Iterators

FD Singapore Office

2019/10/17

kdb+ 3.6 2019.06.09

What is an iterator?

- Firstly, the iterators are what were previously known as adverbs. Since V2 of the Kx documentation, this naming convention was done away with. (So don't use it !!)
- According to code.kx.com: An iterator is a native higher-order operator, that takes applicable values as arguments and return derived functions. But what exactly does this mean?
- An applicable value is just something that we can index into or apply to arguments (functions). Note however, that this definition does not limit iterators to just functions. We will see this in action later.
- A derived function is just the result of applying an iterator to a applicable value e.g: `like/:` is a derived function.
- For right now though, we will just consider functions, and how we use the iterators with them.

There are six iterators in q, but note that different applications lead to different outputs:

Iterator	Name
`'`	Each/Each-Both/Case/Compose
`/:`	Each Right
`\:`	Each Left
`':`	Each Prior/Each Parallel
`/`	Over/Converge/Do/While
`\`	Scan/Converge/Do/While

' Each/Each-Both/Case/Compose

- When used with a unary value, `'` is known as each and can be replaced by the keyword `each`. It applies a value item-wise to a list, dictionary, table etc. Good q style prefers the use of `each` rather than a single apostrophe (Avoids the use of parentheses also).

```
q)L:("A";"presentation";"on";"iterators")
q)count each L    //Prefered style
1 12 2 9
q)count'[L]       //Avoid where possible
1 12 2 9
q)(count')L       //Avoid where possible
1 12 2 9
```

- Unfortunately, to apply a unary function at depth we do need to use the apostrophe.

```
q)count each'L
1
1 1 1 1 1 1 1 1 1 1 1 1
1 1
1 1 1 1 1 1 1 1 1
```

- When used with a binary value, `'` is known as each-both and has no q keyword equivalent. It applies a value pair-wise to lists, dictionaries, tables etc. When one arguement is an atom, it is extended to match the length of the second arguement.

```
q)"Aqns"in'L
1011b
q)"Aqns",'L
"AA"
"qpresentation"
"non"
"siterators"
q)"A",'L         // Atom "A" is extended
"AA"
"Apresentation"
"Aon"
"Aiterators"
```

- Higher order values (3<) are applied with brackets. Atoms again are extended.

```
q){x+y*z}'[1000000;1 0 1;5000 6000 7000]    // First arg here is extended
1005000 1000000 1007000
```

- `Case` is one iterator not very often seen in the wild. It is used to select succesive items from muliptle list arguments. The left argument determines from which of the arguments each item is picked.

```
q)1 0 1'["abc";"xyz"]
"xbz"
q)2 0 1'["a";"b";"cde"]    // Atoms still extended as neccesary
"cab"
```

- This is comparable to the vector conditional usage of `?` but case is not limited to just two lists. Also the left argument to case must be a an integer or long list whereas `?` accepts a list of booleans.

```
(1 0 1'["abc";"xyz"])~?[010b;"abc";"xyz"]
```

- It is not hard to see that case could also be used for conditonal updates to tables, allowing for more than two possible options:

```
q)t:([]pref:5?`home`office`mobile;home:5?100;office:5?100;mobile:5?100)
q)update call:(`home`office`mobile?pref)'[home;office;mobile] from t
pref   home office mobile call
------------------------------
office 42   52     85     52
home   73   39     89     73
mobile 74   84     23     23
home   18   68     60     18
mobile 37   70     6      6
```

- `Compose` is also not seen very often. It can be used to compose a unary function with other functions of rank greater than or equal to 1.

```
q)f:+
q)ff:2*
q)'[ff;f][9;3]    // Applies f followed by ff e.g. ff f[9;3]
24
```

- One of the more useful aspects of `compose` is alongside the repeated uses of the `each` keyword. For example, to test for files vs directories in your current directory:

```
q)type each key each hsym key`:.    // We can compose type and key here
-11 -11 -11 -11 11 -11 11 -11 -11 11 11 -11 11 11 11 11 11 11h
q)'[type;key]each hsym key`:.
-11 -11 -11 -11 11 -11 11 -11 -11 11 11 -11 11 11 11 11 11 11h
```

- In general, composition of eaches is going to be quicker than sequential. In my opinion though, it does come at the cost of readabilty, especially when composing more than two functions.

```
q)\ts:1000 type each key each hsym key`:.
437 2912
q)\ts:1000 '[type;key]each hsym key`:.      // One iteration rather than many
152 2656
q)('[;] over (f;g;…;h)) each v              // Composing more than two functions requires over
```

/: \: Each-Right and Each-Left

- Each-Right and Each-Left operate on binary values and return another binary value, pairing one argument to each item of the other. (Similar to each-both when one argument is an atom, but these extend to non-atomic arguments). Tip: To remember which is which, just look at the direction the slash is leaning.

```
q)3 in/:(0 1 2;0 2 4;0 3 6)
001b
q)3 in'(0 1 2;0 2 4;0 3 6)
001b
q)0 1 2 in/:(0 1 2;0 2 4;0 3 6)     // Note each of 0,1,2 is compared against each on the right
111b
101b
100b
q)(0 1 2;0 2 4;0 3 6)?\:0 2         // Granular on the right (reflects ? behaviour)
0 2
0 1
0 3
q)"string"like/:("str*";"str???";"match";"foo*")      // Non atomic left arguement
1100b
q)` sv/:`trade`order,\:`csv        // Creating filenames using each right and each left
`trade.csv`order.csv
```

- We can, of course, combine iterators on the same value to produce further derived functions. Furthering an example from above, we are comparing each string on the left to each string on the right using a combination of both the each-right and each-left iterators. (Each both can also be used in conjuction with each right and each left on different functions.)

```
q)("string";"foo")like\:/:("str*";"str???";"match";"foo*")
10b
10b
00b
01b
```

- Each-right, each left and `join` can be used to produce a result similar to that of `cross`

```
q)show a:{x,/:\:x}til 2
0 0 0 1
1 0 1 1
q)show b:{x cross x}til 2
0 0
0 1
1 0
1 1
q)b~raze a
1b
```

`':` Each-Prior/Each-Parallel

- When used on a binary value, `':` is known as each prior and its keyword is `prior`. It applies the binary to each item in a list and to the previous item (i.e each adjacent pair)

```
q)show L:10?20
4 13 9 2 7 0 17 14 9 18
q)(-':)L
4 9 -4 -7 5 -7 17 -3 -5 9
q)(-)prior L                      // Good q style prefers the use of the prior keyword
4 9 -4 -7 5 -7 17 -3 -5 9
q)deltas L
4 9 -4 -7 5 -7 17 -3 -5 9
```

- Some uses of `':` are so common they have they their own keywords, e.g. `deltas`, `ratios`, `differ`.
- Since each prior applies it's value pairwise, a reasonable question to ask is what about the first item of the list, what is that paired with? Without going into too much detail, it is paired with the identity element of the binary (.i.e for a function f the value x such that y~f[x;y] for any y). In this case, the identity element is 0, since subtracting zero does nothing.

- We do however, have the option to specify what the intial value is using `deltas`. (Again, without going into too much detail, each-prior derives a variadic function, so it can accept either one or two arguments)

```
q)deltas[1;L]
3 9 -4 -7 5 -7 17 -3 -5 9
```

- When used on a unary value `':` is known as peach and it's keyword is also `peach`. Peach applies it's value to each item of it's argument (similar to `each`). Peach, however partitions its work between any available slave processes. Again, good style favours the use of the `peach` keyword. For example, say we have a computationally heavy function and a q process running a number of slaves:

```
$ q -s 4 -q
\ts { sum (x?1.0) xexp 1.7 } each 6#1000000
528 16778384
\ts { sum (x?1.0) xexp 1.7 } peach 6#1000000
276 2144
```

- The advantage here is obvious, however it is important to note that for small computations, `peach` can be more expensive, incurring overheard with serialization between slave processes. This overhead can be estimated by manually serialising and deserialising the argument. e.g `\t -9!-8!x`

```
$ q -s 4 -q
\ts:1000 { x*1?10 } each 10?1.0
9 1840
\ts:1000 { x*1?10 } peach 10?1.0
56 1840
```

/ \ Over/Scan/Converge/Do/While/

- The first thing to note here is that both `/` and `\` have the same syntax and perform the same computation, the only difference being that `/` only returns the last value of the computation, whereas `\` returns all intermediate values.
- We first consider a binary values. It is only in operating on a binary value that `/` and `\` are known better as the keywords `over` and `scan`. The number of evaluations is the count of the right argument. Both `over` and `scan` return a variadic function which affects how the computation is performed.
- For example, consider `+\`. `+` is our binary value and `\` is the iterator. This function is variadic, i.e. we can pass either 1 or 2 arguments. When we pass 2 arguments (binary), the first evaluation applies `+` to the left argument and the first item of the right argument. This result becomes the left argument of the next evaluation and the second item of the right argument becomes the new right argument to the second evaluation.

```
q)10+\1 2 3 4    // (((10+1)+2)+3)+4
11 13 16 20
```

- Remember that `+\` is variadic, so we can also pass it 1 argument. In this case, we can think of just applying the binary between successive values. (Technically, q uses identity elements again here but for sake of brevity I'll omit details). Note the neccesary parentheses when passing a single argument.

```
q)(+\)1 2 3 4    // 1+2+3+4
1 3 6 10
```

- Personally, I don't like the use of `over` and `scan`. They limit what should be variadic functions back down to unary functions and also can be ambiguous when used incorrectly e.g.:

```
q)10(+)over 1 2 3
'Cannot write to handle 10. OS reports: Bad file descriptor
  [0]  10(+)over 1 2 3
         ^
q)(+)over[10;1 2 3]
'Cannot write to handle 10. OS reports: Bad file descriptor
  [0]  (+)over[10;1 2 3]
            ^
q)over[10+;1 2 3]      // Don't run - infinite loop (See appendix)
q)over[+[10];1 2 3]    // Also don't run, same as above
q)not scan 0b          // To be seen later in the show
01b
```

- Ternary values and higher are similar to binary. In this case the arguments must be atoms or have a matching count. The first evaluation is applied to the first argument, and the first items of remaining arguments. The result then becomes the first argument to the next iteration.

```
q)ssr\[string .z.p;".:";"-."]
"2019-10-07D09:47:43-367643000"
"2019-10-07D09.47.43-367643000"
```

- Finally we deal with unary values. We have three different methods of applying `/` and `\` to a unary value `f`. Again, since `/` and `\` derive variadic functions, we can call them with one or two parameters.

```
(f/)x     // Converge   (Can also be called as f/[x])
x f/y     // Do (where x is a non-negative integer)
x f/y     // While (where x is a function returning 1b or 0b)
```

- Converge: Runs until the result converges (within comparision tolerance or we reach the original argument). Note it is very easy to throw q into an infinite loop here.

```
q)(not\)0b      // From above, this is actually what not scan 0b is doing
01b
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)(not/)42      // Never returns
```

- As mentioned earlier, iterators are not just applicable to functions. We can also use them on dictionaries, lists, tables etc. For example, given a dictionary of pubs in Singapore and the route taken from pub to pub:

```
q)show route:`muddys`dallas`bqBar`magumbos`heros`towers!`dallas`bqBar`magumbos`heros`towers`muddys
muddys  | dallas
dallas  | bqBar
bqBar   | magumbos
magumbos| heros
heros   | towers
towers  | muddys
q)(route\)`muddys    // Stops when we return to the original argument (or pub)
`muddys`dallas`bqBar`magumbos`heros`towers
```

- Do: Runs the specified number of times with each successive result becoming the argument in the next evaluation. Note the first evaluation always just returns the initial argument.

```
q)2(2*)\5 7
5  7
10 14
20 28
q)10{x,sum -2#x}/1 1               // Generate the first 10 Fibonnaci numbers
1 1 2 3 5 8 13 21 34 55 89 144
q)2 route\`muddys                  // For anyone that wants to take it easy...
`muddys`dallas`bqBar
```

- While: Runs while the truth function returns `1b`. The function is called with the return value of each iteration. Note that execution returns the value that causes the truth function to return `0b`.

```
q)(100>)(2*)\2
2 4 8 16 32 64 128
q)(8>count@){x,sum -2#x}/1 1       // While the number of fibonacci numbers is less than 8
1 1 2 3 5 8 13 21
q)(`magumbos<>)route\`muddys       // For anyone that promises they're not going to Heroes tonight...
`muddys`dallas`bqBar`magumbos
```

- Note, the examples with the route contain no functions. (And no sign of `do` or `while` loops either. See Appendix)

Putting it all together

- Here are a few examples of iterators at work. Here we first select the columns from a table of type 0 and then iterate through them, casting each to an empty string. Useful before writing a table to disk. This uses `/` on a binary value. The number of iterations is the number of columns found in the table.

```
q)c:exec c from meta[tab] where t in" ";
q)tab:{![x;enlist(~\:;y;());0b;(enlist y)!enlist(#;(count;`i);(enlist;""))]}/[tab;c]
```

- Pascal's triangle. We can use iterators to efficiently generate a common mathematical structure called Pascal's triangle. Each row is generated by summing adjacent pairs in the row above, easily achieved using both `prior` and `do` here. (Aside, this can then be used to calculate large combinatorics in q then. See Appendix)

```
q)4{(+)prior x,0}\1      // First 5 rows
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

- Given a list of prices, how might you find the last non null price? The obvious answer is:

```
q)show L:(9?0N,1+til 5),0N
1 0N 0N 2 1 2 3 4 2 0N
q)L last where not null L
2
```

- But we can produce a more efficient answer using an iterator. Recall the `fills` keyword, which will replace any nulls in a list with the last non-null entry:

```
q)fills L                // So we just want the last entry here
1 1 1 2 1 2 3 4 2 2
q)last fills L
2
q)fills                  // But fills is just a derived function using \, which returns all intermediate results....
^\
q)^/[L]
2
```

- A useful trick for generating successive arguments to iterative functions such as `\` and `/` is:

```
q)(;;)\[`green;1 2 3;4 5 6]     // Generates lists of successive args
`green              1 4
(`green;1;4)        2 5
((`green;1;4);2;5) 3 6
```

References

- https://code.kx.com/v2/ref/iterators/
- https://code.kx.com/v2/ref/maps/
- https://code.kx.com/v2/ref/accumulators
- https://code.kx.com/v2/basics/iteration/
- https://code.kx.com/v2/wp/iterators/
- https://code.kx.com/v2/basics/glossary/

Appendix

- Infinite loops with `over`. We saw that `over[10+;1 2 3]` throws q into an infinite loop. Why? Well in this case our first argument is a projection of `+` which is unary. The `over` keyword applies `/` to that, and derives a variadic function. We then pass this a single argument `1 2 3` which is parsed as converge. q executes `10+1 2 3` and returns this to the next evaluation, `10+11 12 13` and so on. Obviously, this will never converge.

```
q)over
k){x/y}
q)((10+)/)1 2 3
1 2 3
11 12 13
...
```

- Avoid using control words `while` and `do`. Don't write loopy code.

```
q)r:(),`muddys
q)while[`magumbos<>last r;r,:route last r]         // Please don't do this
q)r
`muddys`dallas`bqBar`magumbos
q)\ts:100000 r:(),`muddys;while[`magumbos<>last r;r,:route last r];r
429 2640
q)\ts:100000 (`magumbos<>)route/`muddys
144 1184
```

- Calculating large combinatorics in q is not possible with a brute force method sometimes. For those of a maths background, this is 'N choose R' for which the equation in q would be:

```
q)fac:prd 1+til@
q)choose:{[n;r]fac[n]%fac[r]*fac[n-r]}
q)choose[4;2]
6f
q)choose[10;2]
45f
q)choose[20;3]
1140f
q)choose[25;3]          // Doesn't look right....
-0.9374425
q)fac[21]               // The issue is with large factorials
-4249290049419214848
```

- To get around this, we can index into Pascals triangle. This method is also faster as there is no calculation inolved in each call, only indexing.

```
q)pt:50{(+)prior x,0}\1           // Take 51 rows
q)pt[4;2]
6
q)pt[10;2]
45
q)pt[20;3]
1140
q)pt[25;3]
2300
q)pt[35;10]
183579396
```