CHAPTER 1

Fundamentals of the q Programming Language

This chapter starts our journey to programming with q. It aims to equip the reader with the basic concepts and ideas needed to code in q. In the subsequent chapters, we will delve deeper into the detail of the language and its features. We will always consider their applications to practical tasks as the primary motivation for our book.

1.1 THE (NOT SO VERY) FIRST STEPS IN q

We start with the assumption that the reader already has a working installation of kdb+ on his or her computer. Hence the *not so very* in this section's title. If the reader is fortunate enough, kdb+ has already been installed by the system administrator. Otherwise, the best strategy is to follow the instructions under "Download" on http://kx.com/.

kdb+ is available for industry-standard 32- and 64-bit architectures (including AMD Opteron, Intel Xeon, and Sun) running Linux, Windows or Solaris. As of the time of writing, Kx Systems offer a limited version of kdb+/q for non-commercial use. "Commercial use" is defined as any use for the user's or any third party's financial gain or other economic benefits, whether this is production use or beta testing. However, developing a proof-of-concept application, even in a commercial setting, is not regarded as commercial use. The non-commercial version can also be used for academic research. It is also fully suitable for exploration of the language as presented in this book. This book does not dwell on the nuances of running kdb+ on different architectures and operating systems.

The maximum amount of memory available to a 32-bit application is around 4 GB. This limits the applicability of the 32-bit version of kdb+. The 64-bit version can be used to implement large (multi-TB) in-memory databases that are common in high-frequency trading and big data applications.

To start kdb+, open your favourite shell, change the current directory to the one that contains the kdb+ executable (usually named q or q.exe), and launch that executable by typing in q. This executable is the q programming language interpreter – the face of

the kdb+ system as seen by the user. By default, it is installed in one of the following directories:

```
// main linux/mac directory with ~ being home
    ~/q
// linux executables
    ~/q/l32
// mac executables
    ~/q/m32
// main windows directory
    c:\q
// windows executables
    c:\q\w32
```

For the 64-bit version, "32" in the path is replaced with "64".

We recommend that the reader will install kdb+ in the default directory. If this is not possible, one can customise the installation using the environment variables (QHOME, QINIT, etc.). In the sequel, we assume that kdb+ is installed in the default directory.

Once kdb+ is running, we will see the prompt

```
q)
```

This prompt indicates that the q interpreter is awaiting our first commands. From now on, we shall assume that we are always in an open q session and therefore shall omit the prompt q) from our listings.

Let us experiment with the q language by typing in the expression

```
1+1
```

We shall see the output

```
2
```

We have just evaluated our first q expression!

Any introduction to a programming language requires the presence of a "hello world" example. We create a script file, named helloWorld.q in the directory from which we have launched our q session. The file contains the following line:

```
show "hello world!"
```

From the q session, we launch the script with system command \l:

```
\l helloWorld.q
```

which produces the output:

```
"hello world!"
```

The system command \l loads the file helloWorld.q, which applies the show function to its argument, the string "hello world!". The show function outputs its argument to the console.

To close the q session, we enter at the prompt the system command

```
\\
```

You may have noticed that the system commands, which control the q environment, all start with a backslash "\". The quit system command, \\, is no exception. Alternatively, we may evaluate the `exit` function

```
exit 0
```

where the integer argument passed to `exit` (in this case, 0) will be set as the exit status of the q interpreter process. The convention is to use `0` for the normal exit status and integers larger than zero to signal errors. Alternatively, pressing `Ctrl+d` will do the same job as either \\ or `exit`.

Having had our first glimpse of the q programming language in action, let us introduce its basic elements.

1.2 ATOMS AND LISTS

The most basic element of the q programming language is the atom, the smallest irreducible element of the language. An atom may be an integer, boolean or floating point variable, a function, or a date. We distinguish atoms by their types. We can find out the type of the number 2 using the command `type`:

```
type 2
```

```
-7h
```

the negative sign is used to distinguish atoms from lists. A negative type number, such as –7h in our case, indicates atoms, whereas a positive type number (or zero) is reserved for vectors, since q is by design a vector language. –7h tells us that 2 is an atom of long type, as indicated by the number 7. The extra letter, h, actually says that number −7, which indicates the *type of 2*, is itself a short integer.

So the `type` function always returns a short integer. Yes, q decides the default format of number 2 for its user, in this case, a long integer. We may, however, change it and define number 2 as a short integer, which will only require a quarter of memory space compared to a long integer, by appending the suffix "h" to the literal 2:

```
type 2h
```

```
-5h
```

which corresponds to short type, the result of `type` equal to −5 stands for the short integer.

A float can be defined as:

```
type 2f
```

```
-9h
```

or

```
type 2.0
```

```
-9h
```

where the number 9 stands for the float type.

If we define a list of atoms, the `type` of such a list does not express the type of all individual elements, but the type of a list as a whole. The list thus has a special place in `q` on its own, which is not surprising since `q` is a vector language. Let us find out the type of a list consisting of 2 and 3:

```
type (2;3)
```

```
7h
```

The result of `type` changed the sign from negative to positive. We conclude that the provided object is a list – positive sign – and all the entries are of the same type – long integer. The list itself is treated as one object.

What if we want to construct a list which contains only one element? The function `enlist` does the job for us:

```
type 2
```

```
-7h
```

while the single element list reads

```
enlist 2
```

```
,2
```

which suggests the output is a list. We can confirm this using

```
type enlist 2
```

```
7h
```

We may achieve the same result with

```
2,()
```

```
,2
```

where `()` plays the role of an empty list.

`q` enables one to define a list containing elements of mixed types. The type number of such a list is 0h:

```
type (2; 2h; 3; 2.0)
```

```
0h
```

By default, this is also the type of an empty list:

```
type ()
```

```
0h
```

An empty list can be explicitly cast to a different type using a type cast:

```
type `short$()
```

```
5h
```

In our example

```
2,()
```

type coercion still ensures that the result is a list of type float.

Mixed lists may also contain other lists as their elements:

```
type (2; 1; (1; 2; 3))
```

```
0h
```

If we want to see the type of each elements of a list, we can run

```
type each (2;3)
```

giving the expected output

```
-7 -7h
```

```
type each (2; 2h; 3; 2.0)
```

```
-7 -5 -7 -9h
```

We will devote more space in the subsequent sections to explain adverbs, such as `each`, which are fundamental for efficient programming in `q`.

Table 1.1 summarises all data types in `q`. By calling `type` on an object, we obtain one of the following types. The `number` column summarises the output of the `type` as printed out by `q`, which is a short integer. Let us proceed to introduce all types formally.

Recall a positive type is a list, a negative type is an individual atom. And a list of atoms of different types is typed `0h` as we can see from

```
type (2h;2f)
```

```
0h
```

Types 1 to 19 correspond to primitive types, i.e. simple data objects. There are two special "values" for some of the primitive types: the null/missing, and the infinity. Let us look at nulls, or missing values, which are typed entities without a value. Nulls can exist as

TABLE 1.1 The list of all types in kdb/q+ when using `type`

name	character	size	number	format	null	infinity
boolean	b	1	1	0b		
guid	g	16	2		0Ng	
byte	x	1	4	0x00		
short	h	2	5	0h	0Nh	0Wh
int	i	4	6	0i	0Ni	0Wi
long	j	8	7	0 or 0j	0Nj or 0N	0W or 0Wj
real	e	4	8	0e	0Ne	0We
float	f	8	9	0.0 or 0f	0Nf or 0n	0w
char	c	1	10	"a"	" "	
symbol	s		11	`abc	`	
timestamp	p	8	12	2010.01.01D00:01:00.000	0Np	0Wp
month	m	4	13	2010.10m	0Nm	0Wm
date	d	4	14	2010.10.10	0Nd	0Wd
datetime	z	8	15	dateTtime	0Nz	0Wz
timespan	n	8	16	00:00:00.000000000	0Nn	0Wn
minute	u	4	17	00:00	0Nu	0Wu
second	v	4	18	00:00:00	0Nv	0Wv
time	t	4	19	00:00:00.000	0Nt	0Wt
enumerations			20-76			
unused			77			
mapped list of lists t			78-96			
nested sym enumeration			97			
table			98			
dictionary			99			
lambda			100			
unary primitive			101			
binary primitive			102			
ternary operator			103			
projection			104			
composition			105			
f'			106			
f/			107			
f\			108			
f:			109			
f/:			110			
f\:			111			
dynamic load			112			

elements of a vector or list. The general format of a null is `0N?` where `?` is the character entry from Table 1.1.

We can check whether a value is null using the `null` function. This function returns an atom of type boolean, one that can take on the value `0b` (false) or `1b` (true):

```
null 3h
```

```
0b
```

```
null 0Nh
```

```
1b
```

There are a few exceptions among the primitive types: the boolean and byte types do not have a null at all, whereas char and symbol have null values `" "` and `` ` ``, respectively, reflecting their non-numeric format. Besides, for a null float, we may use the shorter notation `0n` instead of `0Nf`, and for a null long we may use the shorter notation `0N` instead of `0Nj`.

```
type 0Nj
```

```
-7h
```

while

```
type 0Nf
```

```
-9h
```

as well as

```
type 0n
```

```
-9h
```

The code is case sensitive, which we can see from

```
type 0N
```

```
-7h
```

The result of basic arithmetic operations involving a null is usually a null itself:

```
0Nj+2
```

```
0N
```

Nulls follow type promotion:

```
0Nj+2.0
```

```
0n
```

```
type 0Nj+2.0
```

```
-9h
```

similar to

```
type 1j+2.0
```

```
-9h
```

In terms of ordering, nulls have the smallest numerical value as can be seen when using `<,>,<=,>=,=`

```
0Nj<-1000
```

```
1b
```

saying it is true that `0Nj` is smaller than a given number `-1000`. This is an illustration. For proof that the null has the lowest numerical value, we need infinities. This is the second special "value" that we have introduced above.

The infinity is generally represented by `0W?`, where `?` is the character from Table 1.1. The concept is in line with what we saw for null types, except for the guid type, which does not have infinity. A float infinity is denoted using lower case, where `0Wf` is the same as `0w`, while `0W` is the same as `0Wj`. Since temporal variables are internally stored as some numeric value, nulls/missing values and infinities are well defined for them, too.

One more thing regarding infinities: they are only well-behaved for floats, where infinity behaves in a proper mathematical sense. Let us illustrate the difference as follows:

```
0W+1
```

```
0N
```

```
0W+2
```

```
-0W
```

```
0W+3
```

```
-9223372036854775806j
```

Long infinity is the largest possible long number. If such a number is incremented by one, we obtain the null long, which, as we have pointed out above, is the smallest long number. Adding two to the long infinity, we obtain the negative long infinity. Adding three to the long infinity, we obtain the next smallest long after the negative long infinity.

On the other hand, the floating point infinities (i.e. the float and real infinities) are well-behaved infinities as we can see from an analogous exercise:

```
0w+1f
```

```
0w
```

```
0w+2f
```

```
0w
```

```
0w=(0w-3f)
```

```
1b
```

Only the floating point infinities demonstrate arithmetic behaviour consistent with the notion of infinity in mathematics.

1.2.1 Casting Types

The casting, i.e. explicit conversion, of types in q is done through the dollar function with the infix notation `"newType"$variable`, where `newType` is a character representation of the new type for the `variable`. The conversion works as intuition suggests:

```
type 1
```

```
-7h
```

while the converted version

```
type "f"$1
```

has the type of

```
-9h
```

An alternative syntax is `` `newType$variable ``, where `newType` is a name representation of the new type for the `variable`:

```
type `float$1
```

```
-9h
```

The character representation is more in the spirit of q to use concise code. For illustration, let us consider type casting for temporal variables:

```
now:2099.01.01D01:02:03.456789000;
type now
```

```
-12h
```

The conversion of the variable now can be directly seen in the output:

```
"j"$now
```

```
3124227723456789000
```

```
"t"$now
```

```
01:02:03.456
```

```
"d"$now
```

```
2099.01.01
```

```
"m"$now
```

```
2099.01m
```

```
"u"$now
```

```
01:02
```

```
"v"$now
```

```
01:02:03
```

```
"n"$now
```

```
0D01:02:03.456789000
```

We encourage the reader to experiment with the casting of various data types on her or his own. We will focus on a special type of conversion which is quite frequent in empirical work: converting a string (i.e. a character vector) into a particular type. The data we work with may come from various sources and may be available in text format.

In order to analyse the data, it will need to be converted into the appropriate data types. Let us illustrate this with `"1.1"`, which is meant to be a float:

```
"f"$"1.1"
```

```
49 46 49f
```

This may be a surprise at first. If we think of the string as an array of individual characters and the conversion as done element-wise, i.e. element-by-element, we realise that the first number `49` is an ASCII representation of the character `"1"`, while the second number `46` is an ASCII representation of the character `"."`.

In order to cast a string into a single float number, we have to use upper case:

```
"F"$"1.1"
```

```
1.1f
```

The conversion is done using the syntax `"NEWTYPE"$variable`, where `NEWTYPE` is a capitalised character representation of the new type for the `variable`. We can see that

```
"J"$"1.1"
```

```
0Nj
```

as we cannot convert it into an integer. This works for other types as well:

```
"D"$"2017.01.01"
```

```
2017.01.01
```

while an improper use of casting gives:

```
"d"$"2017.01.01"
```

```
2000.02.20 2000.02.18 2000.02.19 2000.02.25 2000.02.16 2000.02.18
    ↪ 2000.02.19 2000.02.16 2000.02.18 2000.02.19
```

The reason why the latter casting "works", i.e. does not raise an error, is that the underlying numeric type behind `"d"` is an integer. The casting is thus done character by character, where the dot, for example, gives

```
"d"$"."
```

```
2000.02.16
```

Let us finish this section with conversion to string and symbol. First, the conversion to string is achieved by using the function `string`, which takes one argument, and the

function's result is that argument converted to a string. Let us see `string`'s application in the following examples:

```
string 123.45
```

```
"123.45"
```

The double quotes mean the output is a string. Analogously, we can convert other types like dates, times, etc.

```
string 2017.01.01
```

```
"2017.01.01"
```

The conversion to a symbol, on the other hand, is done using the following notation:

```
`$"123"
```

```
`123
```

We can only convert a string into a symbol. Symbols play a special role in `kdb+` in large tables, and we will cover them in the subsequent sections. For now, they can be described as internalised strings or a string pool which efficiently stores strings internally for fast lookups.

1.3 BASIC LANGUAGE CONSTRUCTS

In the previous paragraphs, we have reviewed some of the basic building blocks for data. In order to build a complex program, we have to learn more operators and language syntax. The syntactic rules of `q` are derived from `k` and do not bear much similarity to standard programming languages such as `C`, `Python`, or `Java`. Indeed, `q` belongs to a different language family.

1.3.1 Assigning, Equality and Matching

The basic element of any program is variables. To assign a value to a variable, the usual = is represented by the assignment operator `:`. It works as follows:

```
a:1.0
```

We confirm that the variable `a` is now equal to 1.0 as:

```
a
```

```
1.0
```

Conveniently, the assignment operator also returns the result of the expression which is assigned. We can, therefore, continue processing:

```
2+a:1.0
```

```
3f
```

In this example, the variable `a` ends up having the value `1f`, whereas the overall result of the expression is `3f`.

Thus we can chain multiple assignments:

```
b:a:1.0
```

It is worth repeating that `q` is case sensitive. Many languages carry some conventions on which types of variables would be lower case and which would be upper case. In `q`, there are no such conventions, partly since few standard libraries exist; however, it is common to use single-letter variables and functions. This is convenient if we want to fit complex functions into a single line – one could argue that solutions to a large number of problems can fit into a single line in `q`. In this book, we follow the *camelCase* convention. This allows us to give intuitive names to building blocks of code and indicate the role of each variable and function by its name.

We should avoid the use of underscore (`_`) in names, as it is an operator in `q`. Another character to avoid is a dot, (`.`), which defines the context or namespace. Finally, `q` contains a limited number of operators and reserved words. We should avoid using them and be especially careful not to define them as table column names accidentally. A common mistake is to name a column `value`, which is a reserved word. In addition, as we will see in the next chapter dedicated to functions, `x`, `y`, and `z` play a special role as default arguments to a function.

Another useful syntactic rule to follow is using the semicolon, `;`. A semicolon denotes the end of a statement. Therefore, the following throws an error, as variable `b` is not defined:

```
b:1 b
```

```
'b
```

yet the following works:

```
b:1; b
```

```
1f
```

For clarity, we encourage putting each statement on a separate line:

```
b:1;
```

and

```
b
```

```
1f
```

Semicolons are *required* when we want to separate statements inside functions. In integrated development environments (IDEs), the use of semicolons allows us to run blocks of code.

We have already seen that the assignment operator, `:`, allows us to assign values to variables. We can check whether the values of the variables are the same using the equality operator, `(=)`:

```
a1:1;a2:2;a1=a2
```

```
0b
```

where we have used the semicolon to fit the example into a single, dense, line. The usage is straightforward when comparing two atoms: we get either `1b` if both atoms are the same, or `0b` if not.

The inequality operator is `<>` and works as expected:

```
a1<>a2
```

```
1b
```

The equality operator is overloaded for lists: if two lists of the same length are provided, the equality operator performs comparison element-wise, and the result is a list of booleans:

```
1 2 3=1 2 4
```

```
110b
```

If a list and an atom are provided, each element of the list is compared against the atom, and the result is again a list of booleans:

```
1 2 3=2
```

```
010b
```

If two lists of different length are provided, the operation is not well defined, and we get a `'length` error, suggesting that unequal lists were provided:

```
1 2 3=1 2
```

```
'length
```

If we want to check whether two lists are an exact match (same length and all respective elements are equal), we use the operator `(~)`, which will compare the lists (or any two variables) as a whole:

```
1 2 3~1 2 3
```

```
1b
```

```
1 2 3~1 2
```

```
0b
```

1.3.2 Arithmetic Operations and Right-to-Left Evaluation: Introduction to q Philosophy

Before we dive into the world of big data analytics and machine learning, we should at least be confident that we know how to sum up or multiply two numbers. Feeling courageous, we attempt to do both:

```
10*100+1000
```

```
11000
```

Wait! The answer should be `2000` under the usual mathematical conventions. `q` thinks otherwise, though. In its quest for efficiency, speed and less program noise – aka "We Don't Need No Stinkin' Brackets" (Borror, 2015) – `q` follows a simple parsing rule: Expressions are evaluated right-to-left, and all operators have the same priority. This may look bizarre to a newbie but results in simplicity, since there is only one rule. One may disagree with this decision, but nothing can be done about it. It also makes the parsing of statements simpler and faster. As a result, the following two expressions yield different results:

```
v:1 2 3
neg[v]+3
```

```
2 1 0
```

```
neg v+3
```

```
-4 -5 -6
```

Our initial example can be written as:

```
1000+10*100
```

```
2000
```

We have explicitly stated that we want to first evaluate `10*100` and then add `1000`. Even though such decomposition may seem obvious, thinking in terms of the operator precedence commonly used in other programming languages, such as `Python`, `Java`, and `C++` may result in trivial but hard to detect errors until our brains get used to the `q` way.

We could have achieved the desired result of `2000` by using brackets in our initial example:

```
(10*100)+1000
```

The motivation for no precedence of operators and using a strict right-to-left evaluation stems from the fact that operator precedence requires to evaluate the full expression to determine the evaluation tree step by step. This is avoided in q and thus the evaluation may proceed as the expression is being read letter by letter. This was also demonstrated in an earlier example with the assignment of a variable and subsequent addition, all in the same statement.

The right-to-left evaluation is one of the biggest sources of misunderstanding and errors during the early stages of coding in q. The difficulty is further pronounced as the same lack of priorities holds for functions. We thus encourage readers who are starting with q to pay extra attention even to the most simple expressions until they get the right-to-left evaluation under their skin. So we will illustrate this concept with another example:

```
2 xexp 2
```

```
4
```

which is as expected. However, when we want to add 1 to the result of the evaluation above, we cannot write:

```
2 xexp 2 + 1
```

as this results in

```
8
```

rather than 5. The solution to this problem is to either use brackets:

```
(2 xexp 2) + 1
```

```
5
```

or

```
1 + 2 xexp 2
```

```
5
```

Finally, in order to divide two numbers in q, we use %

```
5%2
```

```
2.5
```

For now, we should keep in mind that / does something entirely different in q.

1.4 BASIC OPERATORS

There is a number of useful basic operators, or *verbs*, defined in q. These native symbol operators are in fact functions, which are heavily overloaded based on their argument. They form the basics of the programming language and offer powerful functionality for data analysis – familiarity with these and their variations will be required to read and write q.

Random numbers/find/conditional

The symbol ? generates a vector of random numbers. The command to do so is:

```
10?5
```

```
1 2 0 3 4 2 1 4 3 1
```

The previous display generates 10 integers from 0 . . . 4. Since q is derived from C, indexing starts at 0.

In order to generate random floats, we simply replace the integer on the right with a float:

```
10?5.0
```

```
4.096725 2.802929 2.239135 2.933982 1.828879 2.121043 1.233712 3.857119
    ↪ 4.44315 0.2670197
```

where numbers are generated from the range $[0, 5.0]$. This basic functionality thus allows us to generate uniformly distributed random numbers. Random numbers following any other distribution function have to be derived from the uniform distribution.

We can also generate a random array from any specified list. For example, to generate a random word from a list of letters, we type:

```
10?"abcdefg"
```

```
"gfdafbcade"
```

The domain can also be a list of symbols:

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

```
`a`g`b`c`a`a`e`e`a`b
```

If we want to sample without replacement, the first argument needs to be negative. This will guarantee uniqueness of resultant elements:

```
-5?10
```

```
1 9 0 3 4
```

Attempting to generate more numbers than the number of unique instances will give us an error:

```
-10?5
```

```
'length
```

The unique random number generator also works on any list of unique values:

```
-4?`a`b`c`d`e`f`g
```

```
`a`g`b`c
```

However, the unique random number generator does not work when the second argument is a float:

```
-2?2.0
```

```
'type
```

The random number generator uses a seed which can be set using the -S start-up flag or seen by typing:

```
\S
```

```
-314159
```

The seed does not change during the call of the random number generator. If we want to set another seed, we specify an argument to \S which is a non-zero integer.

The function ? further serves as an operator to find an element within a vector. In such a case, the first argument is the list to search through and the second argument is an element to find:

```
arr:1 2 3 4 5 6;
arr?3
```

```
2
```

The function will exit upon finding the first occurrence of the element:

```
arr2:1 2 3 3 3 3;
arr2?3
```

```
2
```

If the element to be found is not present in the list, the query returns an index equal to 1+count[arr]:

```
arr?7
```

```
6
```

Finally, ? allows us to replace elements of one array by elements of another array according to a boolean list:

```
inp1:1 2 3 4 5;
inp2:10 20 30 40 50;
booleanSwitch:01010b;
?[booleanSwitch;inp1;inp2]
```

```
10 2 30 4 50
```

where we have generated the output by choosing all values in `inp1` where `booleanSwitch` is true and values from `inp2` where `booleanSwitch` is false. All three arrays have to have the same length. The boolean vector can be generated by any logical expression; for example, we may produce a vector which always takes the maximum value of two vectors:

```
inp1:1 30 2 6 5;
inp2:10 20 30 40 50;
?[ inp1 > inp2 ;inp1;inp2]
```

```
10 30 30 40 50
```

Note that a more optimal way to do the above is using the built-in `or (|)` function

```
inp1:1 30 2 6 5;
inp2:10 20 30 40 50;
inp1|inp2
```

```
10 30 30 40 50
```

xbar

The function `xbar` rounds each element of the vector on the right down to the nearest multiple of the number on the left.

```
5 xbar 2 -1 3.5 10 13 21
```

```
0 -5 0 10 10 20f
```

But also:

```
0D00:00:05 xbar 2099D01:00:05.1 2099D01:00:06.5 2099D01:00:20.1
```

```
2099D01:00:05.000000000 2099D01:00:05.000000000 2099D01:00:20.000000000
```

The latter being a very convenient way to *bucket* a time vector into 5-second buckets.

Fill

The symbol ^ fills the null elements of an array by a specified value.

```
0^1 2 3 0N 5 6 0N 8
```

```
1 2 3 0 5 6 0 8
```

where the element on the left-hand side of ^ is the value used to replace any `null` in the array on the right-hand side of ^. Fill will promote the value when replacing nulls for each type. Thus, for floats, we may use:

```
10^1.0 2.0 3.0 0n 5.0 6.0 0n 8.0
```

```
1 2 3 10 5 6 10 8f
```

In addition, the argument on the left can be a vector of the same length as the vector on the right. In this case, the null symbols are not replaced by a single value but by the corresponding value from the vector on the left:

```
(10 20 30 40 50 60 70 80)^1.0 2.0 3.0 0n 5.0 6.0 0n 8.0
```

```
1 2 3 40 5 6 70 8f
```

Take/reshape

The symbol # creates a list from the data provided on the right-hand side of # with a dimension specified on the left-hand side. The basic specification is such that the left-hand side argument is an integer and the right-hand side argument is a vector. # takes the first elements of the vector, as specified by the first argument:

```
3#1 2 3 4 5 6
```

```
1 2 3
```

If the first argument is negative, # takes the specified number of elements from the *end* of the vector:

```
-3#1 2 3 4 5 6
```

```
4 5 6
```

If the provided vector is shorter than the length demanded by the integer, the array itself will be repeated:

```
13#1 2 3 4 5 6
```

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

thus, the output has always the length specified by the argument.

The symbol # can also reshape the array. If the left-hand side argument is a two-dimensional array, the output is a matrix with dimensions specified by the arguments formed of an array on the right-hand side, where the vector itself may be repeated:

```
3 3#1 2 3 4 5 6
```

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

Drop/cut

The (_) function drops a subset from a vector. It removes the elements either from the start or the end. In order to drop elements from the start, the first argument is a positive integer denoting the number of elements to drop while the second argument on the right is a list from which the elements are dropped:

```
3_0 1 2 3 4 5 6 7 8 9
```

```
3 4 5 6 7 8 9
```

In order to drop elements from the end, the left-hand side argument of _ is a negative integer denoting the number of elements to drop:

```
-3_0 1 2 3 4 5 6 7 8 9
```

```
0 1 2 3 4 5 6
```

The drop command can be used to drop a particular element from a list. In this case, the argument on the left is the list from which to drop an element while the argument on the right is an integer denoting the position at which the drop is performed:

```
0 1 2 3 4 5 6 7 8 9_3
```

```
0 1 2 4 5 6 7 8 9
```

We have to keep in mind that q starts vector indexing at zero.

We can combine the drop operator with the find operator ? described above. Given the provided list and a certain element, which may be in the list, we may want to keep the part of the list from the first occurrence of the element onwards. Let us illustrate this:

```
array:0 1 2 3 4 5 6 7 8 9;
element:4;
(array?element)_array
```

```
4 5 6 7 8 9
```

The find operation (`array?element`) identifies the position of the first occurrence of `element` in `array`. The drop command is then used to remove the part of the list to the left of the `element`. Alternatively, we may remove elements from the right.

In order to remove the element itself, we swap the order of arguments in the drop command:

```
array _ (array?element)
```

```
0 1 2 3 5 6 7 8 9
```

Finally, if the element is not in the list, the command does nothing as the find operation returns an index equal to the count of the array plus 1:

```
array _ (array?999)
```

```
0 1 2 4 3 5 6 7 8 9
```

The function (_) also cuts the list into partitions, which are specified by the list of indices. The provided indices denote the start of each partition. The cut is used as:

```
0 3 6_ 1 2 3 4 5 6 7 8 9
```

giving as the output the list of partitions, each being an individual list:

```
1 2 3
4 5 6
7 8 9
```

This is as requested: the first partition starts at index 0, the second starts at index 3, i.e. a 4th element of the list, and, finally, the last partition is the rest of the list from index 6 onwards. If the first index specified is different from zero, we obtain the following:

```
1 3 6_ 1 2 3 4 5 6 7 8 9
```

```
2 3
4 5 6
7 8 9
```

Also, the cut always returns a partition in the form of a list:

```
2 3 6 9_ 1 2 3 4 5 6 7 8 9
```

```
,3
4 5 6
7 8 9
`long$()
```

where the first partition is a one-element list while the last partition is an empty list. The command thus always returns the number of partitions which were requested and if the index reaches beyond the length of the array, an empty list is returned.

Max/reverse

The symbol | goes through the provided list element by element and returns the maximum compared to the provided number. It can be used as follows:

```
4|0 1 2 3 4 5 6 7 8 9
```

```
4 4 4 4 4 5 6 7 8 9
```

which is as expected: every element of the vector on the right of the max operator is checked against the number on the left, and the maximum value is returned.

The left-hand side argument can be also a list of the same length as the list to be iterated through. In such a case, the max operator is applied *pairwise*, element by element on both lists:

```
0 10 2 30 4 50 6 70 8 90|0 1 2 3 4 5 6 7 8 9
```

```
0 10 2 30 4 50 6 70 8 90
```

If lists of unequal length are provided, an error is returned:

```
1 1|0 1 2 3 4 5 6 7 8 9
```

```
'length
```

If the max operator is applied on a boolean vector, it acts as boolean "or", which returns `0b` when both arguments are of type `0b`, otherwise it returns `1b`. In particular:

```
1100b | 1010b
```

```
1110b
```

The `(|)` operator can be used to reverse the ordering of the list. In such a case, a single argument is provided:

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

```
9 8 7 6 5 4 3 2 1 0
```

The operator works on any type; in particular, it can be used to reverse the letters in a word:

```
(|) "hello"
```

```
"olleh"
```

`q` prefers to keep the valence constant for a given operator (but overload for a given valence), so the above can also just be written as

```
reverse "hello"
```

```
"olleh"
```

Min/where

The symbol (`&`) goes through the provided list element by element and returns the minimum compared to the provided number, analogously to the `|` operator. The min operator works as follows:

```
4&0 1 2 3 4 5 6 7 8 9
```

```
0 1 2 3 4 4 4 4 4 4
```

which works as expected: every element of the array on the right-hand side of the min operator is checked against the number on the left-hand side of the operator and the minimum value is returned.

The left-hand side argument can be also a list of the same length as the list to be iterated through. Similar to the max operator, the min operator is applied element by element on both lists:

```
0 10 -2 30 -4 50 -6 70 -8 90&0 1 2 3 4 5 6 7 8 9
```

```
0 1 -2 3 -4 5 -6 7 -8 9
```

When lists of unequal length are provided, the error `'length` is returned.

Analogously to the max operator, when min operator is applied on a boolean vector it acts as a boolean "and", which returns `1b` when both arguments are true, `1b`, otherwise it returns false `0b`:

```
1100b & 1010b
```

```
1000b
```

Finally, the `where` operator takes a single argument of a boolean list:

```
where 0101b
```

```
1 3
```

which returns the indices of all elements which can be interpreted as true, i.e. non-zero.

More generally, `where` applied to a vector of integers (or a dictionary with integer values) can be defined as a function returning the number of occurrences of those integers according to the integer values:

```
where `a`b`c`d!til 4
```

```
`b`c`c`d`d`d
```

The argument on the left in the above example is a *dictionary*, which will be introduced in the next chapter.

Join

The symbol , is used to join two lists:

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

```
01 2 3 5 6 7 8 9
```

The join operator can join lists of any type:

```
(0 1 2 3 4), `a, "hello"
```

```
((0 1 2 3 4);`a;"hello")
```

Print to console

The symbol ! has important functionalities which will be reviewed later. The particular function involving ! is:

```
0N!3
```

```
3
```

which prints the argument on the right-hand side to console in an unformatted form but also allows q to use the argument further down the statement for subsequent operations on the left. In particular, when using the q prompt, we can use 0N! as control prints:

```
1+0N!1 +0N!1+0N!1
```

```
1
2
3
4
```

This is useful to track the evaluation of a complicated statement – such functionality, however, does not work when using IDEs.

Separator

We saw that the symbol ; denotes the end of a statement. However, when surrounded by brackets, it acts as a separator for elements within a list:

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

```
1 2 3 4
```

or nested lists:

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

```
1
2
1 2
4
```

As we will see in the next chapter, the symbol ; also separates arguments when defining a function. Finally – as we saw previously – it separates statements:

```
a:1;b:2;a+b
```

```
3
```

We may easily write complicated commands within one line. If a separator is used to separate statements, the statements are evaluated one by one from the left, but within a statement, the right-to-left priority is used. q does not consider the end of a line (newline operator) as the end of a statement so one statement can also be written in multiple lines for better clarity. For reducing clutter in our code, we should also keep separate statements in separate lines.

Quit

The symbol \\ is not a symbol per se but plays an important role. When entered in the q session, it will terminate the session.

Index

The "at" sign, @, is known as "apply" and can be used to apply a monadic (one-argument) function to its argument. Since q is a functional language, indexing is a function, too, and we can use apply to index a vector to a list of indices. The following forms are all equivalent:

```
max[1 2 3]
max 1 2 3
max@1 2 3
```

```
x:1 2 3
x[0 1]
x 0 1
x@0 1
```

Precedence/list

`()` is primarily used to define the precedence execution since `q` evaluates expressions from right to left. When using brackets, the evaluation is commanded to start from the innermost bracket pair and move outwards:

```
((1+2)*(3+4))
```

```
28
```

and contrast this to the unbracketed form of the expression

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

```
15
```

The usage of brackets is very convenient as we can get the precedence order as required; however, it brings some overhead and in some cases can create more "noisy" code. The `()` can be used to form lists and create nested lists:

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

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

which is a nested list of lists. In the topmost layer, the list contains nested list `(((1;2);3);4)` and atom `5`. `()` is also used to create tables, as we will see in the next chapter.

We saw earlier that `enlist` is a compelling function which constructs a list out of its input. The opposite operation of 'flattening' a list (or 'razing' it) is achieved by `raze`:

```
raze ((((1;2);3);4);5)
```

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

Block

The square brackets, `[]`, form a block of code which can be useful inside conditional expressions:

```
[a:1;b:2;a+b]
```

```
3
```

`[]` does not change evaluation precedence of statements but rather forms a sub-statement, which is evaluated as a block of code.

The brackets [] have two other functions in q: First, they denote the argument of the function, and, second, they form special multi-valence functions. Both functionalities will be described in Chapter 3.

Assign/amend

The symbol : is the assignment operator as we previously saw. The existing value assigned to a variable can be amended either by explicitly stating the operation:

```
a:1;
a:a+1;
a
```

```
2
```

or by using the short notation of

```
a:1;
a+:1;
a
```

```
2
```

This can be applied to other arithmetic operations like +,*,%,:

```
a:2;
a*:2;
a
```

```
4
```

Further, the same works if we want to amend the list by joining it with another list:

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

```
1 2 3 4
```

Colon : also provides the "return" functionality within functions.

Identity

The symbol :: stands for the identity function, which returns its argument:

```
(::) 1
```

```
1
```

The identity is useful whenever we need to use a function, which outputs the arguments. We shall cover a use case for this in later chapters when we cover table joins.

The symbol :: stands for the identify function, but also denotes nil (generic null), global amend, and view. This functionality will be revealed through examples in the subsequent sections, when we will cover q in more detail.

1.5 DIFFERENCE BETWEEN STRINGS AND SYMBOLS

In q, we have two similar types, the string and the symbol, which seemingly do the same job. Imagine we have a database, which contains a timestamp, price and also the name of the asset the price belongs to, e.g. IBM, INTEL, AAPL. We may store the asset names as strings; however, we end up having a large number of repeated values, which will be consuming memory. We may prevent wasting memory by replacing the names by short integers, for example, and keep a separate lookup table to recover the map between short integers and names, in the form of a dictionary:

```
nameToShortMap:`IBM`INTC`AAPL!(1 2 3h);
```

This is not a very convenient way to work with data and in particular to share our work with others. In order to avoid such a troublesome approach, q uses the concept of symbols, which are internalised strings. Such strings are kept in memory only once, i.e. each distinct value is kept in memory once, and references to the original string are internally used. Under the surface, q maintains a pool of strings used as symbols and every usage of a symbol points to the value in this table.

The notion of symbols thus clearly suggests that whenever we need to use strings repeatedly, symbols are the type of choice. As the pool of strings is maintained in memory, we should constrain the use of symbols to domains with repeated values to avoid blowing up the memory. If we need unique and complicated names, we should stick to characters, i.e. strings. Alternatively, if we need to use unique identifiers, like trade IDs, we should use the guid type.

Another advantage of using symbols is that the comparison is much faster since we do not need to check every character, but merely the reference to that string. For instance, queries like

```
where name=`IBM
```

or

```
where name in `IBM`INTC
```

significantly improve the speed and simplicity of evaluation compared to using character vectors.

1.5.1 Enumeration

The q language allows us to go one step further when working with symbols and create enumerated lists. This should be a familiar concept from other languages (q *does* contain various features we are used to in "common" programming languages). The enumerated list of symbols form a new type – enumerations have a type range of short integers – and restrict the values within the existing domain. Any changes to the new type in terms of appending or changing the values are restricted to the provided list.

Let us create an enumerated list out of our three names of assets used above:

```
assetNames:`IBM`INTC`AAPL;
```

and a list of names within our hypothetical analysis involving multiple occurrences of the vector of assetNames:

```
name:`IBM`INTC`IBM`INTC`AAPL`IBM`IBM`INTC`AAPL`IBM`INTC`INTC`AAPL`IBM`AAPL;
type name
```

```
11h
```

We now "cast", i.e. enumerate the list of symbols against the domain of values:

```
name2:`assetNames$name;
type name2
```

```
20h
```

as expected from Table 1.1; enumerated types take values from 20h to 76h so the actual outcome will depend on what was already enumerated within the q session.

We can see what name2 looks like:

```
name2
```

```
`assetNames$`IBM`INTC`IBM`INTC`AAPL`IBM`IBM`INTC`AAPL`IBM`INTC`INTC`AAPL`
    ↪ IBM`AAPL
```

Let us try to append a value into the list name2 which is within the specified type, and one which is not;

```
name2,:`IBM
```

i.e. no problem, the new item was added. In contrast,

```
name2,:`C
```

```
'cast
```

and thus the specified type restricts us to the list of values within an enumerated list. We cannot add `C into our analysis unless we specify a new item into assetNames.

If we try a slightly different version of the previous command

```
name2:name2,`C
```

we do not obtain any error message and the command will go through without any issue. Why? Let us investigate:

```
type name2
```

```
11h
```

q has recast name2 back into a non-enumerated symbol type. This is because in the first case we amended name2 *in-place*, i.e. by reference, and the new value was missing from our enumeration. In the second example, q un-enumerated the result of name2,`C since `C was not an element of the enumeration domain.

1.6 MATRICES AND BASIC LINEAR ALGEBRA IN q

Matrices are represented in q as nested lists, specifically a list of columns, where the elements of the outer list are the rows of the matrix:

```
a: (2.4 -0.7 20.4; 5.7 9.8 -2.3)
```

which represents the matrix

$$\begin{pmatrix} 2.4 & -0.7 & 20.4 \\ 5.7 & 9.8 & -2.3 \end{pmatrix}.$$

One can use the dyadic take function # to create such lists of lists:

```
c: 2 3 # 2.4 -0.7 20.4 5.7 9.8 -2.3
```

The match function ~ can be used to check for matrix equality:

```
a ~ c
```

```
1b
```

or

```
b: (2.4 10.8 2.6; 1.2 3.1 0.8);
a ~ b
```

```
0b
```

Function `flip` can be used to transpose a matrix. We encourage readers to write an example. Functions + and – operate element-wise, so can be used to add and subtract matrices:

```
a + b
```

```
(4.8 10.1 23;6.9 12.9 -1.5)
```

```
a - b
```

```
(0 -11.5 17.8;4.5 6.7 -3.1)
```

```
a - c
```

```
(0 0 0f;0 0 0f)
```

There are several ways to reference an element of a matrix. With the dot verb and using the fact that the matrix is also a binary function (the @ verb cannot index more than one dimension):

```
a . 1 2
```

```
-2.3
```

```
.[a; 1 2]
```

```
-2.3
```

We can access the relevant row of the matrix, then index within that row:

```
a[1][2]
```

```
-2.3
```

Finally, we can use the fact that the matrix is also a binary function:

```
a[1;2]
```

```
-2.3
```

However, only this latter method can be used to mutate an element:

```
a[1;2]: -3.2;
a
```

```
(2.4 -0.7 20.4;5.7 9.8 -3.2)
```

We can access the entire row . . .

```
a[1]
```

```
5.7 9.8 -3.2
```

. . . or column of a matrix:

```
a[;1]
```

```
-0.7 9.8
```

and mutate them:

```
a[;1]: 1.4 -8.9;
a
```

```
(2.4 1.4 20.4;5.7 -8.9 -3.2)
```

Here we have *elided* the very top index of a matrix: `a[;1]` retrieves the items with index 1 from each item of the nested list `a`.

The `@` verb can also be used to index rows of the matrix:

```
@[a; 1]
```

```
5.7 -8.9 -3.2
```

Matrices of compatible shapes can be multiplied using the matrix multiplication verb `mmu`. Thus to multiply `a` by a transpose of `b`

```
d: a mmu flip b;
d
```

```
(73.92 23.54;-90.76 -23.31)
```

The inverse of a nonsingular matrix can be computed using `inv`:

```
inv d
```

```
(-0.05638399 -0.05694034;0.2195372 0.1788033)
```

Note that for `mmu` and `inv` to work the matrix must have floating point entries. Thus the following will work:

```
(3 5 7f; 6 4 7f) mmu (3 2f; 12 2f; 5 2f)
```

whereas

```
(3 5 7; 6 4 7) mmu (3 2; 12 2; 5 2)
```

will fail with error `'length`.

Dot product is supported through the $ operator:

```
a[0]$b[0]
```

```
51.24
```

We have demonstrated that we can implement basic linear algebra calculations within `q` in a very straightforward way. The provided commands can be extended by readers into more complex functions and solve more advanced problems.

1.7 LAUNCHING THE SESSION: ADDITIONAL OPTIONS

Let us conclude this section with an overview of various startup flags we can use when starting our `q` session:

```
> q <additional options>
```

where a number of useful `<additional options>` is specified below.

Block writing

The option `–b` prevents the user, or the client launched by the user, to write into memory. If we launch a session with this option, any client which connects to this session cannot

write anything into any existing variable or create a new one. It returns `'noupdate` error when any attempt to write is made:

```
a:1
```

```
'noupdate
```

In Chapter 5 we will see another way to replicate read-only mode dynamically when desired, by using the `reval` function.

Console dimension

The option `–c rows columns` sets the console size in terms of rows and columns. The default setup is `–c 25 80`. In the case when `q` is invoked through the HTTP protocol, the dimension of the display can be set by using the capital "C", i.e. `–C rows columns`.

Error trap

The option `–e B` with `B` being binary variable sets the error trapping in the console. The error trapping is crucial for code debugging, which we will cover in Chapter 5.

Garbage collection

The option `–g b` sets the garbage collection mode. `q` has two memory modes: the deferred mode, `–g 0`, and the immediate mode, `–g 1`. The default is the deferred mode, where the memory is returned to the operating system when `q` is instructed to do so by running `.Q.gc` function or when memory allocation fails. The immediate mode is returning the memory to the operating system when it is not referenced any more. The trade-off is that immediate mode allows for more continuous memory management but comes with some overhead.

On the other hand, the deferred mode gives us control over when it happens. Notice that running `.Q.gc[]` explicitly also attempts to coalesce diced memory blocks and can, therefore, be more expensive – but also return more memory – when run, especially in a process with a very fragmented heap, which usually happens when there are many nested lists created. Depending on the trade-off between required memory to be freed and performance-critical code, we may want to choose the appropriate tool for the appropriate scenario. For example, we would usually not add frequent calls to `.Q.gc` on a time-sensitive complex event processing (CEP) process, but may do so during a quiet period or after persisting data to disk, say, once a day. Note that `q` *always returns unreference memory to the heap*, so the `g` flag or explicitly running `.Q.gc` are useful when wanting to *return memory back to the OS*, recommended when having multiple `q` processes or other applications competing for the same memory on a server.

Logs

The option `–l` sets the logging of the session. The log can be enabled by setting:

```
›q fileName –l
```

This option will create a file `fileName.log`, which stores the messages for any queries which change the state of the data. This option is useful to prevent the loss of data when the server crashes.

Time offset

The option `-o N` sets the offset of the internal time from GMT. The variable `N` denotes hours, if $|N| < 23$, otherwise it represents minutes. Thus, the shift by one hour forward can be expressed as:

```
> q -o -1
```

This is useful when working in different time zones and `q` is used to store the data, for example.

Port

The option `-p N` sets the port for the session. This option is needed if we want to connect to the session remotely, or if we use GUI to code in `q`. The variable `N` is the port number. It is convenient to use large values, e.g. the following option would conveniently set the port number to 6000:

```
> q -p 6000
```

The variable `N` can be negative, in which case the port will be set for a multi-threaded mode.

Display precision

The option `-P N` sets the number of decimals of the float type displayed by the terminal and when exporting data to text, such as `csv` format. The variable `N` is the integer denoting the number of decimals.

Quiet start

The option `-q` suppresses the announcement at the start-up, and the user can start coding without a single word being heard off from the `q` terminal.

Parallel executions

`q` can execute code in parallel on multiple slave threads or processes. The option `-s N` sets the environment for parallel execution, with `N` being positive or negative integer. In Chapter 8 we will cover the required setup.

Timeout

The option `-T N` sets the timeout for queries. The variable `N` is in seconds. The timeout works such that if the query lasts more than `N` seconds, the query is interrupted unfinished and the session ends up with an error. The default is no timeout, `-T 0`. The timeout option is useful when there are limited resources and the administrator has to prevent long queries.

User restrictions

The option `-u X` sets the user restrictions for the given session. The main objective is to limit the users who connect to the session remotely, e.g. through the GUI, which we will cover later, or over the network. The first option is `-u 1`, which prevents users from launching any system command. In particular, this prevents any remotely connected user to quit the session (recall the command `\\`).

The second option is to launch session with `-u fileName`, where "fileName" is a file to the username and password. The file can be located in the main `q` directory – in such a case the name of the file is provided – or anywhere else – in such a case the path has to be added. The file itself can contain, for example, the following line:

```
userName:password
```

In order to access the session remotely, the provided `userName` and `password` has to be specified. When this option is set, the user does not have access above the directory where the session started. This prevents access into other parts of the system. Such a restriction can be levied when we use a capital letter in the option `-U fileName`. In such a case, the user who knows the login details gets the full access to the other parts of the system.

Workspace limit

The option `-w N` sets the workspace limit for the session. The provided number, `N`, is in MB.

Day of the week

The option `-W N` sets the start of the week. The week by default starts on Monday, which corresponds to $N = 2$. This can be overridden by setting a different value to `N`.

Date format

The option `-z B` with `B` being binary variable sets the format of a date. Namely, the (default) option `-z 0` sets the US style for the date "MM/DD/YYYY", while the option `-z 1` sets the European style for the date "DD/MM/YYYY".

1.8 SUMMARY AND HOW-TO'S

In this final section we provide a refresher of basic operations on vectors and present common how-to's:

Create a vector of natural numbers

```
N:10
v:1+til N
```

Access the first, last, first *n*, and last *n* elements of a vector

```
first v
last v
n:5
n#v
neg[n]#v
```

Drop *n* elements from the start or end of a vector

```
n _v
neg[n]_v
```

Update multiple elements of a vector

```
v[0 9]:0N  / updates with null integer
```

Update multiple elements of a vector where a specific condition occurs

```
v[where 0=v mod 2]:0  / update the values with no remainder after dividing
    ↪ by 2, with zero
```

Sample *n* float numbers uniformly between −5 and 5

```
r:-5+n?10f
```

Update the first element of all records of a matrix or a 2-dimensional list

```
r:-5+n?10f
m:2 2#r
m[;0]:1f
```