Quantipy Documentation

Release 0.1.3

Kerstin Müller, Alexander Buchhammer, Alasdair Eaglestone, Jam

Jul 18, 2023

Contents

CHAPTER 1

Latest (09/04/2019)

New Nesting in `Batch.add_crossbreak()`

Nested crossbreaks can be defined for Excel deliverables, the nesting can be defined by `"var1 > var2"`. Nesting in more than two levels is available `"var1 > var2 > var3 > ..."`, but nesting a group of variables is NOT supported *"var1 > (var2, var3)"*.

New Leveling

Running `Batch.level(array, levels={})` gives the option to aggregate leveled arrays. If no `levels` are provided, automatically the `Batch.yks` are taken.

New `DataSet.used_text_keys()`

This new method loops over text objects in `DataSet._meta` and returns all found `text_keys`.

Update Batch (transposed) summaries

As announced a while ago, `Batch.make_summaries()` is fully deprecated now and gives a NotImplementedError. Per default, all arrays in the downbreak list are added to the `Batch.x_y_map`. The array `exclusive` functionality (add array, but skip items) is now supported by the new method `Batch.exclusive_arrays()`.

Additionally `Batch.transpose_array()` is deprecated. Instead `Batch.transpose()` is available, which does not support `replace` anymore, because the "normal" arrays needs to be included always. If the summaries are not requested in the deliverables, they can be hidden in the ChainManager.

CHAPTER 2

Archived release notes

2.1 sd (14/01/2019)

New: `Chain.export()` / `assign()` and custom calculations

Expanding on the current `Chain` editing features provided via `cut()` and `join()`, it is now possible to calculate additional row and column results using plain `pandas.dataframe` methods. Use `Chain.export()` to work on a simplified `Chain.dataframe` and `assign()` to rebuild it properly when finished.

New: `Batch.as_main(keep=True)` to change `qp.Batch` relations

It is now possible to promote an `.additional` Batch to a main/regular one. Optionally, the original parent Batch can be erased by setting `keep=False`. Example:

Starting from:

```
>>> dataset.batches(main=True, add=False)
['batch 2', 'batch 5']
```

```
>>> dataset.batches(main=False, add=True)
['batch 4', 'batch 3', 'batch 1']
```

We turn `batch 3` *into a normal one:*

```
>>> b = dataset.get_batch('batch 3')
>>> b.as_main()
```

```
>>> dataset.batches(main=True, add=False)
['batch 2', 'batch 5', 'batch 3']
```

```
>>> dataset.batches(main=False, add=True)
['batch 4', 'batch 1']
```

New: On-the-fly rebasing via `Quantity.normalize(on='y', per_cell=False)`

Quantipy's engine will now accept another variable's base for (column) percentage computations. Furthermore, it is possible to rebase the percentages to the *cell frequencies of the other variable's cross-tabulation* by setting `per_cell=True`, i.e. rebase variables with identical categories to their respective per-category results. The following example shows how `'A1'` results are serving as cell bases for the percentages of `'A2'`:

```
>>> l = stack[stack.keys()[0]]['no_filter']['A1']['datasource']
>>> q = qp.Quantity(l)
>>> q.count()
Question        datasource
Values                 All      1       2       3      4       5       6
Question Values
A1       All        6984.0  767.0  1238.0  2126.0  836.0  1012.0  1005.0
         1          1141.0  503.0    78.0   109.0  102.0   155.0   194.0
         2          2716.0  615.0   406.0   499.0  499.0   394.0   303.0
         3          1732.0  603.0    89.0   128.0  101.0   404.0   407.0
         4          5391.0  644.0   798.0  1681.0  655.0   796.0   817.0
         5          4408.0  593.0   177.0  1649.0  321.0   818.0   850.0
         6          3584.0  615.0   834.0   834.0  327.0   507.0   467.0
         7          4250.0  588.0   724.0  1717.0  540.0    55.0   626.0
         8          3729.0  413.0  1014.0   788.0  311.0   539.0   664.0
         9          3575.0  496.0   975.0   270.0  699.0   230.0   905.0
         10         4074.0  582.0   910.0  1148.0  298.0   861.0   275.0
         11         2200.0  446.0   749.0   431.0  177.0   146.0   251.0
         12         5554.0  612.0   987.0  1653.0  551.0   860.0   891.0
         13          544.0   40.0   107.0   232.0   87.0    52.0    26.0
```

```
>>> l = stack[stack.keys()[0]]['no_filter']['A2']['datasource']
>>> q = qp.Quantity(l)
>>> q.count()
Question        datasource
Values                 All      1       2       3      4      5      6
Question Values
A2       All        6440.0  727.0  1131.0  1894.0  749.0  960.0  979.0
         1           568.0  306.0    34.0    32.0   48.0   63.0   85.0
         2          1135.0  417.0   107.0    88.0  213.0  175.0  135.0
         3           975.0  426.0    43.0    49.0   49.0  220.0  188.0
         4          2473.0  350.0   267.0   599.0  431.0  404.0  422.0
         5          2013.0  299.0    88.0   573.0  162.0  417.0  474.0
         6          1174.0  342.0   219.0   183.0  127.0  135.0  168.0
         7          1841.0  355.0   161.0   754.0  285.0   21.0  265.0
         8          1740.0  265.0   376.0   327.0  160.0  212.0  400.0
         9          1584.0  181.0   390.0    89.0  398.0   94.0  432.0
         10         1655.0  257.0   356.0   340.0  137.0  443.0  122.0
         11          766.0  201.0   241.0   101.0   76.0   53.0   94.0
         12         2438.0  217.0   528.0   497.0  247.0  459.0  490.0
         13         1532.0   72.0   286.0   685.0  118.0  183.0  188.0
```

```
>>> q.normalize(on='A1', per_cell=True)
Question         datasource
Values                  All          1          2          3          4
↪5          6
Question Values
A2       All       92.210767   94.784876   91.357027   89.087488   89.593301   94.
↪861660   97.412935
         1         49.780894   60.834990   43.589744   29.357798   47.058824   40.
↪645161   43.814433
         2         41.789396   67.804878   26.354680   17.635271   42.685371   44.
↪416244   44.554455
```

(continues on next page)

(continued from previous page)

```
         3        56.293303   70.646766   48.314607   38.281250   48.514851   54.
↪455446   46.191646
         4        45.872751   54.347826   33.458647   35.633551   65.801527   50.
↪753769   51.652387
         5        45.666969   50.421585   49.717514   34.748332   50.467290   50.
↪977995   55.764706
         6        32.756696   55.609756   26.258993   21.942446   38.837920   26.
↪627219   35.974304
         7        43.317647   60.374150   22.237569   43.913803   52.777778   38.
↪181818   42.332268
         8        46.661303   64.164649   37.080868   41.497462   51.446945   39.
↪332096   60.240964
         9        44.307692   36.491935   40.000000   32.962963   56.938484   40.
↪869565   47.734807
         10       40.623466   44.158076   39.120879   29.616725   45.973154   51.
↪451800   44.363636
         11       34.818182   45.067265   32.176235   23.433875   42.937853   36.
↪301370   37.450199
         12       43.896291   35.457516   53.495441   30.066546   44.827586   53.
↪372093   54.994388
         13      281.617647  180.000000  267.289720  295.258621  135.632184  351.
↪923077  723.076923
```

New: `DataSet.missings(name=None)`

This new method returns the missing data definition for the provided variable or all missing definitions found in the dataset (if `name` is omitted).

```
>>> dataset.missings()
{u'q10': {u'exclude': [6]},
 u'q11': {u'exclude': [977]},
 u'q17': {u'exclude': [977]},
 u'q23_1_new': {u'exclude': [8]},
 u'q25': {u'exclude': [977]},
 u'q32': {u'exclude': [977]},
 u'q38': {u'exclude': [977]},
 u'q39': {u'exclude': [977]},
 u'q48': {u'exclude': [977]},
 u'q5': {u'exclude': [977]},
 u'q9': {u'exclude': [977]}}
```

Update: `DataSet.batches(main=True, add=False)`

The collection of `Batch` sets can be separated by `main` and additional ones (see above) to make analyzing Batch setups and relations easier. The default is still to return all Batch names.

Bugfix: Stack, `other_source` statistics failing for delimited sets

A bug that prevented `other_source` statistics being computed on delimited set type variables has been resolved by adjusting the underlying data type checking mechanic.

2.2 sd (26/10/2018)

New: Filter variables in DataSet and Batch

To avoid complex logics stored in the background and resulting problem with json serializing, the filter concept in DataSet and Batch has changed.

Now actual variables are added to the data and meta, which have the property `recoded_filter`. The values of depend on the included logic, and all logics summarized in the value `0: keep`. Because of the an easy logic can be used at several places in qp: `{'filter_var': 0}`

DataSet methods

All included filters of a Dataset can be shown running `dataset.filters()`.

A filter variable can be easily created:

```
dataset.add_filter_var(name, logic, overwrite=False)
```

- `name` is the name of the new filter-variable.
- `logic` should be (a list of) dictionaries in form of:

  ```
  >>> {
  ...     'label': 'reason',
  ...     'logic': {var: keys} / intersection/ ....
  ... }
  ```

  or stings (var_name), which are automatically transformed into the following dict

  ```
  >>> {
  ...     'label': 'var_name not empty',
  ...     'logic': {var_name: not_count(0)}
  ... }
  ```

  If a list is provided, each item results in an own value of the filter variable.

An existing filter variable can also be extended:

```
dataset.extend_filter_var(name, logic, extend_as=None)
```

- `name` is the name of the existing filter-variable.
- `logic` should be the same as above, then additional categories are added to the filter and the 0 value is recalculated.
- `extend_as` determines if a new filter var is created or the initial variable is modified. If `extend_as=None` the variable is modified inplace. Otherwise `extend_as` is used as suffix for the new filter variable.

Known methods like:

```
.copy()
.drop()
.rename()
```

can be applied on filter-variables, all others are not valid!

Batch methods

`Batch.add_filter(filter_name, filter_logic=None, overwrite=False)`

A filter can still be added to a batch, by adding a `filter_logic`, but also it's possible to add only the `filter_name` of an existing filter variable. If `filter_name` is an existing filter-variable, a `filter_logic` is provided and `overwrite` is turned off, the scripts will return an error.

`Batch.remove_filter()`

This method only removes filters from the Batch definitions, the created filter-variables still exist in the belonging DataSet object.

Batch methods that use filters:

```
.extend_filter()
.add_y_on_y()
.add_open_ends()
```

create new extended filter variables if the used filter differs from the batch global filter. So it's recommended to add the global filter first, it's taken over automatically for the mentioned methods.

New: Summarizing and rearranging `qp.Chain` elements via `ChainManager`

- `cut(values, ci=None, base=False, tests=False)`
- `join(title='Summary')`

It is now possible to summarize `View` aggregation results from existing `Chain` items by restructuring and editing them via their `ChainManager` methods. The general idea behind building a summary `Chain` is to unify a set of results into items by restructuring and editing them via their `ChainManager` methods. The general idea behind building a summary Chain is to unify a set of results into one cohesive representation to offer an easy way to look at certain key figures of interest in comparison to each other. To achieve this, the `ChainManager` class has gained the new `cut()` and `join()` methods. Summaries are built post- aggregation and therefore rely on what has been defined (via the `qp.Batch` class) and computed (via the `qp.Stack` methods) at previous stages.

The intended way of working with this new feature can be outlined as

1. `reorder()`
2. `cut()`
3. `join()`
4. `insert()`

In more detail:

A) Grouping the results for the summary

Both methods will operate on the *entire set* of `Chains` collected in a `ChainManager`, so building a summary `Chain` will normally start with restricting a copy of an existing `ChainManager` to the question variables that you're interested in. This can be done via `clone()` with `reorder(..., inplace=True)` or by assigning back the new instance from `reorder(..., inplace=False)`.

B) Selecting View results via `cut()`

This method lets you target the kind of results (nets, means, NPS scores, only the frequencies, etc.) from a given `qp.Chain.dataframe`. Elements must be targeted by their underlying regular index values, e.g. `'net_1'`, `'net_2'`, `'mean'`, `1`, `'calc'`, `...` . Use the `base` and `tests` parameters to also carry over the matching base rows and/or significance testing results. The `ci` parameter additionally allows targeting only the `'counts'` or `'c%'` results if cell items are grouped together.

C) Unifying the individual results with `join()`

Merging all new results into one, the `join()` method concatenates vertically and relabels the x-axis to separate all variable results by their matching metadata `text` that has also been applied while creating the regular set of and relabels the x-axis to separate all variable results by their matching metadata `text` that has has also been applied while creating the regular set of `Chain` items. The new summary can then also be inserted back into its originating `ChainManager` with `insert()` if desired.

Update: `Batch.add_variables(varlist)`

A `qp.Batch` can now carry a collection of variables that is **explicitly not** directed towards any table-like builds. Variables from `varlist` will solely be used in non-aggregation based, data transformation and export oriented applications. To make this distinction more visible in the API, `add_x()` and `add_y()` have been renamed to `add_downbreak()` and `add_crossbreak()`. Users are warned and advised to switch to the new method versions via a `DeprecationWarning`. In a future version of the library `add_x()` and `add_y()` will be removed.

Update: `Batch.copy()` -> `Batch.clone()`

Since `qp.Batch` is a subclass of `qp.DataSet`, the `copy()` method is renamed into `Batch.clone()`.

2.3 sd (01/10/2018)

New: "rewrite" of Rules module (affecting sorting):

sorting "normal" columns:

- `sort_on` always '@'
- `fix` any categories
- `sort_by_weight` default is unweighted (None), but each weight (included

in data) can be used

If sort_by_weight and the view-weight differ, a warning is shown.

sorting "expanded net" columns:

- `sort_on` always '@'
- `fix` any categories
- sorting `within` or `between` net groups is available
- `sort_by_weight`: as default the weight of the first found

expanded-net-view is taken. Only weights of aggregated net-views are possible

sorting "array summaries":

- `sort_on` can be any desc ('median', 'stddev', 'sem', 'max', 'min',

'mean', 'upper_q', 'lower_q') or nets ('net_1', 'net_2', ... enumerated by the net_def) * `sort_by_weight`: as default the weight of the first found desc/net-view is taken. Only weights of aggregated desc/net-views are possible * `sort_on` can also be any category, here each weight can be used to sort_on

---

New: `DataSet.min_value_count()`

A new wrapper for `DataSet.hiding()` is included. All values are hidden, that have less counts than the included number `min`. The used data can be weighted or filtered using the parameters `weight` and `condition`.

Usage as Batch method: `Batch.min_value_count()` without the parameters `weight` and `condition` automatically grabs `Batch.weights[0]` and `Batch.filter` to calculate low value counts.

New: Prevent weak duplicated in data

As Python is case sensitive it is possible to have two or more variables with the same name, but in lower- and uppercases. Most other software do not support that, so a warning is shown if a weak dupe is created. Additionally `Dataset.write_dimensions()` performs auto-renaming is weak dupes are detected.

New: Prevent single-cat delimited sets

`DataSet.add_meta(..., qtype='delimited set', categories=[...], ...)` automatically switches `qtype` to single if only one category is defined. `DataSet.convert(name, 'single')` allows conversion from `delimited set` to `single` if the variable has only one category. `DataSet.repair()` and `DataSt.remove_values()` convert delimited sets automatically to singles if only one category is included.

Update: merge warnings + merging delimites sets

Warnings in `hmerge()` and `vmerge()` are updated. If a column exists in the left and the right dataset, the type is compared. Some type inconsistencies are allowed, but return a warning, while others end up in a raise.

delimited sets in `vmerge()`:

If a column is a delimited set in the left dataset, but a single, int or float in the right dataset, the data of the right column is converted into a delimited set.

delimited sets in `hmerge(...merge_existing=None)`:

For the hmerge a new parameter `merge_existing` is included, which can be `None`, a list of variable-names or `'all'`.

If delimited sets are included in left and right dataset:

- `merge_existing=None`: Only meta is adjusted. Data is untouched (left data

is taken). * `merge_existing='all'`: Meta and data are merged for all delimited sets, that are included in both datasets. * `merge_existing=[variable-names]`: Meta and data are merged for all delimited sets, that are listed and included in both datasets.

Update: encoding in `DataSet.get_batch(name)`

The method is not that encoding sensitive anymore. It returns the depending `Batch`, no matter if `'...'`, `u'...'` or `'...'.decode('utf8')` is included as name.

Update: warning in weight engine

Missing codes in the sample are only alerted, if the belonging target is not 0.

Update: `DataSet.to_array(..., variables, ...)`

Duplicated vars in `variables` are not allowed anymore, these were causing problems in the ChainManager class.

Update: `Batch.add_open_ends()`

Method raises an error if no vars are included in `oe` and `break_by`. The empty dataframe was causing issues in the ChainManager class.

Update: `Batch.extend_x()`

The method automatically checks if the included variables are arrays and adds them to `Batch.summaries` if they are included yet.

2.4 sd (04/06/2018)

New: Additional variable (names) "getter"-like and resolver methods

- `DataSet.created()`
- `DataSet.find(str_tags=None, suffixed=False)`
- `DataSet.names()`
- `DataSet.resolve_name()`

A bunch of new methods enhancing the options of finding and testing for variable names have been added. `created()` will list all variables that have been added to a dataset using core functions, i.e. `add_meta()` and `derive()`, resp. all helper methods that use them internally (as `band()` or `categorize()` do for instance).

The `find()` method is returning all variable names that contain any of the provided substrings in `str_tags`. To only consider names that end with these strings, set `suffixed=True`. If no `str_tags` are passed, the method will use a default list of tags including `['_rc', '_net', ' (categories', ' (NET', '_rec']`.

Sometimes a dataset might contain "semi-duplicated" names, variables that differ in respect to case sensitivity but have otherwise identical names. Calling `names()` will report such cases in a `pd.DataFrame` that lists all name variants under the respective `str.lower()` version. If no semi-duplicates are found, `names()` will simply return `DataSet.variables()`.

Lastly, `resolve_name()` can be used to return the "proper", existing representation(s) of a given variable name's spelling.

New: `Batch.remove()`

Not needed batches can be removed from `meta`, so they are not aggregated anymore.

New: `Batch.rename(new_name)`

Sometimes standard batches have long/ complex names. They can now be changed into a custom name. Please take into account, that for most hubs the name of omnibus batches should look like 'client ~ topic'.

Update: Handling verbatims in `qp.Batch`

Instead of holding the well prepared open-end dataframe in `batch.verbatims`, the attribute is now filled by `batch.add_open_ends()` with instructions to create the open-end dataframe. It is easier to to modify/ overwrite existing verbatims. Therefore also a new parameter is included `overwrite=True`.

Update: `Batch.copy(..., b_filter=None, as_addition=False)`

It is now possible to define an additional filter for a copied batch and also to set it as addition to the master batch.

Update: Regrouping the variable list using `DataSet.order(..., regroup=True)`

A new parameter called `regroup` will instruct reordering all newly created variables into their logical position of the dataset's main variable order, i.e. attempting to place *derived* variables after the *originating* ones.

Bugfix: `add_meta()` and duplicated categorical `values` codes

Providing duplicated numerical codes while attempting to create new metadata using `add_meta()` will now correctly raise a `ValueError` to prevent corrupting the `DataSet`.

```
>>> cats = [(1, 'A'), (2, 'B'), (3, 'C'), (3, 'D'), (2, 'AA')]
>>> dataset.add_meta('test_var', 'single', 'test label', cats)
ValueError: Cannot resolve category definition due to code duplicates: [2, 3]
```

2.5 sd (04/04/2018)

New: Emptiness handlers in `DataSet` and `Batch` classes

- `DataSet.empty(name, condition=None)`
- `DataSet.empty_items(name, condition=None, by_name=True)`
- `DataSet.hide_empty_items(condition=None, arrays=None)`
- `Batch.hide_empty(xks=True, summaries=True)`

`empty()` is used to test if regular variables are completely empty, `empty_items()` checks the same for the items of an array mask definition. Both can be run on lists of variables. If a single variable is tested, the former returns simply boolean, the latter will list all empty items. If lists are checked, `empty()` returns the sublist of empty variables, `empty_items()` is mapping the list of empty items per array name. The `condition` parameter of these methods takes a `Quantipy logic` expression to restrict the test to a subset of the data, i.e. to check if variables will be empty if the dataset is filtered a certain way. A very simple example:

```
>>> dataset.add_meta('test_var', 'int', 'Variable is empty')
>>> dataset.empty('test_var')
True
```

```
>>> dataset[dataset.take({'gender': 1}), 'test_var'] = 1
>>> dataset.empty('test_var')
False
```

```
>>> dataset.empty('test_var', {'gender': 2})
True
```

The `DataSet` method `hide_empty_items()` uses the emptiness tests to automatically apply a **hiding rule** on all empty items found in the dataset. To restrict this to specific arrays only, their names can be provided via the `arrays` argument. `Batch.hide_empty()` takes into account the current `Batch.filter` setup and by drops/hides *all* relevant empty variables from the `xks` list and summary aggregations by default. Summaries that would end up without valid items because of this are automatically removed from the `summaries` collection and the user is warned.

New: `qp.set_option('fast_stack_filters', True)`

A new option to enable a more efficient test for already existing filters inside the `qp.Stack` object has been added. Set the `'fast_stack_filters'` option to `True` to use it, the default is `False` to ensure compatibility in different versions of production DP template workspaces.

Update: `Stack.add_stats(..., factor_labels=True, ...)`

The parameter `factor_labels` is now also able to take the string `'()'`, then factors are written in the normal brackets next to the label (instead of `[]`).

In the new version factor_labels are also just added if there are none included before, except new scales are used.

Bugfix: `DataSet` `np.NaN` insertion to `delimited_set` variables

`np.NaN` was incorrectly transformed when inserted into `delimited_set` before, leading to either `numpy` type conflicts or type casting exceptions. This is now fixed.

2.6 sd (27/02/2018)

New: `DataSet._dimensions_suffix`

`DataSet` has a new attribute `_dimensions_suffix`, which is used as mask suffix while running `DataSet.dimensionize()`. The default is `_grid` and it can be modified with `DataSet.set_dim_suffix()`.

Update: `Stack._get_chain()` (old chain)

The method is speeded-up. If a filter is already included in the Stack, it is not calculated from scratch anymore. Additionally the method has a new parameter `described`, which takes a describing dataframe of the Stack, so it no longer needs to be calculated in each loop.

Nets that are applied on array variables will now also create a new recoded array that reflects the net definitions if `recoded` is used. The method has been creating only the item versions before.

Update: `Stack.add_stats()`

The method will now create a new metadata property called `'factor'` for each variable it is applied on. You can only have one factor assigned to one categorical value, so for multiple statistic definitions (exclusions, etc.) it will get overwritten.

Update: `DataSet.from_batch()` (`additions` parameter)

The `additions` parameter has been updated to also be able to create recoded variables from existing "additional" Batches that are attached to a parent one. Filter variables will get the new meta `'properties'` tag `'recoded_filter'` and only have one category (`1`, `'active'`). They are named simply `'filter_1'`, `'filter_2'` and so on. The new possible values of the parameters are now:

- `None`: `as_addition()`-Batches are not considered.
- `'variables'`: Only cross- and downbreak variables are considered.
- `'filters'`: Only filters are recoded.
- `'full'`: `'variables'` + `'filters'`

Bugfix: `ViewManager._request_views()`

Cumulative sums are only requested if they are included in the belonging `Stack`. Additionally the correct related sig-tests are now taken for cumulative sums.

2.7 sd (12/01/2018)

New: `Audit`

`Audit` is a new class which takes `DataSet` instances, compares and aligns them.

The class compares/ reports/ aligns the following aspects:

- datasets are valid (`DataSet.validate()`)
- mismatches (variables are not included in all datasets)
- different types (variables are in more than one dataset, but have different types)
- labels (variables are in more than one dataset, but have different labels for the same text_key)
- value codes (variables are in more than one dataset, but have different value codes)
- value texts (variables are in more than one dataset, but have different value texts)
- array items (arrays are in more than one dataset, but have different items)
- item labels (arrays are in more than one dataset, but their items have different labels)

This is the first draft of the class, so it will need some testing and probably adjustments.

New: `DataSet.reorder_items(name, new_order)`

The new method reorders the items of the included array. The ints in the `new_order` list match up to the number of the items (`DataSet.item_no('item_name')`), not to the position.

New: `DataSet.valid_tks`, Arabic

Arabic (`ar-AR`) is included as default valid text-key.

New: `DataSet.extend_items(name, ext_items, text_key=None)`

The new method extends the items of an existing array.

Update: `DataSet.set_missings()`

The method is now limited to `DataSet`, `Batch` does not inherit it.

Update: `DataSet`

The whole class is reordered and cleaned up. Some new deprecation warnings will appear.

Update: `DataSet.add_meta()` / `DataSet.derive()`

Both methods will now raise a `ValueError: Duplicated codes provided. Value codes must be unique!` if categorical `values` definitions try to apply duplicated codes.

2.8 sd (18/12/2017)

New: `Batch.remove_filter()`

Removes all defined (global + extended) filters from a Batch instance.

Update: `Batch.add_filter()`

It's now possible to extend the global filter of a Batch instance. These options are possible.

Add first filter:

```
>>> batch.filter, batch.filter_names
'no_filter', ['no_filter']
>>> batch.add_filter('filter1', logic1)
>>> batch.filter, batch.filter_names
{'filter1': logic1}, ['filter1']
```

Extend filter:

```
>>> batch.filter, batch.filter_names
{'filter1': logic}, ['filter1']
>>> batch.add_filter('filter2', logic2)
>>> batch.filter, batch.filter_names
{'filter1' + 'filter2': intersection([logic1, logic2])}, ['filter1' + 'filter2']
```

Replace filter:

```
>>> batch.filter, batch.filter_names
{'filter1': logic}, ['filter1']
>>> batch.add_filter('filter1', logic2)
>>> batch.filter, batch.filter_names
{'filter1': logic2}, ['filter1']
```

Update: `Stack.add_stats(..., recode)`

The new parameter `recode` defines if a new numerical variable is created which satisfies the stat definitions.

Update: `DataSet.populate()`

A progress tracker is added to this method.

Bugfix: `Batch.add_open_ends()`

= is removed from all responsess in the included variables, as it causes errors in the Excel-Painter.

Bugfix: `Batch.extend_x()` and `Batch.extend_y()`

Check if included variables exist and unroll included masks.

Bugfix: `Stack.add_nets(..., calc)`

If the operator in calc is `div`/ `/`, the calculation is now performed correctly.

2.9 sd (28/11/2017)

New `DataSet.from_batch()`

Creates a new `DataSet` instance out of `Batch` definitions (xks, yks, filter, weight, language, additions, edits).

New: `Batch.add_total()`

Defines if total column `@` should be included in the downbreaks (yks).

New: `Batch.set_unwgt_counts()`

If cellitems are `cp` and a weight is provided, it is possible to request unweighted count views (percentages are still weighted).

Update: `Batch.add_y_on_y(name, y_filter=None, main_filter='extend')`

Multiple `y_on_y` aggregations can now be added to a `Batch` instance and each can have an own filter. The y_on_y-filter can `extend` or `replace` the main_filter of the `Batch`.

Update: `Stack.add_nets(..., recode)`

The new parameter `recode` defines if a new variable is created which satisfies the net definitions. Different options for `recode` are:

- `'extend_codes'`: The new variable contains all codes of the original variable and all nets as new categories.
- `'drop_codes'`: The new variable contains only all nets as new categories.
- `'collect_codes'` or `'collect_codes@cat_name'`: The new variable contains all nets as new categories and another new category which sums all cases that are not in any net. The new category text can be defined by adding `@cat_name` to `collect_codes`. If none is provided `Other` is used as default.

Update: `Stack.add_nets()`

If a variable in the `Stack` already has a net_view, it gets overwritten if a new net is added.

Update: `DataSet.set_missings(..., missing_map)`

The parameter `missing_map` can also handle lists now. All included codes are be flagged as `'exclude'`.

Update: `request_views(..., sums='mid')` (`ViewManager/query.py`)

Allow different positions for sums in the view-order. They can be placed in the middle (`'mid'`) between the basics/nets and the stats or at the `'bottom'` after the stats.

Update/ New: `write_dimensions()`

Converting qp data to mdd and ddf files by using `write_dimensions()` is updated now. A bug regarding encoding texts is fixed and additionally all included `text_keys` in the meta are transferred into the mdd. Therefore two new classes are included: `DimLabels` and `DimLabel`.

2.10 sd (13/11/2017)

New ‘‘DataSet.to_delimited_set(name, label, variables, from_dichotomous=True, codes_from_name=True)‘‘

Creates a new delimited set variable out of other variables. If the input- variables are dichotomous (`from_dichotomous`), the new value-codes can be taken from the variable-names or from the order of the variables (`codes_from_name`).

Update `Stack.aggregate(..., bases={})`

A dictionary in form of:

```
bases = {
   'cbase': {
      'wgt': True,
      'unwgt': False},
   'cbase_gross': {
      'wgt': True,
      'unwgt': True},
   'ebase': {
      'wgt': False,
      'unwgt': False}
      }
```

defines what kind of bases will be aggregated. If `bases` is provided the old parameter `unweighted_base` and any bases in the parameter `views` will be ignored. If bases is not provided and any base is included in `views`, a dictionary is automatically created out of `views` and `unweighted_base`.

2.11 sd (17/10/2017)

New: `del DataSet['var_name']` and `'var_name' in DataSet` syntax support

It is now possible to test membership of a variable name simply using the `in` operator instead of `DataSet.var_exists('var_name')` and delete a variable definition from `DataSet` using the `del` keyword inplace of the `drop('var_name')` method.

New: `DataSet.is_single(name)`, `.is_delimited_set(name)`, `.is_int(name)`, `.is_float(name)`, `.is_string(name)`, `.is_date(name)`, `.is_array(name)`

These new methods make testing a variable's type easy.

Update: `DataSet.singles(array_items=True)` and all other non-`array` type iterators

It is now possible to exclude `array` items from `singles()`, `delimited_sets()`, `ints()` and `floats()` variable lists by setting the new `array_items` parameter to `False`.

Update: `Batch.set_sigtests(..., flags=None, test_total=None)`, `Batch.sigproperties`

The significancetest-settings for flagging and testing against total, can now be modified by the two parameters `flags` and `test_total`. The `Batch` attribute `siglevels` is removed, instead all sig-settings are stored in `Batch.sigproperties`.

Update: `Batch.make_summaries(..., exclusive=False)`, `Batch.skip_items`

The new parameter `exclusive` can take a list of arrays or a boolean. If a list is included, these arrays are added to `Batch.skip_items`, if it is True all variables from `Batch.summaries` are added to `Batch.skip_items`

Update: `quantipy.sandbox.sandbox.Chain.paint(..., totalize=True)`

If `totalize` is `True`, `@`-Total columns of a (x-oriented) `Chain.dataframe` will be painted as `'Total'` instead of showing the corresponsing `x`-variables question text.

Update: `quantipy.core.weights.Rim.Rake`

The weighting algorithm's `generate_report()` method can be caught up in a `MemoryError` for complex weight schemes run on very large sample sizes. This is now prevented to ensure the weight factors are computed with priority and the algorithm is able to terminate correctly. A warning is raised:

```
UserWarning: OOM: Could not finish writing report...
```

Update: `Batch.replace_y()`

Conditional replacements of y-variables of a `Batch` will now always also automatically add the `@`-Total indicator if not provided.

Bugfix: `DataSet.force_texts(..., overwrite=True)`

Forced overwriting of existing `text_key` meta data was failing for `array` `mask` objects. This is now solved.

2.12 sd (15/09/2017)

New: `DataSet.meta_to_json(key=None, collection=None)`

The new method allows saving parts of the metadata as a json file. The parameters `key` and `collection` define the metaobject which will be saved.

New: `DataSet.save()` and `DataSet.revert()`

These two new methods are useful in interactive sessions like **Ipython** or **Jupyter** notebooks. `save()` will make a temporary (only im memory, not written to disk) copy of the `DataSet` and store its current state. You can then use `revert()` to rollback to that snapshot of the data at a later stage (e.g. a complex recode operation went wrong, reloading from the physical files takes too long...).

New: `DataSet.by_type(types=None)`

The `by_type()` method is replacing the soon to be deprecated implementation of `variables()` (see below). It provides the same functionality (`pd.DataFrame` summary of variable types) as the latter.

Update: `DataSet.variables()` absorbs `list_variables()` and `variables_from_set()`

In conjunction with the addition of `by_type()`, `variables()` is replacing the related `list_variables()` and `variables_from_set()` methods in order to offer a unified solution for querying the `DataSet`'s (main) variable collection.

Update: `Batch.as_addition()`

The possibility to add multiple cell item iterations of one `Batch` definition via that method has been reintroduced (it was working by accident in previous versions with subtle side effects and then removed). Have fun!

Update: `Batch.add_open_ends()`

The method will now raise an `Exception` if called on a `Batch` that has been added to a parent one via `as_addition()` to warn the user and prevent errors at the build stage:

```
NotImplementedError: Cannot add open end DataFrames to as_addition()-Batches!
```

2.13 sd (31/08/2017)

New: `DataSet.code_from_label(..., exact=True)`

The new parameter `exact` is implemented. If `exact=True` codes are returned whose belonging label is equal the included `text_label`. Otherwise the method checks if the labels contain the included `text_label`.

New: `DataSet.order(new_order=None, reposition=None)`

This new method can be used to change the global order of the `DataSet` variables. You can either pass a complete `new_order` list of variable names to set the order or provide a list of dictionaries to move (multiple) variables before a reference variable name. The order is reflected in the case data `pd.DataFrame.columns` order and the meta `'data file' set` object's items.

New: `DataSet.dichotomize(name, value_texts=None, keep_variable_text=True, ignore=None, replace=False, text_key=None)`

Use this to convert a `'delimited set'` variable into a set of binary coded `'single'` variables. Variables will have the values 1/0 and by default use `'Yes'` / `'No'` as the corresponding labels. Use the `value_texts` parameter to apply custom labels.

New: `Batch.extend_x(ext_xks)`

The new method enables an easy extension of `Batch.xks`. In `ext_xks` included `str` are added at the end of `Batch.xks`. Values of included `dicts` are positioned in front of the related key.

Update: `Batch.extend_y(ext_yks, ...)`

The parameter `ext_yks` now also takes `dicts`, which define the position of the additional `yks`.

Update: `Batch.add_open_ends(..., replacements)`

The new parameter `replacements` is implemented. The method loops over the whole pd.DataFrame and replaces all keys of the included `dict` with the belonging value.

Update: `Stack.add_stats(..., other_source)`

Statistic views can now be added to delimited sets if `other_source` is used. In this case `other_source` must be a single or numerical variable.

Update: `DataSet.validate(..., spss_limits=False)`

The new parameter `spss_limits` is implemented. If `spss_limits=True`, the validate output dataframe is extended by 3 columns which show if the SPSS label limitations are satisfied.

Bugfix: `DataSet.convert()`

A bug that prevented conversions from `single` to numeric types has been fixed.

Bugfix: `DataSet.add_meta()`

A bug that prevented the creation of numerical arrays outside of `to.array()` has been fixed. It is now possible to create `array` metadata without providing category references.

Bugfix: `Stack.add_stats()`

Checking the statistic views is skipped now if no single typed variables are included even if a checking cluster is provided.

Bugfix: `Batch.copy()`

Instead of using a deepcopy of the `Batch` instance, a new instance is created and filled with the attributes of the initial one. Then the copied instance can be used as additional `Batch`.

Bugfix: `qp.core.builds.powerpoint`

Access to bar-chart series and colour-filling is now working for different Powerpoint versions. Also a bug is fixed which came up in `PowerPointpainter()` for variables which have fixed categories and whose values are located in `lib`.

2.14 sd (24/07/2017)

New: `qp.set_option()`

It is now possible to set library-wide settings registered in `qp.OPTIONS` by providing the setting's name (key) and the desired value. Currently supported are:

```
OPTIONS = {
        'new_rules': False,
        'new_chains': False,
        'short_item_texts': False
}
```

So for example, to work with the currently refactored `Chain` interim class we can use `qp.set_options('new_chains', True)`.

New: `qp.Batch()`

This is a new object aimed at defining and structuring aggregation and build setups. Please see an *extensive overview here*.

New: `Stack.aggregate()` / `add_nets()` / `add_stats()` / `add_tests()` / ...

Connected to the new `Batch` class, some new `Stack` methods to ease up view creation have been added. You can *find the docs here*.

New: `DataSet.populate()`

Use this to create a `qp.Stack` from `Batch` definitions. This connects the `Batch` and `Stack` objects; check out the *Batch* and *Analysis & aggregation* docs.

New: `DataSet.write_dimensions(path_mdd=None, path_ddf=None, text_key=None, mdm_lang='ENG', run=True, clean_up=True)`

It is now possible to directly convert a `DataSet` into a Dimensions .ddf/.mdd file pair (given SPSS Data Collection Base Professional is installed on your machine). By default, files will be saved to the same location in that the `DataSet` resides and keep its `text_key`.

New: `DataSet.repair()`

This new method can be used to try to fix common `DataSet` metadata problems stemming from outdated versions, incorrect manual editing of the meta dictionary or other inconsistencies. The method is checking and repairing following issues:

- `'name'` is present for all variable metadata
- `'source'` and `'subtype'` references for array variables
- correct `'lib'`-based `'values'` object for array variables
- `text key`-dependent `'x edits'` / `'y edits'` meta data
- `['data file']['items']` set entries exist in `'columns'` / `'masks'`

New: `DataSet.subset(variables=None, from_set=None, inplace=False)`

As a counterpart to `filter()`, `subset()` can be used to create a new `DataSet` that contains only a selection of variables. The new variables collection can be provided either as a list of names or by naming an already existing set containing the desired variables.

New: `DataSet.variables_from_set(setname)`

Get the list of variables belonging to the passed set indicated by `setname`.

New: `DataSet.is_like_numeric(name)`

A new method to test if all of a `string` variable's values can be converted to a numerical (`int` / `float`) type. Returns a boolean `True` / `False`.

Update: `DataSet.convert()`

It is now possible to convert inplace from `string` to `int` / `float` if the respective internal `is_like_numeric()` check identifies numeric-like values.

Update: `DataSet.from_components(..., reset=True)`, `DataSet.read_quantipy(..., reset=True)`

Loaded `.json` metadata dictionaries will get cleaned now by default from any user-defined, non-native objects inside the `'lib'` and `'sets'` collections. Set `reset=False` to keep any extra entires (restoring the old behaviour).

Update: `DataSet.from_components(data_df, meta_dict=None, ...)`

It is now possible to create a `DataSet` instance by providing a `pd.DataFrame` alone, without any accompanying meta data. While reading in the case data, the meta component will be created by inferring the proper `Quantipy` variable types from the `pandas dtype` information.

Update: `Quantity.swap(var, ..., update_axis_def=True)`

It is now possible to `swap()` the `'x'` variable of an array based `Quantity`, as long as the length oh the constructing `'items'` collection is identical. In addition, the new parameter `update_axis_def` is now by default enforcing an update of the axis defintions (`pd.DataFrame` column names, etc) while previously the method was keeping the original index and column names. The old behaviour can be restored by setting the parameter to `False`.

Array example:

```
>>> link = stack[name_data]['no_filter']['q5']['@']
>>> q = qp.Quantity(link)
>>> q.summarize()
Array                    q5
Questions              q5_1         q5_2         q5_3         q5_4         q5_5      ⌐
↪   q5_6
Question Values
q5       All     8255.000000  8255.000000  8255.000000  8255.000000  8255.000000 ⌐
↪8255.000000
         mean      26.410297    22.260569    25.181466    39.842883    24.399758   ⌐
↪28.972017
         stddev    40.415559    38.060583    40.018463    46.012205    40.537497   ⌐
↪41.903322
         min        1.000000     1.000000     1.000000     1.000000     1.000000   ⌐
↪1.000000
         25%        3.000000     3.000000     3.000000     3.000000     1.000000   ⌐
↪3.000000
```

(continues on next page)

(continued from previous page)

```
         median     5.000000     3.000000     3.000000     5.000000     3.000000
↪5.000000
         75%        5.000000     5.000000     5.000000    98.000000     5.000000
↪97.000000
         max       98.000000    98.000000    98.000000    98.000000    98.000000
↪98.000000
```

Updated axis definiton:

```
>>> q.swap('q7', update_axis_def=True)
>>> q.summarize()
Array                      q7
Questions                 q7_1         q7_2         q7_3       q7_4       q7_5       q7_
↪6
Question Values
q7       All     1195.000000  1413.000000  3378.000000  35.000000  43.000000  36.
↪000000
         mean       5.782427     5.423213     5.795145   4.228571   4.558140   5.
↪333333
         stddev     2.277894     2.157226     2.366247   2.073442   2.322789   2.
↪552310
         min        1.000000     1.000000     1.000000   1.000000   1.000000   1.
↪000000
         25%        4.000000     4.000000     4.000000   3.000000   3.000000   3.
↪000000
         median     6.000000     6.000000     6.000000   4.000000   4.000000   6.
↪000000
         75%        8.000000     7.000000     8.000000   6.000000   6.000000   7.
↪750000
         max        9.000000     9.000000     9.000000   8.000000   9.000000   9.
↪000000
```

Original axis definiton:

```
>>> q = qp.Quantity(link)
>>> q.swap('q7', update_axis_def=False)
>>> q.summarize()
Array                      q5
Questions                 q5_1         q5_2         q5_3       q5_4       q5_5       q5_
↪6
Question Values
q5       All     1195.000000  1413.000000  3378.000000  35.000000  43.000000  36.
↪000000
         mean       5.782427     5.423213     5.795145   4.228571   4.558140   5.
↪333333
         stddev     2.277894     2.157226     2.366247   2.073442   2.322789   2.
↪552310
         min        1.000000     1.000000     1.000000   1.000000   1.000000   1.
↪000000
         25%        4.000000     4.000000     4.000000   3.000000   3.000000   3.
↪000000
         median     6.000000     6.000000     6.000000   4.000000   4.000000   6.
↪000000
         75%        8.000000     7.000000     8.000000   6.000000   6.000000   7.
↪750000
         max        9.000000     9.000000     9.000000   8.000000   9.000000   9.
↪000000
```

Update: `DataSet.merge_texts()`

The method will now always overwrite existing `text_key` meta, which makes it possible to merge `texts` from meta of the same `text_key` as the master `DataSet`.

Bugfix: `DataSet.band()`

`band(new_name=None)`'s automatic name generation was incorrectly creating new variables with the name `None_banded`. This is now fixed.

Bugfix: `DataSet.copy()`

The method will now check if the name of the copy already exists in the `DataSet` and drop the referenced variable if found to prevent inconsistencies. Additionally, it is not longer possible to copy isolated `array` items:

```
>>> dataset.copy('q5_1')
NotImplementedError: Cannot make isolated copy of array item 'q5_1'. Please copy␣
↪array variable 'q5' instead!
```

2.15 sd (08/06/2017)

New: `DataSet.extend_valid_tks()`, `DataSet.valid_tks`

`DataSet` has a new attribute `valid_tks` that contains a list of all valid textkeys. All methods that take a textkey as parameter are checked against that list.

If a datafile contains a special/ unusual textkey (for example `'id-ID'` or `'zh-TW'`), the list can be extended with `DataSet.extend_valid_tks()`. This extension can also be used to create a textkey for special conditions, for example to create texts only for powerpoint outputs:

```
>>> dataset.extend_valid_tks('pptx')
>>> dataset.force_texts('pptx', 'en-GB')
>>> dataset.set_variable_text('gender','Gender label for pptx', text_key='pptx')
```

New: Equal error messages

All methods that use the parameters `name`/`var`, `text_key` or `axis_edit`/ `axis` now have a decorator that checks the provided values. The following shows a few examples for the new error messages:

`name` & `var`:

```
'name' argument for meta() must be in ['columns', 'masks'].
q1 is not in ['columns', 'masks'].
```

`text_key`:

```
'en-gb' is not a valid text_key! Supported are: ['en-GB', 'da-DK', 'fi-FI', 'nb-NO',
↪'sv-SE', 'de-DE']
```

`axis_edit` & `axis`:

```
'xs' is not a valid axis! Supported are: ['x', 'y']
```

New: `DataSet.repair_text_edits(text_key)`

This new method can be used in trackers, that were drawn up in an older `Quantipy` version. Text objects can be repaired if are not well prepared, for example if it looks like this:

```
{'en-GB': 'some English text',
 'sv_SE': 'some Swedish text',
 'x edits': 'new text'}
```

`DataSet.repair_text_edits()` loops over all text objects in the dataset and matches the `x edits` and `y edits` texts to all included textkeys:

```
>>> dataset.repair_text_edits(['en-GB', 'sv-SE'])
{'en-GB': 'some English text',
 'sv_SE': 'some Swedish text',
 'x edits': {'en-GB': new text', 'sv-SE': 'new text'}}
```

Update: `DataSet.meta()/.text()/.values()/.value_texts()/.items()/.item_texts()`

All these methods now can take the parameters `text_key` and `axis_edit`. The related text is taken from the meta information and shown in the output. If a text key or axis edit is not included the text is returned as None.

Update: `DataSet.compare(dataset, variables=None, strict=False, text_key=None)`

The method is totally updated, works more precise and contains a few new features. Generally variables included in `dataset` are compared with eponymous variables in the main `DataSet` instance. You can specify witch `variables` should be compared, if question/ value texts should be compared `strict` or not and for which `text_key`.

Update: `DataSet.validate(verbose=True)`

A few new features are tested now and the output has changed. Set `verbose=True` to see the definitions of the different error columns:

```
name: column/mask name and meta[collection][var]['name'] are not identical

q_label: text object is badly formated or has empty text mapping

values: categorical var does not contain values, value text is badly
formated or has empty text mapping

textkeys: dataset.text_key is not included or existing tks are not
consistent (also for parents)

source: parents or items do not exist

codes: codes in .data are not included in .meta
```

Update: `DataSet.sorting()` / `.slicing()` / `.hiding()`

These methods will now also work on lists of variable names.

Update: `DataSet.set_variable_text()`, `Dataset.set_item_texts()`

If these methods are applied to an array item, the new variable text is also included in the meta information of the parent array. The same works also the other way around, if an array text is set, then the array item texts are modified.

Update: `DataSet.__init__(self, name, dimensions_comp=True)`

A few new features are included to handle data coming from Crunch. While initializing a new `DataSet` instance dimensions compatibility can be set to False. In the custom template use `t.get_qp_dataset(name, dim_comp=False)` in the load cells.

Bugfix: `DataSet.hmerge()`

If `right_on` and `left_on` are used and `right_on` is also included in the main file, it is not overwritten any more.

2.16 sd (17/05/2017)

Update: `DataSet.set_variable_text(..., axis_edit=None)`, `DataSet.set_value_texts(.., axis_edit=False)`

The new `axis_edit` argument can be used with one of `'x'`, `'y'` or `['x', 'y']` to instruct a text metadata change that will only be visible in build exports.

> **Warning:** In a future version `set_col_text_edit()` and `set_val_text_text()` will be removed! The identical functionality is provided via this `axis_edit` parameter.

Update: `DataSet.replace_texts(..., text_key=None)`

The method loops over all meta text objects and replaces unwanted strings. It is now possible to perform the replacement only for specified `text_keys`. If `text_key=None` the method replaces the strings for all `text_keys`.

Update: `DataSet.force_texts(copy_to=None, copy_from=None, update_existing=False)`

The method is now only able to force texts for all meta text objects (for single variables use the methods `set_variable_text()` and `set_value_texts()`).

Bugfix: `DataSet.copy()`

Copied variables get the tag `created` and can be listed with `t.list_variables(dataset, 'created')`.

Bugfix: `DataSet.hmerge()`, `DataSet.vmerge()`

Array meta information in merged datafiles is now updated correctly.

2.17 sd (04/05/2017)

New: `DataSet.var_exists()`

Returns True if the input variable/ list of variables are included in the `DataSet` instance, otherwise False.

New: `DataSet.remove_html()`, `DataSet.replace_texts(replace)`

The `DataSet` method `clean_texts()` has been removed and split into two methods to make usage more clear: `remove_html()` will strip all `text` metadata objects from any html and formatting tags. `replace_texts()` will use a `dict` mapping of old to new `str` terms to change the matching `text` throughout the `DataSet` metadata.

New: `DataSet.item_no(name)`

This method will return the positional index number of an array item, e.g.:

```
>>> dataset.item_no('Q4A[{q4a_1}].Q4A_grid')
1
```

New: `QuantipyViews`: `counts_cumsum`, `c%_cumsum`

These two new views contain frequencies with cumulative sums which are computed over the x-axis.

Update: `DataSet.text(name, shorten=True)`

The new parameter `shorten` is now controlling if the variable `text` metadata of array masks will be reported in short format, i.e. without the corresponding mask label text. This is now also the default behaviour.

Update: `DataSet.to_array()`

Created mask meta information now also contains keys `parent` and `subtype`. Variable names are compatible with crunch and dimensions meta:

Example in Dimensions modus:

```
>>> dataset.to_array('Q11', ['Q1', 'Q2', 'Q3', 'Q4', 'Q5'], 'label')
```

The new grid is named `'Q11.Q11_grid'` and the source/column variables are `'Q11[{Q1}].-Q11_grid'` - `'Q11[{Q5}].-Q11_grid'`.

Bugfix: `DataSet.derotate()`

Meta is now Crunch and Dimensions compatible. Also mask meta information are updated.

2.18 sd (24/04/2017)

Update: `DataSet.hiding(..., hide_values=True)`

The new parameter `hide_values` is only necessary if the input variable is a mask. If `False`, mask items are hidden, if `True` mask values are hidden for all mask items and for array summary sheets.

Bugfix: `DataSet.set_col_text_edit(name)`

If the input variable is an array item, the new column text is also added to `meta['mask'][name]['items]`.

Bugfix: `DataSet.drop(name, ignore_items=False)`

If a mask is dropped, but the items are kept, all items are handled now as individual variables and their meta information is not stored in `meta['lib']` anymore.

2.19 sd (06/04/2017)

Only small adjustments.

2.20 sd (29/03/2017)

New: `DataSet.codes_in_data(name)`

This method returns a list of codes that exist in the data of a variable. This information can be used for more complex recodes, for example copying a variable, but keeping only all categories with more than 50 ratings, e.g.:

```
>>> valid_code = dataset.codes_in_data('varname')
>>> keep_code = [x for x in valid_code if dataset['varname'].value_counts()[x] > 49]
>>> dataset.copy('varname', 'rc', copy_only=keep_code)
```

Update: `DataSet.copy(..., copy_not=None)`

The new parameter `copy_not` takes a list of codes that should be ignored for the copied version of the provided variable. The metadata of the copy will be reduced as well.

Update: `DataSet.code_count()`

This method is now alligned with `any()` and `all()` in that it can be used on `'array'` variables as well. In such a case, the resulting `pandas.Series` is reporting the number of answer codes found across all items per case data row, i.e.:

```
>>> code_count = dataset.code_count('Q4A.Q4A_grid', count_only=[3, 4])
>>> check = pd.concat([dataset['Q4A.Q4A_grid'], code_count], axis=1)
>>> check.head(10)
   Q4A[{q4a_1}].Q4A_grid  Q4A[{q4a_2}].Q4A_grid  Q4A[{q4a_3}].Q4A_grid  0
0                    3.0                    3.0                    NaN  2
1                    NaN                    NaN                    NaN  0
```

(continues on next page)

(continued from previous page)

```
2                3.0                3.0                4.0  3
3                5.0                4.0                2.0  1
4                4.0                4.0                4.0  3
5                4.0                5.0                4.0  2
6                3.0                3.0                3.0  3
7                4.0                4.0                4.0  3
8                6.0                6.0                6.0  0
9                4.0                5.0                5.0  1
```

2.21 sd (20/03/2017)

New: `qp.DataSet(dimensions_comp=True)`

The `DataSet` class can now be explicitly run in a Dimensions compatibility mode to control the naming conventions of `array` variables ("grids"). This is also the default behaviour for now. This comes with a few changes related to meta creation and variable access using `DataSet` methods. Please see a brief case study on this topic *here*.

New: enriched `items` / `masks` meta data

`masks` will now also store the `subtype` (`single`, `delimited set`, etc.) while `items` elements will now contain a reference to the defining `masks` entrie(s) in a new `parent` object.

Update: `DataSet.weight(..., subset=None)`

Filters the dataset by giving a Quantipy complex logic expression and weights only the remaining subset.

Update: Defining categorical `values` meta and `array` items

Both `values` and `items` can now be created in three different ways when working with the `DataSet` methods `add_meta()`, `extend_values()` and `derive()`: (1) Tuples that map element code to label, (2) only labels or (3) only element codes. Please see quick guide on that *here*

2.22 sd (07/03/2017)

Update: `DataSet.code_count(..., count_not=None)`

The new parameter `count_not` can be used to restrict the set of codes feeding into the resulting `pd.Series` by exclusion (while `count_only` restricts by inclusion).

Update: `DataSet.copy(..., copy_only=None)`

The new parameter `copy_only` takes a list of codes that should be included for the copied version of the provided variable, all others will be ignored and the metadata of the copy will be reduced as well.

Bugfix: `DataSet.band()`

There was a bug that was causing the method to crash for negative values. It is now possible to create negative single value bands, while negative ranges (lower and/or upper bound < 0) will raise a `ValueError`.

2.23 sd (24/02/2017)

- Some minor bugfixes and updates. Please use latest version.

2.24 sd (16/02/2017)

New: `DataSet.derotate(levels, mapper, other=None, unique_key='identity', dropna=True)`

Create a derotated ("levelled", responses-to-cases) `DataSet` instance by defining level variables, looped variables and other (simple) variables that should be added.

View more information on the topic *here*.

New: `DataSet.to_array(name, variables, label)`

Combine `column` variables with identical `values` objects to an `array` incl. all required `meta['masks']` information.

Update: `DataSet.interlock(..., variables)`

It is now possible to add `dicts` to `variables`. In these `dicts` a `derive()`-like mapper can be included which will then create a temporary variable for the interlocked result. Example:

```
>>> variables = ['gender',
...              {'agegrp': [(1, '18-34', {'age': frange('18-34')}),
...                          (2, '35-54', {'age': frange('35-54')}),
...                          (3, '55+', {'age': is_ge(55)})]},
...              'region']
>>> dataset.interlock('new_var', 'label', variables)
```

2.25 sd (04/01/2017)

New: `DataSet.flatten(name, codes, new_name=None, text_key=None)`

Creates a new `delimited set` variable that groups `grid item` answers to categories. The `items` become `values` of the new variable. If an `item` contains one of the `codes` it will be counted towards the categorical case data of the new variable.

New: `DataSet.uncode(target, mapper, default=None, intersect=None, inplace=True)`

Remove codes from the `target` variable's data component if a logical condition is satisfied.

New: `DataSet.text(var, text_key=None)`

Returns the question text label (per `text_key`) of a variable.

New: `DataSet.unroll(varlist, keep=None, both=None)`

Replaces `masks` names inside `varlist` with their `items`. Optionally, individual `masks` can be excluded or kept inside the list.

New: `DataSet.from_stack(stack, datakey=None)`

Create a `quantipy.DataSet` from the `meta`, `data`, `data_key` and `filter` definition of a `quantipy.Stack` instance.

2.26 sd (8/12/2016)

New:

`DataSet.from_excel(path_xlsx, merge=True, unique_key='identity')`

Returns a new `DataSet` instance with `data` from `excel`. The `meta` for all variables contains `type='int'`.

Example: `new_ds = dataset.from_excel(path, True, 'identity')`

The function is able to modify `dataset` inplace by merging `new_ds` on `identity`.

Update:

`DataSet.copy(..., slicer=None)`

It is now possible to filter the data that statisfies the logical condition provided in the `slicer`. Example:

```
>>> dataset.copy('q1', 'rec', True, {'q1': not_any([99])})
```

2.27 sd (23/11/2016)

Update:

`DataSet.rename(name, new_name=None, array_item=None)`

The function is able to rename `columns`, `masks` or `mask items`. `maks items` are changed by position.

Update:

`DataSet.categorize(..., categorized_name=None)`

Provide a custom name string for `categorized_name` will change the default name of the categorized variable from `OLD_NAME#` to the passed string.

2.28 sd (16/11/2016)

New:

`DataSet.check_dupe(name='identity')`

Returns a list with duplicated values for the variable provided via `name`. Identifies for example duplicated identities.

New:

`DataSet.start_meta(text_key=None)`

Creates an empty QP meta data document blueprint to add variable definitions to.

Update:

```
DataSet.create_set(setname='new_set', based_on='data file', included=None,
...                excluded=None, strings='keep', arrays='both', replace=None,
...                overwrite=False)
```

Add a new `set` to the `meta['sets']` object. Variables from an existing `set` (`based_on`) can be `included` to `new_set` or varibles can be `excluded` from `based_on` with customized lists of variables. Control `string` variables and `masks` with the `kwargs strings` and `arrays`. `replace` single variables in `new_set` with a `dict`.

Update:

`DataSet.from_components(..., text_key=None)`

Will now accept a `text_key` in the method call. If querying a `text_key` from the meta component fails, the method will no longer crash, but raise a `warning` and set the `text_key` to `None`.

Update:

```
DataSet.as_float()
DataSet.as_int()
DataSet.as_single()
DataSet.as_delimited_set()
DataSet.as_string()
DataSet.band_numerical()
DataSet.derive_categorical()
DataSet.set_mask_text()
DataSet.set_column_text()
```

These methods will now print a `UserWarning` to prepare for the soon to come removal of them.

Bugfix:

`DataSet.__setitem__()`

Trying to set `np.NaN` was failing the test against meta data for categorical variables and was raising a `ValueError` then. This is fixed now.

2.29 sd (11/11/2016)

New:

```
DataSet.columns
DataSet.masks
DataSet.sets
DataSet.singles
DataSet.delimited_sets
DataSet.ints
DataSet.floats
DataSet.dates
DataSet.strings
```

New `DataSet` instance attributes to quickly return the list of `columns`, `masks` and `sets` objects from the meta or query the variables by `type`. Use this to check for variables, iteration, inspection, ect.

New:

`DataSet.categorize(name)`

Create a categorized version of `int/string/date` variables. New variables will be named as per `OLD_NAME#`

New:

`DataSet.convert(name, to)`

Wraps the individual `as_TYPE()` conversion methods. `to` must be one of `'int', 'float', 'string', 'single', 'delimited set'`.

New:

`DataSet.as_string(name)`

Only for completeness: Use `DataSet.convert(name, to='string')` instead.

Converts `int/float/single/date` typed variables into a `string` and removes all categorical metadata.

Update:

`DataSet.add_meta()`

Can now add `date` and `text` type meta data.

Bugfix:

`DataSet.vmerge()`

If `masks` in the right `dataset`, that also exist in the left `dataset`, have new `items` or `values`, they are added to `meta['masks']`, `meta['lib']` and `meta['sets']`.

2.30 sd (09/11/2016)

New:

`DataSet.as_float(name)`

Converts `int/single` typed variables into a `float` and removes all categorical metadata.

New:

`DataSet.as_int(name)`

Converts `single` typed variables into a `int` and removes all categorical metadata.

New:

`DataSet.as_single(name)`

Converts `int` typed variables into a `single` and adds numeric values as categorical metadata.

New:

`DataSet.create_set(name, variables, blacklist=None)`

Adds a new `set` to `meta['sets']` object. Create easily `sets` from other `sets` while using customised `blacklist`.

New:

`DataSet.drop(name, ignore_items=False)`

Removes all metadata and data referenced to the variable. When passing an `array mask`, `ignore_items` can be ste to `True` to keep the `item columns` incl. their metadata.

New:

`DataSet.compare(dataset=None, variables=None)`

Compare the metadata definition between the current and another `dataset`, optionally restricting to a pair of variables.

Update:

`DataSet.__setitem__()`

`[..]`-Indexer now checks scalars against categorical meta.

CHAPTER 3

How-to-snippets

3.1 `DataSet` Dimensions compatibility

DTO-downloaded and Dimensions converted variable naming conventions are following specific rules for `array` names and corresponding `items`. `DataSet` offers a compatibility mode for Dimensions scenarios and handles the proper renaming automatically. Here is what you should know...

3.1.1 The compatibility mode

A `DataSet` will (by default) support Dimensions-like `array` naming for its connected data files when constructed. An `array` `masks` meta defintition of a variable called `q5` looking like this...:

```
{u'items': [{u'source': u'columns@q5_1', u'text': {u'en-GB': u'Surfing'}},
  {u'source': u'columns@q5_2', u'text': {u'en-GB': u'Snowboarding'}},
  {u'source': u'columns@q5_3', u'text': {u'en-GB': u'Kite boarding'}},
  {u'source': u'columns@q5_4', u'text': {u'en-GB': u'Parachuting'}},
  {u'source': u'columns@q5_5', u'text': {u'en-GB': u'Cave diving'}},
  {u'source': u'columns@q5_6', u'text': {u'en-GB': u'Windsurfing'}}],
 u'subtype': u'single',
 u'text': {u'en-GB': u'How likely are you to do each of the following in the next␣
↪year?'},
 u'type': u'array',
 u'values': u'lib@values@q5'}
```

...will be converted into its "Dimensions equivalent" as per:

```
>>> dataset = qp.DataSet(name_data, dimensions_comp=True)
>>> dataset.read_quantipy(path_data+name_data, path_data+name_data)
DataSet: ../Data/Quantipy/Example Data (A)
rows: 8255 - columns: 75
Dimensions compatibilty mode: True
```

```
>>> dataset.masks()
['q5.q5_grid', 'q6.q6_grid', 'q7.q7_grid']
```

```
>>> dataset._meta['masks']['q5.q5_grid']
{u'items': [{u'source': 'columns@q5[{q5_1}].q5_grid',
   u'text': {u'en-GB': u'Surfing'}},
  {u'source': 'columns@q5[{q5_2}].q5_grid',
   u'text': {u'en-GB': u'Snowboarding'}},
  {u'source': 'columns@q5[{q5_3}].q5_grid',
   u'text': {u'en-GB': u'Kite boarding'}},
  {u'source': 'columns@q5[{q5_4}].q5_grid',
   u'text': {u'en-GB': u'Parachuting'}},
  {u'source': 'columns@q5[{q5_5}].q5_grid',
   u'text': {u'en-GB': u'Cave diving'}},
  {u'source': 'columns@q5[{q5_6}].q5_grid',
   u'text': {u'en-GB': u'Windsurfing'}}],
 'name': 'q5.q5_grid',
 u'subtype': u'single',
 u'text': {u'en-GB': u'How likely are you to do each of the following in the next␣
↪year?'},
 u'type': u'array',
 u'values': 'lib@values@q5.q5_grid'}
```

3.1.2 Accessing and creating `array` data

Since new names are converted automatically by `DataSet` methods, there is no need to write down the full (DTO-like) Dimensions `array` name when adding new metadata. However, querying variables is always requiring the proper name:

```
>>> name, qtype, label = 'array_var', 'single', 'ARRAY LABEL'
>>> cats =  ['A', 'B', 'C']
>>> items = ['1', '2', '3']
>>> dataset.add_meta(name, qtype, label, cats, items)
```

```
>>> dataset.masks()
['q5.q5_grid', 'array_var.array_var_grid', 'q6.q6_grid', 'q7.q7_grid']
```

```
>>> dataset.meta('array_var.array_var_grid')
single                                                                  items item␣
↪texts  codes texts missing
array_var.array_var_grid: ARRAY LABEL
1                                    array_var[{array_var_1}].array_var_grid       ␣
↪  1      1     A    None
2                                    array_var[{array_var_2}].array_var_grid       ␣
↪  2      2     B    None
3                                    array_var[{array_var_3}].array_var_grid       ␣
↪  3      3     C    None
```

```
>>> dataset['array_var.array_var_grid'].head(5)
   array_var[{array_var_1}].array_var_grid  array_var[{array_var_2}].array_var_grid ␣
↪array_var[{array_var_3}].array_var_grid
0                                      NaN                                      NaN ␣
↪                                    NaN
1                                      NaN                                      NaN ␣
↪                                    NaN
```

(continues on next page)

(continued from previous page)

```
2                                    NaN                                    NaN ␣
↪                                  NaN
3                                    NaN                                    NaN ␣
↪                                  NaN
4                                    NaN                                    NaN ␣
↪                                  NaN
```

As can been seen above, both the `masks` name as well as the `array` item elements are being properly converted to match DTO/Dimensions conventions.

When using `rename()`, `copy()` or `transpose()`, the same behaviour applies:

```
>>> dataset.rename('q6.q6_grid', 'q6new')
>>> dataset.masks()
['q5.q5_grid', 'array_var.array_var_grid', 'q6new.q6new_grid', 'q7.q7_grid']
```

```
>>> dataset.copy('q6new.q6new_grid', suffix='q6copy')
>>> dataset.masks()
['q5.q5_grid', 'q6new_q6copy.q6new_q6copy_grid', 'array_var.array_var_grid', 'q6new.
↪q6new_grid', 'q7.q7_grid']
```

```
>>> dataset.transpose('q6new_q6copy.q6new_q6copy_grid')
>>> dataset.masks()
['q5.q5_grid', 'q6new_q6copy_trans.q6new_q6copy_trans_grid', 'q6new_q6copy.q6new_
↪q6copy_grid', 'array_var.array_var_grid', 'q6new.q6new_grid', 'q7.q7_grid']
```

3.2 Different ways of creating categorical values

The `DataSet` methods `add_meta()`, `extend_values()` and `derive()` offer three alternatives for specifying the categorical values of `'single'` and `'delimited set'` typed variables. The approaches differ with respect to how the mapping of numerical value codes to value text labels is handled.

(1) Providing a list of text labels

By providing the category labels only as a list of `str`, `DataSet` is going to create the numerical codes by simple enumeration:

```
>>> name, qtype, label = 'test_var', 'single', 'The test variable label'
```

```
>>> cats = ['test_cat_1', 'test_cat_2', 'test_cat_3']
>>> dataset.add_meta(name, qtype, label, cats)
```

```
>>> dataset.meta('test_var')
single                              codes       texts missing
test_var: The test variable label
1                                       1  test_cat_1    None
2                                       2  test_cat_2    None
3                                       3  test_cat_3    None
```

(2) Providing a list of numerical codes

If only the desired numerical codes are provided, the label information for all categories consequently will appear blank. In such a case the user will, however, get reminded to add the `'text'` meta in a separate step:

```
>>> cats = [1, 2, 98]
>>> dataset.add_meta(name, qtype, label, cats)
...\\quantipy\core\dataset.py:1287: UserWarning: 'text' label information missing,
only numerical codes created for the values object. Remember to add value 'text'␣
↪metadata manually!
```

```
>>> dataset.meta('test_var')
single                              codes texts missing
test_var: The test variable label
1                                       1          None
2                                       2          None
3                                      98          None
```

(3) Pairing numerical codes with text labels

To explicitly assign codes to corresponding labels, categories can also be defined as a list of tuples of codes and labels:

```
>>> cats = [(1, 'test_cat_1') (2, 'test_cat_2'), (98, 'Don\'t know')]
>>> dataset.add_meta(name, qtype, label, cats)
```

```
>>> dataset.meta('test_var')
single                              codes       texts missing
test_var: The test variable label
1                                       1  test_cat_1    None
2                                       2  test_cat_2    None
3                                      98  Don't know    None
```

Note: All three approaches are also valid for defining the `items` object for `array`-typed `masks`.

3.3 Derotation

3.3.1 What is derotation

Derotation of `data` is necessary if brands, products or something similar (**levels**) are assessed and each respondent (case) rates a different selection of that levels. So each **case** has several **responses**. Derotation now means, that the `data` is switched from case-level to responses-level.

Example: `q1_1/q1_2`: On a scale from 1 to 10, how much do you like the following drinks?

1: water
2: cola
3: lemonade
4: beer

“data“

id	drink_1	drink_2	q1_1	q1_2	gender
case1	1	3	2	8	1
case2	1	4	9	5	2
case3	2	4	6	10	1

derotated ‘‘data‘‘

	drink	drink_levelled	q1	gender
case1	1	1	2	1
case1	2	3	8	1
case2	1	1	9	2
case2	2	4	5	2
case3	1	2	6	1
case3	2	4	10	1

To identify which case rates which levels, some key-/level-variables are included in the `data`, in this example `drink_1` and `drink_2`. Variables (for example `gender`) that are not included to this loop can also be added.

3.3.2 How to use `DataSet.derotate()`

The `DataSet` method takes a few parameters:

- `levels`: `dict` of `list`

  Contains all key-/level-variables and the name for the new levelled variable. All key-/level-variables must have the same `value_map`.

```
>>> levels = {'drink': ['drink_1', 'drink_2']}
```

- `mapper`: `list` of `dicts` of `list`

  Contains the looped questions and the new `column` name to which the looped questions will be combinded.

```
>>> mapper = [{'q1': ['q1_1', 'q1_2']}]
```

- `other`: `str` or `list` of `str`

  Contains all variables that should be assumed to the derotated `data`, but which are not included in the loop.

```
>>> other = 'gender'
```

- `unique_key`: `str`

  Name of varibale that identifies cases in the initial `data`.

```
>>> unique_key = 'id'
```

- `dropna`: `bool`, default `True`

  If a case rates less then the possible counts of levels, these responses will be droped.

```
>>> ds = dataset.derotate(levels = {'drink': ['drink_1', 'drink_2']},
...                       mapper = [{'q1': ['q1_1', 'q1_2']}],
...                       other = 'gender',
...                       unique_key = 'id',
...                       dropna = True)
```

3.3.3 What about `arrays`?

It is possible that also `arrays` are looped. In this case a mapper can look like this:

```
>>> mapper = [{'q12_1': ['q12a[{q12a_1}].q12a_grid', 'q12b[{q12b_1}].q12b_grid',
...                      'q12c[{q12c_1}].q12c_grid', 'q12d[{q12d_1}].q12d_grid']},
...           {'q12_2': ['q12a[{q12a_2}].q12a_grid', 'q12b[{q12b_2}].q12b_grid',
...                      'q12c[{q12c_2}].q12c_grid', 'q12d[{q12d_2}].q12d_grid']},
...           {'q12_3': ['q12a[{q12a_3}].q12a_grid', 'q12b[{q12b_3}].q12b_grid',
...                      'q12c[{q12c_3}].q12c_grid', 'q12d[{q12d_3}].q12d_grid']},
...           {'q12_4': ['q12a[{q12a_4}].q12a_grid', 'q12b[{q12b_4}].q12b_grid',
...                      'q12c[{q12c_4}].q12c_grid', 'q12d[{q12d_4}].q12d_grid']},
...           {'q12_5': ['q12a[{q12a_5}].q12a_grid', 'q12b[{q12b_5}].q12b_grid',
...                      'q12c[{q12c_5}].q12c_grid', 'q12d[{q12d_5}].q12d_grid']},
...           {'q12_6': ['q12a[{q12a_6}].q12a_grid', 'q12b[{q12b_6}].q12b_grid',
...                      'q12c[{q12c_6}].q12c_grid', 'q12d[{q12d_6}].q12d_grid']},
...           {'q12_7': ['q12a[{q12a_7}].q12a_grid', 'q12b[{q12b_7}].q12b_grid',
...                      'q12c[{q12c_7}].q12c_grid', 'q12d[{q12d_7}].q12d_grid']},
...           {'q12_8': ['q12a[{q12a_8}].q12a_grid', 'q12b[{q12b_8}].q12b_grid',
...                      'q12c[{q12c_8}].q12c_grid', 'q12d[{q12d_8}].q12d_grid']},
...           {'q12_9': ['q12a[{q12a_9}].q12a_grid', 'q12b[{q12b_9}].q12b_grid',
...                      'q12c[{q12c_9}].q12c_grid', 'q12d[{q12d_9}].q12d_grid']},
...           {'q12_10': ['q12a[{q12a_10}].q12a_grid', 'q12b[{q12b_10}].q12b_grid',
...                       'q12c[{q12c_10}].q12c_grid', 'q12d[{q12d_10}].q12d_grid']},
...           {'q12_11': ['q12a[{q12a_11}].q12a_grid', 'q12b[{q12b_11}].q12b_grid',
...                       'q12c[{q12c_11}].q12c_grid', 'q12d[{q12d_11}].q12d_grid']},
...           {'q12_12': ['q12a[{q12a_12}].q12a_grid', 'q12b[{q12b_12}].q12b_grid',
...                       'q12c[{q12c_12}].q12c_grid', 'q12d[{q12d_12}].q12d_grid']},
...           {'q12_13': ['q12a[{q12a_13}].q12a_grid', 'q12b[{q12b_13}].q12b_grid',
...                       'q12c[{q12c_13}].q12c_grid', 'q12d[{q12d_13}].q12d_grid']}]]
```

Can be also writen like this:

```
>>> for y in frange('1-13'):
...     q_group = []
...     for x in  ['a', 'b', 'c', 'd']:
...         var = 'q12{}'.format(x)
```

(continues on next page)

(continued from previous page)

```
...         var_grid = var + '[{' +  var + '_{}'.format(y) + '}].' + var + '_grid'
...         q_group.append(var_grid)
...     mapper.append({'q12_{}'.format(y): q_group})
```

So the derotated `dataset` will lose its `meta` information about the `mask` and only the `columns` `q12_1` to `q12_13` will be added. To receive back the `mask` structure, use the method `dataset.to_array()`:

```
>>> variables = [{'q12_1': u'label 1'},
...              {'q12_2': u'label 2'},
...              {'q12_3': u'label 3'},
...              {'q12_4': u'label 4'},
...              {'q12_5': u'label 5'},
...              {'q12_6': u'label 6'},
...              {'q12_7': u'label 7'},
...              {'q12_8': u'label 8'},
...              {'q12_9': u'label 9'},
...              {'q12_10': u'label 10'},
...              {'q12_11': u'label 11'},
...              {'q12_12': u'label 12'},
...              {'q12_13': u'label 13'}]
>>> ds.to_array('qTP', variables, 'Var_name')
```

`variables` can also be a list of variable-names, then the `mask-items` will be named by its belonging `columns`.

`arrays` included in `other` will keep their `meta` structure.

CHAPTER 4

Data processing

4.1 DataSet components

4.1.1 Case and meta data

`Quantipy` builds upon the `pandas` library to feature the `DataFrame` and `Series` objects in the case data component of its `DataSet` object. Additionally, each `DataSet` offers a metadata component to describe the data columns and provide additional information on the characteristics of the underlying structure. The metadata document is implemented as a nested `dict` and provides the following `keys` on its first level:

element	contains
`'type'`	case data type
`'info'`	info on the source data
`'lib'`	shared use references
`'columns'`	info on `DataFrame` columns (Quantipy types, labels, etc.)
`'sets'`	ordered groups of variables pointing to other parts of the meta
`'masks'`	complex variable type definitions (arrays, dichotomous, etc.)

4.1.2 `columns` and `masks` objects

There are two variable collections inside a `Quantipy` metadata document: `'columns'` is storing the meta for each accompanying `pandas.DataFrame` column object, while `'masks'` are building upon the regular `'columns'` metadata but additionally employ special meta instructions to define complex data types. An example is the the `'array'` type that (in MR speak) maps multiple "question" variables to one "answer" object.

"Simple"" data definitons that are supported by `Quantipy` can either be numeric `'float'` and `'int'` types, categorical `'single'` and `'delimited set'` variables or of type `'string'`, `'date'` and `'time'`.

4.1.3 Languages: `text` and `text_key` mappings

Throughout `Quantipy` metadata all label information, e.g. variable question texts and category descriptions, are stored in `text` objects that are mapping different language (or context) versions of a label to a specific `text_key`. That way the metadata can support multi-language and multi-purpose (for example detailed/extensive vs. short question texts) label information in a digestable format that is easy to query:

```
>>> meta['columns']['q1']['text']
{'de-DE': 'Das ist ein langes deutsches Label',
 u'en-GB': u'What is your main fitness activity?',
 'x edits': {'de-DE': 'German build label', 'en-GB': 'English build label'}}
```

Valid `text_key` settings are:

text_key	Language / context
'en-GB'	English
'de-DE'	German
'fr-FR'	French
'da-DK'	Danish
'sv-SV'	Swedish
'nb-NO'	Norwegian
'fi-FI'	Finnish
'x edits'	Build label edit for x-axis
'y edits'	Build label edit for y-axis

4.1.4 Categorical `values` object

`single` and `delimited set` variables restrict the possible case data entries to a list of `values` that consist of numeric answer codes and their `text` labels, defining distinct categories:

```
>>> meta['columns']['q1']['values']
[{'value': 1,
  'text': {'en-GB': 'Dog'}
 },
 {'value': 2,
  'text': {'en-GB': 'Cat'}
 },
 {'value': 3,
  'text': {'en-GB': 'Bird'}
 },
 {'value': -9,
  'text': {'en-GB': 'Not an animal'}
 }]
```

4.1.5 The `array` type

Turning to the `masks` collection of the metadata, `array` variables group together a collection of variables that share a common response options scheme, i.e. different statements (usually referencing a broader topic) that are answered using the same scale. In the `Quantipy` metadata document, an `array` variable has a `subtype` that describes the type of the constructing source variables listed in the `items` object. In contrast to simple variable types, any categorical `values` metadata is stored inside the shared information collection `lib`, for access from both the `columns` and `masks` representation of `array` elements:

```
>>> meta['masks']['q5']
{u'items': [{u'source': u'columns@q5_1', u'text': {u'en-GB': u'Surfing'}},
  {u'source': u'columns@q5_2', u'text': {u'en-GB': u'Snowboarding'}},
  {u'source': u'columns@q5_3', u'text': {u'en-GB': u'Kite boarding'}},
  {u'source': u'columns@q5_4', u'text': {u'en-GB': u'Parachuting'}},
  {u'source': u'columns@q5_5', u'text': {u'en-GB': u'Cave diving'}},
  {u'source': u'columns@q5_6', u'text': {u'en-GB': u'Windsurfing'}}],
 u'name': u'q5',
 u'subtype': u'single',
 u'text': {u'en-GB': u'How likely are you to do each of the following in the next␣
↪year?'},
 u'type': u'array',
 u'values': 'lib@values@q5'}
```

```
>>> meta['lib']['values']['q5']
[{u'text': {u'en-GB': u'I would refuse if asked'}, u'value': 1},
 {u'text': {u'en-GB': u'Very unlikely'}, u'value': 2},
 {u'text': {u'en-GB': u"Probably wouldn't"}, u'value': 3},
 {u'text': {u'en-GB': u'Probably would if asked'}, u'value': 4},
 {u'text': {u'en-GB': u'Very likely'}, u'value': 5},
 {u'text': {u'en-GB': u"I'm already planning to"}, u'value': 97},
 {u'text': {u'en-GB': u"Don't know"}, u'value': 98}]
```

Exploring the `columns` meta of an array item shows the same `values` reference pointer and informs about its `parent` meta structure, i.e. the array's `masks` defintion:

```
>>> meta['columns']['q5_1']
{u'name': u'q5_1',
 u'parent': {u'masks@q5': {u'type': u'array'}},
 u'text': {u'en-GB': u'How likely are you to do each of the following in the next␣
↪year? - Surfing'},
 u'type': u'single',
 u'values': u'lib@values@q5'}
```

4.2 I/O

4.2.1 Starting from native components

4.2.1.1 Using a standalone `pd.DataFrame`

`Quantipy` can create a meta document from a inferring its variable types from the `dtypes` of a `pd.DataFrame`. In that process, `int`, `float` and `string` data types are created inside the meta component of the `DataSet`. In this basic form, `text` label information is missing. For a example, given a `pd.DataFrame` as per:

```
>>> casedata = [[1000, 10, 1.2, 'text1'],
...             [1001, 4, 3.4, 'jjda'],
...             [1002, 8, np.NaN, 'what?'],
...             [1003, 8, 7.81, '---' ],
...             [1004, 5, 3.0, 'hello world!']]
>>> df = pd.DataFrame(casedata, columns=['identity', 'q1', 'q2', 'q3'])
>>> df
   identity  q1    q2           q3
0      1000  10  1.20        text1
```

(continues on next page)

(continued from previous page)

```
1       1001    4  3.40          jjda
2       1002    8   NaN         what?
3       1003    8  7.81           ---
4       1004    5  3.00  hello world!
```

... the conversion is adding matching metadata to the `DataSet` instance:

```
>>> dataset = qp.DataSet(name='example', dimensions_comp=False)
>>> dataset.from_components(df)
Inferring meta data from pd.DataFrame.columns (4)...
identity: dtype: int64 - converted: int
q1: dtype: int64 - converted: int
q2: dtype: float64 - converted: float
q3: dtype: object - converted: string
```

```
>>> dataset.meta()['columns']['q2']
{'text': {'en-GB': ''}, 'type': 'float', 'name': 'q2', 'parent': {}, 'properties': {
↪'created': True}}
```

4.2.1.2 `.csv` / `.json` pairs

We can easily read in `Quantipy` native data with the `read_quantipy()` method and providing the paths to both the `.csv` and `.json` file (file extensions are handled automatically), e.g.:

```
>>> folder = './Data/'
>>> file_name = 'Example Data (A)'
>>> path_csv = path_json = folder + file_name
```

```
>>> dataset = qp.DataSet(name='example', dimensions_comp=False)
>>> dataset.read_quantipy(path_json, path_csv)
DataSet: ./Data/example
rows: 8255 - columns: 76
Dimensions compatibility mode: False
```

We can that access the case and metadata components:

```
>>> dataset.data()['q4'].head()
0    1
1    2
2    2
3    1
4    1
Name: q4, dtype: int64
```

```
>>> meta = dataset.meta()['columns']['q4']
>>> json.dumps(meta)
{
    "values": [
        {
            "text": {
                "en-GB": "Yes"
            },
            "value": 1
        },
```

(continues on next page)

(continued from previous page)

```
        {
            "text": {
                "en-GB": "No"
            },
            "value": 2
        }
    ],
    "text": {
        "en-GB": "Do you ever participate in sports activities with people in your␣
↪household?"
    },
    "type": "single",
    "name": "q4",
    "parent": {}
}
```

4.2.2 Third party conversions

4.2.2.1 Supported conversions

In adddition to providing plain `.csv`/`.json` data (pairs), source files can be read into Quantipy using a number of I/O functions to deal with standard file formats encountered in the market research industry:

Software	Format	Read	Write
SPSS Statistics	.sav	Yes	Yes
SPSS Dimensions	.dff/.mdd	Yes	Yes
Decipher	tab-delimited .json/ .txt	Yes	No
Ascribe	tab-delimited .xml/ .txt	Yes	No

The following functions are designed to convert the different file formats' structures into inputs understood by Quantipy.

4.2.2.2 SPSS Statistics

Reading:

```
>>> from quantipy.core.tools.dp.io import read_spss
>>> meta, data = read_spss(path_sav)
```

Note: On a Windows machine you MUST use `ioLocale=None` when reading from SPSS. This means if you are using a Windows machine your base example for reading from SPSS is `meta, data = read_spss(path_sav, ioLocale=None)`.

When reading from SPSS you have the opportunity to specify a custom dichotomous values map, that will be used to convert all dichotomous sets into Quantipy delimited sets, using the `dichot` argument.

The entire read operation will use the same map on all dichotomous sets so they must be applied uniformly throughout the SAV file. The default map that will be used if none is provided will be `{'yes': 1, 'no': 0}`.

```
>>> meta, data = read_spss(path_sav, dichot={'yes': 1, 'no': 2})
```

SPSS dates will be converted to pandas dates by default but if this results in conversion issues or failures you can read the dates in as Quantipy strings to deal with them later, using the `dates_as_strings` argument.

```
>>> meta, data = read_spss(path_sav, dates_as_strings=True)
```

Writing:

```
>>> from quantipy.core.tools.dp.io import write_spss
>>> write_spss(path_sav, meta, data)
```

By default SPSS files will be generated from the `'data file'` set found in `meta['sets']`, but a custom set can be named instead using the `from_set` argument.

```
>>> write_spss(path_sav_analysis, meta, data, from_set='sav-export')
```

The custom set must be well-formed:

```
>>> "sets" : {
...     "sav-export": {
...         "items": [
...             "columns@Q1",
...             "columns@Q2",
...             "columns@Q3",
...             ...
...         ]
...     }
... }
```

4.2.2.3 Dimensions

Reading:

```
>>> from quantipy.core.tools.dp.io import read_dimensions
>>> meta, data = read_dimensions(path_mdd, path_ddf)
```

4.2.2.4 Decipher

Reading:

```
>>> from quantipy.core.tools.dp.io import read_decipher
>>> meta, data = read_decipher(path_json, path_txt)
```

4.2.2.5 Ascribe

Reading:

```
>>> from quantipy.core.tools.dp.io import read_ascribe
>>> meta, data = read_ascribe(path_xml, path_txt)
```

4.3 DataSet management

4.3.1 Setting the variable order

The global variable order of a `DataSet` is dictated by the content of the `meta['sets']['data file']['items']` list and reflected in the structure of the case data component's `pd.DataFrame.columns`. There are two ways to set a new order using the `order(new_order=None, reposition=None)` method:

Define a full order

Using this apporach requires that all `DataSet` variable names are passed via the `new_order` parameter. Providing only a subset of the variables will raise a `ValueError`:

```
>>> dataset.order(['q1', 'q8'])
ValueError: 'new_order' must contain all DataSet variables.
```

Text...

Change positions relatively

Often only a few changes to the natural order of the `DataSet` are necessary, e.g. derived variables should be moved alongside their originating ones or specific sets of variables (demographics, etc.) should be grouped together. We can achieve this using the `reposition` parameter as follows:

Text...

4.3.2 Cloning, filtering and subsetting

Sometimes you want to cut the data into sections defined by either case/respondent conditions (e.g. a survey wave) or a collection of variables (e.g. a specific part of the questionnaire). To not permanently change an existing `DataSet` by accident, draw a copy of it first:

```
>>> copy_ds = dataset.clone()
```

Then you can use `filter()` to restrict cases (rows) or `subset()` to keep only a selected range of variables (columns). Both methods can be used inplace but will return a new object by default.

```
>>> keep = {'Wave': [1]}
>>> copy_ds.filter(alias='first wave', condition=keep, inplace=True)
>>> copy_ds._data.shape
(1621, 76)
```

After the filter has been applied, the `DataSet` is only showing cases that contain the value 1 in the `'Wave'` variable. The filter alias (a short name to describe the arbitrarily complex filter `condition`) is attached to the instance:

```
>>> copy_ds.filtered
only first wave
```

We are now further reducing the `DataSet` by dropping all variables except the three `array` variables `'q5'`, `'q6'`, and `'q7'` using `subset()`.

```
>>> reduced_ds = copy_ds.subset(variables=['q5', 'q6', 'q7'])
```

We can see that only the requested variables (`masks` defintitions and the constructing `array` items) remain in `reduced_ds`:

```
>>> reduced_ds.by_type()
size: 1621 single delimited set array int float string date time N/A
0             q5_1                   q5
1             q5_2                   q7
2             q5_3                   q6
3             q5_4
4             q5_5
5             q5_6
6             q6_1
7             q6_2
8             q6_3
9             q7_1
10            q7_2
11            q7_3
12            q7_4
13            q7_5
14            q7_6
```

4.3.3 Merging

Intro text... As opposed to reducing an existing file...

4.3.3.1 Vertical (cases/rows) merging

Text

4.3.3.2 Horizontal (variables/columns) merging

Text

4.3.4 Savepoints and state rollback

When working with big `DataSets` and needing to perform a lot of data preparation (deriving large amounts of new variables, lots of meta editing, complex cleaning, ...) it can be beneficial to quickly store a snapshot of a clean and consistent state of the `DataSet`. This is most useful when working in interactive sessions like **IPython** or **Jupyter notebooks** and might prevent you from reloading files from disk or waiting for previous processes to finish.

Savepoints are stored via `save()` and can be restored via `revert()`.

Note: Savepoints only exists in memory and are not written to disk. Only one savepoint can exist, so repeated `save()` calls will overwrite any previous versions of the `DataSet`. To permanently save your data, please use one of the `write` methods, e.g. `write_quantipy()`.

4.4 Inspecting variables

4.4.1 Querying and slicing case data

A `qp.DataSet` is mimicking `pandas`-like item access, i.e. passing a variable name into the `[]`-accessor will return a `pandas.DataFrame` view of the case data component. That means that we can chain any `pandas.DataFrame` method to the query:

```
>>> ds['q9'].head()
     q9
0   99;
1  1;4;
2   98;
3  1;4;
4   99;
```

There is the same support for selecting multiple variables at once:

```
>>> ds[['q9', 'gender']].head()
     q9  gender
0   99;       1
1  1;4;       2
2   98;       1
3  1;4;       1
4   99;       1
```

To integrate `array` (`masks`) variables into this behaviour, passing an `array` name will automatically call its item list:

```
>>> ds['q6'].head()
   q6_1  q6_2  q6_3
0     1     1     1
1     1   NaN     1
2     1   NaN     2
3     2   NaN     2
4     2    10    10
```

This can be combined with the `list`-based selection as well:

```
>>> ds[['q6', 'q9', 'gender']].head()
   q6_1  q6_2  q6_3    q9  gender
0     1     1     1   99;       1
1     1   NaN     1  1;4;       2
2     1   NaN     2   98;       1
3     2   NaN     2  1;4;       1
4     2    10    10   99;       1
```

`DataSet` case data supports row-slicing based on complex logical conditions to inspect subsets of the data. We can use the `take()` with a `Quantipy` logic operation naturally for this:

```
>>> condition = intersection(
...     [{'gender': [1]},
...      {'religion': [3]},
...      {'q9': [1, 4]}])
>>> take = ds.take(condition)
```

```
>>> ds[take, ['gender', 'religion', 'q9']].head()
     gender  religion      q9
52        1         3  1;2;4;
357       1         3  1;3;4;
671       1         3  1;3;4;
783       1         3  2;3;4;
802       1         3      4;
```

See also:

Please find an overview of `Quantipy` logical operators and data slicing and masking in the *docs about complex logical conditions*!

4.4.2 Variable and value existence

any, all, code_count, is_nan, var_exists, codes_in_data, is_like_numeric variables

---

We can use `variables()` and `var_exists()` to generally test the membership of variables inside `DataSet`. The former is showing the list of all variables registered inside the `'data file' set`, the latter is checking if a variable's `name` is found in either the `'columns'` or `'masks'` collection. For our example data, the variables are:

```
>>> dataset.variables()
```

So a test for the `array 'q5'` should be positive:

```
>>> dataset.var_exists('q5')
True
```

In addition to `Quantipy`'s complex logic operators, the `DataSet` class offers some quick case data operations for code existence tests. To return a `pandas.Series` of all empty rows inside a variable use `is_nan()` as per:

```
>>> dataset.is_nan('q8').head()
0    True
1    True
2    True
3    True
4    True
Name: q8, dtype: bool
```

Which we can also use to quickly check the number of missing cases...

```
>>> dataset.is_nan('q8').value_counts()
True     5888
False    2367
Name: q8, dtype: int64
```

... as well as use the result as slicer for the `DataSet` case data component, e.g. to show the non-empty rows:

```
>>> slicer = dataset.is_nan('q8')
>>> dataset[~slicer, 'q8'].head()
Name: q8, dtype: int64
7       5;
11      5;
13    1;4;
```

(continues on next page)

(continued from previous page)

```
14    4;5;
23    1;4;
Name: q8, dtype: object
```

Especially useful for `delimited set` and `array` data, the `code_count()` method is creating the `pandas.Series` of response values found. If applied on an `array`, the result is expressed across all source item variables:

```
>>> dataset.code_count('q6').value_counts()
3    5100
2    3155
dtype: int64
```

. . . which means that not all cases contain answers in all three of the array's items.

With some basic `pandas` we can double-check this result:

```
>>> pd.concat([dataset['q6'], dataset.code_count('q6')], axis=1).head()
   q6_1  q6_2  q6_3  0
0     1   1.0     1  3
1     1   NaN     1  2
2     1   NaN     2  2
3     2   NaN     2  2
4     2  10.0    10  3
```

`code_count()` can optionally ignore certain codes via the `count_only` and `count_not` parameters:

```
>>> q2_count = dataset.code_count('q2', count_only=[1, 2, 3])
>>> pd.concat([dataset['q2'], q2_count], axis=1).head()
          q2  0
0  1;2;3;5;  3
1      3;6;  1
2       NaN  0
3       NaN  0
4       NaN  0
```

Similarly, the `any()` and `all()` methods yield slicers for cases obeying the condition that at least one / all of the provided codes are found in the response. Again, for `array` variables the conditions are extended across all the items:

```
>>> dataset[dataset.all('q6', 5), 'q6']
      q6_1  q6_2  q6_3
374      5   5.0     5
2363     5   5.0     5
2377     5   5.0     5
4217     5   5.0     5
5530     5   5.0     5
5779     5   5.0     5
5804     5   5.0     5
6328     5   5.0     5
6774     5   5.0     5
7269     5   5.0     5
8148     5   5.0     5
```

```
>>> dataset[dataset.all('q8', [1, 2, 3, 4, 96]), 'q8']
845     1;2;3;4;5;96;
6242      1;2;3;4;96;
7321      1;2;3;4;96;
Name: q8, dtype: object
```

```
>>> dataset[dataset.any('q8', [1, 2, 3, 4, 96]), 'q8'].head()
13      1;4;
14      4;5;
23      1;4;
24    1;3;4;
25      1;4;
Name: q8, dtype: object
```

4.4.3 Variable types

To get a summary of the all variables grouped by type, call `by_type()` on the `DataSet`:

```
>>> ds.by_type()
size: 8255     single delimited set array            int     float string        date␣
↪     time N/A
0              gender            q2    q5  record_number    weight    q8a  start_time␣
↪ duration
1            locality            q3    q7      unique_id  weight_a    q9a    end_time
2           ethnicity            q8    q6            age  weight_b
3            religion            q9           birth_day
4                  q1                      birth_month
5                 q2b                       birth_year
6                  q4
7                q5_1
8                q5_2
9                q5_3
10               q5_4
11               q5_5
12               q5_6
13               q6_1
14               q6_2
15               q6_3
16               q7_1
17               q7_2
18               q7_3
19               q7_4
20               q7_5
21               q7_6
```

We can restrict the output to certain types by providing the desired ones in the `types` parameter:

```
>>> ds.by_type(types='delimited set')
size: 8255 delimited set
0                    q2
1                    q3
2                    q8
3                    q9
```

```
>>> ds.by_type(types=['delimited set', 'float'])
size: 8255 delimited set     float
0                    q2    weight
1                    q3  weight_a
2                    q8  weight_b
3                    q9       NaN
```

In addition to that, `DataSet` implements the following methods that return the corresponding variables as a `list` for easy iteration:

```
DataSet.singles
       .delimied_sets()
       .ints()
       .floats()
       .dates()
       .strings()
       .masks()
       .columns()
       .sets()
```

```
>>> ds.delimited_sets()
[u'q3', u'q2', u'q9', u'q8']
```

```
>>> for delimited_set in ds.delimited_sets():
...     print delimited_set
q3
q2
q9
q8
```

4.4.4 Slicing & dicing metadata objects

Although it is possible to access a `DataSet` meta component via its `_meta` attribute directly, the prefered way to inspect and interact with with the metadata is to use `DataSet` methods. For instance, the easiest way to view the most important meta on a variable is to use the `meta()` method:

```
>>> ds.meta('q8')
delimited set                                        codes                          ␣
↪texts missing
q8: Which of the following do you regularly skip?
1                                                        1                        ␣
↪Breakfast    None
2                                                        2          Mid-morning␣
↪snacking    None
3                                                        3                          ␣
↪Lunch    None
4                                                        4        Mid-afternoon␣
↪snacking    None
5                                                        5                         ␣
↪Dinner    None
6                                                       96                  None of␣
↪them    None
7                                                       98  Don't know (it varies a␣
↪lot)    None
```

This output is extended with the `item` metadata if an `array` is passed:

```
>>> ds.meta('q6')
single                                               items                    item texts␣
↪ codes                          texts missing
q6: How often do you take part in any of the fo...
1                                                     q6_1               Exercise alone␣
↪     1     Once a day or more often    None
```

(continues on next page)

(continued from previous page)

```
2                                                    q6_2       Join an exercise class␣
↪      2               Every few days    None
3                                                    q6_3  Play any kind of team sport␣
↪      3                   Once a week    None
4                                                                                      ␣
↪      4             Once a fortnight    None                                          ␣
5                                                                                      ␣
↪      5                  Once a month    None                                       ␣
6                                                                                      ␣
↪      6        Once every few months    None                                       ␣
7                                                                                      ␣
↪      7        Once every six months    None                                       ␣
8                                                                                      ␣
↪      8                  Once a year    None                                       ␣
9                                                                                      ␣
↪      9  Less often than once a year    None                                       ␣
10                                                                                     ␣
↪     10                       Never    None
```

If the variable is not categorical, `meta()` returns simply:

```
>>> ds.meta('weight_a')
                            float
weight_a: Weight (variant A)   N/A
```

`DataSet` also provides a lot of methods to access and return the several meta objects of a variable to make various data processing tasks easier:

Variable labels: `quantipy.core.dataset.DataSet.text()`

```
>>> ds.text('q8', text_key=None)
Which of the following do you regularly skip?
```

`values` **object**: `quantipy.core.dataset.DataSet.values()`

```
>>> ds.values('gender', text_key=None)
[(1, u'Male'), (2, u'Female')]
```

Category codes: `quantipy.core.dataset.DataSet.codes()`

```
>>> ds.codes('gender')
[1, 2]
```

Category labels: `quantipy.core.dataset.DataSet.value_texts()`

```
>>> ds.value_texts('gender', text_key=None)
[u'Male', u'Female']
```

`items` **object**: `quantipy.core.dataset.DataSet.items()`

```
>>> ds.items('q6', text_key=None)
[(u'q6_1', u'How often do you exercise alone?'),
 (u'q6_2', u'How often do you take part in an exercise class?'),
 (u'q6_3', u'How often do you play any kind of team sport?')]
```

Item `'columns'` **sources**: `quantipy.core.dataset.DataSet.sources()`

```
>>> ds.sources('q6')
[u'q6_1', u'q6_2', u'q6_3']
```

Item labels: `quantipy.core.dataset.DataSet.item_texts()`

```
>>> ds.item_texts('q6', text_key=None)
[u'How often do you exercise alone?',
 u'How often do you take part in an exercise class?',
 u'How often do you play any kind of team sport?']
```

4.5 Editing metadata

4.5.1 Creating meta from scratch

It is very easy to add new variable metadata to a `DataSet` via `add_meta()` which let's you create all supported variable types. Each new variable needs at least a `name`, `qtype` and `label`. With this information a `string`, `int`, `float` or `date` variable can be defined, e.g.:

```
>>> ds.add_meta(name='new_int', qtype='int', label='My new int variable')
>>> ds.meta('new_int')
                            int
new_int: My new int variable  N/A
```

Using the `categories` parameter we can create categorical variables of type `single` or `delimited set`. We can provide the `categories` in two different ways:

```
>>> name, qtype, label = 'new_single', 'single', 'My new single variable'
```

Providing a list of category labels (codes will be enumerated starting from `1`):

```
>>> cats = ['Category A', 'Category B', 'Category C']
```

```
>>> ds.add_meta(name, qtype, label, categories=cats)
>>> ds.meta('new_single')
single                              codes       texts missing
new_single: My new single variable
1                                       1  Category A    None
2                                       2  Category B    None
3                                       3  Category C    None
```

Providing a list of tuples pairing codes and labels:

```
>>> cats = [(1, 'Category A'), (2, 'Category B'), (99, 'Category C')]
```

```
>>> ds.add_meta(name, qtype, label, categories=cats)
>>> ds.meta('new_single')
single                              codes       texts missing
new_single: My new single variable
1                                       1  Category A    None
2                                       2  Category B    None
3                                      99  Category C    None
```

Note: `add_meta()` is preventing you from adding ill-formed or inconsistent variable information, e.g. it is not possible to add `categories` to an `int`...

```
>>> ds.add_meta('new_int', 'int', 'My new int variable', cats)
ValueError: Numerical data of type int does not accept 'categories'.
```

...and you must provide `categories` when trying to add categorical data:

```
>>> ds.add_meta(name, 'single', label, categories=None)
ValueError: Must provide 'categories' when requesting data of type single.
```

Similiar to the usage of the `categories` argument, `items` is controlling the creation of an `array`, i.e. specifying `items` is automatically preparing the `'masks'` and `'columns'` metadata. The `qtype` argument in this case always refers to the type of the corresponding `'columns'`.

```
>>> name, qtype, label = 'new_array', 'single', 'My new array variable'
>>> cats = ['Category A', 'Category B', 'Category C']
```

Again, there are two alternatives to construct the `items` object:

Providing a list of item labels (item identifiers will be enumerated starting from `1`):

```
>>> items = ['Item A', 'Item B', 'Item C', 'Item D']
```

```
>>> ds.add_meta(name, qtype, label, cats, items=items)
>>> ds.meta('new_array')
single                                  items item texts codes       texts missing
new_array: My new array variable
1                                 new_array_1     Item A     1  Category A    None
2                                 new_array_2     Item B     2  Category B    None
3                                 new_array_3     Item C     3  Category C    None
4                                 new_array_4     Item D
```

Providing a list of tuples pairing item identifiers and labels:

```
>>> items = [(1, 'Item A'), (2, 'Item B'), (97, 'Item C'), (98, 'Item D')]
```

```
>>> ds.add_meta(name, qtype, label, cats, items)
>>> ds.meta('new_array')
single                                    items item texts codes       texts missing
new_array: My new array variable
1                                  new_array_1     Item A     1  Category A    None
2                                  new_array_2     Item B     2  Category B    None
3                                 new_array_97     Item C     3  Category C    None
4                                 new_array_98     Item D
```

Note: For every created variable, `add_meta()` is also adding the relevant `columns` into the `pd.DataFrame` case data component of the `DataSet` to keep it consistent:

```
>>> ds['new_array'].head()
   new_array_1  new_array_2  new_array_97  new_array_98
0          NaN          NaN           NaN           NaN
1          NaN          NaN           NaN           NaN
2          NaN          NaN           NaN           NaN
```

(continues on next page)

(continued from previous page)

```
3          NaN          NaN          NaN          NaN
4          NaN          NaN          NaN          NaN
```

4.5.2 Renaming

It is possible to attach new names to `DataSet` variables. Using the `rename()` method will replace all former variable `keys` and other mentions inside the metadata document and exchange the `DataFrame` column names. For `array` variables only the `'masks'` name reference is updated by default – to rename the corresponding `items` a dict mapping item position number to new name can be provided.

```
>>> ds.rename(name='q8', new_name='q8_with_a_new_name')
```

As mentioned, renaming a `'masks'` variable will leave the items untouched:

```
>>>
```

But we can simply provide their new names as per:

```
>>>
```

```
>>>
```

4.5.3 Changing & adding `text` info

All `text`-related `DataSet` methods expose the `text_key` argument to control to which language or context a label is added. For instance we can add a German variable label to `'q8'` with `set_variable_text()`:

```
>>> ds.set_variable_text(name='q8', new_text='Das ist ein deutsches Label', text_key=
↪'de-DE')
```

```
>>> ds.text('q8', 'en-GB')
Which of the following do you regularly skip?
```

```
>>> ds.text('q8', 'de-DE')
Das ist ein deutsches Label
```

To change the `text` inside the `values` or `items` metadata, we can similarly use `set_value_text` and `set_item_text()`:

```
>>>
```

When working with multiple language versions of the metadata, it might be required to copy one language's `text` meta to another one's, for instance if there are no fitting translations or the correct translation is missing. In such cases you can use `force_texts()` to copy the meta of a source `text_key` (specified in the `` `copy_from `` parameter) to a target `text_key` (indicated via `copy_to`).

```
>>>
```

```
>>>
```

With `clean_texts()` you also have the option to replace specific characters, terms or formatting tags (i.e. `html`) from all `text` metadata of the `DataSet`:

```
>>>
```

4.5.4 Extending the `values` object

We can add new category defintitons to existing `values` meta with the `extend_values()` method. As when adding full metadata for categorical variables, new `values` can be generated by either providing only labels or tuples of codes and labels.

```
>>>
```

While the method will never allow adding duplicated numeric values for the categories, setting `safe` to `False` will enable you to add duplicated `text` meta, i.e. `values` could contain both `{'text': {'en-GB': 'No answer'}, 'value': 98}` and `{'text': {'en-GB': 'No answer'}, 'value': 99}`. By default, however, the method will strictly prohibit any duplicates in the resulting `values`.

```
>>>
```

4.5.5 Reordering the `values` object

4.5.6 Removing `DataSet` objects

4.6 Transforming variables

4.6.1 Copying

It's often recommended to draw a clean copy of a variable before starting to editing its meta or case data. With `copy()` you can add a copy to the `DataSet` that is identical to the original in all respects but its name. By default, the copy's name will be suffixed with `'_rec'`, but you can apply a custom suffix by providing it via the `suffix` argument (leaving out the `'_'` which is added automatically):

```
>>> ds.copy('q3')
>>> ds.copy('q3', suffix='version2')
```

```
>>> ds.delimited_sets
[u'q3', u'q2', u'q9', u'q8', u'q3_rec', u'q3_version2']
```

Querying the `DataSet`, we can see that all three version are looking identical:

```
>>> ds[['q3', 'q3_rec', 'q3_version2']].head()
       q3  q3_rec q3_version2
0  1;2;3;  1;2;3;      1;2;3;
1  1;2;3;  1;2;3;      1;2;3;
2  1;2;3;  1;2;3;      1;2;3;
3    1;3;    1;3;        1;3;
4      2;      2;          2;
```

We can, however, prevent copying the case data and simply add an "empty" copy of the variable by passing `copy_data=False`:

```
>>> ds.copy('q3', suffix='no_data', copy_data=False)
```

```
>>> ds[['q3', 'q3_rec', 'q3_version2', 'q3_no_data']].head()
       q3  q3_rec q3_version2  q3_no_data
0  1;2;3;  1;2;3;      1;2;3;         NaN
1  1;2;3;  1;2;3;      1;2;3;         NaN
2  1;2;3;  1;2;3;      1;2;3;         NaN
3    1;3;    1;3;        1;3;         NaN
4      2;      2;          2;         NaN
```

If we wanted to only copy a subset of the case data, we could also use a *logical slicer* and supply it in the `copy()` operation's `slicer` parameter:

```
>>> slicer = {'gender': [1]}
>>> ds.copy('q3', suffix='only_men', copy_data=True, slicer=slicer)
```

```
>>> ds[['q3', 'gender', 'q3_only_men']].head()
       q3  gender q3_only_men
0  1;2;3;       1      1;2;3;
1  1;2;3;       2         NaN
2  1;2;3;       1      1;2;3;
3    1;3;       1        1;3;
4      2;       1          2;
```

4.6.2 Inplace type conversion

You can change the characteristics of existing `DataSet` variables by converting from one `type` to another. Conversions happen `inplace`, i.e. no copy of the variable is taken prior to the operation. Therefore, you might want to take a `DataSet.copy()` before using the `convert(name, to)` method.

Conversions need to modify both the `meta` and `data` component of the `DataSet` and are limited to transformations that keep the original and new state of a variable consistent. The following conversions are currently supported:

name (from-type)	to='single'	to='delimited set'	to='int'	to='float'	to='string'
'single'	[X]	X	X	X	X
'delimited set'		[X]			
'int'	X		[X]	X	X
'float'				[X]	X
'string'	X		X*	X*	[X]
'date'	X				X

* *If all values of the variable are numerical, i.e.* `DataSet.is_like_numeric()` *returns* `True`.

Each of these conversions will rebuild the variable meta data to match the `to` type. This means, that for instance a variable that is `single` will lose its `values` object when transforming to `int`, while the reverse operation will create a `values` object that categorizes the unqiue numeric codes found in the case data with their `str` representation as `text` meta. Consider the variables `q1` (`single`) and `age` (`int`):

From type `single` **to** `int`:

```
>>> ds.meta('q1')
single                                      codes                                   ␣
↪texts missing
q1: What is your main fitness activity?
1                                             1                                  ␣
↪Swimming    None
2                                             2                        Running/
↪jogging    None
3                                             3                        Lifting␣
↪weights    None
4                                             4                                  ␣
↪Aerobics    None
5                                             5                                     ␣
↪Yoga    None
6                                             6                                  ␣
↪Pilates    None
7                                             7                       Football␣
↪(soccer)    None
8                                             8                                ␣
↪Basketball    None
9                                             9                                  ␣
↪Hockey    None
10                                           96                                  ␣
↪Other    None
11                                           98  I regularly change my fitness␣
↪activity    None
12                                           99       Not applicable - I don't␣
↪exercise    None
```

```
>>> ds.convert('q1', to='int')
>>> ds.meta('q1')
                                            int
q1: What is your main fitness activity?  N/A
```

From type `int` **to** `single`:

```
>>> ds.meta('age')
          int
age: Age  N/A
```

```
>>> ds.convert('age', to='single')
>>> ds.meta('age')
single    codes texts missing
age: Age
1            19    19    None
2            20    20    None
3            21    21    None
4            22    22    None
5            23    23    None
6            24    24    None
7            25    25    None
8            26    26    None
9            27    27    None
10           28    28    None
11           29    29    None
12           30    30    None
13           31    31    None
```

(continues on next page)

(continued from previous page)

```
14          32    32    None
15          33    33    None
16          34    34    None
17          35    35    None
18          36    36    None
19          37    37    None
20          38    38    None
21          39    39    None
22          40    40    None
23          41    41    None
24          42    42    None
25          43    43    None
26          44    44    None
27          45    45    None
28          46    46    None
29          47    47    None
30          48    48    None
31          49    49    None
```

4.6.3 Banding and categorization

In contrast to `convert()`, the `categorize()` method creates a new variable of type `single`, acting as a short-hand for creating a renamed copy and then type-transforming it. Therefore, it lets you quickly categorize the unique values of a `text`, `int` or `date` variable, storing `values` meta in the form of `{'text': {'en-GB': str(1)}, 'value': 1}`.

```
>>>
```

Flexible banding of numeric data is provided thorugh `DataSet.band()`: If a variable is banded, it will standardly be added to the `DataSet` via the original's name suffixed with `'banded'`, e.g. `'age_banded'`, keeping the originating variables `text` label. The `new_name` and `label` parameters can be used to create custom variable names and labels. The banding of the incoming data is controlled with the `bands` argument that expects a list containing `int`, `tuples` or `dict`, where each type is used for a different kind of group definition.

Banding with `int` and `tuple`:

- Use an `int` to make a band of only one value
- Use a `tuple` to indicate (inclusive) group limits
- `values` `text` meta is infered
- Example: `[0, (1, 10), (11, 14), 15, (16, 25)]`

Banding with `dict`:

- The dict `key` will dicate the group's `text` label meta
- The dict `value` can pick up an `int` / `tuple` (see above)
- Example: `[{'A': 0}, {'B': (1, 10)}, {'C': (11, 14)}, {'D': 15}, {'E': (16, 25)}]`
- Mixing allowed: `[0, {'A': (1, 10)}, (11, 14), 15, {'B': (16, 25)}]`

For instance, we could band `'age'` into a new variable called `'grouped_age'` with `bands` being:

```
>>> bands = [{'Younger than 35': (19, 34)},
...          (35, 39),
...          {'Exactly 40': 40},
...          41,
...          (42, 60)]
```

```
>>> ds.band(name='age', bands=bands, new_name='grouped_age', label=None)
```

```
>>> ds.meta('grouped_age')
single            codes            texts missing
grouped_age: Age
1                     1  Younger than 35    None
2                     2            35-39    None
3                     3       Exactly 40    None
4                     4               41    None
5                     5            42-60    None
```

```
>>> ds.crosstab('age', 'grouped_age')
Question        grouped_age. Age
Values                      All Younger than 35 35-39 Exactly 40   41 42-60
Question Values
age. Age All               8255            4308  1295        281  261  2110
         19                 245             245     0          0    0     0
         20                 277             277     0          0    0     0
         21                 270             270     0          0    0     0
         22                 323             323     0          0    0     0
         23                 272             272     0          0    0     0
         24                 263             263     0          0    0     0
         25                 246             246     0          0    0     0
         26                 252             252     0          0    0     0
         27                 260             260     0          0    0     0
         28                 287             287     0          0    0     0
         29                 270             270     0          0    0     0
         30                 271             271     0          0    0     0
         31                 264             264     0          0    0     0
         32                 287             287     0          0    0     0
         33                 246             246     0          0    0     0
         34                 275             275     0          0    0     0
         35                 258               0   258          0    0     0
         36                 236               0   236          0    0     0
         37                 252               0   252          0    0     0
         38                 291               0   291          0    0     0
         39                 258               0   258          0    0     0
         40                 281               0     0        281    0     0
         41                 261               0     0          0  261     0
         42                 290               0     0          0    0   290
         43                 267               0     0          0    0   267
         44                 261               0     0          0    0   261
         45                 257               0     0          0    0   257
         46                 259               0     0          0    0   259
         47                 243               0     0          0    0   243
         48                 271               0     0          0    0   271
         49                 262               0     0          0    0   262
```

4.6.4 Array transformations

Transposing arrays

`DataSet` offers tools to simplify common `array` variable operations. You can switch the structure of `items` vs. `values` by producing the one from the other using `transpose()`. The transposition of an array will always result in `items` that have the `delimited set` type in the corresponding `'columns'` metadata. That is because the transposed array is collecting what former items have been assignd per former value:

```
>>> ds.transpose('q5')
```

Original

```
>>> ds['q5'].head()
   q5_1  q5_2  q5_3  q5_4  q5_5  q5_6
0     2     2     2     2     1     2
1     5     5     3     3     3     5
2     5    98     5     5     1     5
3     5     5     1     5     3     5
4    98    98    98    98    98    98
```

```
>>> ds.meta('q5')
single                                                items     item texts  codes
↪            texts missing
q5: How likely are you to do each of the follow...
1                                                      q5_1        Surfing      1  I
↪would refuse if asked    None
2                                                      q5_2   Snowboarding      2
↪    Very unlikely    None
3                                                      q5_3  Kite boarding      3
↪Probably wouldn't    None
4                                                      q5_4    Parachuting      4
↪Probably would if asked    None
5                                                      q5_5    Cave diving      5
↪      Very likely    None
6                                                      q5_6    Windsurfing     97  I'm
↪already planning to    None
7                                                                              98
↪       Don't know    None
```

Transposition

```
>>> ds['q5_trans'].head()
  q5_trans_1  q5_trans_2 q5_trans_3 q5_trans_4 q5_trans_5 q5_trans_97   q5_trans_98
0         5;  1;2;3;4;6;        NaN        NaN        NaN         NaN           NaN
1        NaN         NaN     3;4;5;        NaN     1;2;6;         NaN           NaN
2         5;         NaN        NaN        NaN   1;3;4;6;         NaN            2;
3         3;         NaN         5;        NaN   1;2;4;6;         NaN           NaN
4        NaN         NaN        NaN        NaN        NaN         NaN  1;2;3;4;5;6;
```

```
>>> ds.meta('q5_trans')
delimited set                                                items                item
↪texts codes          texts missing
q5_trans: How likely are you to do each of the ...
1                                                      q5_trans_1  I would refuse if
↪asked     1        Surfing    None
2                                                      q5_trans_2             Very
↪unlikely     2   Snowboarding    None
```

(continues on next page)

(continued from previous page)

```
3                                                    q5_trans_3        Probably wouldn
↪'t     3  Kite boarding    None
4                                                    q5_trans_4  Probably would if␣
↪asked     4    Parachuting    None
5                                                    q5_trans_5              Very␣
↪likely     5    Cave diving    None
6                                                   q5_trans_97  I'm already planning␣
↪to     6    Windsurfing    None
7                                                   q5_trans_98               Don't␣
↪know
```

The method's `ignore_items` and `ignore_values` arguments can pick up `items` (indicated by their order number) and `values` to leave aside during the transposition.

Ignoring items

The new `values` meta's numerical codes will always be enumerated from 1 to the number of valid items for the transposition, so ignoring items 2, 3 and 4 will lead to:

```
>>> ds.transpose('q5', ignore_items=[2, 3, 4])
```

```
>>> ds['q5_trans'].head(1)
  q5_trans_1 q5_trans_2 q5_trans_3 q5_trans_4 q5_trans_5 q5_trans_97 q5_trans_98
0         2;        1;3;        NaN        NaN        NaN         NaN         NaN
```

```
>>> ds.values('q5_trans')
[(1, 'Surfing'), (2, 'Cave diving'), (3, 'Windsurfing')]
```

Ignoring values

```
>>> ds.transpose('q5', ignore_values=[1, 97])
```

```
>>> ds['q5_trans'].head(1)
   q5_trans_2 q5_trans_3 q5_trans_4 q5_trans_5 q5_trans_98
0  1;2;3;4;6;        NaN        NaN        NaN         NaN
```

```
>>> ds.items('q5_trans')
[('q5_trans_2', u'Very unlikely'),
 ('q5_trans_3', u"Probably wouldn't"),
 ('q5_trans_4', u'Probably would if asked'),
 ('q5_trans_5', u'Very likely'),
 ('q5_trans_98', u"Don't know")]
```

Ignoring both items and values

```
>>> ds.transpose('q5', ignore_items=[2, 3, 4], ignore_values=[1, 97])
```

```
>>> ds['q5_trans'].head(1)
  q5_trans_2 q5_trans_3 q5_trans_4 q5_trans_5 q5_trans_98
0       1;3;        NaN        NaN        NaN         NaN
```

```
>>> ds.meta('q5_trans')
delimited set                                          items               item␣
↪texts codes        texts missing
q5_trans: How likely are you to do each of the ...
```

(continues on next page)

(continued from previous page)

```
1                                                    q5_trans_2            Very␣
↪unlikely     1      Surfing    None
2                                                    q5_trans_3        Probably wouldn
↪'t     2  Cave diving    None
3                                                    q5_trans_4  Probably would if␣
↪asked     3  Windsurfing    None
4                                                    q5_trans_5              Very␣
↪likely
5                                                   q5_trans_98               Don't␣
↪know
```

Flatten item answers

- `flatten()`

4.7 Logic and set operaters

4.7.1 Ranges

The `frange()` function takes a string of abbreviated ranges, possibly delimited by a comma (or some other character) and extrapolates its full, unabbreviated list of ints.

```
>>> from quantipy.core.tools.dp.prep import frange
```

Basic range:

```
>>> frange('1-5')
[1, 2, 3, 4, 5]
```

Range in reverse:

```
>>> frange('15-11')
[15, 14, 13, 12, 11]
```

Combination:

```
>>> frange('1-5,7,9,15-11')
[1, 2, 3, 4, 5, 7, 9, 15, 14, 13, 12, 11]
```

May include spaces for clarity:

```
>>> frange('1-5, 7, 9, 15-11')
[1, 2, 3, 4, 5, 7, 9, 15, 14, 13, 12, 11]
```

4.7.2 Complex logic

Multiple conditions can be combined using `union` or `intersection` set statements. Logical mappers can be arbitrarily nested as long as they are well-formed.

4.7.2.1 `union`

`union` takes a list of logical conditions that will be treated with **or** logic.

Where **any** of logic_A, logic_B **or** logic_C are `True`:

```
>>> union([logic_A, logic_B, logic_C])
```

4.7.2.2 intersection

`intersection` takes a list of conditions that will be treated with **and** logic.

Where **all** of logic_A, logic_B **and** logic_C are `True`:

```
>>> intersection([logic_A, logic_B, logic_C])
```

4.7.2.3 "List" logic

Instead of using the verbose `has_any` operator, we can express simple, non-nested *or* logics simply as a list of codes. For example `{"q1_1": [1, 2]}` is an example of list-logic, where `[1, 2]` will be interpreted as `has_any([1, 2])`, meaning if **q1_1** has any of the values **1** or **2**.

`q1_1` has any of the responses 1, 2 or 3:

```
>>> l = {"q1_1": [1, 2, 3]}
```

4.7.2.4 has_any

`q1_1` has any of the responses 1, 2 or 3:

```
>>> l = {"q1_1": has_any([1, 2, 3])}
```

`q1_1` has any of the responses 1, 2 or 3 and no others:

```
>>> l = {"q1_1": has_any([1, 2, 3], exclusive=True)}
```

4.7.2.5 not_any

`q1_1` doesn't have any of the responses 1, 2 or 3:

```
>>> l = {"q1_1": not_any([1, 2, 3])}
```

`q1_1` doesn't have any of the responses 1, 2 or 3 but has some others:

```
>>> l = {"q1_1": not_any([1, 2, 3], exclusive=True)}
```

4.7.2.6 has_all

`q1_1` has all of the responses 1, 2 and 3:

```
>>> l = {"q1_1": has_all([1, 2, 3])}
```

`q1_1` has all of the responses 1, 2 and 3 and no others:

```
>>> l = {"q1_1": has_all([1, 2, 3], exclusive=True)}
```

4.7.2.7 not_all

q1_1 doesn't have all of the responses 1, 2 and 3:

```
>>> l = {"q1_1": not_all([1, 2, 3])}
```

q1_1 doesn't have all of the responses 1, 2 and 3 but has some others:

```
>>> l = {"q1_1": not_all([1, 2, 3], exclusive=True)}
```

4.7.2.8 has_count

q1_1 has exactly 2 responses:

```
>>> l = {"q1_1": has_count(2)}
```

q1_1 has 1, 2 or 3 responses:

```
>>> l = {"q1_1": has_count([1, 3])}
```

q1_1 has 1 or more responses:

```
>>> l = {"q1_1": has_count([is_ge(1)])}
```

q1_1 has 1, 2 or 3 responses from the response group 5, 6, 7, 8 or 9:

```
>>> l = {"q1_1": has_count([1, 3, [5, 6, 7, 8, 9]])}
```

q1_1 has 1 or more responses from the response group 5, 6, 7, 8 or 9:

```
>>> l = {"q1_1": has_count([is_ge(1), [5, 6, 7, 8, 9]])}
```

4.7.2.9 not_count

q1_1 doesn't have exactly 2 responses:

```
>>> l = {"q1_1": not_count(2)}
```

q1_1 doesn't have 1, 2 or 3 responses:

```
>>> l = {"q1_1": not_count([1, 3])}
```

q1_1 doesn't have 1 or more responses:

```
>>> l = {"q1_1": not_count([is_ge(1)])}
```

q1_1 doesn't have 1, 2 or 3 responses from the response group 5, 6, 7, 8 or 9:

```
>>> l = {"q1_1": not_count([1, 3, [5, 6, 7, 8, 9]])}
```

q1_1 doesn't have 1 or more responses from the response group 5, 6, 7, 8 or 9:

```
>>> l = {"q1_1": not_count([is_ge(1), [5, 6, 7, 8, 9]])}
```

4.7.3 Boolean slicers and code existence

`any()`,`all()` `code_count()`,`is_nan()`

4.8 Custom data recoding

4.8.1 The `recode()` method in detail

This function takes a mapper of `{key: logic}` entries and injects the key into the target column where its paired logic is True. The logic may be arbitrarily complex and may refer to any other variable or variables in data. Where a pre-existing column has been used to start the recode, the injected values can replace or be appended to any data found there to begin with. Note that this function does not edit the target column, it returns a recoded copy of the target column. The recoded data will always comply with the column type indicated for the target column according to the meta.

method `recode(target, mapper, default=None, append=False, intersect=None, initialize=None, fillna=None, inplace=True)`

4.8.1.1 `target`

`target` controls which column meta should be used to control the result of the recode operation. This is important because you cannot recode multiple responses into a 'single'-typed column.

The `target` column **must** already exist in meta.

The `recode` function is effectively a request to return a copy of the `target` column, recoded as instructed. `recode` does not edit the `target` column in place, it returns a recoded copy of it.

If the `target` column does not already exist in `data` then a new series, named accordingly and initialized with `np.NaN`, will begin the recode.

Return a recoded version of the column `radio_stations_xb` edited based on the given mapper:

```
>>> recoded = recode(
...     meta, data,
...     target='radio_stations_xb',
...     mapper=mapper
... )
```

By default, recoded data resulting from the the mapper will replace any data already sitting in the target column (on a cell-by-cell basis).

4.8.1.2 `mapper`

A mapper is a dict of `{value: logic}` entries where value represents the data that will be injected for cases where the logic is True.

Here's a simplified example of what a mapper looks like:

```
>>> mapper = {
...     1: logic_A,
...     2: logic_B,
...     3: logic_C,
... }
```

1 will be generated where `logic_A` is `True`, 2 where `logic_B` is `True` and 3 where `logic_C` is `True`.

The recode function, by referencing the type indicated by the meta, will manage the complications involved in single vs delimited set data.

```
>>> mapper = {
...     901: {'radio_stations': frange('1-13')},
...     902: {'radio_stations': frange('14-20')},
...     903: {'radio_stations': frange('21-25')}
... }
```

This means: inject 901 if the column `radio_stations` has any of the values 1-13, 902 where `radio_stations` has any of the values 14-20 and 903 where `radio_stations` has any of the values 21-25.

4.8.1.3 `default`

If you had lots of values to generate from the same reference column (say most/all of them were based on `radio_stations`) then we can omit the wildcard logic format and use recode's default parameter.

```
>>> recoded = recode(
...     meta, data,
...     target='radio_stations_xb',
...     mapper={
...         901: frange('1-13'),
...         902: frange('14-20'),
...         903: frange('21-25')
...     },
...     default='radio_stations'
... )
```

This means, all unkeyed logic will default to be keyed to `radio_stations`. In this case the three codes 901, 902 and 903 will be generated based on the data found in `radio_stations`.

You can combine this with reference to other columns, but you can only provide one default column.

```
>>> recoded = recode(
...     meta, data,
...     target='radio_stations_xb',
...     mapper={
...         901: frange('1-13'),
...         902: frange('14-20'),
...         903: frange('21-25'),
...         904: {'age': frange('18-34')}
...     },
...     default='radio_stations'
... )
```

Given that logic can be arbitrarily complicated, mappers can be as well. You'll see an example of a mapper that recodes a segmentation in **Example 4**, below.

4.8.1.4 append

If you want the recoded data to be appended to whatever may already be in the target column (this is only applicable for 'delimited set'-typed columns), then you should use the append parameter.

```
>>> recoded = recode(
...     meta, data,
...     target='radio_stations_xb',
...     mapper=mapper,
...     append=True
... )
```

The precise behaviour of the append parameter can be seen in the following examples.

Given the following data:

```
>>> df['radio_stations_xb']
1    6;7;9;13;
2          97;
3          97;
4    13;16;18;
5         2;6;
Name: radio_stations_xb, dtype: object
```

We generate a recoded value of 901 if any of the values 1-13 are found. With the default `append=False` behaviour we will return the following:

```
>>> target = 'radio_stations_xb'
>>> recode(meta, data, target, mapper)
1    901;
2     97;
3     97;
4    901;
5    901;
Name: radio_stations_xb, dtype: object
```

However, if we instead use `append=True`, we will return the following:

```
>>> target = 'radio_stations_xb'
>>> recode(meta, data, target, mapper, append=True)
1    6;7;9;13;901;
2              97;
3              97;
4    13;16;18;901;
5         2;6;901;
Name: radio_stations_xb, dtype: object
```

4.8.1.5 intersect

One way to help simplify complex logical conditions, especially when they are in some way repetitive, is to use `intersect`, which accepts any logical statement and forces every condition in the mapper to become the intersection of both it and the intersect condition.

For example, we could limit our recode to males by giving a logical condition to that effect to `intersect`:

```
>>> recoded = recode(
...     meta, data,
...     target='radio_stations_xb',
...     mapper={
...         901: frange('1-13'),
...         902: frange('14-20'),
...         903: frange('21-25'),
...         904: {'age': frange('18-34')}
...     },
...     default='radio_stations',
...     intersect={'gender': [1]}
... )
```

4.8.1.6 initialize

You may also `initialize` your copy of the target column as part of your recode operation. You can `initalize` with either np.NaN (to overwrite anything that may already be there when your recode begins) or by naming another column. When you name another column a copy of the data from that column is used to initialize your recode.

Initialization occurs **before** your recode.

```
>>> recoded = recode(
...     meta, data,
...     target='radio_stations_xb',
...     mapper={
...         901: frange('1-13'),
...         902: frange('14-20'),
...         903: frange('21-25'),
...         904: {'age': frange('18-34')}
...     },
...     default='radio_stations',
...     initialize=np.NaN
... )
```

```
>>> recoded = recode(
...     meta, data,
...     target='radio_stations_xb',
...     mapper={
...         901: frange('1-13'),
...         902: frange('14-20'),
...         903: frange('21-25'),
...         904: {'age': frange('18-34')}
...     },
...     default='radio_stations',
...     initialize='radio_stations'
... )
```

4.8.1.7 fillna

You may also provide a `fillna` value that will be used as per `pd.Series.fillna()` **after** the recode has been performed.

```
>>> recoded = recode(
...     meta, data,
```

(continues on next page)

(continued from previous page)

```
...     target='radio_stations_xb',
...     mapper={
...         901: frange('1-13'),
...         902: frange('14-20'),
...         903: frange('21-25'),
...         904: {'age': frange('18-34')}
...     },
...     default='radio_stations',
...     initialize=np.NaN,
...     fillna=99
... )
```

4.8.2 Custom recode examples

4.8.2.1 Building a net code

Here's an example of copying an existing question and recoding onto it a net code.

Create the new metadata:

```
>>> meta['columns']['radio_stations_xb'] = copy.copy(
...     meta['columns']['radio_stations']
... )
>>> meta['columns']['radio_stations_xb']['values'].append(
...     {
...         "value": 901,
...         "text": {"en-GB": "NET: Listened to radio in past 30 days"}
...     }
... )
```

Initialize the new column. In this case we're starting with a copy of the `radio_stations` column:

```
>>> data['radio_stations_xb'] = data['radio_stations'].copy()
```

Recode the new column by appending the code 901 to it as indicated by the mapper:

```
>>> data['radio_stations_xb'] = recode(
...     meta, data,
...     target='radio_stations_xb',
...     mapper={
...         901: {'radio_stations': frange('1-23, 92, 94, 141')}
...     },
...     append=True
... )
```

Check the result:

```
>>> data[['radio_stations', 'radio_stations_xb']].head(20)
   radio_stations radio_stations_cb
0              5;            5;901;
1             97;               97;
2             97;               97;
3             97;               97;
4             97;               97;
5              4;            4;901;
```

(continues on next page)

(continued from previous page)

```
6              11;            11;901;
7               4;             4;901;
8              97;               97;
9              97;               97;
10             97;               97;
11             92;            92;901;
12             97;               97;
13       1;13;17;       1;13;17;901;
14              6;             6;901;
15      1;5;6;10;      1;5;6;10;901;
16              6;             6;901;
17        2;4;16;        2;4;16;901;
18          6;10;          6;10;901;
19              6;             6;901;
```

4.8.2.2 Create-and-fill

Here's an example where the value 1 is generated based on some logic and then all remaining cases are given the value 2 using the pandas.Series.fillna() method.

Create the new metadata

```
>>> meta['columns']['age_xb'] = {
...     'type': 'single',
...     'text': {'en-GB': 'Age'},
...     'values': [
...         {'value': 1, 'text': {'en-GB': '16-25'}},
...         {'value': 2, 'text': {'en-GB': 'Others'}}
...     ]
...     }
```

Initialize the new column:

```
>>> data['age_xb'] = np.NaN
```

Recode the new column:

```
>>> data['age_xb'] = recode(
...     meta, data,
...     target='age_xb',
...     mapper={
...         1: {'age': frange('16-40')}
...     }
... )
```

Fill all cases that are still empty with the value 2:

```
>>> data['age_xb'].fillna(2, inplace=True)
```

Check the result:

```
>>> data[['age', 'age_xb']].head(20)
    age  age_grp_rc
0    22           1
1    68           2
```

(continues on next page)

(continued from previous page)

```
2    32           1
3    44           2
4    33           1
5    52           2
6    54           2
7    44           2
8    62           2
9    49           2
10   64           2
11   73           2
12   43           2
13   28           1
14   66           2
15   39           1
16   51           2
17   50           2
18   77           2
19   42           2
```

4.8.2.3 Numerical banding

Here's a typical example of recoding age into custom bands.

In this case we're using list comprehension to generate the first ten values objects and then concatenate that with a final '65+' value object which doesn't folow the same label format.

Create the new metadata:

```
>>> meta['columns']['age_xb_1'] = {
...     'type': 'single',
...     'text': {'en-GB': 'Age'},
...     'values': [
...         {
...             'value': i,
...             'text': {'en-GB': '{}-{}'.format(r[0], r[1])}
...         }
...         for i, r in enumerate(
...             [
...                 [18, 20],
...                 [21, 25], [26, 30],
...                 [31, 35], [36, 40],
...                 [41, 45], [46, 50],
...                 [51, 55], [56, 60],
...                 [61, 65]
...             ],
...             start=1
...         )
...     ] + [
...         {
...             'value': 11,
...             'text': {'en-GB': '65+'}
...         }
...     ]
... }
```

Initialize the new column:

```
>>> data['age_xb_1'] = np.NaN
```

Recode the new column:

```
>>> data['age_xb_1'] = recode(
...     meta, data,
...     target='age_xb_1',
...     mapper={
...         1: frange('18-20'),
...         2: frange('21-25'),
...         3: frange('26-30'),
...         4: frange('31-35'),
...         5: frange('36-40'),
...         6: frange('41-45'),
...         7: frange('46-50'),
...         8: frange('51-55'),
...         9: frange('56-60'),
...         10: frange('61-65'),
...         11: frange('66-99')
...     },
...     default='age'
... )
```

Check the result:

```
>>> data[['age', 'age_xb_1']].head(20)
    age  age_cb
0    22       2
1    68      11
2    32       4
3    44       6
4    33       4
5    52       8
6    54       8
7    44       6
8    62      10
9    49       7
10   64      10
11   73      11
12   43       6
13   28       3
14   66      11
15   39       5
16   51       8
17   50       7
18   77      11
19   42       6
```

4.8.2.4 Complicated segmentation

Here's an example of using a complicated, nested series of logic statements to recode an obscure segmentation.

The segemenation was given with the following definition:

1 - Self-directed:

- If q1_1 in [1,2] and q1_2 in [1,2] and q1_3 in [3,4,5]

2 - Validators:

- If q1_1 in [1,2] and q1_2 in [1,2] and q1_3 in [1,2]

3 - Delegators:

- If (q1_1 in [3,4,5] and q1_2 in [3,4,5] and q1_3 in [1,2])
- Or (q1_1 in [3,4,5] and q1_2 in [1,2] and q1_3 in [1,2])
- Or (q1_1 in [1,2] and q1_2 in [3,4,5] and q1_3 in [1,2])

4 - Avoiders:

- If (q1_1 in [3,4,5] and q1_2 in [3,4,5] and q1_3 in [3,4,5])
- Or (q1_1 in [3,4,5] and q1_2 in [1,2] and q1_3 in [3,4,5])
- Or (q1_1 in [1,2] and q1_2 in [3,4,5] and q1_3 in [3,4,5])

5 - Others:

- Everyone else.

Create the new metadata:

```
>>> meta['columns']['segments'] = {
...     'type': 'single',
...     'text': {'en-GB': 'Segments'},
...     'values': [
...         {'value': 1, 'text': {'en-GB': 'Self-directed'}},
...         {'value': 2, 'text': {'en-GB': 'Validators'}},
...         {'value': 3, 'text': {'en-GB': 'Delegators'}},
...         {'value': 4, 'text': {'en-GB': 'Avoiders'}},
...         {'value': 5, 'text': {'en-GB': 'Other'}},
...     ]
... }
```

Initialize the new column?

```
>>> data['segments'] = np.NaN
```

Create the mapper separately, since it's pretty massive!

See the **Complex logic** section for more information and examples related to the use of `union` and `intersection`.

```
>>> mapper = {
...     1: intersection([
...         {"q1_1": [1, 2]},
...         {"q1_2": [1, 2]},
...         {"q1_3": [3, 4, 5]}
...     ]),
...     2: intersection([
...         {"q1_1": [1, 2]},
...         {"q1_2": [1, 2]},
...         {"q1_3": [1, 2]}
...     ]),
...     3: union([
...         intersection([
...             {"q1_1": [3, 4, 5]},
...             {"q1_2": [3, 4, 5]},
...             {"q1_3": [1, 2]}
...         ]),
```

(continues on next page)

(continued from previous page)

```
...             intersection([
...                 {"q1_1": [3, 4, 5]},
...                 {"q1_2": [1, 2]},
...                 {"q1_3": [1, 2]}
...             ]),
...             intersection([
...                 {"q1_1": [1, 2]},
...                 {"q1_2": [3, 4, 5]},
...                 {"q1_3": [1, 2]}
...             ]),
...         ]),
...         4: union([
...             intersection([
...                 {"q1_1": [3, 4, 5]},
...                 {"q1_2": [3, 4, 5]},
...                 {"q1_3": [3, 4, 5]}
...             ]),
...             intersection([
...                 {"q1_1": [3, 4, 5]},
...                 {"q1_2": [1, 2]},
...                 {"q1_3": [3, 4, 5]}
...             ]),
...             intersection([
...                 {"q1_1": [1, 2]},
...                 {"q1_2": [3, 4, 5]},
...                 {"q1_3": [3, 4, 5]}
...             ])
...         ])
... }
```

Recode the new column:

```
>>> data['segments'] = recode(
...     meta, data,
...     target='segments',
...     mapper=mapper
... )
```

Note: Anything not at the top level of the mapper will not benefit from using the `default` parameter of the recode function. In this case, for example, saying `default='q1_1'` would not have helped. Everything in a nested level of the mapper, including anything in a `union` or `intersection` list, must use the explicit dict form `{"q1_1": [1, 2]}`.

Fill all cases that are still empty with the value 5:

```
>>> data['segments'].fillna(5, inplace=True)
```

Check the result:

```
>>> data[['q1_1', 'q1_2', 'q1_3', 'segments']].head(20)
    q1_1  q1_2  q1_3  segments
0      3     3     3         4
1      3     3     3         4
2      1     1     3         1
```

(continues on next page)

(continued from previous page)

```
3       1       1       2           2
4       2       2       2           2
5       1       1       5           1
6       2       3       2           3
7       2       2       3           1
8       1       1       4           1
9       3       3       3           4
10      3       3       4           4
11      2       2       4           1
12      1       1       5           1
13      2       2       4           1
14      1       1       1           2
15      2       2       4           1
16      2       2       3           1
17      1       1       5           1
18      5       5       1           3
19      1       1       4           1
```

4.8.2.5 Variable creation

4.8.2.6 Adding derived variables

4.8.2.7 Interlocking variables

4.8.2.8 Condition-based code removal

CHAPTER 5

Weights

5.1 Background and methodology

`quantipy` utilizes the *Rim* (sometimes also called *Raking*) weighting method, an iterative fitting algorithm that tries to balance out multiple sample frequencies simultaneously. It is rooted in the mathematical model developed in the seminal academic paper by Deming/Stephan (1940) ([DeSt40]). The following chapters draw heavily from it.

5.1.1 The statistical problem

More often than not, market research professionals (and not only them!) are required to weight their raw data collected via a survey to match a known specific real-world distribution. This is the case when you try to weight your sample to reflect the population distribution of a certain characteristic to make it "representative" in one or more terms. Leaving unconsidered what a "representative" sample actually is in the first place, let's see what "weighting data" comes down to and why weighting in order to achieve representativeness can be quite a difficult task. Look at the following two examples:

1. Your data contains an equal number of male and female respondents while in the real world you know that women are a little bit more frequent than men. In relative terms you have sampled 2 percentage points more men than women:

	Sample (N=100)	Population	Factors
Men	50 %	48%	48/50 = 0.96
Women	50%	52%	52/50 = 1.04

That one is easy because you know each cell's population frequencies and can simply find the factors that will correct your sample to mirror the real-world population. To weight you would simply compute the relevant factors by dividing the desired population figure by the sample frequency and assign each case in your data the respective result (based on his or her gender). The factors are coming from your **one-dimensional** weighting matrix above.

2. You have a survey project that requires the sample to match the gender and age distributions in real-world Germany and additionally should take into account the distribution of iPad owners and the population frequencies of the federal states.

Again, to weight the data you would need to calculate the cell ratios of target vs. sample figures for the different sample characteristics. While you may be able to find the **joint** distribution of age categories by gender, you will have a hard time coming up e.g. with the correct figures for a **joint** distribution of iPad owners per federal state by gender and age group.

To put it differently: You will not know the population's cell target figures for all weighting dimensions in all relevant cells of the **multi-dimensional** weighting matrix. Since you need this information to assign each case a weight factor to come up with the correct weighted distributions for the four sample characteristics you would not be able to weight the data. To illustrate the complexity of such a weighting scheme, the table below should suit:

State:	Bavaria						Saxony					
Age:	18-25		26-35		36-55		18-25		26-35		36-55	...
Gender:	m	f	m	f	m	f	m	f	m	f	m	f ...
iPad	?	?	?	?	?	?	?	?	?	?	?	?
no iPad	?	?	?	?	?	?	?	?	?	?	?	?

Note that you would also need to take into account the other joint distributions of age by gender per federal state, iPad owners by age, and so on to get the correct weight factors step by step: all cross-tabulation information for the population that will not be available to you. Additionally, even if you would have all the information necessary for your calculations, try to imagine the amount of work that awaits to come up with the weight factors per cell regarding getting all possible combinations right, then creating variables, recoding those variables and then finally computing the ratios.

What is available regularly, however, is the distribution of people living in Germany's federal states and the distribution of iPad owners in general (as per "Yes, have one," "do not own one"), plus the age and gender frequencies. This is where rim weighting comes into play.

5.1.2 Rim weighting concept

Rim weighting in short can be described as an **iterative data fitting** process that aims to apply a weight factor to each respondent's case record in order to match the target figures by altering the sample cell frequencies relevant to the weighting matrix. Doing that, it will find the single cell's ratios that are required to come up with the correct targets per weight dimension – it will basically **estimate** all the joint distribution information that is unknown.

The way this works can be summarized as follows: For each interlocking cell coming from all categories of all the variables that are given to weight to, an algorithm will compute the proportion necessary in a single specific cell that, when summed over per column or respectively by row, will result in a column (row) total per category that matches the target distribution. However, it will occur that having balanced a column total to match, the row totals will be off. This is where one iteration ends and another one begins starting now with the weighted values from the previous run. This iterative process will continue until a satisfying result in terms of an acceptable low amount of mismatch between produced sample results and weight targets is reached.

In short: Simultaneous adjustment of all weight variables with the smallest amount of data manipulation possible while forcing the maximum match between sample and weight scheme.

References

5.2 Weight scheme setup

5.2.1 Using the `Rim` class

The `Rim` object's purpose is to define the required setup of the weighting process, i.e. the *weight scheme* that should be used to compute the actual factor results per case in the dataset. While its main purpose is to provide a simple interface to structure weight schemes of all complexities, it also offers advanced options that control the underlying weighting algorithm itself and thus might impact the results.

To start working with a `Rim` object, we only need to think of a name for our scheme:

```
>>> scheme = qp.Rim('my_first_scheme')
```

5.2.2 Target distributions

A major and (probably the most important) step in specifying a weight scheme is mapping the desired target population proportions to the categories of the related variables inside the data. This is done via a `dict` mapping.

For example, to equally weight female and male respondents in our sample, we simply define:

```
>>> gender_targets = {}
>>> gender_targets['gender'] = {1: 50.0, 2: 50.0}
>>> gender_targets
{'gender': {1: 50.0, 2: 50.0}}
```

Since we are normally dealing with multiple variables at once, we collect them in a `list`, adding other variables naturally in the same way:

```
>>> dataset.band('age', [(19, 25), (26-35), (36, 49)])
>>> age_targets = {'age_banded': {1: 45.0, 2: 29.78, 3: 25.22}}
>>> all_targets = [gender_targets, age_targets]
```

The `set_targets()` method can now use the `all_targets` list to apply the target distributions to the `Rim` weight scheme setup (we are also providing an optional name for our group of variables) .

```
>>> scheme.set_targets(targets=all_targets, group_name='basic weights')
```

The `Rim` instance also allows inspecting these targets from itself now (you can see the `group_name` parameter reflected here, it would fall back to `'_default_name_'` if none was provided):

```
>>> scheme.groups['basic weights']['targets']
[{'gender': {1: 50.0, 2: 50.0}}, {'age_banded': {1: 45.0, 2: 29.78, 3: 25.22}}]
```

5.2.3 Weight groups and filters

For more elaborate weight schemes, we are instead using the `add_group()` method which is effectively a generalized version of `set_targets()` that supports addressing subsets of the data by filtering. For example, differing target distributions (or even the scheme defining variables of interest) might be required across several market segments or between survey periods.

We can illustrate this using the variable `'Wave'` from the dataset:

```
>>> dataset.crosstab('Wave', text=True, pct=True)
Question          Wave. Wave
Values                   @
Question   Values
Wave. Wave All         100.0
           Wave 1       19.6
           Wave 2       20.2
           Wave 3       20.5
           Wave 4       19.8
           Wave 5       19.9
```

Let's assume we want to use the original targets for the first three waves but the remaining two waves need to reflect some changes in both gender and the age distributions. We first define a new set of targets that should apply only to the waves 4 and 5:

```
gender_targets_2 = {'gender': {1: 30.0, 2: 70.0}}
age_targets_2 = {'age_banded': {1: 35.4, 2: 60.91, 3: 3.69}}
all_targets_2 = [gender_targets_2, age_targets_2]
```

We then set the filter expressions for the respective subsets of the data, as per:

```
>>> filter_wave1 = 'Wave == 1'
>>> filter_wave2 = 'Wave == 2'
>>> filter_wave3 = 'Wave == 3'
>>> filter_wave4 = 'Wave == 4'
>>> filter_wave5 = 'Wave == 5'
```

And add our weight specifications accordingly:

```
>>> scheme = qp.Rim('my_complex_scheme')
>>> scheme.add_group(name='wave 1', filter_def=filter_wave1, targets=all_targets)
>>> scheme.add_group(name='wave 2', filter_def=filter_wave2, targets=all_targets)
>>> scheme.add_group(name='wave 3', filter_def=filter_wave3, targets=all_targets)
>>> scheme.add_group(name='wave 4', filter_def=filter_wave4, targets=all_targets_2)
>>> scheme.add_group(name='wave 5', filter_def=filter_wave5, targets=all_targets_2)
```

Note: For historical reasons, the *logic operators* currently **do not** work within the `Rim` module. This means that all filter definitions need to be valid string expressions suitable for the `pandas.DataFrame.query()` method. We are planning to abandon this limitation as soon as possible to enable easier and more complex filters that are consistent with the rest of the library.

5.2.4 Setting group targets

At this stage it might also be needed to balance out the survey waves themselves in a certain way, e.g. make each wave count exactly the same (as you can see above each wave accounts for roughly 20% of the full sample but not quite exactly).

With `Rim.group_targets()` we can apply an **outer** weighting to the **between** group distribution while keeping the already set **inner** target proportions **within** each of them. Again we are using a `dict`, this time mapping the group names from above to the desired outcome percentages:

```
>>> group_targets = {'wave 1': 20.0,
...                  'wave 2': 20.0,
```

(continues on next page)

(continued from previous page)

```
...                  'wave 3': 20.0,
...                  'wave 4': 20.0,
...                  'wave 5': 20.0}
```

```
>>> scheme.group_targets(group_targets)
```

To sum it up: Our weight scheme consists of five groups based on `'Wave'` that resp. need to match two different sets of target distributions on the `'gender'` and `'age_banded'` variables with each group coming out as 20% of the full sample.

5.3 Integration within `DataSet`

The computational core of the weighting algorithm is the `quantipy.core.weights.rim.Rake` class which can be accessed by working with `qp.WeightEngine()`, but it is much easier to directly use the `DataSet.weight()` method. Its full signature looks as follows:

```
DataSet.weight(weight_scheme,
               weight_name='weight',
               unique_key='identity',
               subset=None,
               report=True,
               path_report=None,
               inplace=True,
               verbose=True)
```

5.3.1 Weighting and weighted aggregations

As can been seen, we can simply provide our weight scheme `Rim` instance to the method. Since the dataset already contains a variable called `'weight'` (and we do not want to overwrite that one) we set `weight_name` to be `'weights_new'`. We also need to set `unique_key='unique_id'` as that is our identifying key variable (that is needed to map the weight factors back into our dataset):

```
>>> dataset.weight(scheme, weight_name='weights_new', unique_key='unique_id')
```

Before we take a look at the report that is printed (because of `report=True`), we want to manually check our results. For that, we can simply analyze some cross- tabulations, weighted by our new weights! For a start, we check if we arrived at the desired proportions for `'gender'` and `'age_banded'` per `'Wave'`:

```
>>> dataset.crosstab(x='gender', y='Wave', w='weights_new', pct=True)
Question                              Wave. Wave
Values                                      All Wave 1 Wave 2 Wave 3 Wave 4 Wave 5
Question                     Values
gender. What is your gender? All         100.0  100.0  100.0  100.0  100.0  100.0
                             Male         42.0   50.0   50.0   50.0   30.0   30.0
                             Female       58.0   50.0   50.0   50.0   70.0   70.0
```

```
>>> dataset.crosstab(x='age_banded', y='Wave', w='weights_new', pct=True,
...                  decimals=2)
Question                Wave. Wave
Values                        All  Wave 1  Wave 2  Wave 3  Wave 4  Wave 5
Question        Values
```

(continues on next page)

(continued from previous page)

```
age_banded. Age All        100.00  100.00  100.00  100.00  100.00  100.00
               19-25        41.16   45.00   45.00   45.00   35.40   35.40
               26-35        42.23   29.78   29.78   29.78   60.91   60.91
               36-49        16.61   25.22   25.22   25.22    3.69    3.69
```

Both results accurately reflect the desired proportions from our scheme. And we can also verify the weighted distribution of `'Wave'`, now completely balanced:

```
>>> dataset.crosstab(x='Wave', w='weights_new', pct=True)
Question          Wave. Wave
Values                   @
Question   Values
Wave. Wave All         100.0
           Wave 1       20.0
           Wave 2       20.0
           Wave 3       20.0
           Wave 4       20.0
           Wave 5       20.0
```

5.3.2 The isolated weight dataframe

By default, the weighting operates `inplace`, i.e. the weight vector will be placed into the `DataSet` instance as a regular `columns` element:

```
>>> dataset.meta('weights_new')
                                      float
weights_new: my_first_scheme weights   N/A
```

```
>>> dataset['weights_new'].head()
   unique_id  weights_new
0     402891     0.885593
1   27541022     1.941677
2     335506     0.984491
3   22885610     1.282057
4     229122     0.593834
```

It is also possible to return a new `pd.DataFrame` that contains all relevant `Rim` scheme variables incl. the factor vector for external use cases or further analysis:

```
>>> wdf = dataset.weight(scheme, weight_name='weights_new', unique_key='unique_id',
                         inplace=False)
>>> wdf.head()
   unique_id  gender  age_banded  weights_new  Wave
0     402891       1         1.0     0.885593     4
1   27541022       2         1.0     1.941677     1
2     335506       1         2.0     0.984491     3
3   22885610       1         2.0     1.282057     5
4     229122       1         3.0     0.593834     1
```

5.4 Diagnostics

We did not yet take a look at the default weight report that offers some additional information on the weighting outcome results and the even the algorithm process itself (the report lists the internal weight variable name that is always just a

suffix of the scheme name):

```
Weight variable       weights_my_complex_scheme
Weight group                             wave 1       wave 2       wave 3       wave␣
↪4       wave 5
Weight filter                          Wave == 1    Wave == 2    Wave == 3    Wave ==␣
↪4    Wave == 5
Total: unweighted                   1621.000000  1669.000000  1689.000000  1637.
↪000000  1639.000000
Total: weighted                     1651.000000  1651.000000  1651.000000  1651.
↪000000  1651.000000
Weighting efficiency                  74.549628    78.874120    77.595143    53.
↪744060    50.019937
Iterations required                   13.000000     8.000000    11.000000    12.
↪000000    10.000000
Mean weight factor                     1.018507     0.989215     0.977501     1.
↪008552     1.007322
Minimum weight factor                  0.513928     0.562148     0.518526     0.
↪053652     0.050009
Maximum weight factor                  2.243572     1.970389     1.975681     2.
↪517704     2.642782
Weight factor ratio                    4.365539     3.505106     3.810189    46.
↪926649    52.846124
```

5.4.1 The weighting efficiency

After all, getting the sample to match to the desired population proportions **always** comes at a cost. This cost is captured in a statistical measure called the **weighting efficiency** and is featured in the report as well. It is a metric for evaluation of the sample vs. targets match, i.e. the sample balance compared to the weight scheme. You can also inversely view it as the amount of distortion that was needed to arrive at the weighted figures, that is, how much the data is manipulated by the weighting. A **low** efficiency indicates a **larger** bias introduced by the weights.

Let w denote our weight vector containing the factor for each i respondent, then the mathematical definititon of the (total) weighting efficiency we is:

$$we = \frac{\frac{[\sum w_i]^2}{\sum i}}{\sum w_i^2} * 100$$

Which is the quotient of the squared sum of weights and the number of cases divided by the sum of squared weights (expressed as a percentage).

We can manually check the figure for group `'wave 1'`. We first recreate the filter that has been used, which we can also derive the number of cases `n` from:

```
>>> f = dataset.take({'Wave': [1]})
>>> n = len(f)
>>> n
1621
```

The sum of weights squared `sws` is then:

```
>>> sws = (dataset[f, 'weights_new'].sum()) ** 2
>>> sws
2725801.0
```

And the sum of squared weights `ssw`:

```
>>> ssw = (dataset[f, 'weights_new']**2).sum()
>>> ssw
2255.61852968
```

Which enables us to calculate the weighting efficiency `we` as per:

```
>>> we = (sws / n) / ssw * 100
>>> we
74.5496275503
```

Generally, weighting efficiency results below the 80% mark indicate a high sample vs. population mismatch. Dropping below 70% should be a reason to reexamine the weight scheme specifications or analysis design.

To better understand why the weighting efficiency is good for judging the quality of the weighting, we can look at its relation to the **effective sample size** (the effective base). In our example, the effective base of the weight group would be around 0.75 * 1621 = 1215.75. This means that we are dealing with an effective sample of only 1216 cases for weighted statistical analysis and inference. In other words, the weighting reduces the reliability of the sample as if we had sampled roughly 400 (about 25%) respondents less.

5.4.2 Gotchas

[A] Subsets and targets

In the example we have defined five weight groups, one for each of the waves, although we only had two differing sets of targets we wanted to match. One could be tempted to only set two weight groups because of this, using the filters:

```
>>> f1 = 'Wave in [1, 2, 3]'
```

and

```
>>> f2 = 'Wave in [4, 5]'
```

It is crucial to remember that the algorithm is applied on the weight group's overall data base, i.e. the above definition would achieve the targets inside the two groups (Waves 1/2/3 and Waves 4/5) and **not within** each of the waves.

CHAPTER 6

Batch

qp.Batch is a subclass of qp.DataSet and is a container for structuring a qp.Link collection's specifications.

qp.Batch is not only a subclass of qp.DataSet, it also takes a DataSet instance as input argument, inheriting a few of its attributes, e.g. _meta, _data, valid_tks and text_key. All other Batch attributes are used as construction plans for populating a qp.Stack, these get stored in the belonging DataSet meta component in _meta['sets']['batches'][batchname].

In general, it does not matter in which order Batch attributes are set by methods, the class ensures that all attributes are kept consistent.

All next sections are working with the following qp.DataSet instance:

```
import quantipy as qp

dataset = qp.DataSet('Example Data (A)')
dataset.read_quantipy('Example Data (A).json', 'Example Data (A).csv')
```

The json and csv files you can find in quantipy/tests.

6.1 Creating/ Loading a qp.Batch instance

As mentioned, a Batch instance has a close connection to its belonging DataSet instance and we can easily create a new Batch from a DataSet as per:

```
batch1 = dataset.add_batch(name='batch1')
batch2 = dataset.add_batch(name='batch2', ci=['c'], weights='weight')
```

It is also possible to load an already existing instance out of the meta stored in dataset._meta['sets']['batches']:

```
batch = dataset.get_batch('batch1')
```

Both methods, `.add_batch()` and `.get_batch()`, are an easier way to use the `__init__()` method of `qp.Batch`.

An other way to get a new `qp.Batch` instance is to copy an existing one, in that case all added open ends are removed from the new instance:

```
copy_batch = batch.copy('copy_of_batch1')
```

6.2 Adding variables to a `qp.Batch` instance

6.2.1 x-keys and y-keys

The included variables in a `Batch` constitute the main structure for the `qp.Stack` construction plan. Variables can be added as x-keys or y-keys, for arrays all belonging items are automatically added and the `qp.Stack` gets populated with all cross-tabulations of these keys:

```
>>> batch.add_x(['q1', 'q2', 'q6'])
>>> batch.add_y(['gender', 'q1'])
Array summaries setup: Creating ['q6'].
```

x-specific y-keys can be produced by manipulating the main y-keys, this edit can be extending or replacing the existing keys:

```
>>> batch.extend_y(['locality', 'ethnicity'], on=['q1'])
>>> batch.replace_y(['locality', 'ethnicity'], on=['q2'])
```

With these settings the construction plan looks like that:

```
>>> print batch.x_y_map
OrderedDict([('q1', ['@', 'gender', 'q1', 'locality', 'ethnicity']),
             ('q2', ['locality', 'ethnicity']),
             ('q6', ['@']),
             (u'q6_1', ['@', 'gender', 'q1']),
             (u'q6_2', ['@', 'gender', 'q1']),
             (u'q6_3', ['@', 'gender', 'q1'])])
```

6.2.2 Arrays

A special case exists if the added variables contain arrays. As default for all arrays in x-keys array summaries are created (array as x-key and `'@'`-referenced total as y-key), see the output below (`Array summaries setup: Creating ['q6'].`). If array summaries are requested only for a selection of variables or for none, use `.make_summaries()`:

```
>>> batch.make_summaries(None)
Array summaries setup: Creating no summaries!
```

Arrays can also be transposed (`'@'`-referenced total as x-key and array name as y-key). If they are not in the batch summary list before, they are automatically added and depending on the `replace` parameter only the transposed or both types of summaries are added to the `qp.Stack`:

```
>>> batch.transpose_array('q6', replace=False)
Array summaries setup: Creating ['q6'].
```

The construction plan now shows that both summary types are included:

```
>>> print batch.x_y_map
OrderedDict([('q1', ['@', 'gender', 'q1', 'locality', 'ethnicity']),
             ('q2', ['locality', 'ethnicity']),
             ('q6', ['@']),
             ('@', ['q6']),
             (u'q6_1', ['@', 'gender', 'q1']),
             (u'q6_2', ['@', 'gender', 'q1']),
             (u'q6_3', ['@', 'gender', 'q1'])])
```

6.2.3 Verbatims/ open ends

Another special case are verbatims. They will not be aggregated in a `qp.Stack`, but they have to be defined in a `qp.Batch` to add them later to a `qp.Cluster`.

There are two different ways to add verbatims: Either all to one `qp.Cluster` key or each gets its own key. But both options can be done with the same method.

For splitting the verbatims, set `split=True` and insert as many titles as included verbatims/ open ends:

```
>>> batch.add_open_ends(['q8a', 'q9a'], break_by=['record_number', 'age'],
                                                split=True, title=['oe_q8', 'oe_q9'])
```

For collecting all verbatims in one Cluster key, set `split=False` and add only one `title` or use the default parameters:

```
>>> batch.add_open_ends(['q8a', 'q9a'], break_by=['record_number', 'age'])
```

6.2.4 Special aggregations

It is possible to add some special aggregations to a `qp.Batch`, that are not stored in the main construction plan `.x_y_map`. One option is to give a name for a Cluster key in which all y-keys are cross-tabulated against each other:

```
>>> batch.add_y_on_y('y-keys')
```

Another possibility is to add a `qp.Batch` instance to an other instance. The added Batch loses all information about verbatims and `.y_on_y`, that means only the main construction plan in `.x_y_map` gets adopted. Each of the two batches is aggregated discretely in the `qp.Stack`, but the added instance gets included into the `qp.Cluster` of the first `qp.Batch` in a key named by its instance name.

```
>>> batch1 = dataset.get_batch('batch1')
>>> batch2 = dataset.get_batch('batch2')
>>> batch2.add_x('q2b')
Array summaries setup: Creating no summaries!
>>> batch2.add_y('gender')
>>> batch2.as_addition('batch1')
Batch 'batch2' specified as addition to Batch 'batch1'. Any open end summaries and 'y_
↪on_y' agg. have been removed!
```

The connection between the two `qp.Batch` instances you can see in `.additional` for the added instance and in `._meta['sets']['batches']['batchname']['additions']` for the first instance.

6.3 Set properties of a qp.Batch

The section before explained how the main construction plan (batch.x_y_map) is built, that describes which x-keys and y-keys are used to add qp.Links to a qp.Stack. Now you will get to know how the missing information for the Links are defined and which specific views get extracted for the qp.Cluster by adding some property options the qp.Batch instance.

6.3.1 Filter, weights and significance testing

qp.Links can be added to a qp.Stack data_key-level by defining its x and y-keys, which is already done in .x_y_map, and setting a filter. This property can be edited in a qp.Batch instance with the following methods:

```
>>> batch.add_filter('men only', {'gender': 1})
>>> batch.extend_filter({'q1': {'age': [20, 21, 22, 23, 24, 25]}})
```

Filters can be added globally or for a selection of x-keys only. Out of the global filter, .sample_size is automatically calculated for each qp.Batch defintion.

Now all information are collected in the qp.Batch instance and the Stack can be populated with Links in form of stack[data_key][filter_key][x_key][y_key].

For each Link qp.Views can be added, these views depend on a weight definition, which is also defined in the qp.Batch:

```
>>> batch.set_weights(['weight_a'])
```

Significance tests are a special View; the sig. levels which they are calculated on can be added to the qp.Batch like this:

```
>>> batch.set_sigtests(levels=[0.05])
```

6.3.2 Cell items and language

As qp.Stack is a container for a large amount of aggregations, it will accommodate various qp.Views. The qp.Batch property .cell_items is used to define which specfic Views will be taken to create a qp.Cluster:

```
>>> batch.set_cell_items(['c', 'p'])
```

The property .language allows the user to define which text labels from the meta data should be used for the extracted Views by entering a valid text key:

```
>>> batch.set_language('en-GB')
```

6.4 Inherited qp.DataSet methods

Being a qp.DataSet subclasss, qp.Batch inherits some of its methods. The important ones are these which allow the manipulation of the meta component. That means meta-edits can be applied globally (run methods on qp.DataSet) or Batch-specific (run methods on qp.Batch). Batch meta-edits always overwrite global meta-edits and while building a qp.Cluster from a qp.Batch, the modified meta information is taken from .meta_edits.

The following methods can be used to create meta-edits for a qp.Batch:

```
>>> batch.hiding('q1', [2], axis='y')
>>> batch.sorting('q2', fix=[97, 98])
>>> batch.slicing('q1', [1, 2, 3, 4, 5], axis='x')
>>> batch.set_variable_text('gender', 'Gender???')
>>> batch.set_value_texts('gender', {1: 'Men', 2: 'Women'})
>>> batch.set_property('q1', 'base_text', 'This var has a second filter.')
```

Some methods are not allowed to be used for a `Batch`. These will raise a `NotImplementedError` to prevent inconsistent case and meta data states.

CHAPTER 7

Analysis & aggregation

7.1 Collecting aggregations

All computational results are collected in a so-called `qp.Stack` object which acts as a container for large amount of aggregations in form of `qp.Links`.

7.1.1 What is a `qp.Link`?

A `qp.Link` is defined by four attributes that make it unique and set how it is stored in a `qp.Stack`. These four attributes are `data_key`, `filter`, `x` (downbreak) and `y` (crossbreak), which are positioned in a `qp.Stack` similar to a tree diagram:

- Each `Stack` can have various `data_keys`.
- Each `data_key` can have various `filters`.
- Each `filter` can have various `xs`.
- Each `x` can have various `ys`.

Consequently `qp.Stack[dk][filter][x][y]` is one `qp.Link` that can be added using `add_link(self, data_keys=None, filters=['no_filter'], x=None, y=None, ...)`

`qp.Links` are are storing different `qp.Views` (frequencies, statistics, etc. - all kinds of computations) that are applied on the same four data attributes.

7.1.2 Populating a `qp.Stack`

A `qp.Stack` is able to cope with a large amount of aggregations, so it is impractical to add `Links` one by one with repeated `Stack.add_link()` calls. It is much easier to create a "construction plan" using a `qp.Batch` and apply the settings saved in `DataSet._meta['sets']['batches']` to populate a `qp.Stack` instance. In the following, let's assume `dataset` is containing the definitions of two `qp.Batches`, a `qp.Stack` can be created running:

```
stack = dataset.populate(batches='all')
```

For the `Batch` definitions from *here*, you will get the following *construction plans*:

```
>>> batch1 = dataset.get_batch('batch1')
>>> batch1.add_y_on_y('y_keys')
```

```
>>> print batch1.x_y_map
OrderedDict([('q1', ['@', 'gender', 'q1', 'locality', 'ethnicity']),
             ('q2', ['locality', 'ethnicity']),
             ('q6', ['@']),
             ('@', ['q6']),
             (u'q6_1', ['@', 'gender', 'q1']),
             (u'q6_2', ['@', 'gender', 'q1']),
             (u'q6_3', ['@', 'gender', 'q1'])])
```

```
>>> print batch1.x_filter_map
OrderedDict([('q1', {'(men only)+(q1)': (<function _intersection at␣
↪0x0000000019AE06D8>, [{'gender': 1}, {'age': [20, 21, 22, 23, 24, 25]}])}),
             ('q2', {'men only': {'gender': 1}}),
             ('q6', {'men only': {'gender': 1}}),
             ('q6_1', {'men only': {'gender': 1}}),
             ('q6_2', {'men only': {'gender': 1}}),
             ('q6_3', {'men only': {'gender': 1}})])
```

```
>>> batch2 = dataset.get_batch('batch2')
```

```
>>> print batch2.x_y_map
OrderedDict([('q2b', ['@', 'gender'])])
```

```
>>> print batch2.x_filter_map
OrderedDict([('q2b', 'no_filter')])
```

As both `Batch`es refer to the same data file, the same `data_key` (in this case the name of `dataset`) is defining all `Links`.

After populating the `Stack` content can be viewed using `.describe()`:

```
>>> stack.describe()
                data           filter       x          y  view  #
0   Example Data (A)         men only      q1         q1   NaN  1
1   Example Data (A)         men only      q1          @   NaN  1
2   Example Data (A)         men only      q1     gender   NaN  1
3   Example Data (A)         men only       @         q6   NaN  1
4   Example Data (A)         men only      q2  ethnicity   NaN  1
5   Example Data (A)         men only      q2   locality   NaN  1
6   Example Data (A)         men only    q6_1         q1   NaN  1
7   Example Data (A)         men only    q6_1          @   NaN  1
8   Example Data (A)         men only    q6_1     gender   NaN  1
9   Example Data (A)         men only    q6_2         q1   NaN  1
10  Example Data (A)         men only    q6_2          @   NaN  1
11  Example Data (A)         men only    q6_2     gender   NaN  1
12  Example Data (A)         men only    q6_3         q1   NaN  1
13  Example Data (A)         men only    q6_3          @   NaN  1
14  Example Data (A)         men only    q6_3     gender   NaN  1
15  Example Data (A)         men only  gender         q1   NaN  1
```

(continues on next page)

(continued from previous page)

```
16  Example Data (A)          men only  gender          @   NaN  1
17  Example Data (A)          men only  gender     gender   NaN  1
18  Example Data (A)          men only      q6          @   NaN  1
19  Example Data (A)  (men only)+(q1)      q1         q1   NaN  1
20  Example Data (A)  (men only)+(q1)      q1          @   NaN  1
21  Example Data (A)  (men only)+(q1)      q1   locality   NaN  1
22  Example Data (A)  (men only)+(q1)      q1  ethnicity   NaN  1
23  Example Data (A)  (men only)+(q1)      q1     gender   NaN  1
24  Example Data (A)         no_filter     q2b          @   NaN  1
25  Example Data (A)         no_filter     q2b     gender   NaN  1
```

You can find all combinations defined in the `x_y_map` in the `Stack` structure, but also `Links` like `Stack['Example Data (A)']['men only']['gender']['gender']` are included. These special cases arising from the `y_on_y` setting. Sometimes it is helpful to group a `describe`-dataframe and create a cross-tabulation of the four `Link` attributes to get a better overview, e.g. to see how many `Links` are included for each x-filter combination. :

```
>>> stack.describe('x', 'filter')
filter   (men only)+(q1)  men only  no_filter
x
@                    NaN       1.0        NaN
gender               NaN       3.0        NaN
q1                   5.0       3.0        NaN
q2                   NaN       2.0        NaN
q2b                  NaN       NaN        2.0
q6                   NaN       1.0        NaN
q6_1                 NaN       3.0        NaN
q6_2                 NaN       3.0        NaN
q6_3                 NaN       3.0        NaN
```

7.2 The computational engine

7.3 Significance testing

7.4 View aggregation

All following examples are working with a `qp.Stack` that was populated from a `qp.DataSet` including the following `qp.Batch` definitions:

```
>>> batch1 = dataset.get_batch('batch1')
>>> batch1.add_y_on_y('y_keys')
```

```
>>> print batch1.x_y_map
OrderedDict([('q1', ['@', 'gender', 'q1', 'locality', 'ethnicity']),
             ('q2', ['locality', 'ethnicity']),
             ('q6', ['@']),
             ('@', ['q6']),
             (u'q6_1', ['@', 'gender', 'q1']),
             (u'q6_2', ['@', 'gender', 'q1']),
             (u'q6_3', ['@', 'gender', 'q1'])])
```

```
>>> print batch1.x_filter_map
OrderedDict([('q1', {'(men only)+(q1)': (<function _intersection at␣
↪0x0000000019AE06D8>, [{'gender': 1}, {'age': [20, 21, 22, 23, 24, 25]}])}),
             ('q2', {'men only': {'gender': 1}}),
             ('q6', {'men only': {'gender': 1}}),
             ('q6_1', {'men only': {'gender': 1}}),
             ('q6_2', {'men only': {'gender': 1}}),
             ('q6_3', {'men only': {'gender': 1}})])
```

```
>>> print batch1.weights
['weight_a']
```

```
>>> batch2 = dataset.get_batch('batch2')
```

```
>>> print batch2.x_y_map
OrderedDict([('q2b', ['@', 'gender'])])
```

```
>>> print batch2.x_filter_map
OrderedDict([('q2b', 'no_filter')])
```

```
>>> print batch2.weights
['weight']
```

7.4.1 Basic views

It is possible to add various `qp.Views` to a `Link`. This can be performed by running `Stack.add_link()` providing `View` objects via the `view` parameter. Alternatively, the `qp.Batch` definitions that are stored in the meta data help to add basic `Views` (counts, percentages, bases and sums). By simply running `Stack.aggregate()` we can easily add a large amount of aggregations in one step.

Note: `Stack.aggregate()` can only be used with pre-populated `Stacks`! (see *DataSet.populate()*).

For instance, we can add column percentages and (unweighted and weighted) base sizes to all `Links` of `batch2` like this:

```
>>> stack.aggregate(views=['c%', 'cbase'], unweighted_base=True, batches='batch2',␣
↪verbose=False)
>>> stack.describe()
                data           filter       x          y                 view  #
0   Example Data (A)         men only      q1         q1                  NaN  1
1   Example Data (A)         men only      q1          @                  NaN  1
2   Example Data (A)         men only      q1     gender                  NaN  1
3   Example Data (A)         men only       @         q6                  NaN  1
4   Example Data (A)         men only      q2  ethnicity                  NaN  1
5   Example Data (A)         men only      q2   locality                  NaN  1
6   Example Data (A)         men only    q6_1         q1                  NaN  1
7   Example Data (A)         men only    q6_1          @                  NaN  1
8   Example Data (A)         men only    q6_1     gender                  NaN  1
9   Example Data (A)         men only    q6_2         q1                  NaN  1
10  Example Data (A)         men only    q6_2          @                  NaN  1
11  Example Data (A)         men only    q6_2     gender                  NaN  1
12  Example Data (A)         men only    q6_3         q1                  NaN  1
```

(continues on next page)

(continued from previous page)

```
13  Example Data (A)          men only     q6_3          @                    NaN  1
14  Example Data (A)          men only     q6_3     gender                    NaN  1
15  Example Data (A)          men only   gender         q1                    NaN  1
16  Example Data (A)          men only   gender          @                    NaN  1
17  Example Data (A)          men only   gender     gender                    NaN  1
18  Example Data (A)          men only       q6          @                    NaN  1
19  Example Data (A)  (men only)+(q1)       q1         q1                    NaN  1
20  Example Data (A)  (men only)+(q1)       q1          @                    NaN  1
21  Example Data (A)  (men only)+(q1)       q1   locality                    NaN  1
22  Example Data (A)  (men only)+(q1)       q1  ethnicity                    NaN  1
23  Example Data (A)  (men only)+(q1)       q1     gender                    NaN  1
24  Example Data (A)         no_filter     q2b          @     x|f|:|y|weight|c%  1
25  Example Data (A)         no_filter     q2b          @  x|f|x:||weight|cbase  1
26  Example Data (A)         no_filter     q2b          @        x|f|x:|||cbase  1
27  Example Data (A)         no_filter     q2b     gender     x|f|:|y|weight|c%  1
28  Example Data (A)         no_filter     q2b     gender  x|f|x:||weight|cbase  1
29  Example Data (A)         no_filter     q2b     gender        x|f|x:|||cbase  1
```

Obviously `Views` are only added to `Links` defined by `batch2` and automatically weighted according to the weight definition of `batch2`, which is evident from the view keys (`x|f|:|y|weight|c%`). Combining the information of the four `Link` attributes with a view key, leads to a `pd.DataFrame` and its belonging meta information:

```
>>> link = stack['Example Data (A)']['no_filter']['q2b']['gender']
>>> view_key = 'x|f|:|y|weight|c%'
```

```
>>> link[view_key]
Question               q2b
Values                   @
Question Values
q2b      1       11.992144
         2       80.802580
         3        7.205276
```

```
>>> link[view_key].meta()
{
    "agg": {
        "weights": "weight",
        "name": "c%",
        "grp_text_map": null,
        "text": "",
        "fullname": "x|f|:|y|weight|c%",
        "is_weighted": true,
        "method": "frequency",
        "is_block": false
    },
    "x": {
        "is_array": false,
        "name": "q2b",
        "is_multi": false,
        "is_nested": false
    },
    "shape": [
        3,
        1
    ],
    "y": {
```

(continues on next page)

(continued from previous page)

```
        "is_array": false,
        "name": "@",
        "is_multi": false,
        "is_nested": false
    }
}
```

Now we are adding `Views` to all `batch1`-defined `Links` as well:

```
>>> stack.aggregate(views=['c%', 'counts', 'cbase'], unweighted_base=True, batches=
↪'batch1', verbose=False)
>>> stack.describe(['x', 'view'], 'y').loc[['@', 'q6'], ['@', 'q6']]
y                            @   q6
x  view
@  x|f|:|y|weight_a|c%     NaN  1.0
   x|f|:||weight_a|counts  NaN  1.0
q6 x|f|:|y|weight_a|c%     1.0  NaN
   x|f|:||weight_a|counts  1.0  NaN
```

Even if unweighted bases are requested, they get skipped for array summaries and transposed arrays.

Since `y_on_y` is requested, for a variable used as cross- and downbreak, with an extended filter (in this example `q1`), two `Links` with `Views` are created:

```
>>> stack.describe(['y', 'filter', 'view'], 'x').loc['q1', 'q1']
filter           view
(men only)+(q1)  x|f|:|y|weight_a|c%       1.0
                 x|f|:||weight_a|counts    1.0
                 x|f|x:||weight_a|cbase    1.0
                 x|f|x:|||cbase            1.0
men only         x|f|:|y|weight_a|c%       1.0
                 x|f|:||weight_a|counts    1.0
                 x|f|x:||weight_a|cbase    1.0
                 x|f|x:|||cbase            1.0
```

The first one is the aggregation defined by the `Batch` construction plan, the second one shows the `y_on_y` aggregation using only the main `Batch.filter`.

7.4.2 Non-categorical variables

```
>>> batch3 = dataset.add_batch('batch3')
>>> batch3.add_x('age')
>>> stack = dataset.populate('batch3')
>>> stack.describe()
               data     filter    x  y  view  #
0  Example Data (A)  no_filter  age  @   NaN  1
```

Non-categorical variables (`int` or `float`) are handled in a special way. There are two options:

- Treat them like categorical variables: Append them to the parameter `categorize`, then counts, percentage and sum aggregations can be added alongside the `cbase` `View`.

```
>>> stack.aggregate(views=['c%', 'counts', 'cbase', 'counts_sum', 'c%_sum'],
                    unweighted_base=True,
                    categorize=['age'],
```

(continues on next page)

(continued from previous page)

```
                    batches='batch3',
                    verbose=False)
```

```
>>> stack.describe()
                data     filter    x  y                      view  #
0  Example Data (A)  no_filter  age  @           x|f|:|||counts  1
1  Example Data (A)  no_filter  age  @     x|f.c:f|x:|y||c%_sum  1
2  Example Data (A)  no_filter  age  @              x|f|:|y||c%  1
3  Example Data (A)  no_filter  age  @           x|f|x:|||cbase  1
4  Example Data (A)  no_filter  age  @  x|f.c:f|x:|||counts_sum  1
```

- Do not categorize the variable: Only `cbase` is created and additional descriptive statistics `Views` must be added. The method will raise a warning:

```
>>> stack.aggregate(views=['c%', 'counts', 'cbase', 'counts_sum', 'c%_sum'],
                    unweighted_base=True,
                    batches='batch3',
                    verbose=True)
Warning: Found 1 non-categorized numeric variable(s): ['age'].
Descriptive statistics must be added!
```

```
>>> stack.describe()
                data     filter    x  y            view  #
0  Example Data (A)  no_filter  age  @  x|f|x:|||cbase  1
```

7.4.3 Descriptive statistics

```
>>> b_name = 'batch4'
>>> batch4 = dataset.add_batch(b_name)
>>> batch4.add_x(['q2b', 'q6', 'age'])
>>> stack = dataset.populate(b_name)
>>> stack.aggregate(views=['counts', 'cbase'], batches=b_name, verbose=False)
```

```
>>> stack.describe()
                data     filter     x  y            view  #
0   Example Data (A)  no_filter   q2b  @  x|f|:|||counts  1
1   Example Data (A)  no_filter   q2b  @  x|f|x:|||cbase  1
2   Example Data (A)  no_filter  q6_1  @  x|f|:|||counts  1
3   Example Data (A)  no_filter  q6_1  @  x|f|x:|||cbase  1
4   Example Data (A)  no_filter  q6_2  @  x|f|:|||counts  1
5   Example Data (A)  no_filter  q6_2  @  x|f|x:|||cbase  1
6   Example Data (A)  no_filter  q6_3  @  x|f|:|||counts  1
7   Example Data (A)  no_filter  q6_3  @  x|f|x:|||cbase  1
8   Example Data (A)  no_filter   age  @  x|f|x:|||cbase  1
9   Example Data (A)  no_filter    q6  @  x|f|:|||counts  1
10  Example Data (A)  no_filter    q6  @  x|f|x:|||cbase  1
```

Adding descriptive statistics `Views` like mean, stddev, min, max, median, etc. can be added with the method `stack.add_stats()`. With the parameters `other_source`, `rescale` and `exclude` you can specify the calculation. Again each combination of the parameters refers to a unique view key. Note that in `on_vars` included arrays get unrolled, that means also all belonging array items get equipped with the added `View`:

```
>>> stack.add_stats(on_vars=['q2b', 'age'], stats='mean', _batches=b_name,␣
↪verbose=False)
>>> stack.add_stats(on_vars=['q6'], stats='stddev', _batches=b_name, verbose=False)
>>> stack.add_stats(on_vars=['q2b'], stats='mean', rescale={1:100, 2:50, 3:0},
...                 custom_text='rescale mean', _batches=b_name, verbose=False)
```

```
>>> stack.describe('view', 'x')
x                               age  q2b   q6  q6_1  q6_2  q6_3
view
x|d.mean|x:|||stat              1.0  1.0  NaN   NaN   NaN   NaN
x|d.mean|x[{100,50,0}]:|||stat  NaN  1.0  NaN   NaN   NaN   NaN
x|d.stddev|x:|||stat            NaN  NaN  1.0   1.0   1.0   1.0
x|f|:|||counts                  NaN  1.0  1.0   1.0   1.0   1.0
x|f|x:|||cbase                  1.0  1.0  1.0   1.0   1.0   1.0
```

7.4.4 Nets

```
>>> b_name = 'batch5'
>>> batch5 = dataset.add_batch(b_name)
>>> batch5.add_x(['q2b', 'q6'])
>>> stack = dataset.populate(b_name)
>>> stack.aggregate(views=['counts', 'c%', 'cbase'], batches=b_name, verbose=False)
```

```
>>> stack.describe('view', 'x')
x               q2b  q6  q6_1  q6_2  q6_3
view
x|f|:|y||c%       1   1     1     1     1
x|f|:|||counts    1   1     1     1     1
x|f|x:|||cbase    1   1     1     1     1
```

Net-like `Views` can be added with the method `Stack.add_nets()` by defining `net_maps` for selected variables. There is a distinction between two different types of net `Views`:

- Expanded nets: The existing counts or percentage `Views` are replaced with the new net `Views`, which will the net-defining codes after or before the computed net groups (i.e. "overcode" nets).

```
>>> stack.add_nets('q2b', [{'Top2': [1, 2]}], expand='after', _batches=b_name,␣
↪verbose=False)
```

```
>>> stack.describe('view', 'x')
x                        q2b   q6  q6_1  q6_2  q6_3
view
x|f|:|y||c%              NaN  1.0   1.0   1.0   1.0
x|f|:|||counts           NaN  1.0   1.0   1.0   1.0
x|f|x:|||cbase           1.0  1.0   1.0   1.0   1.0
x|f|x[{1,2}+]*:|y||net   1.0  NaN   NaN   NaN   NaN
x|f|x[{1,2}+]*:|||net    1.0  NaN   NaN   NaN   NaN
```

- Not expanded nets: The new net `Views` are added to the stack, which contain only the computed net groups.

```
>>> stack.add_nets('q2b', [{'Top2': [1, 2]}], _batches=b_name, verbose=False)
```

```
>>> stack.describe('view', 'x')
x                        q2b   q6  q6_1  q6_2  q6_3
```

(continues on next page)

(continued from previous page)

```
view
x|f|:|y||c%              NaN  1.0   1.0   1.0   1.0
x|f|:|||counts           NaN  1.0   1.0   1.0   1.0
x|f|x:|||cbase           1.0  1.0   1.0   1.0   1.0
x|f|x[{1,2}+]*:|y||net   1.0  NaN   NaN   NaN   NaN
x|f|x[{1,2}+]*:|||net    1.0  NaN   NaN   NaN   NaN
x|f|x[{1,2}]:|y||net     1.0  NaN   NaN   NaN   NaN
x|f|x[{1,2}]:|||net      1.0  NaN   NaN   NaN   NaN
```

The difference between the two net types are also visible in the view keys: `x|f|x[{1,2}+]*:|||net` versus `x|f|x[{1,2}]:|||net`.

7.4.4.1 Net definitions

To create more complex net definitions the method `quantipy.net()` can be used, which generates a well-formatted instruction dict and appends it to the `net_map`. It's a helper especially concerning including various texts with different valid `text_keys`. The next example shows how to prepare a net for 'q6' (promoters, detractors):

```
>>> q6_net = qp.net([], [1, 2, 3, 4, 5, 6], 'Promotors', ['en-GB', 'sv-SE'])
>>> q6_net = qp.net(q6_net, [9, 10], {'en-GB': 'Detractors',
...                                   'sv_SE': 'Detractors',
...                                   'de-DE': 'Kritiker'})
>>> qp.net(q6_net[0], text='Promoter', text_key='de-DE')
```

```
>>> print q6_net
[
    {
        "1": [1, 2, 3, 4, 5, 6],
        "text": {
            "en-GB": "Promotors",
            "sv-SE": "Promotors",
            "de-DE": "Promoter"
        }
    },
    {
        "2": [9, 10],
        "text": {
            "en-GB": "Detractors",
            "sv_SE": "Detractors",
            "de-DE": "Kritiker"
        }
    }
]
```

7.4.4.2 Calculations

`Stack.add_nets()` has the parameter `calc`, which allows adding `Views` that are calculated out of the defined nets. The method `qp.calc()` is a helper to create a well-formatted instruction dict for the calculation. For instance, to calculate the NPS (*promoters* - *detractors*) for `'q6'`, see the example above and create the following calculation:

```
>>> q6_calc = qp.calc((1, '-', 2), 'NPS', ['en-GB', 'sv-SE', 'de-DE'])
```

```
>>> print q6_calc
OrderedDict([('calc', ('net_1', <built-in function sub>, 'net_2')),
             ('calc_only', False),
             ('text', {'en-GB': 'NPS',
                       'sv-SE': 'NPS',
                       'de-DE': 'NPS'})])
```

```
>>> stack.add_nets('q6', q6_net, calc=q6_calc, _batches=b_name, verbose=False)
```

```
>>> stack.describe('view', 'x')
x                                                        q2b   q6  q6_1  q6_2  q6_3
view
x|f.c:f|x[{1,2,3,4,5,6}],x[{9,10}],x[{1,2,3,4,5...  NaN  1.0   1.0   1.0   1.0
x|f.c:f|x[{1,2,3,4,5,6}],x[{9,10}],x[{1,2,3,4,5...  NaN  1.0   1.0   1.0   1.0
x|f|:|y||c%                                          NaN  1.0   1.0   1.0   1.0
x|f|:|||counts                                       NaN  1.0   1.0   1.0   1.0
x|f|x:|||cbase                                       1.0  1.0   1.0   1.0   1.0
x|f|x[{1,2}+]*:|y||net                               1.0  NaN   NaN   NaN   NaN
x|f|x[{1,2}+]*:|||net                                1.0  NaN   NaN   NaN   NaN
x|f|x[{1,2}]:|y||net                                 1.0  NaN   NaN   NaN   NaN
x|f|x[{1,2}]:|||net                                  1.0  NaN   NaN   NaN   NaN
```

You can see that nets that are added on arrays are also applied for all array items.

7.4.5 Cumulative sums

Cumulative sum `Views` can be added to a specified collection of xks of the `Stack` using `stack.cumulative_sum()`. These `Views` will always replace the regular counts and percentage `Views`:

```
>>> b_name = 'batch6'
>>> batch6 = dataset.add_batch(b_name)
>>> batch6.add_x(['q2b', 'q6'])
>>> stack = dataset.populate(b_name)
>>> stack.aggregate(views=['counts', 'c%', 'cbase'], batches=b_name, verbose=False)
```

```
>>> stack.cumulative_sum('q6', verbose=False)
```

```
>>> stack.describe('view', 'x')
x                              q2b   q6  q6_1  q6_2  q6_3
view
x|f.c:f|x++:|y||c%_cumsum      NaN  1.0   1.0   1.0   1.0
x|f.c:f|x++:|||counts_cumsum   NaN  1.0   1.0   1.0   1.0
x|f|:|y||c%                    1.0  NaN   NaN   NaN   NaN
x|f|:|||counts                 1.0  NaN   NaN   NaN   NaN
x|f|x:|||cbase                 1.0  1.0   1.0   1.0   1.0
```

7.4.6 Significance tests

```
>>> batch2 = dataset.get_batch('batch2')
>>> batch2.set_sigtests([0.05])
>>> batch5 = dataset.get_batch('batch5')
>>> batch5.set_sigtests([0.01, 0.05])
>>> stack = dataset.populate(['batch2', 'batch5'])
```

```
>>> stack.aggregate(['counts', 'cbase'], batches=['batch2', 'batch5'], verbose=False)
```

```
>>> stack.describe(['view', 'y'], 'x')
x                           q2b   q6  q6_1  q6_2  q6_3
view                  y
x|f|:||weight|counts @       1.0  NaN   NaN   NaN   NaN
                      gender  1.0  NaN   NaN   NaN   NaN
x|f|:|||counts       @       1.0  1.0   1.0   1.0   1.0
x|f|x:||weight|cbase @       1.0  NaN   NaN   NaN   NaN
                      gender  1.0  NaN   NaN   NaN   NaN
x|f|x:|||cbase       @       1.0  1.0   1.0   1.0   1.0
                      gender  1.0  NaN   NaN   NaN   NaN
```

Significance tests can only be added `Batch`-wise, which also means that significance levels must be defined for each `Batch` before running `stack.add_tests()`.

```
>>> stack.add_tests(['batch2', 'batch5'], verbose=False)
```

```
>>> stack.describe(['view', 'y'], 'x')
x                                          q2b   q6  q6_1  q6_2  q6_3
view                                  y
x|f|:||weight|counts                  @       1.0  NaN   NaN   NaN   NaN
                                      gender  1.0  NaN   NaN   NaN   NaN
x|f|:|||counts                        @       1.0  1.0   1.0   1.0   1.0
x|f|x:||weight|cbase                  @       1.0  NaN   NaN   NaN   NaN
                                      gender  1.0  NaN   NaN   NaN   NaN
x|f|x:|||cbase                        @       1.0  1.0   1.0   1.0   1.0
                                      gender  1.0  NaN   NaN   NaN   NaN
x|t.props.Dim.01|:|||significance     @       1.0  NaN   1.0   1.0   1.0
x|t.props.Dim.05|:||weight|significance @      1.0  NaN   NaN   NaN   NaN
                                      gender  1.0  NaN   NaN   NaN   NaN
x|t.props.Dim.05|:|||significance     @       1.0  NaN   1.0   1.0   1.0
```

CHAPTER 8

Builds

8.1 Combining results

8.1.1 Organizing `View` aggregations

Text

8.1.2 Creating `Chain` aggregations

Text

8.2 Deriving post aggregation results

8.2.1 Summarizing and reducing results

Text on `join()` and `cut()`

8.2.2 Custom calculations

Text

CHAPTER 9

API references

9.1 Chain

class quantipy.**Chain**(*name=None*)

Container class that holds ordered Link defintions and associated Views.

The Chain object is a subclassed dict of list where each list contains one or more View aggregations of a Stack. It is an internal class included and used inside the Stack object. Users can interact with the data directly through the Chain or through the related Cluster object.

concat()

Concatenates all Views found for the Chain definition along its orientations axis.

copy()

Create a copy of self by serializing to/from a bytestring using cPickle.

describe(*index=None*, *columns=None*, *query=None*)

Generates a list of all link defining stack keys.

static load(*filename*)

This method loads the pickled object that is made using method: save()

filename

Specifies the name of the file to be loaded. Example of use: new_stack = Chain.load("./tests/ChainName.chain")

Type string

save(*path=None*)

This method saves the current chain instance (self) to file (.chain) using cPickle.

Attributes :

path (string) Specifies the location of the saved file, NOTE: has to end with '/' Example: './tests/'

9.2 Cluster

class quantipy.**Cluster**(*name=''*)
Container class in form of an OrderedDict of Chains.

It is possible to interact with individual Chains through the Cluster object. Clusters are mainly used to prepare aggregations for an export/ build, e.g. MS Excel Workbooks.

add_chain(*chains=None*)
Adds chains to a cluster

bank_chains(*spec*, *text_key*)
Return a banked chain as defined by spec.

This method returns a banked or compound chain where the spec describes how the view results from multiple chains should be banked together into the same set of dataframes in a single chain.

Parameters

- **spec** (*dict*) – The banked chain specification object.
- **text_key** (*str, default='values'*) – Paint the x-axis of the banked chain using the spec provided and this text_key.

Returns **bchain** – The banked chain.

Return type *quantipy.Chain*

static load(*path_cluster*)
Load Stack instance from .stack file.

Parameters **path_cluster** (*str*) – The full path to the .cluster file that should be created, including the extension.

Returns

Return type None

merge()
Merges all Chains found in the Cluster into a new pandas.DataFrame.

save(*path_cluster*)
Load Stack instance from .stack file.

Parameters **path_cluster** (*str*) – The full path to the .cluster file that should be created, including the extension.

Returns

Return type None

9.3 DataSet

class quantipy.**DataSet**(*name*, *dimensions_comp=True*)
A set of casedata (required) and meta data (optional).

DESC.

add_filter_var(*name*, *logic*, *overwrite=False*)
Create filter-var, that allows index slicing using `manifest_filter`

Parameters

- **name** (*str*) – Name and label of the new filter-variable, which gets also listed in DataSet.filters
- **logic** (*complex logic/ str, list of complex logic/ str*) – Logic to keep cases. Complex logic should be provided in form of: ` { 'label': 'any text', 'logic': {var: keys} / intersection/ .... } ` If a str (column-name) is provided, automatically a logic is created that keeps all cases which are not empty for this column. If logic is a list, each included list-item becomes a category of the new filter-variable and all cases are kept that satify all conditions (intersection)
- **overwrite** (*bool, default False*) – Overwrite an already existing filter-variable.

add_meta (*name*, *qtype*, *label*, *categories=None*, *items=None*, *text_key=None*, *replace=True*)

Create and insert a well-formed meta object into the existing meta document.

Parameters

- **name** (*str*) – The column variable name keyed in `meta['columns']`.
- **qtype** (*{'int', 'float', 'single', 'delimited set', 'date', 'string'}*) – The structural type of the data the meta describes.
- **label** (*str*) – The `text` label information.
- **categories** (*list of str, int, or tuples in form of (int, str), default None*) – When a list of str is given, the categorical values will simply be enumerated and mapped to the category labels. If only int are provided, text labels are assumed to be an empty str (‘’) and a warning is triggered. Alternatively, codes can be mapped to categorical labels, e.g.: `[(1, 'Elephant'), (2, 'Mouse'), (999, 'No animal')]`
- **items** (*list of str, int, or tuples in form of (int, str), default None*) – If provided will automatically create an array type mask. When a list of str is given, the item number will simply be enumerated and mapped to the category labels. If only int are provided, item text labels are assumed to be an empty str (‘’) and a warning is triggered. Alternatively, numerical values can be mapped explicitly to items labels, e.g.: `[(1 'The first item'), (2, 'The second item'), (99, 'Last item')]`
- **text_key** (*str, default None*) – Text key for text-based label information. Uses the `DataSet.text_key` information if not provided.
- **replace** (*bool, default True*) – If True, an already existing corresponding `pd.DataFrame` column in the case data component will be overwritten with a new (empty) one.

Returns `DataSet` is modified inplace, meta data and `_data` columns will be added

Return type None

align_order (*vlist, align_against=None, integrate_rc=(['_rc', '_rb'], True), fix=[]*)

Align list to existing order.

Parameters

- **vlist** (*list of str*) – The list which should be reordered.
- **align_against** (*str or list of str, default None*) – The list of variables to align against. If a string is provided, the depending set list is taken. If None, “data file” set is taken.

- **integrate_rc** (*tuple (list, bool)*) – The provided list are the suffixes for recodes, the bool decides whether parent variables should be replaced by their recodes if the parent variable is not in vlist.
- **fix** (*list of str*) – Variables which are fixed at the beginning of the reordered list.

all (*name*, *codes*)
Return a logical has_all() slicer for the passed codes.

Note: When applied to an array mask, the has_all() logic is ex- tended to the item sources, i.e. the it must itself be true for *all* the items.

Parameters

- **name** (*str, default None*) – The column variable name keyed in `_meta['columns']` or `_meta['masks']`.
- **codes** (*int or list of int*) – The codes to build the logical slicer from.

Returns **slicer** – The indices fulfilling has_all([codes]).

Return type pandas.Index

any (*name*, *codes*)
Return a logical has_any() slicer for the passed codes.

Note: When applied to an array mask, the has_any() logic is ex- tended to the item sources, i.e. the it must itself be true for *at least one of* the items.

Parameters

- **name** (*str, default None*) – The column variable name keyed in `_meta['columns']` or `_meta['masks']`.
- **codes** (*int or list of int*) – The codes to build the logical slicer from.

Returns **slicer** – The indices fulfilling has_any([codes]).

Return type pandas.Index

band (*name*, *bands*, *new_name=None*, *label=None*, *text_key=None*)
Group numeric data with band definitions treated as group text labels.

Wrapper around `derive()` for quick banding of numeric data.

Parameters

- **name** (*str*) – The column variable name keyed in `_meta['columns']` that will be banded into summarized categories.
- **bands** (list of int/tuple *or* dict mapping the former to value texts) – The categorical bands to be used. Bands can be single numeric values or ranges, e.g.: [0, (1, 10), 11, 12, (13, 20)]. Be default, each band will also make up the value text of the category created in the `_meta` component. To specify custom texts, map each band to a category name e.g.: [{'A': 0}, {'B': (1, 10)}, {'C': 11}, {'D': 12}, {'E': (13, 20)}]
- **new_name** (*str, default None*) – The created variable will be named `'<name>_banded'`, unless a desired name is provided explicitly here.

- **label** (*str, default None*) – The created variable's text label will be identical to the origi- nating one's passed in `name`, unless a desired label is provided explicitly here.
- **text_key** (*str, default None*) – Text key for text-based label information. Uses the `DataSet.text_key` information if not provided.

Returns `DataSet` is modified inplace.

Return type None

by_type(*types=None*)

Get an overview of all the variables ordered by their type.

Parameters **types** (*str or list of str, default None*) – Restrict the overview to these data types.

Returns **overview** – The variables per data type inside the `DataSet`.

Return type pandas.DataFrame

categorize(*name*, *categorized_name=None*)

Categorize an `int`/`string`/`text` variable to `single`.

The `values` object of the categorized variable is populated with the unique values found in the originating variable (ignoring np.NaN / empty row entries).

Parameters

- **name** (*str*) – The column variable name keyed in `meta['columns']` that will be categorized.
- **categorized_name** (*str*) – If provided, the categorized variable's new name will be drawn from here, otherwise a default name in form of `'name#'` will be used.

Returns DataSet is modified inplace, adding the categorized variable to it.

Return type None

clear_factors(*name*)

Remove all factors set in the variable's `'values'` object.

Parameters **name** (*str*) – The column variable name keyed in `_meta['columns']` or `_meta['masks']`.

Returns

Return type None

clone()

Get a deep copy of the `DataSet` instance.

code_count(*name*, *count_only=None*, *count_not=None*)

Get the total number of codes/entries found per row.

Note: Will be 0/1 for type `single` and range between 0 and the number of possible values for type `delimited set`.

Parameters

- **name** (*str*) – The column variable name keyed in `meta['columns']` or `meta['masks']`.

- **count_only** (*int or list of int, default None*) – Pass a list of codes to restrict counting to.
- **count_not** (*int or list of int, default None*) – Pass a list of codes that should no be counted.

Returns **count** – A series with the results as ints.

Return type pandas.Series

code_from_label(*name*, *text_label*, *text_key=None*, *exact=True*, *flat=True*)
Return the code belonging to the passed `text` label (if present).

Parameters

- **name** (*str*) – The originating variable name keyed in `meta['columns']` or `meta['masks']`.
- **text_label** (*str or list of str*) – The value text(s) to search for.
- **text_key** (*str, default None*) – The desired `text_key` to search through. Uses the `DataSet.text_key` information if not provided.
- **exact** (*bool, default True*) – `text_label` must exactly match a categorical value's `text`. If False, it is enough that the category *contains* the `text_label`.
- **flat** (If a list is passed for `text_label`, return all found codes) – as a regular list. If False, return a list of lists matching the order of the `text_label` list.

Returns **codes** – The list of value codes found for the passed label `text`.

Return type list

codes(*name*)
Get categorical data's numerical code values.

Parameters **name** (*str*) – The column variable name keyed in `_meta['columns']`.

Returns **codes** – The list of category codes.

Return type list

codes_in_data(*name*)
Get a list of codes that exist in data.

compare(*dataset*, *variables=None*, *strict=False*, *text_key=None*)
Compares types, codes, values, question labels of two datasets.

Parameters

- **dataset** (*quantipy.DataSet instance*) – Test if all variables in the provided `dataset` are also in `self` and compare their metadata definitions.
- **variables** (*str, list of str*) – Check only these variables
- **strict** (*bool, default False*) – If True lower/ upper cases and spaces are taken into account.
- **text_key** (*str, list of str*) – The textkeys for which texts are compared.

Returns

Return type None

compare_filter(*name1*, *name2*)
Show if filters result in the same index.

Parameters

- **name1** (*str*) – Name of the first filter variable
- **name2** (*str/ list of st*) – Name(s) of the filter variable(s) to compare with.

convert(*name*, *to*)

Convert meta and case data between compatible variable types.

Wrapper around the separate `as_TYPE()` conversion methods.

Parameters

- **name** (*str*) – The column variable name keyed in `meta['columns']` that will be converted.
- **to** (*{'int', 'float', 'single', 'delimited set', 'string'}*) – The variable type to convert to.

Returns The DataSet variable is modified inplace.

Return type None

copy(*name*, *suffix='rec'*, *copy_data=True*, *slicer=None*, *copy_only=None*, *copy_not=None*)

Copy meta and case data of the variable defintion given per `name`.

Parameters

- **name** (*str*) – The originating column variable name keyed in `meta['columns']` or `meta['masks']`.
- **suffix** (*str, default 'rec'*) – The new variable name will be constructed by suffixing the original `name` with `_suffix`, e.g. `'age_rec`.
- **copy_data** (*bool, default True*) – The new variable assumes the `data` of the original variable.
- **slicer** (*dict*) – If the data is copied it is possible to filter the data with a complex logic. Example: slicer = {'q1': not_any([99])}
- **copy_only** (*int or list of int, default None*) – If provided, the copied version of the variable will only contain (data and) meta for the specified codes.
- **copy_not** (*int or list of int, default None*) – If provided, the copied version of the variable will contain (data and) meta for the all codes, except of the indicated.

Returns DataSet is modified inplace, adding a copy to both the data and meta component.

Return type None

copy_array_data(*source*, *target*, *source_items=None*, *target_items=None*, *slicer=None*)

create_set(*setname='new_set'*, *based_on='data file'*, *included=None*, *excluded=None*, *strings='keep'*, *arrays='masks'*, *replace=None*, *overwrite=False*)

Create a new set in `dataset._meta['sets']`.

Parameters

- **setname** (*str, default 'new_set'*) – Name of the new set.
- **based_on** (*str, default 'data file'*) – Name of set that can be reduced or expanded.
- **included** (*str or list/set/tuple of str*) – Names of the variables to be included in the new set. If None all variables in `based_on` are taken.

- **excluded** (*str or list/set/tuple of str*) – Names of the variables to be excluded in the new set.
- **strings** (*{'keep', 'drop', 'only'}, default 'keep'*) – Keep, drop or only include string variables.
- **arrays** (*{'masks', 'columns'}, default masks*) – For arrays add `masks@varname` or `columns@varname`.
- **replace** (*dict*) – Replace a variable in the set with an other. Example: {'q1': 'q1_rec'}, 'q1' and 'q1_rec' must be included in `based_on`. 'q1' will be removed and 'q1_rec' will be moved to this position.
- **overwrite** (*bool, default False*) – Overwrite if `meta['sets'][name]` already exist.

Returns The `DataSet` is modified inplace.

Return type None

crosstab(*x*, *y=None*, *w=None*, *pct=False*, *decimals=1*, *text=True*, *rules=False*, *xtotal=False*, *f=None*)

cut_item_texts(*arrays=None*)

Remove array text from array item texts.

Parameters **arrays** (*str, list of str, default None*) – Cut texts for items of these arrays. If None, all keys in `._meta['masks']` are taken.

data()

Return the `data` component of the `DataSet` instance.

derive(*name*, *qtype*, *label*, *cond_map*, *text_key=None*)

Create meta and recode case data by specifying derived category logics.

Parameters

- **name** (*str*) – The column variable name keyed in `meta['columns']`.
- **qtype** ([`int`, `float`, `single`, `delimited set`]) – The structural type of the data the meta describes.
- **label** (*str*) – The `text` label information.
- **cond_map** (*list of tuples*) – Tuples of either two or three elements of following structures:

  2 elements, no labels provided: (code, <qp logic expression here>), e.g.: `(1, intersection([{'gender': [1]}, {'age': frange('30-40')}]))`

  2 elements, no codes provided: ('text label', <qp logic expression here>), e.g.: `('Cat 1', intersection([{'gender': [1]}, {'age': frange('30-40')}]))`

  3 elements, with codes + labels: (code, 'Label goes here', <qp logic expression here>), e.g.: `(1, 'Men, 30 to 40', intersection([{'gender': [1]}, {'age': frange('30-40')}]))`
- **text_key** (*str, default None*) – Text key for text-based label information. Will automatically fall back to the instance's text_key property information if not provided.

Returns `DataSet` is modified inplace.

Return type None

derotate (*levels*, *mapper*, *other=None*, *unique_key='identity'*, *dropna=True*)
Derotate data and meta using the given mapper, and appending others.

This function derotates data using the specification defined in mapper, which is a list of dicts of lists, describing how columns from data can be read as a heirarchical structure.

Returns derotated DataSet instance and saves data and meta as json and csv.

Parameters

- **levels** (*dict*) – The name and values of a new column variable to identify cases.
- **mapper** (*list of dicts of lists*) – A list of dicts matching where the new column names are keys to to lists of source columns. Example:

```
>>> mapper = [{'q14_1': ['q14_1_1', 'q14_1_2', 'q14_1_3']},
...           {'q14_2': ['q14_2_1', 'q14_2_2', 'q14_2_3']},
...           {'q14_3': ['q14_3_1', 'q14_3_2', 'q14_3_3']}]
```

- **unique_key** (*str*) – Name of column variable that will be copied to new dataset.
- **other** (*list (optional; default=None)*) – A list of additional columns from the source data to be appended to the end of the resulting stacked dataframe.
- **dropna** (*boolean (optional; default=True)*) – Passed through to the pandas.DataFrame.stack() operation.

Returns

Return type new `qp.DataSet` instance

describe (*var=None*, *only_type=None*, *text_key=None*, *axis_edit=None*)
Inspect the DataSet's global or variable level structure.

dichotomize (*name*, *value_texts=None*, *keep_variable_text=True*, *ignore=None*, *replace=False*, *text_key=None*)

dimensionize (*names=None*)
Rename the dataset columns for Dimensions compatibility.

dimensionizing_mapper (*names=None*)
Return a renaming dataset mapper for dimensionizing names.

Parameters **None** –

Returns **mapper** – A renaming mapper in the form of a dict of {old: new} that maps non-Dimensions naming conventions to Dimensions naming conventions.

Return type dict

drop (*name*, *ignore_items=False*)
Drops variables from meta and data components of the `DataSet`.

Parameters

- **name** (*str or list of str*) – The column variable name keyed in `_meta['columns']` or `_meta['masks']`.
- **ignore_items** (*bool*) – If False source variables for arrays in `_meta['columns']` are dropped, otherwise kept.

Returns DataSet is modified inplace.

Return type None

drop_duplicates(*unique_id='identity'*, *keep='first'*, *sort_by=None*)
Drop duplicated cases from self._data.

Parameters

- **unique_id** (*str*) – Variable name that gets scanned for duplicates.
- **keep** (*str, {'first', 'last'}*) – Keep first or last of the duplicates.
- **sort_by** (*str*) – Name of a variable to sort the data by, for example "endtime". It is a helper to specify *keep*.

duplicates(*name='identity'*)
Returns a list with duplicated values for the provided name.

Parameters **name** (*str, default 'identity'*) – The column variable name keyed in `meta['columns']`.

Returns **vals** – A list of duplicated values found in the named variable.

Return type list

empty(*name*, *condition=None*)
Check variables for emptiness (opt. restricted by a condition).

Parameters

- **name** (*(list of) str*) – The mask variable name keyed in `_meta['columns']`.
- **condition** (*Quantipy logic expression, default None*) – A logical condition expressed as Quantipy logic that determines which subset of the case data rows to be considered.

Returns **empty**

Return type bool

empty_items(*name*, *condition=None*, *by_name=True*)
Test arrays for item emptiness (opt. restricted by a condition).

Parameters

- **name** (*(list of) str*) – The mask variable name keyed in `_meta['masks']`.
- **condition** (*Quantipy logic expression, default None*) – A logical condition expressed as Quantipy logic that determines which subset of the case data rows to be considered.
- **by_name** (*bool, default True*) – Return array items by their name or their index.

Returns **empty** – The list of empty items by their source names or positional index (starting from 1!, mapped to their parent mask name if more than one).

Return type list

extend_filter_var(*name*, *logic*, *extend_as=None*)
Extend logic of an existing filter-variable.

Parameters

- **name** (*str*) – Name of the existing filter variable.
- **logic** (*(list of) complex logic/ str*) – Additional logic to keep cases (intersection with existing logic). Complex logic should be provided in form of: `` ` { 'label': 'any text', 'logic': {var: keys} / intersection/ .... } ` ``

- **extend_as** (*str, default None*) – Addition to the filter-name to create a new filter. If it is None the existing filter-variable is overwritten.

extend_items(*name*, *ext_items*, *text_key=None*)
Extend mask items of an existing array.

Parameters

- **name** (*str*) – The originating column variable name keyed in `meta['masks']`.
- **ext_items** (*list of str/ list of dict*) – The label of the new item. It can be provided as str, then the new column is named by the grid and the item_no, or as dict {'new_column': 'label'}.
- **text_key** (*str/ list of str, default None*) – Text key for text-based label information. Will automatically fall back to the instance's text_key property information if not provided.

extend_values(*name*, *ext_values*, *text_key=None*, *safe=True*)
Add to the 'values' object of existing column or mask meta data.

Attempting to add already existing value codes or providing already present value texts will both raise a `ValueError`!

Parameters

- **name** (*str*) – The column variable name keyed in `_meta['columns']` or `_meta['masks']`.
- **ext_values** (*list of str or tuples in form of (int, str), default None*) – When a list of str is given, the categorical values will simply be enumerated and mapped to the category labels. Alternatively codes can mapped to categorical labels, e.g.: [(1, 'Elephant'), (2, 'Mouse'), (999, 'No animal')]
- **text_key** (*str, default None*) – Text key for text-based label information. Will automatically fall back to the instance's text_key property information if not provided.
- **safe** (*bool, default True*) – If set to False, duplicate value texts are allowed when extending the `values` object.

Returns The `DataSet` is modified inplace.

Return type None

factors(*name*)
Get categorical data's stat. factor values.

Parameters **name** (*str*) – The column variable name keyed in `_meta['columns']` or `_meta['masks']`.

Returns **factors** – A `{value: factor}` mapping.

Return type OrderedDict

filter(*alias*, *condition*, *inplace=False*)
Filter the DataSet using a Quantipy logical expression.

find(*str_tags=None*, *suffixed=False*)
Find variables by searching their names for substrings.

Parameters

- **str_tags** (*(list of) str*) – The strings tags to look for in the variable names. If not provided, the modules' default global list of substrings from VAR_SUFFIXES will be used.

- **suffixed** (*bool, default False*) – If set to True, only variable names that end with a given string sequence will qualify.

Returns **found** – The list of matching variable names.

Return type list

find_duplicate_texts(*name*, *text_key=None*)
Collect values that share the same text information to find duplicates.

Parameters

- **name** (*str*) – The column variable name keyed in `_meta['columns']` or `_meta['masks']`.
- **text_key** (*str, default None*) – Text key for text-based label information. Will automatically fall back to the instance's `text_key` property information if not provided.

first_responses(*name*, *n=3*, *others='others'*, *reduce_values=False*)
Create n-first mentions from the set of responses of a delimited set.

Parameters

- **name** (*str*) – The column variable name of a delimited set keyed in `meta['columns']`.
- **n** (*int, default 3*) – The number of mentions that will be turned into single-type variables, i.e. 1st mention, 2nd mention, 3rd mention, 4th mention, etc.
- **others** (*None or str, default 'others'*) – If provided, all remaining values will end up in a new delimited set variable reduced by the responses transferred to the single mention variables.
- **reduce_values** (*bool, default False*) – If True, each new variable will only list the categorical value metadata for the codes found in the respective data vector, i.e. not the initial full codeframe.

Returns DataSet is modified inplace.

Return type None

flatten(*name*, *codes*, *new_name=None*, *text_key=None*)
Create a variable that groups array mask item answers to categories.

Parameters

- **name** (*str*) – The array variable name keyed in `meta['masks']` that will be converted.
- **codes** (*int, list of int*) – The answers codes that determine the categorical grouping. Item labels will become the category labels.
- **new_name** (*str, default None*) – The name of the new delimited set variable. If None, `name` is suffixed with '_rec'.
- **text_key** (*str, default None*) – Text key for text-based label information. Uses the `DataSet.text_key` information if not provided.

Returns The DataSet is modified inplace, delimited set variable is added.

Return type None

force_texts(*copy_to=None*, *copy_from=None*, *update_existing=False*)
Copy info from existing text_key to a new one or update the existing one.

Parameters

- **copy_to** (*str*) – {'en-GB', 'da-DK', 'fi-FI', 'nb-NO', 'sv-SE', 'de-DE'} None -> _meta['lib']['default text'] The text key that will be filled.
- **copy_from** (*str / list*) – {'en-GB', 'da-DK', 'fi-FI', 'nb-NO', 'sv-SE', 'de-DE'} You can also enter a list with text_keys, if the first text_key doesn't exist, it takes the next one
- **update_existing** (*bool*) – True : copy_to will be filled in any case False: copy_to will be filled if it's empty/not existing

Returns

Return type None

from_batch(*batch_name*, *include='identity'*, *text_key=[]*, *apply_edits=True*, *additions='variables'*)

Get a filtered subset of the DataSet using qp.Batch definitions.

Parameters

- **batch_name** (*str*) – Name of a Batch included in the DataSet.
- **include** (*str/ list of str*) – Name of variables that get included even if they are not in Batch.
- **text_key** (*str/ list of str, default None*) – Take over all texts of the included text_key(s), if None is provided all included text_keys are taken.
- **apply_edits** (*bool, default True*) – meta_edits and rules are used as/ applied on global meta of the new DataSet instance.
- **additions** (*{'variables', 'filters', 'full', None}*) – Extend included variables by the xks, yks and weights of the additional batches if set to 'variables', 'filters' will create new 1/0-coded variables that reflect any filters defined. Selecting 'full' will do both, `None` will ignore additional Batches completely.

Returns **b_ds**

Return type `quantipy.DataSet`

from_components(*data_df*, *meta_dict=None*, *reset=True*, *text_key=None*)

Attach data and meta directly to the `DataSet` instance.

Note: Except testing for appropriate object types, this method offers no additional safeguards or consistency/compability checks with regard to the passed data and meta documents!

Parameters

- **data_df** (*pandas.DataFrame*) – A DataFrame that contains case data entries for the `DataSet`.
- **meta_dict** (*dict, default None*) – A dict that stores meta data describing the columns of the data_df. It is assumed to be well-formed following the Quantipy meta data structure.
- **reset** (*bool, default True*) – Clean the *'lib'* and `'sets'` metadata collections from non-native entries, e.g. user-defined information or helper metadata.
- **text_key** (*str, default None*) – The text_key to be used. If not provided, it will be attempted to use the 'default text' from the `meta['lib']` definition.

Returns

Return type None

from_excel(*path_xlsx*, *merge=True*, *unique_key='identity'*)
Converts excel files to a dataset or/and merges variables.

Parameters

- **path_xlsx** (*str*) – Path where the excel file is stored. The file must have exactly one sheet with data.
- **merge** (*bool*) – If True the new data from the excel file will be merged on the dataset.
- **unique_key** (*str*) – If `merge=True` an hmerge is done on this variable.

Returns **new_dataset** – Contains only the data from excel. If `merge=True` dataset is modified inplace.

Return type `quantipy.DataSet`

from_stack(*stack*, *data_key=None*, *dk_filter=None*, *reset=True*)
Use `quantipy.Stack` data and meta to create a `DataSet` instance.

Parameters

- **stack** (`quantipy.Stack`) – The Stack instance to convert.
- **data_key** (*str*) – The reference name where meta and data information are stored.
- **dk_filter** (*string, default None*) – Filter name if the stack contains more than one filters. If None 'no_filter' will be used.
- **reset** (*bool, default True*) – Clean the *'lib'* and `'sets'` metadata collections from non-native entries, e.g. user-defined information or helper metadata.

Returns

Return type None

fully_hidden_arrays()
Get all array definitions that contain only hidden items.

Returns **hidden** – The list of array mask names.

Return type list

get_batch(*name*)
Get existing Batch instance from DataSet meta information.

Parameters **name** (*str*) – Name of existing Batch instance.

get_property(*name*, *prop_name*, *text_key=None*)

hide_empty_items(*condition=None*, *arrays=None*)
Apply `rules` meta to automatically hide empty array items.

Parameters

- **name** (*(list of) str, default None*) – The array mask variable names keyed in `_meta['masks']`. If not explicitly provided will test all array mask definitions.
- **condition** (*Quantipy logic expression*) – A logical condition expressed as Quantipy logic that determines which subset of the case data rows to be considered.

Returns

Return type None

hiding(*name*, *hide*, *axis='y'*, *hide_values=True*)

Set or update `rules[axis]['dropx']` meta for the named column.

Quantipy builds will respect the hidden codes and *cut* them from results.

Note: This is not equivalent to `DataSet.set_missings()` as missing values are respected also in computations.

Parameters

- **name** (*str or list of str*) – The column variable(s) name keyed in `_meta['columns']`.
- **hide** (*int or list of int*) – Values indicated by their `int` codes will be dropped from `Quantipy.View.dataframes`.
- **axis** (*{'x', 'y'}, default 'y'*) – The axis to drop the values from.
- **hide_values** (*bool, default True*) – Only considered if `name` refers to a mask. If True, values are hidden on all mask items. If False, mask items are hidden by position (only for array summaries).

Returns

Return type None

hmerge(*dataset*, *on=None*, *left_on=None*, *right_on=None*, *overwrite_text=False*, *from_set=None*, *inplace=True*, *update_existing=None*, *merge_existing=None*, *text_properties=None*, *verbose=True*)

Merge Quantipy datasets together using an index-wise identifer.

This function merges two Quantipy datasets together, updating variables that exist in the left dataset and appending others. New variables will be appended in the order indicated by the 'data file' set if found, otherwise they will be appended in alphanumeric order. This merge happend horizontally (column-wise). Packed kwargs will be passed on to the pandas.DataFrame.merge() method call, but that merge will always happen using how='left'.

Parameters

- **dataset** (`quantipy.DataSet`) – The dataset to merge into the current `DataSet`.
- **on** (*str, default=None*) – The column to use as a join key for both datasets.
- **left_on** (*str, default=None*) – The column to use as a join key for the left dataset.
- **right_on** (*str, default=None*) – The column to use as a join key for the right dataset.
- **overwrite_text** (*bool, default=False*) – If True, text_keys in the left meta that also exist in right meta will be overwritten instead of ignored.
- **from_set** (*str, default=None*) – Use a set defined in the right meta to control which columns are merged from the right dataset.
- **inplace** (*bool, default True*) – If True, the `DataSet` will be modified inplace with new/updated columns. Will return a new `DataSet` instance if False.
- **update_existing** (*str/ list of str, default None, {'all', [var_names]}*) – Update values for defined delimited sets if it exists in both datasets.

- **text_properties** (*str/ list of str, default=None, {'all', [var_names]}*) – Controls the update of the dataset_left properties with properties from the dataset_right. If None, properties from dataset_left will be updated by the ones from the dataset_right. If 'all', properties from dataset_left will be kept unchanged. Otherwise, specify the list of properties which will be kept unchanged in the dataset_left; all others will be updated by the properties from dataset_right.
- **verbose** (*bool, default=True*) – Echo progress feedback to the output pane.

Returns **None or new_dataset** – If the merge is not applied `inplace`, a `DataSet` instance is returned.

Return type `quantipy.DataSet`

interlock (*name*, *label*, *variables*, *val_text_sep='/'*)

Build a new category-intersected variable from >=2 incoming variables.

Parameters

- **name** (*str*) – The new column variable name keyed in `_meta['columns']`.
- **label** (*str*) – The new text label for the created variable.
- **variables** (*list of >= 2 str or dict (mapper)*) – The column names of the variables that are feeding into the intersecting recode operation. Or dicts/mapper to create temporary variables for interlock. Can also be a mix of str and dict. Example:

```
>>> ['gender',
...  {'agegrp': [(1, '18-34', {'age': frange('18-34')}),
...              (2, '35-54', {'age': frange('35-54')}),
...              (3, '55+', {'age': is_ge(55)})]},
...  'region']
```

- **val_text_sep** (*str, default '/'*) – The passed character (or any other str value) wil be used to separate the incoming individual value texts to make up the intersected category value texts, e.g.: 'Female/18-30/London'.

Returns

Return type None

is_like_numeric (*name*)

Test if a `string`-typed variable can be expressed numerically.

Parameters **name** (*str*) – The column variable name keyed in `_meta['columns']`.

Returns

Return type bool

is_nan (*name*)

Detect empty entries in the `_data` rows.

Parameters **name** (*str*) – The column variable name keyed in `meta['columns']`.

Returns **count** – A series with the results as bool.

Return type pandas.Series

is_subfilter (*name1*, *name2*)

Verify if index of name2 is part of the index of name1.

item_no (*name*)

Return the order/position number of passed array item variable name.

Parameters **name** (*str*) – The column variable name keyed in `_meta['columns']`.

Returns **no** – The positional index of the item (starting from 1).

Return type int

item_texts(*name*, *text_key=None*, *axis_edit=None*)
Get the `text` meta data for the items of the passed array mask name.

Parameters

- **name** (*str*) – The mask variable name keyed in `_meta['masks']`.
- **text_key** (*str, default None*) – The text_key that should be used when taking labels from the source meta.
- **axis_edit** (*{'x', 'y'}, default None*) – If provided the text_key is taken from the x/y edits dict.

Returns **texts** – The list of item texts for the array elements.

Return type list

items(*name*, *text_key=None*, *axis_edit=None*)
Get the array's paired item names and texts information from the meta.

Parameters

- **name** (*str*) – The column variable name keyed in `_meta['masks']`.
- **text_key** (*str, default None*) – The text_key that should be used when taking labels from the source meta.
- **axis_edit** (*{'x', 'y'}, default None*) – If provided the text_key is taken from the x/y edits dict.

Returns **items** – The list of source item names (from `_meta['columns']`) and their `text` information packed as tuples.

Return type list of tuples

link(*filters=None*, *x=None*, *y=None*, *views=None*)
Create a Link instance from the DataSet.

manifest_filter(*name*)
Get index slicer from filter-variables.

Parameters **name** (*str*) – Name of the filter_variable.

merge_texts(*dataset*)
Add additional `text` versions from other `text_key` meta.

Case data will be ignored during the merging process.

Parameters **dataset** ((A list of multiple) `quantipy.DataSet`) – One or multiple datasets that provide new `text_key` meta.

Returns

Return type None

meta(*name=None*, *text_key=None*, *axis_edit=None*)
Provide a *pretty* summary for variable meta given as per `name`.

Parameters

- **name** (*str, default None*) – The variable name keyed in _meta['columns'] or _meta['masks']. If None, the entire meta component of the DataSet instance will be returned.
- **text_key** (*str, default None*) – The text_key that should be used when taking labels from the source meta.
- **axis_edit** (*{'x', 'y'}, default None*) – If provided the text_key is taken from the x/y edits dict.

Returns **meta** – Either a DataFrame that sums up the meta information on a mask or column or the meta dict as a whole is

Return type dict or pandas.DataFrame

meta_to_json (*key=None*, *collection=None*)
Save a meta object as json file.

Parameters

- **key** (*str, default None*) – Name of the variable whose metadata is saved, if key is not provided included collection or the whole meta is saved.
- **collection** (*str {'columns', 'masks', 'sets', 'lib'}, default None*) – The meta object is taken from this collection.

Returns

Return type None

min_value_count (*name*, *min=50*, *weight=None*, *condition=None*, *axis='y'*, *verbose=True*)
Wrapper for self.hiding(), which is hiding low value_counts.

Parameters

- **variables** (*str/ list of str*) – Name(s) of the variable(s) whose values are checked against the defined border.
- **min** (*int*) – If the amount of counts for a value is below this number, the value is hidden.
- **weight** (*str, default None*) – Name of the weight, which is used to calculate the weigthed counts.
- **condition** (*complex logic*) – The data, which is used to calculate the counts, can be filtered by the included condition.
- **axis** (*{'y', 'x', ['x', 'y']}, default None*) – The axis on which the values are hidden.

names (*ignore_items=True*)
Find all weak-duplicate variable names that are different only by case.

Note: Will return self.variables() if no weak-duplicates are found.

Returns **weak_dupes** – An overview of case-sensitive spelling differences in otherwise equal variable names.

Return type pd.DataFrame

order (*new_order=None*, *reposition=None*, *regroup=False*)
Set the global order of the DataSet variables collection.

The global order of the DataSet is reflected in the data component's pd.DataFrame.columns order and the variable references in the meta component's 'data file' items.

Parameters

- **new_order** (*list*) – A list of all DataSet variables in the desired order.
- **reposition** (*(List of) dict*) – Each dict maps one or a list of variables to a reference variable name key. The mapped variables are moved before the reference key.
- **regroup** (*bool, default False*) – Attempt to regroup non-native variables (i.e. created either manually with `add_meta()`, `recode()`, `derive()`, etc. or automatically by manifesting `qp.View` objects) with their originating variables.

Returns

Return type None

parents(*name*)

Get the `parent` meta information for masks-structured column elements.

Parameters **name** (*str*) – The mask variable name keyed in `_meta['columns']`.

Returns **parents** – The list of parents the `_meta['columns']` variable is attached to.

Return type list

populate(*batches='all'*, *verbose=True*)

Create a `qp.Stack` based on all available `qp.Batch` definitions.

Parameters **batches** (*str/ list of str*) – Name(s) of `qp.Batch` instances that are used to populate the `qp.Stack`.

Returns

Return type qp.Stack

read_ascribe(*path_meta*, *path_data*, *text_key*)

Load Dimensions .xml/.txt files, connecting as data and meta components.

Parameters

- **path_meta** (*str*) – The full path (optionally with extension `'.xml'`, otherwise assumed as such) to the meta data defining `'.xml'` file.
- **path_data** (*str*) – The full path (optionally with extension `'.txt'`, otherwise assumed as such) to the case data defining `'.txt'` file.

Returns The `DataSet` is modified inplace, connected to Quantipy data and meta components that have been converted from their Ascribe source files.

Return type None

read_dimensions(*path_meta*, *path_data*)

Load Dimensions .ddf/.mdd files, connecting as data and meta components.

Parameters

- **path_meta** (*str*) – The full path (optionally with extension `'.mdd'`, otherwise assumed as such) to the meta data defining `'.mdd'` file.
- **path_data** (*str*) – The full path (optionally with extension `'.ddf'`, otherwise assumed as such) to the case data defining `'.ddf'` file.

Returns The `DataSet` is modified inplace, connected to Quantipy data and meta components that have been converted from their Dimensions source files.

Return type None

read_quantipy(*path_meta*, *path_data*, *reset=True*)

Load Quantipy .csv/.json files, connecting as data and meta components.

Parameters

- **path_meta** (*str*) – The full path (optionally with extension `'.json'`, otherwise assumed as such) to the meta data defining `'.json'` file.
- **path_data** (*str*) – The full path (optionally with extension `'.csv'`, otherwise assumed as such) to the case data defining `'.csv'` file.
- **reset** (*bool, default True*) – Clean the *'lib'* and `'sets'` metadata collections from non-native entries, e.g. user-defined information or helper metadata.

Returns The `DataSet` is modified inplace, connected to Quantipy native data and meta components.

Return type None

read_spss(*path_sav*, ***kwargs*)

Load SPSS Statistics .sav files, converting and connecting data/meta.

Parameters **path_sav** (*str*) – The full path (optionally with extension `'.sav'`, otherwise assumed as such) to the `'.sav'` file.

Returns The `DataSet` is modified inplace, connected to Quantipy data and meta components that have been converted from the SPSS source file.

Return type None

recode(*target*, *mapper*, *default=None*, *append=False*, *intersect=None*, *initialize=None*, *fillna=None*, *inplace=True*)

Create a new or copied series from data, recoded using a mapper.

This function takes a mapper of {key: logic} entries and injects the key into the target column where its paired logic is True. The logic may be arbitrarily complex and may refer to any other variable or variables in data. Where a pre-existing column has been used to start the recode, the injected values can replace or be appended to any data found there to begin with. Note that this function does not edit the target column, it returns a recoded copy of the target column. The recoded data will always comply with the column type indicated for the target column according to the meta.

Parameters

- **target** (*str*) – The column variable name keyed in `_meta['columns']` that is the target of the recode. If not found in `_meta` this will fail with an error. If `target` is not found in data.columns the recode will start from an empty series with the same index as `_data`. If `target` is found in data.columns the recode will start from a copy of that column.
- **mapper** (*dict*) – A mapper of {key: logic} entries.
- **default** (*str, default None*) – The column name to default to in cases where unattended lists are given in your logic, where an auto-transformation of {key: list} to {key: {default: list}} is provided. Note that lists in logical statements are themselves a form of shorthand and this will ultimately be interpreted as: {key: {default: has_any(list)}}.
- **append** (*bool, default False*) – Should the new recoded data be appended to values already found in the series? If False, data from series (where found) will overwrite whatever was found for that item instead.

- **intersect** (*logical statement, default None*) – If a logical statement is given here then it will be used as an implied intersection of all logical conditions given in the mapper.
- **initialize** (*str or np.NaN, default None*) – If not None, a copy of the data named column will be used to populate the target column before the recode is performed. Alternatively, initialize can be used to populate the target column with np.NaNs (overwriting whatever may be there) prior to the recode.
- **fillna** (*int, default=None*) – If not None, the value passed to fillna will be used on the recoded series as per pandas.Series.fillna().
- **inplace** (*bool, default True*) – If True, the `DataSet` will be modified inplace with new/updated columns. Will return a new recoded `pandas.Series` instance if False.

Returns Either the `DataSet._data` is modified inplace or a new `pandas.Series` is returned.

Return type None or recode_series

reduce_filter_var(*name*, *values*)
Remove values from filter-variables and recalculate the filter.

remove_html()
Cycle through all meta `text` objects removing html tags.

Currently uses the regular expression '<.*?>' in _remove_html() classmethod.

Returns

Return type None

remove_items(*name*, *remove*)
Erase array mask items safely from both meta and case data components.

Parameters

- **name** (*str*) – The originating column variable name keyed in `meta['masks']`.
- **remove** (*int or list of int*) – The items listed by their order number in the `_meta['masks'][name]['items']` object will be droped from the `mask` definition.

Returns DataSet is modified inplace.

Return type None

remove_values(*name*, *remove*)
Erase value codes safely from both meta and case data components.

Attempting to remove all value codes from the variable's value object will raise a `ValueError`!

Parameters

- **name** (*str*) – The originating column variable name keyed in `meta['columns']` or `meta['masks']`.
- **remove** (*int or list of int*) – The codes to be removed from the `DataSet` variable.

Returns DataSet is modified inplace.

Return type None

rename(*name*, *new_name*)
Change meta and data column name references of the variable defintion.

Parameters

- **name** (*str*) – The originating column variable name keyed in `meta['columns']` or `meta['masks']`.
- **new_name** (*str*) – The new variable name.

Returns DataSet is modified inplace. The new name reference replaces the original one.

Return type None

rename_from_mapper(*mapper*, *keep_original=False*, *ignore_batch_props=False*)
Rename meta objects and data columns using mapper.

Parameters **mapper** (*dict*) – A renaming mapper in the form of a dict of {old: new} that will be used to rename columns throughout the meta and data.

Returns DataSet is modified inplace.

Return type None

reorder_items(*name*, *new_order*)
Apply a new order to mask items.

Parameters

- **name** (*str*) – The variable name keyed in `_meta['masks']`.
- **new_order** (*list of int, default None*) – The new order of the mask items. The included ints match up to the number of the items (`DataSet.item_no('item_name')`).

Returns DataSet is modified inplace.

Return type None

reorder_values(*name*, *new_order=None*)
Apply a new order to the value codes defined by the meta data component.

Parameters

- **name** (*str*) – The column variable name keyed in `_meta['columns']` or `_meta['masks']`.
- **new_order** (*list of int, default None*) – The new code order of the DataSet variable. If no order is given, the `values` object is sorted ascending.

Returns DataSet is modified inplace.

Return type None

repair()
Try to fix legacy meta data inconsistencies and badly shaped array / datafile items `'sets'` meta definitions.

repair_text_edits(*text_key=None*)
Cycle through all meta `text` objects repairing axis edits.

Parameters **text_key** (*str / list of str, default None*) – {None, 'en-GB', 'da-DK', 'fi-FI', 'nb-NO', 'sv-SE', 'de-DE'} The text_keys for which text edits should be included.

Returns

Return type None

replace_texts(*replace*, *text_key=None*)
Cycle through all meta `text` objects replacing unwanted strings.

Parameters

- **replace** (*dict, default Nonea*) – A dictionary mapping {unwanted string: replacement string}.
- **text_key** (*str / list of str, default None*) – {None, 'en-GB', 'da-DK', 'fi-FI', 'nb-NO', 'sv-SE', 'de-DE'} The text_keys for which unwanted strings are replaced.

Returns

Return type None

resolve_name(*name*)

restore_item_texts(*arrays=None*)
Restore array item texts.

Parameters **arrays** (*str, list of str, default None*) – Restore texts for items of these arrays. If None, all keys in `._meta['masks']` are taken.

revert()
Return to a previously saved state of the DataSet.

Note: This method is designed primarily for use in interactive Python environments like iPython/Jupyter and their notebook applications.

---

roll_up(*varlist*, *ignore_arrays=None*)
Replace any array items with their parent mask variable definition name.

Parameters

- **varlist** (*list*) – A list of meta `'columns'` and/or `'masks'` names.
- **ignore_arrays** (*(list of) str*) – A list of array mask names that should not be rolled up if their items are found inside `varlist`.

Note: varlist can also contain nesting *var1 > var2*. The variables which are included in the nesting can also be controlled by keep and both, even if the variables are also included as a "normal" variable.

Returns **rolled_up** – The modified `varlist`.

Return type list

save()
Save the current state of the DataSet's data and meta.

The saved file will be temporarily stored inside the cache. Use this to take a snapshot of the DataSet state to easily revert back to at a later stage.

Note: This method is designed primarily for use in interactive Python environments like iPython/Jupyter notebook applications.

---

select_text_keys(*text_key=None*)
Cycle through all meta `text` objects keep only selected text_key.

Parameters **text_key** (*str / list of str, default None*) – {None, 'en-GB', 'da-DK', 'fi-FI', 'nb-NO', 'sv-SE', 'de-DE'} The text_keys which should be kept.

Returns

Return type None

classmethod set_encoding(*encoding*)
Hack sys.setdefaultencoding() to escape ASCII hell.

Parameters **encoding** (*str*) – The name of the encoding to default to.

set_factors(*name*, *factormap*, *safe=False*)
Apply numerical factors to (`single`-type categorical) variables.

Factors can be read while aggregating descrp. stat. `qp.Views`.

Parameters

- **name** (*str*) – The column variable name keyed in `_meta['columns']` or `_meta['masks']`.
- **factormap** (*dict*) – A mapping of `{value: factor}` (`int` to `int`).
- **safe** (*bool, default False*) – Set to `True` to prevent setting factors to the `values` meta data of non-`single` type variables.

Returns

Return type None

set_item_texts(*name*, *renamed_items*, *text_key=None*, *axis_edit=None*)
Rename or add item texts in the `items` objects of `masks`.

Parameters

- **name** (*str*) – The column variable name keyed in `_meta['masks']`.
- **renamed_items** (*dict*) – A dict mapping with following structure (array mask items are assumed to be passed by their order number):

```
>>> {1: 'new label for item #1',
...  5: 'new label for item #5'}
```

- **text_key** (*str, default None*) – Text key for text-based label information. Will automatically fall back to the instance's `text_key` property information if not provided.
- **axis_edit** (*{'x', 'y', ['x', 'y']}, default None*) – If the `new_text` of the variable should only be considered temp. for build exports, the axes on that the edited text should appear can be provided.

Returns The `DataSet` is modified inplace.

Return type None

set_missings(*var*, *missing_map='default'*, *hide_on_y=True*, *ignore=None*)
Flag category definitions for exclusion in aggregations.

Parameters

- **var** (*str or list of str*) – Variable(s) to apply the meta flags to.

- **missing_map** (*'default' or list of codes or dict of {'flag': code(s)}, default 'default'*) – A mapping of codes to flags that can either be 'exclude' (globally ignored) or 'd.exclude' (only ignored in descriptive statistics). Codes provided in a list are flagged as 'exclude'. Passing 'default' is using a preset list of (TODO: specify) values for exclusion.
- **ignore** (*str or list of str, default None*) – A list of variables that should be ignored when applying missing flags via the 'default' list method.

Returns

Return type None

set_property(*name*, *prop_name*, *prop_value*, *ignore_items=False*)
Access and set the value of a meta object's `properties` collection.

Parameters

- **name** (*str*) – The originating column variable name keyed in `meta['columns']` or `meta['masks']`.
- **prop_name** (*str*) – The property key name.
- **prop_value** (*any*) – The value to be set for the property. Must be of valid type and have allowed values(s) with regard to the property.
- **ignore_items** (*bool, default False*) – When `name` refers to a variable from the `'masks'` collection, setting to True will ignore any `items` and only apply the property to the `mask` itself.

Returns

Return type None

set_text_key(*text_key*)
Set the default text_key of the `DataSet`.

Note: A lot of the instance methods will fall back to the default text key in `_meta['lib']['default text']`. It is therefore important to use this method with caution, i.e. ensure that the meta contains `text` entries for the `text_key` set.

Parameters **text_key** (*{'en-GB', 'da-DK', 'fi-FI', 'nb-NO', 'sv-SE', 'de-DE'}*) – The text key that will be set in `_meta['lib']['default text']`.

Returns

Return type None

set_value_texts(*name*, *renamed_vals*, *text_key=None*, *axis_edit=None*)
Rename or add value texts in the 'values' object.

This method works for array masks and column meta data.

Parameters

- **name** (*str*) – The column variable name keyed in `_meta['columns']` or `_meta['masks']`.
- **renamed_vals** (*dict*) – A dict mapping with following structure: `{1: 'new label for code=1', 5: 'new label for code=5'}` Codes will be ignored if they do not exist in the 'values' object.

- **text_key** (*str, default None*) – Text key for text-based label information. Will automatically fall back to the instance's text_key property information if not provided.
- **axis_edit** (*{'x', 'y', ['x', 'y']}, default None*) – If renamed_vals should only be considered temp. for build exports, the axes on that the edited text should appear can be provided.

Returns The DataSet is modified inplace.

Return type None

set_variable_text(*name*, *new_text*, *text_key=None*, *axis_edit=None*)

Apply a new or update a column's/masks' meta text object.

Parameters

- **name** (*str*) – The originating column variable name keyed in meta['columns'] or meta['masks'].
- **new_text** (*str*) – The text (label) to be set.
- **text_key** (*str, default None*) – Text key for text-based label information. Will automatically fall back to the instance's text_key property information if not provided.
- **axis_edit** (*{'x', 'y', ['x', 'y']}, default None*) – If the new_text of the variable should only be considered temp. for build exports, the axes on that the edited text should appear can be provided.

Returns The DataSet is modified inplace.

Return type None

set_verbose_errmsg(*verbose=True*)

set_verbose_infomsg(*verbose=True*)

slicing(*name*, *slicer*, *axis='y'*)

Set or update rules[axis]['slicex'] meta for the named column.

Quantipy builds will respect the kept codes and *show them exclusively* in results.

Note: This is not a replacement for DataSet.set_missings() as missing values are respected also in computations.

Parameters

- **name** (*str or list of str*) – The column variable(s) name keyed in _meta['columns'].
- **slice** (*int or list of int*) – Values indicated by their int codes will be shown in Quantipy.View.dataframes, respecting the provided order.
- **axis** (*{'x', 'y'}, default 'y'*) – The axis to slice the values on.

Returns

Return type None

sorting(*name*, *on='@'*, *within=False*, *between=False*, *fix=None*, *ascending=False*, *sort_by_weight='auto'*)

Set or update rules['x']['sortx'] meta for the named column.

Parameters

- **name** (*str or list of str*) – The column variable(s) name keyed in `_meta['columns']`.
- **within** (*bool, default True*) – Applies only to variables that have been aggregated by creating a an `expand` grouping / overcode-style `View`: If True, will sort frequencies inside each group.
- **between** (*bool, default True*) – Applies only to variables that have been aggregated by creating a an `expand` grouping / overcode-style `View`: If True, will sort group and regular code frequencies with regard to each other.
- **fix** (*int or list of int, default None*) – Values indicated by their `int` codes will be ignored in the sorting operation.
- **ascending** (*bool, default False*) – By default frequencies are sorted in descending order. Specify `True` to sort ascending.

Returns

Return type None

sources(*name*)

Get the `_meta['columns']` elements for the passed array mask name.

Parameters **name** (*str*) – The mask variable name keyed in `_meta['masks']`.

Returns **sources** – The list of source elements from the array definition.

Return type list

split(*save=False*)

Return the `meta` and `data` components of the DataSet instance.

Parameters **save** (*bool, default False*) – If True, the `meta` and `data` objects will be saved to disk, using the instance's `name` and `path` attributes to determine the file location.

Returns **meta, data** – The meta dict and the case data DataFrame as separate objects.

Return type dict, pandas.DataFrame

static start_meta(*text_key='main'*)

Starts a new/empty Quantipy meta document.

Parameters **text_key** (*str, default None*) – The default text key to be set into the new meta document.

Returns **meta** – Quantipy meta object

Return type dict

subset(*variables=None*, *from_set=None*, *inplace=False*)

Create a cloned version of self with a reduced collection of variables.

Parameters

- **variables** (*str or list of str, default None*) – A list of variable names to include in the new DataSet instance.
- **from_set** (*str*) – The name of an already existing set to base the new DataSet on.

Returns **subset_ds** – The new reduced version of the DataSet.

Return type qp.DataSet

take(*condition*)

Create an index slicer to select rows from the DataFrame component.

Parameters **condition** (*Quantipy logic expression*) – A logical condition expressed as Quantipy logic that determines which subset of the case data rows to be kept.

Returns **slicer** – The indices fulfilling the passed logical condition.

Return type pandas.Index

text (*name*, *shorten=True*, *text_key=None*, *axis_edit=None*)
Return the variables text label information.

Parameters

- **name** (*str, default None*) – The variable name keyed in `_meta['columns']` or `_meta['masks']`.
- **shorten** (*bool, default True*) – If True, `text` label meta from array items will not report the parent mask's `text`. Setting it to False will show the "full" label.
- **text_key** (*str, default None*) – The default text key to be set into the new meta document.
- **axis_edit** (*{'x', 'y'}, default None*) – If provided the text_key is taken from the x/y edits dict.

Returns **text** – The text metadata.

Return type str

to_array (*name*, *variables*, *label*, *safe=True*)
Combines column variables with same `values` meta into an array.

Parameters

- **name** (*str*) – Name of new grid.
- **variables** (*list of str or list of dicts*) – Variable names that become items of the array. New item labels can be added as dict. Example: variables = ['q1_1', {'q1_2': 'shop 2'}, {'q1_3': 'shop 3'}]
- **label** (*str*) – Text label for the mask itself.
- **safe** (*bool, default True*) – If True, the method will raise a `ValueError` if the provided variable name is already present in self. Select `False` to forcefully overwrite an existing variable with the same name (independent of its type).

Returns

Return type None

to_delimited_set (*name*, *label*, *variables*, *from_dichotomous=True*, *codes_from_name=True*)
Combines multiple single variables to new delimited set variable.

Parameters

- **name** (*str*) – Name of new delimited set
- **label** (*str*) – Label text for the new delimited set.
- **variables** (*list of str or list of tuples*) – variables that get combined into the new delimited set. If they are dichotomous (from_dichotomous=True), the labels of the variables are used as category texts or if tuples are included, the second items will be used for the category texts. If the variables are categorical (from_dichotomous=False) the values of the variables need to be eqaul and are taken for the delimited set.
- **from_dichotomous** (*bool, default True*) – Define if the input variables are dichotomous or categorical.

- **codes_from_name** (*bool, default True*) – If from_dichotomous=True, the codes can be taken from the Variable names, if they are in form of 'q01_1', 'q01_3', ... In this case the codes will be 1, 3, ...

Returns

Return type None

transpose(*name*, *new_name=None*, *ignore_items=None*, *ignore_values=None*, *copy_data=True*, *text_key=None*, *overwrite=False*)

Create a new array mask with transposed items / values structure.

This method will automatically create meta and case data additions in the `DataSet` instance.

Parameters

- **name** (*str*) – The originating mask variable name keyed in `meta['masks']`.
- **new_name** (*str, default None*) – The name of the new mask. If not provided explicitly, the new_name will be constructed constructed by suffixing the original `name` with '_trans', e.g. `'Q2Array_trans`.
- **ignore_items** (*int or list of int, default None*) – If provided, the items listed by their order number in the `_meta['masks'][name]['items']` object will not be part of the transposed array. This means they will be ignored while creating the new value codes meta.
- **ignore_codes** (*int or list of int, default None*) – If provided, the listed code values will not be part of the transposed array. This means they will not be part of the new item meta.
- **text_key** (*str*) – The text key to be used when generating text objects, i.e. item and value labels.
- **overwrite** (*bool, default False*) – Overwrite variable if *new_name* is already included.

Returns DataSet is modified inplace.

Return type None

unbind(*name*)

Remove mask-structure for arrays

uncode(*target*, *mapper*, *default=None*, *intersect=None*, *inplace=True*)

Create a new or copied series from data, recoded using a mapper.

Parameters

- **target** (*str*) – The variable name that is the target of the uncode. If it is keyed in `_meta['masks']` the uncode is done for all mask items. If not found in `_meta` this will fail with an error.
- **mapper** (*dict*) – A mapper of {key: logic} entries.
- **default** (*str, default None*) – The column name to default to in cases where unattended lists are given in your logic, where an auto-transformation of {key: list} to {key: {default: list}} is provided. Note that lists in logical statements are themselves a form of shorthand and this will ultimately be interpreted as: {key: {default: has_any(list)}}.
- **intersect** (*logical statement, default None*) – If a logical statement is given here then it will be used as an implied intersection of all logical conditions given in the mapper.

- **inplace** (*bool, default True*) – If True, the DataSet will be modified inplace with new/updated columns. Will return a new recoded pandas.Series instance if False.

Returns Either the DataSet._data is modified inplace or a new pandas.Series is returned.

Return type None or uncode_series

undimensionize (*names=None*, *mapper_to_meta=False*)
Rename the dataset columns to remove Dimensions compatibility.

undimensionizing_mapper (*names=None*)
Return a renaming dataset mapper for un-dimensionizing names.

Parameters **None** –

Returns **mapper** – A renaming mapper in the form of a dict of {old: new} that maps Dimensions naming conventions to non-Dimensions naming conventions.

Return type dict

unify_values (*name*, *code_map*, *slicer=None*, *exclusive=False*)
Use a mapping of old to new codes to replace code values in _data.

Note: Experimental! Check results carefully!

Parameters

- **name** (*str*) – The column variable name keyed in meta['columns'].
- **code_map** (*dict*) – A mapping of {old: new}; old and new must be the int-type code values from the column meta data.
- **slicer** (*Quantipy logic statement, default None*) – If provided, the values will only be unified for cases where the condition holds.
- **exclusive** (*bool, default False*) – If True, the recoded unified value will replace whatever is already found in the _data column, ignoring delimited set typed data to which normally would get appended to.

Returns

Return type None

unroll (*varlist*, *keep=None*, *both=None*)
Replace mask with their items, optionally excluding/keeping certain ones.

Parameters

- **varlist** (*list*) – A list of meta 'columns' and/or 'masks' names.
- **keep** (*str or list, default None*) – The names of masks that will not be replaced with their items.
- **both** (*'all', str or list of str, default None*) – The names of masks that will be included both as themselves and as collections of their items.

Note: varlist can also contain nesting *var1 > var2*. The variables which are included in the nesting can also be controlled by keep and both, even if the variables are also included as a "normal" variable.

Example::

```
>>> ds.unroll(varlist = ['q1', 'q1 > gender'], both='all')
['q1',
 'q1_1',
 'q1_2',
 'q1 > gender',
 'q1_1 > gender',
 'q1_2 > gender']
```

Returns **unrolled** – The modified `varlist`.

Return type list

update(*data*, *on='identity'*, *text_properties=None*)

Update the `DataSet` with the case data entries found in `data`.

Parameters

- **data** (`pandas.DataFrame`) – A dataframe that contains a subset of columns from the `DataSet` case data component.
- **on** (*str, default 'identity'*) – The column to use as a join key.
- **text_properties** (*str/ list of str, default=None, {'all', [var_names]}*) – Controls the update of the dataset_left properties with properties from the dataset_right. If None, properties from dataset_left will be updated by the ones from the dataset_right. If 'all', properties from dataset_left will be kept unchanged. Otherwise, specify the list of properties which will be kept unchanged in the dataset_left; all others will be updated by the properties from dataset_right.

Returns DataSet is modified inplace.

Return type None

used_text_keys()

Get a list of all used textkeys in the dataset instance.

validate(*spss_limits=False*, *verbose=True*)

Identify and report inconsistencies in the `DataSet` instance.

name: column/mask name and `meta[collection][var]['name']` are not identical

q_label: text object is badly formatted or has empty text mapping

values: categorical variable does not contain values, value text is badly formatted or has empty text mapping

text_keys: dataset.text_key is not included or existing text keys are not consistent (also for parents)

source: parents or items do not exist

codes: codes in data component are not included in meta component

spss limit name: length of name is greater than spss limit (64 characters) (only shown if spss_limits=True)

spss limit q_label: length of q_label is greater than spss limit (256 characters) (only shown if spss_limits=True)

spss limit values: length of any value text is greater than spss limit (120 characters) (only shown if spss_limits=True)

value_texts(*name*, *text_key=None*, *axis_edit=None*)
Get categorical data's text information.

Parameters

- **name** (*str*) – The column variable name keyed in `_meta['columns']`.
- **text_key** (*str, default None*) – The text_key that should be used when taking labels from the source meta.
- **axis_edit** (*{'x', 'y'}, default None*) – If provided the text_key is taken from the x/y edits dict.

Returns **texts** – The list of category texts.

Return type list

values(*name*, *text_key=None*, *axis_edit=None*)
Get categorical data's paired code and texts information from the meta.

Parameters

- **name** (*str*) – The column variable name keyed in `_meta['columns']` or `_meta['masks']`.
- **text_key** (*str, default None*) – The text_key that should be used when taking labels from the source meta.
- **axis_edit** (*{'x', 'y'}, default None*) – If provided the text_key is taken from the x/y edits dict.

Returns **values** – The list of the numerical category codes and their `texts` packed as tuples.

Return type list of tuples

variables(*setname='data file'*, *numeric=True*, *string=True*, *date=True*, *boolean=True*, *blacklist=None*)
View all DataSet variables listed in their global order.

Parameters

- **setname** (*str, default 'data file'*) – The name of the variable set to query. Defaults to the main variable collection stored via 'data file'.
- **numeric** (*bool, default True*) – Include `int` and `float` type variables?
- **string** (*bool, default True*) – Include `string` type variables?
- **date** (*bool, default True*) – Include `date` type variables?
- **boolean** (*bool, default True*) – Include `boolean` type variables?
- **blacklist** (*list, default None*) – A list of variables names to exclude from the variable listing.

Returns **varlist** – The list of variables registered in the queried `set`.

Return type list

vmerge(*dataset*, *on=None*, *left_on=None*, *right_on=None*, *row_id_name=None*, *left_id=None*, *right_id=None*, *row_ids=None*, *overwrite_text=False*, *from_set=None*, *uniquify_key=None*, *reset_index=True*, *inplace=True*, *text_properties=None*, *verbose=True*)
Merge Quantipy datasets together by appending rows.

This function merges two Quantipy datasets together, updating variables that exist in the left dataset and appending others. New variables will be appended in the order indicated by the 'data file' set if found, otherwise they will be appended in alphanumeric order. This merge happens vertically (row-wise).

Parameters

- **dataset** ((A list of multiple) quantipy.DataSet) – One or multiple datasets to merge into the current DataSet.
- **on** (*str, default=None*) – The column to use to identify unique rows in both datasets.
- **left_on** (*str, default=None*) – The column to use to identify unique in the left dataset.
- **right_on** (*str, default=None*) – The column to use to identify unique in the right dataset.
- **row_id_name** (*str, default=None*) – The named column will be filled with the ids indicated for each dataset, as per left_id/right_id/row_ids. If meta for the named column doesn't already exist a new column definition will be added and assigned a reductive-appropriate type.
- **left_id** (*str/int/float, default=None*) – Where the row_id_name column is not already populated for the dataset_left, this value will be populated.
- **right_id** (*str/int/float, default=None*) – Where the row_id_name column is not already populated for the dataset_right, this value will be populated.
- **row_ids** (*list of str/int/float, default=None*) – When datasets has been used, this list provides the row ids that will be populated in the row_id_name column for each of those datasets, respectively.
- **overwrite_text** (*bool, default=False*) – If True, text_keys in the left meta that also exist in right meta will be overwritten instead of ignored.
- **from_set** (*str, default=None*) – Use a set defined in the right meta to control which columns are merged from the right dataset.
- **uniquify_key** (*str, default None*) – A int-like column name found in all the passed DataSet objects that will be protected from having duplicates. The original version of the column will be kept under its name prefixed with 'original'.
- **reset_index** (*bool, default=True*) – If True pandas.DataFrame.reindex() will be applied to the merged dataframe.
- **inplace** (*bool, default True*) – If True, the DataSet will be modified inplace with new/updated rows. Will return a new DataSet instance if False.
- **merge_existing** (*str/ list of str, default None, {'all', [var_names]}*) – Merge values for defined delimited sets if it exists in both datasets. (update_existing is prioritized)
- **text_properties** (*str/ list of str, default=None, {'all', [var_names]}*) – Controls the update of the dataset_left properties with properties from the dataset_right. If None, properties from dataset_left will be updated by the ones from the dataset_right. If 'all', properties from dataset_left will be kept unchanged. Otherwise, specify the list of properties which will be kept unchanged in the dataset_left; all others will be updated by the properties from dataset_right.
- **verbose** (*bool, default=True*) – Echo progress feedback to the output pane.

Returns **None or new_dataset** – If the merge is not applied inplace, a DataSet instance is returned.

Return type quantipy.DataSet

weight(*weight_scheme*, *weight_name='weight'*, *unique_key='identity'*, *subset=None*, *report=True*, *path_report=None*, *inplace=True*, *verbose=True*)

Weight the `DataSet` according to a well-defined weight scheme.

Parameters

- **weight_scheme** (*quantipy.Rim instance*) – A rim weights setup with defined targets. Can include multiple weight groups and/or filters.
- **weight_name** (*str, default 'weight'*) – A name for the float variable that is added to pick up the weight factors.
- **unique_key** (*str, default 'identity'.*) – A variable inside the `DataSet` instance that will be used to the map individual case weights to their matching rows.
- **subset** (*Quantipy complex logic expression*) – A logic to filter the DataSet, weighting only the remaining subset.
- **report** (*bool, default True*) – If True, will report a summary of the weight algorithm run and factor outcomes.
- **path_report** (*str, default None*) – A file path to save an .xlsx version of the weight report to.
- **inplace** (*bool, default True*) – If True, the weight factors are merged back into the `DataSet` instance. Will otherwise return the `pandas.DataFrame` that contains the weight factors, the `unique_key` and all variables that have been used to compute the weights (filters, target variables, etc.).

Returns Will either create a new column called `'weight'` in the `DataSet` instance or return a `DataFrame` that contains the weight factors.

Return type None or `pandas.DataFrame`

write_dimensions(*path_mdd=None*, *path_ddf=None*, *text_key=None*, *run=True*, *clean_up=True*, *CRLF='CR'*)

Build Dimensions/SPSS Base Professional .ddf/.mdd data pairs.

Note: SPSS Data Collection Base Professional must be installed on the machine. The method is creating .mrs and .dms scripts which are executed through the software's API.

Parameters

- **path_mdd** (*str, default None*) – The full path (optionally with extension `'.mdd'`, otherwise assumed as such) for the saved the DataSet._meta component. If not provided, the instance's `name` and `` `path `` attributes will be used to determine the file location.
- **path_ddf** (*str, default None*) – The full path (optionally with extension `'.ddf'`, otherwise assumed as such) for the saved DataSet._data component. If not provided, the instance's `name` and `` `path `` attributes will be used to determine the file location.
- **text_key** (*str, default None*) – The desired `text_key` for all `text` label information. Uses the `DataSet.text_key` information if not provided.
- **run** (*bool, default True*) – If True, the method will try to run the metadata creating .mrs script and execute a DMSRun for the case data transformation in the .dms file.
- **clean_up** (*bool, default True*) – By default, all helper files from the conversion (.dms, .mrs, paired .csv files, etc.) will be deleted after the process has finished.

Returns

Return type A .ddf/.mdd pair is saved at the provided path location.

write_quantipy(*path_meta=None*, *path_data=None*)
Write the data and meta components to .csv/.json files.

The resulting files are well-defined native Quantipy source files.

Parameters

- **path_meta** (*str, default None*) – The full path (optionally with extension '.json', otherwise assumed as such) for the saved the DataSet._meta component. If not provided, the instance's name and `path attributes will be used to determine the file location.
- **path_data** (*str, default None*) – The full path (optionally with extension '.csv', otherwise assumed as such) for the saved DataSet._data component. If not provided, the instance's name and `path attributes will be used to determine the file location.

Returns

Return type A .csv/.json pair is saved at the provided path location.

write_spss(*path_sav=None*, *index=True*, *text_key=None*, *mrset_tag_style='__'*, *drop_delimited=True*, *from_set=None*, *verbose=True*)
Convert the Quantipy DataSet into a SPSS .sav data file.

Parameters

- **path_sav** (*str, default None*) – The full path (optionally with extension '.json', otherwise assumed as such) for the saved the DataSet._meta component. If not provided, the instance's name and `path attributes will be used to determine the file location.
- **index** (*bool, default False*) – Should the index be inserted into the dataframe before the conversion happens?
- **text_key** (*str, default None*) – The text_key that should be used when taking labels from the source meta. If the given text_key is not found for any particular text object, the DataSet.text_key will be used instead.
- **mrset_tag_style** (*str, default '__'*) – The delimiting character/string to use when naming dichotomous set variables. The mrset_tag_style will appear between the name of the variable and the dichotomous variable's value name, as taken from the delimited set value that dichotomous variable represents.
- **drop_delimited** (*bool, default True*) – Should Quantipy's delimited set variables be dropped from the export after being converted to dichotomous sets/mrsets?
- **from_set** (*str*) – The set name from which the export should be drawn.

Returns

Return type A SPSS .sav file is saved at the provided path location.

9.4 quantify.engine

class quantipy.**Quantity**(*link*, *weight=None*, *base_all=False*, *ignore_flags=False*)
The Quantity object is the main Quantipy aggregation engine.

Consists of a link's data matrix representation and sectional defintion of weight vector (wv), x-codes section (xsect) and y-codes section (ysect). The instance methods handle creation, retrieval and manipulation of the data input matrices and section definitions as well as the majority of statistical calculations.

calc(*expression*, *axis='x'*, *result_only=False*)
Compute (simple) aggregation level arithmetics.

count(*axis=None*, *raw_sum=False*, *cum_sum=False*, *effective=False*, *margin=True*, *as_df=True*)
Count entries over all cells or per axis margin.

Parameters

- **axis** (*{None, 'x', 'y'}, deafult None*) – When axis is None, the frequency of all cells from the uni- or multivariate distribution is presented. If the axis is specified to be either 'x' or 'y' the margin per axis becomes the resulting aggregation.
- **raw_sum** (*bool, default False*) – If True will perform a simple summation over the cells given the axis parameter. This ignores net counting of qualifying answers in favour of summing over all answers given when considering margins.
- **cum_sum** (*bool, default False*) – If True a cumulative sum of the elements along the given axis is returned.
- **effective** (*bool, default False*) – If True, compute effective counts instead of traditional (weighted) counts.
- **margin** (*bool, deafult True*) – Controls whether the margins of the aggregation result are shown. This also applies to margin aggregations themselves, since they contain a margin in (form of the total number of cases) as well.
- **as_df** (*bool, default True*) – Controls whether the aggregation is transformed into a Quantipy- multiindexed (following the Question/Values convention) pandas.DataFrame or will be left in its numpy.array format.

Returns Passes a pandas.DataFrame or numpy.array of cell or margin counts to the `result` property.

Return type self

exclude(*codes*, *axis='x'*)
Wrapper for _missingfy(...keep_codes=False, ..., keep_base=False, ...) Excludes specified codes from aggregation.

filter(*condition*, *keep_base=True*, *inplace=False*)
Use a Quantipy conditional expression to filter the data matrix entires.

group(*groups*, *axis='x'*, *expand=None*, *complete=False*)
Build simple or logical net vectors, optionally keeping orginating codes.

Parameters

- **groups** (*list, dict of lists or logic expression*) – The group/net code defintion(s) in form of...
  - a simple list: `[1, 2, 3]`
  - a dict of list: `{'grp A': [1, 2, 3], 'grp B': [4, 5, 6]}`
  - a logical expression: `not_any([1, 2])`
- **axis** ({`'x'`, `'y'`}, default `'x'`) – The axis to group codes on.

- **expand** ({None, 'before', 'after'}, default None) – If 'before', the codes that are grouped will be kept and placed before the grouped aggregation; vice versa for 'after'. Ignored on logical expressions found in groups.
- **complete** (*bool, default False*) – If True, codes that define the Link on the given axis but are not present in the groups defintion(s) will be placed in their natural position within the aggregation, respecting the value of expand.

Returns

Return type None

limit(*codes*, *axis='x'*)

Wrapper for _missingfy(...keep_codes=True, ..., keep_base=True, ...) Restrict the data matrix entires to contain the specified codes only.

normalize(*on='y'*, *per_cell=False*)

Convert a raw cell count result to its percentage representation.

Parameters

- **on** (*{'y', 'x', 'counts_sum', str}, default 'y'*) – Defines the base to normalize the result on. 'y' will produce column percentages, 'x' will produce row percentages. It is also possible to use another question's frequencies to compute rebased percentages providing its name instead.
- **per_cell** (*bool, default False*) – Compute percentages on a cell-per-cell basis, effectively treating each categorical row as a base figure on its own. Only possible if the on argument does not indidcate an axis result ('x', 'y', 'counts_sum'), but instead another variable's name. The related xdef codes collection length must be identical for this for work, otherwise a ValueError is raised.

Returns Updates a count-based aggregation in the result property.

Return type self

rescale(*scaling*, *drop=False*)

Modify the object's xdef property reflecting new value defintions.

Parameters

- **scaling** (*dict*) – Mapping of old_code: new_code, given as of type int or float.
- **drop** (*bool, default False*) – If True, codes not included in the scaling dict will be excluded.

Returns

Return type self

summarize(*stat='summary'*, *axis='x'*, *margin=True*, *as_df=True*)

Calculate distribution statistics across the given axis.

Parameters

- **stat** (*{'summary', 'mean', 'median', 'var', 'stddev', 'sem', varcoeff',}*) – 'min', 'lower_q', 'upper_q', 'max'}, default 'summary' The measure to calculate. Defaults to a summary output of the most important sample statistics.
- **axis** (*{'x', 'y'}, default 'x'*) – The axis which is reduced in the aggregation, e.g. column vs. row means.
- **margin** (*bool, default True*) – Controls whether statistic(s) of the marginal distribution are shown.

- **as_df** (*bool, default True*) – Controls whether the aggregation is transformed into a Quantipy- multiindexed (following the Question/Values convention) pandas.DataFrame or will be left in its numpy.array format.

Returns Passes a pandas.DataFrame or numpy.array of the descriptive (summary) statistic(s) to the `result` property.

Return type self

swap(*var*, *axis='x'*, *update_axis_def=True*, *inplace=True*)
Change the Quantity's x- or y-axis keeping filter and weight setup.

All edits and aggregation results will be removed during the swap.

Parameters

- **var** (*str*) – New variable's name used in axis swap.
- **axis** ({'x', 'y'}, default `'x'`) – The axis to swap.
- **update_axis_def** (*bool, default False*) – If self is of type `'array'`, the name and item definitions (that are e.g. used in the `to_df()` method) can be updated to reflect the swapped axis variable or kept to show the original's ones.
- **inplace** (*bool, default True*) – Whether to modify the Quantity inplace or return a new instance.

Returns swapped

Return type New Quantity instance with exchanged x- or y-axis.

unweight()
Remove any weighting by dividing the matrix by itself.

weight()
Weight by multiplying the indicator entries with the weight vector.

class quantipy.**Test**(*link*, *view_name_notation*, *test_total=False*)
The Quantipy Test object is a defined by a Link and the view name notation string of a counts or means view. All auxiliary figures needed to arrive at the test results are computed inside the instance of the object.

get_se()
Compute the standard error (se) estimate of the tested metric.

The calculation of the se is defined by the parameters of the setup. The main difference is the handling of variances. **unpooled** implicitly assumes variance inhomogenity between the column pairing's samples. **pooled** treats variances effectively as equal.

get_sig()
TODO: implement returning tstats only.

get_statistic()
Returns the test statistic of the algorithm.

run()
Performs the testing algorithm and creates an output pd.DataFrame.

The output is indexed according to Quantipy's Questions->Values convention. Significant results between columns are presented as lists of integer y-axis codes where the column with the higher value is holding the codes of the columns with the lower values. NaN is indicating that a cell is not holding any sig. higher values compared to the others.

set_params(*test_total=False*, *level='mid'*, *mimic='Dim'*, *testtype='pooled'*, *use_ebase=True*, *ovlp_correc=True*, *cwi_filter=False*, *flag_bases=None*)
Sets the test algorithm parameters and defines the type of test.

This method sets the test's global parameters and derives the necessary measures for the computation of the test statistic. The default values correspond to the SPSS Dimensions Column Tests algorithms that control for bias introduced by weighting and overlapping samples in the column pairs of multi-coded questions.

Note: The Dimensions implementation uses variance pooling.

Parameters

- **test_total** (*bool, default False*) – If set to True, the test algorithms will also include an existent total (@-) version of the original link and test against the unconditial data distribution.
- **level** (*str or float, default 'mid'*) – The level of significance given either as per 'low' = 0.1, 'mid' = 0.05, 'high' = 0.01 or as specific float, e.g. 0.15.
- **mimic** (*{'askia', 'Dim'} default='Dim'*) – Will instruct the mimicking of a software specific test.
- **testtype** (*str, default 'pooled'*) – Global definition of the tests.
- **use_ebase** (*bool, default True*) – If True, will use the effective sample sizes instead of the the simple weighted ones when testing a weighted aggregation.
- **ovlp_correc** (*bool, default True*) – If True, will consider and correct for respondent overlap when testing between multi-coded column pairs.
- **cwi_filter** (*bool, default False*) – If True, will check an incoming count aggregation for cells that fall below a treshhold comparison aggregation that assumes counts to be independent.
- **flag_bases** (*list of two int, default None*) – If provided, the output dataframe will replace results that have been calculated on (eff.) bases below the first int with `'**'` and mark results in columns with bases below the second int with `'*'`

Returns

Return type self

9.5 QuantipyViews

class quantipy.**QuantipyViews**(*views=None*, *template=None*)

A collection of extendable MR aggregation and statistic methods.

View methods are used to generate various numerical or categorical data aggregations. Their behaviour is controlled via `kwargs`.

coltests(*link*, *name*, *kwargs*)

Will test appropriate views from a Stack for stat. sig. differences.

Tests can be performed on frequency aggregations (generated by `frequency`) and means (from `summarize`) and will compare all unique column pair combinations.

Parameters

- **link** (*Quantipy Link object.*) –
- **name** (*str*) – The shortname applied to the view.
- **kwargs** (*dict*) –

- **arguments (specific)** (*Keyword*) –
- **text** (*str, optional, default None*) – Sets an optional label in the meta component of the view that is used when the view is passed into a Quantipy build (e.g. Excel, Powerpoint).
- **metric** (*{'props', 'means'}, default 'props'*) – Determines whether a proportion or means test algorithm is performed.
- **test_total** (*bool, deafult False*) – If True, the each View's y-axis column will be tested against the uncoditional total of its x-axis.
- **mimic** (*{'Dim', 'askia'}, default 'Dim'*) – It is possible to mimic the test logics used in other statistical software packages by passing them as instructions. The method will then choose the appropriate test parameters.
- **level** (*{'high', 'mid', 'low'} or float*) – Sets the level of significance to which the test is carried out. Given as str the levels correspond to `'high'` = 0.01, `'mid'` = 0.05 and `'low'` = 0.1. If a float is passed the specified level will be used.
- **flags** (*list of two int, default None*) – Base thresholds for Dimensions-like tests, e.g. [30, 100]. First int is minimum base for reported results, second int controls small base indication.

Returns

- *None* – Adds requested View to the Stack, storing it under the full view name notation key.
- .. *note::* – Mimicking the askia software (`mimic = 'askia'`) restricts the values to be one of `'high'`, `'low'`, `'mid'`. Any other value passed will make the algorithm fall back to `'low'`. Mimicking Dimensions (`mimic = 'Dim'`) can use either the str or float version.

default (*link*, *name*, *kwargs*)

Adds a file meta dependent aggregation to a Stack.

Checks the Link definition against the file meta and produces either a numerical or categorical summary tabulation including marginal the results.

Parameters

- **link** (*Quantipy Link object.*) –
- **name** (*str*) – The shortname applied to the view.
- **kwargs** (*dict*) –

Returns Adds requested View to the Stack, storing it under the full view name notation key.

Return type None

descriptives (*link*, *name*, *kwargs*)

Adds num. distribution statistics of a Link defintion to the Stack.

`descriptives` views can apply a range of summary statistics. Measures include statistics of centrality, dispersion and mass.

Parameters

- **link** (*Quantipy Link object.*) –
- **name** (*str*) – The shortname applied to the view.
- **kwargs** (*dict*) –
- **arguments (specific)** (*Keyword*) –

- **text** (*str, optional, default None*) – Sets an optional label suffix for the meta component of the view which will be appended to the statistic name and used when the view is passed into a Quantipy build (e.g. Excel, Powerpoint).
- **stats** (*str, default 'mean'*) – The measure to compute.
- **exclude** (*list of int*) – Codes that will not be considered calculating the result.
- **rescale** (*dict*) – A mapping of {old code: new code}, e.g.:

```
{
 1: 0,
 2: 25,
 3: 50,
 4: 75,
 5: 100
}
```

- **drop** (*bool*) – If `rescale` provides a new scale defintion, `drop` will remove all codes that are not transformed. Acts as a shorthand for manually passing any remaining codes in `exclude`.

Returns Adds requested View to the Stack, storing it under the full view name notation key.

Return type None

frequency (*link*, *name*, *kwargs*)

Adds count-based views on a Link defintion to the Stack object.

`frequency` is able to compute several aggregates that are based on the count of code values in uni- or bivariate Links. This includes bases / samples sizes, raw or normalized cell frequencies and code summaries like simple and complex nets.

Parameters

- **link** (*Quantipy Link object.*) –
- **name** (*str*) – The shortname applied to the view.
- **kwargs** (*dict*) –
- **arguments (specific)** (*Keyword*) –
- **text** (*str, optional, default None*) – Sets an optional label in the meta component of the view that is used when the view is passed into a Quantipy build (e.g. Excel, Powerpoint).
- **logic** (*list of int, list of dicts or core.tools.view.logic operation*) – If a list is passed this instructs a simple net of the codes given as int. Multiple nets can be generated via a list of dicts that map names to lists of ints. For complex logical statements, expression are parsed to identify the qualifying rows in the data. For example:

```
# simple net
'logic': [1, 2, 3]

# multiple nets/code groups
'logic': [{'A': [1, 2]}, {'B': [3, 4]}, {'C', [5, 6]}]

# code logic
'logic': has_all([1, 2, 3])
```

- **calc** (*TODO*) –

- **calc_only** (*TODO*) –

Returns

- *None* – Adds requested View to the Stack, storing it under the full view name notation key.
- *.. note:: Net codes take into account if a variable is* – multi-coded. The net will therefore consider qualifying cases and not the raw sum of the frequencies per category, i.e. no multiple counting of cases.

9.6 Rim

class quantipy.**Rim**(*name*, *max_iterations=1000*, *convcrit=0.01*, *cap=0*, *dropna=True*, *impute_method='mean'*, *weight_column_name=None*, *total=0*)

add_group(*name=None*, *filter_def=None*, *targets=None*)
Set weight groups using flexible filter and target defintions.

Main method to structure and specify complex weight schemes.

Parameters

- **name** (*str*) – Name of the weight group.
- **filter_def** (*str, optional*) – An optional filter defintion given as a boolean expression in string format. Must be a valid input for the pandas DataFrame.query() method.
- **targets** (*dict*) – Dictionary mapping of DataFrame columns to target proportion list.

Returns

Return type None

group_targets(*group_targets*)
Set inter-group target proportions.

This will scale the weight factors per group to match the desired group proportions and thus effectively change each group's weighted total number of cases.

Parameters **group_targets** (*dict*) – A dictionary mapping of group names to the desired proportions.

Returns

Return type None

report(*group=None*)
TODO: Docstring

set_targets(*targets*, *group_name=None*)
Quickly set simple weight targets, optionally assigning a group name.

Parameters

- **targets** (*dict or list of dict*) – Dictionary mapping of DataFrame columns to target proportion list.
- **group_name** (*str, optional*) – A name for the simple weight (group) created.

Returns

Return type None

validate()
Summary on scheme target variables to detect and handle missing data.

Returns **df** – A summary of missing entries and (rounded) mean/mode/median of value codes per target variable.

Return type pandas.DataFrame

9.7 Stack

class quantipy.**Stack**(*name=''*, *add_data=None*)
Container of quantipy.Link objects holding View objects.

A Stack is nested dictionary that structures the data and variable relationships storing all View aggregations performed.

add_data(*data_key*, *data=None*, *meta=None*)
Sets the data_key into the stack, optionally mapping data sources it.

It is possible to handle the mapping of data sources in different ways:

- no meta or data (for proxy links not connected to source data)
- meta only (for proxy links with supporintg meta)
- data only (meta will be inferred if possible)
- data and meta

Parameters

- **data_key** (*str*) – The reference name for a data source connected to the Stack.
- **data** (*pandas.DataFrame*) – The input (case) data source.
- **meta** (*dict or OrderedDict*) – A quantipy compatible metadata source that describes the case data.

Returns

Return type None

add_link(*data_keys=None, filters=['no_filter'], x=None, y=None, views=None, weights=None, variables=None*)
Add Link and View defintions to the Stack.

The method can be used flexibly: It is possible to pass only Link defintions that might be composed of filter, x and y specifications, only views incl. weight variable selections or arbitrary combinations of the former.

TODO Remove `variables` from parameter list and method calls.

Parameters

- **data_keys** (*str, optional*) – The data_key to be added to. If none is given, the method will try to add to all data_keys found in the Stack.
- **filters** (*list of str describing filter defintions, default ['no_filter']*) – The string must be a valid input for the pandas.DataFrame.query() method.
- **y** (*x,*) – The x and y variables to constrcut Links from.

- **views** (*list of view method names.*) – Can be any of Quantipy's preset Views or the names of created view method specifications.
- **weights** (*list, optional*) – The names of weight variables to consider in the data aggregation process. Weight variables must be of type `float`.

Returns

Return type None

add_nets (*on_vars*, *net_map*, *expand=None*, *calc=None*, *rebase=None*, *text_prefix='Net:'*, *checking_cluster=None*, *_batches='all'*, *recode='auto'*, *mis_in_rec=False*, *verbose=True*)

Add a net-like view to a specified collection of x keys of the stack.

Parameters

- **on_vars** (*list*) – The list of x variables to add the view to.
- **net_map** (*list of dicts*) – The listed dicts must map the net/band text label to lists of categorical answer codes to group together, e.g.:

```
>>> [{'Top3': [1, 2, 3]},
...  {'Bottom3': [4, 5, 6]}]
It is also possible to provide enumerated net definition␣
↪dictionaries
that are explicitly setting ``text`` metadata per ``text_key``␣
↪entries:
```

```
>>> [{1: [1, 2], 'text': {'en-GB': 'UK NET TEXT',
...                       'da-DK': 'DK NET TEXT',
...                       'de-DE': 'DE NET TEXT'}}]
```

- **expand** (*{'before', 'after'}, default None*) – If provided, the view will list the net-defining codes after or before the computed net groups (i.e. "overcode" nets).
- **calc** (*dict, default None*) – A dictionary that is attaching a text label to a calculation expression using the the net definitions. The nets are referenced as per 'net_1', 'net_2', 'net_3', ... Supported calculation expressions are add, sub, div, mul. Example:

```
>>> {'calc': ('net_1', add, 'net_2'), 'text': {'en-GB': 'UK CALC␣
↪LAB',
...                                         'da-DK': 'DA CALC␣
↪LAB',
...                                         'de-DE': 'DE CALC␣
↪LAB'}}
```

- **rebase** (*str, default None*) – Use another variables margin's value vector for column percentage computation.
- **text_prefix** (*str, default 'Net:'*) – By default each code grouping/net will have its `text` label prefixed with 'Net: '. Toggle by passing None (or an empty str, '').
- **checking_cluster** (`quantipy.Cluster`, *default None*) – When provided, an automated checking aggregation will be added to the `Cluster` instance.
- **_batches** (*str or list of str*) – Only for `qp.Links` that are defined in this `qp.Batch` instances views are added.
- **recode** ({'extend_codes', 'drop_codes', 'collect_codes', 'collect_codes@cat_name'},) – default 'auto' Adds variable with nets as codes to DataSet/Stack. If 'extend_codes', codes are extended with nets. If 'drop_codes', new variable only contains nets as codes.

If 'collect_codes' or 'collect_codes@cat_name' the variable contains nets and another category that summarises all codes which are not included in any net. If no cat_name is provided, 'Other' is taken as default

- **mis_in_rec** (*bool, default False*) – Skip or include codes that are defined as missing when recoding from net definition.

Returns The stack instance is modified inplace.

Return type None

add_stats(*on_vars, stats=['mean'], other_source=None, rescale=None, drop=True, exclude=None, factor_labels=True, custom_text=None, checking_cluster=None, _batches='all', recode=False, verbose=True*)

Add a descriptives view to a specified collection of xks of the stack.

Valid descriptives views: {'mean', 'stddev', 'min', 'max', 'median', 'sem'}

Parameters

- **on_vars** (*list*) – The list of x variables to add the view to.
- **stats** (list of str, default `['mean']`) – The metrics to compute and add as a view.
- **other_source** (*str*) – If provided the Link's x-axis variable will be swapped with the (numerical) variable provided. This can be used to attach statistics of a different variable to a Link definition.
- **rescale** (*dict*) – A dict that maps old to new codes, e.g. {1: 5, 2: 4, 3: 3, 4: 2, 5: 1}
- **drop** (*bool, default True*) – If `rescale` is provided all codes that are not mapped will be ignored in the computation.
- **exclude** (*list*) – Codes/values to ignore in the computation.
- **factor_labels** (*bool / str, default True*) – Writes the (rescaled) factor values next to the category text label. If True, square-brackets are used. If '()', normal brackets are used.
- **custom_text** (*str, default None*) – A custom string affix to put at the end of the requested statistics' names.
- **checking_cluster** (`quantipy.Cluster`, *default None*) – When provided, an automated checking aggregation will be added to the `Cluster` instance.
- **_batches** (*str or list of str*) – Only for `qp.Links` that are defined in this `qp.Batch` instances views are added.
- **recode** (*bool, default False*) – Create a new variable that contains only the values which are needed for the stat computation. The values and the included data will be rescaled.

Returns The stack instance is modified inplace.

Return type None

add_tests(*_batches='all', verbose=True*)

Apply coltests for selected batches.

Sig. Levels are taken from `qp.Batch` definitions.

Parameters **_batches** (*str or list of str*) – Only for `qp.Links` that are defined in this `qp.Batch` instances views are added.

Returns

Return type None

aggregate(*views*, *unweighted_base=True*, *categorize=[]*, *batches='all'*, *xs=None*, *bases={}*, *verbose=True*)

Add views to all defined `qp.Link` in `qp.Stack`.

Parameters

- **views** (*str or list of str or qp.ViewMapper*) – `views` that are added.
- **unweighted_base** (*bool, default True*) – If True, unweighted 'cbase' is added to all non-arrays. This parameter will be deprecated in future, please use bases instead.
- **categorize** (*str or list of str*) – Determines how numerical data is handled: If provided, the variables will get counts and percentage aggregations (`'counts'`, `'c%'`) alongside the `'cbase'` view. If False, only `'cbase'` views are generated for non-categorical types.
- **batches** (*str/ list of str, default 'all'*) – Name(s) of `qp.Batch` instance(s) that are used to aggregate the `qp.Stack`.
- **xs** (*list of str*) – Names of variable, for which views are added.
- **bases** (*dict*) – Defines which bases should be aggregated, weighted or unweighted.

Returns

Return type None, modify `qp.Stack` inplace

apply_meta_edits(*batch_name*, *data_key*, *filter_key=None*, *freeze=False*)

Take over meta_edits from Batch definitions.

Parameters

- **batch_name** (*str*) – Name of the Batch whose meta_edits are taken.
- **data_key** (*str*) – Accessing this metadata: `self[data_key].meta` Batch definitions are takes from here and this metadata is modified.
- **filter_key** (*str, default None*) – Currently not implemented! Accessing this metadata: `self[data_key][filter_key].meta` Batch definitions are takes from here and this metadata is modified.

cumulative_sum(*on_vars*, *_batches='all'*, *verbose=True*)

Add cumulative sum view to a specified collection of xks of the stack.

Parameters

- **on_vars** (*list*) – The list of x variables to add the view to.
- **_batches** (*str or list of str*) – Only for `qp.Links` that are defined in this `qp.Batch` instances views are added.

Returns The stack instance is modified inplace.

Return type None

describe(*index=None*, *columns=None*, *query=None*, *split_view_names=False*)

Generates a structured overview of all Link defining Stack elements.

Parameters

- **columns** (*index,*) – optional Controls the output representation by structuring a pivot-style table according to the index and column values.
- **query** (*str*) – A query string that is valid for the pandas.DataFrame.query() method.

- **split_view_names** (*bool, default False*) – If True, will create an output of unique view name notations split up into their components.

Returns description – DataFrame summing the Stack's structure in terms of Links and Views.

Return type pandas.DataFrame

freeze_master_meta(*data_key*, *filter_key=None*)
Save `.meta` in `.master_meta` for a defined data_key.

Parameters

- **data_key** (*str*) – Using: `self[data_key]`
- **filter_key** (*str, default None*) – Currently not implemented! Using: `self[data_key][filter_key]`

static from_sav(*data_key*, *filename*, *name=None*, *path=None*, *ioLocale='en_US.UTF-8'*, *ioUtf8=True*)
Creates a new stack instance from a .sav file.

Parameters

- **data_key** (*str*) – The data_key for the data and meta in the sav file.
- **filename** (*str*) – The name to the sav file.
- **name** (*str*) – A name for the sav (stored in the meta).
- **path** (*str*) – The path to the sav file.
- **ioLocale** (*str*) – The locale used in during the sav processing.
- **ioUtf8** (*bool*) – Boolean that indicates the mode in which text communicated to or from the I/O module will be.

Returns stack – A stack instance that has a data_key with data and metadata to run aggregations.

Return type stack object instance

static load(*path_stack*, *compression='gzip'*, *load_cache=False*)
Load Stack instance from .stack file.

Parameters

- **path_stack** (*str*) – The full path to the .stack file that should be created, including the extension.
- **compression** (*{'gzip'}, default 'gzip'*) – The compression type that has been used saving the file.
- **load_cache** (*bool, default False*) – Loads MatrixCache into the Stack a .cache file is found.

Returns

Return type None

static recode_from_net_def(*dataset*, *on_vars*, *net_map*, *expand*, *recode='auto'*, *text_prefix='Net:'*, *mis_in_rec=False*, *verbose=True*)
Create variables from net definitions.

reduce(*data_keys=None*, *filters=None*, *x=None*, *y=None*, *variables=None*, *views=None*)
Remove keys from the matching levels, erasing discrete Stack portions.

Parameters filters, x, y, views(*data_keys,*) –

Returns

Return type None

refresh(*data_key*, *new_data_key=''*, *new_weight=None*, *new_data=None*, *new_meta=None*)
Re-run all or a portion of Stack's aggregations for a given data key.

refresh() can be used to re-weight the data using a new case data weight variable or to re-run all aggregations based on a changed source data version (e.g. after cleaning the file/ dropping cases) or a combination of the both.

Note: Currently this is only supported for the preset QuantipyViews(), namely: `'cbase'`, `'rbase'`, `'counts'`, `'c%'`, `'r%'`, `'mean'`, `'ebase'`.

Parameters

- **data_key** (*str*) – The Links' data key to be modified.
- **new_data_key** (*str, default ''*) – Controls if the existing data key's files and aggregations will be overwritten or stored via a new data key.
- **new_weight** (*str*) – The name of a new weight variable used to re-aggregate the Links.
- **new_data** (*pandas.DataFrame*) – The case data source. If None is given, the original case data found for the data key will be used.
- **new_meta** (*quantipy meta document*) – A meta data source associated with the case data. If None is given, the original meta definition found for the data key will be used.

Returns

Return type None

remove_data(*data_keys*)
Deletes the data_key(s) and associated data specified in the Stack.

Parameters **data_keys** (*str or list of str*) – The data keys to remove.

Returns

Return type None

restore_meta(*data_key*, *filter_key=None*)
Restore the `.master_meta` for a defined data_key if it exists.

Undo self.apply_meta_edits()

Parameters

- **data_key** (*str*) – Accessing this metadata: `self[data_key].meta`
- **filter_key** (*str, default None*) – Currently not implemented! Accessing this metadata: `self[data_key][filter_key].meta`

save(*path_stack*, *compression='gzip'*, *store_cache=True*, *decode_str=False*, *dataset=False*, *describe=False*)
Save Stack instance to .stack file.

Parameters

- **path_stack** (*str*) – The full path to the .stack file that should be created, including the extension.
- **compression** (*{'gzip'}, default 'gzip'*) – The intended compression type.

- **store_cache** (*bool, default True*) – Stores the MatrixCache in a file in the same location.
- **decode_str** (*bool, default=True*) – If True the unicoder function will be used to decode all str objects found anywhere in the meta document/s.
- **dataset** (*bool, default=False*) – If True a json/csv will be saved parallel to the saved stack for each data key in the stack.
- **describe** (*bool, default=False*) – If True the result of stack.describe().to_excel() will be saved parallel to the saved stack.

Returns

Return type None

variable_types(*data_key*, *only_type=None*, *verbose=True*)

Group variables by data types found in the meta.

Parameters

- **data_key** (*str*) – The reference name of a case data source hold by the Stack instance.
- **only_type** (*{'int', 'float', 'single', 'delimited set', 'string',}*) – 'date', time', 'array'}, optional Will restrict the output to the given data type.

Returns types – A summary of variable names mapped to their data types, in form of {type_name: [variable names]} or a list of variable names confirming only_type.

Return type dict or list of str

9.8 View

class quantipy.**View**(*link=None*, *name=None*, *kwargs=None*)

get_edit_params()

Provides the View's Link edit kwargs with fallbacks to default values.

Returns edit_params – A tuple of kwargs controlling the following supported Link data edits: logic, calc, ...

Return type tuple

get_std_params()

Provides the View's standard kwargs with fallbacks to default values.

Returns std_parameters – A tuple of the common kwargs controlling the general View method behaviour: axis, relation, rel_to, weights, text

Return type tuple

has_other_source()

Tests if the View is generated with a swapped x-axis.

is_base()

Tests if the View is a base size aggregation.

is_counts()

Tests if the View is a count representation of a frequency.

is_cumulative()
Tests if the View is a cumulative frequency.

is_meanstest()
Tests if the View is a statistical test of differences in means.

is_net()
Tests if the View is a code group/net aggregation.

is_pct()
Tests if the View is a percentage representation of a frequency.

is_propstest()
Tests if the View is a statistical test of differences in proportions.

is_stat()
Tests if the View is a sample statistic.

is_sum()
Tests if the View is a plain sum aggregation.

is_weighted()
Tests if the View is performed on weighted data.

meta()
Get a summary on a View's meta information.

Returns **viewmeta** – A dictionary that contains global aggregation information.

Return type dict

missing()
Returns any excluded value codes.

nests()
Slice a nested `View.dataframe` into its innermost column sections.

notation(*method*, *condition*)
Generate the View's Stack key notation string.

Parameters **shortname, relation** (*aggname,*) – Strings for the aggregation name, the method's shortname and the relation component of the View notation.

Returns **notation** – The View notation.

Return type str

rescaling()
Returns the rescaling specification of value codes.

spec_condition(*link*, *conditionals=None*, *expand=None*)
Updates the View notation's condition component based on agg. details.

Parameters **link** (*Link*) –

Returns **relation_string** – The relation part of the View name notation.

Return type str

weights()
Returns the weight variable name used in the aggregation.

9.9 ViewMapper

class quantipy.**ViewMapper**(*views=None*, *template=None*)

Applies View computation results to Links based on the view method's kwargs, handling the coordination and structuring side of the aggregation process.

add_method(*name=None*, *method=None*, *kwargs={}*, *template=None*)

Add a method to the instance of the ViewMapper.

Parameters

- **name** (*str*) – The short name of the View.
- **method** (*view method*) – The view method that will be used to derivce the result
- **kwargs** (*dict*) – The keyword arguments needed by the view method.
- **template** (*dict*) – A ViewMapper template that contains information on view method and kwargs values to iterate over.

Returns Updates the ViewMapper instance with a new method definiton.

Return type None

make_template(*method*, *iterators=None*)

Generate a view method template that cycles through kwargs values.

Parameters

- **method** (*{'frequency', 'descriptives', 'coltests'}*) – The baseline view method to be used.
- **iterators** (*dict*) – A dictionary mapping of view method kwargs to lists of values.

Returns Sets the template inside ViewMapper instance.

Return type None

subset(*views*, *strict_selection=True*)

Copy ViewMapper instance retaining only the View names provided.

Parameters

- **views** (*list of str*) – The selection of View names to keep.
- **strict_selection** (*bool, default True*) – TODO

Returns subset

Return type ViewMapper instance

CHAPTER 10

Quantipy: Python survey data toolkit

Quantipy is an open-source data processing, analysis and reporting software project that builds on the excellent pandas and numpy libraries. Aimed at social and marketing research survey data, Quantipy offers support for native handling of special data types like multiple choice variables, statistical analysis using case or observation weights, dataset metadata and customizable reporting exports.

Note: We are currently moving our documentation and reorganizing it. Sorry for the lack of latest information.

10.1 Key features

- Reads plain .csv, converts from Dimensions, SPSS, Decipher, or Ascribe
- Open metadata format to describe and manage datasets
- Powerful, metadata-driven cleaning, editing, recoding and transformation of datasets
- Computation and assessment of data weights
- Easy-to-use analysis interface
- Automated data aggregation using `Batch` defintions
- Structured analysis and reporting via Chain and Cluster containers
- Export to SPSS, Dimensions ddf/mdd, table spreadsheets and chart decks

Contributors Kerstin Müller, Alexander Buchhammer, Alasdair Eaglestone, James Griffiths

Birgir Hrafn Sigurðsson and Geir Freysson of datasmoothie

- genindex
- modindex

Bibliography

[DeSt40] Deming, W. Edwards; Stephan, Frederick F. (1940): On a Least Squares Adjustment of a Sampled Frequency Table When the Expected Marginal Totals are Known. In: Ann. Math. Statist. 11 , no. 4, pp. 427 - 444.

Index

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W