03_Lists

September 22, 2019

1 3. Lists

1.1 Basic operations on lists

- Create:
  - General list: (item1;item2;...;item_n)
  - Simple list: item1 item2 item3
  - Create singleton list from atom: enlist atom
  - Create empty list: ()
- Access:
  - Iterated indexing: list[i][j][k]
  - Indexing at depth: list[i;j;k]
  - Accessing multiple elements: list[i1 i2 i3 ... in]
  - reverse lookup (find item index in a list): list?10 -> returns the index of the first occurrence of 10
- Assignment:
  - list[i]:42
  - list[i;j;k]:42 / only works with indexing at depth with multi-dimensional list
- Add:
  - add to front: new_item,list
  - add to end: list,new_item
- Insert:
  - at index ?????
- Removel item from list
  - by item: list except num / removes first element from list that matches num
  - by index: list _ i
  - drom from beginning or end: num _ list / from beginning if number is positive, from end if number is negative
- Length of list:
  - one dimensional list: count list
  - multi-dimensional list: count each list_of_lists
- Transpose multi-dimensional list:
  - flip list_of_lists
- Reversing the order of a list:
  - reverse list
- Rotating a list
  - num rotate list

- Get first x elements of a list: # (take)
  - num # list / if num > count list, # is circular
- Operations on string lists:
  - join: "delimiter" sv string_list
  - split: "delimiter" vs string

1.2 3.1. Overview

- Lists are recursively defined as an ordered collection of atoms and other lists
  - q lists are inherently ordered based on their declaration (as opposed to SQL which is based on sets).
    - this has implications on the semantics of q-sql queries compared to SQL queries
    - the ordered nature of lists make large sets of time series data processing fast and natural
- Lists can be thought of as dynamically allocated arrays (similar to ArrayList in Java?)
- All data structures in q are ultimately built from lists:
  - a dictionary is a pair of lists;
  - a table is a special dictionary;
  - a keyed table is a pair of tables.
- Notation for general lists: (1;2;3;4)
- Notation for simple lists:
  - same as general lists
  - 1 2 3 4
  - singleton list: (enlist 1)
- **SIMPLE LISTS** are made up of atoms of homogeneous type (VECTORS in mathematics)
- Simple lists have optimum storage and performance characteristics
  - simple lists occupy contiguous storage space in memory
  - items are appended at the end of the list
  - direct item access via indexing
- General lists:
  - are pointers in contiguous storage
  - can hold items of different type without resorting to a union (sum) type

1.3 3.1 Introduction to Lists

[ ]:
```
(1; 1.1; `1)
(1;2;3)
("a";"b";"c";"d")
(`Life;`the;`Universe;`and;`Everything)
(-10.0; 3.1415e; 1b; `abc; "z")
((1; 2; 3); (4; 5))
((1; 2; 3); (`1; "2"; 3); 4.4)
```

- **count**: same as lentgth(), size(), len(), etc
  - returns a long data type number
- **first**, **last**: returns the first and last elements in a list, respectively

1.4 3.2 Simple Lists

- a simple list is a list of atoms of a uniform type
- corresponds to **vectors** in math
- general lists are converted implicitly to simple lists (IN WHAT CASES?) -> might cause problems...

[ ]:
```
(100;200;300)
100 200 300
100 200 300~(100; 200 ; 300) / identity notation ~
```

- simple and floating point lists

1.4.1 3.2.6 Lists of Temporal Data

- To force the type of a mixed list of temporal values, append a type specifier.

[ ]:
```
01:02:03 12:34 11:59:59.999u / u refers to minUtes
```

1.5 3.3. Empty and singleton lists

- empty list notation: ()

[ ]:
```
L:()
L
-3!L / -3! is an internal function: format to text
```

- singleton list: a list consisting of a single item of q entity (not only atoms)
- notation: enlist q entity
- () is for grouping elements

[ ]:
```
(42)~enlist 42 / false, they are not identical
(42)~42
```

1.6 3.4. Indexing

- index notation: []
- item indexing
- index assignment lista[0]:42 set the first element to 42
  - type of value assigned has to match the list type
  - there is no automatic type promotion in q!!!!!!
- **type specific null values** (indicating missing data) instead of error messages
- Index assignment into a simple list enforces strict type matching with no type promotion: assigned element has to have the same type as the list. The narrower type is not promoted to the wider type.

  - (however, joining or appending to a simple list do not use strict type matching. if the item and the list do not have the same type, the list is propagated to a general list)
- to prevent implicit conversion of a general list to a simple list, place the nil value at the beginning of a list: (::;1;2;3)
- lists can hold expressions (count list; sum list;1;2)

[ ]:
```
l2:48 58 73 55 437 2
l2[2]:1
-3!l2[()]
-3!l2[::]
(count l2;sum l2;33;53)
```

1.7 3.5. Combining lists

- Joining lists
  - to join two lists, use comma: l1,l2
  - to make a single atom a list: (),x or x,()
- Merging lists
  - notation ^
  - right item prevails over the left item except when it is null

1.8 3.6. Lists as maps

- Lists can be considered as maps where their domain is a list of conscutive integers starting with zero, and their codomain is their elements.
  - A list with atomic items acts like a monadic map
  - A list with lists as items acts like a multivalent map

1.9 3.7. Nesting

- Data complexity is built by using lists as items of lists
- The number of levels of nesting is called **depth**: the depth measures how much repeated indexing is necessary to arrive at atoms

1.10 3.8. Iterated indexing and indexing at depth

- iterated item indexing: list[2][1][3] the item at the 3rd index of the list which is at the 1st index if the list at the 2nd index of the top-most list
  - considers this q entity as an array of arrays
- indexing at depth: list[2;1;3]. this results in the same as above. the same as a multi-valent function
  - considers this q entity as a multi-dimensional matrix
- assignment works with indexing at depth but does not work with iterated indexing

- **ADVANCED**: Create a matrix from a list with the **reshape operator**: number_of_rows number_of_columns #list

[ ]:
```
matr:3 4#til 9
matr
0N 3#til 10
```

[ ]:
```
m:((11; 12; 13; 14); (21; 22; 23; 24); (31; 32; 33; 34))
m
m[0][0]
m[0; 0]
m[0; 1]
m[1; 2]
```

1.11 3.9. Indexing with lists

- Retreiving multiple items with a list of indices: L[]

[ ]:
```
l2
l2[1_til 3]
```

- Indexing via a variable
- Indexing nested lists with a list which contains the elements to access
- Assignment with list indexing

[ ]:
```
L:100 200 300 400
```

[ ]:
```
L[1 2 3]:2000 3000 4000
L
```

[ ]:
```
L[til 3]:1 2 333
L
```

[ ]:
```
K[til 3]:til 3
K
```

- Juxtaposition

[ ]:
```
L ::
```

- Find
  - find maps an item to its index ->
  - find is the inverse of indexing
  - find operator: x ? y returns the firsts appearance of y from x
  - if y is not in x, the length of the list is returned
  - y is atomic

[ ]:
```
L
L?333 2
```

[ ]:
```
L ~ 1
```

1.12 3.10. Elided indexes

[ ]:
```
m:(1 2 3 4; 100 200 300 400; 1000 2000 3000 4000)
```

[ ]:
```
m[;1]
```

1.13 3.11. Rectangular lists and matrices

1.13.1 Rectangular lists

- all sublists have the same length
- flip: transpose matrix

1.13.2 Matrices

- Matrices are a special case of rectangular lists and are defined recursively.
- A matrix of dimension **0** is a **scalar**.
- A matrix of dimension **1** is a **simple list** (vector).
- For **n>1**, **a matrix of dimension n is a list of matrices of dimension n - 1 all having the same size**.
- Thus, a matrix of dimension 2 is a list of vectors, all having the same size.
- If all atoms in a matrix have the same type, we call this the **type of the matrix**.
- The console display of m in tabular form motivates defining the list m[;j] as the jth column of m.
- The notations m[i][j] and m[i;j] both retrieve the same item – namely, the item in row i and column j.
- Matrices are nested lists stored in row order.
- When the rows are simple, each occupies contiguous storage. This makes **row retrieval very fast**.
- On the other hand, columns must be picked out of the rows, so **column operations are slower**.

[ ]:
```
mm:((1 2;3 4;5 6);(10 20;30 40;50 60))
mm
mm[0]
mm[1]
```

1.13.3 Matrix flexibility

[ ]:
```
m
m 0 2
```

1.14 1.12 Usefil list operations

- til
- distinct
- where
- group

[ ]:
```
til 10
```

[ ]:
```
each til count m
```

[ ]:
```
distinct m / only distinct elements
```

[ ]:
```
bool:01011001b
where bool / returns true items. use it in predicates
L[where L>2]
```

- group takes a list and returns a dictionary in which each disctinct item of the argument is mapped to its indices of occurences in order of occurrence

[ ]:
```
group "i miss mississippi"
```