Reference card

Keywords

abs	cor	ej	gtime	like	mins	prev	scov	system	wavg
acos	cos	ema	hclose	lj ljf	mmax	prior	sdev	tables	where
aj aj0	count	enlist	hcount	load	mmin	rand	select	tan	while
ajf ajf0	cov	eval	hdel	log	mmu	rank	set	til	within
all	cross	except	hopen	lower	mod	ratios	setenv	trim	wj wj1
and	csv	exec	hsym	lsq	msum	raze	show	type	wsum
any	cut	exit	iasc	ltime	neg	read0	signum	uj ujf	xasc
asc	delete	exp	idesc	ltrim	next	read1	sin	ungroup	xbar
asin	deltas	fby	if	mavg	not	reciprocal	sqrt	union	xcol
asof	desc	fills	ij ijf	max	null	reval	ss	update	xcols
atan	dev	first	in	maxs	or	reverse	ssr	upper	xdesc
attr	differ	fkeys	insert	mcount	over	rload	string	upsert	xexp
avg	distinct	flip	inter	md5	parse	rotate	sublist	value	xgroup
avgs	div	floor	inv	mdev	peach	rsave	sum	var	xkey
bin binr	do	get	key	med	pj	rtrim	sums	view	xlog
ceiling	dsave	getenv	keys	meta	prd	save	sv	views	xprev
cols	each	group	last	min	prds	scan	svar	vs	xrank

By category

control	do, exit, if, while
env	getenv, gtime, ltime, setenv
interpret	eval, parse, reval, show, system, value
io	dsave, get, hclose, hcount, hdel, hopen, hsym, load, read0, read1, rload, rsave, save, set
iterate	each, over, peach, prior, scan
join	aj aj0, ajf ajf0, asof, ej, ij ijf, lj ljf, pj, uj ujf, wj wj1
list	count, cross, cut, enlist, except, fills, first, flip, group, in, inter, last, mcount, next, prev, raze, reverse, rotate, sublist, sv, til, union, vs, where, xprev
logic	all, and, any, not, or
math	abs, acos, asin, atan, avg, avgs, ceiling, cor, cos, cov, deltas, dev, div, ema, exp, floor, inv, log, lsq, mavg, max, maxs, mdev, med, min, mins, mmax, mmin, mmu, mod, msum, neg, prd, prds, rand, ratios, reciprocal, scov, sdev, signum, sin, sqrt, sum, sums, svar, tan, var, wavg, within, wsum, xexp, xlog
meta	attr, null, tables, type, view, views
query	delete, exec, fby, select, update
sort	asc, bin binr, desc, differ, distinct, iasc, idesc, rank, xbar, xrank
table	cols, csv, fkeys, insert, key, keys, meta, ungroup, upsert, xasc, xcol, xcols, xdesc, xgroup, xkey
text	like, lower, ltrim, md5, rtrim, ss, ssr, string, trim, upper

`.Q.id` (sanitize), `.Q.res` (reserved words)

Operators

`.` Apply, Index, Trap, Amend

`@` Apply At, Index At, Trap At, Amend At

`$`	Cast, Tok, Enumerate, Pad, `mmu`
`!`	Dict, Enkey, Unkey, Enumeration, Flip Splayed, Display, internal, Update, Delete, `lsq`
`?`	Find, Roll, Deal, Enum Extend, Select, Exec, Simple Exec, Vector Conditional
`+ - * %`	Add, Subtract, Multiply, Divide
`= <> ~`	Equals, Not Equals, Match
`< <= >= >`	Less Than, Up To, At Least, Greater Than
`| &`	Greater (OR), Lesser, AND
`#`	Take, Set Attribute
`^`	Fill, Coalesce
`0: 1: 2:`	File Text, File Binary, Dynamic Load
`0 ±1 ±2 ±n`	write to console, stdout, stderr, handle *n*

`_`	Cut, Drop	`:`	Assign
`,`	Join	`'`	Compose

Overloaded glyphs

Iterators

```
maps                                          accumulators
'   Each, each, Case        /:   Each Right     /   Over, over
':  Each Parallel, peach    \:   Each Left      \   Scan, scan
':  Each Prior, prior
```

Execution control

```
.[f;x;e] Trap          : Return        do  exit        $[x;y;z] Cond
@[f;x;e] Trap-At       ' Signal        if  while
```

Debugging

Other

```
`   pop stack        ::   identity         \x  system cmd x
.   push stack            generic null     \    abort
                          global amend     \\   quit q
                          set view         /    comment


()     precedence    [;]  expn block       {}  lambda       `   symbol
(;)    list               argt list        ;   separator    `:  filepath
([]..) table
```

Attributes

```
g grouped     p parted     s sorted     u unique
```

Set Attribute

Command-line options and system commands

file

\a	tables	\r	rename
-b	blocked	-s \s	secondary processes
\b \B	views	-S \S	random seed
-c \c	console size	-t \t	timer ticks
-C \C	HTTP size	\ts	time and space
\cd	change directory	-T \T	timeout
\d	directory	-u -U \u	usr-pwd
-e \e	error traps	-u	disable syscmds
-E \E	TLS server mode	\v	variables
\f	functions	-w \w	memory
-g \g	garbage collection	-W \W	week offset
\l	load file or directory	\x	expunge
-l -L	log sync	-z \z	date format
-o \o	UTC offset	\1 \2	redirect
-p \p	listening port	_	hide q code
-P \P	display precision	\	terminate
-q	quiet mode	\	toggle q/k
-r \r	replicate	\\	quit

system

Command-line options, System commands, OS commands

Datatypes

```
Basic datatypes
n   c   name      sz  literal            null inf SQL       Java      .Net
-------------------------------------------------------------------------------
0   *   list
1   b   boolean   1   0b                                   Boolean   boolean
2   g   guid      16                     0Ng               UUID      GUID
4   x   byte      1   0x00                                 Byte      byte
5   h   short     2   0h                 0Nh  0Wh smallint  Short     int16
6   i   int       4   0i                 0Ni  0Wi int       Integer   int32
7   j   long      8   0j                 0Nj  0Wj bigint    Long      int64
                      0                  0N   0W
8   e   real      4   0e                 0Ne  0We real      Float     single
9   f   float     8   0.0                0n   0w  float     Double    double
                      0f                 0Nf
10  c   char      1   " "                " "               Character char
11  s   symbol        `                  `        varchar
12  p   timestamp 8   dateDtimespan      0Np  0Wp           Timestamp DateTime (RW)
13  m   month     4   2000.01m           0Nm
14  d   date      4   2000.01.01         0Nd  0Wd date      Date
15  z   datetime  8   dateTtime          0Nz  0wz timestamp Timestamp DateTime (RO)
16  n   timespan  8   00:00:00.000000000 0Nn  0Wn           Timespan  TimeSpan
17  u   minute    4   00:00              0Nu  0Wu
18  v   second    4   00:00:00           0Nv  0Wv
19  t   time      4   00:00:00.000       0Nt  0Wt time      Time      TimeSpan
```

```
Columns:
n    short int returned by type and used for Cast, e.g. 9h$3
c    character used lower-case for Cast and upper-case for Tok and Load CSV
sz   size in bytes
inf  infinity (no math on temporal types); 0Wh is 32767h

RO: read only; RW: read-write

Other datatypes
20-76   enums
77      anymap                                  104  projection
78-96   77+t - mapped list of lists of type t   105  composition
97      nested sym enum                         106  f'
98      table                                   107  f/
99      dictionary                              108  f\
100     lambda                                  109  f':
101     unary primitive                         110  f/:
102     operator                                111  f\:
103     iterator                                112  dynamic load
```

Above, `f` is an applicable value.

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

Cast `$`: where `char` is from the `c` column above `char$data:CHAR$string`

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

Namespaces

`.h` (markup)

HTTP, markup and data conversion.

`.j` (JSON)

De/serialize as JSON.

`.m` (memory backed files)

Memory backed by files.

`.Q` (utils)

Utilities: general, environment, IPC, datatype, database, partitioned database state, segmented database state, file I/O, debugging, profiling.

`.z` (environment, callbacks)

Environment, callbacks

Datatypes

```
Basic datatypes
n   c   name      sz  literal            null inf SQL

-----------------------------------------------------------
0   *   list
1   b   boolean   1   0b
2   g   guid      16                     0Ng
4   x   byte      1   0x00

5   h   short     2   0h                 0Nh  0Wh smallint
6   i   int       4   0i                 0Ni  0Wi int
7   j   long      8   0j                 0Nj  0Wj bigint
                      0                  0N   0W
8   e   real      4   0e                 0Ne  0We real
9   f   float     8   0.0                0n   0w  float
                      0f                 0Nf
10  c   char      1   " "                " "
11  s   symbol        `                  `        varchar
12  p   timestamp 8   dateDtimespan      0Np  0Wp
13  m   month     4   2000.01m           0Nm  0Wm
14  d   date      4   2000.01.01         0Nd  0Wd date
15  z   datetime  8   dateTtime          0Nz  0wz timestamp
16  n   timespan  8   00:00:00.000000000 0Nn  0Wn
17  u   minute    4   00:00              0Nu  0Wu
18  v   second    4   00:00:00           0Nv  0Wv
19  t   time      4   00:00:00.000       0Nt  0Wt time

Columns:
n    short int returned by type and used for Cast, e.g. 9h$3
c    character used lower-case for Cast and upper-case for Tok and Load CSV
sz   size in bytes
inf  infinity (no math on temporal types); 0Wh is 32767h

RO: read only; RW: read-write

Other datatypes
20-76   enums
77      anymap                                  104  projection
78-96   77+t - mapped list of lists of type t   105  composition
97      nested sym enum                         106  f'
98      table                                   107  f/
99      dictionary                              108  f\
100     lambda                                  109  f':
101     unary primitive                         110  f/:
102     operator                                111  f\:
103     iterator                                112  dynamic load
```

Above, `f` is an applicable value.

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

The type is a short int:

- zero for a general list
- negative for atoms of basic datatypes
- positive for everything else

📖 Cast, Tok, `type`, `key`, `.Q.ty` (type)
📖🎓 Temporal data, Timezones

Basic types

> **The default type for an integer is long (`7h` or `"j"`).**
>
> Before V3.0 it was int (`6h` or `"i"`).

Strings

There is no string datatype. On this site, *string* is a synonym for character vector (type `10h`). In q, the nearest equivalent to an atomic string is the symbol.

Strings can include multibyte characters, which each occupy the respective number of bytes. For example, assuming that the input encoding is UTF-8:

```
q){(x;count x)}"Zürich"
"Z\303\274rich"
7
q){(x;count x)}"日本"
"\346\227\245\346\234\254"
6
```

Other encodings may give different results.

```
q)\chcp
"Active code page: 850"
q)"Zürich"
"Z\201rich"

q)\chcp 1250
"Active code page: 1250"
q)"Zürich"
"Z\374rich"
```

🎓 Unicode

Temporal

The valid date range is `0001.01.01` to `9999.12.31`. (Since V3.6 2017.10.23.)

> ⚠ **The 4-byte datetime datatype (15) is deprecated in favour of the 8-byte timestamp datatype (12).**

```
q)"D"$"3001.01.01"
3001.01.01
```

Internally, dates, times and timestamps are represented by integers:

```
q)show noon:`minutes`seconds`nanoseconds!(12:00;12:00:00;12:00:00.000000000)
minutes    | 12:00
seconds    | 12:00:00
nanoseconds| 0D12:00:00.000000000
q)"j"$noon
minutes    | 720
seconds    | 43200
nanoseconds| 43200000000000
```

Date calculations assume the proleptic Gregorian calendar.

Casting to timestamp from date or datetime outside of the timestamp supported year range results in `±0Wp`. Out-of-range dates and datetimes display as `0000.00.00` and `0000.00.00T00:00:00:.000`.

```
q)`timestamp$1666.09.02
-0Wp
q)0001.01.01-1
0000.00.00
q)"z"$0001.01.01-1
0000.00.00T00:00:00.000
```

Valid ranges can be seen by incrementing or decrementing the infinities.

```
q)-0W 0Wp+1 -1      / limit of timestamp type
1707.09.22D00:12:43.145224194 2292.04.10D23:47:16.854775806

q)0p+ -0W 0Wp+1 -1  / timespan offset of those from 0p
-106751D23:47:16.854775806 106751D23:47:16.854775806

q)-0W 0Wn+1 -1      / coincide with the min/max for timespan
```

Symbols

A back tick ` followed by a series of characters represents a *symbol*, which is not the same as a string.

```
q)`symbol ~ "symbol"
0b
```

A back tick without characters after it represents the *empty symbol*: ` .

Cast string to symbol

The empty symbol can be used with Cast to cast a string into a symbol, creating symbols whose names could not otherwise be written, such as symbols containing spaces. `` `$x `` is shorthand for `"S"$x`.

```
q)s:`hello world
'world
q)s:`$"hello world"
q)s
`hello world
```

Q for Mortals: §2.4 Basic Data Types – Atoms

Filepaths

Filepaths are a special form of symbol.

```
q)count read0 `:path/to/myfile.txt  / count lines in myfile.txt
```

Infinities

Note that arithmetic for integer infinities (`0Wh` , `0Wi` , `0Wj`) is undefined, and does not retain the concept when cast.

```
q)0Wi+5
2147483652
q)0Wi+5i
-2147483644i
q)`float$0Wj
9.223372e+18
q)`float$0Wi
2.147484e+09
```

Arithmetic for float infinities (`0we` , `0w`) behaves as expected.

```
q)0we + 5
0we
q)0w + 5
0w
```

`.Q.M` (long infinity)

To infinity and beyond

Floating-point arithmetic follows IEEE754.

Integer arithmetic does no checks for infinities, just treats them as a signed integer.

```
q)vs[0b]@/:0N!0W+til 3
0W 0N -0W
01111111111111111111111111111111111111111111111111111111111111111b
10000000000000000000000000000000000000000000000000000000000000000b
10000000000000000000000000000000000000000000000000000000000000001b
```

but it does check for nulls.

```
q)10+0W+til 3
-9223372036854775799 0N -9223372036854775797
```

This can be **abused** to push infinities on nulls which then become sticky and can be filtered out altogether, e.g.

```
q)1+-1+-1+1+ -0W 0N 0W 1 2 3
0N 0N 0N 1 2 3
```

There is no display for short infinity.

```
q)0Wh
32767h
q)-0Wh
-32767h
```

Integer promotion is documented for Add.

Integer infinities

- do not promote, other than the signed bit; there is no special treatment over any other int value
- map to int_min+1 and int_max, with `0N` as int_min; so there is no number smaller than `0N`

Best practice is to view infinities as placeholders only, and not perform arithmetic on them.

Guid

The guid type (since V3.0) is a 16-byte type, and can be used for storing arbitrary 16-byte values, typically transaction IDs.

Generation

Use Deal to generate a guid (global unique: uses `.z.a .z.i .z.p`).

```
q)-2?0Ng
337714f8-3d76-f283-cdc1-33ca89be59e9 0a369037-75d3-b24d-6721-5a1d44d4bed5
```

If necessary, manipulate the bytes to make the uuid a Version-4 'standard' uuid.

Guids can also be created from strings or byte vectors, using `sv` or `"G"$`, e.g.

```
q)0x0 sv 16?0xff
8c680a01-5a49-5aab-5a65-d4bfddb6a661
q)"G"$"8c680a01-5a49-5aab-5a65-d4bfddb6a661"
8c680a01-5a49-5aab-5a65-d4bfddb6a661
```

`0Ng` is null guid.

```
q)0Ng
00000000-0000-0000-0000-000000000000
q)null 0Ng
1b
```

There is no literal entry for a guid, it has no conversions, and the only scalar primitives are `=`, `<` and `>` (similar to sym). In general, since V3.0, there should be no need for char vectors for IDs. IDs should be int, sym or guid. Guids are faster (much faster for `=`) than the 16-byte char vecs and take 2.5 times less storage (16 per instead of 40 per).

Other types

Enumerated types

Enumerated types are numbered from `20h` up to `76h`. For example, in a new session with no enumerations defined:

```
q)type `sym$10?sym:`AAPL`AIG`GOOG`IBM
20h
q)type `city$10?city:`london`paris`rome
20h
```

(Since V3.0, type `20h` is reserved for `` `xxx$ `` where `xxx` is the name of a variable.)

Enumerate, Enumeration, Enum Extend
Enumerations

Nested types

These types are used for mapped lists of lists of the same type. The numbering is 77 + primitive type (e.g. 77 is anymap, 78 is boolean, 96 is time and 97 is `` `sym$ `` enumeration.)

```
q)`:t1.dat set 2 3#til 6
`:t1.dat
q)a:get `:t1.dat
q)type a            /integer nested type
83h
q)a
0 1 2
3 4 5
```

Dictionary and table

Dictionary is `99h` and table is `98h`.

```
q)type d:`a`b`c!(1 2;3 5;7 11)     / dict
99h
q)type flip d                      / table
98h
```

Functions, iterators, derived functions

Functions, lambdas, operators, iterators, projections, compositions and derived functions have types in the range [100–112].

```
q)type each({x+y};neg;-;\;+[;1];<>;,';+/;+\;prev;+/:;+\:;`f 2:`f,1)
100 101 102 103 104 105 106 107 108 109 110 111 112h
```

Overloaded glyphs

Many non-alphabetic keyboard characters are overloaded. Operator overloads are resolved by **rank**, and sometimes by the **type** of argument/s.

@ at

rank	syntax	semantics
2	`l@i` , `@[l;i]`	Index At
2	`f@y` , `@[f;y]`	Apply At
3	`@[f;y;e]`	Trap At
3	`@[d;i;u]`	Amend At
4	`@[d;i;m;my]`	Amend At
4	`@[d;i;:;y]`	Replace At

\ backslash

rank	syntax	semantics
n/a	`\`	ends multiline comment
n/a	`\`	Abort, Toggle
1	`(u\)` , `u\[d]`	Converge
2	`n u\d` , `u\[n;d]`	Do
2	`t u\d` , `u\[t;d]`	While
2	`x v\y` , `v\[x;y;z;…]`	map-reduce

```
d: data                 n: non-negative integer atom
u: unary value          t: test value
v: value rank>1         x: atom or vector
                        y, z…: conformable atoms or lists
```

! bang

rank	syntax	semantics
2	`x!y`	Dict: make a dictionary
2	`i!ts`	Enkey: make a simple table keyed
2	`0!tk`	Unkey: make a keyed table simple
2	`noasv!iv`	Enumeration from index
2	`sv!h`	Flip Splayed or Partitioned
2	`0N!y`	display `y` and return it
2	`-i!y`	internal function
4	`![t;c;b;a]`	Update, Delete

```
a: select specifications
b: group-by specifications
c: where-specifications
h: handle to a splayed or partitioned table
i: integer >0
noasv: symbol atom, the name of a symbol vector
sv: symbol vector
t: table
tk: keyed table
ts: simple table
x,y: same-length lists
```

: colon

```
a:42    assign
:42     explicit return
```

: : colon colon

```
v::select from t where a in b     define a view
global::42                        amend a global from within a lambda
::                                Identity
::                                Null
```

- dash

Syntax: immediately left of a number, indicates its negative.

```
q)neg[3]~-3
1b
```

Otherwise

rank	example	semantics
2	`2-3`	Subtract

. dot

rank	syntax	semantics
2	`l . i , .[l;i]`	Index
2	`g . gx , .[g;gx]`	Apply
3	`.[g;gx;e]`	Trap
3	`.[d;i;u]`	Amend
4	`.[d;i;m;my]`	Amend
4	`.[d;i;:;y]`	Replace

In the Debugger, push the stack.

$ dollar

rank	example	semantics
3	`$[x>10;y;z]`	Cond: conditional evaluation
2	`"h"$y , `short$y , 11h$y`	Cast: cast datatype
2	`"H"$y , -11h$y`	Tok: interpret string as data
2	`x$y`	Enumerate `y` from `x`
2	`10$"abc"`	Pad: pad string
2	`(1 2 3f;4 5 6f)$(7 8f;9 10f;11 12f)`	dot product, matrix multiply, `mmu`

hash

rank	example	semantics
2	`2 3#til 6`	Take
2	`s#1 2 3`	Set Attribute

? query

rank	example	semantics
2	`"abcdef"?"cab"`	Find `y` in `x`
2	`10?1000 , 5?01b`	Roll
2	`-10?1000 , -1?`yes`no`	Deal
2	`0N?1000 , 0N?`yes`no`	Permute
2	`x?v`	extend an enumeration: Enum Extend
3	`?[11011b;"black";"flock"]`	Vector Conditional

rank	example	semantics
3	`?[t;i;p]`	Simple Exec
4	`?[t;c;b;a]`	Select, Exec
5	`?[t;c;b;a;n]`	Select
6	`?[t;c;b;a;n;(g;cn)]`	Select

' quote

rank	syntax	semantics
1	`(u')x`, `u'[x]`, `x b'y`, `v'[x;y;…]`	Each: iterate `u`, `b` or `v` itemwise
1	`'msg`	Signal an error
1	`int'[x;y;…]`	Case: successive items from lists
2	`'[u;v]`	Compose `u` with `v`

```
u:  unary value          int:  int vector
b:  binary value         msg:  symbol or string
v: value of rank ≥1      x, y: data
```

': quote-colon

rank	example	semantics
1	`u':`	Each Parallel with unary `u`
1	`b':`	Each Prior with binary `b`

/ slash

rank	syntax	semantics
n/a	`/a comment`	comment: ignore rest of line
1	`(u/)y , u/[y]`	Converge
1	`n u/ y , u/[n;y]`	Do
1	`t u/ y , u/[t;y]`	While
1	`(v/)y , v/[y]`	map-reduce: reduce a list or lists

```
u: unary value              t: test value
v: value rank ≥1            y: list
n: non-negative int atom
```

Syntax: a space followed by `/` begins a **trailing comment**. Everything to the right of `/` is ignored.

```
q)2+2 / we know this one
4
```

A `/` at the beginning of a line marks a **comment line**. The entire line is ignored.

```
q)/nothing in this line is evaluated
```

In a script, a line with a solitary `/` marks the beginning of a **multiline comment**. A multiline comment is terminated by a `\` or the end of the script.

```
/
A script to add two numbers.
Version 2018.1.19
\
2+2
/
That's all folks.
```

_ underscore

rank	example	semantics
2	`3_ til 10`	Cut, Drop

Names can contain underscores

Best practice is to use a space to separate names and the Cut and Drop operators.

Unary forms

Many of the operators tabulated above have unary forms in k.

☞ Exposed infrastructure

Application, projection, and indexing

Values

Everything in q is a value, and almost all values can be applied.

- A list can be applied to its indexes to get its items.
- A list with an elided item or items can be applied to a fill item or list of items
- A dictionary can be applied to its keys to get its values.
- A matrix can be applied its row indexes to get its rows; or to its row and column indexes to get its items.
- A table can be applied to its row indexes to get its tuples; to its column names to get its columns; or to its row indexes and column names to get its items.
- A function (operator, keyword, or lambda) can be applied to its argument/s to get a result.
- A file or process handle can be applied to a string or parse tree

The *domain* of a function is all valid values of its argument/s; its *range* is all its possible results. For example, the domain of Add is numeric and temporal values, as is its range. By extension,

- the domain of a list is its indexes; its range, its items
- the domains of a matrix are its row and column indexes
- the domain of a dictionary is its keys; its range is its values
- the domains of a table are its row indexes and column names

Atoms need not apply

The only values that cannot be applied are atoms that are not file or process handles, nor the name of a variable or lambda.

In what follows, *value* means *applicable value*.

Application and indexing

Most programming languages treat the indexing of arrays and the application of functions as separate. Q conflates them. This is deliberate, and fundamental to the design of the language.

It also provides useful alternatives to control structures. See *Application and indexing* below.

Q for Mortals §6.5 Everything Is a Map

Application

To *apply a value* means

- to evaluate a function on its arguments
- to select items from a list or dictionary
- to write to a file or process handle

The syntax provides several ways to apply a value.

Bracket application

All values can be applied with bracket notation.

```
q)"abcdef"[1 4 3]
"bed"
q)count[1 4 3]
3
q){x*x}[4]
16
q)+[2;3]
5
q)d:`cat`cow`dog`sheep!`chat`vache`chien`mouton
q)d[`cow`sheep]
`vache`mouton
q)ssr["Hello word!";"rd";"rld"]
"Hello world!"
q)m:("abc";"def";"ghi";"jkl")       / a matrix
q)m[3 1]                            / m is a list (unary)
"jkl"
"def"
q)m[0;2 0 1]                        / and also a matrix (binary)
"cab"
q)main[]                            / nullary lambda
```

Infix application

Operators, and some binary keywords and derived functions can also be applied infix.

```
q)2+3                           / operator
5
q)2 3 4 5 mod 2                 / keyword
0 1 0 1
q)1000+\2 3 4                   / derived function
1002 1005 1009
```

Apply operator

Any applicable value can be applied by the Apply operator to a list of its arguments: one item per argument.

```
q)(+) . 2 3                        / apply + to a list of its 2 arguments
5
q).[+;2 3]                         / apply + to a list of its 2 arguments
5
q)ssr . ("Hello word!";"rd";"rld")  / apply ssr to a list of its 3 arguments
"Hello world!"
q)count . enlist 1 4 3             / apply count to a list of its 1 argument
3
```

Apply At operator

Lists, dictionaries and unary functions can be applied more conveniently with the Apply At operator.

```
q)"abcdef"@1 4 3
"bed"
q)@[count;1 4 3]
3
q)d @ `cow`sheep                   / dictionary to its keys
`vache`mouton
q)@[d;`cow`sheep]                  / dictionary to its keys
`vache`mouton
```

Apply At is syntactic sugar: `x@y` is equivalent to `x . enlist y`.

Prefix application

Lists, dictionaries and unary keywords and lambdas can also be applied prefix. As this is equivalent to simply omitting the Apply At operator, the `@` is mostly redundant.

```
q)"abcdef" 1 4 3
"bed"
q)count 1 4 3
3
q){x*x}4
16
q)d`cow`sheep
`vache`mouton
```

Postfix application

Iterators are unary operators that can be (and almost always are) applied postfix. They derive functions from their value arguments.

Some derived functions are variadic: they can be applied either unary or binary.

```
q)+\[2 3 4]                        / derived fn applied unary
2 5 9
q)+\[1000;2 3 4]                   / derived fn applied binary
```

```
1002 1005 1009
q)count'[("the";"quick";"brown";"fox")] / derived fn applied unary
3 5 5 3
```

Postfix yields infix.

Functions derived by applying an iterator postfix have infix syntax – *no matter how many arguments they take*.

Derived functions `+\` and `count'` have infix syntax. They can be applied unary by parenthesizing them.

```
q)(+\)2 3 4
100 1005 1009
q)(count')("the";"quick";"brown";"fox")
3 5 5 3
```

Application syntax

```
rank   bracket                                  other
of f   notation       Apply            Apply At  syntax        note
.....................................................................
0      f[]            f . enlist(::)   f@(::)
1      f[x]           f . enlist x     f@x       f x,  x f     prefix, postfix
2      f[x;y]         f . (x;y)                  x f y         infix
3-8    f[x;y;z;…]     f . (x;y;z;…)
```

Long right scope

Values applied prefix or infix have long right scope. In other words:

When a unary value is applied prefix, its argument is *everything to its right*.

```
q)sqrt count "It's about time!"
4
```

When a binary value is applied infix, its right argument is *everything to its right*.

```
q)7 * 2 + 4
42
```

Republic of values

There is no precedence among values. In `7*2+4` the right argument of `*` is the result of evaluating the expression on its right.

This rule applies without exception.

Iterators

The iterators are almost invariably applied postfix.

```
q)+/[17 13 12]
42
```

In the above, the Over iterator `/` is applied postfix to its single argument `+` to derive the function `+/` (sum).

An iterator applied postfix has *short left scope*. That is, its argument is the *value immediately to its left*. For the Case iterator that value is an int vector. An iterator's argument may itself be a derived function.

```
q)txt:(("Now";"is";"the";"time");("for";"all";"good";"folk"))
q)txt
"Now" "is"  "the"  "time"
"for" "all" "good" "folk"
q)count[txt]
2
q)count'[txt]
4 4
q)count''[txt]
3 2 3 4
3 3 4 4
```

In the last example, the derived function `count'` is the argument of the second `'` (Each).

Only iterators can be applied postfix.

📖 Apply/Index and Apply/Index At for how to apply functions and index lists

Rank and syntax

The *rank* of a value is the number of

- arguments it evaluates, if it is a function
- indexes required to select an atom, if it is a list or dictionary

A value is *variadic* if it can be used with more than one rank. All matrixes and some derived functions are variadic.

```
q)+/[til 5]          / unary
10
q)+/[1000000;til 5]  / binary
1000010
```

Rank is a semantic property, and is *independent of syntax*. This is a ripe source of confusion.

Postfix yields infix

The syntax of a derived function is determined by the application that produced it.

The derived function `+/` is variadic but has infix syntax. Applying it infix is straightforward.

```
q)1000000+/til 5
1000010
```

How then to apply it as a unary? Bracket notation 'overrides' infix syntax.

```
q)+/[til 5]           / unary
10
q)+/[1000000;til 5]   / binary
1000010
```

Or isolate it with parentheses. It remains variadic.

```
q)(+/)til 5           / unary
10
q)(+/)[1000000;til 5] / binary
1000010
```

The potential for confusion is even greater when the argument of a unary operator is a unary function. Here the derived function is unary – but it is still an infix! Parentheses or brackets can save us.

```
q)count'[txt]
4 4
q)(count')txt
4 4
```

Or a keyword.

```
q)count each txt
4 4
```

Conversely, if the unary operator is applied not postfix but with bracket notation, the derived function is *not* an infix. But it can still be variadic.

```
q)'[count]txt             / unary derived function, applied prefix
4 4
q)/[+]til 5               / oops, a comment
q);/[+]til 5              / unary derived function, applied prefix
10
q);\[+][til 5]            / variadic derived function: applied unary
0 1 3 6 10
q);\[+][1000;til 5]       / variadic derived function: applied binary
1000 1001 1003 1006 1010
q)1000/[+]til 5           / but not infix
'type
  [0]  1000/[+]til 5
            ^
```

Applying a unary operator with bracket notation is unusual and discouraged.

Projection

When a value of rank n is applied to m arguments and $m < n$, the result is a *projection* of the value onto the supplied arguments (indexes), now known as the *projected* arguments or indexes.

In the projection, the values of projected arguments (or indexes) are fixed.

The rank of the projection is $n - m$.

```
q)double:2*
q)double 5                      / unary
10
q)halve:%[;2]
q)halve[10]                     / unary
5
q)f:{x+y*z}                     / ternary
q)f[2;3;4]
14
q)g:f[2;;4]
q)g 3                           / unary
14
q)(f . 2 3) 4
14
q)l:("Buddy can you spare";;"?")
q)l "a dime"                    / unary
"Buddy can you spare"
"a dime"
"?"
q)m:("The";;;"fox")
q)m["quick";"brown"]            / binary
"The"
"quick"
"brown"
"fox"
```

The function definition in a projection is set at the time of projection. If the function is subsequently redefined, the projection is unaffected.

```
q)f:{x*y}
q)g:f[3;]   / triple
q)g 5
15
q)f:{x%y}
q)g 5       / still triple
15
```

Make projections explicit

When projecting a function onto an argument list, make the argument list full-length. This is not always necessary but it is good style, because it makes it clear the value is being projected, not applied.

```
q)foo:{x+y+z}
q)goo:foo[2]    / discouraged
q)goo:foo[2;;]  / recommended
```

You could reasonably make an exception for operators and keywords, where the rank is well known.

```
q)f:?["brown"]
q)f "fox"
5 2 5
q)g:like["brown"]
q)g "\*ow\*"
1b
```

When projecting a variadic function the argument list must always be full-length.

Since 4.1t 2021.12.07 projection creation from a lambda/foreign results in a rank error if too many parameters are defined, e.g.

```
q){x}[;1]
'rank
```

Q for Mortals §6.4 Projection
Currying

Applying a list with elided items

A list with elided items can be applied as if it were a function of the same rank as the number of elided items.

```
q)("the";"quick";;"fox")"brown"
"the"
"quick"
"brown"
"fox"

q)("the";"quick";;"fox") @ "brown"
"the"
"quick"
"brown"
"fox"

q)("the";;;"fox") . ("quick";"brown")
"the"
"quick"
"brown"
"fox"
```

This is subject to the same limitation as function notation. If there are more than eight elided items, a rank error is signalled.

Indexing

Indexing a list employs the same syntax as applying a function to arguments and works similarly.

```
q)show m:4 3#.Q.a
"abc"
"def"
"ghi"
"jkl"

q)m[3][1]
"k"

q)m[3;1]
"k"

q)m[3 1;1]
"ke"

q)m[3 1;]       / eliding an index means all positions
"jkl"
"def"

q)m[3 1]        / trailing indexes can be elided
"jkl"
"def"

q)m 3 1         / brackets can be elided for a single index
"jkl"
"def"

q)m @ 3 1       / Index At (top level)
"jkl"
"def"

q)m . 3 1       / Index (at depth)
"k"

q)m . (3 1;1)   / Index (at depth)
"ke"
```

Indexing out of bounds

Indexing a list at a non-existent position returns a null of the type of the first item/s.

```
q)(til 5) 99
0N
q)(`a`b`c!1.414214 2.718282 3.141593) `x
0n
```

```
q)t
name dob        sex
-------------------
dick 1980.05.24 m
jane 1990.09.03 f
q)t 2
name| `
dob | 0Nd
sex | `

q)kt
name city | eye   sex
----------| ---------
Tom  NYC  | green m
Jo   LA   | blue  f
Tom  Lagos| brown m
q)kt `Jack`London
eye|
sex|
```

The thing and the name of the thing

> What's in a name? That which we call a rose
> By any other name would smell as sweet;
> –*Romeo and Juliet*

In all of the above you can use the name of a value (as a symbol) as an alternative.

```
q)f:{x+y*3}
q)f[5;3]              / the rose
14
q)`f[5;3]             / the name of the rose
14
q)`f . 5 3
14
q)g:`f[5;]
q)`g 3
14
```

This applies to values you define in the default or other namespaces. It does not apply to system names, nor to names local to lambdas.

Application and indexing

The conflation of application and indexing is deliberate and useful.

```
q)(sum;dev;var)[1;til 5]
1.414214
```

Above, the list of three keywords is applied to (indexed by) the first argument, selecting `dev`, which is then applied to the second argument, `til 5`.

🕹 *Q for Mortals* §6.8 General Application

Internal functions

The operator ! with a negative integer as left argument calls an internal function.

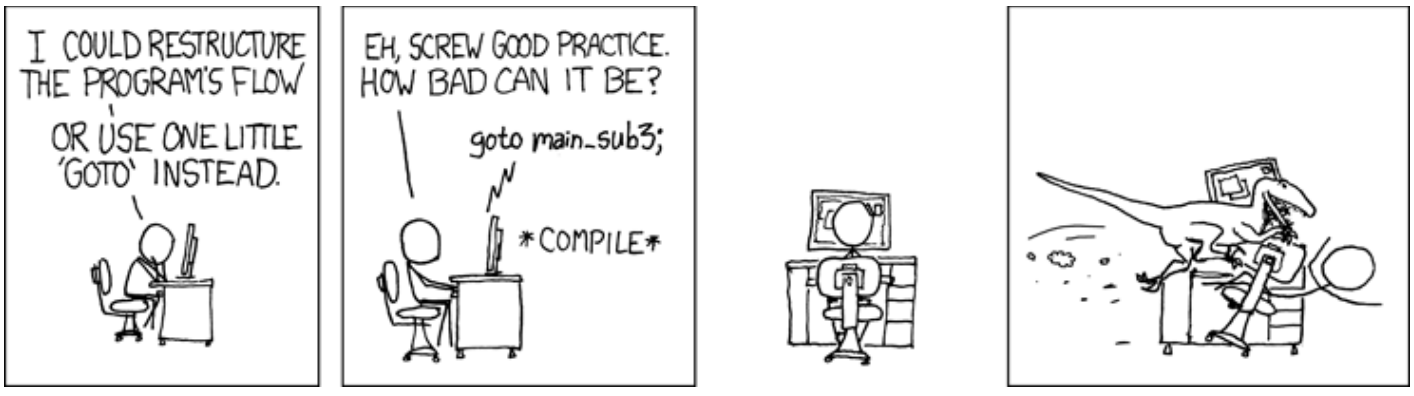
```
0N!x        show                          Replaced:
-4!x        tokens                        -1!   hsym
-8!x        to bytes                      -2!   attr
-9!x        from bytes                    -3!   .Q.s1
-10!x       type enum                     -5!   parse
-11!        streaming execute             -6!   eval
-14!x       quote escape                  -7!   hcount
-16!x       ref count                     -12!  .Q.host
-18!x       compress bytes                -13!  .Q.addr
-21!x       compression/encryption stats  -15!  md5
-22!x       uncompressed length           -19!  set
-23!x       memory map                    -20!  .Q.gc
-25!x       async broadcast               -24!  reval
-26!x       SSL                           -29!  .j.k
-27!(x;y)   format                        -31!  .j.jd
-30!x       deferred response             -32!  .Q.btoa
-33!x       SHA-1 hash                    -34!  .Q.ts
-36!        load master key               -35!  .Q.gz
-38!x       socket table                  -37!  .Q.prf0
-120!x      memory domain
```

Internal functions are for use by language implementors.

They are exposed infrastructure and may be redefined in subsequent releases.

They also allow new language features to be tried on a provisional basis.

Where they are replaced by keywords or utilities, **use the replacements**.



xkcd.com

`0N!x` (show)

The identity function. Returns `x` after writing it to the console.

An essential tool for debugging.

`-4!x` (tokens)

Returns the list of q tokens found in string `x`. (Note the q parsing of names with embedded underscores.)

```
q)-4!"select this from that"
"select"
," "
"this"
," "
"from"
," "
"that"

q)-5!"select this from that" / compare with -5!
?
`that
()
0b
(,`this)!,`this

q)-4!"a variable named aa_bb"
,"a"
," "
"variable"
," "
"named"
," "
"aa_bb"
q)
```

> **Warning**
>
> Should not be used with input data over 2GB in length (0Wi). Returns domain error with this condition since 4.1 2022.04.15.

`-8!x` (to bytes)

Returns the IPC byte representation of `x`.

```
q)-8!1 2 3
0x010000001a000000060003000000010000000200000003000000
```

`-9!x` (from bytes)

`-9!x` (from bytes)

Creates data from IPC byte representation `x`.

```
q)-9!-8!1 2 3
1 2 3
```

`-8!x` (to bytes), `-18!x` (compress bytes)

`-10!x` (type enum)

Resolve a type number to an enum vector and check if it is available.

```
q)-10!20h
1b
q)ee:`a`b`c
q)vv:`ee$`a`a`b
q)type vv
20h
q)-10!20h
0b
```

`-11!` (streaming execute)

Replay the q interpreter on messages stored in a log file.

```
-11!x
-11!(-1;x)
-11!(-2;x)
-11!(n;x)
```

Where `n` is a non-negative integer and `x` is a logfile handle

`-11!x` and `-11!(-1;x)`

replay `x` and return the number of chunks executed; if end of file is corrupted, signal `badtail`.

`-11!(-2;x)`

returns the number of consecutive valid chunks in `x` and the length of the valid part of the file

`-11!(n;x)`

replays `n` chunks from top of logfile and returns the number of chunks executed

Each chunk from a log is passed to `.z.ps` for execution. In replaying, if the logfile references an undefined function, the function name is signalled as an error.

-14!x (quote escape)

Handles " escaping in strings: used to prepare data for CSV export.

-16!x (ref count)

Returns the reference count for a variable.

```
q)-16!a
1
q)a:b:c:d:e:1 2 3
q)-16!a
5
```

-18!x (compress bytes)

Returns the IPC byte representation of `x` according to `-8!x`, applying compression based on IPC compression rules:

- Uncompressed serialized data has a length greater than 2000 bytes
- Size of compressed data is less than ½ the size of uncompressed data

```
q)count -8!til 1000     / uncompressed
8014
q)count -18!til 1000    / compressed
3276
```

-9!x can be used to uncompress and deserialise.

```
q)a:til 1000           / original data to convert
q)x:-18!a              / serialize and compression to bytes using IPC serialisation
q)a~-9!x               / test if deserialised version is same as original
1b
```

`-22!x` (uncompressed length), `-9!x` (from bytes), .Q.gz (gzip)

-21!x (compression/encryption stats)

Where `x` is a file symbol, returns a dictionary of compression/encryption statistics for it. Encryption available since 4.0 2019.12.12. The dictionary is empty if the file is not compressed/encrypted.

```
q)-21!`:ztest       / compressed
compressedLength   | 137349
```

```
uncompressedLength| 80000016
algorithm         | 2i
logicalBlockSize  | 17i
zipLevel          | 6i
q)-21!`:test        / not compressed
q)count -21!`:test
0
q)-21!`:ztest       / encrypted
compressedLength  | 40088
uncompressedLength| 40008
algorithm         | 16i
logicalBlockSize  | 17i
zipLevel          | 6i
```

set
File compression
Data at rest encryption (DARE)

-22!x (uncompressed length)

An optimized shortcut to obtain the length of uncompressed serialized `x`, i.e. `count -8!x`

```
q)v:til 100000
q)\t do[5000;-22!v]
1
q)\t do[5000;count -8!v]
226
q)(-22!v)=count -8!v
1b
```

`-18!x` (compress bytes)

-23!x (memory map)

Since V3.1t 2013.03.04

Attempts to force the object `x` to be resident in memory by hinting to the OS and/or faulting the underlying memory pages. Useful for triggering sequential access to the storage backing `x`.

-25!x (async broadcast)

Since V3.4

Broadcast data as an async msg to specified handles. The advantage of using `-25!(handles;msg)` over `neg[handles]@\:msg` is that `-25!msg` will serialize `msg` just once – thereby reducing CPU and memory load.

Use as

```
q)-25!(handles; msg)
```

Handles should be a vector of positive int or longs.

`msg` will be serialized just once, to the lowest capability of the list of handles. I.e. if handles are connected to a mix of versions of kdb+, it will serialize limited to the types supported by the lowest version. If there is an error, no messages will have been sent, and it will return the handle whose cap caused the error.

Just as with `neg[handles]@\:msg`, `-25!x` queues the msg as async on those handles – they don't get sent until the next spin of the main loop, or are flushed with `neg[handles]@\:(::)`.

> `-25!(handles; ::)` **can also flush the handles**

Possible error scenarios:

- from trying to serialize data for a handle whose remote end does not support a type, or size of the data.

```
/ connect to 2.8 and 3.4
q)h:hopen each 5000 5001
q)h
5 6i
q)(-5) 0Ng / 2.8 does not support guid
'type
q)(-6) 0Ng / 3.4 does support guid
q)-25!(h;0Ng)
'type error serializing for handle 5
```

- an int is passed which is not a handle

```
q)-25!(7 8;0Ng)
'7 is not an ipc handle
```

`-26!x` (SSL)

View TLS settings on a handle or current process `-26!handle` or `-26!()`. Since V3.4 2016.05.12.

```
q)(-26!)[]
SSLEAY_VERSION    | OpenSSL 1.0.2g  1 Mar 2016
SSL_CERT_FILE     | /Users/kdb/certs/server-crt.pem
SSL_CA_CERT_FILE  | /Users/kdb/certs/ca.pem
SSL_CA_CERT_PATH  | /Users/kdb/certs/
SSL_KEY_FILE      | /Users/kdb/certs/server-key.pem
SSL_CIPHER_LIST   | ALL
SSL_VERIFY_CLIENT| NO
SSL_VERIFY_SERVER| YES
```

In the result, all keys except `SSLEAY_VERSION` are initialized from their corresponding environment variables.

`.z.e` TLS connection status
SSL

`-27!(x;y)` (IEEE754 precision format)

Where

- `x` is an int atom
- `y` is a float

returns `y` as a string or strings formatted as a float to `x` decimal places. (Since V3.6 2018.09.26.) It is atomic and doesn't take `\P` into account. For example:

```
q)-27!(3i;0 1+123456789.4567)
"123456789.457"
"123456790.457"
```

This is a more precise, built-in version of `.Q.f` but uses IEEE754 rounding:

```
q).045
0.044999999999999998
q)-27!(2i;.045)
"0.04"
q).Q.f[2;.045]
"0.05"
```

You might want to apply a rounding before applying `-27!`.

`-30!x` (deferred response)

Defer response to a sync message. Since V3.6 2018.05.18.

`-30!(::)`

allows the currently-executing callback to complete without responding to the client, for example `.z.pg`. The handle to use for the subsequent deferred reply can be obtained via `.z.w`. The deferred reply should be provided later via one of the following methods:

`-30!(handle;1b;errorMsg)`

responds to the deferred sync call with an error message populated with the string/symbol provided in `errorMsg`

`-30!(handle;0b;msg)`

responds to the deferred sync call with the contents of `msg`

A `'domain` error will returned if the handle is not a member of `.z.W`.

Using a handle that is not expecting a response message will return an error, for example:

```
q)key .z.W / list of socket handles being monitored by kdb+ main thread
, 8i
q)-30!(8i;0b;`hello`world) / try to send a response of (0b;`hello`world)
'Handle 8 was not expecting a response msg
  [0]  -30!(8i;0b;`hello`world)
         ^
```

🎓 Deferred response

-33!x (SHA-1 hash)

```
-33!x
```

where `x` is a string, returns its SHA-1 hash as a list of strings of hex codes.

```
q)raze string -33!"mypassword"
"91dfd9ddb4198affc5c194cd8ce6d338fde470e2"
```

📖 Command-line options `-u` and `-U`

-36! Load master key

```
-36!(::)    / since 4.1 2024.03.12 and 4.0 2024.03.02
-36!(x;y)
-36!(x;y;z) / since 4.1 2024.03.12 and 4.0 2024.03.02
```

Where

- `x` is a master-key file as a file symbol
- `y` is a password as a string
- `z` is whether to unlock/lock as a bool

`-36!(::)`

Expose whether a key has already been loaded, returning 0b or 1b accordingly.

`-36!(x;y) and -36!(x;y;z)`

loads and validates the master key into memory as the key to use when decrypting or encrypting data on disk.

🎓 Create master key

Expect this call to take about 500 milliseconds to execute. It can be executed from handle 0 only.

Signals errors:

```
Encryption lib unavailable      failed to load OpenSSL libs
Invalid password
Main thread only                can be executed from the main thread only
PKCS5_PBKDF2_HMAC               library invocation failed
Restricted                      must be executed under handle 0
Unrecognized key format         master key file format unrecognized
```

`z` indicates unlock/lock. To reload using a new key, unlock using current key and then proceed with the new key. If load is attempted while locked, it throws `'DARE key locked`.

`-38!x` (socket table)

```
-38!x
```

where `x` is a list of socket handles, returns a table with columns

- `p` (protocol): `q` (IPC) or `w` (WebSocket)
- `f` (family): `t` (TCP) or `u` (Unix domain socket)
- `z` (compression enabled flag): since v4.1 2024.05.31
- `n` (count unsent msgs): since v4.1 2024.05.31
- `m` (total unsent bytes, like .z.W): since v4.1 2024.05.31

Since v4.0 2020.06.01.

```
q)h:hopen 5000
q)-38!h
p| "q"
f| "t"
z| 0b
n| 0
m| 0
q){([]h)!-38!h:.z.H}[]
h| p f z n m
-| ---------
8| q u 0 0 0
9| q t 0 0 0
```

`.z.H` active sockets, `.z.W` handles, `.z.w` handle

`-120!x` (memory domain)

```
-120!x
```

returns `x`'s memory domain (currently 0 or 1), e.g.

```
q)-120!'(1 2 3;.m.x:1 2 3)
0 1
```

.m namespace

Iteration

The primary means of iteration in q are

- implicit in its operators and keywords
- the **map iterator** Each and its variants distribute evaluation through data structures
- the **accumulating iterators** Scan and Over control successive iterations, where the result of one evaluation becomes an argument to the next
- the control words `do` and `while`

Implicit iteration

Most operators and keywords have iteration built into them.

> ⚠ **A common beginner error is to specify iteration where it is already implicit**

Iterators

The iterators are unary operators. They take values as arguments and derive functions that apply them repeatedly.

> **Value**
>
> An applicable value is a q object that can be indexed or applied to one or more arguments:
>
> - function: operator, keyword, lambda, or derived function
> - list: vector, mixed list, matrix, or table
> - dictionary
> - file- or process handle

The iterators can be applied postfix, and almost always are. For example, the Over iterator `/` applied to the Add operator `+` derives the function `+/` , which reduces a list by summing it.

```
q)(+/)2 3 4 5
14
```

There are two groups of iterators: maps and accumulators.

Maps

The maps – Each, Each Left, Each Right, Each Prior, and Each Parallel – apply a map to each item of a list or dictionary.

```
q)count "zero"                          / count the chars (items) in a string
4
q)(count')("The";"quick";"brown";"fox")    / count each string
3 5 5 3
```

Accumulators

The accumulators – Scan and Over – apply a value successively, first to the argument, then to the results of successive applications.

Control words

The control words `do`, and `while` also enable iteration, but are rarely required.

Do as little as possible

First see if the iteration you want is already implicit in the operators and keywords.

If not, use the map and accumulator iterators to specify the iteration you need.

If you find yourself using the `do` or `while` control words, you probably missed something.

> "I'll say no more than necessary. If that."
> – 'Chili' Palmer in *Get Shorty*.

Implicit iteration

Implicit iteration

> ⚠ **Before you specify iteration, see whether what you need is already implicit in the operators and keywords**

🎞 This tutorial as a video presentation

Lists and dictionaries are first-class entities in q, and most operators and keywords iterate through them. This article is about when to *leave it to q*.

That is, when *not* to specify iteration.

Recall:

Map iteration

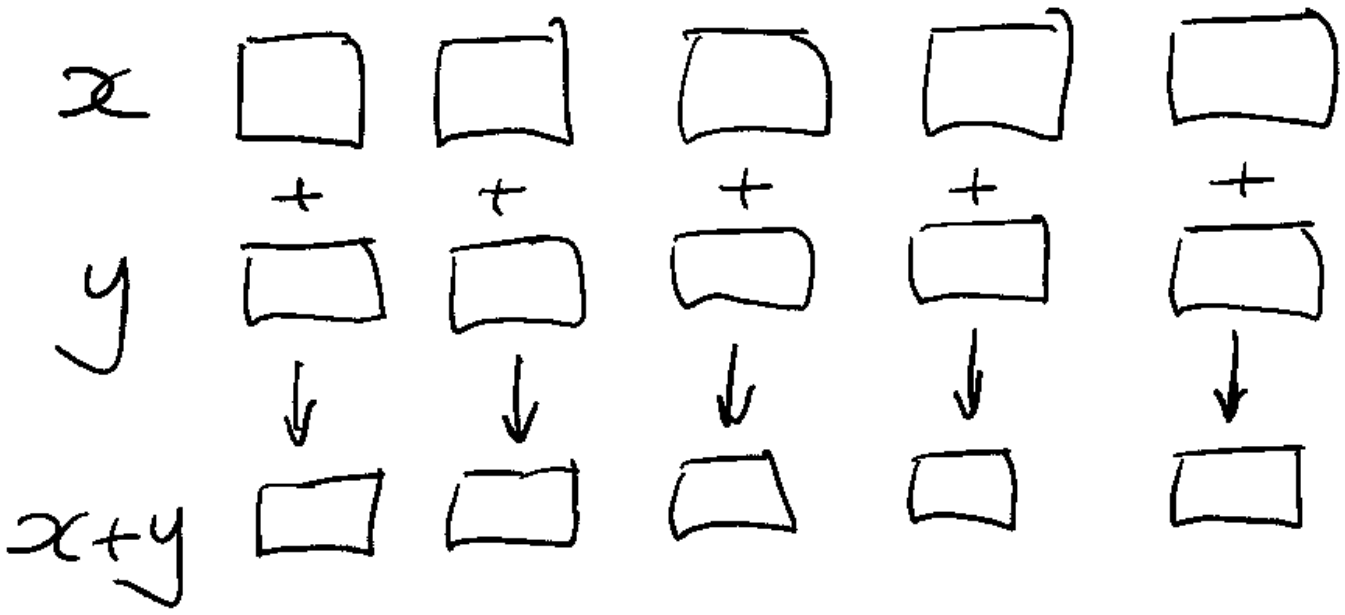
> evaluates an expression *once* on each item in a list or dictionary.

Accumulator iteration

> evaluates an expression *successively*: the result of one evaluation becomes an argument of the next.

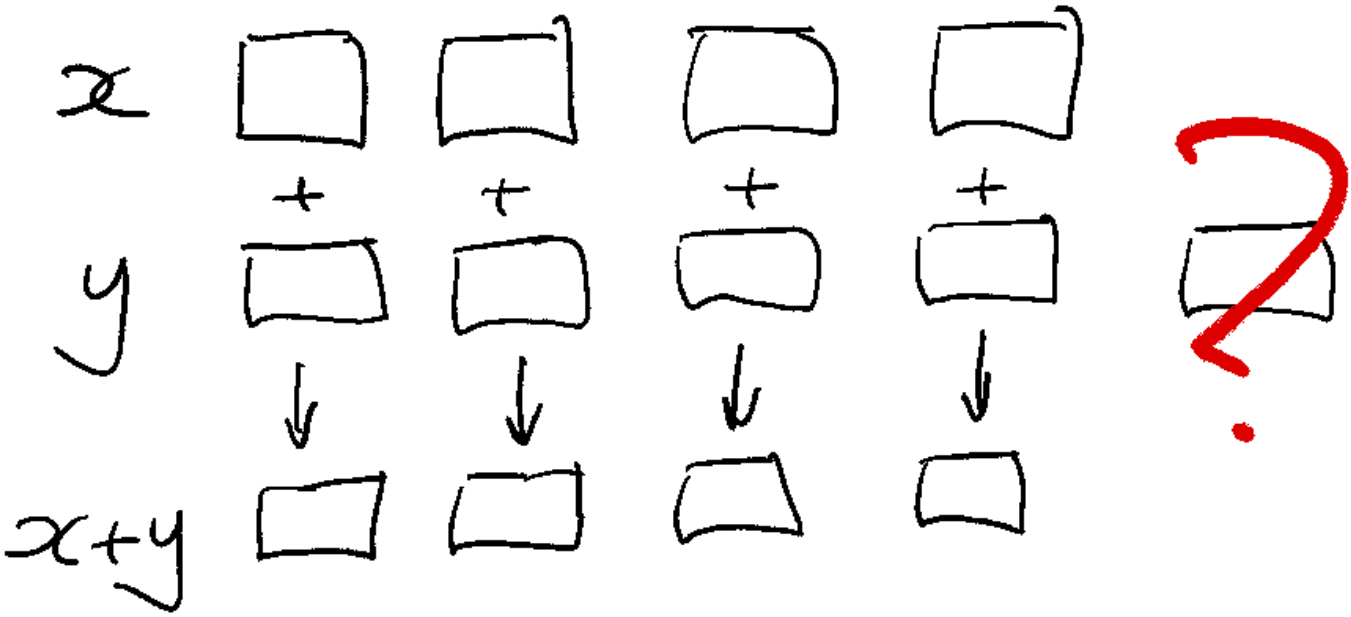
Implicit map iterations

The simplest and most common implicit map iteration is *pairwise*: between corresponding list items.



```
q)10 100 1000 * (1 2 3;4 5 6;7 8)
10 20 30
400 500 600
7000 8000
```

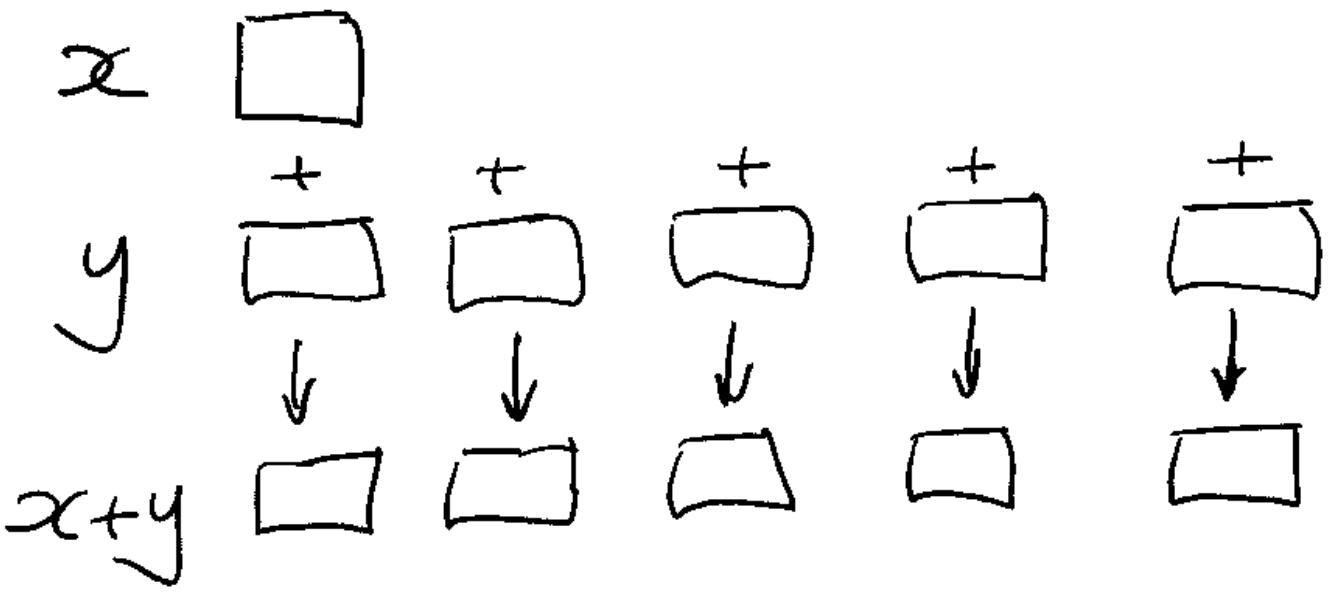
Of course, this requires the lists to have the same number of items.




```
q)10 100 1000 * (1 2 3;4 5 6)
'length
  [0]  10 100 1000 * (1 2 3;4 5 6)
                     ^
```

Scalar extension

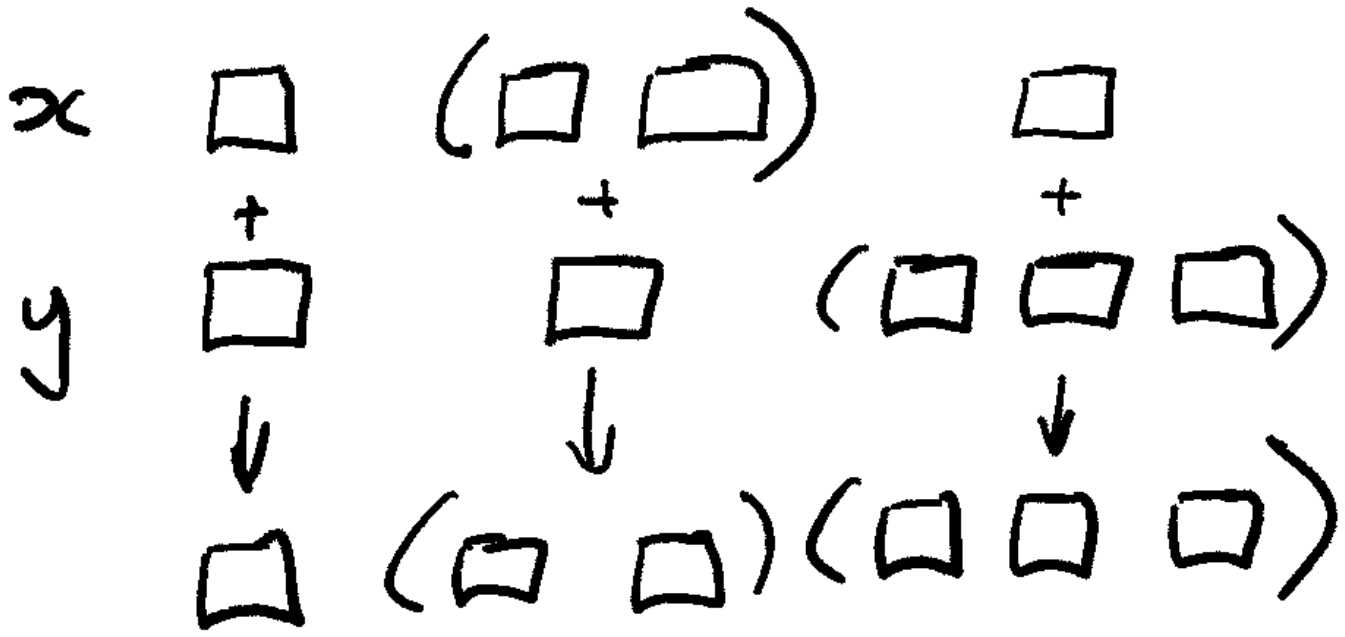
Unless! If one of the operands is an atom, *scalar extension* pairs it with every list item.




```
q)5 < 1 2 3 4 5 6 7 8
00000111b
q)"f" < ("abc";"def";"gh")
000b
000b
11b
```

Atomic iteration

Many operators have *atomic iteration*: they iterate recursively, pairwise and with scalar extension, until they find the atoms in a list.



```
q)1 4 7 < (1 2 3;4 5 6;7 8)
011b
011b
01b

q)(1;2 3 4; 7) < (1 2 3;4 5 6;7 8)
011b
111b
01b

q)(1;2 3 4;(5 6 7;8)) < (1 2 3;4 5 6;7 8)
011b
111b
(110b;0b)
```

Similarly, some unary keywords implicitly apply to each item of a list argument – and recurse to atoms.

```
q)cos (1 2 3; 4 5 6)
0.5403023  -0.4161468 -0.9899925
-0.6536436 0.2836622  0.9601703

q)lower("THE";("Quick";"Brown");"FOX")
"the"
("quick";"brown")
"fox"
```

Atomic operators are atomic in both their left and right domains.

```
4 < (1;2 3 4;(5 6 7;8))
0b
```

```
000b
(111b;1b)
```

Some binary keywords are atomic in only one domain. For example, the right argument of `within` is an ascending pair of sortable type. But in its left domain, `within` is atomic.

```
q)2 3 4 within 3 6
011b
q)(2 3 4;(5; 6 7;8)) within 3 6
0  1   1
1b 10b 0b
```

List iteration

List iteration is through list items only – not atomic. The `like` keyword has list iteration in its left domain.

```
q)`quick like "qu?ck"
1b
q)`quick`quack`quark like "qu?ck"       / list iteration
110b
q)(`quick;`quack`quark) like "qu?ck"    / but not atomic
'type
  [0]  (`quick;`quack`quark) like "qu?ck"
                             ^
```

List iteration stops after the first level: it does not recurse.

Simple visualizations

Even a simple visual display can be useful. Here are sines of the first twenty positive integers, tested to see which of them is greater than 0.5.

```
q).5 < sin 1 + til 20
11000011000001100001b
```

We can take that boolean vector and use it to index a short string, getting us a simple visual display. And, as you probably know, Index At `@` can be elided and replaced with prefix notation.

```
q)".#" @ .5 < sin 1 + til 20
"##....##.....##....#"
q)".#" .5 < sin 1 + til 20
"##....##.....##....#"
```

Index At is atomic in its right domain; that is, *right-atomic*.

Here we'll index a string with an integer vector and we'll get a string result.

```
q)" -|+" @ 0 3 1 1 1 3 0
" +---+ "
```

If we index it with a 2-row matrix – two integer vectors – we'll get a character matrix back.

```
q)" -|+" @ (0 3 1 1 1 3 0;0 2 0 0 0 2 0)
" +---+ "
" |   | "
```

And if we take that 2-row matrix and index it – to make selections from it – the result is a numeric matrix.

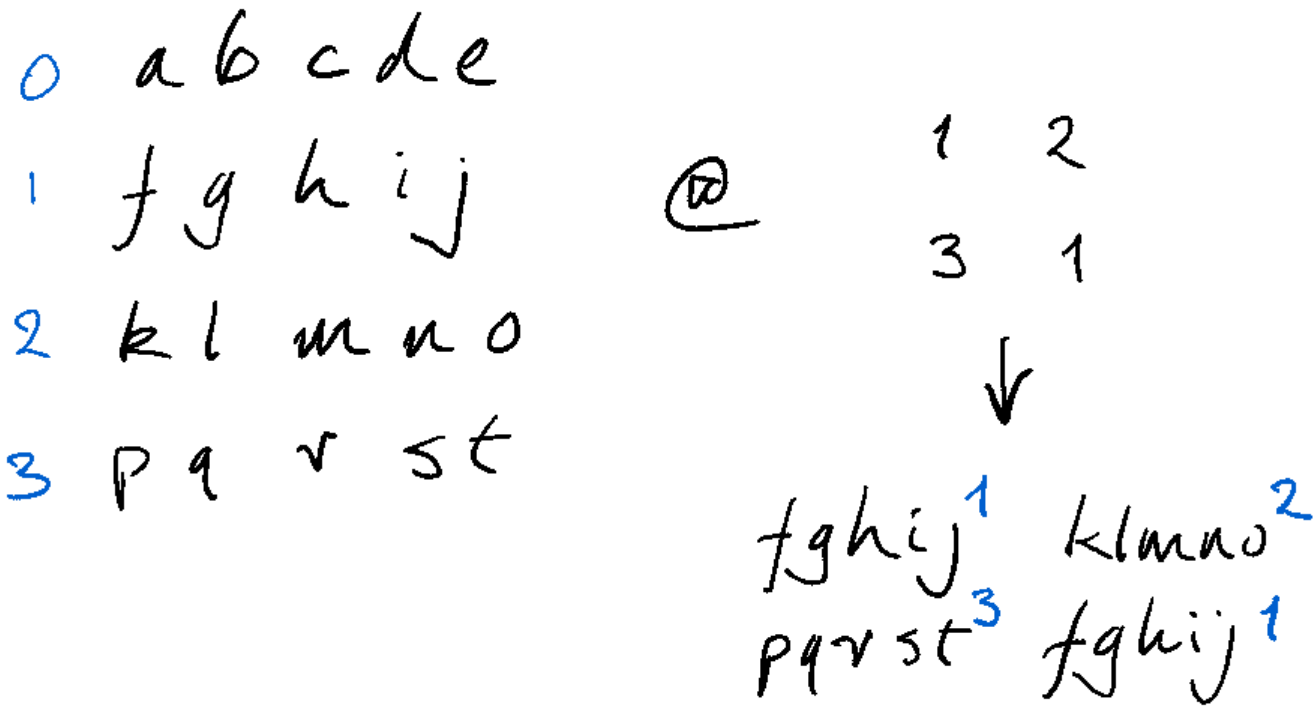
```
q)(0 3 1 1 1 3 0;0 2 0 0 0 2 0) @ 0 1 1 1 0
0 3 1 1 1 3 0
0 2 0 0 0 2 0
0 2 0 0 0 2 0
0 2 0 0 0 2 0
0 3 1 1 1 3 0
```

And because Index At is right-atomic we can use the numeric matrix to index the string.

```
q)" -|+" @(0 3 1 1 1 3 0;0 2 0 0 0 2 0) @ 0 1 1 1 0
" +---+ "
" |   | "
" |   | "
" |   | "
" +---+ "
```

Index At is right-atomic, but in its left domain it has list iteration: list items need not be atoms. In this example, the list items are themselves strings. If we index that list of strings with an integer matrix, we get back a matrix of strings.

```
q)show L:("the";"quick";"brown";"fox")
"the"
"quick"
"brown"
"fox"
q)(1 3;2 0)
1 3
2 0
q)L@(1 3;2 0)
"quick" "fox"
"brown" "the"
```




```
q)show q:4 5#.Q.a
"abcde"
"fghij"
"klmno"
"pqrst"

q)q @ (1 2;3 1)  / Index At: right-atomic
"fghij" "klmno"
"pqrst" "fghij"

q)q . (1 2;3 1)  / Index: list iteration on the right
"ig"
"nl"
```

Some keywords evaluate a binary expression between adjacent items in a list.

```
q)deltas 1 5 0 9 5 2
1 4 -5 9 -4 -3
q)ratios 2 3 4 5
2 1.5 1.333333 1.25
```

These are map iterations: the evaluations are independent and can be performed in parallel.

Exercise 1

`sensors.txt` contains (24) hourly sensor readings over a 12-day period. Sensor readings are in the range 0-9.

```
$ wget https://code.kx.com/download/learn/iteration/sensors.txt
--2022-01-03 11:27:18--  https://code.kx.com/download/learn/iteration/sensors.txt
```

```
Resolving code.kx.com (code.kx.com)... 74.50.49.235
Connecting to code.kx.com (code.kx.com)|74.50.49.235|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 300 [text/plain]
Saving to: ‘sensors.txt’

sensors.txt         100%[===================>]     300  --.-KB/s    in 0s

2022-01-03 11:27:19 (143 MB/s) - ‘sensors.txt’ saved [300/300]
```

```
q)show s:read0`:sensors.txt
"0305572462511572657360 86"
"7572511099999932701883 77"
"7764394486251268963475 68"
"1164911581371375890311 87"
"8559387995416992629466 23"
"1049488061868670579360 25"
"3289644798586964849450 53"
"8615961029999337291456 53"
"6235890721024304975787 80"
"2406634399999977462466 72"
"3115515724142723840052 63"
"8508840464572142322007 14"
```

a. For each of the 24 hours, on how many days of the period did the sensor reading for that hour fall to zero?

Converting the sensor readings to numbers is not necessary: they can be compared directly to `"0"`.

```
q)s="0"
10100000000000000000100b
00000001000000001000000b
00000000000000000000000b
00000000000000000100000b
00000000000000000000000b
01000001000000100000100b
00000000000000000000100b
00000001000000000000000b
00000010001000100000001b
00100000000000000000000b
00000000000000000110000b
00100010000000000011000b
```

The Equals operator has implicit atomic iteration. Here it iterates across the items (rows) of the list `s`. Each item (row) is a character list (string) and Equals continues iterating through the items.

The result of `s="0"` is a boolean matrix of the same shape as `s`. Summing it simply adds the rows together.

```
q)sum s="0"
1 1 3 0 0 0 2 3 0 0 1 0 0 0 1 1 0 1 2 2 1 3 0 1i
```

Your maintenance manager gets automated reports printed, but the last report got damaged. She needs your help.

b. On which days did the sensor readings begin (8, 6, 1, 5, ...) and (1, 1, 6, 4, ...)?

We can search the first four columns of `s` for these sequences.

```
q)s[;til 4]
"0305"
"7572"
"7764"
"1164"
"8559"
"1049"
"3289"
"8615"
"6235"
"2406"
"3115"
"8508"
```

The Find operator has list iteration in both left and right domains.

```
q)s[;til 4]?("8615";"1164")
7 3
```

Visualizations help us find patterns in datasets. Even simple visualizations can be valuable.

Normal operating levels are in the range (2,7).

c. Display a simple plot showing when the sensors reported levels outside that range.

The `within` keyword take as right argument a 2-item vector of sortable type. It has atomic iteration in its left domain. Keyword `not` is atomic.

```
q)not s within "27"
1010000000011000000000110b
0000011111111100011111000b
0000010010001001100000001b
1100111011001000111011110b
1001010110010110001000000b
1101011101101001001001000b
0011000011010100101001000b
1010101101111000011000000b
0000111001100010100010011b
0010000011111110000000000b
0110010000100000101100000b
1011101000000100000011010b
```

Because Index At is right-atomic we can use the boolean matrix to index a string.

```
".#"not s within "27"
"#.#........##........##."
".....#########...####..."
".....#..#...#..##......#"
"##..###.##..#...###.###."
"#..#.#.##..#.##...#....."
"##.#.###.##.#..#..#..#.."
"..##....##.#.#..#.#..#.."
"#.#.#.##.####....##....."
"....###..##...#.#...#.##"
"..#.....######.........."
".##..#....#.....#.##...."
"#.###.#......#.....##.#."
```

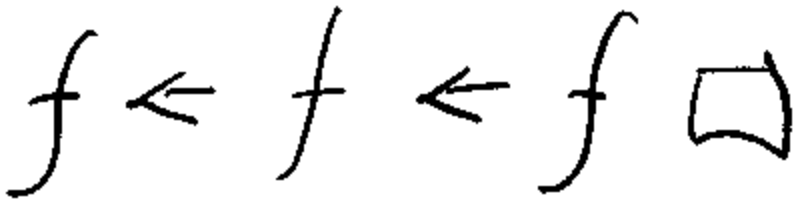
At level 9 productivity is highest.

d. Plot when in the period this occurred.

```
".#"s="9"
"........................"
"........######.........."
".....#..........#......."
"....#............#......"
"...#...##....##...#....."
"...#..............#....."
"...#....#....#....#....."
"....#....####....#......"
".....#..........#......."
"........######.........."
"........................"
"........................"
```

Implicit accumulator iterations

Accumulator iterations evaluate some expression *successively*: the result of one evaluation becomes the argument of the next.



We have already used the `sum` keyword, which implicitly evaluates Add between successive items of a list.

```
q)((2+3)+4)+5
14
q)sum 2 3 4 5
14

q)a:`cats`dogs!2 3; b:`cows`sheep!3 4; c:`dogs`sheep!5 6
q)sum (a;b;c)
cats | 2
dogs | 8
cows | 3
sheep| 10
```

`sum` is an *aggregator*: it returns the result of its *last* evaluation. `sums` also iterates successively, but returns the results of *all* the evaluations.

```
q)(2;2+3;2+3+4;2+3+4+5)
2 5 9 14
q)sums 2 3 4 5
2 5 9 14
```

Notice that the result has the same length as the argument: `sums` is a *uniform* function. Notice also that the index of the result corresponds to the number of evaluations: `(sums 2 3 4 5)[3]` is the result of three additions and `(sums 2 3 4 5)[0]` is the result of no additions.

Keywords such as `mavg` and `msum` combine map iterations (e.g. evaluate on each group of three successive items) with an aggregator which might employ accumulator iteration, e.g. `sum` .

```
q)3 msum 1 5 0 9 5 2 2 4 0 5 3 0
1 6 6 14 14 16 9 8 6 9 8 8
```

Exercise 2

Factory productivity is thought to be most affected by the machinery's fuddling level. An automated process adjusts the fuddling level every 20 minutes to keep it stable; the level resets to zero each midnight.

We have in `fudadj.csv` a log of the adjustments.

```
q)\wget -q https://code.kx.com/download/learn/iteration/fudadj.csv

q)read0 `:fudadj.csv / fuddling adjustments
"-1,-1,3,3,2,3,3,3,1,-1,3,0,2,1,2,1,0,-1,3,0,3,1,1,1,3,0,-1,3,-1,2,0,2,1,3,0,0,0,..
"0,1,-1,-1,3,-1,-1,3,1,1,2,1,-1,1,3,2,2,3,2,2,2,3,3,3,2,3,0,3,3,1,2,1,-1,-1,-1,0,..
"1,0,-1,2,3,-1,1,-1,-1,-1,2,3,2,0,0,3,3,2,2,-1,2,-1,2,0,1,2,2,0,0,-1,1,3,-1,1,-1,..
"3,2,2,1,3,-1,-1,-1,1,-1,1,1,0,-1,0,3,-1,0,2,0,2,0,1,2,3,2,1,3,-1,2,-1,1,2,1,-1,3..
"1,0,3,-1,2,3,3,1,1,2,-1,1,1,3,-1,2,2,2,2,2,0,3,-1,1,2,-1,3,0,0,1,2,3,3,0,-1,0,-1..
"2,-1,3,2,1,2,3,3,1,2,-1,-1,1,-1,0,-1,3,2,-1,-1,-1,1,1,2,2,3,0,2,1,0,1,2,3,3,2,-1..
"3,1,-1,2,1,3,-1,1,0,1,2,2,1,3,1,1,1,3,2,-1,-1,1,0,3,3,0,0,2,1,0,2,3,2,2,2,0,-1,-..
"-1,2,-1,-1,1,2,-1,0,2,3,0,2,0,1,2,-1,3,3,1,2,-1,-1,-1,3,3,0,1,1,1,3,2,1,-1,1,2,2..
"-1,3,-1,2,0,0,1,1,1,3,0,2,2,2,2,-1,-1,-1,-1,1,1,3,0,3,-1,1,2,3,0,-1,2,2,2,2,0,2,..
"3,2,-1,-1,0,-1,3,2,0,3,1,0,0,2,3,2,1,1,-1,2,3,-1,3,3,3,-1,1,3,2,1,1,1,2,3,2,1,1,..
"0,2,0,1,-1,3,0,2,-1,2,-1,2,0,0,-1,3,0,3,1,0,2,2,3,-1,2,0,1,1,2,0,2,2,0,0,0,-1,1,..
"2,-1,2,-1,3,0,1,1,0,-1,2,2,3,3,0,0,-1,1,3,-1,1,2,2,3,2,-1,0,2,0,3,0,1,1,0,3,3,-1..
```

What were the fuddling levels corresponding to the sensor readings in Exercise 1?

The file has no column headers, so Load CSV returns not a table but a list of columns.

```
q)show fa:(prd[24 3]#"J";csv)0: read0 `:fudadj.csv / fuddling adjustments
-1 0  1  3  1  2  3  -1 -1 3  0  2
-1 1  0  2  0  -1 1  2  3  2  2  -1
3  -1 -1 2  3  3  -1 -1 -1 -1 0  2
3  -1 2  1  -1 2  2  -1 2  -1 1  -1
2  3  3  3  2  1  1  1  0  0  -1 3
..
```

That suits us. The 72 rows correspond to 20-minute intervals. We take cumulative sums across the intervals, and select every third sum to get the hourly levels. Transposing the result gives us 12×24 fuddling levels.

```
q)flip sums[fa]@2+3*til 24
1 9  16 18 23 23 29 32 34 38 41 44 44 50 52 54 59 62 65 72 72 76 78 80
0 1  4  8  11 18 24 33 38 45 47 45 50 51 54 55 58 58 57 61 64 68 69 66
0 4  3  7  9  17 20 21 26 25 28 27 28 33 34 34 36 42 46 49 51 55 55 61
7 10 9  10 9  11 15 18 24 28 30 33 35 41 41 43 43 49 51 56 57 57 62 63
4 8  13 15 18 24 28 31 35 36 44 43 45 47 49 55 58 60 64 62 65 67 68 70
4 9  16 16 16 20 17 21 26 29 35 39 42 49 57 59 65 69 73 73 75 74 74 77
3 9  9  14 19 24 24 28 31 34 41 45 46 49 55 57 59 62 68 71 75 77 79 79
0 2  3  8  11 16 18 19 23 28 30 35 36 37 41 41 42 47 54 59 58 59 63 67
1 3  6  11 17 14 15 21 23 25 31 35 41 46 47 51 54 59 63 67 73 77 81 86
4 2  7  11 16 20 24 29 32 38 42 48 51 56 60 67 64 72 75 78 79 82 85 91
2 5  6  9  8  14 17 21 24 27 31 30 32 36 38 39 41 44 42 47 50 50 51 53
3 5  7  10 16 16 19 26 27 32 34 40 39 40 43 51 51 53 57 62 65 63 65 71
```

It is clear that the automatic adjustments are not keeping the fuddling levels stable.

Yet another way q is weird?

If in other languages you are used to specifying iterations, you may at first experience this as an annoying distraction. Besides solving your problem, you also have to learn and keep in mind q's implicit iterations. You already know how to write iterations. Why now learn this?

The reward is that, as implicit iteration becomes familiar to you, you stop thinking about most of the iterations in your code, which leaves you more mental space for problem solving. (Only when we put on noise-cancelling headphones do we discover how much annoying background noise we had been filtering out.)

As a bonus, many algorithms are startlingly simple to write in q. It's way cool.

Conclusion

That's it. The big takeaway is that there is a *lot* of iteration built into the q primitives. It will almost always give you your shortest, fastest code – and the most readable.

Iterators

```
--------- maps --------     --------- accumulators ----------
'  Each           each      / Over  over  Converge, Do, While
': Each Parallel  peach     \ Scan  scan  Converge, Do, While
': Each Prior     prior
\: Each Left
/: Each Right
'  Case
```

The iterators (once known as *adverbs*) are native higher-order operators: they take applicable values as arguments and return derived functions. They are the primary means of iterating in q.

📖 Iteration in q
🕮 Iterators

> **Applicable value**
>
> An applicable value is a q object that can be indexed or applied to arguments: a function (operator, keyword, lambda, or derived function), a list (vector, mixed list, matrix, or table), a file- or process handle, or a dictionary.

For example, the iterator Over (written `/`) uses a value to reduce a list or dictionary.

```
q)+/[2 3 4]      /reduce 2 3 4 with +
9
q)*/[2 3 4]      /reduce 2 3 4 with *
24
```

Over is applied here postfix, with `+` as its argument. The derived function `+/` returns the sum of a list; `*/` returns its product. (Compare *map-reduce* in some other languages.)

Variadic syntax

Each Prior, Over, and Scan applied to binary values derive functions with both unary and binary forms.

```
q)+/[2 3 4]           / unary
9
q)+/[1000000;2 3 4]   / binary
1000009
```

📖 Variadic syntax

Postfix application

Like all functions, the iterators can be applied with Apply or with bracket notation. But unlike any other functions, they can also be applied postfix. They almost always are.

```
q)'[count][("The";"quick";"brown";"fox")]   / ' applied with brackets
3 5 5 3
q)count'[("The";"quick";"brown";"fox")]     / ' applied postfix
3 5 5 3
```

Only iterators can be applied postfix.

> **Regardless of its rank, a function derived by postfix application is always an infix.**

To apply an infix derived function in any way besides infix, you can use bracket notation, as you can with any function.

```
q)1000000+/2 3 4       / variadic function applied infix
1000009
q)+/[100000;2 3 4]     / variadic function applied binary with brackets
1000009
q)+/[2 3 4]            / variadic function applied unary with brackets
9
q)txt:("the";"quick";"brown";"fox")
q)count'[txt]          / unary function applied with brackets
3 5 5 4
```

If the derived function is unary or variadic, you can also parenthesize it and apply it prefix.

```
q)(count')txt          / unary function applied prefix
3 5 5 4
q)(+/)2 3 4            / variadic function applied prefix
9
```

Glyphs

Six glyphs are used to denote iterators. Some are overloaded.

Iterators

- in bold type derive **uniform** functions;
- in italic type, *variadic* functions.

Subscripts indicate the rank of the value; superscripts, the rank of the *derived function*. (Ranks 4-8 follow the same rule as rank 3.)

glyph	iterator/s
'	$_1$ **Case**; **Each**
\:	$_2$ **Each Left** [2]
/:	$_2$ **Each Right** [2]
':	$_1$ **Each Parallel** [1] ; $_2$ *Each Prior* [1 2]
/	$_1$ Converge [1] ; $_1$ Do [2] ; $_1$ While [2] ; $_2$ *Reduce* [1 2] ; $_3$ Reduce [3]
\	$_1$ Converge [1] ; $_1$ Do [2] ; $_1$ While [2] ; $_2$ ***Accumulate*** [1 2] ; $_3$ **Accumulate** [3]

Over and Scan, with values of rank >2, derive functions of the same rank as the value.

The overloads are resolved according to the following table of syntactic forms.

Two groups of iterators

There are two kinds of iterators: *maps* and *accumulators*.

Maps

> distribute the application of their values across the items of a list or dictionary. They are implicitly *parallel*.

Accumulators

> apply their values *successively*: first to the entire (left) argument, then to the result of that evaluation, and so on. With values of rank ≥2 they correspond to forms of *map reduce* and *fold* in other languages.

Application

A derived function, like any function, can be applied by **bracket notation**. Binary derived functions can also be applied **infix**. Unary derived functions can also be applied **prefix**. Some derived functions are **variadic** and can be applied as either unary or binary functions.

This gives rise to multiple equivalent forms, tabulated here. Any function can be applied with bracket notation or with Apply. So to simplify, such forms are omitted here in favour of prefix or infix application. For example, `u'[x]` and `@[u';x]` are valid, but only `(u')x` is shown here. (Iterators are applied here postfix only.)

The mnemonic keywords `each`, `over`, `peach`, `prior` and `scan` are also shown.

value rank	syntax	name	semantics

1
2
3+
	`(u')x`, `u each x`
	`x b'y`
	`v'[x;y;z;…]`
		Each	apply `u` to each item of `x`
			apply `g` to corresponding items of `x` and `y`
			apply `v` to corresponding items of `x`, `y`, `z` …

2	`x b\:d`	Each Left	apply `b` to `d` and items of `x`
2	`d b/:y`	Each Right	apply `b` to `d` and items of `y`
1	`(u':)x`, `u peach x`	Each Parallel	apply `u` to items of `x` in parallel tasks
2	`(b':)y`, `b prior y`, `d b':y`	Each Prior	apply `b` to (`d` and) successive pairs of items of `y`
1	`int'[x;y;…]`	Case	select from `[x;y;…]`
1	`(u/)d`, `(u\)d`	Converge	apply `u` to `d` until result converges
1	`n u/d`, `n u\d`	Do	apply `u` to `d`, `n` times
1	`t u/d`, `t u\d`	While	apply `u` to `d` until `t` of result is 0

1
2
3+
	`(b/)y`, `b over y`
	`d b/y`
	`vv/[d;y;z;…]`
		Over	reduce a list or lists

1
2
3+
	`(g\)y`, `g scan y`
	`d g\y`
	`vv\[d;y;z;…]`
		Scan	scan a list or lists

Key:

```
d:   data
int: int vector        n: int atom ≥0
v:   value             t: test value
u:   unary value       y: list
b:   binary value      x: list
```

Map iterators

```
map                      rank
 Each           v'       same as v    each
 Each Left      v2\:        2
 Each Right     v2/:        2
 Each Parallel  v1':        1         peach
 Each Prior     v2':     variadic     prior
 Case           i'        1+max i
```

```
v1: value (rank 1)         v: value (rank 1-8)
v2: value (rank 2)         i: vector of ints≥0
```

The maps are iterators that derive **uniform** functions that apply their values once to each item of a dictionary, a list, or conforming lists.

Each

Apply a value item-wise to a dictionary, list, or conforming lists and/or dictionaries.

```
(v1')x    v1'[x]       v1 each x
x v2'y    v2'[x;y]
          v3'[x;y;z]
```

Where `v` is an applicable value, `v'` applies `v` to each item of a list, dictionary or to corresponding items of conforming lists. The derived function has the same rank as `v`.

```
q)(count')`a`b`c!(1 2 3;4 5;6 7 8 9)       / unary
a| 3
b| 2
c| 4
```

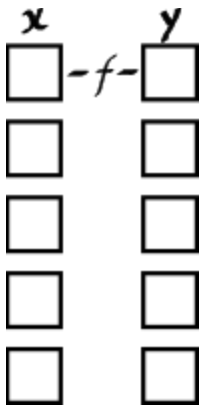
Each applied to a binary value is sometimes called *each both* and can be applied infix.

```
q)1 2 3 in'(1 0 1;til 100;5 6 7)  / in' is binary, infix
110b
```

Iterations of ternary and higher-rank values are applied with brackets.

```
q){x+y*z}'[1000000;1 0 1;5000 6000 7000]    / ternary
1005000 1000000 1007000
```

⚠ **Each is redundant with atomic functions. (Common qbie mistake.)**



Each Both

`each` keyword

The mnemonic keyword `each` can be used to apply a unary value without parentheses or brackets.

```
q)(count')string `Clash`Fixx`The`Who
5 4 3 3
q)count'[string `Clash`Fixx`The`Who]
5 4 3 3
q)count each string `Clash`Fixx`The`Who
5 4 3 3
```

Each Left and Each Right

Apply a binary value between one argument and each item of the other.

```
Each Left     x v2\: y    v2\:[x;y]   |->   v2[;y] each x
Each Right    x v2/: y    v2/:[x;y]   |->   v2[x;] each y
```

The maps Each Left and Each Right take **binary** values and derive binary functions that pair one argument to each item of the other. Effectively, the map projects its value on one argument and applies Each.

	Each Left	Each Right
syntax:	`x f\:y`	`x f/:y`
equivalent:	`f[;y] each x`	`f[x;] each y`
		


```
q)"abcde",\:"XY"            / Each Left
"aXY"
"bXY"
```

```
"cXY"
"dXY"
"eXY"
q)"abcde",/:"XY"              / Each Right
"abcdeX"
"abcdeY"
q)m                           / binary map
"abcd"
"efgh"
"ijkl"
q)m[0 1;2 3] ~ 0 1 m\:2 3
1b
q)0 1 m/:2 3
"cg"
"dh"
q)(flip m[0 1;2 3]) ~ 0 1 m/:2 3
1b
```

Left, right, `cross`

Each Left combined with Each Right resembles the result obtained by `cross`.

```
q)show a:{x,/:\:x}til 3
0 0 0 1 0 2
1 0 1 1 1 2
2 0 2 1 2 2
q)show b:{x cross x}til 3
0 0
0 1
0 2
1 0
1 1
1 2
2 0
2 1
2 2
q){}0N!a
((0 0;0 1;0 2);(1 0;1 1;1 2);(2 0;2 1;2 2))
q){}0N!b
(0 0;0 1;0 2;1 0;1 1;1 2;2 0;2 1;2 2)
q)raze[a] ~ b
1b
```

Atoms and lists in the domains of these iterators

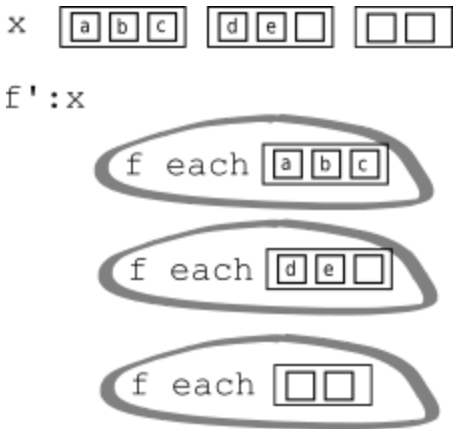
The domains of `\:` and `/:` extend beyond binary values to include certain atoms and lists.

```
q)(", "/:)("quick";"brown";"foxes")
"quick, brown, foxes"
q)(0x0\:)3.14156
0x400921ea35935fc4
```

This is exposed infrastructure. Use the keywords `vs` and `sv` instead.

Each Parallel

Assign sublists of the argument list to secondary tasks, in which the unary value is applied to each item of the sublist.




```
(v1':)x   v1':[x]   v1 peach x
```

The Each Parallel map takes a **unary** value as argument and derives a unary function. The iteration `v1':` divides its list or dictionary argument `x` between available secondary tasks. Each secondary task applies `v1` to each item of its sublist.

📖 Command-line option `-s`, Parallel processing

```
q) q -s 2
KDB+ 4.1t 2021.07.12 Copyright (C) 1993-2021 Kx Systems
m64/ 12()core 65536MB sjt mackenzie.local 127.0.0.1 EXPIRE ..
```

```
q)\s
2i
q)\t inv each 2 1000 1000#2000000?1f
2601
q)\t inv peach 2 1000 1000#2000000?1f
1462
```

`peach` keyword

The binary keyword `peach` can be used as a mnemonic alternative. The following are equivalent.

```
v1':[list]
(v1':)list
v1 peach list
```

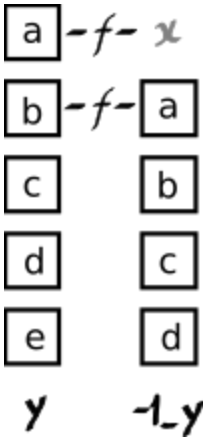
> **Higher-rank values**
>
> To parallelize a value of rank >1, use Apply to evaluate it on a list of arguments.
>
> Alternatively, define the value as a function that takes a parameter dictionary as argument, and pass the derived function a table of parameters to evaluate.

📖 `.Q.fc` parallel on cut
📖 Parallel processing
🎓 Table counts in a partitioned database
🕹 *Q for Mortals* A.68 `peach`

Each Prior

Apply a binary value between each item of a list and its preceding item.

```
(v2':)x    v2':[x]      (v2)prior x
x v2':y    v2':[x;y]
```



The Each Prior map takes a **binary** value and derives a variadic function. The derived function applies the value between each item of a list or dictionary and the item prior to it.

```
q)(-':)1 1 2 3 5 8 13
1 0 1 1 2 3 5
```

The first item of a list has, by definition, no prior item. If the derived function is applied as a binary, its left argument is taken as the 'seed' – the value preceding the first item.

```
q)1950 -': `S`J`C!1952 1954 1960
S| 2
J| 2
C| 6
```

If the derived function is applied as a unary, and the value is an operator with an identity element I known to q, I will be used as the seed.

```
q)(*':)2 3 4                    / 1 is I for *
2 6 12
q)(,':)2 3 4                    / () is I for ,
2
3 2
4 3
q)(-':) `S`J`C!1952 1954 1960   / 0 is I for -
S| 1952
J| 2
C| 6
```

If the derived function is applied as a unary, and the value is not an operator with a known identity element, a null of the same type as the argument (`first 0#x`) is used as the seed.

```
q){x+2*y}':[2 3 4]
0N 7 10
```

Q for Mortals §6.7.9 Each Prior

`prior` keyword

The mnemonic keyword `prior` can be used as an alternative to `':` .

```
q)(-':) 5 16 42 103
5 11 26 61
q)(-) prior 5 16 42 103
5 11 26 61
```

```
q)deltas 5 16 42 103
5 11 26 61
```

Case

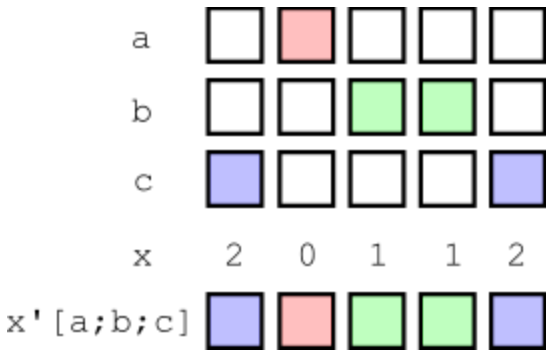
Pick successive items from multiple list arguments: the left argument of the iterator determines from which of the arguments each item is picked.

```
int'[a;b;c;…]
```

Where

- `int` is an integer vector
- $args$ `[a;b;c;…]` are the arguments to the derived function

the derived function `int'` returns r such that r_i is $(args_{int_i})_i$



The derived function `int'` has rank `max[int]+1`.

Atom arguments are treated as infinitely-repeated values.

```
q)0 1 0'["abc";"xyz"]
"ayc"
q)e:`one`two`three`four`five
q)f:`un`deux`trois`quatre`cinq
q)g:`eins`zwei`drei`vier`funf
q)l:`English`French`German
q)l?`German`English`French`French`German
2 0 1 1 2
q)(l?`German`English`French`French`German)'[e;f;g]
`eins`two`trois`quatre`funf

q)/extra arguments don't signal a rank error
q)0 2 0'["abc";"xyz";"123";"789"]
"a2c"
q)0 1 0'["a";"xyz"]  /atom "a" repeated as needed
"aya"
```

You can use Case to select between record fields according to a test on some other field.

Suppose we have lists `h` and `o` of home and office phone numbers, and a third list `p` indicating at which number the subject prefers to be called.

```
q)([]pref: p;home: h; office: o; call: (`home`office?p)'[h;o])
pref   home             office           call
-----------------------------------------------------------
home   "(973)-902-8196" "(431)-158-8403" "(973)-902-8196"
office "(448)-242-6173" "(123)-993-9804" "(123)-993-9804"
office "(649)-678-6937" "(577)-671-6744" "(577)-671-6744"
home   "(677)-200-5231" "(546)-864-5636" "(677)-200-5231"
home   "(463)-653-5120" "(636)-437-2336" "(463)-653-5120"
```

Case is a map. Consider the iteration's arguments as a matrix, of which each row corresponds to an argument.

```
q)a:`Kuh`Hund`Katte`Fisch
q)b:`vache`chien`chat`poisson
q)c:`cow`dog`cat`fish
q)show m:(a;b;c)
Kuh   Hund  Katte Fisch
vache chien chat  poisson
cow   dog   cat   fish
```

Case iterates the int vector as a mapping from column number to row number. It is a simple form of scattered indexing.

```
q)i:0 1 0 2
q)i,'til count i
0 0
1 1
0 2
2 3
q)m ./:i,'til count i
`Kuh`chien`Katte`fish
q)i'[a;b;c]
`Kuh`chien`Katte`fish
```

🎓 Table counts in a partitioned database

Empty lists

A map's derived function is uniform. Applied to an empty right argument it returns an empty list *without an evaluation*.

```
q)()~{x+y*z}'[`foo;mt;mt]    / generic empty list ()
1b
```

⚠ **Watch out for type changes when evaluating lists of unknown length.**

```
q)type (2*')til 5
7h
q)type (2*')til 0
0h
q)type (2*)til 0
7h
```

Accumulators

```
Converge  (v1\)x    v1\[x]       v1 scan x
          (v1/)x    v1/[x]       v1 over x

Do        n v1\x    v1\[n;x]
          n v1/x    v1/[n;x]

While     t v1\x    v1\[t;x]
          t v1/x    v1/[t;x]

Scan      (v2\)x    v2\[x]      (v2)scan x
Over      (v2/)x    v2/[x]      (v2)over x

Scan      x v2\y    v2\[x;y]
Over      x v2/y    v2/[x;y]

Scan                v3\[x;y;z]   x y\z
Over                v3/[x;y;z]
```

```
v1, v2, v3: applicable value (rank 1-3)
n:          integer≥0
t:          unary truth map
x, y:       arguments/indexes of v
```

An accumulator is an iterator that takes an applicable value as argument and derives a function that evaluates the value, first on its entire (first) argument, then on the results of **successive** evaluations.

There are two accumulators, Scan and Over. They have the same syntax and perform the same computation. But where the Scan-derived functions return the result of each evaluation, those of Over return only the last result.

Over resembles *map reduce* in some other programming languages.

```
q)(+\)2 3 4    / Scan
2 5 9
q)(+/)2 3 4    / Over
9
```

Debugging

If puzzled by the result of using Over, replace it with Scan and examine the intermediate results. They are usually illuminating.

Scan, Over and memory

While Scan and Over perform the same computation, in general, Over requires less memory, because it does not store intermediate results.

The number of successive evaluations is determined differently for unary and for higher-rank values.

The domain of the accumulators is functions, lists, and dictionaries that represent finite-state machines.

```
q)yrp                              / a European tour
from   to     wp
---------------
London Paris  0
Paris  Genoa  1
Genoa  Milan  1
Milan  Vienna 1
Vienna Berlin 1
Berlin London 0
q)show route:yrp[`from]!yrp[`to]    / finite-state machine
London| Paris
Paris | Genoa
Genoa | Milan
Milan | Vienna
Vienna| Berlin
Berlin| London
```

Unary values

```
(v1\)x    (v1/)x   / unary application
x v1\y    x v1/y   / binary application
```

The function an accumulator derives from a unary value is variadic. The result of the first evaluation is the right argument for the second evaluation. And so on.

The value is evaluated on the entire right argument, not on items of it.

When applied as a binary, the number of evaluations the derived function performs is determined by its left argument, or (when applied as a unary) by convergence.

syntax	name	number of successive evaluations
`(v1\)x` , `(v1/)x`	Converge	until two successive evaluations match, or an evaluation matches `x`
`i v1\x` , `i v1/x`	Do	`i` , a non-negative integer

syntax	name	number of successive evaluations
`t v1\x` , `t v1/x`	While	until unary value `t` , evaluated on the result, returns 0

Converge

```
q)(neg\)1                                / Converge
1 -1
q)l:-10?10
q)(l\)iasc l
4 0 8 5 7 2 6 3 1 9
0 1 2 3 4 5 6 7 8 9
1 8 5 7 0 3 6 4 2 9
8 2 3 4 1 7 6 0 5 9
2 5 7 0 8 4 6 1 3 9
5 3 4 1 2 0 6 8 7 9
3 7 0 8 5 1 6 2 4 9
7 4 1 2 3 8 6 5 0 9
q)(rotate[1]\)"abcd"
"abcd"
"bcda"
"cdab"
"dabc"
q)({x*x}\)0.1
0.1 0.01 0.0001 1e-08 1e-16 1e-32 1e-64 1e-128 1e-256 0
q)(route\)`Genoa                         / a circular tour
`Genoa`Milan`Vienna`Berlin`London`Paris
q)(not/) 42                              / never returns!
```

Are we there yet?

Matching is governed by comparison tolerance.

Do

```
q)dbl:2*
q)3 dbl\2 7                               / Do
2  7
4  14
8  28
16 56
q)5 enlist\1
1
,1
,,1
,,,1
,,,,1
,,,,,1

q)5(`f;)\1
1
(`f;1)
(`f;(`f;1))
```

```
(`f;(`f;(`f;1)))
(`f;(`f;(`f;(`f;1))))
(`f;(`f;(`f;(`f;(`f;1)))))
q)/first 10+2 numbers of Fibonacci sequence
q)10{x,sum -2#x}/0 1                    / derived binary applied infix
0 1 1 2 3 5 8 13 21 34 55 89
q)/first n+2 numbers of Fibonacci sequence
q)fibonacci:{x,sum -2#x}/[;0 1]         / projection of derived function
q)fibonacci 10
0 1 1 2 3 5 8 13 21 34 55 89
q)m:(0 1f;1 1f)
q)10 (m mmu)\1 1f                       / first 10 Fibonacci numbers
1  1
1  2
2  3
3  5
5  8
8  13
13 21
21 34
34 55
55 89
89 144
q)3 route\`London                       / 3 legs of the tour
`London`Paris`Genoa`Milan
```

A form of the conditional:

```
q)("j"$a=b) foo/bar                     / ?[a=b;foo bar;bar]
```

While

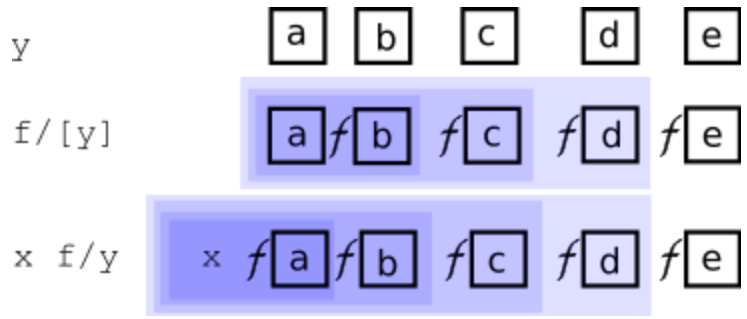
```
q)(10>)dbl\2                            / While
2 4 8 16
q){x<1000}{x+x}\2
2 4 8 16 32 64 128 256 512 1024
q)inc:1+
q)inc\[105>;100]
100 101 102 103 104 105
q)inc\[105>sum@;84 20]
84 20
85 21
q)(`Berlin<>)route\`Paris               / Paris to Berlin
`Paris`Genoa`Milan`Vienna`Berlin
q)waypoints:(!/)yrp`from`wp
q)waypoints route\`Paris                / Paris to the end
`Paris`Genoa`Milan`Vienna`Berlin
```

In the last example, both applicable values are dictionaries.

Binary values

```
x v\y    x v/y
```

The function an accumulator derived from a binary value is variadic. Functions derived by Scan are uniform; functions derived by Over are aggregates. The number of evaluations is the count of the right argument.



Unary and binary application of f/

Binary application

When the derived function is applied as a binary, the first evaluation applies the value to the function's left argument and the first item of the its right argument, i.e. `m[x;first y]`. The result of this becomes the left argument in the next evaluation, for which the right argument is the second item of the right argument. And so on.

```
q)1000+\2 3 4
1002 1005 1009
q)m                     / finite-state machine
1 6 4 4 2
2 7 2 0 5
7 5 6 7 0
2 1 8 1 0
7 3 3 6 8
2 3 8 9 0
1 1 9 6 9
7 8 4 3 0
4 5 8 0 4
9 8 0 3 9
q)c                     / columns of m
4 1 3 3 1 4
q)7 m\c
0 6 6 6 1 5
```

Items of `x` must be in the left domain of the value, and items of `y` in its right domain.

Unary application

When the derived function is applied as a unary, and the value is **a function with a known identity element** I, then I is taken as the left argument of the first evaluation.

```
q)(,\)2 3 4       / I is ()
,2
```

```
2 3
2 3 4
```

In such cases `(I f\x)~(f\)x` and `(I f/x)~(f/)x` and items of `x` must be in the right domain of the value.

Otherwise, the first item of the right argument is taken as the result of the first evaluation.

```
q){x,y}\[2 3 4]       / () not known as I
2
2 3
2 3 4
q)42{[x;y]x}\2 3 4    / 42 is the first left argument
42 42 42
q)({[x;y]x}\)2 3 4    / 2 is the first left argument
2 2 2
q)(m\)c               / c[0] is the first left argument
4 3 1 0 6 9
```

In this case, for `(v\)x` and `(v/)x`

- `x[0]` is in the *left* domain of `v`
- items `1_x` are in the *right* domain of `v`

but `x[0]` need not be in the range of `v`.

```
q)({count x,y}\)("The";"quick";"brown";"fox")
"The"
8
6
4
```

Keywords `scan` and `over`

Mnemonic keywords `scan` and `over` can be used to apply a binary value to a list or dictionary.

Parenthesize an infix to pass it as a left argument.

```
q)(+) over til 5          / (+/)til 5
10
q)(+) scan til 5          / (+\)til 5
0 1 3 6 10
q)m scan c                / (m\)c
4 3 1 0 6 9
```

☞ over, scan

Ternary values

```
v\[x;y;z]    v/[x;y;z]
```

The function an accumulator derives from an value of rank >2 has the same rank as the value. Functions derived by Scan are uniform; functions derived by Over are aggregates. The number of evaluations is the maximum of the count of the right arguments.

For `v\[x;y;z]` and `v/[x;y;z]`

- `x` is in the left domain of `v`
- `y` and `z` are atoms or conforming lists or dictionaries in the right domains of `v`

The first evaluation is `v[x;first y;first z]`. Its result becomes the left argument of the second evaluation. And so on. For `r:v\[x;y;z]`

```
r[0]: v[x  ; y 0; z 0]
r[1]: v[r 0; y 1; z 1]
r[2]: v[r 1; y 2; z 2]
…
```

The result of `v/[x;y;z]` is simply the last item of the above.

`v/[x;y;z]`

$$v[\ v[\ldots\ v[\ v[x;y_0;z_0]\ ;y_1;z_1];\ \ldots\ y_{n-2};z_{n-2}];\ y_{n-1};z_{n-1}]$$

```
q){x+y*z}\[1000;5 10 15 20;2 3 4 5]
1010 1040 1100 1200
q){x+y*z}\[1000 2000;5 10 15 20;3]
1015 2015
1045 2045
1090 2090
1150 2150
q)// Chinese whispers
q)s:"We are going to advance. Send reinforcements."
q)ssr\[s;("advance";"reinforcements");("a dance";"three and fourpence")]
"We are going to a dance. Send reinforcements."
"We are going to a dance. Send three and fourpence."
```

The above description of functions derived from ternary values applies by extension to values of higher ranks.

Alternative syntax

As of V3.1 2013.07.07, Scan has a built-in function for the following.

```
q)a:1000f;b:1 2 3 4f;c:5 6 7 8f
q){z+x*y}\[a;b;c]
1005 2016 6055 24228f
q)a b\c
1005 2016 6055 24228f
```

Note that the built-in version is for floats.

Empty lists

> ⚠ **Accumulators can change datatype**

In iterating through an empty list **the value is not evaluated.** The result might not be in the range of the value.

Allow for a possible change of type to `0h` when scanning or reducing lists of unknown length.

```
q)mt:0#0
q)type each (mt;*/[mt];{x*y}/[mt])  / Over can change type
7 -7 0h
q)type each (mt;*\[mt];{x*y}\[mt])  / so can Scan
7 -7 0h
```

Scan

The function Scan derives from a non-unary value is a uniform function: for empty right argument/s it returns the generic empty list. It does not evaluate the value.

```
q)()~{x+y*z}\[`foo;mt;mt]           / lambda is not evaluated
1b
```

Over

The function that Over derives from a non-unary value is an aggregate: it reduces lists and dictionaries to atoms.

For empty right argument/s the atom result depends on the value and, if the derived function is variadic, on how it is applied.

If the value is a **binary function with a known identity element** I, and the derived function is applied as a unary, the result is I.

```
q)(+/)mt    / 0 is I for +
0
q)(*/)mt    / 1 is I for *
1
```

If the value is a **binary function with no known identity element**, and the derived function is applied as a unary, the result is `()`, the generic empty list.

```
q)()~({x+y}/)mt
1b
```

If the value is a **list** and the derived function is applied as a unary, the result is an empty list of the same type as the list.

```
q)type 1 0 3h/[til 0]
5h
q)type (3 4#til 12)/[0#0]
0h
```

Otherwise, the result is the left argument.

```
q)42+/mt
42
q){x+y*z}/[42;mt;mt]
42
q)42 (3 4#til 12)/[0#0]
42
```

The value is not evaluated.

```
q)`foo+/mt
`foo
q){x+y*z}/[`foo;mt;mt]
`foo
```

Q for Mortals §6.7.6 Over (/) for Accumulation

Iterators

by Conor Slattery & Stephen Taylor

Iterators (formerly known as *adverbs*) are the primary means of iteration in q, and in almost all cases the most efficient way to iterate. Loops are rare in q programs and are almost always candidates for optimization. Mastery of iterators is a core q skill.

The first part of this paper introduces iterators informally. This provides ready access to the two principal forms of iteration: *maps* and *accumulators*.

The second part of the paper reviews iterators more formally and with greater attention to syntax. We see how iterators apply not only to functions but also to lists, dictionaries and tables. From their syntax we see when parentheses are required, and why.

We discuss examples of their use.

Basics

Iterators are higher-order unary operators: they take a single argument and return a derived function. The single argument is an *applicable value*: a list, dictionary, table, process handle, or function. The derived function iterates its normal application.

Iterators are the only operators that can be applied postfix. They almost always are.

For example, the iterator Scan, written `\`, applied to the Add operator `+` derives the function Add Scan, written `+\`, which extends Add to return cumulative sums.

```
q)+\[2 3 4]
2 5 9
```

Applied to the Multiply operator `*` it derives the function Multiply Scan, written `*\`, which returns cumulative products.

```
q)*\[2 3 4]
2 6 24
```

(Writers of some other programming languages might recognize these uses of Scan as *fold*.)

Another example. The iterator Each, written `'`, applied to the Join operator `,`, derives the function Join Each, written `,'`.

```
q)show a:2 3#"abcdef"
"abc"
"def"
```

```
q)show b:2 3#"uvwxyz"
"uvw"
"xyz"
q)a,b
"abc"
"def"
"uvw"
"xyz"
q)a,'b
"abcuvw"
"defxyz"
```

Above, `a` and `b` are both 2×3 character matrixes. That is to say, they are both 2-lists, and their items are character 3-lists. While `a,b` joins the two lists to make a 4-list, the derived function Join Each `a,'b` joins their corresponding items to make two character 6-lists.

Scan and Each are the cores of the accumulator and map iterators. The other iterators are variants of them.

Three kinds of iteration

Atomic iteration

Many native q operators have iteration built into them. They are atomic. They apply to conforming arguments.

```
q)2+2           / two atoms
4
q)2 3 4+5 6 7   / two lists
7 9 11
q)2+5 6 7       / atom and list
7 8 9
```

Two arguments conform if they are lists of the same length, or if one or both is an atom. In atomic iteration this definition recurses *to any depth of nesting*.

```
q)(1;2;3 4)+( (10 10 10;20 20); 30; ((40 40; (50 50 50; 60)); 70) )
(11 11 11;21 21)
32
((43 43;(53 53 53;63));74)
```

Because atomic iteration burrows deep into nested structure it is not easy to parallelize. A simpler form of it is.

Mapping

The map iterators apply a function across items of a list or lists. They do not burrow into a nested structure, but simply iterate across its top level. That is just what **Each** does.

```
q)x:("the";"quick";"brown";"fox")
q)count[x]          / count x
4
```

```
q)count'[x]         / count each item of x
3 5 5 3
```

The Each iterator has four variants. A function derived by **Each Right** `/:` applies its entire left argument to each item of its right argument. Correspondingly, a function derived by **Each Left** `\:` applies its entire right argument to each item of its left argument.

```
q)"abc",/:"xyz"     / Join Each Right
"abcx"
"abcy"
"abcz"
q)"abc",\:"xyz"     / Join Each Left
"axyz"
"bxyz"
"cxyz"
```

Each Left and Each Right

Remember which is which by the direction in which the slash leans.

Each Prior takes a binary value as its argument. The derived function is unary: it applies the binary between each item of a list (or dictionary) and its preceding item. The differences between items in a numeric or temporal vector:

```
q)-':[1 1 2 3 5 8 13 21 34]     / Subtract Each Prior
1 0 1 1 2 3 5 8 13
```

Each Parallel takes a unary argument and applies it, as Each does, to each item in the derived function's argument. Unlike Each, it partitions its work between any available secondary processes. Suppose `analyze` is CPU-intensive and takes a single symbol atom as argument.

```
q)analyze':[`ibm`msoft`googl`aapl]
```

With a unary function, the mnemonic keyword `each` is generally preferred as a cover for the iterator Each. Similarly, `prior` is preferred for Each Prior and `peach` for Each Parallel.

```
q)count each ("the";"quick";"brown";"fox")
3 5 5 3
q)(-) prior 1 1 2 3 5 8 13 21 34
1 0 1 1 2 3 5 8 13
q)analyze peach `ibm`msoft`googl`aapl
..
```

With map iterators the number of evaluations is the number of top-level items in the derived function's argument/s. These functions are right-uniform.

The map iterators:

glyph	name	mnemonic keyword
'	Each	each
\:	Each Left	
/:	Each Right	
':	Each Prior	prior
':	Each Parallel	peach
'	Case	

Accumulation

In accumulator iterations the value is applied repeatedly, first to the entire (first) argument of the derived function, next to the result of that evaluation, and so on.

The number of evaluations is determined according to the value's rank.

For a **unary** value, there are three forms:

- Converge: iterate until a result matches either the previous result or the original argument
- Do: iterate a specified number of times
- While: iterate until the result fails a test

```
q){x*x}\[0.1]                       / Converge
0.1 0.01 0.0001 1e-08 1e-16 1e-32 1e-64 1e-128 1e-256 0
q)5{x*x}\0.1                        / Do
0.1 0.01 0.0001 1e-08 1e-16 1e-32
q)(1e-6<){x*x}\0.1                  / While
0.1 0.01 0.0001 1e-08
```

For values of **higher-rank** the number of evaluations is the count of the right argument/s. For example, the result `r` of applying a ternary derived function `f\` to arguments `x`, `y`, and `z`:

```
r[0]:f[x;   y 0; z 0]
r[1]:f[r 0; y 1; z 1]
r[2]:f[r 1; y 2; z 2]
..
```

From this we see that the right arguments `y` and `z` must conform and that `count r` – the number of evaluations – is `count[y]|count[z]`.

There are two accumulator iterators.

- Functions derived by **Scan** `\` return as a list the results of each evaluation. They are thus right-uniform functions: their results conform to their right arguments. They resemble *fold* in some other programming languages.
- Functions derived by **Over** `/` perform the same computation as those from Scan, but return only the last result. They resemble *map reduce* in some other programming languages.

```
q)+\[2 3 4]    / Add Scan
2 5 9
q)+/[2 3 4]    / Add Over
9
```

With Scan and Over and binary values, the mnemonic keywords `scan` and `over` are generally preferred.

```
q)(+) scan 2 3 4
2 5 9
q)(+) over 2 3 4
9
```

The accumulators:

glyph	name	mnemonic keyword
\	Scan	scan
/	Over	over

Brackets and parentheses

The result of applying an iterator to a value is a derived function. Like any other function, a derived function can be applied with brackets.

```
q)+\[3 4 5]
3 7 12
q)+\[1000;3 4 5]
1003 1007 1012
```

Notice that the derived function here is *variadic*: it can be applied as a unary or as a binary.

Postfix yields infix

An iterator applied postfix derives a function with infix syntax.

This is true regardless of the derived function's rank. For example, `count'` is a unary function but has infix syntax.

We can also apply `+\` as a binary using infix syntax.

```
q)1000+\3 4 5
1003 1007 1012
```

The syntax of q allows clear expression of a sequence of operations. To apply a function `primus` to `x`, then `secundus` to the result, then `tertius` to the result of that, we could write:

```
q)tertius[secundus[primus[x]]]
```

but better style would be to apply the unaries prefix and write:

```
q)tertius secundus primus x
```

Good q style minimizes use of both brackets and parentheses. (Where they must be used, the less that appears between an opening bracket or parenthesis and its close, the better.) So prefix application is usually better.

An infix function can be applied prefix as a unary by parenthesizing it.

```
q)(+\)3 4 5 6 7
3 7 12 18 25
```

Some common derived functions are covered by keywords for readability. Good q style prefers them.

```
q)sums 3 4 5 6 7
3 7 12 18 25
```

The iterator Each is covered by the keyword `each` for unary values. Good q style prefers it.

```
q)(count')  ("the";"quick";"brown";"fox")
3 5 5 3
q)count each("the";"quick";"brown";"fox")  / better q style
3 5 5 3
```

Parenthesize an infix function to use it as the left argument of a another function.

```
q)(+) scan 3 4 5 6 7
3 7 12 18 25
```

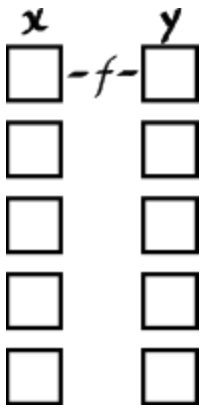
Map iterators

Each, Each Parallel

The Each iterator applies a unary to each item of an argument.

```
q)x:("the";"quick";"brown";"fox")
q)reverse x
"fox"
"brown"
"quick"
"the"
q)reverse each x
"eht"
"kciuq"
"nworb"
"xof"
```

With a binary value, the iterator is sometimes known as *each both*. You can think of it as a zip fastener, applying the value between pairs of items from its arguments.



Each Both

```
q)2 0 1 -2 rotate' x
"eth"
"quick"
"rownb"
"oxf"
q)show m:4 5#"the  quickbrownfox  "
"the  "
"quick"
"brown"
"fox  "
q)1 2 1 1 1 0 0 m' 3 1 2 3 4 2 0        / scattered indexing
"cricket"
```

The Each Parallel iterator takes unary values. It derives functions that perform exactly the same computation as functions derived by Each, but delegates computation to secondary tasks, if any are available.

Good q style prefers use of the `peach` keyword.

```
q)sum peach 3?'5000#10
13 12 13 22 3 14 17 14 7 12 13 17 19 15 8 16 17 18 19 10 16 10 9 13 15 8 25 8..
```

The Each iterator has three variants that take binary values as arguments: Each Left, Each Right and Each Prior.

Each Left, Each Right

With a function derived from Each and a binary, if one of the two arguments is an atom, it is paired with each item of the other argument.

```
q)3,'til 4
3 0
3 1
3 2
3 3
q)"o" in' ("the";"quick";"brown";"fox")
0011b
```

```
q)2 m' 0 2 1 2 4
"boron"
```

To extend this behavior to non-atom arguments, use Each Left or Each Right.

Each Right

```
q)x:("the";"quick";"brown";"fox")
q)y:("brown";"windsor";"soup")
q)z:("red";"riding";"hood")
q)"brown" in/: (x;y;z)                  / Each Right
110b
```

For example, find the file handle of each column of a table.

```
q){x,/:key[x]except `.d} `:/mydb/2013.05.01/trade
`:/mydb/2013.05.01/trade`sym
`:/mydb/2013.05.01/trade`time
`:/mydb/2013.05.01/trade`price
`:/mydb/2013.05.01/trade`size
`:/mydb/2013.05.01/trade`ex
```

The file handle of the table is joined to each element in the list of columns, creating five 2-lists. The Each Right iterator can then apply the `sv` keyword to create the file handles of each column.

```
q)` sv/: {x,/:key[x]except `.d} `:/mydb/2013.05.01/trade
`:/mydb/2013.05.01/trade/sym`:/mydb/2013.05.01/trade/time`:/mydb/2013.05.01/t..
```

Each Prior

The Each Prior iterator applies its binary to each item of a list `x` and to the previous item; i.e. to each adjacent pair in the list. The result is a list of the same length as `x`: the derived function is a uniform function.

```
q)(-':) 4 8 3 2 2
4 4 -5 -1 0
```

Good q style prefers use of the `prior` keyword.

```
q)(-) prior 4 8 3 2 2
4 4 -5 -1 0
```

One use of `-':` is so common it is built in as the `deltas` keyword.

```
q)deltas 4 8 3 2 2
4 4 -5 -1 0
```

As a uniform function `-':` returns a list as long as its argument. The first item of the result corresponds to the first item of the argument. But, by definition, the first item of the argument has no previous item. So, in the expression $4 - y$, what is y? Above, it is zero. Zero is the identity element for Subtract: when y is zero, $x - y$ is always x. (See

Each Prior in the Reference for more on this and what happens with values which do not have a known identity element.)

We can use `{x,y}` to display the pairs that Each Prior finds.

```
q){x,y}prior til 5
0
1 0
2 1
3 2
4 3
```

Here we see that the first item, `0`, is paired with `0N`. The Join operator has no identity element, so it uses the argument `til 5` as a prototype.

```
q)(0,1#0#x) ~ first {x,y}':[x]
1b
```

A table in the Reference tells us that with a binary value Each Prior derives a variadic function. So Subtract Each Prior can also be applied as a binary.

```
q)0 -': 4 8 3 2 2
4 4 -5 -1 0
```

The zero left argument is the 'seed' – the y subtracted from the first item, 4. We can use another 'seed' value.

```
q)1 -': 4 8 3 2 2
3 4 -5 -1 0
```

Watch out Using a float as the seed shifts the type of the first item of the result. But *only* the first item: the result is no longer a vector, but a mixed list.

```
q)0.5 -': 4 8 3 2 2
3.5
4
-5
-1
0
q)type each 0.5 -': 4 8 3 2 2
-9 -7 -7 -7 -7h
```

Each Prior can be useful in tracking down errors within lists which should be identical, e.g. the `.d` files for a table in a partitioned database. Let's use the `differ` keyword to check for inconsistencies in `.d` files. It uses the Each Prior iterator and is equivalent to `{not(~':)x}`.

```
q)1_ date where differ {get hsym `$"/mydb/",x,"/trade/.d"} each string date
2013.05.03 2013.05.04
```

The values of the `.d` files are extracted from each partition. The `differ` keyword then compares each item with the one before it. If a `.d` file differs from the previous `.d` file in the list, then that date will be returned. The first item of the result is dropped, because the first item of list `x` will be compared to `x[-1]`, which is always null and so will never match. In the example above, the `.d` files for the `2013.05.03` and `2013.05.04` partitions differ, and should be investigated further.

Higher ranks

Each Parallel, `peach`, and `each` apply unary values. Each Left, Each Right, Each Prior, and `prior` apply binary values.

The Each iterator applies values of any rank.

```
q)1 2 3 in' (1 2 3;3 4 5;5 6 7)
100b
q)ssr'[("mad";"bud";"muy");"aby";"umd"]
"mud"
"mud"
"mud"
```

As with atomic iteration, the arguments must conform: if lists, their lengths match; if atoms, they are replicated to the length of the list/s.

```
q)ssr'[("mad";"bud";"muy");"d";"pnx"]
"map"
"bun"
"muy"
```

Accumulating iterators

There are two accumulating iterators (or *accumulators*) and they are really the same. The Scan iterator is the core; the Over iterator is a slight variation of it.

Here is Scan at work with `ssr`.

```
q)ssr\["hello word."; ("h";".";"rd"); ("H";"!";"rld")]
"Hello word."
"Hello word!"
"Hello world!"
```

Where we want only the final result, we use the Over iterator.

```
q)ssr/["hello word."; ("h";".";"rd"); ("H";"!";"rld")]
"Hello world!"
```

It is generally true that for a value `v`, `v/[x]` is the same as `last v\[x]`. Otherwise, what is true for Scan is true for Over. (Using Over rather than Scan allows kdb+ to use a little less memory by dropping interim results.)

Debugging

If puzzled by the result of using Over, replace it with Scan and examine the intermediate results. They are usually illuminating.

Consider the application of the functions Scan derives from values of ranks 1, 2, 3...

```
v1\[x]          (v1\)x          Converge
v1\[i;x]       i v1\ x          Do
v1\[t;x]       t v1\ x          While
v2\[x;y]       x v2\ y
v3\[x;y;z]
…
```

And so on, up to `f8\`. In each form, `x` is the (first) argument of the first evaluation. The result of the first evaluation becomes the (first) argument for the next evaluation, if any. And so on.

For how many evaluations? It depends first on the rank of the value.

value rank	number of evaluations
2-8	length of the right argument/s
1	depends on the results, and the left argument (if any) of the derived function

If `f` is rank 2 or higher, the number of evaluations is determined by the length of the right argument/s. In the `ssr` example above, the right arguments have length 3 and `ssr` is evaluated three times.

There are three ways to determine the number of evaluations of `f1` performed by `f1\`.

A table in the Reference tells us `f1\` is variadic and can be applied as a unary (Converge) or as a binary (Do or While).

Converge

Applying `f1\` as a unary is known as Converge because `f1\` is evaluated until the result matches either the previous result or the original argument.

```
q)({x*x}\)0.1          / converge on result
0.1 0.01 0.0001 1e-08 1e-16 1e-32 1e-64 1e-128 1e-256 0
q)(not\)0b             / return to first argument
01b
```

Watch out Not all sequences converge in this way. You can use the accumulators to throw kdb+ into an infinite loop.

Set a timeout

When developing with Converge, it is wise to set the timeout in your session via the `\T` command. This will cause any functions to terminate after a set number of seconds. An infinite loop will not lock your session.

Recursion with `.z.s`

The 'self' function `.z.s` can also be used for recursion, and is more flexible than Scan.

```
q)list:(`a`n;(1 2;"efd");3;("a";("fes";3.4)))
q){$[0h=type x;.z.s'[x];10h=abs type x;upper x;x]}list
`a`n
(1 2;"EFD")
3
("A";("FES";3.4))
```

The above function operates on a list of any structure and data types, changing strings and characters to upper case and leaving all other elements unaltered.

Note that when using `.z.s` the function will signal a stack error after 2000 loops.

```
{.z.s[0N!x+1]}0
```

No such restriction exists on Scan and Over.

Use `.z.s` only where it is not possible to use iterators.

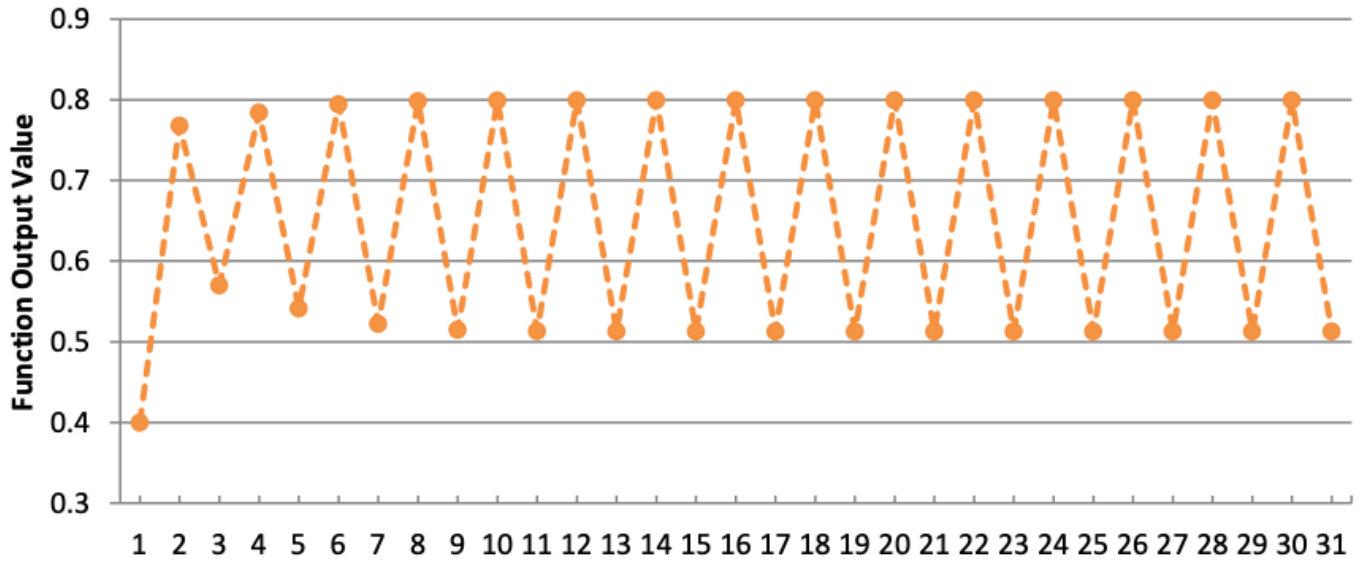
Do and While

Applying `f1\` as a binary allows us to set an exit condition as the left argument:

- an integer: exit after this number of evaluations (Do)
- a truth function: apply this to each result and exit when it evaluates to zero (While)

Consider the function defined and illustrated below.

```
q)30 {3.2*x*1-x}\ 0.4
0.4 0.768 0.5701632 0.7842468 0.541452 0.7945015 0.5224603 0.7983857 0.515091..
```



It is evident the function results in a loop with period 2 (at least within floating point tolerance). Without the exit condition (above, 30 evaluations) it will not terminate.

Fibonacci numbers are calculated by joining an integer vector to the sum of its last two items.

```
q)fib:{x,sum -2#x}
q)5 fib\1 1                          / exit after 5 evaluations
1 1
1 1 2
1 1 2 3
1 1 2 3 5
1 1 2 3 5 8
1 1 2 3 5 8 13
q){last[x]<200} fib/1 1              / exit when result exceeds 200
1 1 2 3 5 8 13 21 34 55 89 144 233
```

Lists and dictionaries

The arguments of iterators are applicable values: functions, file- and process handles, lists and dictionaries. Functions are the most familiar as iterator arguments, but lists and dictionaries reward study.

```
q)yrp                                / a European tour
from   to     wp
----------------
London Paris  0
Paris  Genoa  1
Genoa  Milan  1
Milan  Vienna 1
Vienna Berlin 1
Berlin London 0
q)show route:yrp[`from]!yrp[`to]
London| Paris
Paris | Genoa
Genoa | Milan
```

```
Milan | Vienna
Vienna| Berlin
Berlin| London
```

The dictionary `route` is a finite-state machine: its values are also valid keys.

```
q)(route\)`Genoa                          / a circular tour
`Genoa`Milan`Vienna`Berlin`London`Paris
q)3 route\`London                         / 3 legs of the tour
`London`Paris`Genoa`Milan
q)(`Berlin<>)route\`Paris                 / Paris to Berlin
`Paris`Genoa`Milan`Vienna`Berlin
q)waypoints:(!/)yrp`from`wp
q)waypoints route\`Paris                  / Paris to the end
`Paris`Genoa`Milan`Vienna`Berlin
```

In the last expression, both the value and the truth value are dictionaries. No functions are involved.

Combining iterators

We can calculate Pascal's Triangle using Scan and Each Prior.

We already have a sufficient grasp of the accumulators to see the Triangle immediately as successive results from some use of Scan. We need only a function to define one row of the Triangle in terms of the row above it. That could hardly be simpler. Each row is derived from its parent by summing adjacent pairs of items.

```
q)(+) prior 1 3 3 1       / nearly...
1 4 6 4
q)(+) prior 1 3 3 1,0     / ...there!
1 4 6 4 1
q)7 {(+)prior x,0}\ 1
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
```

Notice that the last expression gave us *eight* rows of the Triangle, not seven. The first item of the result was the original argument, followed by the results of seven successive evaluations.

If the left argument of the derived function is zero, there will be no evaluations. The original argument will still be returned as the first (and only) item of the result. It doesn't even have to be in the domain of the value.

```
q)0 {(+)prior x,0}\ 1
,1
q)0 {(+)prior x,0}\ "foo"
"foo"
```

You can use Do in this way as a conditional.

The argument of an iterator is a function, list, or dictionary. Its result is a derived function – so that too can be the argument to an iterator.

Generate all possible pairs from two lists:

```
q)raze 1 2 3,/:\:4 5 6
1 4
1 5
1 6
2 4
2 5
2 6
3 4
3 5
3 6
```

Here the derived function `,/:` (Join Each Right) is the argument to the iterator `\:` (Each Left). The resulting binary derived function `,/:\:` (Join Each Right Each Left) has infix syntax – remember *Postfix yields infix*. In the example above it returns a 3×3 matrix of pairs, here razed into a list of pairs because the default display does not reveal the matrix structure.

Switch the iterators to change the order of the result.

```
q)raze 1 2 3,\:/:4 5 6
1 4
2 4
3 4
1 5
2 5
3 5
1 6
2 6
3 6
```

The `raze` keyword is no more than `,/` (Join Over). It is often combined with Converge to flatten deeply nested structures. Good q style prefers the keywords, but `,//` illustrates again how a derived function can be the argument to another iterator.

```
q)raze over ("the ";("quick ";("brown ";"fox.")))
"the quick brown fox."
q)(,//)("the ";("quick ";("brown ";"fox.")))
"the quick brown fox."
```

You can use the Each operator to apply a function at a specific depth in a nested list.

```
q)lst:(3 2 8;(3.2;6h);("AS";4))
q)type lst
0h
q)type'[lst]
7 0 0h
```

```
q)type''[lst]
-7 -7 -7h
-9 -5h
10 -7h
q) type'''[lst]
-7 -7 -7h
-9 -5h
(-10 -10h;-7h)
```

Good q style avoids brackets and parentheses, and prefers keywords where available. These principles conflict when composing multiple iterators.

```
q)((type each)each)each lst         / NOT recommended
-7 -7 -7h
-9 -5h
(-10 -10h;-7h)
```

When composing iterators, prefer concise forms. They are easier to analyze.

Iterators versus loops

The control words `do` and `while` allow q programmers to write explicit loops. Sometimes this is the only way to write an iteration. However, most common forms of iteration can be defined using the iterators, yielding code that is shorter, faster and less prone to error.

Often the implementation is relatively easy, using Each, Each Left and Each Right to cycle through a list and amend its items.

Suppose we wanted to check if either of the integers 2 or 3 are present in some lists. This can be achieved with a `while` loop.

```
q)m:(1 2 3;3 4 5;4 5 6)   / three lists
q){i:0;a:();while[i<count x;a,:enlist any 2 3 in x[i];i+:1];a} m
110b
q)\t:100000 {i:0;a:();while[i<count x;a,:enlist any 2 3 in x[i];i+:1];a} m
475
```

However, iterators allow neater, more efficient code; easier to read and cheaper to maintain.

```
q)any each 2 3 in/: m
110b
q)\t:10000 any each 2 3 in/: m
30
```

Similarly we can use the Over iterator to deal easily with situations which would be handled by loops in C-like languages.

Suppose you wanted to join several tables.

```
//Create a list of tables, of random length
q)tt:{1!flip(`sym;`$"pr",x;`$"vol",x)!(`a`b`c;3?50.0;3?100)}each string til 2+rand 10
//Join the tables using a while loop
q){a:([]sym:`a`b`c);i:0;while[i<count[x];a:a lj x[i];i+:1];a}tt
sym pr0      vol0 pr1      vol1 pr2      vol2 pr3      vol3 pr4       vol4 pr..
-------------------------------------------------------------------------------..
a   25.41992 86   3.315151 58   22.37118 81   7.696889 56   14.84522  80   27..
b   2.124254 50   19.3025  82   34.57479 69   47.95351 85   0.5641467 45   33..
c   37.36208 19   41.11905 31   33.52813 90   30.03506 78   1.949377  8    28..
q)\t:10000 {a:([]sym:`a`b`c);i:0;while[i<count[x];a:a lj x[i];i+:1];a}tt
271
//Join the tables using Over
q)0!(lj/)tt
sym pr0      vol0 pr1      vol1 pr2      vol2 pr3      vol3 pr4       vol4 pr..
-------------------------------------------------------------------------------..
a   25.41992 86   3.315151 58   22.37118 81   7.696889 56   14.84522  80   27..
b   2.124254 50   19.3025  82   34.57479 69   47.95351 85   0.5641467 45   33..
c   37.36208 19   41.11905 31   33.52813 90   30.03506 78   1.949377  8    28..
q)\t:10000 0!(lj/)tt
257
```

Write loops only when you can find no solution using iterators.

Nested columns

Best practice avoids nested columns wherever possible. However in some situations operating on nested data is necessary or may lower execution time for certain queries. The most common occasion for this is that the keyword `ungroup`, which flattens a table containing nested columns, is computationally expensive, especially when you are dealing only with a subset of the entire table.

There are also situations in which storing the data in a nested structure makes more sense. For example, you may want to use strings instead of symbols, to avoid a bloated sym file.

So we will now take a look at using iterators to apply functions to a table as a whole, and to apply functions within a select statement.

Iterators can be used to examine and modify tables. To do this we need to understand how tables are structured. In kdb+, a table is a list of dictionaries, which may have non-integer values. This means we can apply functions to individual values, just like any other nested list or dictionary structure. For example:

```
q)a:([]a:`a`b`c`d;b:1 2 3 4;c:(1 2;2 3;3 4;4 5))
q)type[a]
98h
q)type'[a]
99 99 99 99h
q)type''[a]
a   b  c
--------
-11 -7 7
-11 -7 7
```

```
-11 -7 7
-11 -7 7
```

We see here that `type[a]` returns `98h`, so `a` is a table. Then `type'[a]` returns the type of each item of `a` – they are dictionaries, with type `99h`. Next, `type''[a]` finds the type of each value of each dictionary. The result is a list of dictionaries, which collapses back to a table showing the type of each field in the table.

```
q)distinct type''[a]
a   b  c
--------
-11 -7 7
```

This statement can be used to ensure all rows of the table are the same type. This is useful if your table contains nested columns, as the `meta` keyword looks only at the first row of nested columns. And if the table is keyed then `meta` will be applied only to the non-key columns.

```
q)a:([]a:`a`b`c`d;b:1 2 3 4;c:(1 2;2 3;3 4.;4 5))
q)meta a
c| t f a
-| -----
a| s
b| j
c| J
q)distinct type''[a]
a b c
--------
-11 -7 7
-11 -7 9
```

Looking only at the results of `meta`, we might suppose column `c` contains only integer lists. However `distinct type''[a]` clearly shows column `c` contains lists of different types, and thus is not mappable. This is a common cause of error when writing to a splayed table. Dealing with nested data in a table via a select/update statement often requires the use of iterators. To illustrate this, let us define a table with three columns, two of which are nested.

```
q)show tab:([]sym:`AA`BB`CC;time:3#enlist 09:30+til 30;price:{30?100.0}each til 3)
sym time                                                                    ..
------------------------------------------------------------------------------..
AA  09:30 09:31 09:32 09:33 09:34 09:35 09:36 09:37 09:38 09:39 09:40 09:41 0..
BB  09:30 09:31 09:32 09:33 09:34 09:35 09:36 09:37 09:38 09:39 09:40 09:41 0..
CC  09:30 09:31 09:32 09:33 09:34 09:35 09:36 09:37 09:38 09:39 09:40 09:41 0..
```

Suppose we wanted the range of each row.

```
q)rng:{max[x]-min[x]}
```

We can use `rng` with Each within a `select` statement to apply the function to each row of the table.

```
q) select sym, rng'[price] from tab
sym price
```

```
------------
AA  96.3872
BB  95.79704
CC  98.31252
```

Suppose instead we wanted the range of a subset of the data in the table. One way would be to `ungroup` the table and find the range.

```
q)select rng price by sym from ungroup tab where time within 09:40 09:49
sym| price
---| --------
AA | 77.67457
BB | 80.14611
CC | 67.48254
```

However, it is faster to index into the nested list as this avoids the costly `ungroup` function. First find the index of the prices within our time range.

```
q)inx:where (exec first time from tab) within 09:40 09:49
```

Then use this to index into each price list and apply `rng` to the resulting prices.

```
q)select sym, rng'[price@\:inx] from tab
sym inx
------------
AA  77.67457
BB  80.14611
CC  67.48254
```

This offers a significant improvement in latency over using `ungroup` .

```
q)\t:10000 select rng price by sym from ungroup tab where time within 09:40 09:49
175
q)\t:10000 inx:where (exec first time from tab) within 09:40 09:49;select sym, rng'[price@\:inx] from tab
83
```

If the nested lists are not uniform, change the code:

```
q)inx:where each (exec time from tab) within 09:40 09:49
q)select sym, rng'[price@'inx] from tab
sym inx
------------
AA  77.67457
BB  80.14611
CC  67.48254
```

Authors

Conor Slattery is a financial engineer who has designed kdb+ applications for a range of asset classes. Conor is currently working with a New York-based investment firm, developing kdb+ trading platforms for the US equity markets.

Stephen Taylor FRSA has followed the evolution of the Iversonian languages through APL, J, k, and q, and is a former editor of *Vector*, the journal of the British APL Association.

each , peach

Iterate a unary

```
 v1 each x   each[v1;x]       v1 peach x   peach[v1;x]
(vv)each x   each[vv;x]     (vv)peach x   peach[vv;x]
```

Where

- `v1` is a unary applicable value
- `vv` is a variadic applicable value

applies `v1` or `vv` as a unary to each item of `x` and returns a result of the same length.

That is, the projections `each[v1;]` , `each[vv;]` , `peach[v1;]` , and `peach[vv;]` are uniform functions.

```
q)count each ("the";"quick";" brown";"fox")
3 5 5 3
q)(+\)peach(2 3 4;(5 6;7 8);9 10 11 12)
2 5 9
(5 6;12 14)
9 19 30 42
```

`each` and `peach` perform the same computation and return the same result.

`peach` will divide the work between available secondary tasks.

Changes since 4.1t 2024.01.04

peach workload distribution methodology changed to dynamically redistribute workload and allow nested invocation. The limitations on nesting have been removed, so peach (and multi-threaded primitives) can be used inside peach. To facilitate this, round-robin scheduling has been removed. Even though the initial work is still distributed in the same manner as before for compatibility, the workload is dynamically redistributed if a thread finishes its share before the others.

`each` is a wrapper for the Each iterator. `peach` is a wrapper for the Each Parallel iterator. It is good q style to use `each` and `peach` for unary values.

`each` is redundant with atomic functions. (Common qbie mistake.)

Maps for uses of Each with binary and higher-rank values
`.Q.fc` parallel on cut
Parallel processing

🎓 Table counts in a partitioned database
🕹 *Q for Mortals* A.68 `peach`

Higher-rank values

`peach` applies only unary values. For a values of rank ≥2, use Apply to project `v` as a unary value.

For example, suppose `m` is a 4-column matrix and each row has values for the arguments of `v4`. Then `.[v4;]peach m` will apply `v4` to each list of arguments.

Alternatively, suppose `t` is a table in which columns `b`, `c`, and `a` are arguments of `v3`. Then `.[v3;]peach flip t`b`c`a` will apply `v3` to the arguments in each row of `t`.

⚠ Blocked within `peach`

```
hopen socket
websocket open
socket broadcast (25!x)
amending global variables
load master decryption key (-36!)
```

And any **system command** which might cause a change of global state.

Generally, do not use a **socket** within `peach`, unless it is encapsulated via one-shot sync request or HTTP client request (TLS/SSL support added in 4.1t 2023.11.10). Erroneous socket usage is blocked and signals a `nosocket` error.

If you are careful to manage your **file handles/file access** so that there is no parallel use of the same handle (or file) across threads, then you can open and close files within `peach`.

Streaming execute (`-11!`) should also be fine. However updates to global variables are not possible, so use cases might be quite restricted within `peach`.

raze

Return the items of `x` joined, collapsing one level of nesting

```
raze x    raze[x]
```

To collapse all levels, use Converge i.e. `raze/[x]`.

```
q)raze (1 2;3 4 5)
1 2 3 4 5
q)b:(1 2;(3 4;5 6);7;8)
q)raze b                / flatten one level
1
2
3 4
5 6
7
8
q)raze/[b]              / flatten all levels
1 2 3 4 5 6 7 8
q)raze 42               / atom returned as a list
,42
```

Returns the flattened values from a dictionary.

```
q)d:`q`w`e!(1 2;3 4;5 6)
q)value d
1 2
3 4
5 6
q)raze d
1 2 3 4 5 6
```

> **Use only on items that can be joined**

`raze` is the extension `,/` (Join Over) and requires items that can be joined together.

```
q)d:`a`b!(1 2;3 5)
q)10,d          / cannot join integer and dictionary
'type
q)raze (10;d)   / raze will not work
'type
```

Join