PanthRBase Documentation

PanthR Team

December 16, 2014

Contents

1 Variable

Variable(values, options)

Create a new variable. `values` is one of the following:

- an array or vector with the desired values,
- a variable (which is simply cloned)
- a function `f(i)` for generating the values, in which case `length` is a required option. `options` is an object indicating properties of the variable:
  - `length`: Will be ignored if `values` is not a function
  - `label`: The label to use in graphs/tables/descriptions.
  - `mode`: A string describing what type of variable to create. If `mode` is missing it will be determined based on the first non-missing entry in `values`.
  - `_names`: An optional vector/array/variable of equal length containing names for the values. Access it via the `names` method.

Further options depend on the particular mode chosen. See the subclass documentations for details.

A default label value will be generated if not provided. So creating a `Variable` can be as simple as passing a `values` argument to the constructor.

Variable construction and setting needs to preserve the invariant that all entries are either `null` or a "meaningful" value. All `undefined`, missing and `NaN` entries will be converted to `null`.

If `values` is a `Variable` it will simply be cloned (`options` will be ignored).

Variable.concat(vars)

Concatenate the inputs into a single variable. All inputs must be variables, and a common mode will be inferred based on the variable modes:

- Variables of mode ordinal are treated as having mode factor
- If one of the variables is of mode string, the result is of mode string
- If all variables have the same mode, the result is of that mode
- Otherwise the result is of mode scalar

Variable.dateTime(values, label)

Create a date-time variable. `label` is optional.

Variable.ensureArray(val)

Convert `val` into a (Javascript) array, with "missing values" replaced by `utils.missing`. The argument may be:

- A single number
- A value that is `utils.isMissing` (`NaN`, `null`, `undefined`)
- An array, `Vector` or `Variable`

Variable.factor(values, label)

Create a factor variable. `label` is optional.

Variable.fiveNum(skipMissing)

Return a 'named' scalar variable of the five-number of the values of the variable. `skipMissing` defaults to `false`. If `skipMissing` is `false` and the variable has missing values, return `utils.missing`.

Variable.format(options)

Return a string variable for displaying the given variable with a specified numerical formatting. The `options` object may include:

- `type`: A string value, either 'scientific' or 'fixed'.
- `decimals`: The number of decimal digits to be displayed to the right of the decimal point. Defaults to 4 for 'scientific' format and to 2 for 'fixed' format.

Variable.groupIndices()

Return a `List` of arrays of indices corresponding to the distribution of the factor variable. If the variable is not factor or ordinal, it will be treated as a factor variable.

If missing values are present, an extra (unnamed) list item to hold those indices will be created at the end of the list.

```
\texttt{Variable.factor(['a','a','b']).groupIndices(); // { a: [1, 2], b: [3] }}
```

Variable.layOut(ncol)

Lay out the variable's values in rows. Return an array with one entry for each row. Each row entry is an array of objects representing the values. The objects have the form `index: i, value: val, name: s` .

The parameter `ncol` specifies how many entries will be in each row (default is 1). If the variable length is not an exact multiple of `ncol`, then the last row will contain fewer entries.

Mostly intended as an internal method for use in `Variable#toHTML`.

Variable.logical(values, label)

Create a logical variable. `label` is optional.

Variable.max(skipMissing)

Return the maximum of the values of the variable. `skipMissing` defaults to `false`. If `skipMissing` is `false` and the variable has missing values, return `utils.missing`.

Variable.mean(skipMissing)

Return the mean of the values of the variable. `skipMissing` defaults to `false`. If `skipMissing` is `false` and the variable has missing values, return `utils.missing`.

Variable.median(skipMissing)

Return the median of the values of the variable. `skipMissing` defaults to `false`. If `skipMissing` is `false` and the variable has missing values, return `utils.missing`.

Variable.min(skipMissing)

Return the minimum of the values of the variable. `skipMissing` defaults to `false`. If `skipMissing` is `false` and the variable has missing values, return `utils.missing`.

Variable.oneDimToArray(val)

Convert `val` into an array, if `val` is "one-dimensional" (`Variable`, `Vector`, array). Non-one-dimensional arguments are returned unchanged.

Variable.oneDimToVariable(val)

Convert `val` into a `Variable`, if `val` is "one-dimensional" (`Variable`, `Vector`, array). Non-one-dimensional arguments are returned unchanged.

Variable.oneDimToVector(val)

Convert `val` into a `Vector`, if `val` is "one-dimensional" (`Variable`, `Vector`, array). Non-one-dimensional arguments are returned unchanged.

Variable.order(desc)

Return a `Variable` representing the permutation that sorts the values of the original variable according to the order specified by `desc`.

- If `desc` is a boolean value, then `false` indicates ascending order, `true` indicates descending order.

- If `desc` is a function `f(a, b)`, then it is interpreted as the comparator for sorting, and must return `-1` if `a` precedes `b`, `0` if `a` and `b` are "equal" in order, and `1` if `b` precedes `a`.
- If `desc` is omitted, it defaults to `false` (ascending order).

Variable.ordinal(values, levels, label)

Create an ordinal variable. `levels` and `label` are optional. If `levels` is omitted, an alphabetical ordering of the levels will be used.

Variable.read(vals, mode)

Read values from a string (e.g., text file) into a `Variable`.

`Variable#read` makes a sequence of tokens by breaking the string at any sequence of newlines, spaces, commas and semicolons.

- If a token starts with a double quote, then it must also end with a double-quote, and its contents are interpreted as follows:
  - Consecutive double-quotes (`&quot;&quot;`) are interpreted as a double-quote (`&quot;`).
  - Escaped (backslashed) characters (`\c`) are interpreted as the character (`c`).
  - No unescaped un-doubled double-quotes are allowed in the term.
- Analogous conditions apply for a term starting with a single quote (`'`).
- If the token does not start with a quote, then it is interpreted literally.

If the mode is not specified, it will be inferred as `scalar` if all the tokens can be interpreted as numbers, or as `factor` otherwise.

Variable.scalar(values, label)

Create a scalar variable. `label` is optional.

Variable.scale(center, scale)

Rescale the variable based on the provided `center` and `scale`. Return a `List` holding three items:

- `center`
- `scale`
- `values` (a `Variable` holding the rescaled values).

Must be called with two arguments.

Variable.sd(skipMissing)

Return the standard deviation of the values of the variable. `skipMissing` defaults to `false`. If `skipMissing` is `false` and the variable has missing values, return `utils.missing`.

Variable.seq(from, to, step, options)

Construct a scalar variable from an arithmetic sequence. Can be called as:

- `seq(to[, options])` where `from` equals 1
- `seq(from, to[, options])` where `step` equals -1 or +1
- `seq(from, to, step[, options])` `step` must have the same sign as `to - from`. `options` parameter is an optional options object that is passed to the `Variable` constructor

  seq(5) // [1, 2, 3, 4, 5] seq(5, 7.5) // [5, 6, 7] seq(4, 1.2) // [4, 3, 2] seq(5.1, 6.1, .5) // [5.1, 5.6, 6.1] seq(4, 1.2, -2) // [4, 2]

Variable.sort(desc)

Return a new `Variable` with the values sorted in the order specified by `desc`. See `Variable#order`.

Variable.string(values, label)

Create a string variable. `label` is optional.

Variable.sum(skipMissing)

Return the sum of the values of the variable. `skipMissing` defaults to `false`. If `skipMissing` is `false` and the variable has missing values, return `utils.missing`.

Variable.table()

Return a frequency table for the variable, in the form of a 'named' scalar variable. The variable is treated as a factor variable in order to accumulate the frequencies.

Variable.tabulate(f, from, to, options)

Create a new variable with values `f(from), f(from+1), ..., f(to)`. The `options` parameter is passed to the `Variable` constructor.

Variable.toHTML(options)

Return an HTML string displaying the variable. Each value is wrapped in a tag, and that tag may be preceded by a tagged name. Each row is further wrapped in a tag. The default format can be used as the contents of a `table` tag.

The `options` object may include:

- `ncol`: The number of "columns" (default is 1).
- `withNames`: Whether names should be included (default is `false`).
- `value`: An object with properties `tag` and `class` for specifying the html tag and the class attributes to be used for the values.
- `name`: A similar object to be used for the names.
- `row`: A similar object to be used for wrapping around each row. All three of these objects are optional, and their individual parts are optional as well. The defaults are `&lt;td&gt;` or `&lt;tr&gt;` for the tag and `var-value`, `var-name` or `var-row` for the class.

Variable.var(skipMissing)

Return the variance of the values of the variable. `skipMissing` defaults to `false`. If `skipMissing` is `false` and the variable has missing values, return `utils.missing`.

Variable.write(options)

Write the variable to a string.

`options` is an object that can include:

- `sep`: A character or string to use as separator. Defaults to `','`.
- `quote`: A boolean value specifying whether to quote string values/names. Defaults to `false`.
- `qescape`: A boolean value specifying whether to escape embedded quotes via a backslash. Defaults to `false`, meaning escape via an extra double-quote.

Variable.zscore()

Return the standardized values using `Variable#rescale` where `center` is the mean of the variable and `scale` is the standard deviation.

Missing values are preserved, but are ignored in the computation.

Variable.prototype.asScalar()

Convert the variable to scalar mode.

For factor variables, the codes are used.

Variable.prototype.asString()

Convert the variable to string mode.

For factor variables, the values are used.

Variable.prototype.clone()

Clone the variable, creating a new variable with the same values and mode.

Variable.prototype.concat(vars)

See `Variable.concat`.

Variable.prototype.each(f, skipMissing)

Apply the function `f(val, i)` to each value in the variable. If `skipMissing` is set to `true` (default is `false`), it will only apply `f` to non-missing values (as determined by `utils.isNotMissing`).

Variable.prototype.filter(pred)

Given a predicate `pred(val, i)`, return a new variable containing those values from the original variable that satisfy the predicate.

Variable.prototype.get(i)

Return the values(s) indicated by `i`. (Keep in mind that variables are indexed starting from 1.)

- If `i` is a positive integer, return the value at index `i`.
- If `i` is an array of non-negative integers, return an array of the corresponding values (skipping indices of value 0).
- If `i` is an array of non-positive integers, return an array of all values of the variable except those indicated by the negative indices.
- If `i` is a scalar variable, it is converted into an array.
- If `i` is a logical variable, it must have the same length as the original variable, in which case, return an array of the values which correspond to the `true` values in `i`.

For factor variables, the values are returned, not the codes.

Variable.prototype.hasMissing()

Return a boolean indicating whether the variable contains missing values as indicated by `utils.isMissing`.

Variable.prototype.length()

Return the length of the variable.

Variable.prototype.map(f, skipMissing, mode)

Create a new variable from the results of applying the function `f(val, i)` to the values of the original variable. If `skipMissing` is set to `true` (default is `false`), then missing values will be preserved, and `f` will only be applied to the non-missing values. The optional parameter `mode` specifies the desired mode of the new variable.

Variable.prototype.names(newNames)

Called with no arguments, return the names associated with the variable's entries.

Otherwise `newNames` is passed to the `Variable` constructor to create a string variable of the new names. If the provided names do not have the correct length, `Variable#resize` will be used on the names.

Variable.prototype.nonMissing()

Return a new variable containing the non-missing values from the original variable as indicated by `utils.isNotMissing`.

Variable.prototype.reduce(f, initial, skipMissing)

Apply the function `f(acc, val, i)` to each value in the variable, accumulating the result to be returned. If `skipMissing` is set to `true` (default is `false`), it will only apply `f` to non-missing values (as determined by `utils.isNotMissing`).

Similar to Javascript's `Array.prototype.reduce`.

Variable.prototype.rep(times)

Repeat a variable according to a pattern to make a new variable. `times` can be used in several different ways, depending on its type:

- If `times` is a number, repeat the variable that many times.
- If `times` is a variable or array, use the values as frequencies for corresponding entries. `times` must have same length as the original variable.
- If `times` is an object with a `length` property, cycle the values in the variable up to the specified length.
- If `times` is an object with an `each` property, repeat each value that many times (before going on to the next value).

Variable.prototype.reproduce(newValues, newNames)

Return a new variable with all the same settings as the original but with values taken from `newValues`, which may be a `Vector` or an array.

Note: If the variable is a factor or an ordinal variable, it is assumed that the new values are codes which are in agreement with the codes used by the variable.

If `newNames` is provided, it must be one-dimensional (`Variable`, `Vector` or array) and it is used to set names for the new variable.

Variable.prototype.resize(length, fill)

Resize the variable. If fill is `true`, recycle the values to reach the specified length. If fill is `false` or omitted, the new values will be filled with `utils.missing`.

Variable.prototype.sameLength(other)

Return a boolean indicating whether the variable has the same length as the variable `other`.

Variable.prototype.select(indices)

From a given array or `Vector` of indices, create a new variable based on the values of the original variable corresponding to those indices.

Variable.prototype.set(i, val)

Set the entries indicated by `i` to the values indicated by `val`. (Keep in mind that Variables are indexed starting from 1.)

`val` may be a single value, or a `Variable` or array of values of the appropriate length.

- If `i` is a positive integer, set the value at index `i`.
- If `i` is an array of non-negative integers, set the corresponding values (skipping indices of value 0).
- If `i` is an array of non-positive integers, set all values of the variable except those indicated by the negative indices.
- If `i` is a scalar variable, it is converted into an array.
- If `i` is a logical variable, it must have the same length as the original variable, in which case set the values which correspond to the `true` values in `i`.

In all cases, if there are any null/undefined/NaN indices, an error occurs.

This method cannot be used to append values. To set values out of bounds, call `Variable#resize` first.

Variable.prototype.toArray()

Return a Javascript array of the values of the variable.
For factor variables, the values are returned.

Variable.prototype.toVector()

Return a `Vector` of the values of the variable.
For factor variables, the codes are returned.

2 Dataset

Dataset(values)

Create a dataset out of the provided `values`. A dataset is a `List` whose items are variables of the same length. Unlike lists, datasets are required to have names for all their "columns", and those names are unique.

`values` is one more more arguments of the following types:

- An object, a `List`, or `Matrix`; in this case it will be 'unpacked' to create the columns of the dataset.
- A `Variable` or `Vector`.

Properties:

- `nrow`: The number of rows in the dataset (the length of each variable)
- `ncol`: The number of columns in the dataset (the number of variables)

Dataset.read(vals, options)

Read a dataset from a string `vals` which is the contents of a delimited file.
Quote-escaping rules are similar to `Variable#read`.
`options` is an object that can include:

- `sep`: A character or string specifying the separator. If not provided, an attempt to infer the separator will be made. Typical separators include `','`, `';'`, `'^'`, and `' '`. In this last case, any sequence of whitespace, including tabs, will be treated as a single separator.
- `header`: A boolean value specifying whether headers are included. Defaults to `false`.

Dataset.split(select)

Split a `Dataset` into a `List` of sub-datasets, based on the specified subsets of the rows. `select` can be:

- A `List` whose elements are one-dimensional collections of row indices
- A factor `Variable` of length `nrow`. Rows with the same corresponding factor value will be grouped together.
- A function `f(row, i)`. Rows with the same function value will be grouped together.

If an empty group of rows is created by `select`, it will generate an empty `Dataset`.

Dataset.write(options)

Write the dataset to a string.

`options` is an object that can include:

- `sep`: A character or string to use as separator. Defaults to `','`.
- `header`: A boolean value specifying whether to include headers. Defaults to `true`.
- `quote`: A boolean value specifying whether to quote string values/names. Defaults to `false`.
- `qescape`: A boolean value specifying whether to escape embedded quotes via a backslash. Defaults to `false`, meaning escape via an extra double-quote.

Dataset.prototype.appendCols(names, values)

Append to the columns of the dataset. If called with two arguments, then the first argument is the names for the new columns. If called with only one argument, names will be generated automatically.

The `values` argument needs to be one of the following:

- A 2-dimensional object (`Matrix` or `Dataset`).
- A 1-dimensional object (`Array`, `Vector` or `Variable`).
- A `List` of columns to be appended. Corresponding names will be copied over. In this case, the provided list will be fed into the dataset constructor in order to deduce the new variables to be appended.
- A function `f(i)` for computing the values in the new column.

Dataset.prototype.appendRows(rows, values)

Append to the rows of the dataset. When called with one argument, the argument needs to be 2-dimensional (`Matrix` or dataset) or 1-dimensional (`Array`, `Variable` or `Vector`) and then the number rows to be appended will be inferred. When called with two arguments, `rows` is the number of rows to append, and `values` is a single value or a function `f(i, j, colName)` to be used for filling the rows. In the case of a function, the index `i` is relative to the new rows to be added (so `i` is 1 for the first row to be added, 2 for the second row to be added, etc.).

```
\texttt{// dSet assumed to be a 2x3 dataset
dSet.appendRows([1, 2, 3]) // Add a single row at row index 3
dSet.appendRows(dSet)      // Add duplicates of the 3 rows
dSet.appendRows(2, function(i, j) { return i + j }); // Adds rows [2,3,4], [3,4,5]}
```

Dataset.prototype.clone()

Clone the dataset.

Dataset.prototype.deleteCols(cols)

Delete the specified columns from the dataset. `cols` may be:

- A single number or string name.
- A 1-dimensional object of single numbers or string names.

Dataset.prototype.deleteRows(rows)

Delete the specified rows from the dataset. `rows` may be:

- A single number.
- A 1-dimensional object.
- A predicate function `f(row, i)`.

Dataset.prototype.get(rows, cols)

Return a subset of the values in the dataset. This method may be called with no arguments, in which case an array of arrays of the columns is returned. Otherwise, the method requires two arguments, `rows` and `cols`, specifying respectively the rows and columns to be used.

- `cols` can be:
  - A single number or string. In this case a single column is used.
  - The boolean `true`, indicating that all columns should be used.
  - A one-dimensional object (`Array`, `Variable`, `Vector`) of numbers, strings or booleans. In the case where the values are booleans, the length of the object must match `ncol`.
  - A predicate of the form `pred(colName, j)`, which returns true for those columns that are to be used.
- `rows` can be:
  - A single number. In this case a single row is used.
  - The boolean `true`, indicating all rows should be used.
  - An `Array`, `Variable` or `Vector` of numbers or booleans (similar to `cols`)
  - A predicate that has form `pred(row, i)`, where `row` is a function as returned by `Dataset#rowFun`, giving access to the `i`-th row. If given two single values, returns the corresponding single value at the i-th row/j-th column. Otherwise returns a dataset that contains copies of the appropriate entries.

Dataset.prototype.getVar(col)

Get a single column (variable). `col` is a positive number or string name.

Dataset.prototype.names(i, newNames)

Get or set the names of the dataset's columns. See `List#names` for details. This method enforces uniqueness of names.

Dataset.prototype.rowFun(i)

Given a row index `i`, return a function `f(col)` which "simulates" row `i`.

```
\texttt{l.rowFun(2)('a') // Returns the second value in column 'a'.
l.rowFun(2)(2)   // Returns the second value in the second column.}
```

Dataset.prototype.set(rows, cols, vals)

Set the values at specified rows and columns, using the values specified by `vals`. See `Dataset#get` for how to use `rows` and `cols` to specify the positions to be set. All 3 arguments are required. `vals` is used to specify new values in one of the following ways:

- A single value (to be used in all specified positions)
- A `Variable`, `Vector` or `Array` (only valid when setting within a single row or column)
- A `Dataset` or `Matrix` (whose dims match those of the selected region)
- A function `f(i, j, name)` where `i` is a row number, `j` is a column number, and `name` is a column name.

Dataset.prototype.setVar(col, val)

Replace the variable at column `col` with the variable `val`. The length of `val` must match `nrow`.

Dataset.prototype.toArray()

Return an array of arrays representing the columns of the dataset.

Dataset.prototype.which(pred)

Given a predicate `pred(row, i)`, return a `Variable` of the row numbers of the rows for which the predicate is `true`.

3 List

List(values)

A list is a collection of Javascript entities that can be accessed by index or by name. One can remove an item from a list, which results in the other items shifting place. One can also insert at the end of a list, or alter the contents of the list.

Indexing starts at 1.

We can create a list by providing:

- an object containing the items to be placed in the list along with their names
- an array of items to be placed in the list, without names
- no arguments, resulting in an empty list

List.prototype._set(i, val)

Internal method used to set one item value. Requires two arguments, an integer or string name `i` and a value `val`.

List.prototype.clone()

Clone the list. This method will attempt to make a deep clone by calling `clone` on any top-level items in the list that have a clone method.

List.prototype.delete(i)

Delete the item at index/name `i`. `i` may be a positive integer or string name.

List.prototype.each(f)

Apply the function `f(val, i, name)` to each item in the list. For any items with no associated name, `name` will be `utils.missing`.

List.prototype.get(i)

Return a list item by index. The index `i` can be:

- a positive number
- a string name
- `null` or `utils.missing`; in this case, an array of all the items is returned.

List.prototype.getIndexOf(name)

Given a name, return the index of the item with that name or utils.missing if there isn't one. You may also instead pass a single number, or an array of names and numbers, in which case an array of indices is returned. Mostly meant as an internal method.

List.prototype.length()

Return the length of the list (number of items)

List.prototype.map(f)

Create a new list from the results of applying the function `f(val, i, name)` to the items of the original list. For any values with no associated name, `name` will be supplied as `utils.missing`.

List.prototype.names(i, newNames)

Get or set the item names.

- When called with no arguments, return a string `Variable` of all the names, with `utils.missing` in place of any missing names. If no names exist, returns `utils.missing`.
- When called with a single numeric argument `i`, return the name at the given index.
- When called with a single array or `Variable` argument, set the names of the list using the array/variable's elements.
- When called with a single `null` or `utils.missing` argument, set the names to `utils.missing`.
- When called with two arguments `i`, `newName`, set the name of the `i`-th item to `newName`.

  var l = new List(a: [1, 2], b: 3); l.names(); // Variable(['a', 'b']) l.names(2); // 'b' l.names(2, 'c'); // `l` is now a: [1,2], c: 3 l.names(['d', 'e']); // `l` is now d: [1,2], e: 3 l.names(null); // `l` now has no names

List.prototype.reduce(f, initial)

Apply the function `f(acc, val, i, name)` to each item in the list, accumulating the result to be returned. For any values with no associated name, `name` will be supplied as `utils.missing`.
Similar to Javascript's `Array.prototype.reduce`.

List.prototype.set(i, val)

Called with two arguments `i`, `val`. Set the list item at a given index `i`. `i` can be:

- a positive number. If `i` is greater than the length of the list, create a new item at index `i` and fill the resulting gap with `utils.missing`.
- a string name: If the name `i` is not already a name in the list, append a new item with name `i`. Otherwise, update the existing item with the new value.
- an object of name-value pairs, causing a series of updates or appends, one for each pair.
- an array of values, causing a series of appends of these (unnamed) items.

`val` can be any Javascript entity. If `i` is an object or array, then `val` is omitted.

List.prototype.toVariable()

Return a `Variable` by concatenating the values from the list's items. Works with items of any of the following types:

- single value
- Array, `Vector`, `Variable`
- `List`

Names are generated for the new Variable base on the items' names in the list as well as their names (if any) as PanthR objects. The idea is to preserve any names in the list and/or in the values of the list in some reasonable way, wherever they exist.

- If the item is a variable with names, and it is a named list item, join the names
- If the item is a variable with names, and it is an unnamed list item, use the variable names
- If the item is a variable without names, and it is a named list item, provide names of the form `itemName.1, itemName.2, ...`

List.prototype.unnest(levels)

Unnest a number of levels out of a nested list, starting at the top level. `levels` is the number of levels it will attempt to unnest. Level 0 indicates no change. Default is 1. Level `Infinity` indicates complete unnesting.

This method will only attempt to resolve nesting formed via `List` constructs, and will not recurse into Javascript Objects or Arrays.

BEWARE: This operation changes the list(s) *in place*.

4 stats

stats.correlate(xs, ys, skipMissing)

Return the Pearson correlation coefficient between two variables, `xs` and `ys`. By default, uses all the values of both variables. If `skipMissing` is not set to `true` and missing values exist, return `utils.missing`.

The two variables must have the same length.

5 utils

utils.allMissing(arr)

Return true if all entries in the array are missing.

utils.areEqualArrays(A, B)

Test for array element equality that respects missing values. Makes a shallow comparison.

utils.equal(a, b)

Test for equality that respects missing values.

utils.format

An object containing formatting functions for numbers.

utils.getDefault(val, deflt)

If `val` is a missing value, return `deflt`, else return `val`.

utils.getOption(s, optList, deflt)

Take a user-provided option description `s` (a string) and an array `optList` of allowable option settings. Return the first element of the array that has `s` as its initial substring. Return `null` if no such match is found.

If `s` is empty, `null` or `undefined`, return the default setting `deflt`.

utils.hasMissing(arr)

For an array, return whether the array has any missing values in it.

utils.isMissing(val)

Return true if `val` is `undefined`, `null`, or `NaN`. *

utils.isNotMissing(val)

Return true if `val` is not `undefined`, `null`, or `NaN`. *

utils.isOfType(v, types)

Test if `v` is of one of the listed `types` (an array of strings).

utils.makePreserveMissing(f)

Return a new function `g` such that `g(any missing)` is `utils.missing`, and `g(val)` is either `f(val)` or `utils.missing`, depending on whether `f(val)` is a missing value.

utils.missing

Value to be used for all missing values.

utils.mixin(target)

Mixes into the first object the key-value pairs from the other objects. Shallow copy.

utils.op

Arithmetic operators

utils.op.add(a, b)

The function that adds two numbers. Also available as `utils.op['+']`.

utils.op.div(a, b)

The function that divides two numbers. Also available as `utils.op['/']`.

utils.op.max2(a, b)

The function that takes two values and returns the maximum.

utils.op.min2(a, b)

The function that takes two values and returns the minimum.

utils.op.mult(a, b)

The function that multiplies two numbers. Also available as `utils.op['*']`.

utils.op.sub(a, b)

The function that subtracts two numbers. Also available as `utils.op['-']`.

utils.optionMap(val, f)

If `val` is a missing value, return `utils.missing`. Otherwise return `f(val)`.

utils.seq(from, to, step)

Create an array of sequential values. Similar options to `Variable.seq`.

utils.singleMissing(val)

Return `val` if it is non-missing; otherwise return `utils.missing`.