(continued from previous page)

```
   ....:              D=lambda x: x['A'] + x['C'])
   ....:
Out[80]:
   A  B  C   D
0  1  4  5   6
1  2  5  7   9
2  3  6  9  12
```

In the second expression, `x['C']` will refer to the newly created column, that's equal to `dfa['A'] + dfa['B']`.

Indexing / selection

The basics of indexing are as follows:

Operation	Syntax	Result
Select column	`df[col]`	Series
Select row by label	`df.loc[label]`	Series
Select row by integer location	`df.iloc[loc]`	Series
Slice rows	`df[5:10]`	DataFrame
Select rows by boolean vector	`df[bool_vec]`	DataFrame

Row selection, for example, returns a Series whose index is the columns of the DataFrame:

```
In [81]: df.loc['b']
Out[81]:
one              2
bar              2
flag         False
foo            bar
one_trunc        2
Name: b, dtype: object

In [82]: df.iloc[2]
Out[82]:
one             3
bar             3
flag         True
foo           bar
one_trunc     NaN
Name: c, dtype: object
```

For a more exhaustive treatment of sophisticated label-based indexing and slicing, see the *section on indexing*. We will address the fundamentals of reindexing / conforming to new sets of labels in the *section on reindexing*.

Data alignment and arithmetic

Data alignment between DataFrame objects automatically align on **both the columns and the index (row labels)**. Again, the resulting object will have the union of the column and row labels.

```
In [83]: df = pd.DataFrame(np.random.randn(10, 4), columns=['A', 'B', 'C', 'D'])

In [84]: df2 = pd.DataFrame(np.random.randn(7, 3), columns=['A', 'B', 'C'])

In [85]: df + df2
Out[85]:
          A         B         C   D
0  0.045691 -0.014138  1.380871 NaN
1 -0.955398 -1.501007  0.037181 NaN
2 -0.662690  1.534833 -0.859691 NaN
3 -2.452949  1.237274 -0.133712 NaN
4  1.414490  1.951676 -2.320422 NaN
5 -0.494922 -1.649727 -1.084601 NaN
6 -1.047551 -0.748572 -0.805479 NaN
7       NaN       NaN       NaN NaN
8       NaN       NaN       NaN NaN
9       NaN       NaN       NaN NaN
```

When doing an operation between DataFrame and Series, the default behavior is to align the Series **index** on the DataFrame **columns**, thus broadcasting row-wise. For example:

```
In [86]: df - df.iloc[0]
Out[86]:
          A         B         C         D
0  0.000000  0.000000  0.000000  0.000000
1 -1.359261 -0.248717 -0.453372 -1.754659
2  0.253128  0.829678  0.010026 -1.991234
3 -1.311128  0.054325 -1.724913 -1.620544
4  0.573025  1.500742 -0.676070  1.367331
5 -1.741248  0.781993 -1.241620 -2.053136
6 -1.240774 -0.869551 -0.153282  0.000430
7 -0.743894  0.411013 -0.929563 -0.282386
8 -1.194921  1.320690  0.238224 -1.482644
9  2.293786  1.856228  0.773289 -1.446531
```

In the special case of working with time series data, if the DataFrame index contains dates, the broadcasting will be column-wise:

```
In [87]: index = pd.date_range('1/1/2000', periods=8)

In [88]: df = pd.DataFrame(np.random.randn(8, 3), index=index, columns=list('ABC'))

In [89]: df
Out[89]:
                   A         B         C
2000-01-01 -1.226825  0.769804 -1.281247
2000-01-02 -0.727707 -0.121306 -0.097883
2000-01-03  0.695775  0.341734  0.959726
2000-01-04 -1.110336 -0.619976  0.149748
2000-01-05 -0.732339  0.687738  0.176444
2000-01-06  0.403310 -0.154951  0.301624
2000-01-07 -2.179861 -1.369849 -0.954208
```

(continues on next page)

(continued from previous page)

```
2000-01-08  1.462696 -1.743161 -0.826591

In [90]: type(df['A'])
Out[90]: pandas.core.series.Series

In [91]: df - df['A']
Out[91]:
            2000-01-01 00:00:00  2000-01-02 00:00:00  2000-01-03 00:00:00  2000-01-04␣
↪00:00:00  ...  2000-01-08 00:00:00   A   B   C
2000-01-01                  NaN                  NaN                  NaN            ␣
↪      NaN  ...                  NaN NaN NaN NaN
2000-01-02                  NaN                  NaN                  NaN            ␣
↪      NaN  ...                  NaN NaN NaN NaN
2000-01-03                  NaN                  NaN                  NaN            ␣
↪      NaN  ...                  NaN NaN NaN NaN
2000-01-04                  NaN                  NaN                  NaN            ␣
↪      NaN  ...                  NaN NaN NaN NaN
2000-01-05                  NaN                  NaN                  NaN            ␣
↪      NaN  ...                  NaN NaN NaN NaN
2000-01-06                  NaN                  NaN                  NaN            ␣
↪      NaN  ...                  NaN NaN NaN NaN
2000-01-07                  NaN                  NaN                  NaN            ␣
↪      NaN  ...                  NaN NaN NaN NaN
2000-01-08                  NaN                  NaN                  NaN            ␣
↪      NaN  ...                  NaN NaN NaN NaN

[8 rows x 11 columns]
```

Warning:

```
df - df['A']
```

is now deprecated and will be removed in a future release. The preferred way to replicate this behavior is

```
df.sub(df['A'], axis=0)
```

For explicit control over the matching and broadcasting behavior, see the section on *flexible binary operations*.

Operations with scalars are just as you would expect:

```
In [92]: df * 5 + 2
Out[92]:
                   A         B         C
2000-01-01 -4.134126  5.849018 -4.406237
2000-01-02 -1.638535  1.393469  1.510587
2000-01-03  5.478873  3.708672  6.798628
2000-01-04 -3.551681 -1.099880  2.748742
2000-01-05 -1.661697  5.438692  2.882222
2000-01-06  4.016548  1.225246  3.508122
2000-01-07 -8.899303 -4.849247 -2.771039
2000-01-08  9.313480 -6.715805 -2.132955

In [93]: 1 / df
Out[93]:
                   A         B          C
```

(continues on next page)

(continued from previous page)

```
2000-01-01 -0.815112  1.299033  -0.780489
2000-01-02 -1.374179 -8.243600 -10.216313
2000-01-03  1.437247  2.926250   1.041965
2000-01-04 -0.900628 -1.612966   6.677871
2000-01-05 -1.365487  1.454041   5.667510
2000-01-06  2.479485 -6.453662   3.315381
2000-01-07 -0.458745 -0.730007  -1.047990
2000-01-08  0.683669 -0.573671  -1.209788

In [94]: df ** 4
Out[94]: 
                    A         B         C
2000-01-01   2.265327  0.351172  2.694833
2000-01-02   0.280431  0.000217  0.000092
2000-01-03   0.234355  0.013638  0.848376
2000-01-04   1.519910  0.147740  0.000503
2000-01-05   0.287640  0.223714  0.000969
2000-01-06   0.026458  0.000576  0.008277
2000-01-07  22.579530  3.521204  0.829033
2000-01-08   4.577374  9.233151  0.466834
```

Boolean operators work as well:

```
In [95]: df1 = pd.DataFrame({'a': [1, 0, 1], 'b': [0, 1, 1]}, dtype=bool)

In [96]: df2 = pd.DataFrame({'a': [0, 1, 1], 'b': [1, 1, 0]}, dtype=bool)

In [97]: df1 & df2
Out[97]: 
       a      b
0  False  False
1  False   True
2   True  False

In [98]: df1 | df2
Out[98]: 
      a     b
0  True  True
1  True  True
2  True  True

In [99]: df1 ^ df2
Out[99]: 
       a      b
0   True   True
1   True  False
2  False   True

In [100]: -df1
Out[100]: 
       a      b
0  False   True
1   True  False
2  False  False
```

Transposing

To transpose, access the `T` attribute (also the `transpose` function), similar to an ndarray:

```
# only show the first 5 rows
In [101]: df[:5].T
Out[101]:
   2000-01-01  2000-01-02  2000-01-03  2000-01-04  2000-01-05
A   -1.226825   -0.727707    0.695775   -1.110336   -0.732339
B    0.769804   -0.121306    0.341734   -0.619976    0.687738
C   -1.281247   -0.097883    0.959726    0.149748    0.176444
```

DataFrame interoperability with NumPy functions

Elementwise NumPy ufuncs (log, exp, sqrt, . . .) and various other NumPy functions can be used with no issues on Series and DataFrame, assuming the data within are numeric:

```
In [102]: np.exp(df)
Out[102]:
                   A         B         C
2000-01-01  0.293222  2.159342  0.277691
2000-01-02  0.483015  0.885763  0.906755
2000-01-03  2.005262  1.407386  2.610980
2000-01-04  0.329448  0.537957  1.161542
2000-01-05  0.480783  1.989212  1.192968
2000-01-06  1.496770  0.856457  1.352053
2000-01-07  0.113057  0.254145  0.385117
2000-01-08  4.317584  0.174966  0.437538

In [103]: np.asarray(df)
Out[103]:
array([[-1.2268,  0.7698, -1.2812],
       [-0.7277, -0.1213, -0.0979],
       [ 0.6958,  0.3417,  0.9597],
       [-1.1103, -0.62  ,  0.1497],
       [-0.7323,  0.6877,  0.1764],
       [ 0.4033, -0.155 ,  0.3016],
       [-2.1799, -1.3698, -0.9542],
       [ 1.4627, -1.7432, -0.8266]])
```

DataFrame is not intended to be a drop-in replacement for ndarray as its indexing semantics and data model are quite different in places from an n-dimensional array.

`Series` implements `__array_ufunc__`, which allows it to work with NumPy's universal functions.

The ufunc is applied to the underlying array in a Series.

```
In [104]: ser = pd.Series([1, 2, 3, 4])

In [105]: np.exp(ser)
Out[105]:
0     2.718282
1     7.389056
2    20.085537
3    54.598150
dtype: float64
```

Changed in version 0.25.0: When multiple `Series` are passed to a ufunc, they are aligned before performing the operation.

Like other parts of the library, pandas will automatically align labeled inputs as part of a ufunc with multiple inputs. For example, using `numpy.remainder()` on two *Series* with differently ordered labels will align before the operation.

```
In [106]: ser1 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

In [107]: ser2 = pd.Series([1, 3, 5], index=['b', 'a', 'c'])

In [108]: ser1
Out[108]:
a    1
b    2
c    3
dtype: int64

In [109]: ser2
Out[109]:
b    1
a    3
c    5
dtype: int64

In [110]: np.remainder(ser1, ser2)
Out[110]:
a    1
b    0
c    3
dtype: int64
```

As usual, the union of the two indices is taken, and non-overlapping values are filled with missing values.

```
In [111]: ser3 = pd.Series([2, 4, 6], index=['b', 'c', 'd'])

In [112]: ser3
Out[112]:
b    2
c    4
d    6
dtype: int64

In [113]: np.remainder(ser1, ser3)
Out[113]:
a    NaN
b    0.0
c    3.0
d    NaN
dtype: float64
```

When a binary ufunc is applied to a *Series* and *Index*, the Series implementation takes precedence and a Series is returned.

```
In [114]: ser = pd.Series([1, 2, 3])

In [115]: idx = pd.Index([4, 5, 6])
```

(continues on next page)

(continued from previous page)

```
In [116]: np.maximum(ser, idx)
Out[116]:
0    4
1    5
2    6
dtype: int64
```

NumPy ufuncs are safe to apply to `Series` backed by non-ndarray arrays, for example `arrays.SparseArray` (see *Sparse calculation*). If possible, the ufunc is applied without converting the underlying data to an ndarray.

Console display

Very large DataFrames will be truncated to display them in the console. You can also get a summary using `info()`. (Here I am reading a CSV version of the **baseball** dataset from the **plyr** R package):

```
In [117]: baseball = pd.read_csv('data/baseball.csv')

In [118]: print(baseball)
       id     player  year  stint team  lg   g   ab   r    h  X2b  X3b  hr   rbi   sb␣
↪  cs  bb    so  ibb  hbp   sh   sf  gidp
0   88641  womacto01  2006      2  CHN  NL  19   50   6   14    1    0   1   2.0  1.0␣
↪ 1.0   4   4.0  0.0  0.0  3.0  0.0   0.0
1   88643  schilcu01  2006      1  BOS  AL  31    2   0    1    0    0   0   0.0  0.0␣
↪ 0.0   0   1.0  0.0  0.0  0.0  0.0   0.0
..    ...        ...   ...    ...  ...  ..  ..  ...  ..  ...  ...  ...  ..   ...  ...␣
↪ ...  ..   ...  ...  ...  ...  ...   ...
98  89533   aloumo01  2007      1  NYN  NL  87  328  51  112   19    1  13  49.0  3.0␣
↪ 0.0  27  30.0  5.0  2.0  0.0  3.0  13.0
99  89534  alomasa02  2007      1  NYN  NL   8   22   1    3    1    0   0   0.0  0.0␣
↪ 0.0   0   3.0  0.0  0.0  0.0  0.0   0.0

[100 rows x 23 columns]

In [119]: baseball.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100 entries, 0 to 99
Data columns (total 23 columns):
 #   Column  Non-Null Count  Dtype  
---  ------  --------------  -----  
 0   id      100 non-null    int64  
 1   player  100 non-null    object 
 2   year    100 non-null    int64  
 3   stint   100 non-null    int64  
 4   team    100 non-null    object 
 5   lg      100 non-null    object 
 6   g       100 non-null    int64  
 7   ab      100 non-null    int64  
 8   r       100 non-null    int64  
 9   h       100 non-null    int64  
 10  X2b     100 non-null    int64  
 11  X3b     100 non-null    int64  
 12  hr      100 non-null    int64  
 13  rbi     100 non-null    float64
 14  sb      100 non-null    float64
 15  cs      100 non-null    float64
```

(continues on next page)

(continued from previous page)

```
 16  bb      100 non-null    int64
 17  so      100 non-null    float64
 18  ibb     100 non-null    float64
 19  hbp     100 non-null    float64
 20  sh      100 non-null    float64
 21  sf      100 non-null    float64
 22  gidp    100 non-null    float64
dtypes: float64(9), int64(11), object(3)
memory usage: 18.1+ KB
```

However, using `to_string` will return a string representation of the DataFrame in tabular form, though it won't always fit the console width:

```
In [120]: print(baseball.iloc[-20:, :12].to_string())
       id     player  year  stint team  lg    g   ab   r    h  X2b  X3b
80  89474  finlest01  2007      1  COL  NL   43   94   9   17    3    0
81  89480  embreal01  2007      1  OAK  AL    4    0   0    0    0    0
82  89481  edmonji01  2007      1  SLN  NL  117  365  39   92   15    2
83  89482  easleda01  2007      1  NYN  NL   76  193  24   54    6    0
84  89489  delgaca01  2007      1  NYN  NL  139  538  71  139   30    0
85  89493  cormirh01  2007      1  CIN  NL    6    0   0    0    0    0
86  89494  coninje01  2007      2  NYN  NL   21   41   2    8    2    0
87  89495  coninje01  2007      1  CIN  NL   80  215  23   57   11    1
88  89497  clemero02  2007      1  NYA  AL    2    2   0    1    0    0
89  89498  claytro01  2007      2  BOS  AL    8    6   1    0    0    0
90  89499  claytro01  2007      1  TOR  AL   69  189  23   48   14    0
91  89501  cirilje01  2007      2  ARI  NL   28   40   6    8    4    0
92  89502  cirilje01  2007      1  MIN  AL   50  153  18   40    9    2
93  89521  bondsba01  2007      1  SFN  NL  126  340  75   94   14    0
94  89523  biggicr01  2007      1  HOU  NL  141  517  68  130   31    3
95  89525  benitar01  2007      2  FLO  NL   34    0   0    0    0    0
96  89526  benitar01  2007      1  SFN  NL   19    0   0    0    0    0
97  89530  ausmubr01  2007      1  HOU  NL  117  349  38   82   16    3
98  89533   aloumo01  2007      1  NYN  NL   87  328  51  112   19    1
99  89534  alomasa02  2007      1  NYN  NL    8   22   1    3    1    0
```

Wide DataFrames will be printed across multiple rows by default:

```
In [121]: pd.DataFrame(np.random.randn(3, 12))
Out[121]:
          0         1         2         3         4         5         6         7  ␣
↪    8         9        10        11
0 -0.345352  1.314232  0.690579  0.995761  2.396780  0.014871  3.357427 -0.317441 -1.
↪236269  0.896171 -0.487602 -0.082240
1 -2.182937  0.380396  0.084844  0.432390  1.519970 -0.493662  0.600178  0.274230  0.
↪132885 -0.023688  2.410179  1.450520
2  0.206053 -0.251905 -2.213588  1.063327  1.266143  0.299368 -0.863838  0.408204 -1.
↪048089 -0.025747 -0.988387  0.094055
```

You can change how much to print on a single row by setting the `display.width` option:

```
In [122]: pd.set_option('display.width', 40)  # default is 80

In [123]: pd.DataFrame(np.random.randn(3, 12))
Out[123]:
          0         1         2         3         4         5         6         7  ␣
↪    8         9        10        11
```

(continues on next page)

(continued from previous page)

```
0  1.262731  1.289997  0.082423 -0.055758  0.536580 -0.489682  0.369374 -0.034571 -2.
↪484478 -0.281461  0.030711  0.109121
1  1.126203 -0.977349  1.474071 -0.064034 -1.282782  0.781836 -1.071357  0.441153  2.
↪353925  0.583787  0.221471 -0.744471
2  0.758527  1.729689 -0.964980 -0.845696 -1.340896  1.846883 -1.328865  1.682706 -1.
↪717693  0.888782  0.228440  0.901805
```

You can adjust the max width of the individual columns by setting `display.max_colwidth`

```
In [124]: datafile = {'filename': ['filename_01', 'filename_02'],
   .....:             'path': ["media/user_name/storage/folder_01/filename_01",
   .....:                      "media/user_name/storage/folder_02/filename_02"]}
   .....:

In [125]: pd.set_option('display.max_colwidth', 30)

In [126]: pd.DataFrame(datafile)
Out[126]:
      filename                           path
0  filename_01  media/user_name/storage/fo...
1  filename_02  media/user_name/storage/fo...

In [127]: pd.set_option('display.max_colwidth', 100)

In [128]: pd.DataFrame(datafile)
Out[128]:
      filename                                           path
0  filename_01  media/user_name/storage/folder_01/filename_01
1  filename_02  media/user_name/storage/folder_02/filename_02
```

You can also disable this feature via the `expand_frame_repr` option. This will print the table in one block.

DataFrame column attribute access and IPython completion

If a DataFrame column label is a valid Python variable name, the column can be accessed like an attribute:

```
In [129]: df = pd.DataFrame({'foo1': np.random.randn(5),
   .....:                    'foo2': np.random.randn(5)})
   .....:

In [130]: df
Out[130]:
       foo1      foo2
0  1.171216 -0.858447
1  0.520260  0.306996
2 -1.197071 -0.028665
3 -1.066969  0.384316
4 -0.303421  1.574159

In [131]: df.foo1
Out[131]:
0    1.171216
1    0.520260
2   -1.197071
3   -1.066969
```

(continues on next page)

(continued from previous page)

```
4   -0.303421
Name: foo1, dtype: float64
```

The columns are also connected to the IPython completion mechanism so they can be tab-completed:

```
In [5]: df.fo<TAB>  # noqa: E225, E999
df.foo1  df.foo2
```

1.4.7 Comparison with other tools

Comparison with R / R libraries

Since `pandas` aims to provide a lot of the data manipulation and analysis functionality that people use R for, this page was started to provide a more detailed look at the R language and its many third party libraries as they relate to `pandas`. In comparisons with R and CRAN libraries, we care about the following things:

- **Functionality / flexibility**: what can/cannot be done with each tool
- **Performance**: how fast are operations. Hard numbers/benchmarks are preferable
- **Ease-of-use**: Is one tool easier/harder to use (you may have to be the judge of this, given side-by-side code comparisons)

This page is also here to offer a bit of a translation guide for users of these R packages.

For transfer of `DataFrame` objects from `pandas` to R, one option is to use HDF5 files, see *External compatibility* for an example.

Quick reference

We'll start off with a quick reference guide pairing some common R operations using dplyr with pandas equivalents.

Querying, filtering, sampling

R	pandas
`dim(df)`	`df.shape`
`head(df)`	`df.head()`
`slice(df, 1:10)`	`df.iloc[:9]`
`filter(df, col1 == 1, col2 == 1)`	`df.query('col1 == 1 & col2 == 1')`
`df[df$col1 == 1 & df$col2 == 1,]`	`df[(df.col1 == 1) & (df.col2 == 1)]`
`select(df, col1, col2)`	`df[['col1', 'col2']]`
`select(df, col1:col3)`	`df.loc[:, 'col1':'col3']`
`select(df, -(col1:col3))`	`df.drop(cols_to_drop, axis=1)` but see[1]
`distinct(select(df, col1))`	`df[['col1']].drop_duplicates()`
`distinct(select(df, col1, col2))`	`df[['col1', 'col2']].drop_duplicates()`
`sample_n(df, 10)`	`df.sample(n=10)`
`sample_frac(df, 0.01)`	`df.sample(frac=0.01)`

[1] R's shorthand for a subrange of columns (`select(df, col1:col3)`) can be approached cleanly in pandas, if you have the list of columns, for example `df[cols[1:3]]` or `df.drop(cols[1:3])`, but doing this by column name is a bit messy.

Sorting

R	pandas
`arrange(df, col1, col2)`	`df.sort_values(['col1', 'col2'])`
`arrange(df, desc(col1))`	`df.sort_values('col1', ascending=False)`

Transforming

R	pandas
`select(df, col_one = col1)`	`df.rename(columns={'col1': 'col_one'})['col_one']`
`rename(df, col_one = col1)`	`df.rename(columns={'col1': 'col_one'})`
`mutate(df, c=a-b)`	`df.assign(c=df['a']-df['b'])`

Grouping and summarizing

R	pandas
`summary(df)`	`df.describe()`
`gdf <- group_by(df, col1)`	`gdf = df.groupby('col1')`
`summarise(gdf, avg=mean(col1, na.rm=TRUE))`	`df.groupby('col1').agg({'col1': 'mean'})`
`summarise(gdf, total=sum(col1))`	`df.groupby('col1').sum()`

Base R

Slicing with R's `c`

R makes it easy to access `data.frame` columns by name

```
df <- data.frame(a=rnorm(5), b=rnorm(5), c=rnorm(5), d=rnorm(5), e=rnorm(5))
df[, c("a", "c", "e")]
```

or by integer location

```
df <- data.frame(matrix(rnorm(1000), ncol=100))
df[, c(1:10, 25:30, 40, 50:100)]
```

Selecting multiple columns by name in `pandas` is straightforward

```
In [1]: df = pd.DataFrame(np.random.randn(10, 3), columns=list('abc'))

In [2]: df[['a', 'c']]
Out[2]:
          a         c
0  0.469112 -1.509059
1 -1.135632 -0.173215
2  0.119209 -0.861849
```

(continues on next page)

(continued from previous page)

```
3 -2.104569  1.071804
4  0.721555 -1.039575
5  0.271860  0.567020
6  0.276232 -0.673690
7  0.113648  0.524988
8  0.404705 -1.715002
9 -1.039268 -1.157892

In [3]: df.loc[:, ['a', 'c']]
Out[3]: 
          a         c
0  0.469112 -1.509059
1 -1.135632 -0.173215
2  0.119209 -0.861849
3 -2.104569  1.071804
4  0.721555 -1.039575
5  0.271860  0.567020
6  0.276232 -0.673690
7  0.113648  0.524988
8  0.404705 -1.715002
9 -1.039268 -1.157892
```

Selecting multiple noncontiguous columns by integer location can be achieved with a combination of the `iloc` indexer attribute and `numpy.r_`.

```
In [4]: named = list('abcdefg')

In [5]: n = 30

In [6]: columns = named + np.arange(len(named), n).tolist()

In [7]: df = pd.DataFrame(np.random.randn(n, n), columns=columns)

In [8]: df.iloc[:, np.r_[:10, 24:30]]
Out[8]: 
           a         b         c         d         e         f         g  ...        ␣
↪9        24        25        26        27        28        29
0  -1.344312  0.844885  1.075770 -0.109050  1.643563 -1.469388  0.357021  ... -0.
↪968914 -1.170299 -0.226169  0.410835  0.813850  0.132003 -0.827317
1  -0.076467 -1.187678  1.130127 -1.436737 -1.413681  1.607920  1.024180  ... -2.
↪211372  0.959726 -1.110336 -0.619976  0.149748 -0.732339  0.687738
2   0.176444  0.403310 -0.154951  0.301624 -2.179861 -1.369849 -0.954208  ... -0.
↪826591  0.084844  0.432390  1.519970 -0.493662  0.600178  0.274230
3   0.132885 -0.023688  2.410179  1.450520  0.206053 -0.251905 -2.213588  ...  0.
↪299368 -2.484478 -0.281461  0.030711  0.109121  1.126203 -0.977349
4   1.474071 -0.064034 -1.282782  0.781836 -1.071357  0.441153  2.353925  ... -0.
↪744471 -1.197071 -1.066969 -0.303421 -0.858447  0.306996 -0.028665
..       ...       ...       ...       ...       ...       ...       ...  ...      ..
↪.       ...       ...       ...       ...       ...       ...
25  1.492125 -0.068190  0.681456  1.221829 -0.434352  1.204815 -0.195612  ... -0.
↪796211  1.944517  0.042344 -0.307904  0.428572  0.880609  0.487645
26  0.725238  0.624607 -0.141185 -0.143948 -0.328162  2.095086 -0.608888  ... -2.
↪513465 -0.846188  1.190624  0.778507  1.008500  1.424017  0.717110
27  1.262419  1.950057  0.301038 -0.933858  0.814946  0.181439 -0.110015  ...  0.
↪307941 -1.341814  0.334281 -0.162227  1.007824  2.826008  1.458383
28 -1.585746 -0.899734  0.921494 -0.211762 -0.059182  0.058308  0.915377  ... -3.
↪060395  0.403620 -0.026602 -0.240481  0.577223 -1.088417  0.326687
```

(continues on next page)

(continued from previous page)

```
29 -0.986248  0.169729 -1.158091  1.019673  0.646039  0.917399 -0.010435  ...  0.
↪869610 -1.209247 -0.671466  0.332872 -2.013086 -1.602549  0.333109

[30 rows x 16 columns]
```

aggregate

In R you may want to split data into subsets and compute the mean for each. Using a data.frame called `df` and splitting it into groups `by1` and `by2`:

```
df <- data.frame(
  v1 = c(1,3,5,7,8,3,5,NA,4,5,7,9),
  v2 = c(11,33,55,77,88,33,55,NA,44,55,77,99),
  by1 = c("red", "blue", 1, 2, NA, "big", 1, 2, "red", 1, NA, 12),
  by2 = c("wet", "dry", 99, 95, NA, "damp", 95, 99, "red", 99, NA, NA))
aggregate(x=df[, c("v1", "v2")], by=list(mydf2$by1, mydf2$by2), FUN = mean)
```

The *groupby()* method is similar to base R `aggregate` function.

```
In [9]: df = pd.DataFrame(
   ...:     {'v1': [1, 3, 5, 7, 8, 3, 5, np.nan, 4, 5, 7, 9],
   ...:      'v2': [11, 33, 55, 77, 88, 33, 55, np.nan, 44, 55, 77, 99],
   ...:      'by1': ["red", "blue", 1, 2, np.nan, "big", 1, 2, "red", 1, np.nan, 12],
   ...:      'by2': ["wet", "dry", 99, 95, np.nan, "damp", 95, 99, "red", 99, np.nan,
   ...:              np.nan]})
   ...:

In [10]: g = df.groupby(['by1', 'by2'])

In [11]: g[['v1', 'v2']].mean()
Out[11]:
            v1    v2
by1  by2
1    95    5.0  55.0
     99    5.0  55.0
2    95    7.0  77.0
     99    NaN   NaN
big  damp  3.0  33.0
blue dry   3.0  33.0
red  red   4.0  44.0
     wet   1.0  11.0
```

For more details and examples see *the groupby documentation*.

match / %in%

A common way to select data in R is using `%in%` which is defined using the function `match`. The operator `%in%` is used to return a logical vector indicating if there is a match or not:

```
s <- 0:4
s %in% c(2,4)
```

The `isin()` method is similar to R `%in%` operator:

```
In [12]: s = pd.Series(np.arange(5), dtype=np.float32)

In [13]: s.isin([2, 4])
Out[13]:
0    False
1    False
2     True
3    False
4     True
dtype: bool
```

The `match` function returns a vector of the positions of matches of its first argument in its second:

```
s <- 0:4
match(s, c(2,4))
```

For more details and examples see *the reshaping documentation*.

tapply

`tapply` is similar to `aggregate`, but data can be in a ragged array, since the subclass sizes are possibly irregular. Using a data.frame called `baseball`, and retrieving information based on the array `team`:

```
baseball <-
  data.frame(team = gl(5, 5,
             labels = paste("Team", LETTERS[1:5])),
             player = sample(letters, 25),
             batting.average = runif(25, .200, .400))

tapply(baseball$batting.average, baseball.example$team,
       max)
```

In `pandas` we may use `pivot_table()` method to handle this:

```
In [14]: import random

In [15]: import string

In [16]: baseball = pd.DataFrame(
   ....:     {'team': ["team %d" % (x + 1) for x in range(5)] * 5,
   ....:      'player': random.sample(list(string.ascii_lowercase), 25),
   ....:      'batting avg': np.random.uniform(.200, .400, 25)})
   ....:

In [17]: baseball.pivot_table(values='batting avg', columns='team', aggfunc=np.max)
Out[17]:
```

(continues on next page)

(continued from previous page)

```
team            team 1    team 2    team 3    team 4    team 5
batting avg  0.352134  0.295327  0.397191  0.394457  0.396194
```

For more details and examples see *the reshaping documentation*.

subset

The `query()` method is similar to the base R `subset` function. In R you might want to get the rows of a `data.frame` where one column's values are less than another column's values:

```
df <- data.frame(a=rnorm(10), b=rnorm(10))
subset(df, a <= b)
df[df$a <= df$b,]  # note the comma
```

In `pandas`, there are a few ways to perform subsetting. You can use `query()` or pass an expression as if it were an index/slice as well as standard boolean indexing:

```
In [18]: df = pd.DataFrame({'a': np.random.randn(10), 'b': np.random.randn(10)})

In [19]: df.query('a <= b')
Out[19]:
          a         b
1  0.174950  0.552887
2 -0.023167  0.148084
3 -0.495291 -0.300218
4 -0.860736  0.197378
5 -1.134146  1.720780
7 -0.290098  0.083515
8  0.238636  0.946550

In [20]: df[df['a'] <= df['b']]
Out[20]:
          a         b
1  0.174950  0.552887
2 -0.023167  0.148084
3 -0.495291 -0.300218
4 -0.860736  0.197378
5 -1.134146  1.720780
7 -0.290098  0.083515
8  0.238636  0.946550

In [21]: df.loc[df['a'] <= df['b']]
Out[21]:
          a         b
1  0.174950  0.552887
2 -0.023167  0.148084
3 -0.495291 -0.300218
4 -0.860736  0.197378
5 -1.134146  1.720780
7 -0.290098  0.083515
8  0.238636  0.946550
```

For more details and examples see *the query documentation*.

with

An expression using a data.frame called `df` in R with the columns `a` and `b` would be evaluated using `with` like so:

```
df <- data.frame(a=rnorm(10), b=rnorm(10))
with(df, a + b)
df$a + df$b  # same as the previous expression
```

In `pandas` the equivalent expression, using the *eval()* method, would be:

```
In [22]: df = pd.DataFrame({'a': np.random.randn(10), 'b': np.random.randn(10)})

In [23]: df.eval('a + b')
Out[23]:
0   -0.091430
1   -2.483890
2   -0.252728
3   -0.626444
4   -0.261740
5    2.149503
6   -0.332214
7    0.799331
8   -2.377245
9    2.104677
dtype: float64

In [24]: df['a'] + df['b']  # same as the previous expression
Out[24]:
0   -0.091430
1   -2.483890
2   -0.252728
3   -0.626444
4   -0.261740
5    2.149503
6   -0.332214
7    0.799331
8   -2.377245
9    2.104677
dtype: float64
```

In certain cases *eval()* will be much faster than evaluation in pure Python. For more details and examples see *the eval documentation*.

plyr

`plyr` is an R library for the split-apply-combine strategy for data analysis. The functions revolve around three data structures in R, `a` for `arrays`, `l` for `lists`, and `d` for `data.frame`. The table below shows how these data structures could be mapped in Python.

R	Python
array	list
lists	dictionary or list of objects
data.frame	dataframe

ddply

An expression using a data.frame called `df` in R where you want to summarize `x` by `month`:

```
require(plyr)
df <- data.frame(
  x = runif(120, 1, 168),
  y = runif(120, 7, 334),
  z = runif(120, 1.7, 20.7),
  month = rep(c(5,6,7,8),30),
  week = sample(1:4, 120, TRUE)
)

ddply(df, .(month, week), summarize,
      mean = round(mean(x), 2),
      sd = round(sd(x), 2))
```

In `pandas` the equivalent expression, using the *groupby()* method, would be:

```
In [25]: df = pd.DataFrame({'x': np.random.uniform(1., 168., 120),
   ....:                    'y': np.random.uniform(7., 334., 120),
   ....:                    'z': np.random.uniform(1.7, 20.7, 120),
   ....:                    'month': [5, 6, 7, 8] * 30,
   ....:                    'week': np.random.randint(1, 4, 120)})
   ....:

In [26]: grouped = df.groupby(['month', 'week'])

In [27]: grouped['x'].agg([np.mean, np.std])
Out[27]:
                  mean        std
month week
5     1      63.653367  40.601965
      2      78.126605  53.342400
      3      92.091886  57.630110
6     1      81.747070  54.339218
      2      70.971205  54.687287
      3     100.968344  54.010081
7     1      61.576332  38.844274
      2      61.733510  48.209013
      3      71.688795  37.595638
8     1      62.741922  34.618153
      2      91.774627  49.790202
      3      73.936856  60.773900
```

For more details and examples see *the groupby documentation*.

reshape / reshape2

melt.array

An expression using a 3 dimensional array called a in R where you want to melt it into a data.frame:

```
a <- array(c(1:23, NA), c(2,3,4))
data.frame(melt(a))
```

In Python, since a is a list, you can simply use list comprehension.

```
In [28]: a = np.array(list(range(1, 24)) + [np.NAN]).reshape(2, 3, 4)

In [29]: pd.DataFrame([tuple(list(x) + [val]) for x, val in np.ndenumerate(a)])
Out[29]:
    0  1  2     3
0   0  0  0   1.0
1   0  0  1   2.0
2   0  0  2   3.0
3   0  0  3   4.0
4   0  1  0   5.0
.. .. .. ..   ...
19  1  1  3  20.0
20  1  2  0  21.0
21  1  2  1  22.0
22  1  2  2  23.0
23  1  2  3   NaN

[24 rows x 4 columns]
```

melt.list

An expression using a list called a in R where you want to melt it into a data.frame:

```
a <- as.list(c(1:4, NA))
data.frame(melt(a))
```

In Python, this list would be a list of tuples, so *DataFrame()* method would convert it to a dataframe as required.

```
In [30]: a = list(enumerate(list(range(1, 5)) + [np.NAN]))

In [31]: pd.DataFrame(a)
Out[31]:
   0    1
0  0  1.0
1  1  2.0
2  2  3.0
3  3  4.0
4  4  NaN
```

For more details and examples see *the Into to Data Structures documentation*.

melt.data.frame

An expression using a data.frame called `cheese` in R where you want to reshape the data.frame:

```
cheese <- data.frame(
  first = c('John', 'Mary'),
  last = c('Doe', 'Bo'),
  height = c(5.5, 6.0),
  weight = c(130, 150)
)
melt(cheese, id=c("first", "last"))
```

In Python, the *melt()* method is the R equivalent:

```
In [32]: cheese = pd.DataFrame({'first': ['John', 'Mary'],
   ....:                        'last': ['Doe', 'Bo'],
   ....:                        'height': [5.5, 6.0],
   ....:                        'weight': [130, 150]})
   ....:

In [33]: pd.melt(cheese, id_vars=['first', 'last'])
Out[33]:
  first last variable  value
0  John  Doe   height    5.5
1  Mary   Bo   height    6.0
2  John  Doe   weight  130.0
3  Mary   Bo   weight  150.0

In [34]: cheese.set_index(['first', 'last']).stack()  # alternative way
Out[34]:
first  last
John   Doe   height      5.5
             weight    130.0
Mary   Bo    height      6.0
             weight    150.0
dtype: float64
```

For more details and examples see *the reshaping documentation*.

cast

In R `acast` is an expression using a data.frame called `df` in R to cast into a higher dimensional array:

```
df <- data.frame(
  x = runif(12, 1, 168),
  y = runif(12, 7, 334),
  z = runif(12, 1.7, 20.7),
  month = rep(c(5,6,7),4),
  week = rep(c(1,2), 6)
)

mdf <- melt(df, id=c("month", "week"))
acast(mdf, week ~ month ~ variable, mean)
```

In Python the best way is to make use of *pivot_table()*:

```
In [35]: df = pd.DataFrame({'x': np.random.uniform(1., 168., 12),
   ....:                    'y': np.random.uniform(7., 334., 12),
   ....:                    'z': np.random.uniform(1.7, 20.7, 12),
   ....:                    'month': [5, 6, 7] * 4,
   ....:                    'week': [1, 2] * 6})
   ....:

In [36]: mdf = pd.melt(df, id_vars=['month', 'week'])

In [37]: pd.pivot_table(mdf, values='value', index=['variable', 'week'],
   ....:                columns=['month'], aggfunc=np.mean)
   ....:
Out[37]:
month                  5           6           7
variable week
x        1     93.888747   98.762034   55.219673
         2     94.391427   38.112932   83.942781
y        1     94.306912  279.454811  227.840449
         2     87.392662  193.028166  173.899260
z        1     11.016009   10.079307   16.170549
         2      8.476111   17.638509   19.003494
```

Similarly for `dcast` which uses a data.frame called `df` in R to aggregate information based on `Animal` and `FeedType`:

```
df <- data.frame(
  Animal = c('Animal1', 'Animal2', 'Animal3', 'Animal2', 'Animal1',
             'Animal2', 'Animal3'),
  FeedType = c('A', 'B', 'A', 'A', 'B', 'B', 'A'),
  Amount = c(10, 7, 4, 2, 5, 6, 2)
)

dcast(df, Animal ~ FeedType, sum, fill=NaN)
# Alternative method using base R
with(df, tapply(Amount, list(Animal, FeedType), sum))
```

Python can approach this in two different ways. Firstly, similar to above using *`pivot_table()`*:

```
In [38]: df = pd.DataFrame({
   ....:     'Animal': ['Animal1', 'Animal2', 'Animal3', 'Animal2', 'Animal1',
   ....:                'Animal2', 'Animal3'],
   ....:     'FeedType': ['A', 'B', 'A', 'A', 'B', 'B', 'A'],
   ....:     'Amount': [10, 7, 4, 2, 5, 6, 2],
   ....: })
   ....:

In [39]: df.pivot_table(values='Amount', index='Animal', columns='FeedType',
   ....:                aggfunc='sum')
   ....:
Out[39]:
FeedType     A     B
Animal
Animal1   10.0   5.0
Animal2    2.0  13.0
Animal3    6.0   NaN
```

The second approach is to use the *`groupby()`* method:

```
In [40]: df.groupby(['Animal', 'FeedType'])['Amount'].sum()
Out[40]:
Animal   FeedType
Animal1  A           10
         B            5
Animal2  A            2
         B           13
Animal3  A            6
Name: Amount, dtype: int64
```

For more details and examples see *the reshaping documentation* or *the groupby documentation*.

factor

pandas has a data type for categorical data.

```
cut(c(1,2,3,4,5,6), 3)
factor(c(1,2,3,2,2,3))
```

In pandas this is accomplished with `pd.cut` and `astype("category")`:

```
In [41]: pd.cut(pd.Series([1, 2, 3, 4, 5, 6]), 3)
Out[41]:
0    (0.995, 2.667]
1    (0.995, 2.667]
2    (2.667, 4.333]
3    (2.667, 4.333]
4      (4.333, 6.0]
5      (4.333, 6.0]
dtype: category
Categories (3, interval[float64]): [(0.995, 2.667] < (2.667, 4.333] < (4.333, 6.0]]

In [42]: pd.Series([1, 2, 3, 2, 2, 3]).astype("category")
Out[42]:
0    1
1    2
2    3
3    2
4    2
5    3
dtype: category
Categories (3, int64): [1, 2, 3]
```

For more details and examples see *categorical introduction* and the *API documentation*. There is also a documentation regarding the *differences to R's factor*.

Comparison with SQL

Since many potential pandas users have some familiarity with SQL, this page is meant to provide some examples of how various SQL operations would be performed using pandas.

If you're new to pandas, you might want to first read through *10 Minutes to pandas* to familiarize yourself with the library.

As is customary, we import pandas and NumPy as follows:

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

In [2]: import numpy as np
```

Most of the examples will utilize the `tips` dataset found within pandas tests. We'll read the data into a DataFrame called *tips* and assume we have a database table of the same name and structure.

```
In [3]: url = ('https://raw.github.com/pandas-dev'
   ...:        '/pandas/master/pandas/tests/io/data/csv/tips.csv')
   ...:

In [4]: tips = pd.read_csv(url)

In [5]: tips.head()
Out[5]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

SELECT

In SQL, selection is done using a comma-separated list of columns you'd like to select (or a `*` to select all columns):

```
SELECT total_bill, tip, smoker, time
FROM tips
LIMIT 5;
```

With pandas, column selection is done by passing a list of column names to your DataFrame:

```
In [6]: tips[['total_bill', 'tip', 'smoker', 'time']].head(5)
Out[6]:
   total_bill   tip smoker    time
0       16.99  1.01     No  Dinner
1       10.34  1.66     No  Dinner
2       21.01  3.50     No  Dinner
3       23.68  3.31     No  Dinner
4       24.59  3.61     No  Dinner
```

Calling the DataFrame without the list of column names would display all columns (akin to SQL's `*`).

In SQL, you can add a calculated column:

```
SELECT *, tip/total_bill as tip_rate
FROM tips
LIMIT 5;
```

With pandas, you can use the *DataFrame.assign()* method of a DataFrame to append a new column:

```
In [7]: tips.assign(tip_rate=tips['tip'] / tips['total_bill']).head(5)
Out[7]:
   total_bill   tip     sex smoker  day    time  size  tip_rate
0       16.99  1.01  Female     No  Sun  Dinner     2  0.059447
1       10.34  1.66    Male     No  Sun  Dinner     3  0.160542
2       21.01  3.50    Male     No  Sun  Dinner     3  0.166587
3       23.68  3.31    Male     No  Sun  Dinner     2  0.139780
4       24.59  3.61  Female     No  Sun  Dinner     4  0.146808
```

WHERE

Filtering in SQL is done via a WHERE clause.

```
SELECT *
FROM tips
WHERE time = 'Dinner'
LIMIT 5;
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using boolean indexing.

```
In [8]: tips[tips['time'] == 'Dinner'].head(5)
Out[8]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

The above statement is simply passing a `Series` of True/False objects to the DataFrame, returning all rows with True.

```
In [9]: is_dinner = tips['time'] == 'Dinner'

In [10]: is_dinner.value_counts()
Out[10]:
True     176
False     68
Name: time, dtype: int64

In [11]: tips[is_dinner].head(5)
Out[11]:
   total_bill   tip     sex smoker  day    time  size
0       16.99  1.01  Female     No  Sun  Dinner     2
1       10.34  1.66    Male     No  Sun  Dinner     3
2       21.01  3.50    Male     No  Sun  Dinner     3
3       23.68  3.31    Male     No  Sun  Dinner     2
4       24.59  3.61  Female     No  Sun  Dinner     4
```

Just like SQL's OR and AND, multiple conditions can be passed to a DataFrame using | (OR) and & (AND).

```
-- tips of more than $5.00 at Dinner meals
SELECT *
FROM tips
WHERE time = 'Dinner' AND tip > 5.00;
```

```
# tips of more than $5.00 at Dinner meals
In [12]: tips[(tips['time'] == 'Dinner') & (tips['tip'] > 5.00)]
Out[12]:
     total_bill    tip     sex smoker  day    time  size
23        39.42   7.58    Male     No  Sat  Dinner     4
44        30.40   5.60    Male     No  Sun  Dinner     4
47        32.40   6.00    Male     No  Sun  Dinner     4
52        34.81   5.20  Female     No  Sun  Dinner     4
59        48.27   6.73    Male     No  Sat  Dinner     4
116       29.93   5.07    Male     No  Sun  Dinner     4
155       29.85   5.14  Female     No  Sun  Dinner     5
170       50.81  10.00    Male    Yes  Sat  Dinner     3
172        7.25   5.15    Male    Yes  Sun  Dinner     2
181       23.33   5.65    Male    Yes  Sun  Dinner     2
183       23.17   6.50    Male    Yes  Sun  Dinner     4
211       25.89   5.16    Male    Yes  Sat  Dinner     4
212       48.33   9.00    Male     No  Sat  Dinner     4
214       28.17   6.50  Female    Yes  Sat  Dinner     3
239       29.03   5.92    Male     No  Sat  Dinner     3
```

```
-- tips by parties of at least 5 diners OR bill total was more than $45
SELECT *
FROM tips
WHERE size >= 5 OR total_bill > 45;
```

```
# tips by parties of at least 5 diners OR bill total was more than $45
In [13]: tips[(tips['size'] >= 5) | (tips['total_bill'] > 45)]
Out[13]:
     total_bill    tip     sex smoker   day    time  size
59        48.27   6.73    Male     No   Sat  Dinner     4
125       29.80   4.20  Female     No  Thur   Lunch     6
141       34.30   6.70    Male     No  Thur   Lunch     6
142       41.19   5.00    Male     No  Thur   Lunch     5
143       27.05   5.00  Female     No  Thur   Lunch     6
155       29.85   5.14  Female     No   Sun  Dinner     5
156       48.17   5.00    Male     No   Sun  Dinner     6
170       50.81  10.00    Male    Yes   Sat  Dinner     3
182       45.35   3.50    Male    Yes   Sun  Dinner     3
185       20.69   5.00    Male     No   Sun  Dinner     5
187       30.46   2.00    Male    Yes   Sun  Dinner     5
212       48.33   9.00    Male     No   Sat  Dinner     4
216       28.15   3.00    Male    Yes   Sat  Dinner     5
```

NULL checking is done using the `notna()` and `isna()` methods.

```
In [14]: frame = pd.DataFrame({'col1': ['A', 'B', np.NaN, 'C', 'D'],
   ....:                       'col2': ['F', np.NaN, 'G', 'H', 'I']})
   ....:

In [15]: frame
Out[15]:
```

(continues on next page)

(continued from previous page)

```
  col1 col2
0    A    F
1    B  NaN
2  NaN    G
3    C    H
4    D    I
```

Assume we have a table of the same structure as our DataFrame above. We can see only the records where `col2` IS NULL with the following query:

```
SELECT *
FROM frame
WHERE col2 IS NULL;
```

```
In [16]: frame[frame['col2'].isna()]
Out[16]:
  col1 col2
1    B  NaN
```

Getting items where `col1` IS NOT NULL can be done with *notna()*.

```
SELECT *
FROM frame
WHERE col1 IS NOT NULL;
```

```
In [17]: frame[frame['col1'].notna()]
Out[17]:
  col1 col2
0    A    F
1    B  NaN
3    C    H
4    D    I
```

GROUP BY

In pandas, SQL's GROUP BY operations are performed using the similarly named *groupby()* method. *groupby()* typically refers to a process where we'd like to split a dataset into groups, apply some function (typically aggregation) , and then combine the groups together.

A common SQL operation would be getting the count of records in each group throughout a dataset. For instance, a query getting us the number of tips left by sex:

```
SELECT sex, count(*)
FROM tips
GROUP BY sex;
/*
Female     87
Male      157
*/
```

The pandas equivalent would be:

```
In [18]: tips.groupby('sex').size()
Out[18]:
```

(continues on next page)