes	Return a list of the row axis labels.
Series.values	Return Series as ndarray or ndarray-like depending on the dtype.
Series.dtype	Return the dtype object of the underlying data.
Series.shape	Return a tuple of the shape of the underlying data.
Series.ndim	Number of dimensions of the underlying data, by definition 1.
Series.size	Return the number of elements in the underlying data.
Series.empty	Indicator whether Series/DataFrame is empty.
Series.dtypes	Return the dtype object of the underlying data.
Series.name	Return the name of the Series.

rapidspy.Series.index

property Series.index

The index (axis labels) of the Series.

rapidspy.Series.axes

property Series.axes

Return a list of the row axis labels.

rapidspy.Series.values

property Series.values

Return Series as ndarray or ndarray-like depending on the dtype.

Returns

numpy.ndarray or ndarray-like

Examples

```
>>> pd.Series([1, 2, 3]).values
array([1, 2, 3])
```

```
>>> pd.Series(list('aabc')).values
array(['a', 'a', 'b', 'c'], dtype=object)
```

```
>>> pd.Series(list('aabc')).astype('category').values
['a', 'a', 'b', 'c']
Categories (3, object): ['a', 'b', 'c']
```

Timezone aware datetime data is converted to UTC:

```
>>> pd.Series(pd.date_range('20130101', periods=3,
...                         tz='US/Eastern')).values
array(['2013-01-01T05:00:00.000000000',
       '2013-01-02T05:00:00.000000000',
       '2013-01-03T05:00:00.000000000'], dtype='datetime64[ns]')
```

rapidspy.Series.dtype

property Series.dtype

Return the dtype object of the underlying data.

rapidspy.Series.shape

property Series.shape

Return a tuple of the shape of the underlying data.

rapidspy.Series.ndim

property Series.ndim

Number of dimensions of the underlying data, by definition 1.

rapidspy.Series.size

property Series.size

Return the number of elements in the underlying data.

rapidspy.Series.empty

property Series.empty

Indicator whether Series/DataFrame is empty.

True if Series/DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

Returns

bool If Series/DataFrame is empty, return True, if not return False.

Notes

If Series/DataFrame contains only NaNs, it is still not considered empty. See the example below.

Examples

An example of an actual empty DataFrame. Notice the index is empty:

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```
>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
    A
0 NaN
>>> df.empty
False
```

(continues on next page)

(continued from previous page)

```
>>> df.dropna().empty
True
```

```
>>> ser_empty = pd.Series({'A' : []})
>>> ser_empty
A    []
dtype: object
>>> ser_empty.empty
False
>>> ser_empty = pd.Series()
>>> ser_empty.empty
True
```

rapidspy.Series.dtypes

property Series.dtypes

Return the dtype object of the underlying data.

rapidspy.Series.name

property Series.name

Return the name of the Series.

The name of a Series becomes its index or column name if it is used to form a DataFrame. It is also used whenever displaying the Series using the interpreter.

Returns

label (hashable object) The name of the Series, also the column name if part of a DataFrame.

Examples

The Series name can be set initially when calling the constructor.

```
>>> s = pd.Series([1, 2, 3], dtype=np.int64, name='Numbers')
>>> s
0    1
1    2
2    3
```

(continues on next page)

(continued from previous page)

```
Name: Numbers, dtype: int64
>>> s.name = "Integers"
>>> s
0    1
1    2
2    3
Name: Integers, dtype: int64
```

The name of a Series within a DataFrame is its column name.

```
>>> df = pd.DataFrame([[1, 2], [3, 4], [5, 6]],
...                   columns=["Odd Numbers", "Even Numbers"])
>>> df
   Odd Numbers  Even Numbers
0            1             2
1            3             4
2            5             6
>>> df["Even Numbers"].name
'Even Numbers'
```

3.4.3 Conversion

Series.to_pandas()	Convert a RapidsPY Series to Pandas Series.
Series.astype(dtype)	Cast a pandas object to a specified dtype dtype.
Series.copy()	Make a deep copy of this object's indices and data.

rapidspy.Series.astype

Series.astype(dtype)

Cast a pandas object to a specified dtype dtype.

Parameters

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, …}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame' s columns to column-specific types.

Returns

casted [same type as caller]

Examples

Create a DataFrame:

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

Cast all columns to int32:

```
>>> df.astype('int32').dtypes
col1    int32
col2    int32
dtype: object
```

Cast col1 to int32 using a dictionary:

```
>>> df.astype({'col1': 'int32'}).dtypes
col1    int32
col2    int64
dtype: object
```

Create a series:

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> cat_dtype = pd.api.types.CategoricalDtype(
...     categories=[2, 1], ordered=True)
>>> ser.astype(cat_dtype)
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Create a series of dates:

```
>>> ser_date = pd.Series(pd.date_range('20200101', periods=3))
>>> ser_date
0   2020-01-01
1   2020-01-02
2   2020-01-03
dtype: datetime64[ns]
```

Datetimes are localized to UTC first before converting to the specified timezone:

```
>>> ser_date.astype('datetime64[ns, US/Eastern]')
0   2019-12-31 19:00:00-05:00
1   2020-01-01 19:00:00-05:00
2   2020-01-02 19:00:00-05:00
dtype: datetime64[ns, US/Eastern]
```

rapidspy.Series.copy

Series.copy()

Make a deep copy of this object's indices and data.

A new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

Returns

copy [Series or DataFrame] Object type matches caller.

Notes

Data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to copy.deepcopy in the Standard Library, which recursively copies object data.

While Index objects are copied, the underlying numpy array is not copied for performance reasons. Since Index is immutable, the underlying data can be safely shared and a copy is not needed.

Examples

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

3.4.4 Indexing, iteration

Series.get(key[, default])	Get item from object for given key (ex: DataFrame column).
Series.at	Access a single value for a row/column label pair.
Series.iat	Access a single value for a row/column pair by integer position.
Series.loc	Access a group of rows and columns by label(s) or a boolean array.
Series.iloc	Purely integer-location based indexing for selection by position.
Series.keys()	Return alias for index.

rapidspy.Series.get

Series.get(key, default=None)

Get item from object for given key (ex: DataFrame column).

Returns default value if not found.

Parameters

key [object]

Returns

value [same type as items contained in object]

Examples

```
>>> df = pd.DataFrame(
...     [
...         [24.3, 75.7, "high"],
...         [31, 87.8, "high"],
...         [22, 71.6, "medium"],
...         [35, 95, "medium"],
...     ],
...     columns=["temp_celsius", "temp_fahrenheit", "windspeed"],
...     index=pd.date_range(start="2014-02-12", end="2014-02-15", freq="D"),
... )
```

```
>>> df
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          24.3             75.7      high
2014-02-13          31.0             87.8      high
2014-02-14          22.0             71.6    medium
2014-02-15          35.0             95.0    medium
```

```
>>> df.get(["temp_celsius", "windspeed"])
            temp_celsius windspeed
2014-02-12          24.3      high
2014-02-13          31.0      high
2014-02-14          22.0    medium
2014-02-15          35.0    medium
```

If the key isn't found, the default value will be used.

```
>>> df.get(["temp_celsius", "temp_kelvin"], default="default_value")
'default_value'
```

rapidspy.Series.at

property Series.at

Access a single value for a row/column label pair.

Similar to loc, in that both provide label-based lookups. Use at if you only need to get or set a single value in a DataFrame or Series.

Raises

KeyError If 'label' does not exist in DataFrame.

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

rapidspy.Series.iat

property Series.iat

Access a single value for a row/column pair by integer position.

Similar to iloc, in that both provide integer-based lookups. Use iat if you only need to get or set a single value in a DataFrame or Series.

使用 MOXE 引擎时暂不支持。

Raises

IndexError When integer position is out of bounds.

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

rapidspy.Series.loc

property Series.loc

Access a group of rows and columns by label(s) or a boolean array.

.loc[] is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. 5 or 'a', (note that 5 is interpreted as a label of the index, and never as an integer position along the index).
- A list or array of labels, e.g. ['a', 'b', 'c'].
- An alignable boolean Series. The index of the key will be aligned before masking.

Raises

KeyError If any items are not found.

IndexingError If an indexed key is passed and its index is unalignable to the frame index.

Examples

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=['cobra', 'viper', 'sidewinder'],
...      columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
viper               4       5
sidewinder          7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using [[]] returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper               4       5
sidewinder          7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder          7       8
```

Alignable boolean Series:

```
>>> df.loc[pd.Series([False, True, False],
...        index=['viper', 'sidewinder', 'cobra'])]
            max_speed  shield
sidewinder          7       8
```

Index (same behavior as df.reindex)

```
>>> df.loc[pd.Index(["cobra", "viper"], name="foo")]
       max_speed  shield
foo
cobra          1       2
viper          4       5
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
            max_speed  shield
cobra              10      10
viper               4      50
sidewinder          7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
```

(continues on next page)

(continued from previous page)

```
viper            30       50
sidewinder        30       50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...    ('cobra', 'mark i'), ('cobra', 'mark ii'),
...    ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...    ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
```

(continues on next page)

(continued from previous page)

```
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using [[]] returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                    max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

rapidspy.Series.iloc

property Series.iloc

Purely integer-location based indexing for selection by position.

.iloc[] is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. 5.
- A list or array of integers, e.g. [4, 3, 0].
- A slice object with ints, e.g. 1:7.
- A boolean array.

.iloc will raise IndexError if a requested indexer is out-of-bounds, except slice indexers which allow out-of-bounds indexing (this conforms with python/numpy slice semantics).

Examples

```
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
>>> df = pd.DataFrame(mydict)
>>> df
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

Indexing just the rows

With a scalar integer.

```
>>> type(df.iloc[0])
<class 'pandas.core.series.Series'>
>>> df.iloc[0]
a    1
b    2
c    3
d    4
Name: 0, dtype: int64
```

With a list of integers.

```
>>> df.iloc[[0]]
   a  b  c  d
0  1  2  3  4
>>> type(df.iloc[[0]])
<class 'pandas.core.frame.DataFrame'>
```

```
>>> df.iloc[[0, 1]]
     a    b    c    d
0    1    2    3    4
1  100  200  300  400
```

With a slice object.

```
>>> df.iloc[:3]
      a     b     c     d
```

(continues on next page)

(continued from previous page)

```
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

With a boolean mask the same length as the index.

```
>>> df.iloc[[True, False, True]]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

With a callable, useful in method chains. The x passed to the lambda is the DataFrame being sliced. This selects the rows whose index label even.

```
>>> df.iloc[lambda x: x.index % 2 == 0]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

Indexing both axes

You can mix the indexer types for the index and columns. Use : to select the entire axis.

With scalar integers.

```
>>> df.iloc[0, 1]
2
```

With lists of integers.

```
>>> df.iloc[[0, 2], [1, 3]]
      b     d
0     2     4
2  2000  4000
```

With slice objects.

```
>>> df.iloc[1:3, 0:3]
      a     b     c
1   100   200   300
2  1000  2000  3000
```

With a boolean array whose length matches the columns.

```
>>> df.iloc[:, [True, False, True, False]]
      a     c
0     1     3
1   100   300
2  1000  3000
```

With a callable function that expects the Series or DataFrame.

```
>>> df.iloc[:, lambda df: [0, 2]]
      a     c
0     1     3
1   100   300
2  1000  3000
```

rapidspy.Series.keys

Series.keys()

Return alias for index.

Returns

Index Index of the Series.

For more information on .at, .iat, .loc, and .iloc, see the indexing documentation.

3.4.5 Binary operator functions

Series.add(other)	Return Addition of series and other, element-wise (binary operator add).
Series.sub(other)	Return Subtraction of series and other, element-wise (binary operator sub).
Series.mul(other)	Return Multiplication of series and other, element-wise (binary operator mul).
Series.div(other)	Return Floating division of series and other, element-wise (binary operator truediv).
Series.truediv(other)	Return Floating division of series and other, element-wise (binary operator truediv).
Series.floordiv(other)	Return Integer division of series and other, element-wise (binary operator floordiv).

continues on next page

Table 10 – continued from previous page

Series.pow(other)	Return Exponential power of series and other, element-wise (binary operator pow).
Series.radd(other)	Return Addition of series and other, element-wise (binary operator radd).
Series.rsub(other)	Return Subtraction of series and other, element-wise (binary operator rsub).
Series.round([decimals])	Round each value in a Series to the given number of decimals.
Series.lt(other)	Return Less than of series and other, element-wise (binary operator lt).
Series.gt(other)	Return Greater than of series and other, element-wise (binary operator gt).
Series.le(other)	Return Less than or equal to of series and other, element-wise (binary operator le).
Series.ge(other)	Return Greater than or equal to of series and other, element-wise (binary operator ge).
Series.ne(other)	Return Not equal to of series and other, element-wise (binary operator ne).
Series.eq(other)	Return Equal to of series and other, element-wise (binary operator eq).

rapidspy.Series.add

Series.add(other)

Return Addition of series and other, element-wise (binary operator add).

Equivalent to series + other.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b)
a    2.0
b    NaN
c    NaN
d    NaN
e    NaN
dtype: float64
```

rapidspy.Series.sub

Series.sub(other)

Return Subtraction of series and other, element-wise (binary operator sub).

Equivalent to series - other.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.subtract(b)
a    0.0
b    NaN
c    NaN
d    NaN
e    NaN
dtype: float64
```

rapidspy.Series.mul

Series.mul(other)

Return Multiplication of series and other, element-wise (binary operator mul).

Equivalent to series * other.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.multiply(b)
a    1.0
b    NaN
c    NaN
d    NaN
e    NaN
dtype: float64
```

rapidspy.Series.div

Series.div(other)

Return Floating division of series and other, element-wise (binary operator truediv).

Equivalent to series / other.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.divide(b)
a    1.0
b    NaN
c    NaN
d    NaN
e    NaN
dtype: float64
```

rapidspy.Series.truediv

Series.truediv(other)

Return Floating division of series and other, element-wise (binary operator truediv).

Equivalent to series / other.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.divide(b)
a    1.0
b    NaN
c    NaN
d    NaN
e    NaN
dtype: float64
```

rapidspy.Series.floordiv

Series.floordiv(other)

Return Integer division of series and other, element-wise (binary operator floordiv).

Equivalent to series // other.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.floordiv(b)
a    1.0
b    NaN
c    NaN
d    NaN
e    NaN
dtype: float64
```

rapidspy.Series.pow

Series.pow(other)

Return Exponential power of series and other, element-wise (binary operator pow).

Equivalent to series ** other.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.pow(b)
a    1.0
b    1.0
c    1.0
d    NaN
e    NaN
dtype: float64
```

rapidspy.Series.radd

Series.radd(other)

Return Addition of series and other, element-wise (binary operator radd).

Equivalent to other + series.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.add(b)
a    2.0
b    NaN
c    NaN
d    NaN
e    NaN
dtype: float64
```

rapidspy.Series.rsub

Series.rsub(other)

Return Subtraction of series and other, element-wise (binary operator rsub).

Equivalent to other - series.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
>>> a.subtract(b)
a    0.0
b    NaN
c    NaN
d    NaN
e    NaN
dtype: float64
```

rapidspy.Series.round

Series.round(decimals=0)

Round each value in a Series to the given number of decimals.

Parameters

decimals [int, default 0] Number of decimal places to round to. If decimals is negative, it specifies the number of positions to the left of the decimal point.

Returns

Series Rounded values of the Series.

Examples

```
>>> s = pd.Series([0.1, 1.3, 2.7])
>>> s.round()
0    0.0
1    1.0
2    3.0
dtype: float64
```

rapidspy.Series.lt

Series.lt(other)

Return Less than of series and other, element-wise (binary operator lt).

Equivalent to series < other.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan, 1], index=['a', 'b', 'c', 'd', 'e'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
e    1.0
dtype: float64
>>> b = pd.Series([0, 1, 2, np.nan, 1], index=['a', 'b', 'c', 'd', 'f'])
>>> b
a    0.0
b    1.0
c    2.0
d    NaN
f    1.0
dtype: float64
```

(continues on next page)

(continued from previous page)

```
>>> a.lt(b)
a    False
b    False
c     True
d    False
e    False
f    False
dtype: bool
```

rapidspy.Series.gt

Series.gt(other)

Return Greater than of series and other, element-wise (binary operator gt).

Equivalent to series > other.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan, 1], index=['a', 'b', 'c', 'd', 'e'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
e    1.0
dtype: float64
>>> b = pd.Series([0, 1, 2, np.nan, 1], index=['a', 'b', 'c', 'd', 'f'])
>>> b
a    0.0
b    1.0
c    2.0
d    NaN
f    1.0
```

(continues on next page)

(continued from previous page)

```
dtype: float64
>>> a.gt(b)
a     True
b    False
c    False
d    False
e    False
f    False
dtype: bool
```

rapidspy.Series.le

Series.le(other)

Return Less than or equal to of series and other, element-wise (binary operator le).

Equivalent to series <= other.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan, 1], index=['a', 'b', 'c', 'd', 'e'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
e    1.0
dtype: float64
>>> b = pd.Series([0, 1, 2, np.nan, 1], index=['a', 'b', 'c', 'd', 'f'])
>>> b
a    0.0
b    1.0
c    2.0
d    NaN
```

(continues on next page)

(continued from previous page)

```
f    1.0
dtype: float64
>>> a.le(b)
a    False
b     True
c     True
d    False
e    False
f    False
dtype: bool
```

rapidspy.Series.ge

Series.ge(other)

Return Greater than or equal to of series and other, element-wise (binary operator ge).

Equivalent to series >= other.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan, 1], index=['a', 'b', 'c', 'd', 'e'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
e    1.0
dtype: float64
>>> b = pd.Series([0, 1, 2, np.nan, 1], index=['a', 'b', 'c', 'd', 'f'])
>>> b
a    0.0
b    1.0
c    2.0
```

(continues on next page)

(continued from previous page)

```
d    NaN
f    1.0
dtype: float64
>>> a.ge(b)
a     True
b     True
c    False
d    False
e    False
f    False
dtype: bool
```

rapidspy.Series.ne

Series.ne(other)

Return Not equal to of series and other, element-wise (binary operator ne).

Equivalent to series != other.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
```

(continues on next page)

(continued from previous page)

```
e    NaN
dtype: float64
>>> a.ne(b)
a    False
b     True
c     True
d     True
e     True
dtype: bool
```

rapidspy.Series.eq

Series.eq(other)

Return Equal to of series and other, element-wise (binary operator eq).

Equivalent to series == other.

Parameters

other [Series or scalar value]

Returns

Series The result of the operation.

Examples

```
>>> a = pd.Series([1, 1, 1, np.nan], index=['a', 'b', 'c', 'd'])
>>> a
a    1.0
b    1.0
c    1.0
d    NaN
dtype: float64
>>> b = pd.Series([1, np.nan, 1, np.nan], index=['a', 'b', 'd', 'e'])
>>> b
a    1.0
b    NaN
d    1.0
e    NaN
dtype: float64
```

(continues on next page)

(continued from previous page)

```
>>> a.eq(b)
a     True
b    False
c    False
d    False
e    False
dtype: bool
```

3.4.6 Function application, GroupBy & window

Series.agg(func)	Aggregate using one or more operations over the specified axis.
Series.aggregate(func)	Aggregate using one or more operations over the specified axis.
Series.groupby([by])	Group Series using a mapper or by a Series of columns.

rapidspy.Series.agg

Series.agg(func)

Aggregate using one or more operations over the specified axis.

Parameters

func [str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply.

Accepted combinations are:

- string function name
- list of function names, e.g. ['sum', 'mean']

Returns

scalar, Series or DataFrame The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

Notes

agg is an alias for aggregate. Use the alias.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.agg('min')
1
```

```
>>> s.agg(['min', 'max'])
min   1
max   4
dtype: int64
```

rapidspy.Series.aggregate

Series.aggregate(func)

Aggregate using one or more operations over the specified axis.

Parameters

func [str, list or dict] Function to use for aggregating the data. If a function, must either work when passed a Series or when passed to Series.apply.

Accepted combinations are:

- string function name
- list of function names, e.g. ['sum', 'mean']

Returns

scalar, Series or DataFrame The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

Notes

agg is an alias for aggregate. Use the alias.

Examples

```
>>> s = pd.Series([1, 2, 3, 4])
>>> s
0    1
1    2
2    3
3    4
dtype: int64
```

```
>>> s.agg('min')
1
```

```
>>> s.agg(['min', 'max'])
min   1
max   4
dtype: int64
```

rapidspy.Series.groupby

Series.groupby(by=None)

Group Series using a mapper or by a Series of columns.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

Parameters

by [label, or list of labels] A label or list of labels may be passed to group by the columns in self. Notice that a tuple is interpreted as a (single) key.

Returns

SeriesGroupBy Returns a groupby object that contains information about the groups.

Examples

```
>>> ser = pd.Series([390., 350., 30., 20.],
...                 index=['Falcon', 'Falcon', 'Parrot', 'Parrot'], name="Max Speed")
>>> ser
Falcon    390.0
Falcon    350.0
Parrot     30.0
Parrot     20.0
Name: Max Speed, dtype: float64
>>> ser.groupby(["a", "b", "a", "b"]).mean()
a    210.0
b    185.0
Name: Max Speed, dtype: float64
>>> ser.groupby(ser > 100).mean()
Max Speed
False     25.0
True     370.0
Name: Max Speed, dtype: float64
```

3.4.7 Computations / descriptive stats

Series.abs()	Return a Series/DataFrame with absolute numeric value of each element.
Series.all()	Return whether all elements are True, potentially over an axis.
Series.any()	Return whether any element is True, potentially over an axis.
Series.between(left, right[, inclusive])	Return boolean Series equivalent to left <= series <= right.
Series.count()	Return number of non-NA/null observations in the Series.
Series.describe()	Generate descriptive statistics.
Series.max()	Return the maximum of the values over the requested axis.

continues on next page

Table 12 – continued from previous page

Series.mean()	Return the mean of the values over the requested axis.
Series.median()	Return the median of the values over the requested axis.
Series.min()	Return the minimum of the values over the requested axis.
Series.nlargest([n])	Return the largest n elements.
Series.nsmallest([n])	Return the smallest n elements.
Series.std()	Return sample standard deviation over requested axis.
Series.sum()	Return the sum of the values over the requested axis.
Series.var()	Return unbiased variance over requested axis.
Series.unique()	Return unique values of Series object.
Series.nunique()	Return number of unique elements in the object.
Series.is_unique	Return boolean if values in the object are unique.
Series.value_counts()	Return a Series containing counts of unique values.

rapidspy.Series.abs

Series.abs()

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

Returns

abs Series/DataFrame containing the absolute value of each element.

Examples

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

rapidspy.Series.all

Series.all()

Return whether all elements are True, potentially over an axis.

Returns True unless there at least one element within a series or along a Dataframe axis that is False or equivalent (e.g. zero or empty).

Returns

scalar

Examples

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
>>> pd.Series([]).all()
True
>>> pd.Series([np.nan]).all()
True
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

rapidspy.Series.any

Series.any()

Return whether any element is True, potentially over an axis.

Returns False unless there is at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

Returns

scalar

Examples

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([False, False]).any()
False
>>> pd.Series([True, False]).any()
True
>>> pd.Series([]).any()
False
>>> pd.Series([np.nan]).any()
False
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

rapidspy.Series.between

Series.between(left, right, inclusive=True)

Return boolean Series equivalent to left <= series <= right.

This function returns a boolean vector containing True wherever the corresponding Series element is between the boundary values left and right. NA values are treated as False.

Parameters

left [scalar or list-like] Left boundary.

right [scalar or list-like] Right boundary.

inclusive [bool] Include boundaries.

Returns

Series Series representing whether each element is between left and right (inclusive).

Notes

This function is equivalent to (left <= ser) & (ser <= right)

Examples

```
>>> s = pd.Series([2, 0, 4, 8, np.nan])
```

Boundary values are included by default:

```
>>> s.between(1, 4)
0     True
1    False
2     True
3    False
4    False
dtype: bool
```

With inclusive set to ”neither” boundary values are excluded:

```
>>> s.between(1, 4, inclusive="neither")
0     True
1    False
2    False
3    False
4    False
dtype: bool
```

left and right can be any scalar value:

```
>>> s = pd.Series(['Alice', 'Bob', 'Carol', 'Eve'])
>>> s.between('Anna', 'Daniel')
0    False
1     True
2     True
3    False
dtype: bool
```

rapidspy.Series.count

Series.count()

Return number of non-NA/null observations in the Series.

Returns

int or Series (if level specified) Number of non-null values in the Series.

Examples

```
>>> s = pd.Series([0.0, 1.0, np.nan])
>>> s.count()
2
```

rapidspy.Series.describe

Series.describe()

Generate descriptive statistics.

Descriptive statistics include those that summarize the central tendency, dispersion and shape of a dataset' s distribution, excluding NaN values.

Analyzes both numeric and object series, as well as DataFrame column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

使用 MOXE 引擎时暂不支持。

Returns

Series or DataFrame Summary statistics of the Series or Dataframe provided.

Notes

For numeric data, the result' s index will include count, mean, std, min, max as well as lower, 50 and upper percentiles. By default the lower percentile is 25 and the upper percentile is 75. The 50 percentile is the same as the median.

For object data (e.g. strings or timestamps), the result' s index will include count, unique, top, and freq. The top is the most common value. The freq is the most common value' s frequency. Timestamps also include the first and last items.

If multiple object values have the highest count, then the count and top results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a DataFrame, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns.

Examples

Describing a numeric Series.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
```

(continues on next page)

(continued from previous page)

```
50%      2.0
75%      2.5
max      3.0
dtype: float64
```

Describing a categorical Series.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a DataFrame. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),
...                    'numeric': [1, 2, 3],
...                    'object': ['a', 'b', 'c']
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing a column from a DataFrame by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
```

(continues on next page)

(continued from previous page)

```
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

rapidspy.Series.max

Series.max()

Return the maximum of the values over the requested axis.

If you want the index of the maximum, use idxmax. This is the equivalent of the numpy.ndarray method argmax.

Returns

scalar

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.max()
8
```

rapidspy.Series.mean

Series.mean()

Return the mean of the values over the requested axis.

Returns

scalar

rapidspy.Series.median

Series.median()

Return the median of the values over the requested axis.

使用 MOXE 引擎时暂不支持。

Returns

scalar

rapidspy.Series.min

Series.min()

Return the minimum of the values over the requested axis.

If you want the index of the minimum, use idxmin. This is the equivalent of the numpy.ndarray method argmin.

Returns

scalar

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
```

(continues on next page)

(continued from previous page)

```
      spider     8
Name: legs, dtype: int64
```

```
>>> s.min()
0
```

rapidspy.Series.nlargest

Series.nlargest(n=5)

Return the largest n elements.

Parameters

n [int, default 5] Return this many descending sorted values.

Returns

Series The n largest values in the Series, sorted in decreasing order.

Notes

Faster than .sort_values(ascending=False).head(n) for small n relative to the size of the Series object.

Examples

```
>>> countries_population = {"Italy": 59000000, "France": 65000000,
...                         "Malta": 434000, "Maldives": 434000,
...                         "Brunei": 434000, "Iceland": 337000,
...                         "Nauru": 11300, "Tuvalu": 11300,
...                         "Anguilla": 11300, "Montserrat": 5200}
>>> s = pd.Series(countries_population)
>>> s
Italy       59000000
France      65000000
Malta         434000
Maldives      434000
Brunei        434000
Iceland       337000
Nauru          11300
Tuvalu         11300
```

(continues on next page)

(continued from previous page)

```
Anguilla       11300
Montserrat      5200
dtype: int64
```

The n largest elements where n=5 by default.

```
>>> s.nlargest()
France      65000000
Italy       59000000
Malta         434000
Maldives      434000
Brunei        434000
dtype: int64
```

The n largest elements where n=3.

```
>>> s.nlargest(3)
France    65000000
Italy     59000000
Malta       434000
dtype: int64
```

rapidspy.Series.nsmallest

Series.nsmallest(n=5)

Return the smallest n elements.

Parameters

n [int, default 5] Return this many ascending sorted values.

Returns

Series The n smallest values in the Series, sorted in increasing order.

Notes

Faster than .sort_values().head(n) for small n relative to the size of the Series object.

Examples

```
>>> countries_population = {"Italy": 59000000, "France": 65000000,
...                         "Brunei": 434000, "Malta": 434000,
...                         "Maldives": 434000, "Iceland": 337000,
...                         "Nauru": 11300, "Tuvalu": 11300,
...                         "Anguilla": 11300, "Montserrat": 5200}
>>> s = pd.Series(countries_population)
>>> s
Italy       59000000
France      65000000
Brunei        434000
Malta         434000
Maldives      434000
Iceland       337000
Nauru          11300
Tuvalu         11300
Anguilla       11300
Montserrat      5200
dtype: int64
```

The n smallest elements where n=5 by default.

```
>>> s.nsmallest()
Montserrat    5200
Nauru        11300
Tuvalu       11300
Anguilla     11300
Iceland     337000
dtype: int64
```

The n smallest elements where n=3.

```
>>> s.nsmallest(3)
Montserrat   5200
Nauru       11300
Tuvalu      11300
```

(continues on next page)

(continued from previous page)

```
dtype: int64
```

rapidspy.Series.std

Series.std()

Return sample standard deviation over requested axis.

Normalized by N-1 by default.

使用 MOXE 引擎时暂不支持。

Returns

scalar

Examples

```
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]}
...                   ).set_index('person_id')
>>> df
           age  height
person_id
0           21    1.61
1           25    1.87
2           62    1.49
3           43    2.01
```

The standard deviation of the columns can be found as follows:

```
>>> df.std()
age       18.786076
height     0.237417
```

rapidspy.Series.sum

Series.sum()

Return the sum of the values over the requested axis.

This is equivalent to the method numpy.sum.

Returns

scalar

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.sum()
14
```

By default, the sum of an empty or all-NA Series is 0.

```
>>> pd.Series([], dtype="float64").sum()
0.0
```

rapidspy.Series.var

Series.var()

Return unbiased variance over requested axis.

Normalized by N-1 by default.

使用 MOXE 引擎时暂不支持。

Returns

scalar

Examples

```
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]}
...                   ).set_index('person_id')
>>> df
           age  height
person_id
0           21    1.61
1           25    1.87
2           62    1.49
3           43    2.01
```

```
>>> df.var()
age       352.916667
height      0.056367
```

rapidspy.Series.unique

Series.unique()

Return unique values of Series object.

Uniques are returned in order of appearance. Hash table-based unique, therefore does NOT sort.

Returns

ndarray or ExtensionArray The unique values returned as a NumPy array. See Notes.

Notes

Returns the unique values as a NumPy array. In case of an extension-array backed Series, a new ExtensionArray of that type with just the unique values is returned. This includes

- Categorical
- Period
- Datetime with Timezone

- Interval
- Sparse
- IntegerNA

See Examples section.

Examples

```
>>> pd.Series([2, 1, 3, 3], name='A').unique()
array([2, 1, 3])
```

```
>>> pd.Series([pd.Timestamp('2016-01-01') for _ in range(3)]).unique()
array(['2016-01-01T00:00:00.000000000'], dtype='datetime64[ns]')
```

```
>>> pd.Series([pd.Timestamp('2016-01-01', tz='US/Eastern')
...            for _ in range(3)]).unique()
<DatetimeArray>
['2016-01-01 00:00:00-05:00']
Length: 1, dtype: datetime64[ns, US/Eastern]
```

An Categorical will return categories in the order of appearance and with the same dtype.

```
>>> pd.Series(pd.Categorical(list('baabc'))).unique()
['b', 'a', 'c']
Categories (3, object): ['b', 'a', 'c']
>>> pd.Series(pd.Categorical(list('baabc'), categories=list('abc'),
...                          ordered=True)).unique()
['b', 'a', 'c']
Categories (3, object): ['a' < 'b' < 'c']
```

rapidspy.Series.nunique

Series.nunique()

Return number of unique elements in the object.

Excludes NA values by default.

Returns

int

Examples

```
>>> s = pd.Series([1, 3, 5, 7, 7])
>>> s
0    1
1    3
2    5
3    7
4    7
dtype: int64
```

```
>>> s.nunique()
4
```

rapidspy.Series.is_unique

property Series.is_unique

Return boolean if values in the object are unique.

Returns

bool

rapidspy.Series.value_counts

Series.value_counts()

Return a Series containing counts of unique values.

The resulting object will be in descending order so that the first element is the most frequently-occurring element. Excludes NA values by default.

Returns

Series

Examples

```
>>> index = pd.Index([3, 1, 2, 3, 4, np.nan])
>>> index.value_counts()
3.0    2
2.0    1
```

(continues on next page)

(continued from previous page)

```
4.0    1
1.0    1
dtype: int64
```

3.4.8 Reindexing / selection / label manipulation

Series.drop([labels, axis, index, columns])	Return Series with specified index labels removed.
Series.drop_duplicates()	Return Series with duplicate values removed.
Series.head([n])	Return the first n rows.
Series.reset_index()	Generate a new DataFrame or Series with the index reset.
Series.filter(like)	Subset the dataframe rows or columns according to the specified index labels.

rapidspy.Series.drop

Series.drop(labels=None, axis=0, index=None, columns=None)

Return Series with specified index labels removed.

Remove elements of a Series based on specifying the index labels. When using a multi-index, labels on different levels can be removed by specifying the level.

Parameters

labels [single label or list-like] Index labels to drop.

axis [0, default 0] Redundant for application on Series.

index [single label or list-like] Redundant for application on Series, but 'index' can be used instead of 'labels'.

columns [single label or list-like] No change is made to the Series; use 'index' or 'labels' instead.

Returns

Series Series with specified index labels removed.

Raises

KeyError If none of the labels are found in the index.

Examples

```
>>> s = pd.Series(data=np.arange(3), index=['A', 'B', 'C'])
>>> s
A  0
B  1
C  2
dtype: int64
```

Drop labels B en C

```
>>> s.drop(labels=['B', 'C'])
A  0
dtype: int64
```

rapidspy.Series.drop_duplicates

Series.drop_duplicates()

Return Series with duplicate values removed.

Returns

Series Series with duplicates dropped.

Examples

Generate a Series with duplicated entries.

```
>>> s = pd.Series(['lama', 'cow', 'lama', 'beetle', 'lama', 'hippo'],
...               name='animal')
>>> s
0      lama
1       cow
2      lama
3    beetle
4      lama
5     hippo
Name: animal, dtype: object
```

```
>>> s.drop_duplicates()
0      lama
```

(continues on next page)

(continued from previous page)

```
1       cow
3    beetle
5     hippo
Name: animal, dtype: object
```

rapidspy.Series.head

Series.head(n=5)

Return the first n rows.

This function returns the first n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

For negative values of n, this function returns all rows except the last n rows, equivalent to df[:-n].

Parameters

n [int, default 5] Number of rows to select.

Returns

same type as caller The first n rows of the caller object.

Examples

```
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
```

Viewing the first n lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

For negative values of n

```
>>> df.head(-3)
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
```

rapidspy.Series.reset_index

Series.reset_index()

Generate a new DataFrame or Series with the index reset.

This is useful when the index needs to be treated as a column, or when the index is meaningless and needs to be reset to the default before another operation.

使用 MOXE 引擎时暂不支持。

Returns

DataFrame a DataFrame is returned. The newly created columns will come first in the DataFrame, followed by the original Series values.

Examples

```
>>> s = pd.Series([1, 2, 3, 4], name='foo',
...               index=pd.Index(['a', 'b', 'c', 'd'], name='idx'))
```

Generate a DataFrame with default index.

```
>>> s.reset_index()
  idx  foo
0   a    1
1   b    2
2   c    3
3   d    4
```

Series with a multi-level index.

```
>>> arrays = [np.array(['bar', 'bar', 'baz', 'baz']),
...           np.array(['one', 'two', 'one', 'two'])]
>>> s2 = pd.Series(
...     range(4), name='foo',
...     index=pd.MultiIndex.from_arrays(arrays,
...                                     names=['a', 'b']))
```

```
>>> s2.reset_index()
     a    b  foo
0  bar  one    0
1  bar  two    1
2  baz  one    2
3  baz  two    3
```

rapidspy.Series.filter

Series.filter(like)

Subset the dataframe rows or columns according to the specified index labels.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

Parameters

like [str] Keep labels from axis for which "like in label == True".

Returns

same type as input object

3.4.9 Missing data handling

Series.dropna()	Return a new Series with missing values removed.
Series.isna()	Detect missing values.
Series.isnull()	Series.isnull is an alias for Series.isna.
Series.notna()	Detect existing (non-missing) values.
Series.notnull()	Series.notnull is an alias for Series.notna.

rapidspy.Series.dropna

Series.dropna()

Return a new Series with missing values removed.

Returns

Series Series with NA entries dropped from it.

Examples

```
>>> ser = pd.Series([1., 2., np.nan])
>>> ser
0    1.0
1    2.0
2    NaN
dtype: float64
```

Drop NA values from a Series.

```
>>> ser.dropna()
0    1.0
1    2.0
dtype: float64
```

Empty strings are not considered NA values. None is considered an NA value.

```
>>> ser = pd.Series([np.NaN, 2, pd.NaT, '', None, 'I stay'])
>>> ser
0       NaN
```

(continues on next page)

(continued from previous page)

```
1          2
2        NaT
3
4       None
5     I stay
dtype: object
>>> ser.dropna()
1          2
3
5     I stay
dtype: object
```

rapidspy.Series.isna

Series.isna()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy.NaN, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True).

Returns

Series Mask of bool values for each element in Series that indicates whether an element is an NA value.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

rapidspy.Series.isnull

Series.isnull()

Series.isnull is an alias for Series.isna.

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy.NaN, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True).

Returns

Series Mask of bool values for each element in Series that indicates whether an element is an NA value.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

rapidspy.Series.notna

Series.notna()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True). NA values, such as None or numpy.NaN, get mapped to False values.

Returns

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
```

(continues on next page)

(continued from previous page)

```
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

rapidspy.Series.notnull

Series.notnull()

Series.notnull is an alias for Series.notna.

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True). NA values, such as None or numpy.NaN, get mapped to False values.

Returns

Series Mask of bool values for each element in Series that indicates whether an element is not an NA value.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

3.4.10 Reshaping, sorting

Series.sort_values([ascending])	Sort by the values.
Series.sort_index([level, ascending])	Sort Series by index labels.

3.4.11 Accessors

Datetimelike properties

Series.dt can be used to access the values of the series as datetimelike and return several properties, like DatetimeProperties or TimedeltaProperties (see as below). These can be accessed like Series.dt.<property>. For example, for DatetimeProperties.date, it can be accessed by Series.dt.date; for TimedeltaProperties.days, it can be accessed by Series.dt.days.

Datetime properties

DatetimeProperties.date	Returns numpy array of python datetime.date objects.
DatetimeProperties.year	The year of the datetime.
DatetimeProperties.month	The month as January=1, December=12.
DatetimeProperties.day	The day of the datetime.
DatetimeProperties.hour	The hours of the datetime.
DatetimeProperties.minute	The minutes of the datetime.
DatetimeProperties.second	The seconds of the datetime.
DatetimeProperties.week	The week ordinal of the year.
DatetimeProperties.weekofyear	The week ordinal of the year.
DatetimeProperties.quarter	The quarter of the date.

Timedelta properties

TimedeltaProperties.days	Number of days for each element.
TimedeltaProperties.seconds	Number of seconds (>= 0 and less than 1 day) for each element.

Timedelta methods

TimedeltaProperties.total_seconds()	Return total duration of each element expressed in seconds.

3.4.12 Serialization / IO / conversion

Series.to_frame()	Convert Series to DataFrame.

rapidspy.Series.to_frame

Series.to_frame()

Convert Series to DataFrame.

Parameters

name [object, default None] The passed name should substitute for the series name (if it has one).

Returns

DataFrame DataFrame representation of Series.

Examples

```
>>> s = pd.Series(["a", "b", "c"],
...               name="vals")
>>> s.to_frame()
  vals
0    a
1    b
2    c
```

3.5 DataFrame

3.5.1 RapidsPY

DataFrame.compute()	Return a computed RapidsPY DataFrame of Series.
DataFrame.to_pandas()	Convert a RapidsPY Dataframe to Pandas Dataframe.

rapidspy.DataFrame.compute

DataFrame.compute()

Return a computed RapidsPY DataFrame of Series.

rapidspy.DataFrame.to_pandas

DataFrame.to_pandas()

Convert a RapidsPY Dataframe to Pandas Dataframe.

Returns

DataFrame

Notes

This method should only be used if the resulting pandas DataFrame is expected to be small, as all the data is loaded into the memory.

3.5.2 Attributes and underlying data

Axes

DataFrame.index	The index (row labels) of the DataFrame.
DataFrame.columns	The column labels of the DataFrame.

rapidspy.DataFrame.index

property DataFrame.index

The index (row labels) of the DataFrame.

rapidspy.DataFrame.columns

property DataFrame.columns

The column labels of the DataFrame.

DataFrame.dtypes	Return the dtypes in the DataFrame.
DataFrame.values	Return a Numpy representation of the DataFrame.

continues on next page

Table 22 – continued from previous page

DataFrame.axes	Return a list representing the axes of the DataFrame.
DataFrame.ndim	Return an int representing the number of axes / array dimensions.
DataFrame.size	Return an int representing the number of elements in this object.
DataFrame.shape	Return a tuple representing the dimensionality of the DataFrame.
DataFrame.empty	Indicator whether Series/DataFrame is empty.

rapidspy.DataFrame.dtypes

property DataFrame.dtypes

Return the dtypes in the DataFrame.

This returns a Series with the data type of each column. The result' s index is the original DataFrame' s columns. Columns with mixed types are stored with the object dtype.

Returns

pandas.Series The data type of each column.

Examples

```
>>> df = pd.DataFrame({'float': [1.0],
...                    'int': [1],
...                    'datetime': [pd.Timestamp('20180310')],
...                    'string': ['foo']})
>>> df.dtypes
float              float64
int                  int64
datetime    datetime64[ns]
string              object
dtype: object
```

rapidspy.DataFrame.values

property DataFrame.values

Return a Numpy representation of the DataFrame.

Only the values in the DataFrame will be returned, the axes labels will be removed.

Returns

numpy.ndarray The values of the DataFrame.

Notes

The dtype will be a lower-common-denominator dtype (implicit upcasting); that is to say if the dtypes (even of numeric types) are mixed, the one that accommodates all will be chosen. Use this with care if you are not dealing with the blocks.

e.g. If the dtypes are float16 and float32, dtype will be upcast to float32. If dtypes are int32 and uint8, dtype will be upcast to int32. By numpy.find_common_type() convention, mixing int64 and uint64 will result in a float64 dtype.

Examples

A DataFrame where all columns are the same type (e.g., int64) results in an array of the same type.

```
>>> df = pd.DataFrame({'age':    [ 3,  29],
...                    'height': [94, 170],
...                    'weight': [31, 115]})
>>> df
   age  height  weight
0    3      94      31
1   29     170     115
>>> df.dtypes
age       int64
height    int64
weight    int64
dtype: object
>>> df.values
array([[  3,  94,  31],
       [ 29, 170, 115]])
```

A DataFrame with mixed type columns(e.g., str/object, int64, float32) results in an ndarray of the broadest type that accommodates these mixed types (e.g., object).

```
>>> df2 = pd.DataFrame([('parrot',   24.0, 'second'),
...                     ('lion',     80.5, 1),
...                     ('monkey', np.nan, None)],
...                    columns=('name', 'max_speed', 'rank'))
>>> df2.dtypes
name          object
max_speed    float64
rank          object
dtype: object
>>> df2.values
array([['parrot', 24.0, 'second'],
       ['lion', 80.5, 1],
       ['monkey', nan, None]], dtype=object)
```

rapidspy.DataFrame.axes

property DataFrame.axes

Return a list representing the axes of the DataFrame.

It has the row axis labels and column axis labels as the only members. They are returned in that order.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.axes
[RangeIndex(start=0, stop=2, step=1), Index(['col1', 'col2'],
dtype='object')]
```

rapidspy.DataFrame.ndim

property DataFrame.ndim

Return an int representing the number of axes / array dimensions.

Return 1 if Series. Otherwise return 2 if DataFrame.

Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.ndim
1
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.ndim
2
```

rapidspy.DataFrame.size

property DataFrame.size

Return an int representing the number of elements in this object.

Return the number of rows if Series. Otherwise return the number of rows times number of columns if DataFrame.

Examples

```
>>> s = pd.Series({'a': 1, 'b': 2, 'c': 3})
>>> s.size
3
```

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.size
4
```

rapidspy.DataFrame.shape

property DataFrame.shape

Return a tuple representing the dimensionality of the DataFrame.

Examples

```
>>> df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4]})
>>> df.shape
(2, 2)
```

```
>>> df = df = pd.DataFrame({'col1': [1, 2], 'col2': [3, 4],
...                         'col3': [5, 6]})
>>> df.shape
(2, 3)
```

rapidspy.DataFrame.empty

property DataFrame.empty

Indicator whether Series/DataFrame is empty.

True if Series/DataFrame is entirely empty (no items), meaning any of the axes are of length 0.

Returns

bool If Series/DataFrame is empty, return True, if not return False.

Notes

If Series/DataFrame contains only NaNs, it is still not considered empty. See the example below.

Examples

An example of an actual empty DataFrame. Notice the index is empty:

```
>>> df_empty = pd.DataFrame({'A' : []})
>>> df_empty
Empty DataFrame
Columns: [A]
Index: []
>>> df_empty.empty
True
```

If we only have NaNs in our DataFrame, it is not considered empty! We will need to drop the NaNs to make the DataFrame empty:

```
>>> df = pd.DataFrame({'A' : [np.nan]})
>>> df
    A
0 NaN
>>> df.empty
False
>>> df.dropna().empty
True
```

```
>>> ser_empty = pd.Series({'A' : []})
>>> ser_empty
A    []
dtype: object
>>> ser_empty.empty
False
>>> ser_empty = pd.Series()
>>> ser_empty.empty
True
```

3.5.3 Conversion

DataFrame.astype(dtype)	Cast a pandas object to a specified dtype dtype.
DataFrame.copy()	Make a deep copy of this object's indices and data.

rapidspy.DataFrame.astype

DataFrame.astype(dtype)

Cast a pandas object to a specified dtype dtype.

Parameters

dtype [data type, or dict of column name -> data type] Use a numpy.dtype or Python type to cast entire pandas object to the same type. Alternatively, use {col: dtype, …}, where col is a column label and dtype is a numpy.dtype or Python type to cast one or more of the DataFrame's columns to column-specific types.

Returns

casted [same type as caller]

Examples

Create a DataFrame:

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = pd.DataFrame(data=d)
>>> df.dtypes
col1    int64
col2    int64
dtype: object
```

Cast all columns to int32:

```
>>> df.astype('int32').dtypes
col1    int32
col2    int32
dtype: object
```

Cast col1 to int32 using a dictionary:

```
>>> df.astype({'col1': 'int32'}).dtypes
col1    int32
col2    int64
dtype: object
```

Create a series:

```
>>> ser = pd.Series([1, 2], dtype='int32')
>>> ser
0    1
1    2
dtype: int32
>>> ser.astype('int64')
0    1
1    2
dtype: int64
```

Convert to categorical type:

```
>>> ser.astype('category')
0    1
1    2
```

(continues on next page)

(continued from previous page)

```
dtype: category
Categories (2, int64): [1, 2]
```

Convert to ordered categorical type with custom ordering:

```
>>> from pandas.api.types import CategoricalDtype
>>> cat_dtype = CategoricalDtype(
...     categories=[2, 1], ordered=True)
>>> ser.astype(cat_dtype)
0    1
1    2
dtype: category
Categories (2, int64): [2 < 1]
```

Create a series of dates:

```
>>> ser_date = pd.Series(pd.date_range('20200101', periods=3))
>>> ser_date
0   2020-01-01
1   2020-01-02
2   2020-01-03
dtype: datetime64[ns]
```

rapidspy.DataFrame.copy

DataFrame.copy()

Make a deep copy of this object's indices and data.

A new object will be created with a copy of the calling object's data and indices. Modifications to the data or indices of the copy will not be reflected in the original object (see notes below).

Returns

copy [Series or DataFrame] Object type matches caller.

Notes

Data is copied but actual Python objects will not be copied recursively, only the reference to the object. This is in contrast to copy.deepcopy in the Standard Library, which recursively copies object data.

While Index objects are copied, the underlying numpy array is not copied for performance reasons. Since Index is immutable, the underlying data can be safely shared and a copy is not needed.

Examples

```
>>> s = pd.Series([1, 2], index=["a", "b"])
>>> s
a    1
b    2
dtype: int64
```

```
>>> s_copy = s.copy()
>>> s_copy
a    1
b    2
dtype: int64
```

3.5.4 Indexing, iteration

DataFrame.head([n])	Return the first n rows.
DataFrame.at	Access a single value for a row/column label pair.
DataFrame.iat	Access a single value for a row/column pair by integer position.
DataFrame.loc	Access a group of rows and columns by label(s) or a boolean array.
DataFrame.iloc	Purely integer-location based indexing for selection by position.
DataFrame.keys()	Get the 'info axis' (see Indexing for more).
DataFrame.get(key[, default])	Get item from object for given key (ex: DataFrame column).

rapidspy.DataFrame.head

DataFrame.head(n=5)

Return the first n rows.

This function returns the first n rows for the object based on position. It is useful for quickly testing if your object has the right type of data in it.

For negative values of n, this function returns all rows except the last n rows, equivalent to df[:-n].

Parameters

n [int, default 5] Number of rows to select.

Returns

same type as caller The first n rows of the caller object.

Examples

```
>>> df = pd.DataFrame({'animal': ['alligator', 'bee', 'falcon', 'lion',
...                    'monkey', 'parrot', 'shark', 'whale', 'zebra']})
>>> df
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
6      shark
7      whale
8      zebra
```

Viewing the first 5 lines

```
>>> df.head()
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
```

Viewing the first n lines (three in this case)

```
>>> df.head(3)
      animal
0  alligator
1        bee
2     falcon
```

For negative values of n

```
>>> df.head(-3)
      animal
0  alligator
1        bee
2     falcon
3       lion
4     monkey
5     parrot
```

rapidspy.DataFrame.at

property DataFrame.at

Access a single value for a row/column label pair.

Similar to loc, in that both provide label-based lookups. Use at if you only need to get or set a single value in a DataFrame or Series.

Raises

KeyError If 'label' does not exist in DataFrame.

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   index=[4, 5, 6], columns=['A', 'B', 'C'])
>>> df
    A   B   C
4   0   2   3
5   0   4   1
6  10  20  30
```

Get value at specified row/column pair

```
>>> df.at[4, 'B']
2
```

Set value at specified row/column pair

```
>>> df.at[4, 'B'] = 10
>>> df.at[4, 'B']
10
```

Get value within a Series

```
>>> df.loc[5].at['B']
4
```

rapidspy.DataFrame.iat

property DataFrame.iat

Access a single value for a row/column pair by integer position.

Similar to iloc, in that both provide integer-based lookups. Use iat if you only need to get or set a single value in a DataFrame or Series.

使用 MOXE 引擎时暂不支持。

Raises

IndexError When integer position is out of bounds.

Examples

```
>>> df = pd.DataFrame([[0, 2, 3], [0, 4, 1], [10, 20, 30]],
...                   columns=['A', 'B', 'C'])
>>> df
    A   B   C
0   0   2   3
1   0   4   1
2  10  20  30
```

Get value at specified row/column pair

```
>>> df.iat[1, 2]
1
```

Set value at specified row/column pair

```
>>> df.iat[1, 2] = 10
>>> df.iat[1, 2]
10
```

Get value within a series

```
>>> df.loc[0].iat[1]
2
```

rapidspy.DataFrame.loc

property DataFrame.loc

Access a group of rows and columns by label(s) or a boolean array.

.loc[] is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

- A single label, e.g. 5 or 'a', (note that 5 is interpreted as a label of the index, and never as an integer position along the index).
- A list or array of labels, e.g. ['a', 'b', 'c'].
- An alignable boolean Series. The index of the key will be aligned before masking.

Raises

KeyError If any items are not found.

IndexingError If an indexed key is passed and its index is unalignable to the frame index.

Examples

Getting values

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=['cobra', 'viper', 'sidewinder'],
...      columns=['max_speed', 'shield'])
>>> df
            max_speed  shield
cobra               1       2
```

(continues on next page)

(continued from previous page)

```
viper           4       5
sidewinder      7       8
```

Single label. Note this returns the row as a Series.

```
>>> df.loc['viper']
max_speed    4
shield       5
Name: viper, dtype: int64
```

List of labels. Note using [[]] returns a DataFrame.

```
>>> df.loc[['viper', 'sidewinder']]
            max_speed  shield
viper           4       5
sidewinder      7       8
```

Single label for row and column

```
>>> df.loc['cobra', 'shield']
2
```

Slice with labels for row and single label for column. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc['cobra':'viper', 'max_speed']
cobra    1
viper    4
Name: max_speed, dtype: int64
```

Boolean list with the same length as the row axis

```
>>> df.loc[[False, False, True]]
            max_speed  shield
sidewinder      7       8
```

Alignable boolean Series:

```
>>> df.loc[pd.Series([False, True, False],
...        index=['viper', 'sidewinder', 'cobra'])]
            max_speed  shield
sidewinder      7       8
```

Index (same behavior as df.reindex)

```
>>> df.loc[pd.Index(["cobra", "viper"], name="foo")]
       max_speed  shield
foo
cobra          1       2
viper          4       5
```

Conditional that returns a boolean Series

```
>>> df.loc[df['shield'] > 6]
            max_speed  shield
sidewinder          7       8
```

Conditional that returns a boolean Series with column labels specified

```
>>> df.loc[df['shield'] > 6, ['max_speed']]
            max_speed
sidewinder          7
```

Callable that returns a boolean Series

```
>>> df.loc[lambda df: df['shield'] == 8]
            max_speed  shield
sidewinder          7       8
```

Setting values

Set value for all items matching the list of labels

```
>>> df.loc[['viper', 'sidewinder'], ['shield']] = 50
>>> df
            max_speed  shield
cobra               1       2
viper               4      50
sidewinder          7      50
```

Set value for an entire row

```
>>> df.loc['cobra'] = 10
>>> df
            max_speed  shield
cobra              10      10
```

(continues on next page)

(continued from previous page)

```
viper          4      50
sidewinder     7      50
```

Set value for an entire column

```
>>> df.loc[:, 'max_speed'] = 30
>>> df
            max_speed  shield
cobra              30      10
viper              30      50
sidewinder         30      50
```

Set value for rows matching callable condition

```
>>> df.loc[df['shield'] > 35] = 0
>>> df
            max_speed  shield
cobra              30      10
viper               0       0
sidewinder          0       0
```

Getting values on a DataFrame with an index that has integer labels

Another example using integers for the index

```
>>> df = pd.DataFrame([[1, 2], [4, 5], [7, 8]],
...      index=[7, 8, 9], columns=['max_speed', 'shield'])
>>> df
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Slice with integer labels for rows. As mentioned above, note that both the start and stop of the slice are included.

```
>>> df.loc[7:9]
   max_speed  shield
7          1       2
8          4       5
9          7       8
```

Getting values with a MultiIndex

A number of examples using a DataFrame with a MultiIndex

```
>>> tuples = [
...     ('cobra', 'mark i'), ('cobra', 'mark ii'),
...     ('sidewinder', 'mark i'), ('sidewinder', 'mark ii'),
...     ('viper', 'mark ii'), ('viper', 'mark iii')
... ]
>>> index = pd.MultiIndex.from_tuples(tuples)
>>> values = [[12, 2], [0, 4], [10, 20],
...           [1, 4], [7, 1], [16, 36]]
>>> df = pd.DataFrame(values, columns=['max_speed', 'shield'], index=index)
>>> df
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Single label. Note this returns a DataFrame with a single index.

```
>>> df.loc['cobra']
         max_speed  shield
mark i          12       2
mark ii          0       4
```

Single index tuple. Note this returns a Series.

```
>>> df.loc[('cobra', 'mark ii')]
max_speed    0
shield       4
Name: (cobra, mark ii), dtype: int64
```

Single label for row and column. Similar to passing in a tuple, this returns a Series.

```
>>> df.loc['cobra', 'mark i']
max_speed    12
shield        2
Name: (cobra, mark i), dtype: int64
```

Single tuple. Note using [[]] returns a DataFrame.

```
>>> df.loc[[('cobra', 'mark ii')]]
               max_speed  shield
cobra mark ii          0       4
```

Single tuple for the index with a single label for the column

```
>>> df.loc[('cobra', 'mark i'), 'shield']
2
```

Slice from index tuple to single label

```
>>> df.loc[('cobra', 'mark i'):'viper']
                     max_speed  shield
cobra      mark i           12       2
           mark ii           0       4
sidewinder mark i           10      20
           mark ii           1       4
viper      mark ii           7       1
           mark iii         16      36
```

Slice from index tuple to index tuple

```
>>> df.loc[('cobra', 'mark i'):('viper', 'mark ii')]
                    max_speed  shield
cobra      mark i          12       2
           mark ii          0       4
sidewinder mark i          10      20
           mark ii          1       4
viper      mark ii          7       1
```

rapidspy.DataFrame.iloc

property DataFrame.iloc

Purely integer-location based indexing for selection by position.

.iloc[] is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. 5.

- A list or array of integers, e.g. [4, 3, 0].
- A slice object with ints, e.g. 1:7.
- A boolean array.

.iloc will raise IndexError if a requested indexer is out-of-bounds, except slice indexers which allow out-of-bounds indexing (this conforms with python/numpy slice semantics).

Examples

```
>>> mydict = [{'a': 1, 'b': 2, 'c': 3, 'd': 4},
...           {'a': 100, 'b': 200, 'c': 300, 'd': 400},
...           {'a': 1000, 'b': 2000, 'c': 3000, 'd': 4000 }]
>>> df = pd.DataFrame(mydict)
>>> df
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

Indexing just the rows

With a scalar integer.

```
>>> type(df.iloc[0])
<class 'pandas.core.series.Series'>
>>> df.iloc[0]
a    1
b    2
c    3
d    4
Name: 0, dtype: int64
```

With a list of integers.

```
>>> df.iloc[[0]]
   a  b  c  d
0  1  2  3  4
>>> type(df.iloc[[0]])
<class 'pandas.core.frame.DataFrame'>
```

```
>>> df.iloc[[0, 1]]
     a    b    c    d
0    1    2    3    4
1  100  200  300  400
```

With a slice object.

```
>>> df.iloc[:3]
      a     b     c     d
0     1     2     3     4
1   100   200   300   400
2  1000  2000  3000  4000
```

With a boolean mask the same length as the index.

```
>>> df.iloc[[True, False, True]]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

With a callable, useful in method chains. The x passed to the lambda is the DataFrame being sliced. This selects the rows whose index label even.

```
>>> df.iloc[lambda x: x.index % 2 == 0]
      a     b     c     d
0     1     2     3     4
2  1000  2000  3000  4000
```

Indexing both axes

You can mix the indexer types for the index and columns. Use : to select the entire axis.

With scalar integers.

```
>>> df.iloc[0, 1]
2
```

With lists of integers.

```
>>> df.iloc[[0, 2], [1, 3]]
      b     d
0     2     4
2  2000  4000
```

With slice objects.

```
>>> df.iloc[1:3, 0:3]
      a     b     c
1   100   200   300
2  1000  2000  3000
```

With a boolean array whose length matches the columns.

```
>>> df.iloc[:, [True, False, True, False]]
      a     c
0     1     3
1   100   300
2  1000  3000
```

With a callable function that expects the Series or DataFrame.

```
>>> df.iloc[:, lambda df: [0, 2]]
      a     c
0     1     3
1   100   300
2  1000  3000
```

rapidspy.DataFrame.keys

DataFrame.keys()

Get the 'info axis' (see Indexing for more).

This is index for Series, columns for DataFrame.

Returns

Index Info axis.

rapidspy.DataFrame.get

DataFrame.get(key, default=None)

Get item from object for given key (ex: DataFrame column).

Returns default value if not found.

Parameters

key [object]

Returns

value [same type as items contained in object]

Examples

```
>>> df = pd.DataFrame(
...     [
...         [24.3, 75.7, "high"],
...         [31, 87.8, "high"],
...         [22, 71.6, "medium"],
...         [35, 95, "medium"],
...     ],
...     columns=["temp_celsius", "temp_fahrenheit", "windspeed"],
...     index=pd.date_range(start="2014-02-12", end="2014-02-15", freq="D"),
... )
```

```
>>> df
            temp_celsius  temp_fahrenheit windspeed
2014-02-12          24.3             75.7      high
2014-02-13          31.0             87.8      high
2014-02-14          22.0             71.6    medium
2014-02-15          35.0             95.0    medium
```

```
>>> df.get(["temp_celsius", "windspeed"])
            temp_celsius windspeed
2014-02-12          24.3      high
2014-02-13          31.0      high
2014-02-14          22.0    medium
2014-02-15          35.0    medium
```

If the key isn't found, the default value will be used.

```
>>> df.get(["temp_celsius", "temp_kelvin"], default="default_value")
'default_value'
```

3.5.5 Binary operator functions

DataFrame.add(other)	Get Addition of dataframe and other, element-wise (binary operator add).
DataFrame.sub(other)	Get Subtraction of dataframe and other, element-wise (binary operator sub).
DataFrame.mul(other)	Get Multiplication of dataframe and other, element-wise (binary operator mul).
DataFrame.div(other)	Get Floating division of dataframe and other, element-wise (binary operator truediv).
DataFrame.truediv(other)	Get Floating division of dataframe and other, element-wise (binary operator truediv).
DataFrame.floordiv(other)	Get Integer division of dataframe and other, element-wise (binary operator floordiv).
DataFrame.pow(other)	Get Exponential power of dataframe and other, element-wise (binary operator pow).
DataFrame.radd(other)	Get Addition of dataframe and other, element-wise (binary operator radd).
DataFrame.rsub(other)	Get Subtraction of dataframe and other, element-wise (binary operator rsub).
DataFrame.lt(other)	Get Less than of dataframe and other, element-wise (binary operator lt).
DataFrame.gt(other)	Get Greater than of dataframe and other, element-wise (binary operator gt).
DataFrame.le(other)	Get Less than or equal to of dataframe and other, element-wise (binary operator le).
DataFrame.ge(other)	Get Greater than or equal to of dataframe and other, element-wise (binary operator ge).
DataFrame.ne(other)	Get Not equal to of dataframe and other, element-wise (binary operator ne).
DataFrame.eq(other)	Get Equal to of dataframe and other, element-wise (binary operator eq).

rapidspy.DataFrame.add

DataFrame.add(other)

Get Addition of dataframe and other, element-wise (binary operator add).

Equivalent to dataframe + other. With reverse version, radd.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, , /, //, %, *.

Parameters

other [scalar] Any single element.

Returns

DataFrame Result of the arithmetic operation.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({{'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]}},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
```

(continues on next page)

(continued from previous page)

```
triangle        4      181
rectangle       5      361
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2])
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({{'angles': [0, 3, 4]}},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other)
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({{'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]}},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

rapidspy.DataFrame.sub

DataFrame.sub(other)

Get Subtraction of dataframe and other, element-wise (binary operator sub).

Equivalent to dataframe - other. With reverse version, rsub.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, , /, //, %, *.

Parameters

other [scalar] Any single element.

Returns

DataFrame Result of the arithmetic operation.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({{'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]}},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

```
>>> df.sub([1, 2])
           angles  degrees
circle         -1      358
triangle        2      178
rectangle       3      358
```

rapidspy.DataFrame.mul

DataFrame.mul(other)

Get Multiplication of dataframe and other, element-wise (binary operator mul).

Equivalent to dataframe * other. With reverse version, rmul.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, , /, //, %, *.

Parameters

other [scalar] Any single element.

Returns

DataFrame Result of the arithmetic operation.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({{'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]}},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Multiply a DataFrame of different shape with operator version.

```
>>> other = pd.DataFrame({{'angles': [0, 3, 4]}},
...                      index=['circle', 'triangle', 'rectangle'])
>>> other
           angles
circle          0
triangle        3
rectangle       4
```

```
>>> df * other
           angles  degrees
circle          0      NaN
triangle        9      NaN
rectangle      16      NaN
```

```
>>> df.mul(other)
           angles  degrees
```

(continues on next page)

(continued from previous page)

```
circle          0       NaN
triangle        9       NaN
rectangle      16       NaN
```

rapidspy.DataFrame.div

DataFrame.div(other)

Get Floating division of dataframe and other, element-wise (binary operator truediv).

Equivalent to dataframe / other. With reverse version, rtruediv.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, , /, //, %, *.

Parameters

other [scalar] Any single element.

Returns

DataFrame Result of the arithmetic operation.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({{'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]}},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
```

(continues on next page)

(continued from previous page)

```
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
B square          4      360
  pentagon        5      540
  hexagon         6      720
```

rapidspy.DataFrame.truediv

DataFrame.truediv(other)

Get Floating division of dataframe and other, element-wise (binary operator truediv).

Equivalent to dataframe / other. With reverse version, rtruediv.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, , /, //, %, *.

Parameters

other [scalar] Any single element.

Returns

DataFrame Result of the arithmetic operation.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({{'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]}},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Divide by constant with reverse version.

```
>>> df.div(10)
           angles  degrees
circle        0.0     36.0
triangle      0.3     18.0
rectangle     0.4     36.0
```

```
>>> df.rdiv(10)
             angles   degrees
circle          inf  0.027778
triangle   3.333333  0.055556
rectangle  2.500000  0.027778
```

Divide by a MultiIndex by level.

```
>>> df_multindex = pd.DataFrame({{'angles': [0, 3, 4, 4, 5, 6],
...                              'degrees': [360, 180, 360, 360, 540, 720]}},
...                             index=[['A', 'A', 'A', 'B', 'B', 'B'],
...                                    ['circle', 'triangle', 'rectangle',
...                                     'square', 'pentagon', 'hexagon']])
>>> df_multindex
             angles  degrees
A circle          0      360
  triangle        3      180
  rectangle       4      360
```

(continues on next page)

(continued from previous page)

```
B square        4      360
  pentagon      5      540
  hexagon       6      720
```

rapidspy.DataFrame.floordiv

DataFrame.floordiv(other)

Get Integer division of dataframe and other, element-wise (binary operator floordiv).

Equivalent to dataframe // other. With reverse version, rfloordiv.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, , /, //, %, *.

Parameters

other [scalar] Any single element.

Returns

DataFrame Result of the arithmetic operation.

Notes

Mismatched indices will be unioned together.

rapidspy.DataFrame.pow

DataFrame.pow(other)

Get Exponential power of dataframe and other, element-wise (binary operator pow).

Equivalent to dataframe ** other. With reverse version, rpow.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, , /, //, %, *.

Parameters

other [scalar] Any single element.

Returns

DataFrame Result of the arithmetic operation.

Notes

Mismatched indices will be unioned together.

rapidspy.DataFrame.radd

DataFrame.radd(other)

Get Addition of dataframe and other, element-wise (binary operator radd).

Equivalent to other + dataframe. With reverse version, add.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, , /, //, %, *.

Parameters

other [scalar] Any single element.

Returns

DataFrame Result of the arithmetic operation.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({{'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]}},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Add a scalar with operator version which return the same results.

```
>>> df + 1
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

```
>>> df.add(1)
           angles  degrees
circle          1      361
triangle        4      181
rectangle       5      361
```

rapidspy.DataFrame.rsub

DataFrame.rsub(other)

Get Subtraction of dataframe and other, element-wise (binary operator rsub).

Equivalent to other - dataframe. With reverse version, sub.

Among flexible wrappers (add, sub, mul, div, mod, pow) to arithmetic operators: +, -, , /, //, %, *.

Parameters

other [scalar] Any single element.

Returns

DataFrame Result of the arithmetic operation.

Notes

Mismatched indices will be unioned together.

Examples

```
>>> df = pd.DataFrame({{'angles': [0, 3, 4],
...                    'degrees': [360, 180, 360]}},
...                   index=['circle', 'triangle', 'rectangle'])
>>> df
           angles  degrees
circle          0      360
triangle        3      180
rectangle       4      360
```

Subtract a list and Series by axis with operator version.

```
>>> df - [1, 2]
           angles  degrees
```

(continues on next page)

(continued from previous page)

```
circle        -1       358
triangle       2       178
rectangle      3       358
```

```
>>> df.sub([1, 2])
           angles  degrees
circle        -1       358
triangle       2       178
rectangle      3       358
```

rapidspy.DataFrame.lt

DataFrame.lt(other)

Get Less than of dataframe and other, element-wise (binary operator lt).

Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.

Equivalent to ==, !=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other [scalar] Any single element.

Returns

DataFrame of bool Result of the comparison.

Notes

Mismatched indices will be unioned together. NaN values are considered different (i.e. NaN != NaN).

rapidspy.DataFrame.gt

DataFrame.gt(other)

Get Greater than of dataframe and other, element-wise (binary operator gt).

Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.

Equivalent to ==, !=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other [scalar] Any single element.

Returns

DataFrame of bool Result of the comparison.

Notes

Mismatched indices will be unioned together. NaN values are considered different (i.e. NaN != NaN).

Examples

```
>>> df = pd.DataFrame({{'cost': [250, 150, 100],
...                    'revenue': [100, 250, 300]}},
...                   index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Compare to a DataFrame of different shape.

```
>>> other = pd.DataFrame({{'revenue': [300, 250, 100, 150]}},
...                      index=['A', 'B', 'C', 'D'])
>>> other
revenue
A     300
B     250
C     100
D     150
```

```
>>> df.gt(other)
    cost  revenue
A  False    False
B  False    False
C  False     True
D  False    False
```

rapidspy.DataFrame.le

DataFrame.le(other)

Get Less than or equal to of dataframe and other, element-wise (binary operator le).

Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.

Equivalent to ==, !=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other [scalar] Any single element.

Returns

DataFrame of bool Result of the comparison.

Notes

Mismatched indices will be unioned together. NaN values are considered different (i.e. NaN != NaN).

rapidspy.DataFrame.ge

DataFrame.ge(other)

Get Greater than or equal to of dataframe and other, element-wise (binary operator ge).

Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.

Equivalent to ==, !=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other [scalar] Any single element.

Returns

DataFrame of bool Result of the comparison.

Notes

Mismatched indices will be unioned together. NaN values are considered different (i.e. NaN != NaN).

rapidspy.DataFrame.ne

DataFrame.ne(other)

Get Not equal to of dataframe and other, element-wise (binary operator ne).

Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.

Equivalent to ==, !=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other [scalar] Any single element.

Returns

DataFrame of bool Result of the comparison.

Notes

Mismatched indices will be unioned together. NaN values are considered different (i.e. NaN != NaN).

rapidspy.DataFrame.eq

DataFrame.eq(other)

Get Equal to of dataframe and other, element-wise (binary operator eq).

Among flexible wrappers (eq, ne, le, lt, ge, gt) to comparison operators.

Equivalent to ==, !=, <=, <, >=, > with support to choose axis (rows or columns) and level for comparison.

Parameters

other [scalar] Any single element.

Returns

DataFrame of bool Result of the comparison.

Notes

Mismatched indices will be unioned together. NaN values are considered different (i.e. NaN != NaN).

Examples

```
>>> df = pd.DataFrame({{'cost': [250, 150, 100],
...                     'revenue': [100, 250, 300]}},
...                    index=['A', 'B', 'C'])
>>> df
   cost  revenue
A   250      100
B   150      250
C   100      300
```

Comparison with a scalar, using either the operator or method:

```
>>> df == 100
    cost  revenue
A  False     True
B  False    False
C   True    False
```

```
>>> df.eq(100)
    cost  revenue
A  False     True
B  False    False
C   True    False
```

3.5.6 Function application, GroupBy & window

DataFrame.agg(func)	Aggregate using one or more operations over the specified axis.
DataFrame.aggregate(func)	Aggregate using one or more operations over the specified axis.
DataFrame.groupby([by])	Group DataFrame using a mapper or by a Series of columns.

rapidspy.DataFrame.agg

DataFrame.agg(func)

Aggregate using one or more operations over the specified axis.

Only float, int, boolean columns will be computed.

Parameters

func [str, list or dict] Function to use for aggregating the data.

Accepted combinations are:

- string function name
- list of function names, e.g. ['sum', 'mean']

Returns

scalar, Series or DataFrame The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

The aggregation operations are always performed over an axis, either the

index (default) or the column axis. This behavior is different from

numpy aggregation functions (mean, median, prod, sum, std,

var), where the default is to compute the aggregation of the flattened

array, e.g., numpy.mean(arr_2d) as opposed to

numpy.mean(arr_2d, axis=0).

agg is an alias for aggregate. Use the alias.

Notes

agg is an alias for aggregate. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                    columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
sum  12.0  NaN
min   1.0  2.0
max   NaN  8.0
```

Aggregate different functions over the columns and rename the index of the resulting DataFrame.

```
>>> df.agg(x=('A', max), y=('B', 'min'), z=('C', np.mean))
     A    B    C
x  7.0  NaN  NaN
y  NaN  2.0  NaN
z  NaN  NaN  6.0
```

rapidspy.DataFrame.aggregate

DataFrame.aggregate(func)

Aggregate using one or more operations over the specified axis.

Only float, int, boolean columns will be computed.

Parameters

func [str, list or dict] Function to use for aggregating the data.

Accepted combinations are:

- string function name
- list of function names, e.g. ['sum', 'mean']

Returns

scalar, Series or DataFrame The return can be:

- scalar : when Series.agg is called with single function
- Series : when DataFrame.agg is called with a single function
- DataFrame : when DataFrame.agg is called with several functions

Return scalar, Series or DataFrame.

The aggregation operations are always performed over an axis, either the

index (default) or the column axis. This behavior is different from

numpy aggregation functions (mean, median, prod, sum, std,

var), where the default is to compute the aggregation of the flattened

array, e.g., numpy.mean(arr_2d) as opposed to

numpy.mean(arr_2d, axis=0).

agg is an alias for aggregate. Use the alias.

Notes

agg is an alias for aggregate. Use the alias.

Functions that mutate the passed object can produce unexpected behavior or errors and are not supported. See Mutating with User Defined Function (UDF) methods for more details.

A passed user-defined-function will be passed a Series for evaluation.

Examples

```
>>> df = pd.DataFrame([[1, 2, 3],
...                    [4, 5, 6],
...                    [7, 8, 9],
...                    [np.nan, np.nan, np.nan]],
...                   columns=['A', 'B', 'C'])
```

Aggregate these functions over the rows.

```
>>> df.agg(['sum', 'min'])
        A     B     C
sum  12.0  15.0  18.0
min   1.0   2.0   3.0
```

Different aggregations per column.

```
>>> df.agg({'A' : ['sum', 'min'], 'B' : ['min', 'max']})
        A    B
sum  12.0  NaN
min   1.0  2.0
max   NaN  8.0
```

Aggregate different functions over the columns and rename the index of the resulting DataFrame.

```
>>> df.agg(x=('A', max), y=('B', 'min'), z=('C', np.mean))
     A    B    C
x  7.0  NaN  NaN
y  NaN  2.0  NaN
z  NaN  NaN  6.0
```

rapidspy.DataFrame.groupby

DataFrame.groupby(by=None)

Group DataFrame using a mapper or by a Series of columns.

A groupby operation involves some combination of splitting the object, applying a function, and combining the results. This can be used to group large amounts of data and compute operations on these groups.

Parameters

by: label, or list of labels A label or list of labels may be passed to group by the columns in self. Notice that a tuple is interpreted as a (single) key.

Returns

——-

DataFrameGroupBy Returns a groupby object that contains information about the groups.

Examples

```
>>> df = pd.DataFrame({'Animal': ['Falcon', 'Falcon',
...                               'Parrot', 'Parrot'],
...                    'Max Speed': [380., 370., 24., 26.]})
>>> df
   Animal  Max Speed
0  Falcon      380.0
1  Falcon      370.0
2  Parrot       24.0
3  Parrot       26.0
>>> df.groupby(['Animal']).mean()
        Max Speed
Animal
Falcon      375.0
Parrot       25.0
```

We can also choose to include NA in group keys

```
>>> l = [[1, 2, 3], [1, None, 4], [2, 1, 3], [1, 2, 2]]
>>> df = pd.DataFrame(l, columns=["a", "b", "c"])
```

```
>>> df.groupby(by=["b"]).sum()
    a   c
b
1.0 2   3
2.0 2   5
```

```
>>> l = [["a", 12, 12], [None, 12.3, 33.], ["b", 12.3, 123], ["a", 1, 1]]
>>> df = pd.DataFrame(l, columns=["a", "b", "c"])
```

```
>>> df.groupby(by="a").sum()
    b     c
a
a   13.0   13.0
b   12.3  123.0
```

3.5.7 Computations / descriptive stats

DataFrame.abs()	Return a Series/DataFrame with absolute numeric value of each element.
DataFrame.all()	Return unbiased variance over requested axis.
DataFrame.any()	Return whether any element is True, potentially over an axis.
DataFrame.count()	Count non-NA cells for each column or row.
DataFrame.describe()	Generate descriptive statistics.
DataFrame.max()	Return the maximum of the values over the requested axis.
DataFrame.mean()	Return the mean of the values over the requested axis.
DataFrame.median()	Return the median of the values over the requested axis.
DataFrame.min()	Return the minimum of the values over the requested axis.
DataFrame.round([decimals])	Round a DataFrame to a variable number of decimal places.
DataFrame.sum()	Return the sum of the values over the requested axis.
DataFrame.std()	Return sample standard deviation over requested axis.
DataFrame.var()	Return unbiased variance over requested axis.
DataFrame.nunique()	Count number of distinct elements in specified axis.
DataFrame.value_counts()	Return a Series containing counts of unique rows in the DataFrame.

rapidspy.DataFrame.abs

DataFrame.abs()

Return a Series/DataFrame with absolute numeric value of each element.

This function only applies to elements that are all numeric.

Returns

abs Series/DataFrame containing the absolute value of each element.

Notes

For complex inputs, 1.2 + 1j, the absolute value is $\sqrt{a^2 + b^2}$.

Examples

Absolute numeric values in a Series.

```
>>> s = pd.Series([-1.10, 2, -3.33, 4])
>>> s.abs()
0    1.10
1    2.00
2    3.33
3    4.00
dtype: float64
```

Absolute numeric values in a Series with complex numbers.

```
>>> s = pd.Series([1.2 + 1j])
>>> s.abs()
0    1.56205
dtype: float64
```

Absolute numeric values in a Series with a Timedelta element.

```
>>> s = pd.Series([pd.Timedelta('1 days')])
>>> s.abs()
0   1 days
dtype: timedelta64[ns]
```

Select rows with data closest to certain value using argsort (from StackOverflow).

```
>>> df = pd.DataFrame({
...     'a': [4, 5, 6, 7],
...     'b': [10, 20, 30, 40],
...     'c': [100, 50, -30, -50]
... })
>>> df
     a    b    c
0    4   10  100
1    5   20   50
2    6   30  -30
```

(continues on next page)

(continued from previous page)

```
3    7    40  -50
>>> df.loc[(df.c - 43).abs().argsort()]
     a    b    c
1    5    20   50
0    4    10  100
2    6    30  -30
3    7    40  -50
```

rapidspy.DataFrame.all

DataFrame.all()

Return unbiased variance over requested axis.

Returns True unless there at least one element within a series or along a Dataframe axis that is False or equivalent (e.g. zero or empty).

Returns

Series

Examples

Series

```
>>> pd.Series([True, True]).all()
True
>>> pd.Series([True, False]).all()
False
>>> pd.Series([], dtype="float64").all()
True
>>> pd.Series([np.nan]).all()
True
```

DataFrames

Create a dataframe from a dictionary.

```
>>> df = pd.DataFrame({'col1': [True, True], 'col2': [True, False]})
>>> df
   col1   col2
0  True   True
1  True  False
```

Default behaviour checks if column-wise values all return True.

```
>>> df.all()
col1     True
col2    False
dtype: bool
```

rapidspy.DataFrame.any

DataFrame.any()

Return whether any element is True, potentially over an axis.

Returns False unless there is at least one element within a series or along a Dataframe axis that is True or equivalent (e.g. non-zero or non-empty).

Returns

Series

Examples

Series

For Series input, the output is a scalar indicating whether any element is True.

```
>>> pd.Series([False, False]).any()
False
>>> pd.Series([True, False]).any()
True
>>> pd.Series([]).any()
False
>>> pd.Series([np.nan]).any()
False
```

DataFrame

Whether each column contains at least one True element (the default).

```
>>> df = pd.DataFrame({"A": [1, 2], "B": [0, 2], "C": [0, 0]})
>>> df
   A  B  C
0  1  0  0
1  2  2  0
```

```
>>> df.any()
A     True
B     True
C    False
dtype: bool
```

any for an empty DataFrame is an empty Series.

```
>>> pd.DataFrame([]).any()
Series([], dtype: bool)
```

rapidspy.DataFrame.count

DataFrame.count()

Count non-NA cells for each column or row.

The values None, NaN, NaT, and optionally numpy.inf (depending on pandas.options.mode.use_inf_as_na) are considered NA.

Returns

Series or DataFrame For each column/row the number of non-NA/null entries. If level is specified returns a DataFrame.

Examples

Constructing DataFrame from a dictionary:

```
>>> df = pd.DataFrame({"Person":
...                    ["John", "Myla", "Lewis", "John", "Myla"],
...                    "Age": [24., np.nan, 21., 33, 26],
...                    "Single": [False, True, True, True, False]})
>>> df
  Person   Age  Single
0   John  24.0   False
1   Myla   NaN    True
2  Lewis  21.0    True
3   John  33.0    True
4   Myla  26.0   False
```

Notice the uncounted NA values:

```
>>> df.count()
Person    5
Age       4
Single    5
dtype: int64
```

rapidspy.DataFrame.describe

DataFrame.describe()

Generate descriptive statistics.

Descriptive statistics include those that summarize the central tendency, dispersion and shape of a dataset’s distribution, excluding NaN values.

Analyzes both numeric and object series, as well as DataFrame column sets of mixed data types. The output will vary depending on what is provided. Refer to the notes below for more detail.

使用 MOXE 引擎时暂不支持。

Returns

Series or DataFrame Summary statistics of the Series or Dataframe provided.

Notes

For numeric data, the result’s index will include count, mean, std, min, max as well as lower, 50 and upper percentiles. By default the lower percentile is 25 and the upper percentile is 75. The 50 percentile is the same as the median.

For object data (e.g. strings or timestamps), the result’s index will include count, unique, top, and freq. The top is the most common value. The freq is the most common value’s frequency. Timestamps also include the first and last items.

If multiple object values have the highest count, then the count and top results will be arbitrarily chosen from among those with the highest count.

For mixed data types provided via a DataFrame, the default is to return only an analysis of numeric columns. If the dataframe consists only of object and categorical data without any numeric columns, the default is to return an analysis of both the object and categorical columns.

Examples

Describing a numeric Series.

```
>>> s = pd.Series([1, 2, 3])
>>> s.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
dtype: float64
```

Describing a categorical Series.

```
>>> s = pd.Series(['a', 'a', 'b', 'c'])
>>> s.describe()
count     4
unique    3
top       a
freq      2
dtype: object
```

Describing a DataFrame. By default only numeric fields are returned.

```
>>> df = pd.DataFrame({'categorical': pd.Categorical(['d','e','f']),
...                    'numeric': [1, 2, 3],
...                    'object': ['a', 'b', 'c']
...                   })
>>> df.describe()
       numeric
count      3.0
mean       2.0
std        1.0
min        1.0
25%        1.5
50%        2.0
75%        2.5
max        3.0
```

Describing a column from a DataFrame by accessing it as an attribute.

```
>>> df.numeric.describe()
count    3.0
mean     2.0
std      1.0
min      1.0
25%      1.5
50%      2.0
75%      2.5
max      3.0
Name: numeric, dtype: float64
```

rapidspy.DataFrame.max

DataFrame.max()

Return the maximum of the values over the requested axis.

If you want the index of the maximum, use idxmax. This is the equivalent of the numpy.ndarray method argmax.

Returns

Series

Notes

Only float, int, boolean columns will be computed.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
```

(continues on next page)

(continued from previous page)

```
        spider      8
Name: legs, dtype: int64
```

```
>>> s.max()
8
```

rapidspy.DataFrame.mean

DataFrame.mean()

Return the mean of the values over the requested axis.

Returns

Series

Notes

Only float, int, boolean columns will be computed.

rapidspy.DataFrame.median

DataFrame.median()

Return the median of the values over the requested axis.

使用 MOXE 引擎时暂不支持。

Returns

Series

Notes

Only float, int, boolean columns will be computed.

rapidspy.DataFrame.min

DataFrame.min()

Return the minimum of the values over the requested axis.

If you want the index of the minimum, use idxmin. This is the equivalent of the numpy.ndarray method argmin.

Returns

Series For each column/row the number of non-NA/null entries.

Notes

Only float, int, boolean columns will be computed.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.min()
0
```

rapidspy.DataFrame.round

DataFrame.round(decimals=0)

Round a DataFrame to a variable number of decimal places.

Parameters

decimals [int, default 0] Number of decimal places to round to. If decimals is negative, it specifies the number of positions to the left of the decimal point.

Returns

DataFrame A DataFrame with the affected columns rounded to the specified number of decimal places.

Examples

```
>>> df = pd.DataFrame([(.21, .32), (.01, .67), (.66, .03), (.21, .18)],
...                   columns=['dogs', 'cats'])
>>> df
    dogs  cats
0  0.21  0.32
1  0.01  0.67
2  0.66  0.03
3  0.21  0.18
```

By providing an integer each column is rounded to the same number of decimal places

```
>>> df.round(1)
    dogs  cats
0   0.2   0.3
1   0.0   0.7
2   0.7   0.0
3   0.2   0.2
```

With a dict, the number of places for specific columns can be specified with the column names as key and the number of decimal places as value

```
>>> df.round({'dogs': 1, 'cats': 0})
    dogs  cats
0   0.2   0.0
1   0.0   1.0
2   0.7   0.0
3   0.2   0.0
```

Using a Series, the number of places for specific columns can be specified with the column names as index and the number of decimal places as value

```
>>> decimals = pd.Series([0, 1], index=['cats', 'dogs'])
>>> df.round(decimals)
    dogs  cats
0   0.2   0.0
1   0.0   1.0
2   0.7   0.0
3   0.2   0.0
```

rapidspy.DataFrame.sum

DataFrame.sum()

Return the sum of the values over the requested axis.

This is equivalent to the method numpy.sum.

Returns

Series

Notes

Only float, int, boolean columns will be computed.

Examples

```
>>> idx = pd.MultiIndex.from_arrays([
...     ['warm', 'warm', 'cold', 'cold'],
...     ['dog', 'falcon', 'fish', 'spider']],
...     names=['blooded', 'animal'])
>>> s = pd.Series([4, 2, 0, 8], name='legs', index=idx)
>>> s
blooded  animal
warm     dog       4
         falcon    2
cold     fish      0
         spider    8
Name: legs, dtype: int64
```

```
>>> s.sum()
14
```

By default, the sum of an empty or all-NA Series is 0.

```
>>> pd.Series([], dtype="float64").sum()
0.0
```

rapidspy.DataFrame.std

DataFrame.std()

Return sample standard deviation over requested axis.

Normalized by N-1 by default.

使用 MOXE 引擎时暂不支持。

Returns

Series

Notes

Only float, int, boolean columns will be computed.

Examples

```
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]}
...                   ).set_index('person_id')
>>> df
           age  height
person_id
0           21    1.61
1           25    1.87
2           62    1.49
3           43    2.01
```

The standard deviation of the columns can be found as follows:

```
>>> df.std()
age       18.786076
height     0.237417
```

rapidspy.DataFrame.var

DataFrame.var()

Return unbiased variance over requested axis.

Normalized by N-1 by default.

使用 MOXE 引擎时暂不支持。

Returns

Series

Notes

Only float, int, boolean columns will be computed.

Examples

```
>>> df = pd.DataFrame({'person_id': [0, 1, 2, 3],
...                    'age': [21, 25, 62, 43],
...                    'height': [1.61, 1.87, 1.49, 2.01]}
...                   ).set_index('person_id')
>>> df
           age  height
person_id
0           21    1.61
1           25    1.87
2           62    1.49
3           43    2.01
```

```
>>> df.var()
age       352.916667
height      0.056367
```

rapidspy.DataFrame.nunique

DataFrame.nunique()

Count number of distinct elements in specified axis.

Return Series with number of distinct elements. Can ignore NaN values.

Returns

Series

Examples

```
>>> df = pd.DataFrame({'A': [4, 5, 6], 'B': [4, 1, 1]})
>>> df.nunique()
A    3
B    2
dtype: int64
```

rapidspy.DataFrame.value_counts

DataFrame.value_counts()

Return a Series containing counts of unique rows in the DataFrame.

Returns

Series

Notes

The returned Series will have a MultiIndex with one level per input column. By default, rows that contain any NA values are omitted from the result. By default, the resulting Series will be in descending order so that the first element is the most frequently-occurring row.

Examples

```
>>> df = pd.DataFrame({'num_legs': [2, 4, 4, 6],
...                    'num_wings': [2, 0, 0, 0]},
...                   index=['falcon', 'dog', 'cat', 'ant'])
>>> df
        num_legs  num_wings
```

(continues on next page)

(continued from previous page)

```
falcon          2           2
dog             4           0
cat             4           0
ant             6           0
```

```
>>> df.value_counts()
num_legs  num_wings
4         0            2
2         2            1
6         0            1
dtype: int64
```

3.5.8 Reindexing / selection / label manipulation

DataFrame.drop([labels, axis, index, columns])	Drop specified labels from rows or columns.
DataFrame.drop_duplicates()	Return DataFrame with duplicate rows removed.
DataFrame.filter([items, like, axis])	Subset the dataframe rows or columns according to the specified index labels.
DataFrame.head([n])	Return the first n rows.
DataFrame.rename(columns)	Alter axes labels.
DataFrame.reset_index()	Reset the index, or a level of it.

rapidspy.DataFrame.drop

DataFrame.drop(labels=None, axis=0, index=None, columns=None)

Drop specified labels from rows or columns.

Remove rows or columns by specifying label names and corresponding axis, or by specifying directly index or column names. When using a multi-index, labels on different levels can be removed by specifying the level.

Parameters

labels [single label or list-like] Index or column labels to drop.

axis [{0 or 'index', 1 or 'columns'}, default 0] Whether to drop labels from the index (0 or 'index') or columns (1 or 'columns').

index [single label or list-like] Alternative to specifying axis (labels, axis=0 is equivalent to index=labels).

columns [single label or list-like] Alternative to specifying axis (labels, axis=1 is equivalent to columns=labels).

Returns

DataFrame DataFrame without the removed index or column labels.

Raises

KeyError If any of the labels is not found in the selected axis.

Examples

```
>>> df = pd.DataFrame(np.arange(12).reshape(3, 4),
...                   columns=['A', 'B', 'C', 'D'])
>>> df
   A  B   C   D
0  0  1   2   3
1  4  5   6   7
2  8  9  10  11
```

Drop columns

```
>>> df.drop(['B', 'C'], axis=1)
   A   D
0  0   3
1  4   7
2  8  11
```

```
>>> df.drop(columns=['B', 'C'])
   A   D
0  0   3
1  4   7
2  8  11
```

Drop a row by index

```
>>> df.drop([0, 1])
   A  B   C   D
2  8  9  10  11
```

Drop columns and/or rows of MultiIndex DataFrame

```
>>> midx = pd.MultiIndex(levels=[['lama', 'cow', 'falcon'],
...                              ['speed', 'weight', 'length']],
...                      codes=[[0, 0, 0, 1, 1, 1, 2, 2, 2],
...                             [0, 1, 2, 0, 1, 2, 0, 1, 2]])
>>> df = pd.DataFrame(index=midx, columns=['big', 'small'],
...                   data=[[45, 30], [200, 100], [1.5, 1], [30, 20],
...                         [250, 150], [1.5, 0.8], [320, 250],
...                         [1, 0.8], [0.3, 0.2]])
>>> df
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        weight  1.0     0.8
        length  0.3     0.2
```

Drop a specific index combination from the MultiIndex DataFrame, i.e., drop the combination 'falcon' and 'weight', which deletes only the corresponding row

```
>>> df.drop(index=('falcon', 'weight'))
                big     small
lama    speed   45.0    30.0
        weight  200.0   100.0
        length  1.5     1.0
cow     speed   30.0    20.0
        weight  250.0   150.0
        length  1.5     0.8
falcon  speed   320.0   250.0
        length  0.3     0.2
```

```
>>> df.drop(index='cow', columns='small')
                big
lama    speed   45.0
        weight  200.0
        length  1.5
falcon  speed   320.0
```

(continues on next page)

(continued from previous page)

```
weight  1.0
length  0.3
```

rapidspy.DataFrame.drop_duplicates

DataFrame.drop_duplicates()

Return DataFrame with duplicate rows removed.

Considering certain columns is optional. Indexes, including time indexes are ignored.

Returns

DataFrame DataFrame with duplicates removed.

Examples

Consider dataset containing ramen rating.

```
>>> df = pd.DataFrame({
...     'brand': ['Yum Yum', 'Yum Yum', 'Indomie', 'Indomie', 'Indomie'],
...     'style': ['cup', 'cup', 'cup', 'pack', 'pack'],
...     'rating': [4, 4, 3.5, 15, 5]
... })
>>> df
    brand style  rating
0  Yum Yum   cup     4.0
1  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

By default, it removes duplicate rows based on all columns.

```
>>> df.drop_duplicates()
    brand style  rating
0  Yum Yum   cup     4.0
2  Indomie   cup     3.5
3  Indomie  pack    15.0
4  Indomie  pack     5.0
```

rapidspy.DataFrame.filter

DataFrame.filter(items=None, like=None, axis=None)

Subset the dataframe rows or columns according to the specified index labels.

Note that this routine does not filter a dataframe on its contents. The filter is applied to the labels of the index.

Parameters

items [list-like] Keep labels from axis which are in items.

like [str] Keep labels from axis for which "like in label == True".

axis [{0 or 'index', 1 or 'columns', None}, default None] The axis to filter on, expressed either as an index (int) or axis name (str). By default this is the info axis, 'index' for Series, 'columns' for DataFrame.

Returns

same type as input object

Notes

The items and like parameters are enforced to be mutually exclusive.

axis defaults to the info axis that is used when indexing with [].

Examples

```
>>> df = pd.DataFrame(np.array(([1, 2, 3], [4, 5, 6])),
...                   index=['mouse', 'rabbit'],
...                   columns=['one', 'two', 'three'])
>>> df
        one  two  three
mouse     1    2      3
rabbit    4    5      6
```

```
>>> # select columns by name
>>> df.filter(items=['one', 'three'])
         one  three
mouse     1      3
rabbit    4      6
```

```
>>> # select rows containing 'bbi'
>>> df.filter(like='bbi', axis=0)
         one  two  three
rabbit    4    5      6
```

rapidspy.DataFrame.rename

DataFrame.rename(columns)

Alter axes labels.

Function / dict values must be unique (1-to-1). Labels not contained in a dict / Series will be left as-is. Extra labels listed don't throw an error.

Parameters

columns [dict-like] Alternative to specifying axis (mapper, axis=1 is equivalent to columns=mapper).

Returns

DataFrame DataFrame with the renamed axis labels.

Raises

KeyError If any of the labels is not found in the selected axis and "errors='raise'".

Examples

DataFrame.rename supports two calling conventions

- (index=index_mapper, columns=columns_mapper, ...)
- (mapper, axis={'index', 'columns'}, ...)

We highly recommend using keyword arguments to clarify your intent.

Rename columns using a mapping:

```
>>> df = pd.DataFrame({"A": [1, 2, 3], "B": [4, 5, 6]})
>>> df.rename(columns={"A": "a", "B": "c"})
   a  c
0  1  4
1  2  5
2  3  6
```

Rename index using a mapping:

```
>>> df.rename(index={0: "x", 1: "y", 2: "z"})
   A  B
x  1  4
y  2  5
z  3  6
```

Cast index labels to a different type:

```
>>> df.index
RangeIndex(start=0, stop=3, step=1)
>>> df.rename(index=str).index
Index(['0', '1', '2'], dtype='object')
```

```
>>> df.rename(columns={"A": "a", "B": "b", "C": "c"}, errors="raise")
Traceback (most recent call last):
KeyError: ['C'] not found in axis
```

Using axis-style parameters:

```
>>> df.rename(str.lower, axis='columns')
   a  b
0  1  4
1  2  5
2  3  6
```

```
>>> df.rename({1: 2, 2: 4}, axis='index')
   A  B
0  1  4
2  2  5
4  3  6
```

rapidspy.DataFrame.reset_index

DataFrame.reset_index()

Reset the index, or a level of it.

Reset the index of the DataFrame, and use the default one instead. If the DataFrame has a MultiIndex, this method can remove one or more levels.

使用 MOXE 引擎时暂不支持。

Returns

DataFrame DataFrame with the new index.

Examples

```
>>> df = pd.DataFrame([('bird', 389.0),
...                    ('bird', 24.0),
...                    ('mammal', 80.5),
...                    ('mammal', np.nan)],
...                   index=['falcon', 'parrot', 'lion', 'monkey'],
...                   columns=('class', 'max_speed'))
>>> df
         class  max_speed
falcon    bird      389.0
parrot    bird       24.0
lion    mammal       80.5
monkey  mammal        NaN
```

When we reset the index, the old index is added as a column, and a new sequential index is used:

```
>>> df.reset_index()
    index   class  max_speed
0  falcon    bird      389.0
1  parrot    bird       24.0
2    lion  mammal       80.5
3  monkey  mammal        NaN
```

3.5.9 Missing data handling

DataFrame.dropna()	Remove missing values.
DataFrame.isna()	Detect missing values.
DataFrame.isnull()	DataFrame.isnull is an alias for DataFrame.isna.
DataFrame.notna()	Detect existing (non-missing) values.
DataFrame.notnull()	DataFrame.notnull is an alias for DataFrame.notna.

rapidspy.DataFrame.dropna

DataFrame.dropna()

Remove missing values.

Returns

DataFrame DataFrame with NA entries dropped from it.

Examples

```
>>> df = pd.DataFrame({"name": ['Alfred', 'Batman', 'Catwoman'],
...                    "toy": [np.nan, 'Batmobile', 'Bullwhip'],
...                    "born": [pd.NaT, pd.Timestamp("1940-04-25"),
...                             pd.NaT]})
>>> df
       name        toy       born
0    Alfred        NaN        NaT
1    Batman  Batmobile 1940-04-25
2  Catwoman   Bullwhip        NaT
```

Drop the rows where at least one element is missing.

```
>>> df.dropna()
     name        toy       born
1  Batman  Batmobile 1940-04-25
```

rapidspy.DataFrame.isna

DataFrame.isna()

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy.NaN, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True).

Returns

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is an NA value.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

rapidspy.DataFrame.isnull

DataFrame.isnull()

DataFrame.isnull is an alias for DataFrame.isna.

Detect missing values.

Return a boolean same-sized object indicating if the values are NA. NA values, such as None or numpy.NaN, gets mapped to True values. Everything else gets mapped to False values. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True).

Returns

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is an NA value.

Examples

Show which entries in a DataFrame are NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.isna()
     age   born   name    toy
0  False   True  False   True
1  False  False  False  False
2   True  False  False  False
```

Show which entries in a Series are NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
```

(continues on next page)

(continued from previous page)

```
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.isna()
0    False
1    False
2     True
dtype: bool
```

rapidspy.DataFrame.notna

DataFrame.notna()

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True). NA values, such as None or numpy.NaN, get mapped to False values.

Returns

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
...                        born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                              pd.Timestamp('1940-04-25')],
...                        name=['Alfred', 'Batman', ''],
...                        toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

rapidspy.DataFrame.notnull

DataFrame.notnull()

DataFrame.notnull is an alias for DataFrame.notna.

Detect existing (non-missing) values.

Return a boolean same-sized object indicating if the values are not NA. Non-missing values get mapped to True. Characters such as empty strings '' or numpy.inf are not considered NA values (unless you set pandas.options.mode.use_inf_as_na = True). NA values, such as None or numpy.NaN, get mapped to False values.

Returns

DataFrame Mask of bool values for each element in DataFrame that indicates whether an element is not an NA value.

Examples

Show which entries in a DataFrame are not NA.

```
>>> df = pd.DataFrame(dict(age=[5, 6, np.NaN],
...                    born=[pd.NaT, pd.Timestamp('1939-05-27'),
...                          pd.Timestamp('1940-04-25')],
...                    name=['Alfred', 'Batman', ''],
...                    toy=[None, 'Batmobile', 'Joker']))
>>> df
   age       born    name        toy
0  5.0        NaT  Alfred       None
1  6.0 1939-05-27  Batman  Batmobile
2  NaN 1940-04-25              Joker
```

```
>>> df.notna()
     age   born  name    toy
0   True  False  True  False
1   True   True  True   True
2  False   True  True   True
```

Show which entries in a Series are not NA.

```
>>> ser = pd.Series([5, 6, np.NaN])
>>> ser
0    5.0
1    6.0
2    NaN
dtype: float64
```

```
>>> ser.notna()
0     True
1     True
2    False
dtype: bool
```

3.5.10 Reshaping, sorting, transposing

DataFrame.sort_values(by[, ascending])	Sort by the values along either axis.
DataFrame.sort_index([level, ascending])	Sort object by labels (along an axis).
DataFrame.nlargest(n, columns)	Return the first n rows ordered by columns in descending order.
DataFrame.nsmallest(n, columns)	Return the first n rows ordered by columns in ascending order.

rapidspy.DataFrame.sort_values

DataFrame.sort_values(by, ascending=True)

Sort by the values along either axis.

Parameters

by [str or list of str] Name or list of names to sort by.

- if axis is 0 or 'index' then by may contain index levels and/or column labels.
- if axis is 1 or 'columns' then by may contain column levels and/or index labels.

ascending [bool or list of bool, default True] Sort ascending vs. descending. Specify list for multiple sort orders. If this is a list of bools, must match the length of the by.

Returns

DataFrame DataFrame with sorted values.

Examples

```
>>> df = pd.DataFrame({
...     'col1': ['A', 'A', 'B', np.nan, 'D', 'C'],
...     'col2': [2, 1, 9, 8, 7, 4],
...     'col3': [0, 1, 9, 4, 2, 3],
...     'col4': ['a', 'B', 'c', 'D', 'e', 'F']
... })
>>> df
  col1  col2  col3 col4
0    A     2     0    a
1    A     1     1    B
2    B     9     9    c
```

(continues on next page)

(continued from previous page)

```
3  NaN    8    4    D
4    D    7    2    e
5    C    4    3    F
```

Sort by col1

```
>>> df.sort_values(by=['col1'])
  col1  col2  col3 col4
0    A    2    0    a
1    A    1    1    B
2    B    9    9    c
5    C    4    3    F
4    D    7    2    e
3  NaN    8    4    D
```

Sort by multiple columns

```
>>> df.sort_values(by=['col1', 'col2'])
  col1  col2  col3 col4
1    A    1    1    B
0    A    2    0    a
2    B    9    9    c
5    C    4    3    F
4    D    7    2    e
3  NaN    8    4    D
```

Sort Descending

```
>>> df.sort_values(by='col1', ascending=False)
  col1  col2  col3 col4
4    D    7    2    e
5    C    4    3    F
2    B    9    9    c
0    A    2    0    a
1    A    1    1    B
3  NaN    8    4    D
```

rapidspy.DataFrame.sort_index

DataFrame.sort_index(level=None, ascending=True)

Sort object by labels (along an axis).

Returns a new DataFrame sorted by label if inplace argument is False, otherwise updates the original DataFrame and returns None.

Parameters

level [int or level name or list of ints or list of level names] If not None, sort on values in specified index level(s).

ascending [bool or list of bools, default True] Sort ascending vs. descending. When the index is a MultiIndex the sort direction can be controlled for each level individually.

Returns

DataFrame The original DataFrame sorted by the labels.

Examples

```
>>> df = pd.DataFrame([1, 2, 3, 4, 5], index=[100, 29, 234, 1, 150],
...                   columns=['A'])
>>> df.sort_index()
     A
1    4
29   2
100  1
150  5
234  3
```

By default, it sorts in ascending order, to sort in descending order, use ascending=False

```
>>> df.sort_index(ascending=False)
     A
234  3
150  5
100  1
29   2
1    4
```

rapidspy.DataFrame.nlargest

DataFrame.nlargest(n, columns)

Return the first n rows ordered by columns in descending order.

Return the first n rows with the largest values in columns, in descending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to df.sort_values(columns, ascending=False).head(n), but more performant.

Parameters

n [int] Number of rows to return.

columns [label or list of labels] Column label(s) to order by.

Returns

DataFrame The first n rows ordered by the given columns in descending order.

Notes

This function cannot be used with all column types. For example, when specifying columns with object or category dtypes, TypeError is raised.

Examples

```
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
...                                   434000, 434000, 337000, 11300,
...                                   11300, 11300],
...                    'GDP': [1937894, 2583560 , 12011, 4520, 12128,
...                            17036, 182, 38, 311],
...                    'alpha-2': ["IT", "FR", "MT", "MV", "BN",
...                                "IS", "NR", "TV", "AI"]},
...                   index=["Italy", "France", "Malta",
...                          "Maldives", "Brunei", "Iceland",
...                          "Nauru", "Tuvalu", "Anguilla"])
>>> df
          population      GDP alpha-2
Italy       59000000  1937894      IT
France      65000000  2583560      FR
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
```

(continues on next page)

(continued from previous page)

```
Iceland       337000    17036      IS
Nauru          11300      182      NR
Tuvalu         11300       38      TV
Anguilla       11300      311      AI
```

In the following example, we will use nlargest to select the three rows having the largest values in column "population".

```
>>> df.nlargest(3, 'population')
          population      GDP alpha-2
France      65000000  2583560      FR
Italy       59000000  1937894      IT
Malta         434000    12011      MT
```

To order by the largest values in column "population" and then "GDP", we can specify multiple columns like in the next example.

```
>>> df.nlargest(3, ['population', 'GDP'])
          population      GDP alpha-2
France      65000000  2583560      FR
Italy       59000000  1937894      IT
Brunei        434000    12128      BN
```

rapidspy.DataFrame.nsmallest

DataFrame.nsmallest(n, columns)

Return the first n rows ordered by columns in ascending order.

Return the first n rows with the smallest values in columns, in ascending order. The columns that are not specified are returned as well, but not used for ordering.

This method is equivalent to df.sort_values(columns, ascending=True).head(n), but more performant.

Parameters

- **n** [int] Number of items to retrieve.
- **columns** [list or str] Column name or names to order by.

Returns

DataFrame

Examples

```
>>> df = pd.DataFrame({'population': [59000000, 65000000, 434000,
...                                   434000, 434000, 337000, 337000,
...                                   11300, 11300],
...                    'GDP': [1937894, 2583560 , 12011, 4520, 12128,
...                            17036, 182, 38, 311],
...                    'alpha-2': ["IT", "FR", "MT", "MV", "BN",
...                                "IS", "NR", "TV", "AI"]},
...                   index=["Italy", "France", "Malta",
...                          "Maldives", "Brunei", "Iceland",
...                          "Nauru", "Tuvalu", "Anguilla"])
>>> df
          population      GDP alpha-2
Italy       59000000  1937894      IT
France      65000000  2583560      FR
Malta         434000    12011      MT
Maldives      434000     4520      MV
Brunei        434000    12128      BN
Iceland       337000    17036      IS
Nauru         337000      182      NR
Tuvalu         11300       38      TV
Anguilla       11300      311      AI
```

In the following example, we will use nsmallest to select the three rows having the smallest values in column "population".

```
>>> df.nsmallest(3, 'population')
          population    GDP alpha-2
Tuvalu         11300     38      TV
Anguilla       11300    311      AI
Iceland       337000  17036      IS
```

To order by the smallest values in column "population" and then "GDP", we can specify multiple columns like in the next example.

```
>>> df.nsmallest(3, ['population', 'GDP'])
          population  GDP alpha-2
Tuvalu         11300   38      TV
Anguilla       11300  311      AI
Nauru         337000  182      NR
```

3.5.11 Combining / comparing / joining / merging

DataFrame.assign(**kwargs)	Assign new columns to a DataFrame.
DataFrame.merge(right[, how, left_on, ...])	Merge DataFrame or named Series objects with a database-style join.

rapidspy.DataFrame.assign

DataFrame.assign(**kwargs)

Assign new columns to a DataFrame.

Returns a new object with all original columns in addition to new ones. Existing columns that are re-assigned will be overwritten.

Parameters

**kwargs [dict of {str: callable() or Series}] The column names are keywords. If the values are callable, they are computed on the DataFrame and assigned to the new columns. The callable must not change input DataFrame (though pandas doesn't check it). If the values are not callable, (e.g. a Series, scalar, or array), they are simply assigned.

Returns

DataFrame A new DataFrame with the new columns in addition to all the existing columns.

Notes

Assigning multiple columns within the same assign is possible. Later items in '**kwargs' may refer to newly created or modified columns in 'df'; items are computed and assigned into 'df' in order.

Examples

```
>>> df = pd.DataFrame({'temp_c': [17.0, 25.0]},
...                   index=['Portland', 'Berkeley'])
>>> df
          temp_c
Portland    17.0
Berkeley    25.0
```

Where the value is a callable, evaluated on df:

```
>>> df.assign(temp_f=lambda x: x.temp_c * 9 / 5 + 32)
          temp_c  temp_f
Portland    17.0    62.6
Berkeley    25.0    77.0
```

Alternatively, the same behavior can be achieved by directly referencing an existing Series or sequence:

```
>>> df.assign(temp_f=df['temp_c'] * 9 / 5 + 32)
          temp_c  temp_f
Portland    17.0    62.6
Berkeley    25.0    77.0
```

You can create multiple columns within the same assign where one of the columns depends on another one defined within the same assign:

```
>>> df.assign(temp_f=lambda x: x['temp_c'] * 9 / 5 + 32,
...           temp_k=lambda x: (x['temp_f'] +  459.67) * 5 / 9)
          temp_c  temp_f  temp_k
Portland    17.0    62.6  290.15
Berkeley    25.0    77.0  298.15
```

rapidspy.DataFrame.merge

DataFrame.merge(right, how='inner', left_on=None, right_on=None, left_index=False, right_index=False) → rapidspy.frame.DataFrame

Merge DataFrame or named Series objects with a database-style join.

A named Series object is treated as a DataFrame with a single named column.

The join is done on columns or indexes. If joining columns on columns, the DataFrame indexes will be ignored. Otherwise if joining indexes on indexes or indexes on a column or columns, the index will be passed on. When performing a cross merge, no column specifications to merge on are allowed.

> Warning: If both key columns contain rows where the key is a null value, those rows will be matched against each other. This is different from usual SQL join behaviour and can lead to unexpected results.

Parameters

right [DataFrame or named Series] Object to merge with.

how [{ 'left' , 'right' , 'inner' }, default 'inner'] Type of merge to be performed.

- left: use only keys from left frame, similar to a SQL left outer join; preserve key order.
- right: use only keys from right frame, similar to a SQL right outer join; preserve key order.
- inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.

left_on [label] Column or index level names to join on in the left DataFrame. Can also be an array or list of arrays of the length of the left DataFrame. These arrays are treated as if they are columns.

right_on [label] Column or index level names to join on in the right DataFrame. Can also be an array or list of arrays of the length of the right DataFrame. These arrays are treated as if they are columns.

left_index [bool, default False] Use the index from the left DataFrame as the join key(s). If it is a MultiIndex, the number of keys in the other DataFrame (either the index or a number of columns) must match the number of levels.

right_index [bool, default False] Use the index from the right DataFrame as the join key. Same caveats as left_index.

Returns

DataFrame A DataFrame of the two merged objects.

Examples

```
>>> df1 = pd.DataFrame({'lkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [1, 2, 3, 5]})
>>> df2 = pd.DataFrame({'rkey': ['foo', 'bar', 'baz', 'foo'],
...                     'value': [5, 6, 7, 8]})
>>> df1
  lkey value
0  foo     1
1  bar     2
2  baz     3
3  foo     5
>>> df2
  rkey value
0  foo     5
1  bar     6
```

(continues on next page)

(continued from previous page)

```
2   baz     7
3   foo     8
```

Merge df1 and df2 on the lkey and rkey columns. The value columns have the default suffixes, _x and _y, appended.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey')
lkey  value_x rkey  value_y
0  foo        1  foo        5
1  foo        1  foo        8
2  foo        5  foo        5
3  foo        5  foo        8
4  bar        2  bar        6
5  baz        3  baz        7
```

Merge DataFrames df1 and df2 with specified left and right suffixes appended to any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey',
...           suffixes=('_left', '_right'))
lkey  value_left rkey  value_right
0  foo           1  foo           5
1  foo           1  foo           8
2  foo           5  foo           5
3  foo           5  foo           8
4  bar           2  bar           6
5  baz           3  baz           7
```

Merge DataFrames df1 and df2, but raise an exception if the DataFrames have any overlapping columns.

```
>>> df1.merge(df2, left_on='lkey', right_on='rkey', suffixes=(False, False))
Traceback (most recent call last):
...
ValueError: columns overlap but no suffix specified:
    Index(['value'], dtype='object')
```

```
>>> df1 = pd.DataFrame({'a': ['foo', 'bar'], 'b': [1, 2]})
>>> df2 = pd.DataFrame({'a': ['foo', 'baz'], 'c': [3, 4]})
>>> df1
    a  b
```

(continues on next page)

(continued from previous page)

```
0   foo  1
1   bar  2
>>> df2
    a  c
0   foo  3
1   baz  4
```

```
>>> df1.merge(df2, how='inner', on='a')
    a  b  c
0   foo  1  3
```

```
>>> df1.merge(df2, how='left', on='a')
    a  b  c
0   foo  1  3.0
1   bar  2  NaN
```

```
>>> df1 = pd.DataFrame({'left': ['foo', 'bar']})
>>> df2 = pd.DataFrame({'right': [7, 8]})
>>> df1
    left
0   foo
1   bar
>>> df2
    right
0   7
1   8
```

```
>>> df1.merge(df2, how='cross')
left  right
0   foo      7
1   foo      8
2   bar      7
3   bar      8
```

3.6 GroupBy

GroupBy objects are returned by groupby calls: rapidspy.DataFrame.groupby(), rapidspy.Series.groupby(), etc.

3.6.1 Computations / descriptive stats

GroupByBase.count()	Compute count of group, excluding missing values.
GroupByBase.max()	Compute max of group values.
GroupByBase.mean()	Compute mean of groups, excluding missing values.
GroupByBase.min()	Compute min of group values.
GroupByBase.sum()	Compute sum of group values.

rapidspy.groupby.GroupByBase.count

GroupByBase.count()

Compute count of group, excluding missing values.

Returns

Series or DataFrame Count of values within each group.

rapidspy.groupby.GroupByBase.max

GroupByBase.max()

Compute max of group values.

Returns

Series or DataFrame Computed max of values within each group.

rapidspy.groupby.GroupByBase.mean

GroupByBase.mean()

Compute mean of groups, excluding missing values.

Returns

pandas.Series or pandas.DataFrame

Examples

```
>>> df = pd.DataFrame({'A': [1, 1, 2, 1, 2],
...                    'B': [np.nan, 2, 3, 4, 5],
...                    'C': [1, 2, 1, 1, 2]}, columns=['A', 'B', 'C'])
```

Groupby one column and return the mean of the remaining columns in each group.

```
>>> df.groupby('A').mean()
     B         C
A
1  3.0  1.333333
2  4.0  1.500000
```

Groupby two columns and return the mean of the remaining column.

```
>>> df.groupby(['A', 'B']).mean()
       C
A B
1 2.0  2
  4.0  1
2 3.0  1
  5.0  2
```

Groupby one column and return the mean of only particular column in the group.

```
>>> df.groupby('A')['B'].mean()
A
1    3.0
2    4.0
Name: B, dtype: float64
```

rapidspy.groupby.GroupByBase.min

GroupByBase.min()

Compute min of group values.

Returns

Series or DataFrame Computed min of values within each group.

rapidspy.groupby.GroupByBase.sum

GroupByBase.sum()

Compute sum of group values.

Returns

Series or DataFrame Computed sum of values within each group.

The following methods are available only for SeriesGroupBy objects.

SeriesGroupBy.strjoin()	Join the str values of groups by ','.

rapidspy.groupby.SeriesGroupBy.strjoin

SeriesGroupBy.strjoin()

Join the str values of groups by ','.

Returns

Series Join the str values within each group by ','.

CHAPTER

FOUR

发行说明

4.1 Version 1.0

4.1.1 1. 简介

RapidsPY 是在 RapidsDB 之上实现 pandas DataFrame API 的一个 Python 库, 您可以在 RapidsDB 上使用 pandas 对大数据进行处理和分析。RapidsPY 可以让您处理任意大的数据集并显著提高计算速度。

- 无学习曲线
  - 和 pandas API 的相似性, 使熟悉 pandas 的数据科学家轻松从 pandas 过渡到 RapidsDB, 而无需学习新的框架。
- 可扩展
  - 更简单地将现有的 pandas 代码移植到 RapidsDB 集群, 使得 pandas 可以从单机扩展到大数据。
- 快速
  - 利用 RapidsDB 处理大数据的性能: 在更短的时间内处理更多的数据; 迭代更快。

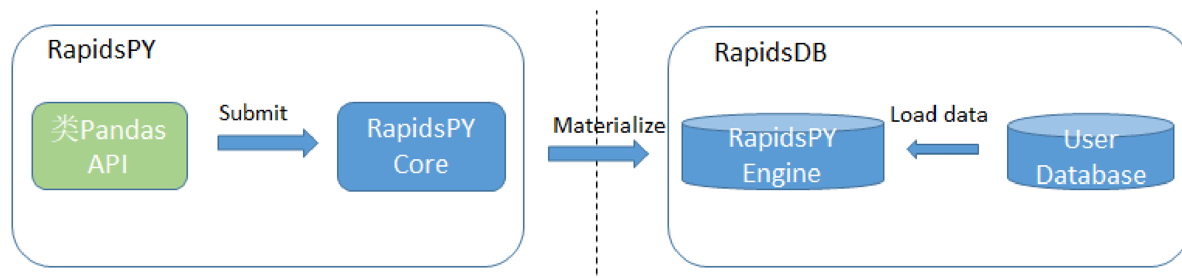
注：pandas 是一个快速、强大、灵活且易于使用的开源的数据分析和数据操作工具，构建在 Python 编程语言之上。

4.1.2 2. 主要功能

- 覆盖 API 目前覆盖了 pandas 以下 API:
  - Input/Output: 读取数据
  - General Functions: 数据操作、缺失数据检测、处理类似日期时间的数据等
  - Series: 获取数据的属性、数据转换、索引和迭代、二元运算符函数、聚合和窗口、计算/描述性统计数据、缺失数据处理、重塑/排序、时间序列等。

  - DataFrame: 获取数据的属性、数据转换、索引和迭代、二元运算符函数、聚合和窗口、计算/描述性统计数据、缺失数据处理、重塑/排序、合并/比较/连接。
  - GroupBy: 对数据进行分组 (groupby), 然后获取其统计变量
- 会话 Session
  - Session 负责所有与数据库相关的操作。
- 延迟计算
  - 存储用户应用程序里的操作 (数据处理和分析的操作), 当用户调用 Compute 方法时, 才会将 SQL 语句发往 RapidsPY Engine 去计算, 并生成实体表。这样可以提高性能, 同时可以避免产生太多数据表而导致内存溢出。

4.1.3 3. 技术架构



- 类 pandas API
  - 给用户提供的类 pandas API。提供 3 类: DataFrame, Series 和 Scalar。
  - 用户调用这些 API 编写数据处理和分析的应用程序。由于这些 API 和 pandas API 的相似性, 使用它们就像在使用 pandas。
- RapidsPY Core
  - 将用户调用的 API 转换成对应的 SQL, 发往 RapidsPY Engine 进行计算。
- RapidsPY Engine
  - RapidsPY Engine 在 RapidsDB 中, 是 RapidsPY 的数据存储和计算引擎。
- User Database
  - 用户的数据表存放在用户数据库中。开始数据处理和分析前, 需要将用户数据读入到 RapidsPY Engine 中。

4.1.4 4. 系统要求

- 操作系统

  安装本产品所支持的操作系统是：

  - Windows
  - Linux
  - MacOS

- 系统要求

  - 机器数量：单机
  - CPU：4 核
  - 内存：32GB

  注：关于系统要求，RapidsPY 本身没有要求，主要是作为 RapidsPY Engine 的 RapidsDB 的要求。

4.1.5 5. 相关文档

以下是 RapidsPY 相关文档:

- <RapidsPY_v1.0 发行说明 >
- <RapidsPY 安装及使用文档 >
- <RapidsPY_v1.0FAQ>
- <RapidsPY 支持的 API 列表 >

4.1.6 6. 介质

- Windows
  - RapidsPY-1.0.0-cp37-cp37m-win_amd64.whl
  - RapidsPY-1.0.0-cp38-cp38-win_amd64.whl
  - RapidsPY-1.0.0-cp39-cp39-win_amd64.whl
- Mac
  - RapidsPY-1.0.0-cp37-cp37m-macosx_10_9_x86_64.whl
  - RapidsPY-1.0.0-cp38-cp38-macosx_10_9_x86_64.whl
  - RapidsPY-1.0.0-cp39-cp39-macosx_10_15_x86_64.whl

- Linux
  - RapidsPY-1.0.0-cp37-cp37m-linux_x86_64.whl
  - RapidsPY-1.0.0-cp38-cp38-linux_x86_64.whl
  - RapidsPY-1.0.0-cp39-cp39-linux_x86_64.whl

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- search

INDEX

A

abs() (rapidspy.DataFrame method), 166
abs() (rapidspy.Series method), 90
add() (rapidspy.DataFrame method), 145
add() (rapidspy.Series method), 70
agg() (rapidspy.DataFrame method), 161
agg() (rapidspy.Series method), 86
aggregate() (rapidspy.DataFrame method), 162
aggregate() (rapidspy.Series method), 87
all() (rapidspy.DataFrame method), 168
all() (rapidspy.Series method), 91
any() (rapidspy.DataFrame method), 169
any() (rapidspy.Series method), 92
assign() (rapidspy.DataFrame method), 201
astype() (rapidspy.DataFrame method), 127
astype() (rapidspy.Series method), 54
at (rapidspy.DataFrame property), 132
at (rapidspy.Series property), 59
axes (rapidspy.DataFrame property), 124
axes (rapidspy.Series property), 50

B

between() (rapidspy.Series method), 93

C

close() (rapidspy.RapidsPYSession method), 34
columns (rapidspy.DataFrame property), 121
compute() (rapidspy.DataFrame method), 121
compute() (rapidspy.Series method), 48
con_list() (rapidspy.RapidsPYSession method), 33
configure() (rapidspy.RapidsPYSession method), 33
copy() (rapidspy.DataFrame method), 129
copy() (rapidspy.Series method), 56
count() (rapidspy.DataFrame method), 170
count() (rapidspy.groupby.GroupByBase method), 206
count() (rapidspy.Series method), 94

D

describe() (rapidspy.DataFrame method), 171
describe() (rapidspy.Series method), 95
div() (rapidspy.DataFrame method), 150
div() (rapidspy.Series method), 73
drop() (rapidspy.DataFrame method), 181
drop() (rapidspy.Series method), 107
drop_duplicates() (rapidspy.DataFrame method), 184
drop_duplicates() (rapidspy.Series method), 108
dropna() (rapidspy.DataFrame method), 189
dropna() (rapidspy.Series method), 112
dtype (rapidspy.Series property), 51
dtypes (rapidspy.DataFrame property), 122
dtypes (rapidspy.Series property), 53

E

empty (rapidspy.DataFrame property), 126
empty (rapidspy.Series property), 52
eq() (rapidspy.DataFrame method), 159
eq() (rapidspy.Series method), 85

F

filter() (rapidspy.DataFrame method), 185
filter() (rapidspy.Series method), 111
floordiv() (rapidspy.DataFrame method), 153
floordiv() (rapidspy.Series method), 75
from_pandas() (rapidspy.RapidsPYSession method), 36

G

ge() (rapidspy.DataFrame method), 158
ge() (rapidspy.Series method), 83
get() (rapidspy.DataFrame method), 142
get() (rapidspy.Series method), 58
groupby() (rapidspy.DataFrame method), 164
groupby() (rapidspy.Series method), 88
gt() (rapidspy.DataFrame method), 156
gt() (rapidspy.Series method), 81

H

head() (rapidspy.DataFrame method), 131
head() (rapidspy.Series method), 109

I

iat (rapidspy.DataFrame property), 133
iat (rapidspy.Series property), 60
iloc (rapidspy.DataFrame property), 139
iloc (rapidspy.Series property), 66
index (rapidspy.DataFrame property), 121
index (rapidspy.Series property), 50
is_unique (rapidspy.Series property), 106
isna() (in module rapidspy), 38
isna() (rapidspy.DataFrame method), 189
isna() (rapidspy.Series method), 113
isnull() (in module rapidspy), 40
isnull() (rapidspy.DataFrame method), 191
isnull() (rapidspy.Series method), 114

K

keys() (rapidspy.DataFrame method), 142
keys() (rapidspy.Series method), 69

L

le() (rapidspy.DataFrame method), 158
le() (rapidspy.Series method), 82
loc (rapidspy.DataFrame property), 134
loc (rapidspy.Series property), 61
lt() (rapidspy.DataFrame method), 156
lt() (rapidspy.Series method), 80

M

max() (rapidspy.DataFrame method), 173
max() (rapidspy.groupby.GroupByBase method), 206
max() (rapidspy.Series method), 97
mean() (rapidspy.DataFrame method), 174
mean() (rapidspy.groupby.GroupByBase method), 206
mean() (rapidspy.Series method), 98
median() (rapidspy.DataFrame method), 174
median() (rapidspy.Series method), 98
merge() (rapidspy.DataFrame method), 202
min() (rapidspy.DataFrame method), 175
min() (rapidspy.groupby.GroupByBase method), 207
min() (rapidspy.Series method), 98
mul() (rapidspy.DataFrame method), 148
mul() (rapidspy.Series method), 72

N

name (rapidspy.Series property), 53
ndim (rapidspy.DataFrame property), 124
ndim (rapidspy.Series property), 51
ne() (rapidspy.DataFrame method), 159
ne() (rapidspy.Series method), 84
nlargest() (rapidspy.DataFrame method), 198
nlargest() (rapidspy.Series method), 99
notna() (in module rapidspy), 42
notna() (rapidspy.DataFrame method), 192
notna() (rapidspy.Series method), 116
notnull() (in module rapidspy), 43
notnull() (rapidspy.DataFrame method), 193
notnull() (rapidspy.Series method), 117
nsmallest() (rapidspy.DataFrame method), 199
nsmallest() (rapidspy.Series method), 100
nunique() (rapidspy.DataFrame method), 180
nunique() (rapidspy.Series method), 105

P

pow() (rapidspy.DataFrame method), 153
pow() (rapidspy.Series method), 76

R

radd() (rapidspy.DataFrame method), 154
radd() (rapidspy.Series method), 77
read_sql() (rapidspy.RapidsPYSession method), 36

read_sql_query() (rapidspy.RapidsPYSession method), 35
read_sql_table() (rapidspy.RapidsPYSession method), 34
rename() (rapidspy.DataFrame method), 186
reset_index() (rapidspy.DataFrame method), 187
reset_index() (rapidspy.Series method), 110
round() (rapidspy.DataFrame method), 175
round() (rapidspy.Series method), 79
rsub() (rapidspy.DataFrame method), 155
rsub() (rapidspy.Series method), 78

S

shape (rapidspy.DataFrame property), 125
shape (rapidspy.Series property), 51
size (rapidspy.DataFrame property), 125
size (rapidspy.Series property), 52
sort_index() (rapidspy.DataFrame method), 197
sort_values() (rapidspy.DataFrame method), 195
std() (rapidspy.DataFrame method), 178
std() (rapidspy.Series method), 102
strjoin() (rapidspy.groupby.SeriesGroupBy method), 208
sub() (rapidspy.DataFrame method), 147
sub() (rapidspy.Series method), 71
sum() (rapidspy.DataFrame method), 177
sum() (rapidspy.groupby.GroupByBase method), 208
sum() (rapidspy.Series method), 103

T

to_datetime() (in module rapidspy), 45
to_frame() (rapidspy.Series method), 120
to_pandas() (rapidspy.DataFrame method), 121
to_pandas() (rapidspy.Series method), 48
truediv() (rapidspy.DataFrame method), 151
truediv() (rapidspy.Series method), 74

U

unique() (rapidspy.Series method), 104

V

value_counts() (rapidspy.DataFrame method), 180
value_counts() (rapidspy.Series method), 106
values (rapidspy.DataFrame property), 123
values (rapidspy.Series property), 50
var() (rapidspy.DataFrame method), 179
var() (rapidspy.Series method), 103