K Slices, K Dices

John Earnest

August 15, 2017

K offers several ways to access an element from a list or vector by index:

```
  t: 11 22 33 44 55 66

  t[4]        / subscripting
55
  t 4         / juxtaposition
55
  t@4         / the 'at' verb
55
  t . 4       / the 'dot' verb
55
```

At first glance, these all seem equivalent. How are they different? Indexing in K generalizes to vectors:

```
  t[1 2 3 2 1]
22 33 44 33 22
  t 1 2 3 2 1
22 33 44 33 22
  t@1 2 3 2 1
22 33 44 33 22
```

This is important for the common idiom for sorting:

```
  u: 33 27 19 14 99 50
  <u
3 2 1 0 5 4
  u[<u]
14 19 27 33 50 99
```

However, *dot* doesn't seem to work like the other approaches when indexing by a vector. Let's focus on bracket-indexing, for the sake of consistency, and come back to the alternatives later:

```
  t . 1 2 3 2 1
rank error
t . 1 2 3 2 1
  ^
```

Consider indexing into a matrix. Remember: K matrices are a list of lists, so the "outer" dimension is the row and the "inner" dimension is the column:

```
  m: ("ABCD"
      "EFGH"
      "IJKL"
      "MNOP")

  m[1]          / row
"EFGH"

  m[1;3]        / row; column
"H"

  m[1 2]        / rows
("EFGH"
 "IJKL")

  m[1 2;0 2]    / rows; columns
("EG"
 "IK")

  m[0 2 0 2;1 3 1 3]
("BDBD"
 "JLJL"
 "BDBD"
 "JLJL")
```

When indexing with brackets, an empty dimension is a "wildcard":

```
  m[;1 2]   / all rows, columns 1 and 2
("BC"
 "FG"
 "JK"
 "NO")

  m[1 2;]   / equivalent to m[1 2]
("EFGH"
 "IJKL")
```

Side note: if you really want to access data by column rather than by row, consider taking the transpose of it:

```
  q: ("ABCD";9 8 7 4;`beef`pork`chicken`tofu)

  q[;1]
("B";8;`pork)

  +q
(("A";9;`beef)
 ("B";8;`pork)
 ("C";7;`chicken)
 ("D";4;`tofu))

  (+q)1
("B";8;`pork)
```

In general, juxtaposition of a value and an index is equivalent to the verb *at* (apply), and square brackets are equivalent to the verb *dot* (apply-at-depth). *at* is equivalent to *dot* if we enclose its argument, restricting it to operate on the outermost depth:

```
  m 1 2
("EFGH"
 "IJKL")

  m@1 2
("EFGH"
 "IJKL")

  m . ,1 2
("EFGH"
 "IJKL")

  m[1 2;1 3]
("FH"
 "JL")

  m . (1 2;1 3)
("FH"
 "JL")
```

Indexing dictionaries works the same as with lists or vectors:

```
  d: .((`foo;.((`quux;42);(`plam;99)))
       (`baz;.((`plam;98);(`bar;102)))
       (`bar;1 2 3))

  d @ `foo`baz
(.((`quux;42;)
   (`plam;99;))
 .((`plam;98;)
   (`bar;102;)))

  d[`foo;`quux]
42

  d[`foo`baz;`plam]
99 98
```

Dictionaries can additionally be indexed via dot-notation:

```
  d.foo
.((`quux;42;)
  (`plam;99;))

  d.foo.quux
42

  (`d.foo)`quux  / symbol juxtaposition
42

  `d.foo@ `quux  / symbol application
42
```

You may have noticed that brackets are also used for calling functions. What gives?

```
  f: {x+100*y}

  f[4;9]
904
```

In K, indexing and calling functions (application) are identical *ideas*, so they have identical syntax:

```
  {2+x}[4]       / subscripting
6
  {2+x} 4        / juxtaposition
6
  {2+x}@4        / the 'at' verb
6
  {2+x} . 4      / the 'dot' verb
6
```

What happens if we use a "wildcard" index on a function or don't specify every argument?

```
  f[4]
{x+100*y}[4]
  f[;9]
{x+100*y}[;9]
```

We get back a "projected" function which remembers the arguments that have been supplied already but still needs values for any empty slots. In functional languages the process of producing new function from an existing one by supplying a fixed value for some arguments is called "currying":

```
  (f[4])[9]
904
  (f[;4])[9]
409
```

Note that this works on built-in verbs, too:

```
 ,[;"foo"]
,[;"foo"]
  ,[;"foo"]"bar"
"barfoo"
```

Consider the parallel between a partially applied function and a "partially sliced" matrix or higher-dimensional structure:

```
  b: {x{x+2*y}/:\:x}[!4]

  b
(0 2 4 6
 1 3 5 7
 2 4 6 8
 3 5 7 9)

   b[1]
1 3 5 7
  b[;2]
4 5 6 7
  b[1;2]
5

  {x+2*y}[1]'!4
1 3 5 7
  {x+2*y}[;2]'!4
4 5 6 7
  {x+2*y}[1;2]
5
```

In real-world code you might find many of these ideas combined in a single statement:

```
  md: {.((`name;`$x);(`label;$x);(`type;y);(`table;`$z))}

  md[;2;".pub.demo.retail"]'names@idx
(.((`name;`"Slab Fistmeat";)
   (`label;"Slab Fistmeat";)
   (`type;2;)
   (`table;`.pub.demo.retail;))
 .((`name;`"Brick Hardcheese";)
   (`label;"Brick Hardcheese";)
   (`type;2;)
   (`table;`.pub.demo.retail;)))

  (md[;2;".pub.demo.retail"]'names@idx)[;`name`type]
((`"Slab Fistmeat";2)
 (`"Brick Hardcheese";2))
```

Important caveat! In K3, function projection is *syntactic* and requires square brackets:

```
  {x+100*y} . 1 2    / works: all arguments provided
201

  {x+100*y} . (;2)   / doesn't work: dot-apply with wildcard
type error
{x+100*y}
  ^
>  \

  {x+100*y}[;2]      / works: same as above but with brackets
{x+100*y}[;2]

  {x+100*y} 2        / doesn't work: not enough arguments
valence error
{x+100*y} 2
^
>  \

  {x+100*y}[2]       / works: same as above but with brackets
{x+100*y}[2]
```