1

Introduction

Introduction

The Kx System consists of:

- kdb+ - the database
- q - an analytic and query language for kdb+
- k - the programming language underlying both kdb+ and q

This reference documents q - the main language for working with kdb+.

History

The kdb+ database and q language were introduced in 2003 as part of a 64-bit rewrite of the earlier kdb database. The underlying k language and databases have been developed since 1993.

Resources

- there is an active Kx wiki at https://code.kx.com. Anyone can login as user/password: anonymous
- The book *Q for Mortals* is a good introductory guide to the Q language. It is available on the Kx wiki, and from Amazon.
- the main discussion forum is the *k4 listbox*, and is open to users of the commercial product.

- another discussion forum is the *Kdb+ Personal Developers* forum on google. This is open to everyone.

Source

The source for this documentation is at https://code.kx.com/trac/wiki/Reference.

Grammar

Q has four object types: noun, function, verb and adverb:

```
q)sales:2 7 1 8                 / noun sales

q)sum sales                     / function sum
18

q)sales * 10                    / verb *
20 70 10 80

q)sales */: 2 10                / adverb /:
4  14 2  16
20 70 10 80
```

Verbs differ from functions in that they may be called (and typically are called) with arguments to left and right. A verb may also be called with both arguments on the right, as in:

```
q)*[sales;10]
20 70 10 80
```

Remark: in some kdb+/q texts, the terms verb and function are used interchangeably, with qualifiers to indicate whether arguments are on the right, or to the left and right. Instead, this reference distinguishes verb from function.

Remark: q has no ambivalent functions, unlike the underlying k language.

Syntax

Q has:

- three object syntaxes: *noun*, *verb* and *adverb*. Note that noun syntax is used by both nouns and functions.
- a *template* syntax for SQL-like queries, here known as q-SQL
- other syntaxes for assignment, explicit definition, table definition, punctuation and scripts

Object Syntax

Example:

```
q)sales:2 7 1 8          / assign list of numbers

q)sales 1 3              / noun syntax - argument to the right
7 8

q)sum sales              / a function has noun syntax
18

q)sales * 10             / verb syntax, arguments to left and right
20 70 10 80

q)sales */: 2 10         / adverb syntax, argument to the left
4  14 2  16
20 70 10 80
```

Query Template

Example:

```
q)select from p where weight=17
p | name  color weight city
--| ------------------------
p2| bolt  green 17     paris
p3| screw blue  17     rome
```

Other Syntax

```
a:5                              / local assignment
a::5                             / global assignment
f:{y+2*x}                        / {} explicit definition
([]sym:`amd`intel;vol:300 500)   / table definition
(sales-2) * 10                   / () grouping
/                                / an isolated / is a comment
\                                / an isolated \ is exit
\\                               / exit q
```

See also

* add references here...

Nouns

Nouns can be classified in two ways:

- by *structure*, e.g. atom, list, dictionary
- by *datatype*, e.g. integer, character, date, list

Note that *list* is both a structure (a list of things), and a datatype. Each item of a list will have its own datatype.

Data Structures

Q data structures are: atoms, lists, associations, dictionaries, tables and keyed tables.

The core building blocks are atoms and lists:

- an *atom* is a single data item, such as a single integer, date or symbol
- a *list* is an ordered collection of atoms or lists

Two lists can be joined in an association of keys and values. All other structures are built from lists and associations:

- an *association* relates *keys* to *values*
- a *dictionary* is an association where the keys are symbols
- a *table* is a flipped (transposed) dictionary, where values all have the same length
- a *keyed table* is an association of two tables (the key columns, and the non-key columns)

See Also:

- page 7 - Atoms and Lists
- page 8 - Associations and Dictionaries
- page 10 - Tables

Data Types

See:

- page 13 - Data Types

Atoms and Lists

An *atom* is a single data item, such as a single number, date or symbol. A *list* is an ordered collection of atoms or lists. An atom is not the same as a 1-element list.

The function `enlist` converts its argument into a list:

```
q)a:10                  / a is the atom 10
q)b:enlist 10           / b is the 1-element list 10
q)a~b                   / a and b do not match
0b
q)b                     / b is displayed with a comma prefix
,10
```

A list can have all items of the same type (a simple list), or have mixed types. In a list of lists, the individual sub lists can be all the same length, or of varying lengths. Q will handle any type of list, but is most efficient on lists with all items of the same type. A column in a database is stored as a list, and typically this will be a list with all items of the same type.

The general form of a list of length n is $(p_1;p_1;...;p_n)$ where the p_i are atoms or lists. However, where the data items are all atoms of the same type, there are simpler forms of entry and display. For example:

```
q)m:(2;3;5;7;11)
q)n:2 3 5 7 11
q)m~n                    / m is the same as n
1b
q)m                      / m displays in simple form
2 3 5 7 11
```

Note that a list is a single thing, whose contents may be several things. For example, in the expression below, the function `sum` is applied to a single argument which is a list of 5 numbers. It is incorrect to think of `sum` as being applied to 5 arguments:

```
q)sum 2 3 5 7 11
28
```

Associations and Dictionaries

Associations

An association is a pair of lists in 1-1 relationship, that relates a set of keys to a set of values. It can be created with the verb ! as in `keys!values`. Any data types can be used in an association.

Dictionaries

A dictionary is an association where the keys are a list of symbols. Typically, but not necessarily:

- there would be no duplicates in the keys (true for a keyed database)
- the values are simple lists (most efficient)

For example:

```
q)items:`nut`bolt`cam`cog
q)sales:6 8 0 3
q)prices:10 20 15 20

q)dkeys:items`sales`prices          / list of keys
q)dvals:(items;sales;prices)        / list of values
q)dict:dkeys!dvals                  / dictionary
q)dict
items | nut bolt cam cog
sales | 6   8    0   3
prices| 10  20   15  20
```

A dictionary is indexed on its key, and not on its position:

```
q)dict sales
6 8 0 3
q)dict 1
`type
```

A dictionary can be index assigned. If the index is not already a key, the assignment creates a new key:

```
q)dict[`sales]:10 14 1 7            / assign sales
q)dict[`sizes]:30 20 0 15           / assign new sizes
q)dict
items | nut bolt cam cog
sales | 10 14 1 7
prices| 10 20 15 20
sizes | 30 20 0 15
```

Two dictionaries can be joined together. Items in the second dictionary either overwrite the first dictionary, or are appended to the first dictionary. Thus, the assignments above could be done with a single join:

```
q)dict                              / original dictionary
items | nut bolt cam cog
sales | 6 8 0 3
prices| 10 20 15 20

q)dict2:`sales`sizes!(10 14 1 7;30 20 0 15)  / required changes
q)dict2
sales| 10 14 1 7
sizes| 30 20 0 15

q)dict,dict2                        / join
items | nut bolt cam cog
sales | 10 14 1 7
prices| 10 20 15 20
sizes | 30 20 0 15
```

Arithmetic and other operations can be performed on dictionaries. The operations are performed on the values, if necessary matching on keys. For example, to multiply all sales by a factor of 10:

```
q)dicts:(enlist sales)!enlist 10    / need to enlist the atoms
q)dicts
sales| 10
q)dict * dicts
items | nut bolt cam cog
sales | 60 80 0 30
prices| 10 20 15 20
```

To delete an entry, use verb _ (delete):

```
q)dict _ `items
sales | 10 14 1  7
prices| 10 20 15 20
```

Tables and Keyed Tables

A *table* is a dictionary that has been flipped (transposed), and where the values of the dictionary must have the same length.

A *keyed table* is a dictionary of a pair of tables: the key table with the key columns, and the value table with the remaining columns.

Note that table and keyed table correspond to the usual meaning of table in SQL, where a keyed table in q corresponds to a SQL table with one or more primary key columns.

Note also that tables can be built directly from the underlying lists, for example:

```
q)dict                                  / dict
items | nut bolt cam cog
sales | 6   8    0   3
prices| 10  20   15  20

q)flip dict                             / table
items sales prices
------------------
nut   6     10
bolt  8     20
cam   0     15
cog   3     20

q)ids                                   / another dict
id| 2 3 5 7

q)(flip ids)!flip dict                  / keyed table
id| items sales prices
--| ------------------
2 | nut   6     10
3 | bolt  8     20
5 | cam   0     15
7 | cog   3     20
```

Creating Tables

Tables can also be defined by specifying the columns in the form:

$([c_1:v_1;...;c_j:v_j]c_{j+1}:v_{j+1};...;c_n:v_n)$

Here c_i are symbols with the column name, and v_i the corresponding list of values. The square brackets are used to specify primary keys, and must be given even if there are no primary keys - the example shows a table with j primary keys, and n columns in total.

For example:

```
q)t:([name:nms]dob:dob;sal:sal)
q)t
name | dob        sal
-----| ----------------
anne | 1979.06.21 44350
mary | 1947.11.01 63375
david| 1961.05.14 78900
beth | 1976.01.18 36750
```

Empty Tables

A table can be defined with empty data. The data type of the column is set either when the table is first written, or as null values of a given type. For example:

```
q)w:([]name:();dob:();sal:())    / empty columns with no types
q)w
name dob sal
------------
q)insert[w;(anne;1979.06.21;44350)]
,0
q)w
name dob        sal
--------------------
anne 1979.06.21 44350
```

Alternatively, initialize w with empties of given type:

```
q)w:([]name:symbol$();dob:date$();sal:int$())
```

Table Information

The `cols` function returns the column names of a table:

```
q)cols t
`name`dob`sal
```

The `meta` function returns information about a table:

```
q)meta t
c   | t f a
----| -----
name| s
dob | d
sal | i
```

Here, the t column is type, f is foreign key, and a is attributes. Both f and a are empty for this table.

Data Types

Note: for the nonce, this is just the brief reference datatype list...

char	-size	-type	-literal	-q	-sql	-java	.net
b	1	1	0b	boolean		Boolean	boolean
x	1	4	0x0	byte		Byte	byte
h	2	5	0h	short	smallint	Short	int16
i	4	6	0	int	int	Integer	int32
j	8	7	0j	long	bigint	Long	int64
e	4	8	0e	real	real	Float	single
f	8	9	0.0	float	float	Double	double
c	1	10	" "	char		Character	char
s	.	11	`	symbol	varchar	String	string
p	8	12	dateDtimespan	timestamp			
m	4	13	2000.01m	month			
d	4	14	2000.01.01	date	date	Date	
z	8	15	dateTtime	datetime	timestamp	Timestamp	Date...
n	8	16	00:00:00.000000000	timespan			
u	4	17	00:00	minute			
v	4	18	00:00:00	second			
t	4	19	00:00:00.000	time	time	Time	Time...
*	4	20..	`s$`	enum			
		98		table			
		99		dict			
		100		lambda			
		101		unary prim			
		102		binary prim			
		103		ternary(operator)			
		104		projection			
		105		composition			
		106		f'			
		107		f/			
		108		f\			
		109		f':			
		110		f/:			
		111		f\:			
		112		dynamic load			

the nested types are 77+t (e.g. 78 is boolean. 96 is time.)

`char$data `CHAR$string

The int, float, char and symbol literal nulls are: 0N 0n " " `. The rest use type extensions, e.g. 0Nd. No null for boolean or byte. 0Wd 0Wz 0Wt placeholder infinite dates/times/datetimes (no math)

date.(year month week mm dd) dict:`a`b!.. table:([]x:..;y:..) or +`x`y!.. time.(minute second mm ss) milliseconds=time mod 1000

Functions

A *function* takes its argument on the right. A result is optional.

=[[refheader(Syntax)]]=

The canonical form of the right argument is a semicolon-delimited list of parameters given in brackets:

```
f[a1;a2;...;an]
```

Note that the expression in brackets gives n parameters to the function, but is not itself a q list, i.e. it is *not* the same as:

```
(a1;a2;...;an)
```

Where a function takes only a single parameter, the brackets are not needed, so the following are the same:

```
f[a1]
f a1
```

Where a function takes no parameter, nevertheless an argument must be given for it to be executed; this argument is ignored.

Action on Lists

Functions can be atomic, aggregate or uniform (or none of these):

- an *atomic* function returns an atom for each atom in its argument
- an *aggregate* function returns an atom from a list
- a *uniform* function returns a list from a list

For example:

```
q)signum 0 2 -3 5          / atomic
0 1 -1 1
q)sum 2 3 5 7              / aggregate
17
q)sums 2 3 5 7             / uniform
2 5 10 17
q)distinct 2 3 5 7 2 5     / none of the above
2 3 5 7
```

Definition

A function is created by assignment, in one of two ways:

- as an explicit definition, given as a list of statements in matching braces
- using q functional forms: adverbs and function projection

Explicit Definition

An explicit definition has a list of statements, separated by semi-colons, and enclosed in matching braces, { }. The arguments to the definition can optionally be named, as a list in square brackets, `[ ]`. If the arguments are not so named, then names `x`, `y` and `z` refer to the first three arguments. The format is:

`f:{[p1;...;pn]e1;...;en}`

where the optional `[p1;...;pn]` are named arguments, and `e1;...;en` is a sequence of expressions to be evaluated.

A function can be defined and used without giving it a name, for example:

```
q){10+3*x} 1 2 3 13
13 16 19 49
```

The result of the function (if any) is the result of the last statement evaluated. If the last statement is empty, no result is returned.

Return and Signal

Return and signal end execution early:

- *return* ends successfully
- *signal* aborts execution (after an error)

To terminate and return a value, use assignment `:` with no name to the left.

```
q)f:{a:3*x;:a;a+10}            / the final a+10 is never executed
q)f 1
3
```

To abort function execution, use signal `'` with an error message to its right:

```
q)f:{a:3*x;'"end here";a+10}   / the final a+10 is never executed
q)f 1
'end here
10
```

Local and Global

Names assigned with `:` are local to the definition, and names assigned with `::` are global:

```
q)a:b:0                       / set (global) a and b to 0
q)f:{a:10+3*x;b::100+a;}      / f defines local a, global b
q)f 1 2 3                     / run f
q)a                           / global a is unchanged
0
q)b                           / global b is updated
113 116 119
```

A local variable exists only from the time it is first assigned. A name referenced that is not a local variable, is searched for globally:

```
q)a:100                      / assign global a
q)f:{a+x}                    / f references a
q)f 1 2 3
101 102 103
```

Conditional Evaluation

Functions `$` and `?` allow conditional evaluation, see:

- page **??** - DollarSign
- page **??** - QuestionSymbol

Control Words

Functions `if`, `do` and `while` control the order of execution, see:

- page **??** - if
- page **??** - do
- page **??** - while

Functional Forms

A function can also be defined from one of two functional forms: adverbs and function projection.

For *adverbs*, see page 19 - adverbs.

Function *projection* occurs when a function or verb is given only a subset of its arguments - this results in a new function whose arguments are those not yet given. For example, this is useful when a function is to be called repeatedly with some of its arguments unchanged:

```
q)f:{x + 2 * y - z}       / f takes 3 arguments
q)f[100;10;2 3 5]
116 114 110

q)g:f[100]                / g is projection of f on first argument
q)g[10;2 3 5]             / g takes 2 arguments
116 114 110

q)g:f[100;;2 3 5]         / project on first and third arguments
q)g 10                    / g now takes 1 argument
116 114 110
```

Verbs

A verb takes two arguments and can be called with its arguments to left and right. A verb can also be called with the same syntax as functions, for example:

```
q)2 3 5 + 10
12 13 15

q)+[2 3 5;10]
12 13 15
```

If only the left argument is given, the result is a function of one argument:

```
q)f:2 3 5 +
q)f 10
12 13 15
```

Several verbs are defined by the q system. A new verb can be defined only by applying an adverb to an existing verb, i.e. you cannot define a verb using explicit definition or function projection.

See also

- page 14 - Functions
- page 19 - Adverbs

Adverbs

An adverb takes a function or verb argument to its left, and produces a new function or verb respectively.

For adverbs that are symbols (i.e. adverbs other than `each`), there should be no whitespace between an adverb and its argument:

Example

```
q)reverse (1 2 3;"hello")
"hello"
1 2 3

q)reverse each (1 2 3;"hello")        / adverb each
3 2 1
"olleh"

q)1 2 3 , 100 200 300
1 2 3 100 200 300

q)1 2 3 ,' 100 200 300                / adverb ' (each-both)
1 100
2 200
3 300
```

See also:

* to do: add ref to list of adverbs

2

Names

abs *absolute value function*

The abs function computes the absolute value of its argument. Null is returned if the argument is null.

Syntax

```
q)r:abs X
```

Example

```
q)abs -1.0
1f
q)abs 10 -43 0N
10 43 0N
```

acos *arc cosine function*

The `acos` function computes the arc cosine of its argument; that is the value whose cosine is the given argument. The result is in radians and lies between 0 and PI (the range is approximate due to rounding errors).

Null is returned if the argument is not between -1 and 1.

Syntax

```
q)a:acos 0.5
```

Example

```
q)acos -0.4
1.982313
```

See also

- page 31 - asin
- page 34 - atan

aj *asof join function*

The function `aj` joins tables along columns that are usually time columns. The columns need not be keys.

In the join, the last value (most recent time) is taken.

Syntax

```
q)a:aj[c1...cn;t1;t2]
```

where $c_1 \ldots c_n$ is a symbol list of common column names, and `t1` and `t2` are the tables to be joined. The result is a table with records from the left join of `t1` and t2.

For each record in `t1`, the result has one record with the items in `t1`, and

- if there are matching records in `t2`, the items of the last (in row order) matching record are appended to those of `t1`
- if there is no matching record in `t2`, the remaining columns are null

Example

```
q)t
ti       sym  qty
-----------------
10:01:01 msft 100
10:01:03 ibm  200
10:01:04 ge   150

q)q
ti       sym  px
-----------------
10:01:00 ibm  100
10:01:01 msft 99
10:01:01 msft 101
10:01:03 ibm  98

q)aj[`ti`sym;t;q]
ti       sym  qty px
---------------------
10:01:01 msft 100 101
10:01:03 ibm  200 98
10:01:04 ge   150
```

Remarks

1. `aj` should run at a million or two trade records per second; whether trade/quote are mapped or not is irrelevant. However, for speed,

- in-memory quote must have `` `g#sym `` and `` `s#time ``
- on-disk quote must have `` `p#sym `` and time sorted within sym

Note that on-disk `` `g#sym `` does not help.

2. Unlike in memory, to use `aj` with on-disk, you must map in your splay or day-at-a-time partitioned db:

Splay:

```
aj[`sym`time;select .. from trade where ..;select .. from quote]
```

Partitioned db:

```
aj[`sym`time;select .. from trade where ..;
             select .. from quote where date = ..]
```

Further *where* constraints cannot be used, or the columns will be copied instead of mapped into memory (resulting in slowdown for the `aj`).

3. There is no need to select on quote, i.e. irrespective of the number of quote records, use:

```
aj[`sym`time;select .. from trade where ..;quote]
```

instead of

```
aj[`sym`time;select .. from trade where ..;
             select .. from quote where ..]
```

See also

- page **??** - joins
- page 32 - asof
- page **??** - lj
- page **??** - uj

all *all function*

Function `all` returns a boolean atom `1b` if all values in its argument are non-zero, and otherwise `0b`.

It applies to all data types except symbol, first converting the type to boolean if necessary.

Syntax

```
q)r:all A
```

Example

```
q)all 1 2 3=1 2 4
q)all 1 2 3=1 2 3
q)if[all x in y;....]   / use in control structure
```

Remark

`all` is defined as `min` after converting to boolean, i.e. the following are the same:

```
q)all x
q)min "b"$x
```

See also

- page 28 - any

and *and verb*

Verb `and` returns the minimum of its arguments. It applies to all data types except symbol.

Syntax

```
q)X and Y
```

Example

```
q)1100b and 1010b
1000b
q)1b and 10b
10b
q)1b and 0b
0b
```

Remark

The name `and` is used because it behaves as logical AND on boolean arguments, but is extended to minimum on other data types:

```
q)-2 0 3 7 and 0 1 3 4
-2 0 3 4
```

See also

- page **??** - or

any *any function*

Function `any` returns a boolean atom `1b` if any value in its argument is non-zero, and otherwise `0b`.

It applies to all data types except symbol, first converting the type to boolean if necessary.

Syntax

```
q)r:any X
```

Example

```
q)any 1 2 3=10 20 4
0b
q)any 1 2 3=1 20 30
1b
q)if[any x in y;....]   / use in control structure
```

Remark

`any` is defined as `max` after converting to boolean, i.e. the following are the same:

```
q)any x
q)max "b"$x
```

See also

- page 26 - all

asc *ascending function*

Function `asc` sorts a list.

- on a simple list `asc` acts as expected, and sets the `` `s# `` attribute indicating that the list is sorted.
- on a mixed list it sorts within datatype.
- when applied to a dictionary or table, it sets the `` `s# `` attribute on the first key value or column respectively (if possible), and then sorts by that key or column.

Syntax

```
q)r:asc A
```

Examples

```
q)asc 2 1 3 4 2 1 2
`s#1 1 2 2 2 3 4
```

The next example sorts a mixed list. Note that the boolean is returned first, then the sorted integers, the sorted characters, and then the date:

```
q)asc (1;1b;"b";2009.01.01;"a";0)
1b
0
1
"a"
"b"
2009.01.01
```

Example of sorting a table:

```
q)t:([]a:3 4 1;b:`a`d`s)
q)asc t
a b
---
1 s
3 a
4 d
```

See also

- page **??** - desc
- page **??** - iasc
- page **??** - idesc

asin *arc sine function*

The `asin` function computes the arc sine of its argument; that is the value whose sine is the given argument. The result is in radians and lies between -$\pi/2$ and $\pi/2$ (the range is approximate due to rounding errors).

Null is returned if the argument is not between -1 and 1.

Syntax

```
q)a:asin 0.5
```

Example

```
q)asin 0.8
0.9272952
```

See also

- page 22 - acos
- page 34 - atan

asof *asof verb*

The verb `asof` takes a table as the left argument and a dictionary or a table as a right argument. The last key of the dictionary, or the last column of the table, on the right must correspond to a time column of the table on the left. The result is the values from the last rows matching the rest of the keys and time less or equal to the time in the right hand side table.

Examples

```
q)show trade asof`sym`time!(`IBM;09:30:00.0)
price| 96.3e
size | 200
stop | 0b
corr | 0
cond | "T"
ex   | "D"

q)show trade asof([]sym:`AAPL`IBM;ex:"TD";time:09:30:00.0)
price size stop corr cond
-------------------------
78.14 100  0    0    T
96.3  200  0    0    T
```

The following examples use the `mas` table from TAQ

```
q)`date xasc`mas       / sort by date
`mas

q)show a!mas asof a:([]sym:`A`B`C`GOOG;date:1995.01.01)
sym  date      | cusip     name                              wi ex uot
---------------| ---------------------------------------------------
A    1995.01.01| 049870207 ATTWOODS PLC ADS REP5 ORD/5PNC 0  N  100
B    1995.01.01| 067806109 BARNES GROUP INCORPORATED      0  N  100
C    1995.01.01| 171196108 CHRYSLER CORP                  0  N  100
GOOG 1995.01.01|                                         0
```

```
q)show a!mas asof a:([]sym:`A`B`C`GOOG;date:2006.01.01)
sym  date      | cusip     name                       wi ex uot
---------------| -----------------------------------------
A    2006.01.01| 00846U101 AGILENT TECHNOLOGIES, INC 0  N  100
B    2006.01.01| 067806109 BARNES GROUP INCORPORATED 0  N  100
C    2006.01.01| 172967101 CITIGROUP                 0  N  100
GOOG 2006.01.01| 38259P508 GOOGLE INC CLASS A        0  T  100

q)show a!mas asof a:([]sym:`A;date:1993.01.05 1996.05.23 2000.08...
sym date      | cusip     name                           wi ex uot
--------------| ------------------------------------------------
A   1993.01.05| 049870207 ATTWOODS PLC ADS REP5 ORD/5PNC 0  N  100
A   1996.05.23| 046298105 ASTRA AB CL-A ADS 1CL-ASEK2.50 0  N  100
A   2000.08.04| 00846U101 AGILENT TECHNOLOGIES  INC      0  N  100
```

atan *arc tangent function*

The `atan` function computes the arc tangent of its argument; that is the value whose tangent is the given argument. The result is in radians and lies between -$\pi/2$ and $\pi/2$ (the range is approximate due to rounding errors).

Null is returned if the argument is not between -1 and 1.

Syntax

```
q)a:atan 0.5
```

Example

```
q)atan 0.5
0.4636476
q)atan 42
1.546991
```

See also

- page 22 - acos
- page 31 - asin

attr *attributes function*

Function `attr` returns the attributes of its argument. It can be applied to all data types. The possible attributes are:

- `` `s# `` sorted
- `` `u# `` unique (hash table)
- `` `p# `` partitioned (grouped)
- `` `g# `` true index (dynamic attribute): enables constant time update and access for realtime tables

Syntax

```
q)r:attr L
```

The result is a symbol atom and is one of `` `s`u`p`g` `` with `` ` `` meaning no attributes are set on the argument.

Example

```
q)attr 1 3 4
`
q)attr asc 1 3 4
`s
```

See also

Q for Mortals

avg *average function*

Computes the arithmetic mean of a list of numbers.

Null is returned if the list is empty, or contains both positive and negative infinity. Any null elements in the input list are ignored.

Example

```
q)avg 1 2 3
2f
q)avg 1 0n 2 3       / null values are ignored
2f
q)avg 1.0 0w
0w
q)avg -0w 0w
0n
q)\l trade.q
q)show select ap:avg price by sym from trade
sym| ap
---| -----
a  | 10.75
```

avgs *running averages function*

Computes the running averages of a list of numbers, i.e. applies function `avg` to successive prefixes of the argument.

Example

```
q)avgs 1 2 3 0n 4 -0w 0w
1 1.5 2 2 2.5 -0w 0n
```