# dppd Documentation

***Release 0.25***

**Florian Finkernagel**

**Aug 31, 2023**

# Contents

Welcome to **dppd**, a pythonic dplyr 'grammar of data manipulation' library.

CHAPTER 1

# Contents

## 1.1 Quickstart

### 1.1.1 Style 1: Context managers

```
>>> import pandas as pd
>>> from dppd import dppd
>>> from plotnine.data import mtcars
>>> with dppd(mtcars) as (dp, X):  # note parentheses!
...   dp.select(['name', 'hp', 'cyl'])
...   dp.filter_by(X.hp > 100).head(1)

>>> print(X.head())
        name   hp  cyl
0  Mazda RX4  110    6
>>> print(isinstance(X, pd.DataFrame))
True
>>> type(X)
<class 'dppd.base.DPPDAwareProxy'>
>>> print(len(X))
1
>>>m2 = X.pd
>>>type(m2)
<class 'pandas.core.frame.DataFrame'>
```

Within the context manager, dp is always the latest Dppd object and X is always the latest intermediate DataFrame. Once the context manager has ended, both variables (dp and X here) point to a proxy of the final DataFrame object.

That proxy should, thanks to wrapt , behave just like DataFrames, except that they have a property '.pd' that returns the real DataFrame object.

### 1.1.2 Style 2: dp. . . .pd

```
>>>import pandas as pd
>>>from dppd import dppd
>>>from plotnine.data import mtcars
>>>dp, X = dppd()

>>> mt2 = (dp(mtcars)
  .select(['name', 'hp', 'cyl'])
  .filter_by(X.hp > 100)
  .head()
  .pd
)
>>> print(mt2.head())
                 name   hp  cyl
0           Mazda RX4  110    6
1       Mazda RX4 Wag  110    6
3      Hornet 4 Drive  110    6
4  Hornet Sportabout  175    8
5             Valiant  105    6
>>> print(type(mt2))
<class 'pandas.core.frame.DataFrame'>
```

The inline-style is more casual, but requires the final call .pd to retrieve the DataFrame object, otherwise you have a `dppd.Dppd`.

### 1.1.3 How does it work

dppd follows the old adage that there's only one problem not solvable by another layer of indirection, and achives it's pipeline-style method chaining by having a proxy object X that always points to the latest DataFrame created in the pipeline.

This allows for example the following:

```
>>> with dppd(mtcars) as (dp, X):
...   high_kwh  = dp(mtcars).mutate(kwh = X.hp * 0.74).filter_by(X.kwh > 80).iloc[:2].
↪pd
...
>>> high_kwh
                 name   mpg  cyl   disp   hp  drat     wt   qsec  vs  am  gear  carb  ␣
↪  kwh
0           Mazda RX4  21.0    6  160.0  110  3.90  2.620  16.46   0   1     4     4  ␣
↪81.4
1       Mazda RX4 Wag  21.0    6  160.0  110  3.90  2.875  17.02   0   1     4     4  ␣
↪81.4
```

**Note:** Note that at this point `(X == high_khw).all()` and `(dp == high_khw).all()`.

This approach is different to dplyr and other python implementations of the 'grammar of data manipulation' - see comparisons.

Dppd also contains a single-dispatch mechanism to avoid monkey patching. See the section on extending dppd

### 1.1.4 What's next?

To learn more please refer to the sections on Dpplyr verbs, dppd verbs and grouping.

## 1.2 Verbs

### 1.2.1 Verbs

#### Dppd verbs

#### Pandas DataFrame methods

Within a dp(), **all** `pandas.DataFrame` methods and accessors work as you'd expect them to[1].

Example:

```
>>> dp(mtcars).rank().pd.head(5)
  name   mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
0  18.0  19.5  15.0  13.5  13.0  21.5   9.0   6.0   9.5  26.0  21.5  25.5
1  19.0  19.5  15.0  13.5  13.0  21.5  12.0  10.5   9.5  26.0  21.5  25.5
2   5.0  24.5   6.0   6.0   7.0  20.0   7.0  23.0  25.5  26.0  21.5   4.0
3  13.0  21.5  15.0  18.0  13.0   8.5  16.0  26.0  25.5  10.0   8.0   4.0
4  14.0  15.0  25.5  27.5  21.0  10.5  19.0  10.5   9.5  10.0   8.0  12.5
```

You can even continue working with Series within the dp and convert them back to a DataFrame later on:

```
>>> dp(mtcars).set_index('name').sum().loc[X > 15].to_frame().pd
             0
mpg    642.900
cyl    198.000
disp  7383.100
hp    4694.000
drat   115.090
wt     102.952
qsec   571.160
gear   118.000
carb    90.000
```

#### concat

`concat` combines this DataFrame and another one.

Example:

```
>>> len(mtcars)
32
>>> len(dp(mtcars).concat(mtcars).pd)
64
```

[1] Except for the deprecated `pandas.DataFrame.select()`, which is shadowed by our verb `select`.

### unselect

`unselect` drops by `column specification`[2].

Example:

```
>>> dp(mtcars).unselect(lambda x: len(x) <= 3).pd.head(1)
        name   disp  drat   qsec  gear  carb
0  Mazda RX4  160.0   3.9  16.46     4     4
```

### distinct

*`distinct`* selects unique rows, possibly only considering a `column specification`.

Example:

```
>>> dp(mtcars).distinct('cyl').pd
                name   mpg  cyl   disp   hp  drat    wt   qsec  vs  am  gear  carb
0          Mazda RX4  21.0    6  160.0  110  3.90  2.62  16.46   0   1     4     4
2          Datsun 710  22.8    4  108.0   93  3.85  2.32  18.61   1   1     4     1
4  Hornet Sportabout  18.7    8  360.0  175  3.15  3.44  17.02   0   0     3     2
```

### transassign

*`transassign`* creates a new DataFrame based on this one.

Example:

```
>>> dp(mtcars).head(5).set_index('name').transassign(kwh = X.hp * 0.74).pd
                     kwh
name
Mazda RX4            81.40
Mazda RX4 Wag        81.40
Datsun 710           68.82
Hornet 4 Drive       81.40
Hornet Sportabout   129.50
```

### add_count

`add_count` adds the group count to each row.

This is a good example verb to get started on writting own.

Example:

```
>>> dp(mtcars).groupby('cyl').add_count().ungroup().sort_index().head(5).select(['name
↪','cyl','count']).pd
                name  cyl  count
0          Mazda RX4    6      7
1      Mazda RX4 Wag    6      7
2         Datsun 710    4     11
3     Hornet 4 Drive    6      7
4  Hornet Sportabout    8     14
```

[2] 'drop' is already a pandas method name - `pandas.DataFrame.drop()`

### as_type

`as_type` quickly converts the type of columns by a column_specification.

Example:

```
>>> dp(mtcars).astype(['-qsec', '-name'], int).pd.head()
                name  mpg  cyl  disp   hp  drat  wt   qsec  vs  am  gear  carb
0          Mazda RX4   21    6   160  110     3   2  16.46   0   1     4     4
1      Mazda RX4 Wag   21    6   160  110     3   2  17.02   0   1     4     4
2         Datsun 710   22    4   108   93     3   2  18.61   1   1     4     1
3     Hornet 4 Drive   21    6   258  110     3   3  19.44   1   0     3     1
4  Hornet Sportabout   18    8   360  175     3   3  17.02   0   0     3     2
```

### categorize

Turn columns into pandas.Categoricals. Default categories are unique values in the order they appear in the dataframe. Pass None to use sorted unique values (ie. pandas.Categorical default behaviour).

### unique_in_order

Does what it says on the tin.

### binarize

Convert categorical columns into 'regression columns', i.e. X with values a,b,c becomes three binary columns X-a, X-b, X-c which are True exactly where X was a, etc.

### rename_columns / reset_columns

Wraps df.columns = ... into an inline call. Accepts either a list, a function, a callable (called once for each column with the old columnl, or a string (for single column dataframes). Also accepts None, which resets the columns to list(X.columns) (useful to work around a categorical-columns-can't-add-any bug).

### ends

heads and tails at once.

### natsort

Sort via the natsort package.

### display

call display(X) - for inline display in jupyter notebooks.

### Dplyr verbs

All dplyr verbs stay 'in pipeline' - you can chain them together on a :class:Dppd.

### Mutate

Adds new columns.

mutate() takes keyword arguments that are turned into columns on the DataFrame.

Excluding grouping, this is a straigt forward wrapper around `pandas.DataFrame.assign()`.

Example:

```
>> dp(mtcars).mutate(lower_case_name = X.name.str.lower()).head(1).pd
        name   mpg  cyl   disp   hp  drat    wt   qsec  vs  am  gear  carb lower_case_
↪name
0  Mazda RX4  21.0    6  160.0  110   3.9  2.62  16.46   0   1     4     4
↪mazda rx4
```

### Select

Pick columns, with optional rename.

Example:

```
>>>dp(mtcars).select('name').head(1).pd
        name
0  Mazda RX4

>>> dp(mtcars).select([X.name, 'hp']).columns.pd
Index(['name', 'hp'], dtype='object')

>>> dp(mtcars).select(X.columns.str.startswith('c')).columns.pd
Index(['cyl', 'carb'], dtype='object')


>>> dp(mtcars).select(['-hp','-cyl','-am']).columns.pd
Index(['name', 'mpg', 'disp', 'drat', 'wt', 'qsec', 'vs', 'gear', 'carb'], dtype=
↪'object')

#renaming
>>> dp(mtcars).select({'model': "name"}).columns.pd
Index(['model'], dtype='object')
```

See `select` and `column_specification` for full details.

**Note:** This verb shadows `pandas.DataFrame.select()`, which is deprecated.

### filter_by

Filter a DataFrame's rows.

Examples:

```
# by a comparison / boolean vector
>>> dp(mtcars).filter_by(X.hp > 100).head(2).pd
            name   mpg  cyl   disp   hp  drat     wt   qsec  vs  am  gear  carb
0      Mazda RX4  21.0    6  160.0  110   3.9  2.620  16.46   0   1     4     4
1  Mazda RX4 Wag  21.0    6  160.0  110   3.9  2.875  17.02   0   1     4     4

# by an existing columns
>>> dp(mtcars).filter_by(X.am).head(2).pd
            name   mpg  cyl   disp   hp  drat     wt   qsec  vs  am  gear  carb
1  Mazda RX4 Wag  21.0    6  160.0  110   3.9  2.875  17.02   0   1     4     4
1  Mazda RX4 Wag  21.0    6  160.0  110   3.9  2.875  17.02   0   1     4     4

# by a callback
>>> dp(mtcars).filter_by(lambda X: np.random.rand(len(X)) < 0.5).head(2).pd
        name   mpg  cyl   disp   hp  drat    wt   qsec  vs  am  gear  carb
6  Duster 360  14.3    8  360.0  245  3.21  3.57  15.84   0   0     3     4
7   Merc 240D  24.4    4  146.7   62  3.69  3.19  20.00   1   0     4     2
```

See *filter_by* for full details.

**Note:** This function is not called filter as not to shadow `pandas.DataFrame.filter()`

### arrange

Sort a DataFrame by a `column_specification`

```
>>> dp(mtcars).arrange([X.hp, X.qsec], ascending=[False, True]).select(['name','hp',
↪'qsec']).head(5).pd
                 name   hp   qsec
30      Maserati Bora  335  14.60
28     Ford Pantera L  264  14.50
23         Camaro Z28  245  15.41
6          Duster 360  245  15.84
16  Chrysler Imperial  230  17.42
```

### summarize

Summarize the columns in a DataFrame with callbacks.

Example:

```
>>> dp(mtcars).summarize(
...      ('hp', np.min),
...      ('hp', np.max),
...      ('hp', np.mean),
...      ('hp', np.std),
...      ).pd
  hp_amin  hp_amax   hp_mean     hp_std
0       52      335  146.6875  67.483071
```

```
>>> dp(mtcars).summarize(
...    ('hp', np.min, 'min(hp)'),
```

(continues on next page)

(continued from previous page)

```
...    ('hp', np.max, 'max(hp)'),
...    ('hp', np.mean, 'mean(hp)'),
...    ('hp', np.std, 'stddev(hp)'),
...    ).pd
  min(hp)   max(hp)   mean(hp)   stddev(hp)
0       52       335  146.6875    67.483071
```

Summarize is most useful with grouped DataFrames.

#### do

Map a grouped DataFrame into a concated other DataFrame. Easier shown than explained:

```
>>> dp(mtcars).groupby('cyl').add_count().ungroup().sort_index().head(5).select(['name
↪','cyl','count']).pd
                 name  cyl  count
0           Mazda RX4    6      7
1       Mazda RX4 Wag    6      7
2          Datsun 710    4     11
3      Hornet 4 Drive    6      7
4   Hornet Sportabout    8     14
```

#### TidyR verbs

Dppd also supports tidyr verbs where they are 'easier' to use than the pandas equivalents

#### gather

`pandas.melt()` with `column specifications`

#### spread

*`spread`* spreads a key/value column pair into it's components. Inverse of *`gather`*

#### unite

*`unite`* joins the values of each row as strings

#### seperate

*`seperate`* splits strings on a seperator

### 1.2.2 Verbs on other datatypes verbs

All verbs stay 'in pipeline' - you can chain them together on a :class:Dppd.

#### to_frame

Implemented on: Dict

Turn the value into a DataFrame.

## 1.3 Groups and summaries

Dppd's grouping is based on `pandas.DataFrame.groupby()`, which is supported in the fluent api:

```
>>>dp(mtcars).groupby('cyl').mean().filter_by(X.hp>100).select(['mpg', 'disp', 'hp']).
↪pd
          mpg        disp          hp
cyl
6    19.742857  183.314286  122.285714
8    15.100000  353.100000  209.214286
```

Select, mutate and filter_by work on the underlying DataFrame:

```
>>> dp(mtcars).groupby('cyl').select('name').head(1).pd
                 name  cyl
0           Mazda RX4    6
2          Datsun 710    4
4  Hornet Sportabout    8

# Note how selecting on a DataFrameGroupBy does always preserve the grouping columns
```

During this mutate, X is the DataFrameGroupBy object, and the ranks are per group accordingly:

```
>>> dp(mtcars).groupby('cyl').mutate(hp_rank=X.hp.rank()).ungroup().select(['name',
↪'cyl', 'hp', 'hp_rank']).pd.head()
                 name  cyl   hp  hp_rank
0           Mazda RX4    6  110      3.0
1       Mazda RX4 Wag    6  110      3.0
2          Datsun 710    4   93      7.0
3      Hornet 4 Drive    6  110      3.0
4  Hornet Sportabout    8  175      3.5
```

And the same in filter_by:

```
>>> dp(mtcars).groupby('cyl').filter_by(X.hp.rank() <= 2).ungroup().select(['name',
↪'cyl', 'hp']).pd
                 name  cyl   hp
5             Valiant    6  105
7           Merc 240D    4   62
18       Honda Civic    4   52
21  Dodge Challenger    8  150
22       AMC Javelin    8  150
```

Note that both mutate and filter_by play nice with the callables, they're distributed by group - either directly, or via `pandas.DataFrameGroupBy.apply()`:

```
>>> a = dp(mtcars).groupby('cyl').mutate(str_count = lambda x: "%.2i" % len(x)).
↪ungroup().pd
>>> b = dp(mtcars).groupby('cyl').mutate(str_count = X.apply(lambda x: "%.2i" %␣
↪len(x))).ungroup().pd
```

(continues on next page)

(continued from previous page)

```
>>> (a == b).all().all()
True
>>> a.head()
  cyl               name   mpg   disp   hp  drat     wt   qsec  vs  am  gear  carb␣
↪str_count
0    6          Mazda RX4  21.0  160.0  110  3.90  2.620  16.46   0   1     4     4 ␣
↪     07
1    6      Mazda RX4 Wag  21.0  160.0  110  3.90  2.875  17.02   0   1     4     4 ␣
↪     07
2    4         Datsun 710  22.8  108.0   93  3.85  2.320  18.61   1   1     4     1 ␣
↪     11
3    6     Hornet 4 Drive  21.4  258.0  110  3.08  3.215  19.44   1   0     3     1 ␣
↪     07
4    8  Hornet Sportabout  18.7  360.0  175  3.15  3.440  17.02   0   0     3     2 ␣
↪     14
```

### 1.3.1 Summaries

First off, you can summarize groupby objects with the usual pandas methods `pandas.DataFrame.agg()`, and stay in the pipe:

```
>>> dp(mtcars).groupby('cyl').agg([np.mean, np.std]).select(['hp', 'gear']).pd
            hp                    gear
          mean        std       mean       std
cyl
4     82.636364  20.934530  4.090909  0.539360
6    122.285714  24.260491  3.857143  0.690066
8    209.214286  50.976886  3.285714  0.726273

#note the interaction of select and the MultiIndex column names.
```

In addition, we have the *summarize* verb, which any number of tuples (column_name, function) or (column_name, function, new_name) as arguments:

```
>>> (dp(mtcars).groupby('cyl').summarize(('hp', np.mean), ('hp', np.std), ('gear', np.
↪mean), ('gear', np.std)).pd)
  cyl     hp_mean     hp_std  gear_mean  gear_std
0    4   82.636364  19.960291   4.090909  0.514259
1    6  122.285714  22.460850   3.857143  0.638877
2    8  209.214286  49.122556   3.285714  0.699854
```

## 1.4 The why of dppd

Undoubtly, in R dplyr is a highly useful library since many of it's verbs are not available otherwise.

But pandas, which has been moving to support method chaining in the last few releases already does most of dplyr's verbs, so why is there half a dozen dplyr clones for python, including this one (see <comparison>)?

Part of it is likely to be historic - the clone projects started before pandas DataFrames were as chainable as they are today.

Another part is the power of R's non-standard-evaluation, which, if unpythonic, has a certain allure.

**Dppd brings three things to pandas:**

- the proxy that always points to the latest DataFrame (or object), which 'fakes' non-standard-evaluation at the full power of python
- filtering on groupby()ed DataFrames
- R like column specifications for selection and sorting.

### 1.4.1 Proxy X

X is always the latest object:

```
>>> dp(mtcars).assign(kwh=X.hp * 0.74).filter_by(X.kwh > 100).head(5).pd
                name   mpg  cyl   disp   hp  drat    wt   qsec  vs  am  gear  carb  ␣
↪kwh
4   Hornet Sportabout  18.7    8  360.0  175  3.15  3.44  17.02   0   0     3     2 ␣
↪129.5
6           Duster 360  14.3    8  360.0  245  3.21  3.57  15.84   0   0     3     4 ␣
↪181.3
11          Merc 450SE  16.4    8  275.8  180  3.07  4.07  17.40   0   0     3     3 ␣
↪133.2
12          Merc 450SL  17.3    8  275.8  180  3.07  3.73  17.60   0   0     3     3 ␣
↪133.2
13         Merc 450SLC  15.2    8  275.8  180  3.07  3.78  18.00   0   0     3     3 ␣
↪133.2
```

### 1.4.2 Filtering groupbyed DataFrames

Let's take the example from our Readme, which calculates the highest cars by kwh from the mtcars dataset (allowing for ties):

```
 >>> from plotnine.data import mtcars
 >>> from dppd import dppd
 >>> dp, X = dppd()
>>> (dp(mtcars)
...        .mutate(kwh = X.hp * 0.74)
...        .groupby('cyl')
...        .filter_by(X.kwh.rank() < 2)
...        .ungroup().pd
...        )

    cyl              name   mpg   disp   hp  drat     wt   qsec  vs  am  gear  carb  ␣
↪   kwh
 5     6           Valiant  18.1  225.0  105  2.76  3.460  20.22   1   0     3     1 ␣
↪ 77.70
 18    4       Honda Civic  30.4   75.7   52  4.93  1.615  18.52   1   1     4     2 ␣
↪ 38.48
 21    8  Dodge Challenger  15.5  318.0  150  2.76  3.520  16.87   0   0     3     2 ␣
↪111.00
 22    8       AMC Javelin  15.2  304.0  150  3.15  3.435  17.30   0   0     3     2 ␣
↪111.00
```

And the pandas equivalent:

```
>>> mtcars = mtcars.assign(kwh = mtcars['hp'] * 0.74)
>>> ranks = mtcars.groupby('cyl').kwh.rank()
```

(continues on next page)

(continued from previous page)

```
>>> mtcars[ranks < 2]
                name   mpg  cyl   disp   hp  drat     wt   qsec  vs  am  gear  carb  ␣
↪  kwh
5            Valiant  18.1    6  225.0  105  2.76  3.460  20.22   1   0     3     1  ␣
↪77.70
18       Honda Civic  30.4    4   75.7   52  4.93  1.615  18.52   1   1     4     2  ␣
↪38.48
21  Dodge Challenger  15.5    8  318.0  150  2.76  3.520  16.87   0   0     3     2 ␣
↪111.00
22       AMC Javelin  15.2    8  304.0  150  3.15  3.435  17.30   0   0     3     2 ␣
↪111.00
```

### 1.4.3 Column specifications

Selecting columns in pandas is alread powerful, using the df.columns.str.whatever methods. It is verbose though, and sort_values with it's 'ascending' parameter is way to many characters just to invert the sorting order on a column.

Dppd supports a mini language for column specifications - see *dppd.column_spec.parse_column_specification()* for details:

```
# drop column name
>>> dp(mtcars).select('-name').head(1).pd
    mpg  cyl   disp   hp  drat    wt   qsec  vs  am  gear  carb   kwh
0  21.0    6  160.0  110   3.9  2.62  16.46   0   1     4     4  81.4

# sort by hp inverted
>>> dp(mtcars).arrange('-hp').head(2).select(['name','cyl','hp']).pd
          name  cyl  hp
18  Honda Civic    4  52
7     Merc 240D    4  62
```

### 1.4.4 Single dispatch 'clean monkey patching' engine

Dppd internally is in essence a clean monkey-patching single dispatch engine that allows you to wrap types beyond the DataFrame.e

## 1.5 Comparison of dplyr and various python approaches

There have been various attempts to bring dplyr's ease of use to pandas DataFrames. This document attempts a 'Rosetta stone' style translation and some characterization about the individual libraries.

Please note that I'm not overly familar with each of these libraries, pull requests to improve the 'translations' are welcome.

### 1.5.1 Libraries compared

- dplyr
  - the R based original by the famous Hadley Wickham.
  - based on '*pipeing*' with the %>% operator (see *pipeing*)

  - could be used from python with rpy2
- pandas the referenc Python DataFrame implementation could benefit from a chainable API
- plydata
  - *pipeing*
  - evaluates strings as python code for *non-standard-evaluation*
  - code suggest it could be extended to non-DataFrame objects
- dplython
  - *pipeing*
  - custom DataFrame class
  - magic X for *non-standard-evaluation*
- dfply
  - *pipeing*
  - magic X for *non-standard-evaluation*
  - perhaps the most comprehensive python implementation (before dppd).
  - active development as of Dec. 2018
  - easy definition of new verbs
- pandas_ply
  - fluent API (method chaining) instead of *pipeing*
  - magic X for *non-standard-evaluation*
  - monkey-patches pd.DataFrame with exactly two methods: ply_select and ply_where.
- dpyr
  - no documentation
  - no examples
  - seems to introduce pipeing to ibis
- dppd
  - fluent API (method chaining) instead of *pipeing*
  - no non-standard-evaluation, based on proxy objects (a different magic X)
  - easy definition of new verbs

### 1.5.2 The magic of dpylr

Non standard evaluation is R's killer feature that allows you to write statements that are evaluated 'later' in a different context, for example in that of your DataFrame. *mutate(df, new_column = old_column * 5)* for example creates a new dataframe with an additional column, which is set to *df$old_column * 5*.

Python can approach it with its first-class-functions, but the lambda syntax remains cumbersome in comparison.

To understand how this piece of code works, you need to understand pipeing:

It transforms an expression *df >> g >> f >> h* into the equivalent *h(f(g(df)))*

This works around the limitation of R's object model, which always dispatches on functions and accordingly offers no method-chaining fluent interfaces. It combines beautifully with R's late-binding seemless currying.

```
flights_sml %>%
  group_by(year, month, day) %>%
  filter(rank(desc(arr_delay)) < 10)
```

for example groups a DataFrame of flights by their date, orders a column (descending), turns it into a rank 1..n, and selects those from the original data frame that were in the top 10 (ie. worst delays) on each day.

Dplyr is open in the sense that it's easy to extend both the set of verbs (trivialy by defining functions taking a dataframe and returning one) and the set of supported objects (for example to database tables, less trivial).

### 1.5.3 A critique on the existing python dplyr clones (or why dppd)

Most of the dplyr inspired Python libraries try very hard to reproduce the two core aspects of dplyr, *pipeing* and *non-standard-evaluation*. Pipeing is usually fairly well implemented but reads unpythonic and is usually accompanied with namespace polution. Non-standard-evaluation is harder to implement correctly, and every single implementation so far serverly limits the expressiveness that Python offers (e.g. no list comprehensions)

As an example, the following code is the dfply equivalent to the flight filtering R code above

```
from dfply import *
( flights_sml
  >> group_by(X.year, X.month, X.day)
  >> filter_by(
    make_symbolic(pd.Series.rank)(X.arr_delay, ascending=False) < 10)
)
```

The big insight of dppd is that in many cases, this is not actually non-standard-evaluation that needs to be evaluted later, but simply a 'variable not available in context' problem which can be solved with a proxy variable X that always points to the latest DataFrame created. In the other cases, a fallback to functions/lambdas is not that bad / no more ugly than having to wrap the function in a decorator.

Combined with a pythonic method-chaining fluent API, this looks like this

```
from dppd import dppd
dp, X = dppd()
(
  dp(flights_sml)
  .filter_by(X.groupby(['year', 'month', 'day']).arr_delay.rank(ascending=False) < 10)
  .pd
)
```

### 1.5.4 Examples

All datasets are provided either by ggplot2 (R) or plotnine.data (python) For R, they've been converted to tibbles.

The python libraries are imported as dp (as opposed to *from x import **), with their magic variable also being imported.

dppd uses *from dppd import dppd; dp, X = dppd()*

So let's do some Rosetta-Stoning.

### Select/Drop columns

Select/Drop columns by name

R:

```
mtcars >> select(hp) # select only hp
mtcars >> select(-hp) # drop hp
```

pandas:

```
mtcars[['hp', 'name',]] # select only hp and name
mtcars.drop(['hp', 'name'], axis=1) # drop hp
```

plydata:

```
mtcars >> dp.select('hp', 'name')
mtcars >> dp.select('hp', 'name', drop=True)
# no list passing
```

dplython:

```
dp.DplyFrame(mtcars) >> dp.select(X.hp, X.name)
```

Neither strings nor lists may be passed to select.
No dropping of columns

dfply:

```
mtcars >> dp.select('hp', 'name')
mtcars >> dp.select(['hp', 'name'])
mtcars >> dp.select(X.hp, X.name) # either works
mtcars >> dp.select([X.hp, 'name']) even mixing
mtcars >> dp.drop('hp')
mtcars >> dp.drop(X.hp)
```

dppd:

```
dp(mtcars).select(['hp', 'name']).pd # must be a list
dp(mtcars).unselect(['hp']).pd # we like symetry
dp(mtcars).drop('hp', axis=1).pd # fallback to pandas method
```

### Filter column by list of accepted values

Example cut down mtcars to two named cars.

R:

```
mtcars = as.tibble(mtcars)
mtcars = rownames_to_column(mtcars)
mtcars %>% filter(rowname %in% c("Fiat 128", "Lotus Europa"))
```

pandas:

```
mtcars[mtcars.name.isin(["Fiat 128", "Lotus Europa"]]
```

plydata:

```
mtcars >> dp.query('name.isin(["Fiat 128", "Lotus Europa"])')
```

dplython:

```
dp.DplyFrame(mtcars) >> dp.arrange(X.cyl) >> dp.sift(X.name.isin(['Lotus Europa',
↪'Fiat 128']))
```

dfply:

```
mtcars >> dp.arrange(X.cyl) >>dp.filter_by(X.name.isin(['Lotus Europa', 'Fiat 128']))
```

pandas_ply:

```
mtcars.ply_where(X.name.isin(['Lotus Europa', 'Fiat 128']))
```

dppd:

```
dp(mtcars).arrange('cyl').filter_by(X.name.isin(['Lotus Europa', 'Fiat 128'])).pd
```

### Filter by substring

R:

```
mtcars %>% filter(grepl('RX', rowname))
```

Actually a regexps-match.

**pandas::** mtcars[mtcars.name.str.contains('RX')]

plydata:

```
mtcars >> dp.query("name.str.contains('RX')")
```

dplython:

```
dp.DplyFrame(mtcars) >> dp.sift(X.name.str.contains('RX'))
```

dfply:

```
mtcars >>dp.filter_by(X.name.str.contains('RX'))
```

dppd:

```
dp(mtcars).filter_by(X.name.str.contains('RX')).pd
```

### Select rows by iloc

R:

```
mtcars %>% slice(6:10)
```

pandas:

```
mtcars.iloc[5:10]
```

plydata:

```
(mtcars >>dp.call('.iloc'))[5:10]
```

plydata's call allows easy, if verbose fallbock on the DataFrames methods

dpylython:

```
dp.DplyFrame(mtcars) >> X._.iloc[5:10]
```

dplython offers the original DataFrame's method via a '._' forward.

dfply:

```
mtcars >> dp.row_slice(np.arange(5,10))
```

row_slice does not support the full iloc[] interface, but only single ints or lists of such.

dppd:

```
dp(mtcars).iloc[5:10].pd()
```

### Select by loc / rownames

R:

```
no rownames in tibbles
```

pandas:

```
mtcars.loc[[6]]
```

plydata:

```
(mtcars >>dp.call('.loc'))[[6]]
```

dpylython:

```
dp.DplyFrame(mtcars) >> X._.loc[[6]]
```

dfply:

```
@dfpipe
def loc(df, a=None, b=None, c=None):
    print(type(a))
    if isinstance(a, (tuple, list)):
        indices = np.array(a)
    elif isinstance(a, pd.Series):
        indices = a.values
    elif isinstance(a, int) or isinstance(b, int) or isinstance(c, int):
```

(continues on next page)

(continued from previous page)

```
        indices = slice(a,b,c)
    return df.loc[indices, :]

mtcars >> loc([6,])
```

@dfpipe makes defining verbs easy. Converting function calls to slices is still a bit of work though.

**dppd::** dp(mtcars).loc[6,].pd()

### Replace values

R:

```
mtcars %>% mutate(cyl = replace(cyl, 4, 400))
```

pandas:

```
mtcars.assign(cyl=mtcars.cyl.replace(4, 400))
```

plydata:

```
mtcars >> dp.define(cyl='cyl.replace(4,400)') (mutate ok)
```

dpylython:

```
dp.DplyFrame(mtcars) >> dp.mutate(cyl=X.cyl.replace(4, 400))
```

dfply:

```
mtcars >>dp.mutate(cyl = X.cyl.replace(4, 400))
```

dppd:

```
dp(mtcars).mutate(cyl = X.cyl.replace(4 ,400))
```

### Advanced selecting

R:

```
mtcars %>% select(starts_with("c"))
```

starts_with only works within a dplyr construct

pandas:

```
#by boolean vector
mtcars.loc[:, mtcars.columns.str.startswith('c')]
#by list comprehension
mtcars[[x for x in mtcars.columns if x.startswith('c')]]
#by regexps, using bool vector interface
mtcars.loc[:, mtcars.columns.str.contains('^c')] # search in python re module
#by regexps using list comprehension
mtcars[[x for x in mtcars.columns if re.match('c', x)]]  # very universal
```

plydata:

```
#by select parameter
mtcars >> dp.select(startswith='c')
# Note that dplyr used starts_with
#by select parameter - regexps
mtcars >> dp.select(matches="^c")
```

select(matches=...) is actuall re.match, ie. only searchs at the start This differs from the dplyr behaviour (but is consistent with python re module) and is less general.

dpylython:

```
# bool vectors don't work. -> Exception: "None not in index"
dp.DplyFrame(mtcars) >> dp.select(X.columns.str.startswith('c'))
# list comprehensions never return.
dp.DplyFrame(mtcars) >> dp.select([x for x in X.columns if x.startswith('c')])
```

In summary: not supported

dfply:

```
# by select parameter
mtcars >> dp.select(dp.starts_with('c'))
# by select regexps
mtcars >> dp.select(dp.matches('c')) # 'search' in python re module
# by bool vector
mtcars >> dp.select(X.columns.str.startswith('c'))
# by bool vector gegexps
mtcars >> dp.select(X.columns.str.contains('^c'))
```

Faithful reproduction of dplyr but also works with the pandas-way dp.matches is actuall re.search - arguably the more useful variant, since it can be tuned to perfaorm 're.match' using '^' at the start of the regexps.

dppd:

```
# by boolean vector
dp(mtcars).select(X.columns.str.startswith('c')).pd
# by list comprehension
dp(mtcars).select([c for c in X.columns if c.startswith('c')]).pd
# by function callback
dp(mtcars).select(lambda c: c.startswith('c')).pd
# by regexps
dp(mtcars).select(('^c',)).pd
```

### Drop NaNs

**R::** TODO

pandas:

```
mtcars[~pd.isnull(mtcars.cyl)]
```

pandas:

```
mtcars.dropna(subset=['cyl'])
```

plydata:

```
mtcars >> dp.call('.dropna', subset=['cyl'])
```

dpylython:

```
dp.DplyFrame(mtcars) >> dp.sift(~X.cyl.isnull())
```

dfply:

```
mtcars >>dp.filter_by(~X.cyl.isnull())
```

but beware the inversion bug

dppd:

```
dp(mtcars).filter_by(~X.cyl.isnull()).pd
or
dp(mtcars).dropna(subset=['cyl']).pd
```

### NaN and non python-variable column name

Prelude:

```
mtcars = mtcars.assign(**{'da cyl': mtcars.cyl})
```

R:

```
mtcars = mtcars %>% mutate(`da cyl` = cyl)
mtcars %>% filter(!is.nan(cyl))
```

pandas:

```
mtcars[mtcars['da cyl'].isnull()]
```

plydata:

```
query supports no Q ('variable escape function') (bug?)
```

dpylython:

```
dp.DplyFrame(mtcars) >> dp.sift(~X['da cyl'].isnull())
```

dfply:

```
mtcars >> dp.filter_by(~X['da cyl'].isnull())
```

dppd:

```
dp(mtcars).filter_by(~X['da cyl'].isnull()).pd
```

### Filter by value wise callback

Prelude:

```
cb = lambda x: 'Europa' in str(x)
```

pandas:

```
mtcars.loc[[cb(x) for x in mtcars.name]]
```

plydata:

```
mtcars.query('[cb(x) for x in mtcars.name]') -> 'PandasExprVisitor' object has no␣
↪attribute 'visit_ListComp'
```

plydata:

```
mtcars >> dp.define(mf='[cb(x) for x in name]') >> dp.query('mf') > cb is not defined

mtcars >> dp.define(mf=[cb(x) for x in mtcars.name]) >> dp.query('mf') #note␣
↪repetition

mtcars >> dp.call('.__getitem__', [cb(x) for x in mtcars.name]) # note repetition
```

dpylython:

```
dp.DplyFrame(mtcars) >> dp.sift([cb(x) for x in X.name]) >  does not return

dp.DplyFrame(mtcars) >> dp.sift([cb(x) for x in mtcars.name]) -> list object has no␣
↪attr evaluate # bool array won't work either

mutate + list comprehension -> no return

dp.DplyFrame(mtcars) >> dp.mutate(mf= [cb(x) for x in mtcars.name]) >> dp.sift(X.mf)
↪# note reference to actual DF
```

dfply:

```
mtcars >> dp.filter_by([cb(x) for x in X.name]) _> iter() returned non-iterator of␣
↪type Intention

mtcars >> dp.mutate(mf=[cb(x) for x in mtcars.name]) >> dp.filter_by(X.mf) # note␣
↪repetition

mtcars >> dp.mutate(mf=dfply.make_symbolic(cb)(X.name)) >> dp.filter_by(X.mf) ->␣
↪always True!

Working:

mtcars >> dp.mutate(mf=dfply.make_symbolic(lambda series: [cb(x) for x in series])(X.
↪name)) >> dp.filter_by(X.mf)

mtcars >> dp.filter_by(dfply.make_symbolic(lambda series: pd.Series([cb(x) for x in
series]))(X.name))
```

Functions have to be symbol aware, and return series for filtering to work

dppd:

```
dp(mtcars).filter_by([cb(x) for x in X.name]).pd()
```

### 10: if_else column assignment

This is one where there are multiple good ways to do it.

R:

```
TODO
```

pandas:

```
mtcars.assign(high_powered = (mtcars.hp > 100).replace({True: 'yes', False: 'no'}))

mtcars.assign(high_powered = np.where(mtcars.hp > 100, 'yes', 'no'))
```

plydata:

```
mtcars >> dp.define(high_powered = dp.if_else('hp > 100', '"yes"','"no"'))

mtcars >> dp.define(high_powered = "(hp > 100).replace({True: 'yes', False: 'no'})")

mtcars >> dp.define(high_powered = "np.where(hp > 100, 'yes', 'no')")
```

dpylython:

```
dp.DplyFrame(mtcars) >> dp.mutate(high_powered=(X.hp > 100).replace({True: 'yes',␣
↪False: 'no'}))
```

No support for np.where (out of the box)

dfply:

```
mtcars >> dp.mutate(high_powered=(X.hp > 100).replace({True: 'yes', False: 'no'}))
mtcars >> dp.mutate(high_powered=dp.make_symbolic(np.where)(X.hp > 100, 'yes','no'))
```

np_where has to be wrapped in make_symbolic

dppd:

```
dp(mtcars).mutate(high_powered = (X.hp > 100).replace({True: 'yes', False: 'no'})).pd

dp(mtcars).mutate(high_powered=np.where(X.hp > 100, 'yes','no')).pd
```

### convert a column's type

pandas:

```
mtcars.assign(hp = mtcars.hp.astype(float))
```

plydata:

```
mtcars >> dp.define(hp = 'hp.astype(float)')
```

dpylython:

```
mtcars >> dp.mutate(hp = X.hp.astype(float))
```

dfply:

```
mtcars >> dp.mutate(hp = X.hp.astype(float))
```

dppd:

```
dp(mtcars).mutate(hp = X.hp.astype(float)).pd
```

### Distinct by column

R:

```
mtcars %>% distinct(mpg)
```

This drops all other columns. Pass .keep_all=T to distinct to keep them.

pandas:

```
mtcars[~mtcars.mpg.duplicated()]
```

plydata:

```
mtcars >> dp.distinct('mpg', 'first')
```

Must specify keep

dpylthon:

```
dp.DplyFrame(mtcars) >> dp.sift(~X.mpg.duplicated())
```

dfply:

```
mtcars >> dp.filter_by(~X.mpg.duplicated())
```

dppd:

```
dp(mtcars).filter_by(~X.mpg.duplicated()).pd
```

### summarize all columns by mean and std

**R::** mtcars %>% summarize_all(funs(mean, sd)) # Non-numerical columns turn into NA + a warning mtcars %>% select_if(is.numeric) %>% summarize_all(funs(mean, sd)

plydata:

```
mtcars >> dp.summarise_all((np.mean, np.std))  # exception due to the string column

mtcars >> dp.call('select_dtypes',int) >>dp.summarise_all((np.mean, np.std))

mtcars >> dp.summarize_if('is_numeric', (np.mean, np.std))
```

dpylthon:

```
# won't return - no comprehensions with X.anything
dp.DplyFrame(mtcars) >> dp.summarize(**{f"{x}_mean": X[x].mean() for x in X.columns})

#have to use the datafram, and the merge-dicts syntax (python 3.5+)
dp.DplyFrame(mtcars) >> dp.summarize(**{
  **{f"{x}_mean": X[x].mean() for x in mtcars.select_dtypes(int).columns},
  **{f"{x}_std": X[x].std() for x in mtcars.select_dtypes(int).columns}
  })
```

dfplyr:

```
mtcars >> dp.summarize(**{
  **{f"{x}_mean": X[x].mean() for x in mtcars.select_dtypes(int).columns},
  **{f"{x}_std": X[x].std() for x in mtcars.select_dtypes(int).columns}
  })
```

can't select first, because I can't iterate over X.columns

dpdp:

```
dp(mtcars).select_dtypes(np.number).summarize(*
  [(c, np.mean) for c in X.columns]
  + [(c, np.std) for c in X.columns]
).pd
```

### assign a column transformed by a function (vectorised)

R:

```
stringlength = function(x) {nchar(x)} # trivial example
mtcars %>% mutate(x = stringlength(rowname))
```

pandas:

```
def stringlength(series):
  return series.astype(str).str.len()
mtcars.assign(x = stringlength(mtcars['name']))
```

dpython:

TODO

dply:

```
mtcars >> dp.mutate(x = dp.make_symbolic(stringlength)(X.name))
```

dppd:

```
dp(mtcars).mutate(x = stringlength(X.name)).pd
```

### Iterate over groups / build a new dataframe from dataframe of groups

R:

```
dfnrow = function(df) (data.frame(count=nrow(df)))
mtcars %>% group_by(cyl) %>%  do(dfnrow(.))
```

pandas:

```
def something(grp):
    return pd.DataFrame({'count': [len(grp)]})
pd.concat([something(group).assign(cyl=idx) for idx, group in mtcars.groupby('cyl')],
 ↪axis=0)
```

pyldata:

```
mtcars >> dp.group_by('cyl') >> dp.do(something)
```

No iterator over groups

dpylython:

```
No do, no group iterator
```

dfply version 1:

```
mtcars >> dp.group_by('cyl') >> dfply.dfpipe(something)()
```

Approach 1: turn something into a verb.

dfply version 2:

```
@dfply.dfpipe
def do(df, func, *args, **kwargs):
    return func(df, *args, **kwargs)
mtcars >> dp.group_by('cyl') >> do(something)
```

Approach 2: introduce do verb.

dppd:

```
dp(mtcars).groupby('cyl').do(something).pd

or

for idx, sub_df in dp(mtcars).groupby('cyl').itergroups():
  print(idx, something(sub_df))
```

dppd has a group iterator.

### select dropping a grouping variable - what happens?

**dplython::** (dp.DplyFrame(mtcars) >> dp.group_by(X.cyl, X.am) >> dp.select(X.hp)).columns -> [‘cyl’, ‘am’, ‘hp’], grouping retained # there is no drop?

**dfply::** (mtcars >> dp.group_by(‘cyl’,’am’) >> dp.select(‘hp’)).columns -> [‘hp’], grouping is lost select (mtcars >> dp.group_by(‘cyl’,’am’) >> dp.select(‘cyl’,’am’, ‘hp’)).columns -> [‘cyl’, ‘am’, ‘hp’], grouping is retained!

(mtcars >> dp.group_by(‘cyl’,’am’) >> dp.drop(‘cyl’)).columns -> all but [‘cyl’], grouping is lost on drop

(mtcars >> dp.group_by(‘cyl’,’am’) >> dp.drop(‘hp’)).columns -> all but [‘cyl’], grouping is retained!

It is dependend on the actual columns being kept/dropped whether grouping is retained.

**dppd::** dp(mtcars).groupby([‘cyl’,’am’]).select(‘hp’).pd.columns -> [cyl, am, hp], groups still intact

dp(mtcars).groupby([‘cyl’,’am’]).drop(‘am’).pd.columns -> all columns, groups still intact

dp(mtcars).groupby([‘cyl’,’am’]).loc[:,’am’].pd.columns -> [am], grouping dropped

Verbs/pandas methods implicitly add grouping columns, accesors drop them.

### convert all int columns to float:

pandas:

```
mtcars.assign(**{x[0]: x[1] for x in mtcars.select_dtypes(int).astype(float).items()})
```

plydata:

```
mtcars >> dp.mutate(*[(x[0], list(x[1])) for x in mtcars.select_dtypes(int).
↪astype(float).items()])
```

The conversion to a list is necessary (bug?)

dplython:

```
mtcars >> dp.mutate(**{x[0]: x[1] for x in mtcars.select_dtypes(int).astype(float).
↪items()})
```

dfply:

```
mtcars >> dp.mutate(**{x[0]: x[1] for x in mtcars.select_dtypes(int).astype(float).
↪items()})
```

dppd:

```
dp(mtcars).mutate(**{x[0]: x[1] for x in mtcars.select_dtypes(int).astype(float).
↪items()}).pd
```

### Writing your own non-parametrized 'verb'

R:

```
rc = function(df) ( df[rev(colnames(df))]
mtcars %>% rc()
```

Any function taking a df as first parameter is a verb.

The python example is slightly more 'practical' in that it's not an entirely trivial function.

prelude:

```
def ints_to_floats(df):
  return df.assign(**{x[0]: x[1] for x in df.select_dtypes(int).astype(float).items()}
↪)
```

**pandas::** ints_to_floats(mtcars)

plydata:

```
no documentation on custom verbs
```

dplython:

```
@dp.ApplyToDataframe
def ints_to_floats():
  return lambda df: df.assign(**{x[0]: x[1] for x in df.select_dtypes(int).
↪astype(float).items()})

dp.DplyFrame(mtcars) >> ints_to_floats()
```

Undocumented. Note that custom verbs (and many dplython verbs, but not all of them) need to start with a dp.DplyFrame object, not a pd.DataFrame. Mutate is one exception

dfply:

```
@dp.dfpipe
def ints_to_floats(df):
  return df.assign(**{x[0]: x[1] for x in df.select_dtypes(int).astype(float).items()}
↪)
mtcars >> ints_to_floats()
```

dfpipe decorator is well documented

dppd:

```
#one more import
from dppd register_verb
dp, X = dppd()()

@register_verb()
def ints_to_floats(df):
  return df.assign(**{x[0]: x[1] for x in df.select_dtypes(int).astype(float).items()}
↪)
dp(mtcars).ints_to_floats().pd
```

### Distinct by multiple columns

R:

```
mtcars %>% distinct(mpg, am)
```

This drops all other columns. Pass .keep_all=T to distinct to keep them.

pandas:

```
mtcars[~mtcars[['mpg','am']].duplicated()]
```

plydata:

```
mtcars >> dp.distinct(['mpg', 'am'], 'first')
```

Must specify keep

dpylthon:

```
dp.DplyFrame(mtcars) >> dp.sift(~X[['mpg', 'am']].duplicated())
```

dfply:

```
mtcars >> dp.filter_by(~X[['mpg', 'am']].duplicated())
```

dppd:

```
dp(mtcars).filter_by(~X[['mpg', 'am']].duplicated()).pd
```

### Summarize 2 columns

R:

```
mtcars %>% group_by(cyl) %>% summarize(disp_min = min(disp), hp_max = max(hp))
```

pandas:

```
mtcars.groupby('cyl').agg({'disp': ['min'], 'hp': ['max']},)
```

This creates a multi index.

plydata:

```
mtcars >> dp.arrange('cyl') >> dp.group_by("cyl") >> dp.summarize('min(disp)',
↪'max(hp)')
```

The seperate sorting is necessary, plydata does not sort by default in group_by!

dplython:

```
dp.DplyFrame(mtcars) >> dp.group_by(X.cyl) >> dp.summarize(disp_min=X.disp.min(), hp_
↪max = X.hp.max())
```

dfply:

```
mtcars >> dp.group_by('cyl') >> dp.summarise(disp_min = X.disp.min(), hp_max=X.hp.
↪max())
```

dppd:

```
dp(mtcars).groupby('cyl').summarise((X.disp, np.min, 'disp_min'), (X.hp, np.max, 'hp_
↪max')).pd
```

### Summarize by quantiles

We want to summarise displacement in each cylinder-range quantiles in 0.1 increments:

R:

```
mtcars %>% group_by(cyl) %>% summarize(
  q0.1 = quantile(disp, probs=.1),
  q0.2 = quantile(disp, probs=.2),
  ...
)
```

I'm certain you could make the summarize much smarter.

pandas:

```
mtcars.sort_values('cyl').groupby('cyl')['disp'].aggregate({"q%.2f" % q: lambda x,q=q:
  x.quantile(q) for q in np.arange(0,1.1,0.1)})
```

This is sensitive to function-variable binding issue (only happens on initial define, except for default variables (common bug to forget), and using a dict for the aggregation is deprecated.

pandas version 2:

```
lambdas = [lambda x,q=q: x.quantile(q) for q in np.arange(0,1.1,0.1)]
for l, q in zip(lambdas, np.arange(0,1.1,0.1)):
  l.__name__ = "q%.2f" % q
mtcars.sort_values('cyl').groupby('cyl')['disp'].aggregate(lambdas)
```

Using named functions - not quick, but cleaner

plydata:

```
(mtcars
>> dp.arrange('cyl')
>> dp.group_by("cyl")
>> dp.summarize(**{"q%.2f" % f: "disp.quantile(%.2f)" % f for f in np.arange(0,1.1,0.
↪1)})
)
```

dplython:

```
(dp.DplyFrame(mtcars)
>> dp.arrange(X.cyl)
>> dp.group_by(X.cyl)
>> dp.summarize(**{'q%.2f' % q: X.disp.quantile(q) for q in np.arange(0,1.1,0.1)})
)
```

dfply:

```
(mtcars
>>dp.arrange("cyl")
>> dp.group_by('cyl')
>> dp.summarise(**{"q%.2f" % f: X.disp.quantile(f) for f in np.arange(0,1.1,0.1)})
)
```

dppd:

```
dp(mtcars).sort_values('cyl').groupby('cyl').summarise(*[
  ('disp', lambda x,q=q: x.quantile(q), 'q%.2f' % q) for q in np.arange(0,1.1,0.1)
]).pd
```

### concat to dataframes on their row-axis

R:

```
mtcars %>% rbind(mtcars)
```

**pandas::** pd.concat([mtcars, mtcars])

plydata:

```
?
```

dpylython:

```
?
```

dfply:

```
mtcars >> bind_rows(mtcars)
```

dppd:

```
dp(mtcars).concat([mtcars]).pd
```

#### transpose

R:

```
mtcars %>% t() %>% as.tibble
```

Base R function t()'s result must be converted back into a tibble

pandas:

```
mtcars.transpose()
```

plydata:

```
mtcars >> dp.call('transpose')
```

**dpylython::** dp.DplyFrame(mtcars) >> X._.transpose()

It is undocumented when and when not you can use *mtcars* >> and when you have to use DplyFrame

dfply:

```
@dp.dfpipe
def call(df, method, *args, **kwargs):
  return getattr(df, method)(*args, **kwargs)
mtcars >> call('transpose')
```

dfply has no fall back to pandas methods - this introduces such a fallback instead of wrapping transpose.

dfpyl version 2:

```
mtcars >> dp.dfpipe(pd.DataFrame.transpose)
```

We could also wrap the classes method in a pipe instead

dppd:

```
dp(mtcars).transpose().pd
```

### 1.5.5 Summary:

## Summary notes:

> None of the X-marks-the-non-standard-evaluation is complete with regard to python's capabilitys - they already fail at such basic things as transform a function by a column.
>
> The 'eval a string' approach is better in this regard, but still fails for example on list comprehensions.
>
> > all: could benefit from a 'delayed' row or valuewise callback function, both for columns and for rows. plydata (patsy?) might actually support list comprehensions!

plydata:

could benefit from loc/iloc ver unclean exprots (dataframe) no concat query supports no Q if_else is unecessary -> replace does the job (case when ditto?)

**dfply::** missing loc verb unclean exports (warnings) the easy of dfpipe must not be underestimated

**dpylython:** unnecessary casting, results are not pd.DataFrame, (define rrshift on every method?) unclean exports (types) no concat select really rudimentary / does not take lists? unclear when you can do df >> and when you have to do Dplyframe(df)

## 1.6 Extending dppd

### 1.6.1 Known extensions

- dppd_plotnine allows plotting with plotnine, the python ggplot implementation.

### 1.6.2 Custom verbs

Writing your own verbs can be as easy as sticking `@register_verb()` on a function.

Example:

```
>>>from dppd import register_verb
>>> @register_verb()
... def sideways_head(df, n=5):
...   return df.iloc[:, :n]
...
>>> dp(mtcars).sideways_head(2).pd.head()
                name   mpg
0          Mazda RX4  21.0
1      Mazda RX4 Wag  21.0
2         Datsun 710  22.8
3     Hornet 4 Drive  21.4
4  Hornet Sportabout  18.7
```

A verb registered without passing a types argument to *dppd.base.register_verb* is registered for all types. `sideways_head` does raise an Exception on DataFrameGroupBy objects though, since those don't support iloc.

Let's register a verb specifically for those:

```
>>> @register_verb('sideways_head', types=pd.core.groupby.groupby.DataFrameGroupBy)
... def sideways_head_DataFrameGroupBy(grp, n=5):
...   return grp.apply(lambda X: X.iloc[:,: n])
...
>>> dp(mtcars).groupby('cyl').sideways_head(5).pd.head()
                name   mpg  cyl   disp   hp
0          Mazda RX4  21.0    6  160.0  110
1      Mazda RX4 Wag  21.0    6  160.0  110
2         Datsun 710  22.8    4  108.0   93
3     Hornet 4 Drive  21.4    6  258.0  110
4  Hornet Sportabout  18.7    8  360.0  175
```

### 1.6.3 Extending to other types

Dppd() objects dispatch their verbs on the type of their wrapped object. register_verbs accepts a types argument which can be a single type or a list of types. register_type_methods_as_verbs registers all methods of a type (minus an exclusion list) as verbs for that type.

This allows you to define verbs on arbritrary types.

Just for kicks, because update on dict should always have returned the original dict:

```
>>> register_type_methods_as_verbs(dict, ['update'])
>>> @register_verb('update', types=dict)
... def update_dict(d, other):
  ... res = d.copy()
  ... res.update(other)
  ... return res

>>> @register_verb('map', types=dict)
... def map_dict(d, callback):
...  return {k: callback(d[k]) for k in d}

>>> print(dp({'hello': 'world'}).update({'no': 'regrets'}).map(str.upper).pd)
{'hello': 'WORLD', 'no': 'REGRETS'}
```

## 1.7 License



## 1.8 Contributors

- Florian Finkernagel <finkernagel@imt.uni-marburg.de>

## 1.9 Changelog

### 1.9.1 unreleased

### 1.9.2 0.25

- added .debug to print head&tail inline

### 1.9.3 0.24

- allow dp(collections.Counter).to_frame(key_name='x', count_name='y').pd

### 1.9.4 0.23

- pca() on dataframe
- 'natsorted' for categoricals
- select_and_rename (select no longer renames!)
- level selection in column specifications (by passing in a list of regexps, might be shorter than the number of levels)

### 1.9.5 0.22

- added binarize
- added dp({}).to_frame()
- minor bugfixes and polishing
- improved docs a bit

### 1.9.6 0.21

- Fleshed out reset_columns

### 1.9.7 0.20

- added rename_columns/reset_columns

### 1.9.8 0.19

- support for itertuples on groupBy objects
- column specs now support types (and forward the query to select_dtypes)
- column spec now accepts [True] as 'same columns, but sorted alphabetically'
- ends() for DataFrames
- categorize now by default keeps order as seen in the Series. Pass None to restore old behaviour

### 1.9.9 0.17

### 1.9.10 0.16

- replaced alias_verb by extending register_verb(name=...) to register_verb(names=[...], ...)
- support for pandas 0.22.0

### 1.9.11 0.15

- X is now 'stacked' - dp(...).pd now replaces the X with the one just before the last dp(..) call.

### 1.9.12 0.1

- initial release

## 1.10 dppd

### 1.10.1 dppd package

#### Submodules

#### dppd.base module

**class** dppd.base.**Dppd**(*df*, *dppd_proxy*, *X*, *parent*)
Bases: object

Dataframe maniPulater maniPulates Dataframes

A DataFrame manipulation object, offering verbs, and each verb returns another Dppd.

All pandas.DataFrame methods have been turned into verbs. Accessors like loc also work.

**pd**
Return the actual, unproxyied DataFrame

**class** dppd.base.**dppd**(*df=None*)
Bases: object

Context manager for Dppd.

Usage:

```
```
with cdp(mtcars) as (dp, X):
    dp.groupby('cyl')
    dp.arrange(X.hp))
    dp.head(1)
print(X)
```
```

Both X and dp are a proxyied DataFrame after the context manager. They should work just like a DataFrame, use X.pd() to convert it into a true DataFrame.

Alternate usage:

```
dp, X = dppd()
dp(df).mutate(y=X['column'] * 2, ...).filter(...).select(...).pd
```

or:

```
dp(df).mutate(...)
dp.filter()
dp.select()
new_df = dp.pd
```

dppd.base.**register_property**(*name*, *types=None*)
Register a property/indexed accessor to be forwarded (.something[])

dppd.base.**register_type_methods_as_verbs**(*cls*, *excluded*)

**class** dppd.base.**register_verb**(*name=None*, *types=None*, *pass_dppd=False*)
Bases: object

Register a function to act as a Dppd verb. First parameter of the function must be the DataFrame being worked on. Note that for grouped Dppds, the function get's called once per group.

Example:

```
register_verb('upper_case_all_columns')(
    lambda df: df.assign(**{
        name: df[name].str.upper() for name in df.columns})
```

### dppd.column_spec module

dppd.column_spec.**parse_column_specification**(*df*, *column_spec*, *return_list=False*)
Parse a column specification

**Parameters**

- **column_spec** (*various*) –
  - str, [str] - select columns by name, (always returns an DataFrame, never a Series)
  - [b, a, -c, True] - select b, a (by name, in order) drop c, then add everything else in alphabetical order
  - pd.Series / np.ndarray, dtype == bool: select columns matching this bool vector, example: `select(X.name.str.startswith('c'))`
  - pd.Series, [pd.Series] - select columns by series.name
  - "-column_name" or ["-column_name1","-column_name2"]: drop all other columns (or invert search order in arrange)
  - pd.Index - interpreted as a list of column names - example: select(X.select_dtypes(int).columns)
  - (regexps_str, ) tuple - run re.search() on each column name
  - (regexps_str, None, regexps_str ) tuple - run re.search() on each level of the column names. Logical and (like DataFrame.xs but more so).
  - {level: regexs_str,...) - re.search on these levels (logical and)
  - a callable f, which takes a string column name and returns a bool whether to include the column.

  - a type, in which case the request will be forwarded to pandas.DataFrame.select_dtypes(include=...)). Example: numpy.number
  - None -> all columns
- **return_list** (*int*) –
  - If return_list is falsiy, return a boolean vector.
  - If return_list is True, return a list of columns, either in input order (if available), or in df_columns order if not.
  - if return_list is 2, return (forward_list, reverse_list) if input was a list, other wise see 'return_list is True'

dppd.column_spec.**series_and_strings_to_names**(*columns*)

### dppd.non_df_verbs module

dppd.non_df_verbs.**collection_counter_to_df**(*counter*, *key_name='key'*, *count_name='counts'*)
Turn a collections.Counter into a DataFrame with two columns: key & count

### dppd.single_verbs module

dppd.single_verbs.**add_count**(*df*)
Verb: Add the cardinality of a row's group to the row as column 'count'

dppd.single_verbs.**arrange_DataFrame**(*df*, *column_spec*, *kind='quicksort'*, *na_position='last'*)
Sort DataFrame based on column spec.

Wrapper around sort_values

**Parameters**

- **column_spec** (*column specification*) – see dppd.single_verbs.parse_column_specification()
- .. (see pandas.DataFrame.sort_values()) –

dppd.single_verbs.**arrange_DataFrameGroupBy**(*grp*, *column_spec*, *kind='quicksort'*, *na_position='last'*)

dppd.single_verbs.**astype_DataFrame**(*df*, *columns*, *dtype*, ***kwargs*)

dppd.single_verbs.**binarize**(*df*, *col_spec*, *drop=True*)
Convert categorical columns into 'regression columns', i.e. X with values a,b,c becomes three binary columns X-a, X-b, X-c which are True exactly where X was a, etc.

dppd.single_verbs.**categorize_DataFrame**(*df*, *columns=None*, *categories=<object object>*, *ordered=None*)
Turn columns into pandas.Categorical. By default, they get ordered by their occurrences in the column. You can pass False, then pd.Categorical will sort alphabetically, or 'natsorted', in which case they'll be passed through natsort.natsorted

dppd.single_verbs.**colspec_DataFrame**(*df*, *columns*, *invert=False*)
Return columns as defined by your column specification, so you can use colspec in set_index etc

- column specification, see dppd.single_verbs.parse_column_specification()

dppd.single_verbs.**concat_DataFrame**(*df*, *other*, *axis=0*)
Verb: Concat this and one ore multiple others.

Wrapper around `pandas.concat()`.

**Parameters**

- **other** (*df or [df, df, ..]*) –
- **axis** (*join on rows (axis= 0) or columns (axis = 1)*) –

dppd.single_verbs.**distinct_dataframe**(*df*, *column_spec=None*, *keep='first'*)
Verb: select distinct/unique rows

**Parameters**

- **column_spec** (*column specification*) – only consider these columns when deciding on duplication see `dppd.single_verbs.parse_column_specification()`
- **keep** (*str*) – which instance to keep in case of duplicates (see `pandas.DataFrame.duplicated()`)

**Returns** with possibly fewer rows, but unchanged columns.

**Return type** DataFrame

dppd.single_verbs.**distinct_series**(*df*, *keep='first'*)
Verb: select distinct values from Series

**Parameters** **keep** (which instance to keep in case of duplicates (see `pandas.Series.duplicated()`)) –

dppd.single_verbs.**do**(*obj*, *func*, **args*, ***kwargs*)
Verb: Do anything to any DataFrame, returning new dataframes

Apply func to each group, collect results, concat them into a new DataFrame with the group information.

**Parameters** **func** (*callable*) – Should take and return a DataFrame

Example:

```
>>> def count_and_count_unique(df):
...     return pd.DataFrame({"count": [len(df)], "unique": [(~df.duplicated()).
↪sum()]})
...
>>> dp(mtcars).select(['cyl','hp']).group_by('cyl').do(count_and_count_unique).pd
cyl  count  unique
0    4     11      10
1    6      7       4
2    8     14       9
```

dppd.single_verbs.**drop_DataFrameGroupBy**(*grp*, **args*, ***kwargs*)

dppd.single_verbs.**ends**(*df*, *n=5*)
Head(n)&Tail(n) at once

dppd.single_verbs.**filter_by**(*obj*, *filter_arg*)
Filter DataFrame

**Parameters** **filter_arg** (`Series` or `array` or callable or dict or str) – # * Series/Array dtype==bool: return by .loc[filter_arg] * callable: Excepted to return a Series(dtype=bool) * str: a column name -> .loc[X[filter_arg].astype(bool)]

dppd.single_verbs.**gather**(*df*, *key*, *value*, *value_var_column_spec=None*)
Verb: Gather multiple columns and collapse them into two.

This used to be called melting and this is a column spec aware forward for pd.melt

Paramter order is from dplyr.

**Parameters**

- **key** (*str*) – name of the new 'variable' column
- **value** (*str*) – name of the new 'value' column
- **value_var_column_spec** (*column specification*) – which columns contain the values to be mapped into key/value pairs? see dppd.single_verbs.parse_column_specification()

Inverse of *dppd.single_verbs.spread*.

### Example

```
>>> dp(mtcars).select(['name','hp', 'cyl']).gather('variable', 'value', '-name').
↪pd.head()
                name variable  value
0          Mazda RX4       hp    110
1      Mazda RX4 Wag       hp    110
2         Datsun 710       hp     93
3     Hornet 4 Drive       hp    110
4  Hornet Sportabout       hp    175
```

dppd.single_verbs.**group_extract_params**(*grp*)

dppd.single_verbs.**group_variables**(*grp*)

dppd.single_verbs.**identity**(*df*)
Verb: No-op.

dppd.single_verbs.**iter_tuples_DataFrameGroupBy**(*grp*)

dppd.single_verbs.**itergroups_DataFrame**(*df*)

dppd.single_verbs.**itergroups_DataFrameGroupBy**(*grp*)

dppd.single_verbs.**log2**(*df*)

dppd.single_verbs.**mutate_DataFrame**(*df*, ***kwargs*)
Verb: add columns to a DataFrame defined by kwargs:

**Parameters** **kwargs** (*scalar, pd.Series, callable, dict*) –

- scalar, pd.Series -> assign column
- callable - call callable(df) and assign result
- dict (None: column) - result of itergroups on non-grouped DF to have parity with mutate_DataFrameGroupBy

### Examples

add a rank for one column:

```
dp(mtcars).mutate(hp_rank = X.hp.rank)
```

rank all columns:

```
# dict comprehension for illustrative purposes
dp(mtcars).mutate(**{f"{column}_rank": X[column].rank() for column in X.columns}).
↪pd
# more efficient
dp(mtcars).rank().pd()
```

one rank per group using callback:

```
dp(mtcars).group_by('cyl').mutate(rank = lambda X: X['hp'].rank()).pd
```

add_count variant 1 (see *dppd.single_verbs.add_count()*):

```
dp(mtcars).group_by('cyl').mutate(count=lambda x: len(x)).pd
```

add_count variant 2:

```
dp(mtcars).group_by('cyl').mutate(count={grp: len(sub_df) for (grp, sub_df) in X.
↪itergroups()}).pd
```

dppd.single_verbs.**mutate_DataFrameGroupBy**(*grp*, ***kwargs*)

Verb: add columns to the DataFrame used in the GroupBy.

**Parameters** ****kwargs** (*scalar, pd.Series, callable, dict*) –

- scalar, pd.Series -> assign column
- callable - call callable once per group (sub_df) and assign result
- dict {grp_key: scalar_or_series}: assign this (these) value(s) for the group. Use in conjunction with *dppd.Dppd.itergroups*.

dppd.single_verbs.**natsort_DataFrame**(*df*, *column*)

dppd.single_verbs.**norm_0_to_1**(*df*, *axis=1*)

Normalize a (numeric) data frame so that it goes from 0 to 1 in each row (axis=1) or column (axis=0) Usefully for PCA, correlation, etc. because then the dimensions are comparable in size

dppd.single_verbs.**norm_zscore**(*df*, *axis=1*)

apply zcore transform (X - mu) / std via scipy.stats.zcore an the given axis

dppd.single_verbs.**pca_dataframe**(*df*, *whiten=False*, *random_state=None*)

Perform 2 component PCA using sklearn.decomposition.PCA. Expects samples in rows! Returns a tuple (DataFrame{sample, 1st, 2nd}, whith an additiona l, **explained_variance_ratio_** attribute

dppd.single_verbs.**print_DataFrameGroupBy**(*grps*)

dppd.single_verbs.**reset_columns_DataFrame**(*df*, *new_columns=None*)

Rename *all* columns in a dataframe (and return a copy). Possible new_columns values:

- None: df.columns = list(df.columns)
- List: df.columns = new_columns
- callable: df.columns = [new_columns(x) for x in df.columns]
- str && df.shape[1] == 1: df.columns = [new_columns]

new_columns=None is useful when you were transposing categorical indices and now can no longer assign columns. (Arguably a pandas bug)

dppd.single_verbs.**select_DataFrame**(*df*, *columns*)
Verb: Pick columns from a DataFrame

Improved variant of df[columns]

**Parameters**

- **columnns** (*column specifiation or dict*) – see dppd.single_verbs.parse_column_specification()
- **the previous 'rename on dict' behaviour, see select_and_rename**(*(for)* –

dppd.single_verbs.**select_DataFrameGroupBy**(*grp*, *columns*)

dppd.single_verbs.**select_and_rename_DataFrame**(*df*, *columns*)
Verb: Pick columns from a DataFrame, and rename them in the process

**Parameters** **columns** (*dict {new_name: 'old_name'} - select and rename. old_name may be a str, or a*) – Series (in which case the .name attribute is used)

dppd.single_verbs.**seperate**(*df*, *column*, *new_names*, *sep='.'*, *remove=False*)
Verb: split strings on a seperator.

Inverse of *unite()*

**Parameters**

- **column** (*str or pd.Series*) – column to split on (Series.name is named in case of a series)
- **new_names** (*list*) – list of new column names
- **sep** (*str*) – what to split on (pd.Series.str.split)
- **remove** (*bool*) – wether to drop column

dppd.single_verbs.**sort_values_DataFrameGroupBy**(*grp*, *column_spec*, *kind='quicksort'*, *na_position='last'*)
Alias for arrange for groupby-objects

dppd.single_verbs.**spread**(*df*, *key*, *value*)
Verb: Spread a key-value pair across multiple columns

**Parameters**

- **key** (*str or pd.Series*) – key column to spread (if series, .name is used)
- **value** (*str or pd.Series*) – value column to spread (if series, .name is used)

Inverse of *dppd.single_verbs.gather*.

### Example

```
>>> df = pd.DataFrame({'key': ['a','b'] * 5, 'id': ['c','c','d','d','e','e','f','f
↪','g','g'], 'value':np.random.rand(10)})
>>> dp(df).spread('key','value')
>>> dp(df).spread('key','value').pd
key id         a         b
0    c  0.650358  0.931324
1    d  0.633024  0.380125
2    e  0.983000  0.367837
```

(continues on next page)

(continued from previous page)

```
3    f  0.989504  0.706933
4    g  0.245418  0.108165
```

dppd.single_verbs.**summarize**(*obj*, **args*)
Summarize by group.

**Parameters** ***args** (*tuples*) – (column_to_use, function_to_call) or (column_to_use, function_to_call, new_column_name)

dppd.single_verbs.**to_frame_dict**(*d*, ***kwargs*)
`pd.DataFrame.from_dict(d, **kwargs)`, so you can say `dp({}).to_frame()`

dppd.single_verbs.**transassign**(*df*, ***kwargs*)
Verb: Creates a new dataframe from the columns of the old.

This means that the index and row count is preserved

dppd.single_verbs.**ungroup_DataFrameGroupBy**(*grp*)

dppd.single_verbs.**unique_in_order**(*seq*)

dppd.single_verbs.**unite**(*df*, *column_spec*, *sep='_'*)
Verb: string join multiple columns

**Parameters**

- **column_spec** (*column_spec*) – which columns to join. see `dppd.single_verbs.parse_column_specification()`
- **sep** (*str*) – Seperator to join on

dppd.single_verbs.**unselect_DataFrame**(*df*, *columns*)
Verb: Select via an inversed column spec (ie. everything but these)

**Parameters** **columnns** (*column specifiation or dict*) –

- column specification, see `dppd.single_verbs.parse_column_specification()`

dppd.single_verbs.**unselect_DataFrameGroupBy**(*grp*, *columns*)
Verb: Select via an inversed column spec (ie. everything but these)

**Parameters** **columnns** (*column specifiation or dict*) –

- column specification, see `dppd.single_verbs.parse_column_specification()`

### Module contents

CHAPTER 2

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## d

# Index

## R



## S



## T



## U