Shakti tutorial

John Estrada

15 April 2023

1 Introduction

The k9 programming language is designed primarily for the analysis of data. It may surprise new users with two rather different paradigms, (1) fast data analysis and (2) concise syntax. After you become familiar with the language, these features will both seem normal and intuitive. Also, going back to slow and verbose programming will be surprisingly difficult.

1.1 Going fast

Imagine you have a small, on-disk, 100 million row database containing a time-series with two float values at each time. Additionally this data could be split in three different tables covering different measurements. Here's how fast k9 can read in the data from disk and compute a statistic, average difference over each table, which uses each and every row.

This section requires 2: a feature only in the enterprise version of Shakti. If that is not available then use the section below with 1:

```
 bash-3.2$ ./k
 2021.xx.xx 17GB 4core (c) shakti
 \t q:2:`q;{select s:avg a-b from x}'q[!q]
884
```

That's 884 ms to read the data in from disk and compute over all the 100 million values. The data read is the biggest bit. If the data were already in memory then the computation would be faster:

```
 \t {select s:avg a-b from x}'q[!q]
217
```

217 ms, not bad for 100 million calculations.

The code to generate the on-disk database is presented below. Speed of course will depend on your hardware so times will vary.

```
nf:d+*|d:(|-d),d:683 954 997 1000;
T:^`t ?[;_8.64e7]@
B:100++\1e-2*-3+nf bin/:?[;*|nf]@
S:?[;1e-2*2,2,8#1]@
L:{select t,b,a:b+s from +`t`b`s!(T;B;S)@'x}
q:`eurusd`usdjpy`usdchf!L'_60e6 20e6 20e6
`q 2:q
```

This section requires 1: a feature in all versions of Shakti.

```
 bash-3.2$ ./k
 2021.xx.xx 17GB 4core (c) shakti
 \t select s:avg a-b from q:`csv?1:"q.csv"
832
```

That's 832 ms to read the data in from disk and compute over all the 10 million values. The data read and csv conversion process are the biggest bits.

Here is the code to generate the q.csv on-disk file. Note in this example only 10 million lines are generated versus the 100 million lines in the previous example using 2:

```
nf:d+*|d:(|-d),d:683 954 997 1000;
```

```
 T:^`t ?[;_8.64e7]@
 B:100++\1e-2*-3+nf bin/:?[;*|nf]@
 S:?[;1e-2*2,2,8#1]@
 L:{select t,b,a:b+s from +`t`b`s!(T;B;S)x́}
 q:L[_10e6]
 "q.csv"1:`csv@q
"q.csv"
```

1.2 Going concise

The k9 language is more closely related to mathematics syntax than most programming lanauges. It requires the developer to learn to "speak" k9 but once that happens most find an ability to speak quicker and more accurately in k9 than in other languages. At this point an example might help.

In mathematics, "3+2" is read as "3 plus 2" as you learn at an early age that "+" is the "plus" sign. For trival operations like arithmetic most programming languages use symbols also. When moving beyond arithmetic, most programming lanauges switch to words while k9 remains with symbols. As an example, to determine the distinct values of a list most programming languages might use a syntax like `distinct()` while k9 uses `?`. This requires the developer to learn how to say a number of symbols but once that happens it results in much shorter code that is quicker to write, easier to inspect, and easier to maintain.

This should not be surprising. In arithmetic, which do you find easier to understand?

```
/ math with text
Three plus two times open parenthesis six plus fourteen close parenthesis

/ math with symbols
3+2*(6+14)
```

In code, if you're new to k9 then it's unlikley you can understand the second example.

```
/ code with text
x = (0,12,3,11,3);y=5;
distinct_x = list(set(x));
sorted(i for i in distinct_x if i >= y)

/ code with symbols
x:0 12 3 11 3;y:5;
z@&y<z:?x
```

When you first learned arithmetic you likely didn't have a choice. Now you have a choice whether or not you want to learn k9. If you give it a try, then you'll likely get it quickly and move onto the power phase fast enough that you'll be happy you gave it a chance.

1.3 Get k9.

`https://shakti.com`

k9 is available in two versions, standard (under download) and enterprise. The enterprise version has additional features indicated on the k9 help page and also indicated in this tutorial.

Once downloaded you will need to change the file mode with the following commmand

```
chmod +x k
```

On the mac if you then attempt to run this file you likely won't succeed due to MacOS security. You'll need to go to "System Preferences..." and then "Security and Privacy" and select to allow this binary to run. (You'll have to have tried and failed to have it appear here automatically.)

On linux (and macos if you have installed npm) one can download k from the command line via

```
npm i @kparc/k -g
```

1.4 Help / Info Card

Typing \ in a terminal window gives you a concise overview of the language. This document aims to provide details to beginning user where the help screen is a bit too terse. Some commands are not available in the basic version and thus marked with an asterisk, eg. *4: https get.

```
select count first last min max sum avg var dev .. by ..
in n_(rand) n@(multiply) n?(divide) n@n?(bar)

Verb     monad        Adverb           Type
+ +                   ' each           char   " ab"
- -                   / over           sym    ``ab
* *                   \ scan           bool   011b
% div                                  int    2 3 4
! mod    where        System           float  2 3e4
& &      flip         \l load         -fixed  2.0 3.4
| |      reverse      \t time         -locus  -74::40.7
< <      asc          \v vars      z.d date   2001.02.03
> >      desc         \w work      z.t time   12:34:56.789
= =      freq                      z.T datetime
~ ~      ~
, ,      ,
# take   count        I/O              Class
_ drop   first        0' line          expr   :2+a
^ cut    sort         1' char/stdout   func   f[a] 2+a
@ @      type         2' data/stderr
? find   unique      *3' set           list   (2;3.4)
$ parse  str         *4' get           dict   {a:2 3}
. dict   value       *5' ffi           table  [a:2 3]
```

1.5 rlwrap

Although you only need the `k` binary to run k9 most will also install rlwrap, if not already installed, in order to get command history in a terminal window. rlwrap is "Readline wrapper: adds readline support to tools that lack it" and allows one to arrow up to go through the command buffer history.

In order to start k9 you should either run `k` or `rlwrap k` to get started. Here I will show both options but one should run as desired. In this document lines with input are shown with a leading space and output will be without. In the examples below the user starts a terminal window in the directory with the `k` binary file. Then the users enters *`rlwrap ./k`* *RET*. k9 starts and displays the date of the build, (c), and shakti and then listens to user input. In this example I have entered the command to exit k9, \\. Then I start k9 again without rlwrap and again exit the session.

```
 rlwrap ./k
Sep 13 2020 16GB (c) shakti
 \\

 ./k
Sep 13 2020 16GB (c) shakti
 \\
```

1.6 REPL

k9 runs as a read, evaluation, print loop (REPL). This means that one either programs in an interactive programming environment (eg. a shell/terminal window) or by running a script. There is no reason to compile code into an executable.

1.7 Simple example

Here I will start up k9, perform some trivial calculations, and then close the session. After this example it will be assumed the user will have a k9 session running and working in repl mode. Comments (/) will be added to the end of lines as needed. One can review [plus], page 7, [where], page 9, [floor], page 19, and [timing], page 82, as needed.

```
 1+2  / add 1 and 2
3

 !5   / generate a list of 5 integers from 0 to 4
0 1 2 3 4

 1+!5 / add one to each element of the list
1 2 3 4 5

 !_100e6;     / generate a list of 100 million integers (suppress output with ;)
 1+!_100e6;   / do 100 million sums
 \t 1+!_100e6 / time the operations in milliseconds
82
```

Now let's exit the session.

```
 \\
bash-3.2$
```

1.8 Document formatting for code examples

This document uses a number of examples to familiarize the reader with k9. The syntax is to have input with a leading space and output without a leading space. This follows the terminal syntax where the REPL input has space but prints output without.

```
 3+2 / this is input
5    / this is output
```

1.9 k9 idiosyncracies

One will need to understand some basic rules of k9 in order to progress. These may seem strange at first but the faster you learn them, the faster you'll move forward. Also, some of them, like overloading based on number of arguments, add a lot of expressability to the language.

1.9.1 Colon (:) is used to set a variable to a value

`a:3` is used to set the variable, a, to the value, 3. `a=3` is an equality test to determine if a is equal to 3.

1.9.2 Percent (%) is used to divide numbers

Yeah, 2 divided by 5 is written as `2%5`, not `2/5`. This choice is because `%` is similar to ÷, and the \ and / symbols are used elsewhere.

1.9.3 Evaluation is done right to left

2+5*3 is 17 and 2*5+3 is 16. 2+5*3 is first evaluated on the right most portion, 5*3, and once that is computed then it proceeds with 2+15. 2*5+3 goes to 2*8 which becomes 16.

1.9.4 There is no arithmetic order

`+` has equal precedence as *. The order of evaluation is done right to left unless parenthesis are used. (2+5)*3 = 21 as the 2+5 in parenthesis is done before being multiplied by 3.

1.9.5 Operators are overloaded depending on the number of arguments.

```
 *(13;6;9)    / single argument: * returns the first element
13
 2*(13;6;9)   / two arguments: * is multiplication
26 12 18
```

1.9.6 Lists and functions are very similar.

k9 syntax encourages you to treat lists and functions in a similar function. They should both be thought of a mapping from a value to another value or from a domain to a range. Lists and functions do not have the same type.

```
 l:3 4 7 12
 f:{3+x*x}
 l@2
7
 f@2
7
 @l
`I
 @f
`.
```

1.9.7 k9 notions of Noun, Verb, and Adverb

k9 uses an analogy with grammar to describe language syntax. The k9 grammar consists of nouns (data), verbs (functions) and adverbs (function modifiers).

- `The boy ate an appple. (Noun verb noun)`
- `The girl ate each olive. (Noun verb adverb noun)`

In k9 as the Help/Info card shows data are nouns, functions/lists are verbs and modifiers are adverbs.

- `3 > 2 (Noun verb noun)`
- `3 >' (1 12;1 4 5) (Noun verb adverb noun)`

2 Verbs

This chapter covers verbs which are the core primitives of k9. Given how different it is to call functions in k9 than in most other languages, you may have to read this chapter a few times. Once you can "speak" k9 you'll read `|x` better than `reverse(x)`.

Most functions are overloaded and change depending on the number of arguments. This can confuse new users. Eg. `(1 4)++(2 3;5 6;7 8)` contains the plus symbol once as flip and then for addition. (Remember evaluation is right to left!)

```
Verb     monad
+ +
- -
* *
% [div], page 9
! [mod], page 9,    [where], page 9
& &      [flip], page 10
| |      [reverse], page 11
< <      [asc], page 12
> >      [desc], page 12
= =      [freq], page 12
~ ~      ~
, ,      ,
# [take], page 17,   [count], page 17
_ [drop], page 19,   [first], page 8
^ [cut], page 15,    [sort], page 15
@ @      [type], page 21
? [find], page 20,   [unique], page 20
$ [parse], page 20,  [str], page 19
. [dict], page 23,   [value], page 22
```

2.1 plus ⇒ x+y

Add x and y. Arguments can be elements or lists but if both x and y are lists then they must be of equal length.

```
 3+7
10

a:3;
 a+8
11

 3+4 5 6 7
7 8 9 10

3 4 5+4 5 6
7 9 11
```

```
 3 4+1 2 3        / lengths don't match
!length

 10:00+1         / add a minute
10:01

 10:00:00+1      / add a second
10:00:01

 10:00:00.000+1  / add a millisecond
10:00:00.001
```

2.2 negate ⇒ -x.

```
 -3
-3

 --3
3

 x:4;
 -x
-4

 d:`a`b!((1 2 3);(4 5 6))
 -d
a|-1 -2 -3
b|-4 -5 -6
```

2.3 minus ⇒ x-y.

Subtract y from x.

```
 5-2
3

 x:4;y:1;
 x-y
3
```

2.4 first ⇒ _x

Return the first value of x.

```
 _1 2 3
1

 _((1 2);(3 4);(5 6))
1 2
```

```
 __((1 2);(3 4);(5 6))
1

 _`a`b.((1 2 3);(4 5 6))
1 2 3

 _|1 2 3
3
```

2.5 times ⇒ x*y

Mutliply x and y.

```
 3*4
12

3*4 5 6
12 15 18

1 2 3*4 5 6
4 10 18
```

2.6 sqrt ⇒ %x

Return the square root of x.

```
 %25
5.000e+00
 %14.4
3.794e+00
```

2.7 div ⇒ x%y

Divide x by y.

```
 12%5
2.400e+00

6%2
3.000e+00
```

2.8 mod ⇒ x!y

The remainder after y divided by x using integer division. x and y must be integers.

```
 12!27
3
 5!22
2
```

2.9 where ⇒ !x

Given a list of boolean values return the non-zero indices.

```
 !00110011b
2 3 6 7

 "banana"="a"
010101

 !"banana"="a"
1 3 5

 x@!30<x:12.7 0.1 35.6 -12.1 101.101
3.560e+01 1.011e+02
```

2.10 flip ⇒ &x

Flip, or transpose, x where x is list of 2 or more dimensions.

```
 x:(1 2;3 4;5 6);x
(1 2;3 4;5 6)

 &x
(1 3 5;2 4 6)

 `a`b.&x
{a:1 3 5;b:2 4 6}

 &`a`b.&x
[a:1 3 5;b:2 4 6]
```

Given a list of boolean values return a list of indices where the value is zero (index 0) and non-zero (index 1).

```
 &00110011b
(0 1 4 5;2 3 6 7)

 "banana"="a"
010101

 &"banana"="a"
(0 2 4;1 3 5)

 x@&30<x:12.7 0.1 35.6 -12.1 101.101
(1.270e+01 0.100e+00 -1.210e+01;3.560e+01 1.011e+02)
```

Given a list of non-binary values, return the ascending sorted indices.

```
 & 4 1 2 -3 4 1
(,3;1 5;,2;0 4)
```

```
 &`a`b`a`d
(0 2;,1;,3)

 &"blue red greeen"
(4 8;,0;,7;3 6 11 12 13;,9;,1;,14;5 10;,2)

 & 2023.01.01 2023.01.02 2023.01.01
(0 2;,1)
```

2.11 and ⇒ x&y

The smaller of x and y. One can use the over adverb to determine the min value in a list.

```
 3&2
2

 1 2 3&4 5 6
1 2 3

0 1 0 0 1 0&1 1 1 0 0 0
0 1 0 0 0 0

 `a&`b
`a

 &/ 3 2 10 -200 47
-200
```

2.12 reverse ⇒ |x

Reverse the list x.

```
 |0 3 1 2
2 1 3 0

|"banana"
"ananab"

 |((1 2 3);4;(5 6))
(5 6;4;1 2 3)
```

To get the last element, one can take the first element of the reverse list (*|‘a‘b‘c).

2.13 or ⇒ x|y

The greater of x and y. Max of a list can be determine by use of the adverb over.

```
  3|2
3

1 2 3|4 5 6
```

```
4 5 6

 1 0 1 1 0 1|0 0 0 1 1 1
1 0 1 1 1 1

 |/12 2 3 10 / use over to determine the max of a list
12
```

2.14 asc(desc) ⇒ < (>) x

Sort a list or dictionary in ascending (<) or descending (>) order. Applied to a list, these will return the indices to be used to achieve the sort.

```
 <5 7 0 12
2 0 1 3

 x@<x:5 7 0 12
0 5 7 12

 d:`a`b`c!3 2 1;d
a|3
b|2
c|1

 <d / a dictionary is sorted by value
c|1
b|2
a|3
```

2.15 less (more) ⇒ x < (>) y

Return true (1b) if x is less (more) than y else false (0b).

```
 3<2
0b

2<3
1b

 1 2 3<4 5 6
1 1 1

 ((1 2 3);4;(5 6))<((101 0 5);12;(10 0)) / size needs to match
1 0 1
1
1 0

 "a"<"b"
1b
```

2.16 freq⇒ =x

Freq takes a list of uniform type, x, and returns a dictionary of the distinct values (key) and their number of occurences (value).

```
 ="banana"
abn.3 1 2

 =0 1 0 2 10 7 0 1 12
0 1 2 7 10 12.3 2 1 1 1 1
```

2.17 equal ⇒ x=y

Return true (1b) if x is equal to y else false (0b).

```
 2=2
1

2=3
0

2=2.
1

 "banana"="abnaoo" / check strings of equal length by character
0 0 1 1 0 0

 "banana"="apple"  / unequal length strings error
!length
```

2.18 not ⇒ ˜x

Boolean invert of x

```
 ~1
0

 ~1 0 1
0 1 0

 ~37 0 12
0 1 0
```

2.19 match ⇒ x˜y

Return true (1) if x matches y else false (0). A match happens if the two arguments evaluate to the same expression.

```
 2~2
1

 2~3
```

```
0

 2~2.
0

 "banana"~"apple"
0

 `a`b~`a`b  / different than = which is element-wise comparison
1

 `a`b=`a`b
1 1

 f:{x+y}
 f~{x+y}
1
```

2.20 enlist ⇒ ,x

Create a list from x

```
 ,3
,3

,1 2 3
1 2 3

 3=,3
,1

 3~,3
0
```

2.21 cat ⇒ x,y

Concatenate x and y.

```
 3,7
3 7

 "hello"," ","there"
"hello there"

 C:("ab";"c";("def";"ghi"));C
ab
c
("def";"ghi")
```

```
 ,/C     / join the list once
a
b
c
def
ghi

 ,//:C   / converge over join until single list
"abcdefghi"
```

2.22 sort ⇒ ^x

Sort list, dictionary, or table x into ascending order. Dictionaries are sorted using the keys and tables by the first column field. One can sort tables by arbitrary columns by first reordering the columns in the table using [take], page 17, or by extracting the sort column by index or expression.

```
 ^0 3 2 1
0 1 2 3

^`b`a!((0 1 2);(7 6 5))     / sort dictionary by key
[a:7 6 5;b:0 1 2]

 ^[[]z:`c`a`b;y:3 2 1]      / sort table by 1st col
z y
- -
a 2
b 1
c 3

 ^`y`z#[[]z:`c`a`b;y:3 2 1] / sort table by new 1st col
y z
- -
1 b
2 a
3 c
```

2.23 [f]cut ⇒ x^y

Cut list y by size, indices, or function x. Also, cut table y into key x and value.

action	**x**
[cut1], page 16,	positive int
[cut2], page 16,	negative int
[cut3], page 16,	list

[cut4], page 16, function

[cut5], page 16, column

Positve integer x, cut y into x rows

```
 3^101+!18   / 3 rows
101 102 103 104 105 106
107 108 109 110 111 112
113 114 115 116 117 118
```

Negative integer, cut y into x elements per row

```
 -3^101+!18  / 3 columns
101 102 103
104 105 106
107 108 109
110 111 112
113 114 115
116 117 118
```

Positive integer array x, cut at each element in x

```
 2 3 7 ^!17  / left list indicates start number of each row.
,2
3 4 5 6
7 8 9 10 11 12 13 14 15 16

 8 9 10 11 12^.1*!20
,0.8
,0.9
,1.
,1.1
1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9
```

Function returning boolean array, cut at each index where f is non-zero.

```
 {(x*x)within .5 1.5}^.1*!20
0.8
0.9
1.
1.1
1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9
```

Table returning utable (aka keyed table).

```
 t:[[]a:`x`y`z;b:1 20 1];t / an unkeyed table
a b
- --
x  1
y 20
z  1

 kt:`a^t;kt                / set `a as the key
```

```
a|b
-|--
x| 1
y|20
z| 1
```

2.24 count ⇒ #x

Count the number of elements in list, dictionary, or table x.

```
 #0 1 2 12
4

 #((0 1 2);3;(4 5))
3
 #`a`b!((1 2 3);(4 5 6)) / count the number of keys
2

 #5                      / single element is 1
1
```

2.25 [f]take ⇒ x#y

[f]take has a number of different actions depending on x. These include head (take from front of list), tail (take from back of list), common (take common elements), filter (take where function is non-zero), and select (take columns from table).

action	**x**	**y**
[take1], page 17,	positive int	list
[take1], page 17,	negative int	list
[take2], page 18,	list	list / dictionary
[take3], page 18,	function	list
[take4], page 18,	name(s)	table

Postive (negative) x returns the first (last) x elements of y. If abs[x]>count[y] then repeatedly pull from y.

```
 3#0 1 2 3 4 5          / take first 3 elements of a list
0 1 2

 10 # 17 12 25 / take repeat: take from list and repeat as needed
17 12 25 17 12 25 17 12 25 17

 -3#0 1 2 3 4 5         / take last elements of a list
3 4 5
```

If x and y are lists, then take returns any values common in both x and y. If x is a list and y is a dictionary, then take returns those values in y where the keys are in x.

```
 (1 2 3 7 8 9)#(2 8 20) / common
2 8

  (2 4 6 2 6 5 4) # (4 6 7 8 2 4 4 4) / substring of right argument
4 6 2 4 4 4

 (4 6 7 8 2 4 4 4) # (2 4 6 2 6 5 4)
2 4 6 2 6 4

 `b`d#`a`b`c`d!1 4 2 3
b|4
d|3
```

If x is a function, then select the values where the function is non-zero.

```
 (0.5<)#10?1.           / select values greater than 0.5
0.840732 0.5330717 0.7539563 0.643315 0.6993048f
```

If x is a name or list of names, then take the name column or columns from the table.

```
 t:[[]x:`a`b`c;y:3 1 2;z:"tab"];t
x y z
- - -
a 3 t
b 1 a
c 2 b

 `y#t
y
-
3
1
2

 `z`y#t
z y
- -
t 3
a 1
b 2

 `x`z`y#t
x z y
- - -
a t 3
```

```
b a 1
c b 2
```

2.26 floor ⇒ _x

Return the integer floor of float x.

```
 _3.7
3
```

2.27 [f]drop ⇒ x_y

[f]drop has a number of different actions depending on x. This include de-head (drop from front of list), de-tail (drop from back of list), uncommon (drop common elements), and filter (drop where function is zero).

action	**x**	**y**
[drop1], page 19,	positive int	list
[drop1], page 19,	negative int	list
[drop2], page 19,	list	list / dictionary
[drop3], page 19,	function	list

Postive (negative) x drops the first (last) x elements of y.

```
 3_0 1 2 3 4 5          / drop first three
3 4 5

 -2_0 1 2 3 4 5         / drop last two
0 1 2 3
```

If x and y are lists, then drop returns any values from y not found in x. If x is a list and y is a dictionary, then drop returns those values in y where the keys are not in x.

```
 (1 2 3 7 8 9)_(2 8 20) / uncommon
,20

 `b`d_`a`b`c`d!1 4 2 3
a|1
c|2
```

If x is a function, then select values where the function is zero.

```
 (0.5<)_10?1.           / values less than or equal to 0.5
0.02049699 0.1269226
```

2.28 str ⇒ $x

Cast x to string.

```
 $`abc
```

```
"abc"
 $4.7
"4.7"
```

2.29 parse ⇒ x$y

Parse string y into type x.

```
 `i$"23"
23

 `f$"2.3"
2.3

 `f$"2"
2

 `t$"12:34:56.789"
12:34:56.789

`d$"2020.04.20"
2020.04.20
```

2.30 unique ⇒ ?x

Return the distinct values of the list x. The `?` preceeding the return value explicitly shows that the list has no repeated values.

```
 ?`f`a`b`c`a`b`d`e`a
?`f`a`b`c`d`e

?"banana"
?"ban"
```

2.31 find ⇒ x?y

Find the first element of x that matches y otherwise return the end of vector.

```
 `a`b`a`c`b`a`a?`b
1

 `a`b`a`c`b`a`a?`d / missing thus return the list length
7

 0 1 2 3 4?10
5

(1;`a;"blue";7.4)?3
4
```

Return x uniform random numbers between 0 and y or from list y with the same type as y. If x is negative then values are unique.

```
 3?1.      / values [0,1.)
0.01640292 0.8029538 0.9569196

 3?10      / values [0,9)
6 9 4

 5?`a`b`c  / sample from list y
`a`a`c`c`b

 -3?3      / unique values
1 2 0

 3?3       / duplicates are possible
2 2 1
```

2.32 type ⇒ @x

Return the data type of x. Lower-case represents a single element while upper-case represents a list of type indicated.

```
 @2
`i

 @1.2
`f

 @`a
`n

 @"a"
`c

 @2020.04.20
`d

 @12:34:56.789
`t

 @(1;1.2;`a;"a";2020.04.20;12:34:56.789)  / type of a list
`:

 @'(1;1.2;`a;"a";2020.04.20;12:34:56.789) / type of elements of the list
`i`f`s`c`d`t

 @{x*x}
`.
```

```
 @`a`b!1 2
`a

 @[[]x:`a`b;y:1 2]
`A

 @[x:`a`b]y:1 2]
`a
```

2.33 [f]at ⇒ x@y

Given a list or function x return the value(s) at index(indices) y.

```
 (3 4 7 12)@2              / list
7

 `a`b`c@2
`c

 ((1 2);3;(4 5 6))@(0 1)   / values at indices 0 and 1
1 2
3

 {x*x}@3                   / function
9

 (sin;cos;1*)@\:0.01
0.009999833 0.99995 0.01

 (sqr;sqrt;|;#)@'(3;10;"word";`a`b`c) / list of fuctions applied to list of inputs
9
3.162278
drow
3
```

2.34 value ⇒ .x

Evaluate a string of valid k code, list of k code, or a system command. Can also be used to covert from a string to a numeric, temporal or name type.

String of k code

```
 ."3+2"
5

 ."20*1+!3"
20 40 60
```

List of k code

```
 .(*;16;3)
```

```
48

 n:3;p:+(n?(+;-;*;%);1+n?10;1+n?10);p
% 6 3
* 2 7
- 5 5

 .'p
2
14
0

 .(:;`a;12 14 16) / set a to (12 14 16)
 a
12 14 16
```

System command (must be prefixed with \\)

```
 ."\\ls ."
Mi2.0
json.dylib
```

Convert from string to numeric or name.

```
 ."15"
15

 ."12.7"
12.7

 ."`abc"
`abc
```

2.35 dict ⇒ x.y

Create a dictionary from key x and value y. Keys and vlaues can be of the various types, eg. numbers, symbols, characters, strings, dates, and times. There is a an alternative form to create dictionaries using curly brack notation, eg. {a:2 3}.

```
 d:`a`b`c.1 2 3;d
`a`b`c.1 2 3

 d`b
2

 d:1 2 3 . 4 5 6;d
1 2 3.4 5 6

 d:"blue" . 1 2 3 4;d
blue.1 2 3 4
```

```
 d:2023.01.01 2023.02.01 2023.03.01 . 1 2 3;d
2023.01.01 2023.02.01 2023.03.01.1 2 3

 d:`a`b`c . (1 2 3;4 5;6);d
{a:1 2 3;b:4 5;c:6}

 fa[a] a+1
 fb[a] a+2
 fc[a] a+3

 d:`f1`f2`f3 . (fa;fb;fc);d
{f1:(,`a;0a+;(1); a+1);f2:(,`a;0a+;(2); a+2);f3:(,`a;0a+;(3); a+3)}

 (d@`f1)@99
100

(d@`f3)@99
102
```

3 Adverbs

This chapter may be one of the most difficult for users new to array langauges but one must master adverbs to exploit the full power of k9. Nouns and verbs naturally come but adverbs need to be learned so that one can reach for them when needed.

Adverbs modify verbs to operative iteratively over nouns. This previous sentence likely will make sense only once you understand it so let's jump into an example. Imagine you have a list of 4 lists. Using `first` you will retrieve the first sub-list. In order to retrieve the first element of each sub-list you'll have to modify the function `first` with a modifier `each`.

```
 x:(1 2; 3 4; 5 6; 7 8);x
1 2
3 4
5 6
7 8

*x   / first
1 2

*'x  / each first
1 3 5 7
```

Adverbs are also used where one might use a `for`, `do`, or `while` loop to calculate over a list of value or recursively until a condition is met.

```
 f:{x*y}
 f\1 .1 .1 .1 / f\ scan over list
1
0.1
0.01
0.001

 g:{x*x}
 g\:.99       / g\: scan until convergence
0.99 0.9801 0.960596 .. 8.87948e-287 0
```

3.1 each ⇒ f'x

Apply function f to each value in list x in a single thread. Multithread operation is done with [eachprior], page 28.

```
 *((1 2 3);4;(5 6);7)   / first element of the list
```

```
1 2 3

 *'((1 2 3);4;(5 6);7)  / first element of each element
1 4 5 7

 !3                     / simple list
0 1 2
 +/!3                   / sum of the list
3
 2#'!3                  / take 5 copies of the list
0 0
1 1
2 2
 +/2#'!3                / sum
3 3
 +/'2#'!3               / sum each, eg. (0 1 2)+(0 1 2)
0 2 4
```

3.2 over ⇒ f/y or x f/y

Over has the same function as [scan], page 26, but returns only the last value.

3.3 scan ⇒ f\y or x f\y

Compute f, a function of two arguments, over list y with optional initial value x. On the first iteration return y_0. On each subsequent iteration function will be called with previous output and the next element of y as arguments, i.e. f_i=f[f_{i-1};y_i]. Calling with f, a function of one argument, is covered in [each], page 25.

i	**x**	**f[x]**	**a f[x]**	**x:0 1 4 9 16 25**	**+\x**	**3+\x**	**+/x**	**3+/x**
0	x_0	x_0	f_0[a;x_0]	0	0	3		
1	x_1	f_1[f_0;x_1]	f_1[f_0;x_1]	1	1	4		
..	..	..	..	..	..	..		
j	x_j	f_j[f_{j-1};x_j]	f_j[f_{j-1};x_j]	..	..	..		
..	..	..	..	..	..	..		
n-2	x_{n-2}	f_{n-2}[f_{n-3};x_{n-2}]	f_{n-2}[f_{n-3};x_{n-2}]	16	30	33		
n-1	x_{n-1}	f_{n-1}[f_{n-2};x_{n-1}]	f_{n-1}[f_{n-2};x_{n-1}]	25	55	58	55	58

Examples

```
 (,\)("a";"b";"c")   / scan and cat to build up a list
a
```

```
ab
abc

 +\1 20 300          / scan and plus to sum over a list
1 21 321

 {y+10*x}\1 20 300   / scan and user func
1 30 600

 {y+10*x}/1 20 300   / over and user func
600

 x:1 2 3               / x+y0, +y1, +y2
 y:10 10 10
 x+\y
11 12 13
21 22 23
31 32 33

 {x+.5*y}\[1 2 3;20 20 20]  / f\y syntax
11 12 13
21 22 23
31 32 33

 1 2 3{x+.5*y}\20 20 20    / i f\x syntax
11 12 13
21 22 23
31 32 33
```

3.4 join ⇒ c/x

Join a list of strings x using character c between each pair.

```
 "-"/("Some";"random text";"plus";,".")
"Some-random text-plus-."
```

The notaion c/x converts a list of integers x via base c into base 10. In the example below, assuming 2 0 1 are digits in base 8, they convert to 1 + (0*8) + (2*8*8) = 129

```
 8/2 0 1
129
```

3.5 split ⇒ c\x

Split the string x using character c to determine the locations.

```
 " "\"Here is a short sentence."
Here
is
a
short
```

```
sentence.
```

c\x splits a number x into base c digits.

```
 8\129
2 0 1
```

3.6 eachprior ⇒ f':[x;y] or y f':x

Compute f, a function of two arguments, over list x with optional initial value y. Concretely f_i=f[x_i;x_{i-1}] where f_0 returns f[x_0;i].

```
 ,':[`i;(`$"x",'$!5)]
x0 i
x1 x0
x2 x1
x3 x2
x4 x3

 %':[100;100 101.9 105.1 102.3 106.1] / compute returns
1 1.019 1.031403 0.9733587 1.037146

 100%':100 101.9 105.1 102.3 106.1    / using infix notation
1 1.019 1.031403 0.9733587 1.037146

 101.9{($x)," % ",($y)," = ",$x%y}':100.1 101.9 105.1 102.3 106.1
100.1 % 101.9 = 0.9823356
101.9 % 100.1 = 1.017982
105.1 % 101.9 = 1.031403
102.3 % 105.1 = 0.9733587
106.1 % 102.3 = 1.037146
```

Additionally eachp (':) is a multithreaded (parallel) each. This will allow a function to run using multiple threads over a list of arguments.

```
\t {L:x?1.;sum L%sin L*cos L*sin L+L*L}':8#_50e6 / multiple threads
831

\t {L:x?1.;sum L%sin L*cos L*sin L+L*L}'8#_50e6  / single thread
1202
```

3.7 eachright ⇒ f/:[x;y]

Apply f[x;y] to each value of y, i.e. the right one.

```
 (!2)+/:!3
0 1
1 2
2 3

 {x+10*y}/:[!2;!3]
0 1
```

```
10 11
20 21

 "pre-",/:("apple";"banana";"cherry")
pre-apple
pre-banana
pre-cherry
```

3.8 eachleft ⇒ f\:[x;y]

Apply f[x;y] to each value of x, i.e. the left one.

```
 (!2)+\:!3
0 1 2
1 2 3

 {x+10*y}\:[!2;!3]
0 10 20
1 11 21

 ("apple";"banana";"cherry"),\:"-post"
apple-post
banana-post
cherry-post
```

3.9 Converge Over ⇒ g/:x or (n;g)/:x

Converge Over is the same function as [cscan], page 29, but only returns the final value(s).

```
 r:((1;2);3;(,4));r
1 2
3
,4

 ,//:r  / raze over the list until it converges to a flat list
1 2 3 4
```

3.10 Converge Scan ⇒ g\:x or (n;g)\:x

Compute function g where g_i=g[g_{i-1}] and g_0=x. If n is specified then it is either (DO) the number of iterations or (WHILE) a function returning a non-zero result.

```
  {x*x}\:.99           / converge to value
0.99 0.9801 0.9605961 ... 1.323698e-18 1.752177e-36 0.

 (3;{x*x})\:.99        / do 3 times, return 4 values (first is initial)
0.99 0.9801 0.960596 0.9227448

 (.5<;{x*x})\:.99      / while .5 < r, i.e. r is greater than 0.5
0.99 0.9801 0.960596 0.9227448 0.8514579 0.7249806 0.5255968 0.276252
```

```
 (4;{x,x+/x+1})\:,0    / do function 4 times
,0
0 1
0 1 3 4
0 1 3 4 12 13 15 16
0 1 3 4 12 13 15 16 72 73 75 76 84 85 87 88

 (10>#;{x,x+/x+1})\:,0 / while output is less than 10 long
,0
0 1
0 1 3 4
0 1 3 4 12 13 15 16
0 1 3 4 12 13 15 16 72 73 75 76 84 85 87 88

 x:3 4 2 1 4 99        / until output is x
 x\:0
0 3 1 4

 / the latter example worked out manually
 x 0
3
 x 3
1
 x 1
4
 x 4                   / x[4]=4 so will stop
4
```

4 Noun

The basic data types of the k9 language are booleans, numbers (integer and float), text (characters and enumerated/name) and temporal (date and time). It is common to have functions operate on multiple data types.

In additional to the basic data types, data can be put into lists (uniform, i.e., all elements of the same type, and non-uniform), dictionaries (key-value pairs), and tables (transposed/flipped dictionaries). Dictionaries and tables will be covered in another chapter.

Here are some examples of each of these types (note that 0N is a null integer and 0n a null float).

```
Type
[char], page 34,    " ab"
[sym], page 34,     ``ab
[bool], page 32,    011b
[int], page 32,     2 3 4
[float], page 33,   2 3e4
[fixed], page 33,   2.0 3.4
[locus], page 33,   -74::40.7
[date], page 34,    2001.02.03
[time], page 34,    12:34:56.789
[datetime], page 35
```

Data types can be determined by using the `@` function on values or lists of values. In the case of non-uniform lists `@` returns the type of the list `` `L `` but the function can be modified to evaluate the type of each element in the list using `@'`.

```
 @1            / integer atom
`i
 @1 2 3        / integer list
`I
 @12:34:56.789 / time atom
 @(3;3.1;"b";`a;12:01:02.123;2020.04.05) / mixed list
`L
 @'(3;3.1;"b";`a;12:01:02.123;2020.04.05)
`i`f`c`n`t`d
```

4.1 Atom Types

This section lists all the different types available. Generally lower case specifies atoms and upper case as lists.

name	type	example	note
b	boolean	1b	
c	character	"a"	
d	date	2020.06.14	
e	float	3.1	
f	float	3.1f	
g	int	2g	1 byte unsigned

h	int	2h	2 byte unsigned
i	int	2	4 byte unsigned
j	int	2j	8 byte signed
n	name	‘abc	8 char max
s	time	12:34:56	second
S	datetime	2020.06.15T12:34:56	second
t	time	12:34:56.123	millisecond
T	datetime	2020.06.15T12:34:56.123	millisecond
u	time	12:34:56.123456	microsecond
U	datetime	2020.06.15T12:34:56.123456	microsecond
v	time	12:34:56.123456789	nanosecond
V	datetime	2020.06.15T12:34:56.123456789	nanosecond

4.2 bool ⇒ Boolean b

Booleans have two possible values 0 and 1 and have a 'b' to avoid confusion with integers, eg. 0b or 1b.

```
 0b
0b
 1b
1b
 10101010b
10101010b
```

4.2.1 boolean logic

k9 implements logic operations in the usual way.

```
 x:0101b;y:0011b

[[x:x;y:y]AND:x&y;OR:x|y;NAND:~x&y;NOR:~x|y;XOR:~x=y;XNOR:x=y]
x y|AND OR NAND NOR XOR XNOR
- -|--- -- ---- --- --- ----
0 0|  0  0    1   1   0    1
1 0|  0  1    1   0   1    0
0 1|  0  1    1   0   1    0
1 1|  1  1    0   0   0    1
```

4.3 Numeric Data

Numbers can be stored as integers and floats.

4.3.1 int ⇒ Integer g, h, i, j

Integers can be stored in four different ways which correspond 1, 2, 4, and 8 bytes. The first three are unsigned and the last (j) is signed. Positive numbers default to i and negative and very large numbers default to j. One can specify a non-default type by adding one of the four letters immediately following the number.

```
 @37    / will default to i
`i
```

```
 @-37   / negative so will default to j
`j

 @37g   / cast as g, one byte unsigned
`g

b:{-1+*/x#256}
 `g b[1]
255g
 `h b[2]
65535h
 `i b[4]
4294967295
 `j b[7]
72057594037927935
```

4.3.2 float ⇒ Float e, f

Float

```
 3.1
3.1
 3.1+1.2
4.3
 3.1-1.1
2.
 @3.1-1.1
`e
 @3.1
`e
 a:3.1;
 @a
`e
 @1%3
`f
```

4.3.3 fixed ⇒ Float e, f

Fixed TBD

```
 2 3e4
2.000e+00 3.000e+04
```

4.3.4 locus ⇒ Float e, f

Locus TBD

```
 -74::40.7
4.07e+01
```

4.4 Text Data

Text data come in characters, lists of characters (aka strings) and enumerated types. Enumerated types are displayed as text but stored internally as integers.

4.4.1 char ⇒ Character c

Characters are stored as their ANSI value and can be seen by conversion to integers. A string is a list of characters (including blanks).

```
 @"b"
`c
 @"bd"
`C
```

4.4.2 sym ⇒ Name n

A symbol (sym) is an enumerated type displayed as a text string but stored internally as an integer value.

```
 @`blue
`n
 @`blue`red
`N
```

4.5 Temporal Data

Temporal data can be expressed as time, date, or a combined date and time.

4.5.1 time ⇒ Time s, t, u, v

Time has four types depending on the level of precision. The types are seconds (s), milliseconds (t), microseconds (u), and nanoseconds (v). The times are all stored internally as integers. The integers are the number of time units. For example 00:00:00.012 and 00:00:00.000000012 are both stored as 12 internally.

```
 @12:34:56.789            / time
`t
 .z.t                      / current time in GMT (Greenwich Mean Time)
17:32:57.995
 t: .z.t-17:30:00.000; t
00:03:59.986
 t
17:33:59.986
 `i 00:00:00.001           / numeric representation of 1ms
1
 `i 00:00:01.000           / numeric representation of 1s
1000
 `i 00:01:00.000           / numeric representation of 1m
60000
 `t 12345                  / convert number to milliseconds
00:00:12.345
```

4.5.2 date ⇒ Date d

Dates are in yyyy.mm.dd format. Dates are stored internally as integers with 0 corresponding to 2001.01.01.

```
 @2020.04.20              / date
`d
 .z.d                      / current date in GMT
2020.12.05
 `i .z.d                   / numeric representation of date
7278
 `i 2001.01.01             / zero date
0
 `d 0                      / zero date
2001.01.01
```

4.5.3 datetime ⇒ Datetime d

Dates and times can be combined into a single datetime element by combining a date, the letter T, and the time together without spaces. The datetime use the same lettering as the time precision but in uppercase. Datetimes are stored interally as integers. For example 2001.01.02T00:00:00.000 is stored as 86,400,000, the number of milliseconds in a day.

```
 @2020.04.20T12:34:56.789  / date and time
`d
 `T$"2020.04.20 12:34:56.789" / converting from string
2020.04.20T12:34:56.789
```

4.6 Extreme values

Data types can represent in-range, null, and out-of-range values.

type	**null**	**out of range**
i	0N	0W
f	0n	0w

```
 0%0
0n
 1e500
0w
```

5 List

Lists and derivatives of lists are fundamental to k9 which makes sense given that the language is made to process large quantites of data. Performance will be best when working with uniform lists of a single data type but k9 supports list of non-uniform type also.

Lists are automatically formed when a sequence of uniform type are entered or generated by any function.

```
 1 3 12      / list of ints
1 3 12

 3.1 -4.1 5. / list of floats
3.1 -4.1 5.

 "abc"       / list of chars
"abc"

 `x`y`z      / list of names
`x`y`z
```

In order to determine if data is an atom or a list, one can use the [type], page 21, command. The command returns a lower case value for atoms and an upper case value for lists.

```
 @1          / an integer
`i

@1 3 12     / list of ints
`I

@,1         / list of single int via [enlist], page 14
`I
```

Commands that generate sequences of numbers return lists regardless of whether the count (length of the list) is 1 or many.

```
 @!0
`I
 @!1
`I
 @!2
`I
```

5.1 List Syntax

In general, lists consist of elements separated by semicolons and encased by parenthesis.

```
 (1;3;12)         / list of ints
1 3 12

@(1;3.;`a;"b")   / non-uniform list
```

```
`L

@((1;3);(12;0))  / list of integer lists
LI

@'((1;3);(12;0)) / each list is type I
`I`I

,,,,,(3;1)      / a list of a list of a list..
,,,,,3 1
```

5.2 List Indexing

Lists can be indexed in different ways. The @ notation is often used as it's fewer characters than [] and the explicit @ instead of space is likely more clear.

```
 a:2*1+!10 / 2 4 ... 20
 a[9]      / square bracket
20
 a@9       / at
20
 a 9       / space
20
 a(9)      / parenthesis
20
 a[10]     / out of range return zero
0
```

5.3 List of Lists Indexing

A list of lists can be indexed by succesive index operations left to right.

```
 x:3^!12;x  / cut a list into a matrix or list of lists
0 1 2 3
4 5 6 7
8 9 10 11

 x[0]        / index the first element which is a list
0 1 2 3

 x[0 2]      / index two elements, the 0 and 2 elements
0 1 2 3
8 9 10 11

 x[0;2]      / index the first row and then 2 element of that row
2

 x[;2][0]    / index the 2 column and then the 0 row
2
```

```
 x[0][2]     / index the 0 row and then the 2 column
2
```

5.4 Updating List Elements

Lists can be updated elementwise by setting the indexed element to a required value. There is also a syntax for updating many elements and that is found at [amend], page 57.

```
 a:2*1+!10;a
2 4 6 8 10 12 14 16 18 20
 a[3]:80
 a
2 4 6 80 10 12 14 16 18 20
```

5.5 Function Applied to Lists

Most functions can be applied to lists without special syntax as if it was an element.

```
 x:!3;x
0 1 2

 x+10
10 11 12

 +\x
0 1 3

 +/x
3

 {x*x:sin x}[x]
0 0.7080734 0.8268218
```

5.6 Functions Applied to Two Lists

This section will focus on functions (f) that operate on two lists (x and y). As these are internal functions, examples will be shown with infix notation (x+y) but prefix notation (+[x;y]) would also be possible.

5.6.1 Pairwise

These functions operate on list elements pairwise and thus requires that x and y are equal length.

- `x+y : Add`
- `x-y : Subtract`
- `x*y : Multiply`
- `x%y : Divide`
- `x&y : AND/Min`

- `x|y : OR/Max`
- `x>y : Greater Than`
- `x<y : Less Than`
- `x=y : Equals`
- `x!y : Dictionary`

```
 x:1+!5; y:10-2*!5
 x
1 2 3 4 5
 y
10 8 6 4 2
 x+y
11 10 9 8 7
 x-y
-9 -6 -3 0 3
 x*y
10 16 18 16 10
 x%y
0.1 0.25 0.5 1 2.5f
 x&y
1 2 3 4 2
 x|y
10 8 6 4 5
 x>y
00001b
 x<y
11100b
 x=y
00010b
 x!y
1|10
2| 8
3| 6
4| 4
5| 2
```

5.6.2 Each Element of One List Compared to Entire Other List

These functions compare x[i] to y or x to y[i]. They are not symmetric to their inputs, i.e. f[x;y] does not equal f[y;x];

- `x^y : Reshape all element in y by x`
- `x#y : List all elements in x that appear in y`
- `x?y : Indices of elements of y in x else return the length of x.`

```
 x:0 2 5 10
 y:!20
 x^y
0 1
```

```
2 3 4
5 6 7 8 9
10 11 12 13 14 15 16 17 18 19

 x:2 8 20
 y:1 2 3 7 8 9
 x#y
2 8
 x?y
3 0 3 3 1 3
```

5.6.3 Each List Used Symmetrically

This is symmetric in its inputs f[x;y]=f[y;x] and the lists are not required to be equal length.

- `x_y : Values present in only one of the two lists`

```
 x:2 8 20
 y:1 2 3 7 8 9
 x_y
1 3 7 9
```

6 Dictionary

Dictionaries are a data type of key-value pairs typically used to retrieve the value by using the key. In other computer languages they are also known as associative arrays and maps. Keys should be unique to avoid lookup value confusion but uniqueness is not enforced. The values in the dictionary can be single elements, lists, tables, or even other dictionaries.

Dictionaries in k9 are often used. As an example from finance, market quotes and trade data are dictionaries of symbols (name keys) and market data (table values).

6.1 Dictionary Creation ⇒ x!y or [x_i:y_i]

Dictionaries are created by using the key symbol or square bracket notation and listing the keys (x) and values (y).

```
x!y
```

```
 d0:`a37!12; d0                         / key is `a37 and value is 12
a37|12

 d1:`pi`e`c!3.14 2.72 3e8;d1            / three keys `pi, `e, `c
pi|3.14
e |2.72
c |3e+08

 `a`b`c!(1 2 3;10 20 30;100 200 300 499) / values are lists
a|1 2 3
b|10 20 30
c|100 200 300 499
```

[x_0:y_0;x_1:y_1;...;x_n:y_n]

```
 d0:[a37:12]
a37|12

 d1:[pi:3.14;e:2.72;c:3e8];d1
pi|3.14
e |2.72
c |3e+08

 [a:1 2 3;b:10 20 30;c:100 200 300]
a|1 2 3
b|10 20 30
c|100 200 300
```

Often one will need to modify the data while building the dictionary. User defined functions can easily accomplish this task.

```
 `x`sinX`cosX!{(x;sin y;cos z)} . 3#,0.1*(!62)
x   |0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1 1.1 ..
sinX|0 0.09983342 0.1986693 0.2955202 0.3894183 0..
cosX|1 0.9950042 0.9800666 0.9553365 0.921061 0.8..
```

6.2 Dictionary Selection ⇒ x#y or x_y

One can select a portion of a dictionary either by [take], page 17, or [drop], page 19, which keeps and removes keys respectively.

```
 x:`a`b`c`d!4 2 3 1;x
a|4
b|2
c|3
d|1

 `b`c#x / take
b|2
c|3

 `b`c_x / drop
a|4
d|1
```

6.3 Dictionary Join ⇒ x,y

This joins dictionaries together and the right or y dictionary will overwrite the left or x dictionary if common keys are present.

```
 d1:`a`b`c!3 2 1;d1
a|3
b|2
c|1

 d2:`c`d`e!90 80 70;d2
c|90
d|80
e|70

 ^d1,d2 / ^ sort by key
a| 3
b| 2
c|90
d|80
e|70

 ^d2,d1 / ^ sort by key
a| 3
b| 2
c| 1
d|80
e|70
```

6.4 Dictionary Indexing ⇒ x@y

Dictionaries, like lists, can be key indexed in a number of ways.

```
 x:`a`b`c!(1 2;3 4;5 6);x
a|1 2
b|3 4
c|5 6

x@`a   / single index
1 2

x@`a`c / multiple index
1 2
5 6

/ all these notations for indexing work (output not shown)
 x@`b;  / at
 x(`b); / parenthesis
 x `b;  / space
 x[`b]; / square bracket
```

6.5 Dictionary Key ⇒ !x

The keys from a dictionary are retreived by using the ! function.

```
 !d0
`pi`e`c
 !d1
`time`temp
 !d2
0 10 1
```

6.6 Dictionary Value ⇒ x[]

The values from a dictionary are retrieved using bracket notation.

```
 d0[]
3.14 2.72 3e+08
 d1[]
12:00 12:01 12:10
25.   25.1  25.6

 d2[]
37.4 46.3 0.1
```

One could return a specific value by indexing into a specific location. As an example, in order to query the first value of the values associated with the key temp from d1, one would convert d1 into values (a pair of lists), and index that by [1;0].

```
 d1
time|12:00 12:01 12:10
```

```
temp|25 25.1 25.6

 d1[]
12:00 12:01 12:10
25    25.1  25.6

 d1[][1]
25 25.1 25.6
 d1[][1;0]
25.
```

6.7 Sorting a Dictionary by Key ⇒ ^x

```
 d0
pi|3.14
e |2.72
c |3e+08

 ^d0
c |3e+08
e |2.72
pi|3.14
```

6.8 Sorting a Dictionary by Value ⇒ <x (>x)

```
 d0
pi|3.14
e |2.72
c |3e+08

 <d0
e |2.72
pi|3.14
c |3e+08

 >d0
c |3e+08
pi|3.14
e |2.72
```

6.9 Flipping a Dictionary into a Table ⇒ +x

This command flips a dictionary into a table and will be covered in detail in the table chapter.

```
 d1
time|12:00 12:01 12:10
temp|25 25.1 25.6
```

```
 +d1
time  temp
----- ----
12:00 25
12:01 25.1
12:10 25.6

 d1~+d1
0b
```

6.10 Functions that operate on each value in a dictionary

There are a number of simple functions on dictionaries that operate on their values. If 'f' is a function then f applied to a dictionary returns a dictionary with the same keys and the values resulting from the application of 'f'.

- `-d : Negate`
- `d + N : Add N to d`
- `d - N : Subtract N from d`
- `d * N : Multiple d by N`
- `d % N : Divide d by N`
- `|d : Reverse`
- `<d : Sort Ascending`
- `>d : Sort Descending`
- `~d : Not d`
- `&d : Given d:x!y output each x, y times, where y must be an integer`
- `=d : Given d:x!y y!x`

Examples

```
 d2
 0|37.4
10|46.3
 1|0.1

 -d2
 0|-37.4
10|-46.3
 1|-0.1

 d2+3
 0|40.4
10|49.3
 1|3.1

 d2-1.7
 0|35.7
```

```
10|44.6
 1|-1.6

 d2*10
 0|374
10|463
 1|1

 d2%100
 0|0.374
10|0.463
 1|0.001
```

6.11 Functions that return over values from a dictionary

There are functions on dictionaries that operate over the values. Given function f applied to a dictionary d, f d returns one or more values without the original keys.

- *d : First value

```
 d0
pi|3.14
e |2.72
c |3e+08

 *d0
3.14
```

7 User-defined Functions

User-defined functions are treated as verbs in k9, so can benefit from the adverbs.

Multi-line functions must be defined with an initial space on all lines except the first. These can be loaded into memory by using the [load], page 82, command and then used by calling by name.

```
Func {[a;b]a+b}
```

7.1 Function arguments

Functions default to arguments of x, y, and z for the first three parameters but one can explicitly spell out these or other argument names. Given k9's terseness, many k9 programmers prefer short variable names. But this is a matter of taste.

```
 f1:{x+2*y}        / implicit arguments x and y
 f1[2;10]
22

 f2:{[x;y]x+2*y}   / explicit
 f2[2;10]
22

 f3:{[g;h]g+2*h}   / explicit arguments other than x and y
 f3[2;10]
22

 f4:{[please;dont]please+2*dont} / longer argument names are possible
 f4[2;10]
22

 @f1
`.
```

7.2 Function Defitions

Functions can have intermediate calcuations and local variables. Function local variables do not affect the values of global variables.

```
 a:3;a
3

 f:{a:12;x:sqrt x;x+a}

 f 10
15.16228

a
3
```

7.3 Function Return

Function return the last value in the definition unless the definition ends with a semicolon in which case the function returns nothing.

```
 f:{x+2;};f 10  / returns nothing because of final semi-colon

 f:{x+2;27};f 10  / returns the last value which is 27
27

 f:{x+2};f 10
12
```

7.4 Calling functions

Functions, like lists, can be called in a variety of ways. Typically one uses square bracket notation when the function takes multiple argruments. If a function is called with fewer than the required number of arguments then it will return a function that requires the remaining arguemnts.

```
 f1:{[x] x}
 f2:{[x;y](x;y)}
 f3:{[x;y;z](x;y;z)}

 f1[`a]
`a

 f2[37;`a]
37
a

 f3["hi";37;`a]
hi
37
a

 f2[37]
{[x;y](x;y)}[37]
 f2[;`a]
{[x;y](x;y)}[;`a]

 f1[`a]
`a
 f1 `a
`a
 f1@`a
`a
```

7.5 Anonymous functions

It's possible to define a function without naming it. If the function is to be used in just one place, this can make sense.

```
 {x+2*y}[2;10]
22
```

7.6 Recursive functions

k9 allows one to define a function which calls itself. Care should be taken to avoid an infinite loop.

```
 f:{$[x>1;x*f@x-1;1]};f 5
120
```

7.7 Chained functions

It may be necessary to define a function to call a function without specifying arguments. Imagine this trivial case.

```
 fa:{!x}
 fb:{x+2} fa
{x+2}
^
:rank
```

In order to succeed fa needs to have an `@` in the definition of fb. This is required because fb calls fa without specifying an argument for fa. So the argument for fb becomes the argument for fa so the net effect is 2 + !3.

```
 fa:{!x}
 fb:{x+2} fa@
 fb 3
2 3 4
```

8 Expression Evaluation

k9 has a compact way to evaluate expressions on dictionaries. This allows one to use the dictionary keys as local variables to get to results with less code.

`Expr :a+b`

8.1 expr ⇒ x :y

Evaluate expression(s) `y` on table or dictionary `x`. The space between `x` and `:` is required otherwise it becomes set, i.e., assignment.

```
 d:[city:`Chongqing`Shanhai`Beijing`Delhi`Chengdu]; d
city|`Chongqin`Shanhai`Beijing`Delhi`Chengdu

 d,:[country:`cn`cn`cn`in`cn;size:30 24 22 17 16] ; d
city   |`Chongqin`Shanhai`Beijing`Delhi`Chengdu
country|`cn`cn`cn`in`cn
size   |30 24 22 17 16

 d :&country=`in       / using country like a local variable
,3

 d :sum'size@=country  / sum size by country
cn|92
in|17

 d :+/size*country=`cn
92
```

As keys come in scope as variables, variables fall out of scope. One can evaluate the variable name as string to get around this.

```
 d :sum size@&size>20
76

 x:20;
 d :sum size@&size>x  / out of scope
!value

 d :sum size@&size>."x"
76
```

Reorder a table using column index and expresssions.

```
 t:[[]x:`b`a`c;y:1 3 2];t
x y
- -
b 1
a 3
c 2
```

```
 t@<t :y
x y
- -
b 1
c 2
a 3

 t@<t`y
x y
- -
b 1
c 2
a 3
```

9 Named Functions

This chapter covers the built-in named functions. This includes some math (eg. sqrt but not +), wrapper (eg. count for #) and range (eg. within) functions.

[Hcount], page 52, [Hfirst], page 52, [Hlast], page 52, [Hmin], page 52, [Hmax], page 52, [Hsum], page 52, [Hdot], page 52, [avg], page 53, [var], page 53, [dev med mode countd ..] [sqrt], page 9, [sqr], page 54, [exp], page 53, [log], page 53, [sin], page 54, [cos], page 54, [div], page 9, [mod], page 9, [bar], page 56, [in], page 55

9.1 count ⇒ count x

Same as [count], page 17.

```
 count 5 1 2
3
```

9.2 first ⇒ first x

Same as [first], page 8.

```
 first 5 1 2
5
```

9.3 last ⇒ last x

Retrieve the last element of list x.

```
 last 5 1 2
2
```

9.4 min ⇒ min x

Retrieve the minimum element of list x. Same as `|/`.

```
 min 5 1 2
1
```

9.5 max ⇒ max x

Retrieve the maximum element of list x. Same as `&/`.

```
 max 5 1 2
5
```

9.6 sum ⇒ sum x

Compute the sum of list x. Same as `+/`.

```
 sum 5 1 2
8
```

9.7 dot ⇒ dot x

Compute the dot product of list x. Same as `+/x*x`.

```
 dot 1.2 1.3 1.4
5.09

 {+/x*x} 1.2 1.3 1.4
5.09
```

9.8 avg ⇒ avg x

Compute the average of list x. Same as `+/x%#x`.

```
 avg 2 4 7 10
5.75

 {+/x%#x} 2 4 7 10
5.75
```

9.9 var ⇒ var x

Compute the variance of list x.

```
 var !10
8.25

 {(+/x*x:x-avg x)%#x}@!10
8.25
```

9.10 dev ⇒ dev x

TBD

9.11 med ⇒ med x

TBD

9.12 mode ⇒ mode x

TBD

9.13 countd ⇒ countd x

TBD

9.14 exp ⇒ exp x

Compute the exponent of x, i.e. e^x.

```
 exp 1
2.718282
```

9.15 log ⇒ log x

Log computes the natural log.

```
 log 10
2.302585
```

9.16 sin ⇒ sin x

sin computes the sine of x where x is in radians.

```
 sin 0
0f
 sin 3.1416%2
1.
```

9.17 cos ⇒ cos x

cos computes the cosine of x where x is in radians.

```
 cos 0
1f
 cos 3.1416%4
0.7071055
```

9.18 sqr ⇒ sqr x

Compute the square of x.

```
 sqr 2
4.0
```

9.19 prm ⇒ prm x

Write out all permutations of integers up x-1. Display them as columns.

```
 prm 3
0 1 1 0 2 2
1 0 2 2 0 1
2 2 0 1 1 0
```

9.20 sums ⇒ sums x

Compute the running sum of list x. Sames as `+\`.

```
 sums !10
0 1 3 6 10 15 21 28 36 45
```

9.21 deltas ⇒ deltas x and x deltas y

Compute the difference between each element in list x and the previous value. If delta is called with two parameters then x will be used as the first value to delta instead of the default 0.

```
 deltas 1 5 10
```

```
1 4 5

 1 deltas 1 5 10
0 4 5
```

9.22 has ⇒ x has y

Determine whether vector x has element y. Simliar to [in], page 55, but with the arguments reversed.

```
 `a`b`c`a`b has `a
1b

`a`b`c`a`b has `d
0b

 `a`b`c`a`b has `a`b`x`y`z
11000b

 (1 2;4 5 6;7 9)has(1 2;8 9)
10b
```

9.23 bin ⇒ x bin y

Given a sorted (increasing) list x, find the greatest index, i, where y>x[i].

```
 n:exp 0.01*!5;n
1 1.01005 1.020201 1.030455 1.040811
 n bin 1.025
3
```

9.24 in ⇒ x in y

Determine if x is in list y. Similar to [has], page 55, but arguments reversed.

```
 `b in `a`b`d`e
1b
 `c in `a`b`d`e
0b
```

9.25 within ⇒ x within y

Test if x is equal to or greater than y[0] and less than y[1].

```
 3 within 0 12
1b

0 within 0 12
1b

12 within 0 12
```

```
0b

23 within 0 12
0b
```

9.26 bar ⇒ x bar y

For each value in y determine the maximum multiple of x that is less than or equal to each y.

```
 10 bar 9 10 11 19 20 21
0 10 10 10 20 20
```

9.27 msum ⇒ x msum y

Compute the length x moving sum of list y.

```
 3 msum !10
0 1 3 6 9 12 15 18 21 24
```

9.28 mavg ⇒ x mavg y

Compute the length x moving average of list y.

```
 3 mavg !10
0 0.3333333 1 2 3 4 5 6 7 8
```

10 Knit Functions

These functions modify lists and dictionaries given a list of indices and functions or values to replace.

10.1 amend ⇒ @[r;i;f[;y]]

Replace the values in list / dictionary r at indices i with element f or function f and parameter y. The original list is not modified. Indices are rows for lists and keys for dictionaries.

action	**f**	**y**
[amend1], page 57,	element	n/a
[amend2], page 57,	:	element/list
[amend3], page 58,	function	n/a
[amend4], page 58,	function	2nd param

Amend to element.

```
 r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6

 @[r;0 3;29]      / change the first and fourth rows
29
2 3
4 5
29

 r               / r doesn't change
0 1
2 3
4 5
6
```

Amend with element/array. If using an array, then the `i` and `y` must be arrays of equal length.

```
 r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6
```

```
 @[r;1 2;:;(0;3 5)]
0 1
0
3 5
6
```

Amend with function f[r] at indices i.

```
 r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6

 @[r;1 2;sqrt]
0 1
1.414214 1.732051
2.0 2.236068
6

 d:[x:`a`b`c;y:9 4 1];d / dictionary example
x|`a`b`c
y|9 4 1

 @[d;`y;sqrt]
x|`a`b`c
y|3 2 1.
```

Amend with function f[r;y] at indices i.

```
 r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6

 @[r;1 2;*;10 100]
0 1
20 30
400 500
6

 d:[x:`a`b`c;y:9 4 1];d / dictionary
x|`a`b`c
y|9 4 1

 @[d;`y;*;10]
x|`a`b`c
y|90 40 10
```

10.2 dmend ⇒ .[r;i;f[;y]]

Similar to [amend], page 57, but using `i` to fully index `r`.

action	f	y
[dmend1], page 59,	element	n/a
[dmend3], page 59,	function	n/a
[dmend4], page 59,	function	2nd param

Dmend to element.

```
  r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6

  .[r;0 1;12]       / modify the entry at [0;1]
0 12
2 3
4 5
6
```

Dmend with function f[r] at indices i.

```
  r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6

  .[r;1 1;sqrt]
0 1
2.0 1.732051
4 5
6
```

Dmend with function f[r;y] at indices i.

```
  r:(0 1;2 3;4 5;6);r
0 1
2 3
4 5
6

  .[r;1 1;+;100]
0 1
2 103
4 5
```

6

11 I/O and Interface

This chapter covers reading and writing to file (File I/O), interprocess communication (IPC), and working with pre-built shared libraries (csv, json, lz4, and zstd).

```
 i/o(*enterprise)
 0: [r line], page 62/[w line], page 62, line
 1: [r char], page 62/[w char], page 62, char
*2: [r data], page 63/[w data], page 63, data
*3: [k-ipc], page 64, [3set], page 64
*4: [https], page 64, [4get], page 64
 5: [ffi], page 68/[import], page 68

 `[csv], page 65?`[csv], page 65, t;`[json], page 65?`[json], page 65, t
 `[compress], page 65?`[compress], page 65, t;`[compress], page 65?`[compress],
page 65, t
```

11.1 File I/O

k9 reads and writes to files in three different formats including line (list of strings), char (list of characters), and data (k9 format).

11.1.1 Example File I/O

Let's begin with a simple example to creat a sample table, write to csv and then read it back in again. We'll use a function so the table can easily grow to a larger size.

```
 g:{+`s`f`i!(x?"abc";x?10.;x?10)} / generate function for table

g[5]                             / generate table with 5 rows
s f          i
- ---------- -
c 1.296544   4
c 0.03771765 7
c 3.371475   0
b 1.352739   4
b 1.187619   5

`csv@g[5]                        / convert table to csv string
"s,f,i\nb,1.08707,3\na,5.506882,0\nc,8.938667,1\nc,6.217895,6\nb,1.542842,6\n"

 "sample.csv"1:`csv@g[5]         / write to sample.csv file
"sample.csv"

 1:"sample.csv"                  / read from file
"s,f,i\na,3.669518,1\na,0.001226037,7\nb,2.792163,3\nc,8.539221,7\nb,9.333188,7\n"

 `csv?1:"sample.csv"             / read from file and convert from csv
s f           i
```

```
- ----------- -
a 3.669518    1
a 0.001226037 7
b 2.792163    3
c 8.539221    7
b 9.333188    7
```

Given the small size of the file all this happens too quickly to notice the speed. Let's give it a go with a bigger file.

```
 \t "sample.csv"1:`csv@g[_1e7]
2587
 \t t:`csv?1:"sample.csv"
388
```

Ten million rows via csv in under 1/2 a second. Of course results will depend on hardware and number of columns also.

11.1.2 read line ⇒ 0:x

Read from file x.

```
 0:"some.csv"
a,b
1,3.
2,4.
```

11.1.3 write line ⇒ x 0:y

Output to string x (file name or null for stdout) the list of strings in y. y must be a list of strings. If y is a single string then convert to list via [enlist], page 14.

```
 ""0:("blue";"red")      / "" represents stdout
blue
red

 "file.txt" 0: ("blue"; "red") / write to file, one line per element
```

11.1.4 read char ⇒ 1:x

Read in file x (string).

```
 / file.bin contains the binary value 0x0010FA37
 1:"file.bin"
0010fa37

 / file.txt contains the ascii string 37\ntest\n
 1:"file.txt"
"37\ntest\n"
```

11.1.5 write char ⇒ x 1:y

Write to file x (string) data y (list of characters).

```
 "some.txt"1:"0123ABab"
"some.txt"
```

```
 1:"some.txt"
"0123ABab"

 "some.csv" 1:`csv [[]a:1 2;b:3. 4.]
"some.csv"

 1: "some.csv"
"a,b\n1,3.00\n2,4.00\n"

 `csv?1:"some.csv"
a b
- -
1 3
2 4
```

11.1.6 read data ⇒ 2: x

Enterprise Only

Load file, eg. csv or from a (x 2: y) save. For the latter, one can find a “save then load” example in the next section.

```
 2:`t.csv
s    t        e p  z
---- -------- - -- ----
AABL 09:30:00 D 11 4379
AABL 09:30:00 B 40 3950

 2:`r                / read from file
a          b          c          d          e
---------- ---------- ---------- ---------- ----------
0.5366064  0.8250996  0.8978589  0.4895149  0.6811532
0.1653467  0.05017282 0.4831432  0.4657975  0.4434603
0.08842649 0.8885677  0.23108    0.3336785  0.6270692
..
```

11.1.7 write data ⇒ x 2: y

Enterprise only

Save to file x non-atomic data y (e.g., lists, dictionaries, or tables).

This example saves 100 million 8-byte doubles to file. The session is then closed and a fresh session reads in the file. Both the write (420 ms) and compute statistics from the file have impressive speeds (146 ms) given the file size (800 MB).

```
 n:_1e8
 r:+`a`b`c`d`e!5^n?1.;r
 `r 2:r              / write to file
```

Start new session.

```
 \t r:2:`r;select avg a,sum b, max c, min d, sum e from r
148
```

11.2 IPC

k9 has the ability to comunicate between separate k9 instances via interprocess communication (IPC). These processes can be on the same or separate machines. A user will start multiple instances specifying the port (via the command line argurment -p) that the k9 session will use, opening a handle to that port, and then running remote commands via the handle and `3:` or `4:`. The commands to be run are sent as a string, eg. “3+12”.

11.2.1 k-ipc ⇒ 3:x

Open a connection to port `x` and return a handle. If the port has been forwarded from another machine (eg. ssh -L 1280:server.com:1280 laptop.com) then this handle allows remote execution.

```
 / before running this session start another k session in another
 / terminal window specifying that port 1251 should be used
 / k -p 1251
 h:3:1251 / create handle to port 1251 and save to h
 h        / h is an integer
4
```

11.2.2 set ⇒ x 3:y

Exectute string y using handle x. Handle x should have already been created using k-ipc. `set` is asynchronous, thus once the command is sent the session immediately returns to the user. One will not be able to return any results via `set`. If a result is required then one should use [4get], page 64.

```
 h:3:1251              / create handle to port 1251
 h 3:"a:12"            / set a to 12 on remote session
 h 3:"n:_1e8;b:1?n?1."  / set b to a random number on remote
```

11.2.3 https ⇒ 4:x

TBD

11.2.4 get ⇒ x 4:y

Execute string y using handle x. Handle x should have already been created using k-ipc. `get` is synchronous, thus once the command is sent the session waits until there result is ready.

```
 h:3:1251              / create handle to port 1251
 h 3:"a:12"            / set a to 12 on remote session
 h 3:"n:_1e8;b:1?n?1."  / set b to a random number on remote
 h 4:"a"
12

 h 4:"b"
,0.07820029
```

```
 a:"local"              / set a locally to a string value
 a
"local"

4:"a"
12
```

11.3 ffi/import

These topics will be covered in the Chapter 12 [FF], page 67, chapter.

11.3.1 export, import csv ⇒ ‘csv@x, ‘csv?x

Enterprise only

csv tools allows one to export tables into csv and import from csv. Generally these functions are used with file i/o 1:

```
 "test.csv"1:`csv@[[]x:`a`b;y:3 2]  / write sample table to file
"test.csv"

 `csv?1:"test.csv"                  / read csv from file
x y
- -
a 3
b 2
```

11.3.2 export, import json ⇒ ‘json@x, ‘json?x

Export object x to json format or import string x from json format. The json shared library will be automatically pulled from the Shakti website into the current directory. Use of symbolic links with path on the binary may cause issues.

```
 "test.json"1:`json@[[]x:`a`b;y:3 2]
"test.json"

`json?1:"test.json"
(,"x";,"y")!(,"a";3)
(,"x";,"y")!(,"b";2)
```

11.3.3 lz4 zstd ⇒ ‘lz4 ‘zstd

lz4 and zstd are compression algorythms which can be used to reduce the size of files. As with other foreign libraries k9 does the work behind the scenes. Here is an example of writing a large csv file to disk and reading it back in. In the second (lz4) and third (zstd) examples the csv’s are compressed and decompressed with lz4 and zstd respectively. One can look at the file sizes to see how well the compression worked.

```
 g:{+`s`f`i!(x?"abc";x?10.;x?10)}

"t.csv"1:`csv@g[_1e7]
"t.csv"
```

```
\t tc:`csv?1:"t.csv"            / read from csv
314

"t.lz4"1:`lz4@`csv@g[_1e7]      / write to lz4
"t.lz4"

\t tz:`csv?`lz4?1:"t.lz4"       / read from lz4 compressed csv
525

 "t.zstd"1:`zstd@`csv@g[_1e7]   / write to zstd
"t.zstd"

 \t tz:`csv?`zstd?1:"t.zstd"    / read from zstd compressed csv
535
```

12 Foreign Functions

```
[python], page 67:from k import k;k('+',2,3);[nodejs], page 67:k=require('k').k;k('+',
*5: [ffi], page 68/[import], page 68
```

Enterprise only

k9 is able to interface with other programing laguages (python and nodejs) via shared libraries and the foreign function interface (ffi).

12.1 python

In order to have python call k, one has to download the shared object file (k.so for linux and m.so for mac) and save it in the appropriate directory.

```
user1@hw1:~$ python3 -m site --user-site
/Users/user1/Library/Python/3.8/lib/python/site-packages
```

Once the file is stored (and it must be called k.so even on the mac) then one can start a python session and call k.

```
user1@hw1:~$ python3
Python 3.8.5 (v3.8.5:580fbb018f, Jul 20 2020, 12:11:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import k
>>> k.k('x:5?10.')
>>> k.k('x')
[7.691266643814743, 6.843659253790975, 6.399056380614638, 9.637191963847727, 4.2506528
>>> len(k.k('x'))
5
>>>
```

12.2 nodejs

One can call k9 from nodejs using the k.node shared libary. (If using a mac then download m.node and rename it k.node.)

From the terminal...

```
% node
Welcome to Node.js v14.17.4.
Type ".help" for more information.
> k=require('./k')
{}
> k.k('+/!20')
190
>
```

There's also the ability to send information from k9 to nodejs via [k-ipc], page 64.

ipc.js file

```
p=1299;s=new require('net').Server();
s.listen(p, function()            {console.log(`Port ${p} is open.`);})
```

```
s.on('connection',function(socket){console.log(`Port ${p} is connected.`);
 socket.on('data',     function(chunk) {console.log(`Data: ${chunk.toString()}` );});
 socket.on('end',      function()      {console.log(`Port ${p} is disconnected.`);});
});
```

Start nodejs

```
% node ipc.js
```

In k9...

```
 h:3:1299
 h 3: "\nLine1\nLine2\n"
```

In nodejs...

```
Port 1299 is open.
Port 1299 is connected.
Data:
Line1
Line2
```

12.3 ffi ⇒ 5:

Load shared library.

Contents of file 'a.c'

```
int add1(int x){return 1+x;}
int add2(int x){return 2+x;}
int indx(int x[],int y){return x[y];}
```

Compile into a shared library (done on macos here)

```
% clang -dynamiclib -o a.so a.c
```

Load the shared library into the session.

```
 f:"./dev/a.so"5:{add1:"i";add2:"i";indx:"Ii"}
 f[`add1] 12
13
 f[`indx][12 13 14;2]
14
```

12.4 import ⇒ x 5:y

Import as name x library y.

```
 .(:;`json;5:`json) / using dot notation
```

13 Tables

k9 has the ability to store data in a tabular format containing named columns and rows of information as tables. If the data to be stored and queried is large, then you should use tables. This chapter introduces the different types of data tables available in k9. Table and utable are very similar and as you'll see in the Chapter 14 [kSQL], page 71, chapter are easy to query. In the Chapter 20 [Benchmark], page 94, chapter, you'll see that tables are fast to save, read, and query.

```
 [table], page 69, t:[[]n:`b`c;i:2 3]
[utable], page 70, u: [[n:`b`c]i:2 3]
```

13.1 table

The table is the most basic of the three types. A table consists of columns and rows of information where each column has a name. Tables can be created in three different ways (1) specification via table format, (2) flipping a dictionary, or (3) reading in from a file.

13.1.1 Table format

Tables can be created with the table square bracket notation.

As an example, let's create a table with two columns named "a" and "col2" having three rows. The syntax is to surround the definition with square brackets and then have a first element of empty square brackets. Following those brackets comes first column name, colon, and the list of values, then the second column, and continuing for all the columns. For keyed tables, the initial square brackets will contain key column names as we will discuss later.

```
 [[] a:1 20 3; col2: 3.6 4.8 0.1]
a  col2
-- ----
 1 3.6
20 4.8
 3 0.1

 [[] a:1; col2:3.6]   / will error :class as lists required
[[] a:1; col2:3.6]

:class

 [[] a:,1; col2:,3.6] / using list will succeed
[[]a:,1;col2:,3.6]
```

13.1.2 Dictionary format

Tables can also be created by flipping a dictionary into a table.

```
 +`a`col2!(1 20 3;  3.6 4.8 0.1)      / +columnnames!(values)
a  col2
-- ----
 1 3.6
20 4.8
 3 0.1
```

13.1.3 File import

Tables can also be created by reading in a file.

t.csv

```
a, col2
1, 3.6
20, 4.8
3, 0.1
```

Use load file `2:x` which returns a table.

```
 2:`t.csv
a   col2
-- -----
 1  3.6
20  4.8
 3  0.1
```

13.2 utable

utable (or key table) is a table where some of the columns are keyed. The combination of those columns should not have two rows with the same values. This must be enforced by the application.

```
 [[d:2020.09.08 2020.09.09 2020.09.10]p:140 139 150]
d         |p
----------|---
2020.09.08|140
2020.09.09|139
2020.09.10|150

 `d^ [[]d:2020.09.08 2020.09.09 2020.09.10;p:140 139 150]
d         |p
----------|---
2020.09.08|140
2020.09.09|139
2020.09.10|150
```

14 kSQL

kSQL is a powerful query language for tables. It has similarities to ANSI SQL but additional features to make it easier to work with ordered data, such as time series data.

```
Database
{[select], page 71|[update], page 74} [A], page 72, [by B], page 73, from T [where
C], page 73; [delete], page 75, from T [where C], page 73

[Joins], page 75
x,y      / [insert], page 80, [upsert], page 80, [union], page 75, equi-and-asof [left
page 76
x+y      / equi-and-asof outerjoin (e.g. combine markets through time)
x#y      / take/intersect innerjoin
x_y      / drop/difference
```

14.1 Queries

Queries can be done either via the kSQL or functional forms. The kSQL form is general simplier to write while the functional form allows one to more easily run multithreaded or build queries programmatically.

14.1.1 select

There a number of ways to return a complete table with kSQL. You can use the table name, a kSQL query without columns, or a fully specified query with columns.

```
 n:5;t:[[]x:!n;y:sin !n]
 t
x y
- ----------
0 0.
1 0.841471
2 0.9092974
3 0.14112
4 -0.7568025

 select x,y from t
x y
- ----------
0 0.
1 0.841471
2 0.9092974
3 0.14112
4 -0.7568025

 `x`y#t
x y
- ----------
1 0.841471
```

```
2 0.9092974
3 0.14112
4 -0.7568025
```

14.1.2 A

A is the list of fields to return from the select or update query. A can create new column names.

```
 n:5;t:[[]x:!n;y:sin !n];t
x y
- ----------
0 0.
1 0.841471
2 0.9092974
3 0.14112
4 -0.7568025

 select x from t
x
-
0
1
2
3
4

`x#t
x
-
0
1
2
3
4


 select y,x from t    / reorder columns
y          x
---------- -
0.         0
0.841471   1
0.9092974  2
0.14112    3
-0.7568025 4

 `y`x#t
y          x
```

```
---------- -
0          0
0.841471   1
0.9092974  2
0.14112    3
-0.7568025 4

 select x,z:y from t  / create new column z
x z
- ----------
0 0.
1 0.841471
2 0.9092974
3 0.14112
4 -0.7568025
```

14.1.3 by B

kSQL also has a way to group rows using `by B`. The result is a utable where the key is determined by the grouping clause.

```
 n:5;t:[[]x:!n;y:sin !n]
 select sum y by x>2 from t
x|y
-|----------
0|1.750768
1|-0.6156825

 t :+/'y@=x>2
0|1.750768
1|-0.6156825
```

14.1.4 where C

kSQL makes it easy to build up a where clause to filter down the table. C is the list of constraints.

```
 n:5;t:[[]x:!n;y:sin !n]
 select from t where x>0
x y
- ----------
1 0.841471
2 0.9092974
3 0.14112
4 -0.7568025

 (:x>0)#t
x y
- ----------
1 0.841471
```

```
2 0.9092974
3 0.14112
4 -0.7568025

select from t where (x>0), y within 0 .9
x y
- --------
1 0.841471
3 0.14112

 (:x>0;:y within .0 .9)#t
x y
- --------
1 0.841471
3 0.14112
```

14.1.5 Query with By and Where

```
 n:5;t:[[]x:!n;y:sin !n]
 select sum y by x>2 from t where y>0
x|y
-|--------
0|1.750768
1|0.14112

 ((:y>0)#t) :+/'y@=x>2
0|1.750768
1|0.14112
```

14.1.6 update

update allows one to modify values without specifying all the fields that pass through. update is also used to add new columns in a table. update does not, by itself, save the modifications to the table. If you want to preserve those modifications, use assignment.

```
 t:[[]x:`a`b`c;y:1 2 3];t
x y
- -
a 1
b 2
c 3

 update y+18 from t
x y
- --
a 19
b 20
c 21
```

```
 update z:y+18 from t
x y z
- - --
a 1 19
b 2 20
c 3 21

 t:update z:y+18 from t; / save the updates into table t
 t                       / Now t has the updated values.
x y z
- - --
a 1 19
b 2 20
c 3 21
```

14.1.7 delete

Delete rows from a table that satisfy one or more conditions. Currently not working.

```
 t:[[]x:`a`b`c;y:1 2 1];t
x y
- -
a 1
b 2
c 1

 delete from t where y>1
!value

 delete from t where x=`c,y=1
!nyi
```

14.2 Joins

k9 has a number of methods to join tables together which are described below. In this section t, t1 and t2 represent tables and k, k1 and k2 represent utables.

join	**syntax**
[union], page 75,	t1,t2
[leftjoin], page 76,	t,k
[outer], page 77,	k1,k2
[asof], page 78,	t,k
[asof+], page 79,	k1+k2

14.2.1 union join ⇒ t1,t2

Union join table t1 with table t2. The tables should have the same columns and the join results in a table with t2 appended to t1. If the tables do not have the same columns then return t1.

```
 t1:[[]s:`a`b;p:1 2;q:3 4]
```

```
 t2:[[]s:`b`c;p:11 12;q:21 22]
 t1
s p q
- - -
a 1 3
b 2 4

 t2
s p  q
- -- --
b 11 21
c 12 22

 t1,t2
s p  q
- -- --
a  1  3
b  2  4
b 11 21
c 12 22
```

14.2.2 leftjoin ⇒ t,k

leftjoin table t with utable k. Result includes all rows from t and the values from table k having the same key values. If a row of t has key values not found in k, then the t values are shown and 0 for the columns coming from k.

```
 t:[[]s:`a`b`c;p:1 2 3;q:7 8 9]
 k:[[s:`a`b`x`y`z]q:101 102 103 104 105;r:51 52 53 54 55]
 t
s p q
- - -
a 1 7
b 2 8
c 3 9

 k
s|q   r
-|--- --
a|101 51
b|102 52
x|103 53
y|104 54
z|105 55

 / t.s includes the value `c. Because k.s does not include c,
 / the last row shows a 0 under the r column (which comes from k).
 t,k
```

```
s p q   r
- - --- --
a 1 101 51
b 2 102 52
c 3   9  0
```

14.2.3 outer join ⇒ k1,k2

Outer join utable k1 and k2.

```
 k1:[[s:`a`b]p:1 2;q:3 4]
 k2:[[s:`b`c]p:9 8;q:7 6]
 k1
s|p q
-|- -
a|1 3
b|2 4

 k2
s|p q
-|- -
b|9 7
c|8 6

 k1,k2
s|p q
-|- -
a|1 3
b|9 7
c|8 6

 k1:[[s:`a`b]p:1 1;q:10 10]
 k2:[[s:`b`c]p:2 2;q:20 20]
 k3:[[s:`c`d]p:3 3;q:30 30]

k1,k2,k3       / joining a 3rd
s|p q
-|- --
a|1 10
b|2 20
c|3 30
d|3 30

 ,/(k1;k2;k3)  / join over
s|p q
-|- --
a|1 10
b|2 20
```

```
c|3 30
d|3 30
```

14.2.4 asof join ⇒ t,k

Asof joins each row rt of table t to a row rk in utable k (keyed by time) provided rk has the maximum time value of any row in k while obeying the constraint that the time value in rt >= the time value of rk. Intuitively, rk should be the row in k that is most up-to-date with respect to rt.

```
 t:[[]t:09:30+5*!5;p:100+!5];t
t     p
----- ---
09:30 100
09:35 101
09:40 102
09:45 103
09:50 104

 k:[[t:09:32 09:41 09:45]q:50 51 52];k
t    |q
-----|--
09:32|50
09:41|51
09:45|52

 / Notice below the t row at 09:45 is linked with the k row at 09:45.
 / The k row at 09:41 is not linked with any t row.
 / By contrast, both the 09:35 and the 09:40 rows of t
 / are linked to the 09:32 row of k.
 t,k

t     p   q
----- --- --
09:30 100  0
09:35 101 50
09:40 102 50
09:45 103 52
09:50 104 52
```

Scaling this up to a bigger set of tables one can see the performance of k9 on joins.

```
 N:_1e8;T:[[]t:N?`t 0;q:N?100];5#T
t            q
------------ --
00:00:00.001 44
00:00:00.002 46
00:00:00.002 48
00:00:00.003 35
00:00:00.003 43
```

```
 n:_1e5;K:[[t:n?`t 0]p:n?100];5#K
t           |p
------------|--
00:00:00.481|54
00:00:00.961|63
00:00:01.094|67
00:00:01.479|16
00:00:01.917|58

 \t T,K
222
```

14.2.5 asof+ join ⇒ k1+k2

Asof+ joins allows one to aggregate over markets to find the total available at a given time. The utables need to be specified with `` `a``. The effect is to merge the two key fields (the field t in this case) and for each row rk1 from table k1, add the non-key field (bs in this case) from rk1 to the bs field of the most recent row in k2 whose t value is less than or equal to the t value in rk1. And symmetrically for each row of table k2.

```
 k1:`a [[t:09:30+5*!5]bs:100*1 2 3 2 1];k1
t    |bs
-----|---
09:30|100
09:35|200
09:40|300
09:45|200
09:50|100

 k2:`a [[t:09:32 09:41 09:45]bs:1 2 3];k2
t    |bs
-----|--
09:32| 1
09:41| 2
09:45| 3

 k1+k2
t    |bs
-----|---
09:30|100
09:32|101
09:35|201
09:40|301
09:41|302
09:45|203
09:50|103
```

14.3 Insert and Upsert

One can add data to tables via insert or upsert. The difference between the two is that insert adds data to a table while upsert on some key x will replace the values if x is present in the target table or insert x with its associated value otherwise.

14.3.1 insert ⇒ t,d

Insert dictionary d into table t.

```
 t:[[]c1:`a`b`a;c2:1 2 7];t
c1 c2
-- --
a   1
b   2
a   7

 t,`c1`c2!(`a;12)
c1 c2
-- --
a   1
b   2
a   7
a  12

 t,`c1`c2!(`c;12)
c1 c2
-- --
a   1
b   2
a   7
c  12
```

14.3.2 upsert ⇒ k,d

Insert dictionary d into utable k.

```
 k:[[c1:`a`b`c]c2:1 2 7];k
c1|c2
--|--
a | 1
b | 2
c | 7

 k,`c1`c2!(`a;12)
c1|c2
--|--
a |12
b | 2
c | 7
```

```
 k,`c1`c2!(`b;12)
c1|c2
--|--
a | 1
b |12
c | 7

 k,`c1`c2!(`d;12)
c1|c2
--|--
a | 1
b | 2
c | 7
d |12
```

15 System

k9 comes with a few system functions and measurement commands. The commands allow you to load a script, change the working directory, measure execution times and memory usage, and list defined variables.

```
System
\l a.k  [load], page 82
\t:n x  [timing], page 82
\u:n x
\v      [variables], page 82
\w      [memory], page 82

\cmd    [OS command], page 83
```

15.1 load ⇒ \l a.k

Load a text file of k9 commands. The file name must end in .k.

```
 \l func.k
\l func.k

\l func.k9  / will error as not .k
:nyi
```

15.2 timing ⇒ \t x or \t:n x

List time elapsed in milliseconds in evaluating x. If n is supplied then repeat x, n times.

```
 \t ^(_10e6)? _1e8     / sort 10 million numbers
227

 \t:10 ^(_10e6)? _1e8  / perform 10 times the sort
2027
```

15.3 u ⇒ \u x or \u:n x

TBD

15.4 variables ⇒ \v

List variables

```
a:1;b:2;c:3

 \v
,3
i|`a`b`c
```

15.5 memory ⇒ \w

List memory usage

```
 \w
930

 r:(_10e6)?10

 \w
2098106
```

15.6 OS command ⇒ \cmd

Run any valid command via OS.

```
 \cd "/Users/user123/"

 \pwd
,"/Users/user123"

 \echo $PATH             / query environmental variable
,"/opt/local/bin:/usr/bin:/bin"
```

16 Control Flow

Though looping statements are not necessary in k9, if-then-else statements are sometimes useful.

`$[b;t;f] cond`

16.1 cond ⇒ $[b;t;f]

If b is non zero then x else y. x and y are not required to be of the same type.

```
 $[3>2;`a;`b]
`a

 $[2>3;`a;`b]
`b

 $[37;12;10]
12

 $[1b;`a`b!(1 2;3 4);`n]
a|1 2
b|3 4

 $[1b;a:3;b:2]
3
 a
3
 b   / is not set as the f case not evaluated
!value
```

17 Temporal Functions

k9 has functions (within `.z`) to get the current date and time to various degrees of precision. There are also functions to retrieve partial date and time using dot notation.

```
z.d date   2001.02.03
z.t time   12:34:56.789
z.T datetime
```

17.1 z.d date

These functions retrieve the date as day (d) or month (m) precision.

```
 .z.d
2023.03.11
```

17.2 z.t time

These functions retrieve the time millisecond precision.

```
 z.t
09:09:02.451
```

One could use the current time commands to measure run time but typically this is done via `\t`

```
 t1:z.t;(_2e8)?1.;t2:z.t;t2-t1
00:00:00.001

 \t (_2e8)?1.
610
```

17.3 z.T datetime

These functions retrieve the date and time as datetime.

```
 .z.T
2023.03.11 09:09:49.654
```

17.4 Temporal dot functions

These functions use dot notation to retrieve partial date and times.

```
 now:.z.v;now
11:27:18.049558016
 now.h
11
 now.r
11:27
 now.s
11:27:18
 now.t
11:27:18.049
 now.u
```

```
11:27:18.049558
 now.v
11:27:18.049558016
```

18 Errors

Given the terse syntax of k9, it likely won't be a surprise that error messages are also rather short. The errors are listed on the help page and described in more detail below.

```
error: [class], page 87, [rank], page 87, [length], page 87
[_type], page 87, [domain], page 87, [limit], page 87
stack [eparse], page 88, [_value], page 88
```

18.1 :class

Calling a function on mismatched types.

```
 3+`b
:class
```

18.2 :rank

Calling a function with too many parameters.

```
 {x+y}[1;2;3]
{x+y}[1;2;3]
^
:rank
```

18.3 :length

Operations on unequal length lists that require equal length.

```
 (1 2 3)+(4 5)
:length
```

18.4 :type

Calling a function with an unsupported variable type.

```
`a+`b
  ^
:type
```

18.5 :domain

Exhausted the number of input values

```
 -12?10   / only 10 unique value exist
:domain
```

18.6 :limit

Exceeded a limit above the software maximum, eg. writing a single file above 1GB.

```
 n:_100e6;d:+`x`y!(!n;n?1.);`d 2:d
n:_100e6;d:+`x`y!(!n;n?1.);`d 2:d

:limit
```

18.7 :nyi

Running code that is not yet implemented. This may come from running code in this document with a different version of k9.

2020.05.31 (c) shakti

```
 =+`a`b!(1 2;1 3)
a b|
- -|-
1 1|0
2 3|1
```

Aug 6 2020 16GB (c) shakti

```
 =+`a`b!(1 2;1 3)
=+`a`b!(1 2;1 3)
^
:nyi
```

18.8 :parse

Syntax is wrong. This may be due to mismatched parentheses or brackets, e.g., (), {}, [], "".

```
 {37 . "hello"
:parse
```

18.9 :value

Undefined variable is used.

```
 g    / assuming 'g' has not be defining in this session
:value
```

19 Examples

This chapter presents an example from finance, as this is one of the primary application domains for k. For those not familiar with this field, here is a short introduction.

19.1 A Tiny Introduction to Financial Market Data

Finance has a large amount of data associated with it. In this section finance data will be limited to price and transaction information which typically includes prices to buy and sell (called quotes) and transactions (called trades). This data includes the date, financial instrument symbol, time, and exchange / venue. Additionally quotes will have a bid and ask price (where the deal is willing to buy and sell) and trades will have a price and size of the reported transaction.

Although real prices are often shown as fractions of a whole amount, eg. eurusd might be at 1.1904, the actual trades are dealt in whole cents or currencies so can be represented as integers.

Let's start with a simple example of only times (t), bid (b), and ask (a).

```
 n:10
 T:^10:00+`t n?36e5  / sort randomly generated times
 B:100++\-1+n?3      / generate bids near 100, equivalent to: 100 + (+\((-1)+n?3))
 A:B+1+n?2           / generates asks 1 or 2 higher
 q:+`t`b`a!(T;B;A);q / build table t and then display
t            b   a
------------ --- ---
10:01:48.464 100 102
10:23:12.033 100 102
10:30:00.432 101 102
10:34:00.383 101 103
10:34:36.839 101 102
10:42:59.230 100 102
10:46:50.478 100 102
10:52:42.189  99 100
10:55:52.208  99 101
10:59:06.262  98  99
```

Here you see that at 10:42:59.230 the prices update to 100 and 102. The price one could sell is 100 and the price to buy is 102. You might think that 100 seems a bit high so sell there. Later at 10:59:06.262 you might have thought the prices look low and then buy at 99. Here's the trade table for those two transactions.

```
 t:+`t`p!(10:43:00.230 10:59:07.262:;100 99);t
t            p
------------ ---
10:43:00.230 100
10:59:07.262  99
```

You'll note that the times didn't line up, because it apparently took you a second to decide to trade. Because of this delay, you'll often have to look back at the previous prices to join trade (t) and quote (q) data.

Now that you've learned enough finance to understand the data, let's scale up to larger problems to see the power of k9.

19.2 Data Manipulation

Let's use k9 to generate a set of random quotes for a particular day and symbol.

```
 qs:`date`sym`time`exch`bid`ask           / quote table schema
 nf:d+*|d:(|-d),d:683 954 997 1000;       / normal func
 D:#[;2021.03.17]                         / date
 S:#[;`eurusd]                            / symbol
 T:^?[;_8.64e7]@                          / time in number of milliseconds
 E:?[;"ce"]                               / exchange
 B:11904++\-3+nf bin/:?[;*|nf]@           / bid price, start at 11904
 P:?[;2,2,8#1]@                           / bid/ask spread
 Q:{+qs!((D;S;T;E)@'x),(*;+/)@\:(B;P)@'x}  / generator quote table
 q:Q@_1e8;10#q
date       sym    time exch bid   ask
---------- ------ ---- ---- ----- -----
2021.03.17 eurusd    0 c    11904 11905
2021.03.17 eurusd    0 e    11904 11906
2021.03.17 eurusd    2 e    11902 11903
2021.03.17 eurusd    3 c    11902 11903
2021.03.17 eurusd    9 c    11904 11906
2021.03.17 eurusd    9 c    11904 11905
2021.03.17 eurusd   10 c    11904 11905
2021.03.17 eurusd   12 c    11904 11905
2021.03.17 eurusd   12 c    11904 11905
2021.03.17 eurusd   12 e    11904 11906
```

At this point let's run some basic statistics to see how quickly one can work with 100 million rows of data. On a reletaviley recent consumer laptop the spread calculation (the longest calculation of the bunch) is done in 350ms.

```
 select max bid,min ask from q
bid|18449
ask|5972

select mid:avg 0.5*bid+ask from q
[mid:14198.32]

select spread:avg ask-bid from q
[spread:1.200035]

select first bid, first ask from q
bid|11904
ask|11905

select last bid, last ask from q
bid|14906
```

```
ask|14907
```

19.3 Understanding Code Examples

In the shakti mailing list there are a number of code examples that can be used to learn best practices. In order to make sense of other people's codes, one needs to be able to efficiently understand k9 language expressions. Here is an example of how one goes about this process.

```
ss:{*{
      o:o@&(-1+(#y)+*x@1)<o:1_x@1;
      $[0<#x@1;((x@0),*x@1;o);x]}[;y]/:(();&(x@(!#x)+\:!#y)~\:y)
      }
```

This function finds a substring in a string.

```
 00000000001111111111222222222233333
 01234567890123456789012345678901234
"Find the +++ needle in + the ++ text"
```

Here one would expect to find "++" at 9 and 29.

```
 ss["Find the +++ needle in + the ++ text";"++"]
9 29
```

In order to determine how this function works let's strip out the details...

```
ss:{
    *{
      o:o@&(-1+(#y)+*x@1)<o:1_x@1; / set o
      $[0<#x@1;((x@0),*x@1;o);x]   / if x then y else z
      }
   [;y]/:(();&(x@(!#x)+\:!#y)~\:y)    / use value for inner function
   }
```

Given that k9 evaluates right to left, let's start with the rightmost code fragment.

```
 (();&(x@(!#x)+\:!#y)~\:y)          / a list (null;value)
```

And now let's focus on the value in the list.

```
 &(x@(!#x)+\:!#y)~\:y
```

In order to easily check our understanding, we can wrap this in a function and call the function with the parameters shown above. In order to step through, we can start with the inner parenthesis and build up the code until it is complete.

```
 {!#x}["Find the +++ needle in + the ++ text";"++"]
{!#x}["Find the +++ needle in + the ++ text";"++"]
^
:rank
```

This won't work as one cannot call a function with two arguments and then use only one. In order to get around this, we will insert code for the second argument but not use it.

```
 {y;#x}["Find the +++ needle in + the ++ text";"++"]
36
```

```
 {y;!#x}["Find the +++ needle in + the ++ text";"++"]
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 ..
```

As might have been guessed `#x` counts the number of charcters in the first argument and then `!#x` generates a list of integers from 0 to n-1.

```
 {(!#x)+\:!#y}["Find the +++ needle in + the ++ text";"++"]
 0  1
 1  2
 2  3
 3  4
 4  5
 5  6
 6  7
 7  8
 8  9
 9 10
10 11
11 12
12 13
13 14
14 15
15 16
16 17
17 18
18 19
19 20
20 21
..
```

Here the code takes each integer from the previous calculation and then adds an integer list as long as the second argument to each value. In order to verify this, you could write something similar and ensure the output what you predicted.

```
 {(!x)+\:!y}[6;4]
0 1 2 3
1 2 3 4
2 3 4 5
3 4 5 6
4 5 6 7
5 6 7 8
```

Now using the matrix above the code indices as the first argument and pull substrings that match the length of the search string.

```
 {x@(!#x)+\:!#y}["Find the +++ needle in + the ++ text";"++"]
Fi
in
nd
d 
 t
```

```
th
he
e
 +
++
++
+
 n
ne
ee
ed
dl
le
e
 i
in
..
```

At this point one can compare the search substring in this list of substrings to find a match.

```
 {(x@(!#x)+\:!#y)~\:y}["Find the +++ needle in + the ++ text";"++"]
000000000110000000000000000001000000b
```

And then one can use the where function, `&`, to determine the index of the matches.

```
 {&(x@(!#x)+\:!#y)~\:y}["Find the +++ needle in + the ++ text";"++"]
9 10 29
```

The rest of the 'ss' function code is left as an exercise for the reader.

20 Benchmark

Shakti is a fast data analysis language and clear benchmarks illustrate this. The Shakti website has a number of files for such purpose, b.k and taxi.k amongst others.

20.1 b.k

```
T:{09:30:00+_6.5*3600*(!x)%x}
P:{10+x?90};Z:{1000+x?9000};E:?[;"ABCD"]

/m:2;n:6
m:7000;n:5600000;
S:(-m)?`4;N:|1+_n*{x%+/x:exp 15*(!x)%x}m

t:S!{+`t`e`p`z!(T;E;P;Z)@'x}'N
q:S!{+`t`e`b!(T;E;P)@'x}'6*N

a:*A:100#S

\t  {select max p by e from x}'t A
\t  {select sum z by `o t from x}'t A
\t:10 {select last b from x}'q A
\t:10 select from t[a],`t^q a where p<b
\

C:M:?[;"ABCDEFGHIJ"]
trade(sym time exchange price size cond)
quote(sym time exchange bid bz ask az mode)

                Q1      Q2      Q3      Q4  ETL   RAM   DSK
k                1       9       9       1
postg        71000    1500    1900     INF  200   1.5   4.0
spark       340000    7400    8400     INF  160  50.0   2.4
mongo        89000    1700    5800     INF  900   9.0  10.0

960 billion quotes (S has 170 billion. QQQ has 6 billion.)
 48 billion trades (S has 12 billion. QQQ has 80 million.)
```

20.1.1 T

T is a function which generates a uniform list of times from 09:30 to 16:00.

```
 T:{09:30:00+_6.5*3600*(!x)%x}
 T[13]           / 13 times with equal timesteps over [start;end)
^09:30:00 10:00:00 10:30:00 11:00:00 11:30:00 .. 15:00:00 15:30:00
 ?1_-':T[10000]  / determine the unique timesteps
?00:00:02 00:00:03
```

20.1.2 P, Z, E

P is a function to generate values from 10 to 100 (price). Z is a function to generate values from 100 to 1000 (size). E is a function to generate values A, B, C, or D (exchange).

```
 P[10]
78 37 56 85 40 68 88 50 41 78
 Z[10]
4820 2926 1117 4700 9872 3274 6503 6123 9451 2234
 E[10]
"AADCBCCCBC"
```

20.1.3 m, n, S, N

m is the number of symbols. n is the number of trades. S is a list of symbol names. N is a list of numbers in decreasing order which sum approximately to n.

```
 4#S
`EEFD`IOHJ`MEJO`DHNK
 4#N
11988 11962 11936 11911
 +/N
5604390
```

20.1.4 t

t is an ntable of trades. The fields are time (t), exchange (e), price (p), and size (z). The number of trades is set by n.

Pulling one random table from t and showing 10 random rows.

```
 10?*t@1?S
t        e p  z
-------- - -- ----
14:37:53 D 73 4397
11:43:25 B 20 2070
10:21:18 A 53 6190
13:26:03 C 33 7446
14:07:06 B 13 2209
15:08:41 D 12 4779
14:27:37 A 11 6432
11:22:53 D 92 9965
11:12:37 A 14 5255
12:24:28 A 48 3634
```

20.1.5 q

q is a ntable of quotes. The fields are time (t), exchange (e), and bid (b). The number of quotes is set to 6*n.

```
 10?*q@1?S
t        e b
-------- - --
11:31:12 A 80
```

```
14:08:40 C 63
14:05:07 D 12
11:31:43 A 56
12:44:19 A 45
10:13:21 A 71
15:19:08 A 74
13:42:20 D 43
11:31:41 D 66
14:41:38 A 63
```

20.1.6 a, A

a is the first symbol of S. A consists of the first 100 symbols of S.

```
 a
`PKEM
```

20.1.7 Max price by exchange

The query takes 100 tables from the trade ntable and computes the max price by exchange.

```
 *{select max p by e from x}'t A
e|p
-|--
A|99
B|99
C|99
D|99
 \t  {select max p by e from x}'t A
22
```

20.1.8 Compute sum of trade size by hour.

This query takes 100 tables from the trade ntable and computes the sum of trade size done by hour.

```
 *{select sum z by `o t from x}'t A
t |z
--|--------
09| 4885972
10|10178053
11|10255045
12|10243846
13|10071057
14|10203428
15|10176102
 \t  {select sum z by `o t from x}'t A
27
```

20.1.9 Compute last bid by symbol

This query takes the 100 tables from the quote ntable and returns the last bid.

```
 3?{select last b from x}'q A
```

```
b
--
18
98
85

 \t:10 {select last b from x}'q A
2
```

20.1.10 Find trades below the bid

This query operates on one symbol from the q and t ntables, i.e. a single quote and trade table. The quote table is joined to the trade table giving the current bid on each trade.

```
 4?select from t[a],`t^q a where p<b
t        e p  z    b
-------- - -- ---- --
13:54:35 B 94 1345 96
11:59:52 C 26 1917 89
10:00:44 C 40 9046 81
10:59:39 A 25 5591 72
 \t:10 select from t[a],`t^q a where p<b
3
```

20.2 taxi

The taxi data analysis problem has become well known given the ease of acquiring the data and the size of it. It's well written up here (https://toddwschneider.com/posts/analyzing-1-1-billion-nyc-taxi-and-uber-trips-with-a-vengeance/) with a benchmark summary here (https://tech.marksblogg.com/benchmarks.html). Shakti has a benchmark script to generate simulated taxi data in order to check performance.

```
/taxi 1.1billion https://tech.marksblogg.com/benchmarks.html
/type/pcount/distance/amount

g:{[[]t:x rand`y`g;p:x rand 9;d:x rand 100;a:x rand 100.]}
x:d!g':44000+&#d:2009.01.01+!2500  /110 million example

ys:{`y[!x]sum/x} / year sum
\t sum{select[t]count from x}':x
\t ys@{select[p]count from x}':x
\t sum{select[p]sum a from x}':x
\t ys@{select[p,d]count from x}':x

\
x:g 10
select[t]count from x
select[p]count from x
select[p]sum a from x
select[p,d]count from x


Q1 select[t]count from x
```

```
Q2 select[p]avg a from x
Q3 select[d.y,p]count from x
Q4 select[d.y,p,d]count from x

          cpu  cost core/ram  elapsed         machines
k           4 .0004    4/16    1             1*i3.2xlarge(8v/32/$.62+$.93)
redshift  864 .0900  108/1464  8(1 2 2 3)   6*ds2.8xlarge(36v/244/$6.80)
db/spark 1260 .0900   42/336  30(2 4 4 20) 21*m5.xlarge(4v/16/$.20+$.30)
bigquery 1600 .3200  200/3200  8(2 2 1 3)

cost: k/redshift/databricks(1.5*EC2) bigquery(redshift) $5.00*TB  k($.05/TB)

csv
/vendor,pickup,dropoff,pcount,dist1,plong,plat,rate,flag,dlong,dlat,ptype,fare1,sur1,mta1,tip1,toll1,amount1
t:"b  12           e"  / type(2) passenger(8)
\t t:(`t`p`d`a;",";t)0:"taxi.csv"
```

20.2.1 g

Function to generate a table of random data to represent taxi company (**t** either yellow 'y or green 'g), paid fare (**p** 0-9), distance travelled (**d** 0-99) and address (**a** 0-100.).

```
 g:{[[]t:x?`y`g;p:x?9;d:x?100;a:x?100.]}
 g 10
t p d  a
- - -- --------
y 7 38 50.67771
g 5 34 38.51022
y 6 40 9.654263
y 6  5 98.91451
g 7 47 44.40432
g 0 64 66.07784
y 0 99 25.84292
y 5 46 46.87825
y 5 66 81.61647
g 0 10 7.946983
```

20.2.2 x

x is an ntable containing 2500 tables, keyed on day. Each individual table contains 44,000 rows therefore x has 110 million rows in total.

```
 x:d!g':44000+&#d:2009.01.01+!2500  /110 million example
 x
2009.01.01|[[]t:`y`y`g`y`g`g`y`y`g`g`g`g`g`y`y`y`y`g`g`g..
2009.01.02|[[]t:`y`g`g`g`y`g`y`g`y`g`g`g`g`y`g`y`g`y`g`y..
2009.01.03|[[]t:`y`g`g`g`y`g`y`g`y`g`g`g`g`y`g`y`g`y`g`y..
2009.01.04|[[]t:`y`g`g`g`y`g`y`g`y`g`g`g`g`y`g`y`g`y`g`y..
..
```

20.2.3 2009.01.04

Let's work with a single table to understand the queries and limit it to 5 rows.

```
 q:5#x 2009.01.04;q
```

```
t p d  a
- - -- --------
y 3 41 54.5307
g 6 70 70.4241
g 1 63 81.46645
g 7 36 43.12615
y 1 43 23.50114
```

Now let's count the number of rows by field t. Looking at the data above you should expect 2 yellow (y) taxi trips and 3 green (g) taxi trips.

```
 select[t]count from q
t|n
-|-
g|3
y|2
```

We can also sum the paid fare (p) by taxi company (t).

```
 select[t]sum p from q
t|p
-|--
g|14
y| 4
```

Now if we want to compute the same over all the data we need to run the kSQL query over each table. We'll do this on parallel threads ([eachprior], page 28) to speed it up.

```
 {select[t]count from x}':x
2009.01.01|[[t:`g`y]n:21968 22032]
2009.01.02|[[t:`g`y]n:21962 22038]
2009.01.03|[[t:`g`y]n:21962 22038]
2009.01.04|[[t:`g`y]n:21962 22038]
```

and then sum over all days.

```
 sum{select[t]count from x}':x
t|n
-|--------
g|55010927
y|54989073
```

Running the command with the timer will allow us to measure how long it takes to sum over the 110 million examples.

```
 \t sum{select[t]count from x}':x
1077
```

21 Conclusion

I expect you are pleasantly surprised by the speed of k9 and by the fact that it all fits in 134,152 bytes! (For comparison the ls program weighs in at 51,888 bytes and can't even change directory.)

If you're frustrated by the syntax or terse errors, then you're not alone. Many have had the same problems, but persevered, and finally came away a power user able to squeeze information from data faster than previously imagined.

Eventually, you'll realize that this manual isn't needed and it's all here...

```
select count first last min max sum avg var dev .. by ..
in n_(rand) n@(multiply) n?(divide) n@n?(bar)

Verb     monad        Adverb            Type
+ +                   ' each            char   " ab"
- -                   / over            sym    ``ab
* *                   \ scan            bool   011b
% div                                   int    2 3 4
! mod    where        System            float  2 3e4
& &      flip         \l load          -fixed  2.0 3.4
| |      reverse      \t time          -locus  -74::40.7
< <      asc          \v vars       z.d date   2001.02.03
> >      desc         \w work       z.t time   12:34:56.789
= =      freq                       z.T datetime
~ ~      ~
, ,      ,
# take   count        I/O               Class
_ drop   first        0' line           expr   :2+a
^ cut    sort         1' char/stdout    func   f[a] 2+a
@ @      type         2' data/stderr
? find   unique      *3' set            list   (2;3.4)
$ parse  str         *4' get            dict   {a:2 3}
. dict   value       *5' ffi            table  [a:2 3]
```