06_Functions

September 22, 2019

1 6. FUNCTIONS

List of q built-in functions

1.1 6.1. Function specification

- Because a q function can **modify global variables**, q is not a pure functional language.
- Function definition notation: {[param1, param2, ...]expr1;expr2;...}
- if only up to 3 parameters is used x, y, z, you do not have to declare them
- function name is optional: lambda or anonymous function
  - definition {x*x}[3] -> returns 9
- no return statement: the function returns the last expression evaluated
  - finish the function definition with a semi-colon to return nil (::)
  - ; is an opererator that evaluates its left operand and discards its value then evaluates its right operand and returns its value
  - : (return) in a function body terminates the execution of the function early and returns the expression to its right
- a function is a first class value
- a function can be assigned to a variable
- expressions in a function are evaluated from left to right
- valence (rank): number of parameters in a function
  - maximum valence permitted: 8 parameters
    - to exceed this: encapsulate multiple parameters in a list or dictionary
  - niladic function: function with no parameters

[ ]:
```
too_many_params:{[q;w;e;r;t;z;u;o;l] (q;w;e;r;t;z;u;o;l)}
```

Keep it short - Q functions should be compact and modular: - each function should perform a well-defined unit of work. - Due to the power of q operators and built-in functions, many functions are one line. - When a function exceeds 20 statements, you should look to refactor it.

q is NOT ambivalent depending on their valence: - meaning - q operators and built-in functions are **not overloaded on valence**, meaning that the same function does not have different behavior in a monadic and dyadic form. HOWEVER:

q IS ambivalent in terms of several other aspectcts: - meaining: - q operators and built-in functions **are overloaded on**: - type, - sign, - rank of arguments, - etc. - sometimes multiply so.

1.1.1 6.1.3. Function application (function call)

- function application is strict: expressions in arguments are evaluated first, substitution happens only afterwards
- method 1:
  - func[arg1;arg2;...;arg8]
- method 2: with juxtaposition
  - func arg1 arg2 ...
- method 3: call function by symbolic name:
  - ‘func arg1 ...

1.1.2 6.1.4. Functions with no return value

- notation: void_function:{expr;} - ends with a semi-colon
- returns the nil item: ::
- purpose of these functions:
  - for only side effects: sending an asynchronous message; updating a global variable

1.1.3 6.1.7. Implicit parameters

- **x, y, z** are the 3 positional parameters defined implicitly inside every function, you can use them without declaring them as parameters

1.1.4 6.1.8 Anonymous functions and Lambda expressions

- All q functions are anonymous functions (or lambda expression or lambdas) by defalut
  - until assigned to a variable
- Use cases:
  - in-line macro????
  - **dynamic dispatch**: functions are placed in a collection (e.g., list, dictionary, table) and then selected dynamically (i.e., at runtime) rather than by an intrinsic name

[ ]:
```
powers:({1}; {x}; {x*x}; {x*x*x})
powers[3;2]
```

1.1.5 6.1.9. The identity function ::

- an identity function returns its input as its output
- notation: ::[1;32;3] -> returns 1 32 3

1.1.6 6.1.10 Functions are data

- data categories in q:
  - atoms

  - lists
  - dictionaries
  - tables
  - **FUNCTIONS**
    * they can be passed around as any other data categories, e.g. as an item in a list or dictionary
    * functions can be arguments of another (higher-order) function or
    * return values

[ ]:
```
apply:{x y}
sq:{x*x}
apply[sq; 5]
```

1.2 6.2. Call-by-name

q functions are call by value - arguments are reduced to values and copies are made

[ ]:
```
a:5
func1:{x-1}
a
func1 a
a
```

except when the variable size is too large - names of global variables are passed instead of their values - q kernel handles arguments automatically and decides to call them as values or names - built-in functions using call-by-name of global variables: - **get** - **set** - The result of any built-in call-by-name function application is the **symbol** that was passed as input. - This enables call-by-name functions to be chained – i.e., composed.

[ ]:
```
gv:323
get `gv / the symbolic name should be used as parameter
`gv2 set 101 / a symbolic name should be used as the name of the variable
gv2
```

1.3 6.3. Local and global variables

- **Local variable**: assigned with : within a function
- 24 local variables are allowed within a function
- A local variable exists only for the duration of an application. There is no notion of a static local variable that maintains its value across applications.
- A local variable is not visible outside its immediate scope of definition.
- A local variable cannot be used as an argument to a q function that uses call-by-name.
- q does not have lexical scoping: A local variable is not visible within the body of a local function defined in the same scope
  - use projection instead:

[ ]:
```
f:{[p1] a:42; helper:{[a; p2] a*p2}[a;]; helper p1}
f 2
```

[ ]:
```
f:{[p1] a:42; helper:{[a;p2] a*p2}[a;p1]}
f 2
```

Global variable - every variable assigned outside of any function definition is a global variable - global variables are visible inside any function body - 32 is the maximum number of global variables that can be referenced inside a function - to circumvent this, use lists or dictionaries (dictionaries provide pass by name) - you can modify or declare a global variable from inside a function - ‘a set 42 - you can use ::, but it only works for declaring a global variable. if there already exists a global variable, a local variable is created. avoid this usage as it might be confusing if we accidentally choose a name that is already declared.

1.4 6.4. Projection

- **Projection**: specifying only some of the parameters in function application, which results a function with the remaining parameters

1.4.1 6.4.1. Function projection

- projecting a function onto the remaining arguments means: when a portion of arguments in a function is fixed and the remaining ones vary, you can do that to specify only the fixed arguments and leave the specification of the varying ones for later. (like the .bind() method in js)
- notationally, projection is a partial function application, in which some arguments are supplied (the fixed one) and others (the varying ones) are omitted
- the result is a partial function application
- you can assign the partial application of a function to a variable and then use it as a monadic function in case of the example
- projection is related to *currying*
  - Both provide realizations of the mathematical observation that **a function of two parameters is equivalent to a function of the first parameter that returns a function of the second parameter**.

[ ]:
```
add:{x+y}
add[42;] / partial application
add[42;][3] / projected application
pa:add[42;]
pa[3]
```

- projecting out successive parameters:

[ ]:
```
add3:{x+y+z}
add3[2;;][3;][4] / is the same as:
```

```
add3[2][3][4] / do not use this notation as it obscures original intent of the␣
 ↪code
9
```

!!! **important** - The function body in a projection is captured at the time it is first encountered by the interpreter. - Thus if the underlying function is subsequently changed, **the projection is not**.

1.4.2 6.4.2. Operator projection

- a dyadic q operator can be projected by fixing one of its operand
  - you can do the in an infix and a prefix form
    - infix form using parenthesis (only the left one can be fixed due to ambiguity): (7*)[6]
    - prefix form using square brackets (either one of the operands can be fixed): *[7;][6]* *or* [;6][7]

[ ]:
```
7*6
(7*)[6] / or (7*) 6
mult:(7*)
mult 6
```

[ ]:
```
*[7;6] / using the multiplication operator as a function
*[7;][6] / multiplication operator projected onto its second operator by fixing␣
 ↪the first one
*[;6][7] / multiplication operator projected onto its first operator by fixing␣
 ↪the second one
```

1.4.3 6.4.3. Multiple projections

- mutliple projections can be applied to functions having more than two parameters
- these functions can be projected in multiple ways
  - in one step: provide all arguments and omit only one, and project the function to its omitted parameter
  - in multiple steps: provide the arguments to be fixed one by one in separate steps until you reach the that you want your function to be projected onto

[ ]:
```
{x+y+z}[1;;3]
{x+y+z}[1;;3] 2 / project in one step
```

[ ]:
```
 / project in two steps
{x+y+z}[1;;] / first step
{x+y+z}[1;;][;3] / second step
{x+y+z}[1;;][;3] 2 / call the function with the varying parameter
```

1.5 6.5. Everything is a map - the relationship between q data structures and functions

1.5.1 6.5.1. Similarity of notation

- a list is a map: defined by positional retreival via item indexing. In a list of non-negative integer numbers, i.e. the indices are the keys and the items are the values
- a dictionary is a map defined by a key-value lookup
- a function is a map defined by a sequence of expressions representing an algorithm for computing an output value from the input:
  - in a function the output values are mapped to their input values (arguments are the key, the return value is the value)

1.5.2 6.5.2 List of Indices, Keys and Arguments

- that is why you can use a list of indices to get several items in a list or
- a list of keys to get a list of values from a dictionary or
- a list of arguments to get a list of return values from a function
- examples:

[ ]:
```
L:10 20 30 40 50
L[2 4]
d:`a`b`c!10 20 30
d[`a`c]
f:{x*x}
f[2 5]
```

- parameter lists are examples of compositions of maps:

[ ]:
```
/
  I     L
--------------
0 |-> 2 |-> 4
1 |-> 5 |-> 25

  ks     d
---------------
0 |-> `a |-> 10
1 |-> `c |-> 30

  I     f
--------------
0 |-> 2 |-> 4
1 |-> 5 |-> 25
\
```

1.5.3 6.5.2. Indexing at depth

- the relationship between indexing at depth and multivalent function evaluation
- a nested list:
  - notationally, is a lists of lists: nested_list[0][4][2]
  - functionally, can be viewd as a multivariate positional retreival: nested_list[0;4;2]
- a dictionary with complex values can also be viewed:
  - notationally: complex_dict["some_key"]["some_other_key"]
  - functionally as a multivaruate map: complex_dict["some_key";"some_other_key"]

1.5.4 6.5.3. Projection and index elision

- the notation for function projection and elided indices in a list are identical
- in a nested list (or a complex dictionary) the first index is go down (choosing from the row indices) and the second one is go over (choosing from the column indices)
  - ordinary positional retreival: if we omit the first index (the row index), we get all the elements in the specified column

1.5.5 6.5.4. Out of bounds index

- the behaviour of item indexing in case of out of bounds indices is also motivated by viewing lists as maps
- in traditional languages out of bound indexing results in an error or an exception:
  - in Python: IndexError
  - in Java: ArrayIndexOutOfBoundsException (for arrays) or IndexOutOfBoundsException (for lists)
- in q, a list is viewed as a map defined on a sub-domain of integers
  - but the domain of the (indexing?) function is extended to all integers by assigning null output values outside the defined domain, where null values mean missing values
- the behavior of dictionaries are completely analogous

1.6 6.6. Atomic functions

- q is a vector language
- this partially comes from the fact that all built-in operations are atomic:
  - atomic functions recurse into an arguments structure until it gets to atoms and apply there
  - atomic functions do not use loops or other contol flow constructs

1.6.1 6.6.1. Monadic atomic functions and "map"

- the output conforms to the input: the result of the application of a monadic atomic function always has the same shape as its argument
- when applying an atomic function to a list is the same as applying it to all of its elements
  - in traditional languages this is achieved with foreach or

  - in functional languages with the map() function

1.6.2 6.6.2. Dyadic atomic functions and "zip"

- a dyadic function can be atomic in either one or both of its arguments
  - atomic in one argument: fix the non-atomic argument, project on the atomic one. this can be thought of as a monadic atomic function
  - example: the find function (?) is atomic only its second argument, since it consumes the entire first argument in its search
    - (x?y means search for item y in list x and return the index of the item)

[ ]:
```
10 20 30 10?10
```

- the arithmetic, comparison and relation operators are all atomic in both of their operands, which results in 4 cases of application:
  - atom with atom
  - atom with list
  - list with atom (same as atom with list?)
  - list with list (only works with lists of the same length)
- in traditional programming, "zip" results in similar behavior (zip() function in Python)

[ ]:
```
list1:1 2 3 4 5
list2:10 20 30 40 50
int1:1
int2:2
```

[ ]:
```
int1+int2
int1+list1
list1+int1
list1+list2
```

1.6.3 6.6.3. Creating atomic functions

- The composition of atomic functions is atomic. Hence,
  - one way to build custom atomic functions is to **compose built-in atomic functions**. First monadic.
  - (another way is to create advers???)

1.7 6.7. Adverbs

- Adverbs are higher-order functions that modify the behavior of functions (operators) for application on lists.
  - The terminology derives from thinking of q operators as verbs.
- Use case: **adverbs are used to apply non-atomic functions on lists without loops**
- Proficiency in the use of adverbs is one skill that separates q pretenders from q contenders.

1.7.1 6.7.1. Monadic each

- examples for non-atomic functions:
  - aggregate functions: count, sum, min, max, avg
    - and their atomic counterparts: count each, sums, mins, maxs, avgs
  - other non-atomic functions: reverse, enlist
- each is a general higher order function, which can be used as map() functions in functional languages:
  - reverse each a_list
  - enlist each a_list: creates a matrix from a list where each item becomes a singleton list in the matrix
    - flip enlist a_list does the same only faster

[ ]:
```
show nlist:(1 2 3 4;10 20)
show nnlist:((101;102;(103;104));nlist)
```

[ ]:
```
count each nlist
each[count;nlist] / this reveals how each is similar to map() functions in␣
↪functional languages
```

[ ]:
```
count nnlist
count each nnlist
(count each) each nnlist
```

[ ]:
```
freq:(1;4;12;32;36;10;3;2)
sums(freq)
```

[ ]:
```
freq2:(5;23;44;113;134;40;25;2)
freq2
100 * sums(freq2) % (sum freq2)
(`int$10000 * sums(freq2) % (sum freq2)) % 100
```

1.7.2 6.7.2. Each both '

- the dyadic each-both modifies a dyadic function (operator, verb) to apply pair-wise to corresponding items in the lists
- functions modified by the each both modifier has the same properties as dyadic atomic functions
  - throws a length error if the two arguments do not line up, but
  - extends an atom to match a list
  - however, it does not always work as expected on general lists
    - to make a list of pairs from a pair of lists: flip (L1;L2)
- applying each both on tables (with the same number of records), results a sideways join: the joined table will contain columns from both tables and will have the same number of rows

[ ]:
```
strl1:("abc"; "uv")
strl2:("de"; "xyz")
```

```
strl1,strl2
```

```
strl1,'strl2
```

```
,'[strl1;strl2] / in a functional form
```

```
2#("abcde"; "fgh"; "ijklm")
2#'("abcde"; "fgh"; "ijklm")
```

```
show L1:(enlist `a; `b)
L2:1 2
newL:L1,'L2
newL
```

```
flip (L1;L2)
type flip (L1;L2)
```

1.7.3 6.7.3. Each left \:

- The each left adverb modifies a dyadic funcition so that it applies the second argument with each item of the first argument

```
somestr:("abc"; "de"; "f")
```

```
somestr ,\: ">" / the left operand is always considered to be an atom even if␣
↪it is a list
```

```
somestr,' " />" / the second argument is taken as a list based on its type
```

1.7.4 6.7.4. Each right /:

- The each right adverb modifies a dyadic function so that to apply the first argument with each item of the second argument

```
"<",/:somestr
```

```
"<",'somestr
```

- combining the each right and each both

```
"<",/:somestr,\:" />"
```

1.7.5 6.7.5. Cross product

- A cross product of two lists pairs each item on the left with each item on the right

  - it is an adverb of join composed with each left and each right (then razed to remove a level of nesting)
  - notation: cross, which can be written as raze list1,/::list2

[ ]:
```
pre_cross:1 2 3,/:\:10 20
```

[ ]:
```
pre_cross
```

[ ]:
```
((count each) each) each pre_cross
```

[ ]:
```
pre_cross[0;0;0]
```

[ ]:
```
cross_pr:raze 1 2 3,/:\:10 20
```

[ ]:
```
cross_pr
```

[ ]:
```
cross_pr[0]
```

1.7.6 6.7.6 Over (/) for accumulation

- In functional programming languages, over is also called *fold* or *reduce*
  - this is the other half of the *map-reduce* paradigm (each-over)
- The over adverb is a higher-order function that provides the principal mechanism for recursion in q
- The over adverb, in its simplest form, modifies a dyadic function to accumulate results over a list

[ ]:
```
0 +/ til 11
```

[ ]:
```
imp_sum:{[x;y] 0N!(x;y); x+y} / the + operator expressed in an imperative way
```

[ ]:
```
imp_sum[2;3]
```

[ ]:
```
0 imp_sum/til 11
```

- To omit the unnecessary initial value, put the adverb in a paranthesis and make it a monadic function:
  - for summing the items in a list: (+/) this is an adverb in k programming languages
  - in q (+/) is equal to **sum**
  - (|/): maximum
  - (&/): minimum
  - (*/): prd

[ ]:
```
(+/) til 11 / same as above only in a monadic form
```

[ ]:
```
L1:1 2 3 4 5 77
L2: 54 32 64 11 14
```

[ ]:
```
max[L1,L2]
```

[ ]:
```
max[L1]|max[L2]
```

- ,/ (join over) applied over a list concatenates the items eliminating the top level of nesting

[ ]:
```
(,/)((1 2 3; 4 5); (100 200; 300 400 500))
raze ((1 2 3; 4 5); (100 200; 300 400 500))
```

- custum functions can be used with over to make them atomic?
  - they can be used infix too

1.7.7 6.7.7. Iteration

- Another pattern of recursion without loops:
  - number of iterations function/ arguments to the function

[ ]:
```
fib_elem:{x,sum -2#x}
```

[ ]:
```
fib_elem 1 1
```

[ ]:
```
10 fib_elem/1 1 / infix
fib_elem/[10, 1 1] / in a prefix form
```

- Yet another version of over runs the recursion until convergence, or until a loop is detected.
- How does q determine convergence?
  - At each step the result is compared to that of the previous step.
  - If the two agree within equality tolerance (10-14 at the time of this writing – Sep 2015) the algorithm is considered to have converged and the recursion is complete; otherwise it continues.
- How did q detect the cycle and stop the iteration?
  - At each step it compares the result with the initial value.
  - If they match, a cycle has occurred and the recursion is halted; otherwise it continues.
    - Note that we mean "match" literally as in the ~ operator.

While loop - the final overload of the over adverb is the equivalent of the while loop - It provides a declarative way to specify a test to end the iteration. - Thinking functionally, we provide a predicate function that is applied to the result at each step. The iteration continues as long as the predicate result is 1b and stops otherwise. For example, we stop the computation of the Fibonacci sequence once it exceeds 1000 as follows.

[ ]:
```
fib:{x,sum -2#x}
fib/[{1000>last x}; 1 1]
```

1.7.8 6.7.8 Scan \

- The scan adverb is a higher-order function that behaves just like over except it returns all the intermediate accumulations instead of just the final one
  - Scan returns an output that has the same number of items as its input

[ ]:
```
f:{-2+x*x}
```

[ ]:
```
0+\1 2 3 4 5 6 7 8 9 10
(*\)1 2 3 4 5 6 7 8 9 10
(|\)7 8 4 3 10 2 1 9 5 6
(&\)7 8 4 3 10 2 1 9 5 6
10 f\1 2 3 4 5 6 7 8 9 10
(f\)1 2 3 4 5 6 7 8 9 10
```

1.7.9 6.7.9. Each previous (':)

- the adverb each previous provides a declarative way to provide a dyadic operation on each item of a list with its predecessor
- in a dyadic form, the left operand is the initializer value:

[ ]:
```
5 -':5 4 3 2 1
```

- in a monadic form, there is no initializer value, it returns the first element unaltered:

[ ]:
```
(-':)9 8 7 6 5 4
```

[ ]:
```
deltas 9 8 7 6 5 4
```

1.7.10 SUMMARY

- each
- each both (')
  - joint: (,')
- each left (:)
- each right (/:)
- over (/)
  - (+/) sum
- scan (\)
  - (+\) sums
- each previous (':)
  - (-':) deltas
  - (~':) differ

[ ]:
```
(~':) 1 1 1 2 2 3 4 5 5 5 6 6
not (~':) 1 1 1 2 2 3 4 5 5 5 6 6
```

```
differ 1 1 1 2 2 3 4 5 5 5 6 6
```

- An idiom with differ: to cut a list along its identical values

[ ]:
```
L:1 1 1 2 2 3 4 5 5 5 6 6
differ L / list of booleans where item differs from previous one
where differ L / indices of elements which are different from their predecessors
(where differ L) cut L / cut original list where different items start
```

[ ]:
```
show runs:(where differ L) cut L / store runs
show ct:count each runs / store count of each run
runs where ct=max ct / find the runs of maximum length
```

- find runs of increasing or decreasing integer values

[ ]:
```
L:9 8 7 11 10 12 13
(where -0W>':L) cut L
(where 0W<':L) cut L
```

1.8 6.8. General application (of functions)

- The syntacic forms, square brackets and juxtaposition (used with list indexing, key lookup and function application) are both syntactic sugars
- **q actually treats function application as a higher-order function that takes a function and value(s) and evaluates the function at the values**
- the simpler forms of general application are usually written infix and considered to be verbs

[ ]:
```
fsq:{x*x}
```

[ ]:
```
fsq[2 4]
```

1.8.1 6.8.1. The @ verb

- univalent mathematical mapping:
  - in case of lists: retreive a list item by index
  - dictionaries: looking up a value by a key
  - functions: evaluating a monadic function
    - @ applies a monadic mapping to an argument
- both infix and prefix notation is acceptable
- applying a niladic function with @:
  - niladic@(::)
  - @[niladic;::]

[ ]:
```
GL: 10 20 30 40 50
```

[ ]:
```
GL@1
```

[ ]:
```
@[GL;3]
```

[ ]:
```
gd:`a`b`c!11 22 33
```

[ ]:
```
gd@`b
```

[ ]:
```
@[gd;`a]
```

[ ]:
```
gf:{x*x}
```

[ ]:
```
gf@3
```

[ ]:
```
@[gf;6]
```

- general application with a non-atomic argument
  - for lists, dictionaries and atomic functions, @ yields an output that conforms to the shape of the input
  - an aggregate function collapses the top level of nesting from the input
  - a uniform function has the same length of output as the input list

[ ]:
```
t:([]c1:1 2 3; c2:`a`b`c)
t@1
d:`a`b`c!10 20 30
d@`b
kt:([k:`a`b`c] v:1.1 2.2 3.3)
kt@`c
```

6.8.2. Verb Dot .

- Indexing a list at depth, retrieving a nested value from a dictionary and evaluating a function with multiple parameters are all instances of applying a **multi-variate mapping**.
- The higher-order function . is the true form of multi-variate application in q.
- It applies a multi-variate mapping to multiple arguments and can be written infix or prefix.

[ ]:
```
gln:((2 3 4);(6 7 9))
```

[ ]:
```
gln[0][1]
```

[ ]:
```
gln[0;1]
```

[ ]:
```
gln . 0 1
```

[ ]:
```
gcd:`a`b`c!(10 20 30; 40 50; enlist 60)
```

[ ]:
```
gcd . (`c;0)
```

[ ]:
```
gmvf:{x*y}
```

- the dot verb allows us to apply a multi-variate function to a list of arguments instead of multiple individual arguments
  - this means that the function is defined on a vector instead of its indivudial components
  - use case: dynamic dispatch: where we cannot know the valence of the function in advance
    - similar to *args in Python and ... in Java
- you can apply a monadic function with the dot verb, only you have to enlist its only argument to be a singleton vector
  - thus, the dot verb makes the @ verb redundant in q

[ ]:
```
gmvf . 33 22
```

- to denote an elided index with ., use nil instead of empty index

[ ]:
```
gln . (::;1)
```

[ ]:
```
GL . enlist 0
```

[ ]:
```
t:([]c1:1 2 3;c2:`a`b`c)
t . (1; `c2)
```

[ ]:
```
kt:([k:`a`b`c] v:1.1 2.2 3.3)
kt . `b`v
```

[ ]:
```
kt
```

- indexing at depth with a table with nested field values

[ ]:
```
t:([] c1:`a`b`c; c2:(1 2; 100 200 400; enlist 1000))
t . (1; `c2)
t . (1; `c2; 1)
```

1.8.2 6.8.3. General form of verb application

- dot and @ verbs written in prefix form:

[ ]:
```
L:10 20 30 40 50
@[L; 1]
@[L; 0 2]
m:(10 20 30; 100 200 300)
.[m;0 2] / note the lack of semi-colons between the two indices
```

1.8.3 6.8.4. General apply (@) with monadic functions

- general apply: applies a function on the sub-domain of the data structure
- notation: @[data_structure;indices;function_to_apply]
- this allows the result to be chained into further operations

- general application of @ accurs along a sub-domain at the top-level (HOW TO REACH DOWN?)
- The syntax for general application of a monadic atomic function on a list is,
  - @[L;I;f]
  - where L is the list and I is a collection of indices into L and f is the function to apply on I sub-domain of L.
- Viewing L as a mapping, I is a top-level sub-domain of L. In fact, this form generalizes to any data structure viewed as a mapping.

[ ]:
```
L:10 20 30 40 50
```

[ ]:
```
@[L;2 4;neg] / general application returns the entire input data structure with␣
↪the specified item changed
```

[ ]:
```
neg L@2 / normal applicaton only returns the selected item
```

- to modify the original data structure in-place, use pass-by-name

[ ]:
```
L
```

[ ]:
```
@[`L;2;neg]
```

[ ]:
```
L
```

1.8.4 6.8.5. General apply (@) dyadic functions

- the shape of the supplied operand must conform to the specified sub-domain of the input data
- The general form of functional @ for a dyadic atomic function is:
  - @[L; I; g; v]
  - where I is a top-level sub-domain of L; g is a dyadic function; and v is an atom or list conforming to I.
- example use case: assigning multiple values to multiple items in a list
- in-place modification works the same as with the genral apply on monadic functions

[ ]:
```
@[L;1 4;+; 100 200]
```

1.8.5 6.8.6. General apply (.) for monadic functions

- In contrast with @, the vector argument of . reaches down into the data structure and picks out a single point in the domain. Here we target that point with a monadic function.
- The general form of . for monadic functions is:
  - .[L; I; f]
  - Here L is a data structure, I is an in-depth sub-domain of L and f is a monadic atomic function.

[ ]:
```
m:(10 20 30; 100 200 300)
.[m;(0;1)]
.[m;(0;::)] / using nil, we apply the function on all items at that level
.[m;(0;::);neg]
d:`a`b`c!(10 20 30; 40 50; enlist 60)
.[d; (`a; 1)]
```

1.8.6 6.8.7. General apply (.) for dyadic functions

[93]:
```
show mm:(10 20 30 40;100 200 300 400 500)
```

```
10 20 30 40
100 200 300 400 500
```

[95]:
```
.[mm;(1;::);+;1 2 3 4 5]
```

[95]:
```
10 20 30 40
101 202 303 404 505
```