k tutorial

Alexander Belopolsky and Dennis Shasha

date: September 09, 2019 (with frequent updates)

Since 1992, Arthur Whitney's k and its derivatives have served a small number of highly skilled programmers, enabling them to create high performance (and high value) applications for finance and other data-intensive domains. While implementation efforts in other languages such as C++ and Java (and to a lesser extent Python) often involve thousands of lines of code, much of it built on top of libraries, a typical k application is on the order of scores of lines of code without the need for libraries. The expressive power is in the language itself.

This tutorial aims to familiarize users of conventional programming languages with the latest iteration of k, Shakti k. The language is capable of managing streaming, in-memory, historical, relational, and time-series data. The distributed model extends out to multiple machines whether on-premise or in the cloud.

Shakti k provides connectivity via Python, HTTP, SSL/TLS, and json. Shakti k supports compression and encryption for data, whether in-memory, in-flight or on disk. Shakti k also has primitives for blockchain operations.

The tutorial introduces language concepts, then presents examples. A good way to learn the language is to try to program the examples on your own.

Readers are invited to suggest corrections or new examples. You can contact the authors at tutorial@shakti.com.

Section 1: First encounter

Installing k

This can change, but as of March 2019, Shakti Software distributes a free evaluation version of k through Anaconda.org. The use of free version is subject to Evaluation Agreement.

Please follow the installation instructions at https://anaconda.org/shaktidb/shakti.

Shakti k does not have any dependencies and once you install it, you are ready to go. Simply type k at the command prompt and you will see the k banner and the prompt will change to a single space:

```
$ k
2019-04-18 15:45:55 8core 17gb avx2 © shakti m2.0 test
 ▮
```

The banner starts with the timestamp corresponding to the modification time of the k program. The timestamp is followed by a letter that will be M if you are using macOS and L if you are using Linux. The Ncore and Mgb parts show how many CPU cores (N) your k session will use and how much memory (M gigabytes) is earmarked for k use.

Power tip: for better interactive experience we recommend installing the `rlwrap` utility and define an alias

```
alias k="rlwrap k"
```

this will allow editing the expression that you enter at the k prompt and to recall any previous input from history.

Using k as a calculator

You can start using k as a powerful calculator: enter an expression at the prompt, press Enter and k will evaluate the expression and print the result. Many avaliable operations will look familiar, but you will soon discover some features that are unique to k.

Arithmetics

In k, `+`, `-`, and `*` work as the usual addition, subtraction and multiplication operations, e.g.,

```
 3*4
12
```

but the division operator is `%` while `/` has several uses including serving as a prefix for comments that k will ignore:

```
 10%3      / 10 divided by 3
3.333333
```

The next feature that may come as a surprise is that k does not use the traditional order of operations

```
 3*2+4     / addition is performed first
18
```

Instead of the (P)EMDAS order, k consistently evaluates its expressions from right to left with only the parentheses having higher precedence

```
 (3*2)+4   / multiplication is performed first
10
```

Elementary functions

As any good scientific calculator, k comes with a number of built-in functions. You can apply these functions by simpy typing their names before the argument separated by a space

```
 sqrt 2
1.414214
```

Trigonometric functions operate on arguments in radians and you will often need the π constant to convert from degrees. The π constant is built in in k and if you are using k on a Mac, you can type it using the alt-p key combination

```
 sin π%2
1f
```

Special values

Unlike some other languages that are quick to give up and report an error when given invalid input, k tries hard to provide useful answers. Thus if the result of a function is infinite, k will return a special ∞ value and indicate the sign of the infinity and such invalid result may dissapear in the subsequent computations

```
 log 0
-∞
 exp log 0
0f
```

When the result is completely undefined, k will return `ø`, which stands for missing data

```
 (log 0) + 1 % 0
ø
```

Unary operators

In the previous sections, we have seen operators that take two numbers as operands and functions that take one number as an argument. From elementary math, you are familiar with a unary – operator that takes a single operand and returns its negation. Not surprisingly, – does the same in k:

```
 - 42
-42
```

However, k takes this idea of the same operator having both binary and unary forms to the whole new level. Each operator in k has both unary and binary forms. For example, similar to –, the k division operator, %, has a unary form that computes the inverse

```
 % 3
0.3333333
```

Whetting your appetite

As you experiment with k as a calculator, you will soon notice that it does not complain about seemingly meaningless keystrokes. Thus a + sign on its own or following a number are simply echoed back by k

```
 +
+
 2+
2+
```

Entering the entire top row of shifted symbols is somehow understood by k as a valid expression

```
 ~!@#$%^&*()_+
~:!:@:#:$:%:^:&:*:0#,""_+
```

We will explain what is happening here in future sections.

Reference card

Within a k session, you can type `\h` to get a summary of the basic operations.

```
 \h
$k [-p 1234] [f.k]                          .z.i(pid) .z.x(arg) .z.e(env)

Verb                      Adverb               Noun            Atom List
:  assign                 '  each               char  " ab"       `c `C
+  add         flip       /  over               name  ``a`b       `n `N
-  subtract    negate     \  scan               int   Ø 0 2       `i `I
*  multiply    first      ': eachprior          float ø 2.3 π ∞   `f `F
%  divideby    inverse    /: eachright join|sv  time 12:34:56.789 `t `T   .z.t
&  min|and     where      \: eachleft split|vs  date 2019-06-28   `D `D   .z.D
|  max|or      reverse
<  less        ascend     System               list (2;3.4;`c)   `
>  more        descend    0: read/write line   dict {a:2;b:`c}   `a `A(table)
=  equal       group      1: read/write byte   func {(+/x)%#x}   `1..9
~  match       not        2: read/write data   expr  :32+9*f%5   `0
!  key         enumerate  3: conn/set (.z.ms)
,  catenate    list       4: http/get (.z.mg)  \h      help          \l log
^  except      null                            \l f.k  load
#  take        count      #[t;c;b[;a]] select   \t[:n]x time
_  drop        floor      _[t;c;b[;a]] update   \u[:n]x trace
?  find        distinct   ?[x;i;f[;y]] splice   \v [d]  vars
@  index       type       @[x;i;f[;y]] amend    \f [d]  fns
.  apply       eval       .[x;i;f[;y]] dmend    \cd [d] get[set]dir   \gr grep
$  pad|cast    string     $[c;t;f]     cond     \lf [x] file \lc char \ll line

util: in within bin like find [f]asc [f]desc [f]key
math: sqrt sin cos abs prm cmb [n]log [n]exp [n]rand mod div
aggr: count first last min max sum avg var dev med
sql: [select|update|delete] a.. by b.. from t.. where c..

datetime:YMDHRSTUV+duration:ymdhrstuv; T:.z.D+.z.t; 2019-06-28+2m; day:7 mod
`year`month`date`hour`minute`second`millisecond`microsecond`nanosecond

bool: `utf8`ascii`print
1way: `p`m`crc`sha`rip`bad   e.g. hash:  `sha@"crypto"
2way:``j`k`csv`b64`hex`aes   e.g. json: `j?`j@{a:2;b:`c}

if[c;..];while[c;..]

\\ exit
```

Section 2: Lists

Much of the expressiveness of k derives from the fact that most operations that operate on single values (atoms) generalize nicely to lists.

The simplest list is simply a sequence of numbers separated by spaces. When you apply one of the arithmetic operations between an atom and a list, k computes the result of the operation between each element of the list and the atom. For example,

```
 1 2 3 4 * 10
10 20 30 40
```

You can also perform operations on the lists of the same length and k will pair up the corresponding elements and apply the operation to each pair.

```
 1 2 3 + 3 2 1
4 4 4
```

However, if the lengths don't match, k will signal an error

```
 1 2 3 + 3 2
1 2 3 + 3 2
      ^
length error
>
```

We will explain the meaning of error displays later, but for now you just need to know that entering `\` at the `>` prompt will clear the error and allow you to continue.

Generating lists

Entering long lists into k can soon become tedious and k provides nice ways to generate lists either deterministically or randomly. You can generate a uniform list of any length by placing `&`, `!` or `rand` in front of a number:

- `&N` - N zeros
- `!N` - 0 through N-1
- `rand N` - N random numbers drawn uniformly from [0, 1]
- `rand -N` - N random numbers drawn from a normal distribution with μ=0 and σ=1.

Zeros

Instead of typing `0` twelve times for a list of twelve zeros, we can tell k to generate a list for us

```
 &12
0 0 0 0 0 0 0 0 0 0 0 0
```

Index

```
 x: !10
 x / this is the enumeration of 0 1 2 3 4 5 6 7 8 9     (but summarized):
!10
```

```
 5 + x
5+!10
```

Random lists

```
 10 rand 50 / generate randomly with replacement (so there can be duplicates)
24 45 13 28 28 8 43 9 17 30
```

(because of randomness, your result might not exactly match the above)

```
 10 rand 10
1 8 6 8 2 5 2 0 1 0
```

```
 -10 rand 10 / generate randomly without replacement (no duplicates)
2 9 7 1 8 5 0 6 4 3
```

There are more advanced ways to generate random arrays:

```
 20 rand 100 / recall uniform random with replacement
44 11 0 55 17 36 50 73 85 62 93 16 58 75 81 81 72 36 90 21

 rand 6 / uniform random with replacement between 0 and 1
0.782244 0.3937393 0.4717788 0.2755357 0.1641728 0.9861826
```

/ Others are on the way. (To be added)

Saving your work

Once we generated some data, we would want to save it and give it a name by which it can be recalled later. This is done by using an assignment expression that in k is a colon:

```
 a: &12
```

Cut and reshape

From now on, `a` will refer to a list of 12 zeros until we reuse this name by assigning it to something else.

From simple lists, k can create lists of lists by cutting the lists into chunks using the `_` operator (remember that `a` is 12 zeros). In this example, we are taking the first two zeros in the first row then zeros 2 through 5 in the second row and then the rest in the third row.

```
 0 2 6 _ a
0 0
0 0 0 0
0 0 0 0 0 0
```

If we want to cut a list into chunks of equal size, we can use the `#` (reshape) operator:

```
 3 4 # a
0 0 0 0
0 0 0 0
0 0 0 0
```

The reshape operator generalizes even further allowing cutting lists into lists of lists of lists and so on (`a` is still 12 zeros):

```
 3 2 2 # a
(0 0;0 0)
(0 0;0 0)
(0 0;0 0)
```

Note that if k had a 3d display, it could show this as a stack of 2x2 matrices, but because we only have two dimensions, k shows stacked matrices in a linear notation. The same notation can be used for input

```
 (1 2;3 4)
1 2
3 4
```

If the reshape operator is given a list on the right that contains fewer items than is necessary to fill the shape, items from the front of the list will be reused

```
 10#!4
0 1 2 3 0 1 2 3 0 1
```

If k did not have the built-in eye function (`=`), we could also build a unit matrix by filling an n x n shape with a length n+1 unit vector:

```
 5 5 # 1,&5
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
```

While we have not introduced the `,` operation, you have probably guessed that in the above example `1,` prepends `1` to the following list. We will discuss `,` in greater detail next.

Enlist and join

K's simple syntax for entering lists has one drawback. It is not obvious how to enter lists with zero or one element. In fact, k offers no literal syntax for that and such lists have to be generated. You already know one method: simply reshape an atom to a list using `0#` or `1#`. When you create a list of one element, you will see that it is displayed as follows:

```
 1#42
,42
```

That leading `,` in front of the number is the enlist function that turns the number `42` into a 1-element list. When applied to a list, `,` turns it into a 1-element list containing a list. Note the difference between

```
 2#1 2 3    / take first 2 elements
1 2
```

and

```
 2#,1 2 3   / repeat twice
1 2 3
1 2 3
```

In the first case, `2#` reshape gets a 3-element list and cuts it to length 2, but in the second case it gets a 1-element list, so it recycles the first element (which itself is a list `1 2 3`) and makes a 2 x 3 matrix.

When `,` is placed between two lists or between a list and an atom it joins the elements together

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

Joining lists of lists or matrices joins the rows:

```
 x,x: !-3
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
```

(recall that `!-3` above is a unit 3 x 3 matrix)

To join columns, we can use the `,'` operator:

```
 x,'x: !-3
1 0 0 1 0 0
0 1 0 0 1 0
0 0 1 0 0 1
```

To flatten a list of lists, use the `,/` function

```
 ,/(1;2 3;4 5 6)
1 2 3 4 5 6
```

To recursively flatten a nested list, use the `,//` function. Compare

```
 ,/(1;!-2)
1
1 0
0 1
```

and

```
 ,//(1;!-2)
1 1 0 0 1
```

Lists from atoms and back

We can split integers into digits

```
 10\:123456789
1 2 3 4 5 6 7 8 9
```

and put them back together

```
 10/:1 2 3 4 5 6 7 8 9
123456789
```

using the pair of vector from scalar (`\:`) and scalar from vector (`/:`) operators.

For binary expansion - just use 2 instead of 10

```
 2\:42
1 0 1 0 1 0
```

and we can use the same operator to split time in seconds into days, hours and minutes:

```
 24 60 60\:12345
3 25 45
 24 60 60/:3 25 45
12345
```

Generating 2d data

Several primitives exist in k to generate lists of lists in a single operation:

- `=N` - create an NxN unit matrix
- `!v` - odometer
- `prm N` - all permutations of numbers from 0 through N-1
- `M cmb N` - all combinations of M numbers drawn from 0 through N-1

Eye

```
 !-3
1 0 0
0 1 0
0 0 1
```

Odometer

Read this column by column from left to right. The lowest row is counting in base 3. The second row is base 2 and the top row is base 1. The second row changes its values when the lowest row reaches 10 base 3.

```
 !1 2 3
0 0 0 0 0 0
```

```
0 0 0 1 1 1
0 1 2 0 1 2
```

Combinatorics

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


 2 cmb 3
0 1
0 2
1 2
```

Section 3: List operations

Atomic operations

The four arithmetic operator +, −, ∗ and % can operate on both scalars and arrays of arbitrary shape. We call these operator "atomic" because for any scalar function that operates on atoms, this operator works on each element of an array.

For example, let's create a triangular shape array that we've seen before and give it a name b:

```
 b:0 2 6 _ &12
 b
0 0
0 0 0 0
0 0 0 0 0 0
```

We can add a scalar to b and it will be added to each element

```
 b + 2
2 2
2 2 2 2
2 2 2 2 2 2
```

or we can add a vector and its elements will be added to the rows of b:

```
 b + 1 2 3
1 1
2 2 2 2
3 3 3 3 3 3
```

It may take some practice to understand how these rules generalize to deeply nested lists. For example,

```
 (3 2 # !6) + 3 2 2 # &12
(0 0;1 1)
(2 2;3 3)
(4 4;5 5)
```

In addition to the four arithmetic operations, k applies the same rules to

- & - and/min
- | - or/max
- <, >, and = - comparison

Reduction operations

There are ways to reduce lists to single values. Thus `first` and `count` applied to a list return the first element and the count of elements respectively

(because of the use of randomness, your results may not be the same as those you see here)

```
 r: 10 rand 300 / generate randomly with replacement
 r
265 243 125 8 155 17 9 4 36 207

 count r / count of elements
10
 first r / first element
265
 last r  / the last element
207
 max r   / the largest element (maximum)
265
 min r   / the smallest element (minimum)
4
 avg r   / average (arithmetic mean)
106.9
```

Full list operations

In k, we rarely need to process lists one element at a time because we have powerful operations that can transorm the entire list in one go. Thus you can sort a list in either ascending or descending order

```
 asc r
4 8 9 17 36 125 155 207 243 265
 dsc r
265 243 207 155 125 36 17 9 8 4
```

or you can reverse the list using `|` and compare two lists using `~`:

```
 (dsc r) ~ |asc r
1
```

note that when we compare two lists using `~`, we get a single `1` when they match and a single `0` when they don't.

Selection operations

There are ways to index arrays.

```
 z: 22 + !10
 z
22+!10

 z[0]
22

 z[3]
25

 z[3 5]
25 27

 z[2+!6]
24 25 26 27 28 29

 z[(#z)-1]
31

 z[_ (#z) % 2]
27
```

Now consider multi-dimensional arrays.

```
 mymulti: (1 2 3; 4 5 6; 7 8 9; 10 11 12)
 mymulti
1 2 3
4 5 6
7 8 9
10 11 12
```

/ we interpret mymulti as a four row three column matrix

```
 mymulti[0;0]
1

 mymulti[2;0]
7

 mymulti[1;2]
6
```

/ Now we can get full rows

```
 mymulti[1]
4 5 6
```

/ and full columns

```
 mymulti[;1]
2 5 8 11
```

Section 4: Verbs, Adverbs, and User-defined Functions

The operations in k are called 'verbs' and often have two meanings depending on whether they are 'unary' (applied to a single argument) or 'binary' (applied to a pair of arguments). Normally, the binary verb will be the more familiar one. Much of the power comes from applying verbs to arrays.

Verbs

unary + (flip or transpose)

```
 x: (1 2 3 4; 5 6 7 8) / assign to x  a two row array whose first row is 1 2 3 4
 +x                    / a four row array (transpose of x) whose first row is 1 5
1 5
2 6
3 7
4 8
```

binary + (plus)

```
 2 + 3 / scalar (single element) addition
5

 x + x / array addition
2 4 6 8
10 12 14 16

 2 + x / element to array addition
3 4 5 6
7 8 9 10
```

Just as most human languages have verb modifiers called adverbs, k does too. They apply to most unary and binary operators. Thus, the `/` adverb (called 'over'), instead of indicating a comment, can cause the binary version of the verb to apply to the elements in the array in sequence and yields a single result. The `\` adverb (called 'scan') does the same but keeps all the intermediate results.

over and scan

```
 +/ 1 2 3 4  / Apply the + operator between every pair of elements; produce sum
10

  +\ 1 2 3 4 / Same as above but produce all partial sums
1 3 6 10
```

Adverbs can modify verbs directly but can also modify verb-adverb combinations (which are lifted to verb status). The ' (each) adverb takes both roles.

```
 x
1 2 3 4
5 6 7 8

 +/'x / Apply +/ to each row of x
10 26

 +\'x / Apply +\ to each row of y
1 3 6 10
5 11 18 26
```

Adverbs can modify user-defined functions as well.

```
 f:{[a] (a*a)+3 }
 f[4]
19
```

Now we can apply f to each element of an array using the each adverb.

```
 f'1 2 3 4
4 7 12 19
```

While verbs combined with \ and / have the syntactic form of unary verbs, verbs combined with \: and /: have the syntactic form of binary verbs. Examples:

There is \: (each left):

```
 1 2 3 4 +\: 10
11 12 13 14
```

There is each right:

```
 20 +/: 1 2 3 4
21 22 23 24
```

There is each left each right (which should be interpreted as performing an each left on successive elements of the right array):

```
 1 2 3 4 +\:/: 10 20 30 40 50
11 12 13 14
21 22 23 24
31 32 33 34
41 42 43 44
51 52 53 54
```

Eachright eachleft considers each element of the left array one at a time and applies +/: to the that element and the right array

```
 1 2 3 4 +/:\: 10 20 30 40 50
11 21 31 41 51
12 22 32 42 52
13 23 33 43 53
14 24 34 44 54
```

This also applies to binary (and even user-defined) verbs.

lambdas

```
 g:{[a;b] a + (7 * b)}
 g[2;3]
23
```

Eachleft considers each element of the left array one at a time and applies g to that element and to the entire right array.

```
 1 2 3 4 g\: 10 20
71 141
72 142
73 143
74 144
```

Eachright considers each element of the right array one at a time and applies g to the left array and that element.

```
 1 2 3 4 g/: 10 20
71 72 73 74
141 142 143 144
```

Eachleft eachright considers each element of the right array one at a time and applies g\: to the left array and that element.

```
 1 2 3 4 g\:/: 10 20
71 72 73 74
141 142 143 144
```

(Try for example 1 2 3 4 g\: 20)

Eachright eachleft considers each element of the left array one at a time and applies g/: to that element and the right array

```
 1 2 3 4 g/:\: 10 20
71 141
72 142
73 143
74 144
```

(Try for example 3 g/: 10 20)

Finally, each can apply to just one argument

```
 x1: 1 2 3 4
 x2: 50

 g[;x2]'x1
351 352 353 354

 g[x1]'x2
351 352 353 354
```

Extended Example: matrix multiplication

Recall that matrix multiplication involves the dot products between rows of the left matrix and the columns of the right matrix.

```
 leftmat: (1 2 3; 4 5 6; 7 8 9; 10 11 12)
 leftmat
1 2 3
4 5 6
7 8 9
10 11 12

 rightmat: (100 200 300 400 500; 1000 2000 3000 4000 5000; 10000 20000 30000 40000 50000)
 rightmat
100 200 300 400 500
1000 2000 3000 4000 5000
10000 20000 30000 40000 50000
```

```
 dot:{[v1;v2] +/ v1 * v2} / dot product function

 dot[4 5 6; 300 3000 30000]
196200

 matmult:{[m1;m2] m1 dot/:\: +m2}
 matmult[leftmat; rightmat]
32100 64200 96300 128400 160500
65400 130800 196200 261600 327000
98700 197400 296100 394800 493500
132000 264000 396000 528000 660000
```

Adverbs Replace Loops

K programmers tend not to need loops. In fact, some of them disdain loops. The reason is simply that the language uses adverbs instead of loops.

For example, the loop

```
result = 0
for i = 1 to len(myarray)
  result += f(myarray[i])
```

becomes

```
result: +/f'array
```

In principle each invocation of f could (and will eventually) be done in parallel.

By contrast, the loop

```
result = 0
for i = 1 to len(myarray)
  if result = f(result, myarray[i])
```

becomes / or \ at least for some f's

```
 myarray: 2 2 2 2
 f:{[x;y] x + 2*y}

 f\ myarray
2 6 10 14

 f/ myarray
14
```

Second example where k's initializations can be useful:

```
 g: {[x;y] x*y}
 g\myarray
2 4 8 16
```

Section 5: Beyond numbers

"The easiest machine applications are the technical/scientific computations." Edsger W.Dijkstra

Names, characters and strings

Character strings are simply an array of characters

```
 x: "fast, cool, and really concise"
 #x
30
 x[2 4]
"s,"
```

```
 x[<x]
"    ,,aaacccdeefilllnnooorssty"
```

Whereas character strings occupy one byte per character, symbols are hashed and therefore take less space, a useful feature in a large data application in which a symbol is repeated many times.

```
 x: (`abc; `defg)
 x
`abc`defg
 #x
2
```

Here is some guidance in choosing between symbols and characters. If there are a few distinct character sequences and they are repeated many times (e.g. a history of all trades where there are only a few thousand stock symbols but millions of trades), then symbols are best for operations like sorting and matching. Otherwise char vectors are probably better especially if you need to do substring matching.

Dates, times and durations

Date format is year-month-day and you can get the day by .z.D

```
 x: .z.D
 x
2019-04-19

 x + 44
2019-06-02

 .z.D+/:10 rand 24:00:00 / generate random datetimes
2019-04-19T18:26:33 2019-04-19T22:55:55 2019-04-19T03:09:39 2019-04-19T02:03:52 2019-04-19T02:48:02 2019-04-19T18:44
```

Time format is hour:minutes:minutes.milliseconds

```
 x: .z.t / greenwich mean time
 x
03:51:28.435

 x+ 609 / add to milliseconds
03:51:29.044


 09:30+10 rand 06:30 / generate random times
13:46 10:04 12:55 14:38 10:42 09:53 14:48 11:05 11:49 14:11
```

Dictionaries and tables

/ Dictionaries are key to value structures. There are many ways to create a dictionary.

/ From partitions

```
 mypart: ="many sentences have the letter e very often"
 mypart
m|,0
a|1 16
n|2 7 10 42
y|3 36
 |4 14 19 23 30 32 37
s|5 13
e|6 9 12 18 22 25 28 31 34 41
t|8 20 26 27 40
c|,11
h|15 21
v|17 33
l|,24
r|29 35
o|,38
f|,39
```

```
 mypart["e"]
6 9 12 18 22 25 28 31 34 41
```

/ Creating them directly

```
 mydict: `bob`carol!(2;3)
 mydict
bob  |2
carol|3

 mydict[`bob]
2
```

/ dictionaries can be heteogeneous in their values

```
 mydict2: `alice`bill`tom`judy`carol!(1 2 3 4; 8 7; 5; "we the people"; `abc)

 mydict2[`carol]
`abc
 mydict2[`judy]
"we the people"
```

/ dictionaries can be heterogeneous in their keys too

```
 mydict3: (`marie;`jeremie;2)!(34; "hello, world"; 7)
 mydict3
`marie  |34
`jeremie|"hello, world"
2       |7
```

/ find the keys of dictionaries is easy:

```
 ! mydict3
`marie
`jeremie
2
```

/ and the values

```
 . mydict3
34
"hello, world"
7
```

Section 6: Control Flow

Before we get to control flow in the classical sense, it's important to remember how to read a k expression. As mentioned earlier, there is no precedence as there is in some languages where for example * binds more than + or -. Instead the precedence is right to left which by the way conforms with mathematical usage (e.g., for sum of x * y we first multiply x and y and then take the sum)

Order of operations

```
 20*4-3
20

 (20*4) - 3 / to make * bind closer than -, you need parentheses
77

 20*(4-3) / otherwise, precedence is right to left.
20
```

let's say we want the sum of the elements having values greater than 35

```
 x: 90 30 60 40 20 19
 x > 35 / put a 1 where values are greater than 35
1 0 1 1 0 0

 & x > 35 / indexes that are greater than 35
0 2 3

 x[&x > 35] / elements in x that are greater than 35
90 60 40

 +/ x[&x > 35]  / sum of elements of x whose values are greater than 35
190
```

`$[c;t;f]` (Conditional)

if c is true then execute the t branch else the f branch)

```
 x: 3 4 5
 y: 10 20 30

 $[5 > 3; +/x; +/y]
12
 $[5 < 3; +/x; +/y]
60
```

?[x;I;[f;]y] (replace the index positions by what comes afterwards)

```
 x: 3 4 5 6 7 8 9 10
 y: 100 200 300 400

 ?[x;3;y] / replace what's in position 3 by y
3 4 5 100 200 300 400 6 7 8 9 10

 ?[x;3 4; y] / starting in position 3 and counting 4, replace by y
3 4 5 100 200 300 400 10
```

Section 7: Input/Output and Interprocess Communication

Create a file having these three lines and call it tmp:

```
We the people
of the
United States
```

Read in the file

```
 x: 0: "tmp"
 x
We the people
of the
United States
 x[1] / x is just an array so x[1] is the second element, viz.
"of the"

 y: (x[0]; x[1]; x[2]; x[1])
 y
We the people
of the
United States
of the

 "tmp2" 0: y
```

Now look at tmp2 and see that you have:

```
We the people
of the
United States
of the
```

`1:` (write binary image)

```
 x: 1 2 3 4
 "tmp3" 2: x

 y: 2: "tmp3"
 y
1 2 3 4
```

Text input/output is `0:`

```
 "foo"2:("This is line 1\n This is line 2")

 2:"foo"
0x54686973206973206c696e6520310a2054686973206973206c696e652032
```

Interactive prompt is `1:`

```
 name:1:""1:"What is your name? "
What is your name? Carol
 name
"Carol"
```

Section 8: Data Structures

k has atom, list (2;`c), dict [a:2;b:`c] and func {[x;y]x+y} dict [a:2;b:`c] view f::32+1.8*c TODO

Section 9: Debugging

on error(inspect variables and assign to them)

```
 f:{[x;y] x + y}
 f[5;6]
11
 f[5;`abc]
{[x;y] x + y}
         ^
type error
> x
5
> y
`abc
> y:7
> x+y
12
```

`'` up a level in the call stack (e.g. if in function f, go to caller of f) `\` out of debugging mode 2+ \3 trace

Section 10: Scripts and Meta-functions

`\l a.k` load

create a file foo.k with the two lines

```
 x: 1 2 3 4
 f: {[x] x*x}
```

Then start a k session and then load foo.k in that session using the \l command:

```
 \l foo.k
 x
1 2 3 4
 f
{[x] x*x}
```

\v variables \f functions

```
 \l foo.k

 \v
,`x

 \f
``f
```

\w workspace

how much memory are you using

```
 \w
1072128
```

Section 11: Tables

In the row-wise vs. column-wise table debate, k comes out as columnwise. We'll work up to this slowly.

Consider a list:

```
 x: 1 2 3 4 5 10 15
 x
1 2 3 4 5 10 15
```

Create a one column table from this:

```
 xtab: +`numcol!x
 xtab
numcol
------
 1
 2
 3
 4
 5
10
15
```

Here the table has one column and its header is numcol.

```
  select numcol from xtab
numcol
------
 1
 2
 3
 4
 5
10
15
```

```
  select sum numcol from xtab
numcol
------
40

   select sum numcol from xtab where numcol > 4
numcol
------
30
```

Ok, now let's create a multiple column table. Your results may differ because of the use of randomness.

```
 n: 7
 newtab: +(`stock`date`price`vol)!(n rand `ibm`goog`hp;.z.D+/:n rand 16:00:00;100 + n rand 200; n rand 5000)
 newtab
stock date                price vol
----- ------------------- ----- ----
goog  2019-04-19T09:10:36 204   2361
ibm   2019-04-19T10:56:40 269   2923
hp    2019-04-19T04:23:09 157   1391
goog  2019-04-19T07:59:40 147   3381
hp    2019-04-19T02:12:28 118   4419
ibm   2019-04-19T01:20:18 127   4839
ibm   2019-04-19T05:07:01 168   2846

  select sum price*vol by stock from newtab
stock|vol
-----|-------
goog |978651
ibm  |1878968
hp   |739829

  select sum price*vol by stock from newtab where date > 2018-08-25T10:00:00
stock|vol
-----|-------
goog |978651
ibm  |1878968
hp   |739829
```

User-defined functions:

```
 f:{[x] 1.5*x}
 select sum f[price*vol] by stock from newtab where date > 2018-08-25T10:00:00
stock|vol
-----|-------
goog |1467976
ibm  |2818452
hp   |1109744
```

Extracting Data from Tables into Other Structures

/ select always gives a table

```
 select date from newtab
date
-------------------
2019-04-19T09:10:36
2019-04-19T10:56:40
2019-04-19T04:23:09
2019-04-19T07:59:40
2019-04-19T02:12:28
2019-04-19T01:20:18
2019-04-19T05:07:01

 newtab: +`stock`date`price`vol!(n  rand  `ibm`goog`hp;.z.D+/:n rand 16:00:00;100 + n rand 200; n rand 5000)

 newtab
stock date                price vol
----- ------------------- ----- ----
```

```
ibm   2019-04-19T08:23:21 103    174
goog  2019-04-19T10:29:58 167   2830
ibm   2019-04-19T04:46:40 179   1918
hp    2019-04-19T10:54:38 137   1200
goog  2019-04-19T14:00:49 237   4122
ibm   2019-04-19T01:46:34 245   2510
hp    2019-04-19T01:09:32 275   2951
```

Importing from a csv file

Create a small csv file mytrade.csv whose schema is: tradeid,stock,timeindicator,price,vol

```
mytrade.csv:
1,goog,50,1237,100
2,msft,51,109,100
3,goog,52,1240,200
4,msft,53,112,200


 ("isiii";",")0:"mytrade.csv"
1 2 3 4
Ø Ø Ø Ø
50 51 52 53
1237 109 1240 112
100 100 200 200

 mytrade1: +(`tradeid`stock`timeindicator`price`vol)!("isiii";",")0:"mytrade.csv"

 select tradeid, price, vol from mytrade1
tradeid price vol
------- ----- ---
1       1237  100
2        109  100
3       1240  200
4        112  200


 select tradeid, price, vol from mytrade1  where price > 500
tradeid price vol
------- ----- ---
1       1237  100
3       1240  200
```

Then one with proper datetimestamps called mytradebac2.csv

```
1,goog,15:16:50,1237,100
2,msft,15:16:51,109,100
3,goog,15:18:50,1240,200
4,msft,15:18:52,112,200


 mytrade2: +(`tradeid`stock`time`price`vol)!("intfi";",")0:"mytradebac2.csv"

 select tradeid, time, price, vol from mytrade2  where price > 500
tradeid time         price vol
------- ------------ ----- ---
1       15:16:50.000 1237  100
3       15:18:50.000 1240  200
```

/ Note that it is also possible to import from just a set of lists, / because that is what 0:"somefile.csv" gives.

```
 0:"mytradebac2.csv"
1,goog,15:16:50,1237,100
2,msft,15:16:51,109,100
3,goog,15:18:50,1240,200
4,msft,15:18:52,112,200


 select sum price * vol by `minute$time from mytrade2
time |vol
-----|------
15:16|134600
```

```
15:18|270400

 select `minute$time from mytrade2
time
-----
15:16
15:16
15:18
15:18
```

Modifying Tables

```
 select vol from mytrade2
vol
---
100
100
200
200

 update vol:1+vol from mytrade2
tradeid stock time         price vol
------- ----- ------------ ----- ---
1       goog  15:16:50.000 1237  101
2       msft  15:16:51.000 109   101
3       goog  15:18:50.000 1240  201
4       msft  15:18:52.000 112   201
```

/ but the table itself is not changed because there no assignment:

```
  mytrade2
tradeid stock time         price vol
------- ----- ------------ ----- ---
1       goog  15:16:50.000 1237  100
2       msft  15:16:51.000 109   100
3       goog  15:18:50.000 1240  200
4       msft  15:18:52.000 112   200
```

/ On the other hand

```
 mytrade2updated: update vol:1+vol from mytrade2
 mytrade2updated
tradeid stock time         price vol
------- ----- ------------ ----- ---
1       goog  15:16:50.000 1237  101
2       msft  15:16:51.000 109   101
3       goog  15:18:50.000 1240  201
4       msft  15:18:52.000 112   201

 delete from mytrade2 where vol > 100
tradeid stock time         price vol
------- ----- ------------ ----- ---
1       goog  15:16:50.000 1237  100
2       msft  15:16:51.000 109   100
```

/ Would need to assign to mytrade2 to see this effect. / e.g. mytrade2: delete from mytrade2 where vol > 100

/ Here is a row to insert.

```
x: {tradeid:15;stock:`goog;time:15:26:50.123;price:2337f;vol:200}
```

/ An insert:

```
mytrade2: mytrade2,x
```

/ There is a notion of a keyed table where each key value is supposed / to occur only once. tradeid is an example.

```
 u:`tradeid key mytrade2
 u
tradeid|stock time         price vol
-------|----- ------------ ----- ---
 1     |goog  15:16:50.000 1237  100
 2     |msft  15:16:51.000 109   100
 3     |goog  15:18:50.000 1240  200
 4     |msft  15:18:52.000 112   200
15     |goog  15:26:50.123 2337  200
```

/ Note that the , operator will insert if the new row has a new key / (tradeid of 45)

```
 y: {tradeid:45;stock:`goog;time:16:26:50.123;price:3337f;vol:75}
 u,y
tradeid|stock time         price vol
-------|----- ------------ ----- ---
 1     |goog  15:16:50.000 1237  100
 2     |msft  15:16:51.000 109   100
 3     |goog  15:18:50.000 1240  200
 4     |msft  15:18:52.000 112   200
15     |goog  15:26:50.123 2337  200
45     |goog  16:26:50.123 3337   75
```

/ but update if the new row has an existing key (tradeid of 4). / So the , operator is called an upsert.

```
 y: {tradeid:4;stock:`goog;time:16:26:50.123;price:3337f;vol:75}

 u,y
tradeid|stock time         price vol
-------|----- ------------ ----- ---
 1     |goog  15:16:50.000 1237  100
 2     |msft  15:16:51.000 109   100
 3     |goog  15:18:50.000 1240  200
 4     |goog  16:26:50.123 3337   75
15     |goog  15:26:50.123 2337  200
```

/ This is an upsert because this is an operation that specifies / the key and all fields.

```
 u
tradeid|stock time         price vol
-------|----- ------------ ----- ---
 1     |goog  15:16:50.000 1237  100
 2     |msft  15:16:51.000 109   100
 3     |goog  15:18:50.000 1240  200
 4     |msft  15:18:52.000 112   200
15     |goog  15:26:50.123 2337  200
```

Section 12: Shortcuts

1) We would be remiss to fail to mention some shortcuts that k afficionados love to use, even though some of us feel that they reduce clarity. For example, unary functions implicitly perform "each" when applied to arrays. For example,

```
 f:{[a] (a*a)+3 }

 f'1 2 3 4
4 7 12 19

 f 1 2 3 4
4 7 12 19

 x:(1 2 3 4;5 6 7 8)
 f'x
4 7 12 19
28 39 52 67
 f x
4 7 12 19
28 39 52 67
```

default function parameters are x y z, e.g. {z+x*y}[3;2;1] is 7

```
 f:{z + x*y}
 f[10; 20; 30]
230
```

Eval

It is possible to make strings be evaluated as k expressions using the dot operator:

```
 . "2+3"
5
```

Section 13: System Calls (in progress)

e.g. instead of \ls \du \wc -l try

Unix ls is just \ff ?.c

Unix du is \fk ?.c

Unix wc -l is \fl ?.c

Section 14: A gallery of exercises/examples

These examples are going to start from a database of trades.

```
n: 100
secid: n rand (`goog;`facebook;`ibm;`msft)
price: 100 + n rand 200
vol: 10 + n rand 1000
time: !n
```

Find all trades such that the price is over 175.

```
ii: & price > 175
z: secid[ii] ,' price[ii] ,' vol[ii] ,' time[ii]
z[5]
`ibm
235
289
8
```

Find the high and low of each security.

```
mydict: =secid
names: rand secid
maxmin: {[name] name , (|/price[mydict[name]]), (&/price[mydict[name]])}
maxmin'names
`ibm
289
107
```

/ or to be able to take an arbitrary dictionary

```
maxmin2:{[name;somedict] name , (|/price[somedict[name]]), (&/price[somedict[name]])}
maxmin2[;mydict]'names
`ibm
289
107
```

Get the moving average of each security

```
 mavg:{[name] name, (+\price[mydict[name]])%'(1+!#mydict[name])}
 mavg'names
`ibm
171f
203f
175.3333
163.25
174f
193.1667
190f
196f
195.6667
200.2
196f
200f
197.7692
202.1429
201.5333
197.6875
195.4706
197.9444
196.4211
192.7
190.2381
191.5455
188.4348
189.5833
186.88
185.8462
183.4074
180.6786
183.4828
182f
185.0645
187f
185.2121
```

Determining the finishing time of each task if you do them in earliest deadline first order.

```
 n: 10
 taskid: !n
 tasktime: 2 + n rand 20
 deadlines: 40 + n rand 50

 tasktime
14 2 5 13 8 20 4 16 19 13

 deadlines
68 51 54 64 64 66 53 85 55 63
```

/ Put them all together

```
 taskid,'tasktime,'deadlines
0 14 68
1 2 51
2 5 54
3 13 64
4 8 64
5 20 66
6 4 53
7 16 85
8 19 55
9 13 63
```

/ Now determine the order of deadline indexes / for the deadlines to be in order.

```
 inddead: < deadlines
 deadlines[inddead]
51 53 54 55 63 64 64 66 68 85
```

/ Put the tasks in the same order

```
 taskid[inddead]
1 6 2 8 9 3 4 5 0 7
```

/ Put the task times in the same order

```
 tasktime[inddead]
2 4 5 19 13 13 8 20 14 16
```

/ Put all of them in the same order

```
 taskid[inddead],'tasktime[inddead],'deadlines[inddead]
1 2 51
6 4 53
2 5 54
8 19 55
9 13 63
3 13 64
4 8 64
5 20 66
0 14 68
7 16 85
```

/ Find end point if tasks are executed in this order

```
 +\tasktime[inddead]
2 6 11 30 43 56 64 84 98 114
```

/ Determine which deadlines are met (1) and which aren't (0)

```
 deadlines[inddead] > +\tasktime[inddead]
1 1 1 1 1 1 0 0 0 0
```

/ Determine which taskids have their deadlines met

```
 taskid[inddead][& deadlines[inddead] > +\tasktime[inddead]]
1 6 2 8 9 3
```

Section 15: Debugging

```
 fib:{[n] fib[n-1] + fib[n-2]}

 fib[5]
{[n] fib[n-1] + fib[n-2]}
               ^
stack error
```

To debug this, we can do several thing. First, just query the variables, e.g.

```
> n
-189
```

We might realize that n should never be negative. Another thing we can do is store all the values of n in this recursive function.

```
 out: ()
 fib:{[n] out,: n; fib[n-1] + fib[n-2]}
 fib[5]
{[n] out,: n; fib[n-1] + fib[n-2]}
                       ^
stack error
>
```

But now we can query out:

```
> out
5 3 1 -1 -3 -5 -7 -9 -11 -13 -15 -17 -19 -21 -23 -25 -27 -29 -31 -33 -35 -37 -39 -41 -43 -45 -47 -49 -51 -53 -55 -57
```

Section 16: Order-based Relational Algebra

This version is build just on arrables (tables consisting of lists of ordered lists (arrays)). First we review selects, projects, and moving aggregates. Then we show equi-joins then general joins. file: arrable.k

/ functions

/ given a set of indexes give me those values of a vector x

```
i:1 5 7
x:0 10 20 30 40 50 60 70 80
x @ i
10 50 70
```

/ given a bunch of lists alllist / a selection string on alllist selstr / (a typical selstr might be "(alllist[1] > 3) & (alllist[0] < 40)") / and a set of output columns outlist / Output: after selecting based on selstr, output columns cols of alllist

```
mysel:{[alllist;selstr;outlist] ii: & . selstr;alllist[outlist]@\:ii}
```

/ given a bunch of lists alllist / a subset to sort by sortby / other lists to follow that sort outlist / Output: based on the sort order of alllist[sortby] / create rows of outlist in order

```
myasc:{[alllist;sortby;outlist] myind: $[1 < #sortby; < +alllist[sortby]; < alllist[*sortby]]; alllist[outlist]@\:m
```

/ This does moving sum on numpoints (e.g. three point moving sum) / of the array myarray / If there are fewer than numpoint in myarray, it does the moving sum up / to the number of points in myarray.

```
movsum:{[numpoints;myarray] x: +\myarray; xsub: $[numpoints < #myarray;(numpoints # 0),x[!(#x)-numpoints];0]; x - x
```

/ This does moving average on numpoints (e.g. three point moving average) / of the array myarray / If there are fewer than numpoint in myarray, it does the moving average up / to the number of points in myarray.

```
movavg:{[numpoints;myarray] x: movsum[numpoints; myarray]; mydivs: $[numpoints < #myarray; (1+!numpoints), numpoint
```

```
/ relational equijoin on one attribute
/ given two lists of lists LL1 and LL2
/ index from LL1 indLL1
/ index from LL2 indLL2
/ indexes from LL1 outLL1
/ indexes from LL2 outLL2
/ Find all indexes of LL1 and indexes of LL2 that match
/ based on the values in indLL1 and indLL2 and then take the cross product
/ for the columns outLL1 of LL1 and outLL2 of LL2
```

```
eqjoin:{[LL1;LL2;indLL1;indLL2;outLL1;outLL2] mymatch: &:' LL1[indLL1] =\: LL2[indLL2]; outindLL1: ,/ ((#:')mymatch
```

/ Below is unused (and needs to be debugged) until we can get longer functions

```
/  mycross:{[pair; mydict1; mydict2] x: pair[0]; y: pair[1]; ,/mydict1[x] ,/:\: mydict2[y]}

/  fin:{[allmatches; Ll1; LL2; outLL1; outLL2] (indtoval[;allmatches[;0]]'LL1[outLL1]), (indtoval[;allmatches[;1]]'L

/  eqjoindict:{[LL1;LL2;indLL1;indLL2;outLL1;outLL2] d1: = LL1[indLL1]; keys1: ! d1; d2: = LL2[indLL2]; keys2: ! d2;
```

/ data

```
indata: (3 4 3 4 3 9 9 9 9 9;30 40 30 40 30 90 90 90 90 90;7 9 1 2 1 2 3 4 8 7 3)
```

/ execution

```
 mysel[indata; "indata[1] > 60 "; 0 1 2]
9 9 9 9 9 9
90 90 90 90 90 90
2 3 4 8 7 3
 x: myasc[indata; 2 1; ,0]
 x
,3 3 4 9 9 9 9 3 9 9 4

 movavg[3;x]
,3 3 4 9 9 9 9 3 9 9 4f
```

/ can combine these

```
 x1: mysel[indata; "indata[1] > 35 "; 0 1 2]
 x2: myasc[x1; 2 1; ,0]

 movavg[3;x2]
,4 9 9 9 9 9 9 4f
```

/ Now look at the equijoin

```
 LL1: indata

 LL2: (300 400 300 400 300 800 800 800 301 401 402;30 40 30 40 30 80 80 80 30 40 40;7 9 1 2 1 2 3 4 8 7 3)

 eqjoin[LL1; LL2; 1; 1; 0 1; 0 1]
3 3 3 3 4 4 4 4 3 3 3 3 4 4 4 4 3 3 3 3
30 30 30 30 40 40 40 40 30 30 30 30 40 40 40 40 30 30 30 30
300 300 300 301 400 400 401 402 300 300 300 301 400 400 401 402 300 300 300 301
30 30 30 30 40 40 40 40 30 30 30 30 40 40 40 40 30 30 30 30
```

/ Sometimes we want to perform a join, but get results only for one argument / e.g. LL1 in this case:

```
 y1: eqjoin[LL1; LL2; 1; 1; 0 1 2; !0]

 y2: mysel[y1; "y1[1] > 35 "; 0 1 2]

 y3: myasc[y2; 1; ,2]

 movavg[3;y3]
,9 9 9 9 2 2 2 2f
```

String matching (dynamic programming example)

/ data

```
 s1: "rent"

 s2: "let"

 initval: 1 + ((#s1) * (#s2))

 mat: (1+(#s1);1+(#s2)) # initval

 mat[0]: 0, 1+!#s2

 mat: mat@[;0;:;]'0, 1+!#s1
```

Section 17: k reference

Until now, we have touched on only a few of the verbs and types. Here is Arthur Whitney's full list. From what you understand already, these won't be hard to learn.

```
+  plus       flip
-  minus      negate
*  times      first
%  divide     inverse
```

```
&  min|and     where
|  max|or      reverse
<  less        up
>  more        down
=  equal       group
~  match       not
!  dict|mod    key|enum
,  concat      enlist
^  except      null
$  pad|cast    string
#  take|select count
_  drop|delete floor
?  find|rand   uniq|rand
@  index       type
.  apply       value
```

First, let's look at how to read this table. In each row, the binary meaning precedes the unary meaning.

Let's go through each in turn.

/ this is a comment \\ if alone on a line exits the k session or a debugging environment

: gets

```
x: 1 2 3 4 / gets indicates assignment
```

+ plus flip

```
 2 + 3
5
```

/ Unary + (transpose)

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

– minus negate

```
 2 - 3
-1
```

/ Unary - (negation)

```
 - 3 4 5
-3 -4 -5
```

∗ times first

```
 4*5
20
```

/ Unary * (first in list)

```
 * 15 24 19 10
15
```

% divide inverse

```
 5 % 3
1.666667
```

/ Unary % (inverse)

```
 % 81
0.01234568
```

! mod|div enum

/ The following is 14 mod 3 (it turns out to be convenient to put the modulus first)

```
 3 ! 14
3|14
```

/ Unary ! enumerate either by integer (in this case numbers 0, 1, 2, ... 19) or float

```
 !20
!20
```

& min|and (and, if 1 is interpreted as true and 0 as false) where

```
 5 & 3
3

 1 & 1
1

 1 & 0
0
```

/ Unary & (indexes where a there is a non-zero)

```
 x: 4 8 9 2 9 8 4

 x = 9 / 1 will indicate a match
0 0 1 0 1 0 0


 & x = 9 / locations in the list above that are 1
2 4

 x > 4
0 1 1 0 1 1 0

 & x > 4
1 2 4 5
```

| max|or reverse

```
 5 | 3
5
 1 | 0
1
```

/ Unary | reverses lists

```
 | 2 3 4 5 6
6 5 4 3 2
```

< less asc

```
 5 < 3 / returns 0 because false
0
 3 < 5 / returns 1 because true
1
```

/ Unary < says which order of indices gives data in ascending order

```
 x: 6 2 4 1 10 4
 xind: < x / index locations from smallest value to highest
 xind      / Notice that x[3] is 1, the lowest value in the list
3 1 2 5 0 4

 x[xind] / sort the values in ascending order
1 2 4 4 6 10
```

> more dsc

```
 5 > 3
1
 3 > 5
0
 x: 6 2 4 1 10 4
```

/ Unary > says which order of indices gives data in descending order

```
 xind: > x / index locations from highest value to smallest
 xind
4 0 2 5 1 3
 x[xind] / descending sorted order
10 6 4 4 2 1
```

= equal group

```
 5 = 5
1

 1 2 13 10 = 1 2 13 10 / element by element
1 1 1 1

 1 2 14 10 = 1 2 13 10 / 0 at position 2 indicates inequality
1 1 0 1
```

/ Unary = gives a dictionary (more on dictionaries below) mapping values to indexes where / values are present

```
 = 30 20 50 60 30 50 20 20
30|0 4
20|1 6 7
50|2 5
60|,3
```

/ In the above example 20 is at positions 1 6 and 7

```
 x: = 30 20 50 60 30 50 20 20
```

~ match not

```
 1 2 13 10 ~ 1 2 13 10 / should match
1

 1 2 10 13 ~ 1 2 13 10 / does not match (order matters)
0
```

/ Unary ~ (like the Boolean not operator except that any non-zero becomes zero)

```
 ~ 1
0
 ~ 0
1
 ~ 5
0
 ~ -5
0
```

, concat enlist

```
 x: 10 11 12 13
 y: 4 3

 x,y / concatenate one way
10 11 12 13 4 3

 y,x / concatenate the other
4 3 10 11 12 13

 z: y,x
 z[4] / can index these as if these were an array
12

 z[2+!3] / can fetch many indexes
10 11 12
```

/ Unary , converts a scalar (atom) into a list or a list into a deeper list

```
 x: 5
 x / x is an atom
5
 *x / first on an atom is just the atom itself
5

 x: ,5 / x is now a list

 x / Note the comma in front indicating that x is  list
,5

 y: x,15
 y / the comma goes away since two or more elements already form a list
5 15

 *y / The * operator takes the first element of a list
5

 *x  / The * operator takes the first element of a list even a singleton
5
```

^ except null

```
 x: 8 7 6 5 2

 x ^ y / elements of x (preseving order in x) that are not in y
8 7 6 2

 y ^ x
,15
```

/ Unary ^ tests whether the argument is null / You might use this in some missing data applications.

```
 x: `kiscool
 ^x
0

 x: `
```

```
 ^x / output of 1 indicates that x is now null
1
```

take|shape count

```
 x: 10 20 30 40 50 60
 3 # x
10 20 30

 10 # x / Notice x is of length 6; result wraps
10 20 30 40 50 60 10 20 30 40

 3 2 # x / creates a three row, two column matrix
10 20
30 40
50 60


 3 10 # x / creates a three row, 10 column matrix (with wrapping)
10 20 30 40 50 60 10 20 30 40
50 60 10 20 30 40 50 60 10 20
30 40 50 60 10 20 30 40 50 60
```

/ Unary # counts the lengths of lists

```
 x
10 20 30 40 50 60

 # x / number of elements in x
6

 y: 9 8 5
 z: (x;y)
 z
10 20 30 40 50 60
9 8 5
 #:'z / counts each list
6 3
```

_ drop|cut floor

```
 x: 10 20 30 40 50 60
 3 _ x / cut away 3 elements from the beginning
40 50 60

 10 _ x / Notice x is of length 6; so this eliminates more than necessary
!0
```

/ !0 means an empty list

/ Now unary _ is the floor operator

```
 15 % 4
3.75
 _ 15 % 4
3
```

$ cast|+/* string

```
 ` $ "abc" / cast string to symbol (name)
`abc

 . "18" / cast string to int
18
 . "18.2" / cast string to float
18.2
```

/ Unary form

```
 $ `abc / cast symbol to string
"abc"
```

`rand` rand|find unique

/ Random as discussed in section 1

```
 10 rand 12 / with replacement (can be duplicates) from 0 to 11
11 1 2 4 7 4 8 10 9 0


 15 rand 12 / there can be more elements (15) than the domain (0 to 11)
6 1 8 7 9 7 8 11 5 10 6 7 5 0 8

 -10 rand 12 / random and uniform without replacemnt (no duplicsates)
1 3 9 7 10 5 11 0 6 2

 -15 rand 12 / get an error
     ^
^
```

/ With list as left argument we can find the index of the first match

```
 40 20 30 10 20 30 ? 30
2
```

/ Unary `rand` for atoms (scalars)

```
 rand 7 / random between 0 and 1 (uniform)
0.3811082 0.3939469 0.5833342 0.8344042 0.8774682 0.8705262 0.7793733
```

/ Unary ? for arrays removes duplicates but preserves order

```
 ? 40 20 30 10 20 30
40 20 30 10
```

`@` at type

```
 x: 40 20 30 10 20 30
 @[x;2 4 5]
30 20 30

 x / unchanged
40 20 30 10 20 30

 @[x; 2 4 5;: ;  -17 -12 -8]
40 20 -17 10 -12 -8

 x / still unchanged
40 20 30 10 20 30


 f: {[x] x * x}
 @[x; 2 4 5; f] / squares locations 2 4 and 5
40 20 900 10 400 900

 x / still unmodified
40 20 30 10 20 30

 @[x; 2 4 5; f] / squares locations 2 4 and 5
40 20 900 10 400 900
```

/ Unary @ finds the type of an object

```
 @ 18
`i

 @ "18"
`C

 @ `abc
`n
```

`.` dot value

/ Unary . Can evaluate a string

```
 . "18 + 5"
23
 . "f: {[x] x * x * x}"

 . "f[5]"
125
```

21) abs (absolute value)

```
 abs -3.2
3.2
 abs -3.2 4 5.3
3.2 4 5.3
```

`log` (natural log, also known as ln or log base e)

```
 log 8
2.079442
```

`exp` (exponential on e)

```
 exp 1
2.718282

 exp 3
20.08554

 log 20.08554
3f
```

`sin` (takes its argument in radians)

```
 mypi: 3.14 / a crude approximation of pi
 sin[mypi] / should be approximately 0
0.001592653

 sin[mypi % 2] / sin pi/2 is 1, so this is close
0.9999997
```

`cos` also takes its argument in radians

```
 cos[mypi] / close to -1
-0.9999987

 cos[mypi %2]
0.0007963267
```

`in` (membership test)

```
 5 in 10 20 30 5 6 9 10
1

 5 in 10 20 30 5 6 5 10 / even if there are two instances, still return 1
1

 5 in 10 20 30 50 6 15 10
0
```

`bin` (binary search assumes ascending sorted order))

```
 x: 5 * (3 + !10)
 x
15 20 25 30 35 40 45 50 55 60

 x bin 33 / index location that is less than or equal to 33
3

 x bin 35 / index location that is less than or equal to 35 (here, equal)
4
```

`within`

(upper bound for singleton right hand side lists and closed lower and open upper bound for binary right hand side lists)

```
 x: 40 10 20 23 15 16 18
 x within ,20  / less than 20
0 1 0 0 1 1 1

 x
40 10 20 23 15 16 18
 x within 16 23  / between 16 (inclusive) and 23 (exclusive)
0 0 1 0 0 1 1
```

/ No unary version

`find` / substring looking for an exact match

```
 x: "abcdef"
 x find "cde" / look for beginning and length of match
,2 3

 x find "bd" / If there is no match, then is this the return value I want???
0#,0 0

 y: x, x
 y
"abcdefabcdef"

 y find "cde" / get all matches as a list
2 3
8 3
```

`like` (string match with wildcards)

```
 x: "abcdef"
 x like "ab"
0

 x like "ab*" / allows wildcards
1

 x like "*def"
1
```

```
 x like "*bcd*" / arbitrary length wildcards with *
1

 x like "?bcd*" / ? is a single character substitution
1
```

Section 18: Navigation

- k tutorial

fixed 353 lines