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

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
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  ...        25        26
0  -1.344312  0.844885  1.075770 -0.109050  1.643563  ... -0.226169  0.410835  0.813
1  -0.076467 -1.187678  1.130127 -1.436737 -1.413681  ... -1.110336 -0.619976  0.149
2   0.176444  0.403310 -0.154951  0.301624 -2.179861  ...  0.432390  1.519970 -0.493
3   0.132885 -0.023688  2.410179  1.450520  0.206053  ... -0.281461  0.030711  0.109
4   1.474071 -0.064034 -1.282782  0.781836 -1.071357  ... -1.066969 -0.303421 -0.858
5   0.384316  1.574159  1.588931  0.476720  0.473424  ...  0.068159 -0.057873 -0.368
6   0.800193  0.782098 -1.069094 -1.099248  0.255269  ...  2.121453  0.597701  0.563
..       ...       ...       ...       ...       ...  ...       ...       ...
23  1.534417 -1.374226 -0.367477  0.782551  1.356489  ... -1.690959  0.961088  0.052
24  0.859275 -0.995910  0.261263  1.783442  0.380989  ...  0.840316  0.638172  0.890
25  1.492125 -0.068190  0.681456  1.221829 -0.434352  ...  0.042344 -0.307904  0.428
26  0.725238  0.624607 -0.141185 -0.143948 -0.328162  ...  1.190624  0.778507  1.008
27  1.262419  1.950057  0.301038 -0.933858  0.814946  ...  0.334281 -0.162227  1.007
28 -1.585746 -0.899734  0.921494 -0.211762 -0.059182  ... -0.026602 -0.240481  0.577
29 -0.986248  0.169729 -1.158091  1.019673  0.646039  ... -0.671466  0.332872 -2.013

[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]:
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
5   0  1  1   6.0
6   0  1  2   7.0
.. .. .. ..   ...
17  1  1  1  18.0
18  1  1  2  19.0
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.