

Fundamentals of the J Programming Language

In this chapter, we present the basic concepts of J. We introduce some of J's built-in functions and show how they can be applied to data objects. The principals presented in this book are supported by examples. The reader therefore, is strongly advised to download and install a copy of the J interpreter from www.jsoftware.com.

2.1 Data Objects

Here, we give a brief introduction to data objects in J. Data objects come in a number of forms: scalars, vectors, matrices and higher-dimensional arrays. Scalars are single numbers (or characters); the examples below show the values assigned to variables names:

```
i =: 3          NB. integer
j =: _1         NB. negative integer
x =: 1.5983     NB. real
y =: 22r7       NB. rational
z =: 2j3        NB. complex number
m =: _         NB. infinity
n =: __        NB. negative infinity
v =: 5x2        NB. exponential notation 5 * exp(2)
```

Note that negative numbers are denoted by an underscore (`_`) preceding the value rather than by a hyphen (`-`). An underscore on its own denotes infinity ∞ , and two underscores denotes negative infinity $-\infty$. Variables are not limited to numerical values:

```
c =: 'hello world'
```

Variables can be evaluated by entering the variable name at the command prompt:

```

c
hello world

```

Scalars are called *atoms* in J or *0-cells*, and vectors are called lists or 1-cells. The sequence of numbers below is an example of a list and is assigned to a variable name:

```

dvc =: 1 1 2 3 5 8 13 21 34 55
dvc
1 1 2 3 5 8 13 21 34 55

```

The `i.` verb generates ascending integers from zero $n - 1$, where n is the value of the argument. For example:

```

z1 =: i.10 NB. generate list 0 to 9
z1
0 1 2 3 4 5 6 7 8 9

```

The associated verb `i:` generates integers from $-n$ to n , thus:

```

z2 =: i:5 NB. generate list -5 to 5
z2
_5 _4 _3 _2 _1 0 1 2 3 4 5

```

Matrices (tables in J) are 2-cell objects. Here is a table of complex numbers:

```

j =: (i.6) j./ (i.6)
j
0 0j1 0j2 0j3 0j4 0j5
1 1j1 1j2 1j3 1j4 1j5
2 2j1 2j2 2j3 2j4 2j5
3 3j1 3j2 3j3 3j4 3j5
4 4j1 4j2 4j3 4j4 4j5
5 5j1 5j2 5j3 5j4 5j5

```

Matrices (of ascending integers) can be generated with the `i.` verb. The example below shows a 3×2 matrix:

```

i. 2 3 NB. generate a matrix
0 1 2
3 4 5

```

Higher-dimensional arrays are also possible. The expression below generates an array of reciprocals with ascending denominators:

```

%i. 2 4 3
      _      1      0.5
0.333333      0.25      0.2
0.166667 0.142857      0.125
0.111111      0.1 0.0909091

0.0833333 0.0769231 0.0714286
0.0666667      0.0625 0.0588235
0.0555556 0.0526316      0.05
0.047619 0.0454545 0.0434783

```

This 3-cell data object is a three-dimensional array with three columns, four rows and two *planes*, where the planes are delimited by a blank line.

Scalars/atoms, vector/lists and matrices/tables are just special instances of arrays with respective ranks zero, one and two. Table 2.1 shows the J terms for data objects and their mathematical equivalents. Throughout this book, we will use the J and mathematical terms interchangeably.

J term	Mathematical term	Dimension	Object type
atom	scalar	0	0-cell
list	vector	1	1-cell
table	matrix	2	2-cell
array	array	n	n -cell

Table 2.1. J versus mathematical terms

2.2 J Verbs

In J, functions are called *verbs*. J has a number of built-in verbs/functions; for example, the basic arithmetic operators are: +, −, *, % and ^, which are addition, subtraction, multiplication, division and power functions, respectively. Here are some (trivial) examples:

```

2 + 3      NB. addition
5
2 - 3      NB. subtraction
_1
7 * 3      NB. multiplication
21
2 % 7      NB. division: "%" is used instead of "/"
0.285714

```

```
2^3      NB. power
8
```

In J, there is a subtle (but important) difference between `_1` and `-1`. The term `_1` is the value minus one, whereas `-1` is the negation function applied to (positive) one.

Arithmetic can also be performed on lists of numbers:

```
2 % 0 1 2 3 4 NB. divide scalar by a vector
_ 2 1 0.666667 0.5
3 2 1 0 - 0 1 2 3 NB. pointwise vector subtraction
3 1 _1 _3
```

The example above demonstrates how J deals with divide by zero. Any division by zero returns a `_` (∞). In addition to the basic arithmetic operators, J has many more primitives, for example:

```
+ : 1 2 3 4      NB. double
2 4 6 8
- : 1 2 3 4      NB. halve
0.5 1 1.5 2
% : 1 4 9 16     NB. square root
1 2 3 4
* : 1 2 3 4      NB. squared
1 4 9 16
> : _1 0 1 2 3   NB. increment
0 1 2 3 4
< : _1 0 1 2 3   NB. decrement
_2 _1 0 1 2
```

Verbs are denoted by either a single character (such as *addition* `+`) or a pair of characters (such as *double* `+ :`). Some primitives do use alphabetic characters, for example, the *integers* verb `i.` and *complex* verb `j.`, which were introduced above.

2.3 Monadic and Dyadic functions

Each verb in J possesses the property of *valance*, which relates to how many arguments a verb takes. *Monadic* verbs take one argument (and are therefore of valance one), whereas *dyadic* verbs take two arguments (valance two).

Monadic verbs are expressed in the form: `f x`. The (single) argument x is passed to the right of the function f . In functional notation, this is equivalent to $f(x)$. In its dyadic form, f takes two arguments and are passed to the function on either side: `y f x`, equivalent to $f(y, x)$ in functional notation.

J's verb symbols are *overloaded*; that is, they implement two separate (often related, but sometimes inverse) functions depending upon the valance. We use the % primitive to demonstrate. We have already seen it used in its dyadic form as a *division* operator. However, in its monadic form, % performs a *reciprocal* operation:

```
% 2      NB. used monadically is reciprocal
0.5
3 % 2     NB. used dyadically is division
1.5
```

Let us look at a few more examples. The monadic expression \hat{x} is the *exponential* function of x : e^x . The dyad y^x , however, performs y to the power x , that is: y^x . To illustrate:

```
^ 0 1 2 3      NB. used monadically is exp(x)
1 2.71828 7.38906 20.0855
2 ^ 0 1 2 3    NB. used dyadically is y^x
1 2 4 8
```

Used monadically $<:$ performs a *decrement* function:

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

However as a dyad it performs *less-than-or-equal-to*¹:

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

2.4 Positional Parameters

The meaning of a *positional parameter* is given by virtue of its relative position in a sequence of parameters. J does not really have a concept of positional parameters; however, we can pass positional parameters to functions as an ordered list of arguments. In this section, we introduce verbs for argument processing: *left* [and *right*]. These verbs return the left and right arguments, respectively:

```
2 [ 3
2
2 ] 3
3
```

¹ Similarly $>:$ performs *increment* and *greater-than-or-equal-to*.

The *right* provides a convenient means of displaying the result of an assignment:

```
]x =: i.10
0 1 2 3 4 5 6 7 8 9
```

The *left* verb can be used to execute two expressions on one line:

```
x =: i.10 [ n =: 2 NB. assign x and n
x ^ n
0 1 4 9 16 25 36 49 64 81
```

The verbs *head* { . and *tail* { : return the first element and the last element of a list:

```
{. x
0
{: x
10
```

Conversely, *drop* } . and *curtail* } : remove the head and tail of a list and return the remaining elements:

```
} . x
2 4 6 8 10
}: x
0 2 4 6 8
```

The *from* verb { is used to extract a particular element (or elements) within a list, by passing the index of the required element in the list as a left argument:

```
0 { x
0
2 { x
4
3 1 5 { x
6 2 10
0 0 0 1 2 2 2 3 3 4 5 5 5 5 { x
0 0 0 2 4 4 4 6 6 8 10 10 10 10
```

Lists can be combined with *raze* , . and *lamine* , : in columns or rows, respectively. The J expressions below yield two matrices m_1 and m_2 :

```
]m1 =: 1 2 3 4 ,. 5 6 7 8
1 5
2 6
3 7
4 8
```

```

]m2 =: 1 2 3 4 , : 5 6 7 8
1 2 3 4
5 6 7 8

```

We cover higher-dimensional objects in more detail in Section 2.6. Here, we look briefly at applying the *from* verb to matrices:

```

2 { m1
3 7
1 { m2
5 6 7 8

```

Here, *from* returns the third row of m_1 in the first example and the second row of m_2 in the second example. If we wish to reference an individual scalar element, then we need to use *from* twice:

```

0 { 1 { M2
5

```

In order to reference a column, we need to be able to change the *rank* of the verbs. The concept of rank will be covered in the next section. Two data objects can be concatenated with *ravel* (*,*), for example:

```

v1 =: 1 2 3 4
v2 =: 5 6
]v3 =: v1,v2
1 2 3 4 5 6
0 3 4 5 { v3
1 4 5 6

```

This creates a single list of eight elements (v_3). There is no separation between the two original lists v_1 and v_2 . If we wished to retain the separation of the two initial lists, then we combine them with the *link* verb *;*, for example:

```

]v4 =: v1;v2
+-----+----+
| 1 2 3 4 | 5 6 |
+-----+----+

```

The lists are “boxed” and therefore exist as separate data objects. We can reference the two lists in the usual way:

```

0 { v4
+-----+
| 1 2 3 4 |
+-----+

```

The data object returned is a single element; we cannot get at any of the individual scalar elements in the box:

```
1 { 0 { v4
|index error
| 1 { 0{v4
```

Use *open* > to unbox the object:

```
> 0 { v4 NB. unbox v1
1 2 3 4
1 { > 0 { v4
2
```

There is a corresponding inverse function, namely the monadic verb *box* <, which “groups” elements:

```
<1 2 3 4
+-----+
|1 2 3 4|
+-----+
```

Using the primitives described above, we define a number of functions for referencing positional parameters. These functions will be used a great deal in developing functions later in this book. Note that a couple of conjunctions are used here (& and @) will be covered later, in Section 3.1.4

```
lhs0 =: [ NB. all left arguments
lhs1 =: 0&{@lhs0 NB. 1st left argument
lhs2 =: 1&{@lhs0 NB. 2nd left argument
lhs3 =: 2&{@lhs0 NB. 3rd left argument

rhs0 =: ] NB. all right arguments
rhs1 =: 0&{@rhs0 NB. 1st right argument
rhs2 =: 1&{@rhs0 NB. 2nd right argument
rhs3 =: 2&{@rhs0 NB. 3rd right argument
```

The functions *lhs0* and *rhs0* evaluate the left and right arguments, respectively. The other functions are programmed to return positional parameters, thus *lhs1* (respectively *rhs1*) returns the first positional parameter on the left-hand side (respectively right-hand side). We illustrate the use of positional parameters with the following example. Consider the classical M/M/1 queuing model given in Equation (1.6). We can write a function that takes the parameters μ and ρ as left-hand arguments and right-hand arguments, respectively:

```
mm1 =: %@lhs1 % 1:-rhs0
3 mm1 0.5 0.6 0.7 0.8 0.9
0.666667 0.833333 1.11111 1.66667 3.33333
```


2.5 Adverbs

The (default) behaviour of verbs can be altered by combining them with *adverbs*. We have already encountered an adverb with the summation function `+/`. The application of `/` causes `+` to be inserted between the elements of the argument, in this case, the individual (scalar) numbers in the list.

```
+/ i.6                      NB. as we've seen before
15
0 + 1 + 2 + 3 + 4 + 5      NB. and is equivalent to this
15
```

The dyadic case results in a matrix of the sum of the elements of the left argument to each element of the right argument.

```
(i.6) +/ (i.6)
0 1 2 3 4 5
1 2 3 4 5 6
2 3 4 5 6 7
3 4 5 6 7 8
4 5 6 7 8 9
5 6 7 8 9 10
```

The *prefix* adverb `<\` causes the data object to be divided into sublists that increase in size from the left; the associated verb is then applied to each sublist in turn. We can see how the sublist is generated using the *box* verb:

```
<\ i.5
+---+---+---+---+
|0|0 1|0 1 2|0 1 2 3|0 1 2 3 4|
+---+---+---+---+
```

A cumulative summation function can be implemented using the *insert* and *prefix* verbs:

```
+/ \ i.6
0 1 3 6 10 15
```

This function will be useful later on when we wish to convert interval traffic arrival processes to cumulative traffic arrival processes. The *suffix* `<.` operates on decreasing sublists of the argument:

```
<. i.6
+---+---+---+---+---+
|0 1 2 3 4 5|1 2 3 4 5|2 3 4 5|3 4 5|4 5|5|
+---+---+---+---+---+
```

The monadic *reflexive* adverb `~` *duplicates* the right-hand argument as the left-hand argument. So the J expression `f~ x` is equivalent to `x f x`, for example:

```

+/~ i.6 NB. (i.6) +/ (i.6)
0 1 2 3 4 5
1 2 3 4 5 6
2 3 4 5 6 7
3 4 5 6 7 8
4 5 6 7 8 9
5 6 7 8 9 10

```

The `~` verb also has a dyadic form (*passive*). This means that the left and right-hand side arguments are swapped; that is, `y f x` becomes `x f y`, as an illustration:

```

2 %~ i.6 NB. equivalent to (i.6) % 2
0 0.5 1 1.5 2 2.5

```

2.6 Rank, Shape and Arrays

Arithmetic can be performed between a scalar and a list or between two lists, for example:

```

2 * 0 1 2 3 4 5
0 2 4 6 8 10
0 1 2 3 4 5 + 1 2 3 4 5 6
1 3 5 7 9 11

```

Notice that the lists have to be the same length; otherwise the J interpreter throws a “length error”:

```

9 8 - 0 1 2 3 4 5
|length error
| 9 8 -0 1 2 3 4 5

```

J can also perform arithmetic on higher-dimensional objects. In this section, we introduce arrays as well as the concepts of *rank* and *shape*. Rank is synonymous with dimensionality; thus a two-dimensional array has rank two, a three-dimensional array has rank three. Verbs have rank attributes which are used to determine at what rank level they should operate on data objects. We will explore this later. First, let us consider at how we can define array objects by using the dyadic *shape* verb `$`:

```

]x2 =: 2 3 $ i.6
0 1 2
3 4 5

```

As we have already seen, we could have defined this particular array, simply by:

```
i. 2 3
0 1 2
3 4 5
```

This is fine for defining an array with ascending integers (as returned by `i.`), but if we wanted to form an array using some arbitrary list of values, then we need to use `$`. We will continue to use the `$` method, although we acknowledge that it is not necessary, as the data objects used in these examples are merely ascending integers. The shape is specified by the left arguments of `$` and can be confirmed using the `$` in its monadic form:

```
$ x2
2 3
```

The data object x_2 is a (3×2) two-dimensional array, or, in J terms, a rank two object (of shape `2 3`). Arithmetic can be applied in the usual way. This example shows the product of a scalar and an array:

```
2 * x2
0 2 4
6 8 10
```

Here we have the addition of two arrays (of the same shape):

```
x2 + (2 3 $ 1 2 3 4 5 6)
1 3 5
7 9 11
```

J can handle this:

```
2 3 + x2
2 3 4
6 7 8
```

But apparently not this:

```
1 2 3 + x2
|length error
| 1 2 3 +x2
```

J of course, *can* handle this, but we need to understand more about the *rank* control conjunction `"` which will be covered in Section 3.1 below. Consider a $3 \times 2 \times 2$ array:

```

]x3 =: 2 2 3 $ i.12
0 1 2
3 4 5

6 7 8
9 10 11

```

x_3 is a three-dimensional array and, therefore, of rank three. J displays this array arranged into two *planes* of two rows and three columns, where the planes are delimited by the blank line. We can confirm the structure of x_3 by using $\$$ as a monad, where it performs a *shape-of* function:

```

$ x3
2 2 3

```

Now, let us apply summation to x_3 :

```

+ / x3
6 8 10
12 14 16

```

Here, the individual elements of the two *planes* have been summed; that is:

$$\begin{pmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{pmatrix} + \begin{pmatrix} 6 & 7 & 8 \\ 9 & 10 & 11 \end{pmatrix} = \begin{pmatrix} 0+6 & 1+7 & 2+8 \\ 3+9 & 4+10 & 5+11 \end{pmatrix} \\
 = \begin{pmatrix} 6 & 8 & 10 \\ 12 & 14 & 16 \end{pmatrix}$$

It is important to understand why $+ /$ sums across the planes rather down the columns or along the rows. First consider this example:

```

% x3
      _      1      0.5
0.333333      0.25      0.2

0.166667 0.142857      0.125
0.111111      0.1 0.0909091

```

Aside from the difference between the arithmetic functions $+ /$ and $\%$ perform, they also operate on the argument in a different way. Where as $+ /$ operated on the two planes, here $\%$ is applied to each individual scalar element. The difference in the behaviour of the two verbs $+ /$ and $\%$ is governed by their respective rank attributes. We can query the rank attribute of verbs with the expressions below:

```

% b. 0
0 0 0
+ / b. 0
- - -

```

Three numbers are returned. The first number (reading left to right) is the rank of the monad form of the verb. The second and third numbers are the ranks of the left and right arguments of the dyadic form of the verb. When a verb performs an operation on an object, it determines the rank of the cell elements on which it will operate. It does this by either using the rank (dimension) of the object or the rank attribute of the verbs, whichever is smaller. In the example above, x_3 has a rank of three, and the (monadic) rank attribute of `%` is zero. So `%` is applied to x_3 at rank zero. Thus it is applied to each 0-cell (scalar) element. However, the rank attribute of `+/` is infinite, and, therefore, the rank, at which the summation is performed, is three. Thus, `+/` applies to each 3-cell element of x_3 , resulting in a summation across the planes.

Consider another example. We define the data object x_0 as a list of six elements and then apply the fork `($, #)` which (simultaneously) returns the shape and the number elements.

```
]x0 =: i.6
0 1 2 3 4 5
($;#) x0
+---+
| 6 | 6 |
+---+
```

Here both `#` and `$` return six as x_0 consists of six atoms, or 0-cell elements. Now try this on x_2 which was declared earlier:

```
($;#) x2
+---+
| 2 3 | 2 |
+---+
```

The resultant shape is as expected but the number of elements returned by *tally* may not be. To make sense of the result, we need to know what “elements” the *tally* is counting: 0-cell, 1-cell or 2-cell? This depends upon the rank at which `#` is operating. The data object x_2 is clearly rank two. The command-line below shows us that the (monadic) verb attribute of `#` is infinite:

```
# b. 0 NB. monadic rank attribute is infinite
_ 1 _
```

In this particular case we may ignore the dyadic rank attributes. Tally (`#`) is applied to the 2-cell elements which are the rows. Consider this example:

```
]x1 =: 1 6 $ i.6
0 1 2 3 4 5
($;#) x1
+---+
| 1 6 | 6 |
+---+
```

```
|1 6|1|
+---+--+
```

Data objects x_0 and x_1 may appear the same but they are actually different by virtue of their shape and rank. x_1 is a (6×1) two-dimensional array, and, therefore, of rank two (with shape $1\ 6$). It also has only one element (one row) because $\#$ still operates on the 2-cell elements. In contrast, x_0 is a list of six 0-cell elements. In actual fact, x_0 is equivalent to y_0 , defined below:

```
]y0 =: 6 $ i.6
0 1 2 3 4 5
x0 = y0 NB. x0 and y0 are equivalent
1 1 1 1 1 1
x0 = x1 NB. but x0 and x1, as we know, are not
|length error
| x0 =x1
```

The difference between x_0 and x_1 becomes more apparent when we perform some arithmetic operation on them:

```
x1 - x0
|length error
| x1 -x0
```

The interpreter is trying to subtract the first element from x_1 from the first element of x_0 , then subtract the second element from x_1 from the second element of x_0 , and so on. However, while x_0 has six 0-cell elements, x_1 only has one element, which is a 2-cell. However it does not seem unreasonable to want to perform arithmetic operations on x_0 and x_1 , as they both contain six numbers. We can control the rank attribute of a verb, thereby enabling us to perform arithmetic on both x_0 and x_1 . We will return to this example in Chapter 3 when we cover conjunctions.

2.7 Summary

In this chapter we have introduced some of the basic concepts of programming in J. J functions are called verbs. The primitive verbs are (mostly) designated by a single punctuation character (+) or a pair of punctuation characters (+:), though a few use alphabetic characters (i.). Verbs can be monadic (one argument) or dyadic (two arguments). Data objects have properties of rank (dimension) and shape. All objects can be thought of as arrays. Atoms (0-cell), lists (1-cell) or tables (2-cell) are merely special instances of arrays ranked (dimensioned) zero, one and two, respectively. Furthermore any n ranked array can be thought of as a *list* of $n - 1$ cell objects. Note that, an atom has an empty shape and is different from a one-item list. Furthermore, a list is different from a $1 \times n$ table. Verbs have rank attributes that determine at what *cell* level they operate.



<http://www.springer.com/978-1-84628-822-7>

Network Performance Analysis
Using the J Programming Language

Holt, A.

2008, XVI, 216 p., Hardcover

ISBN: 978-1-84628-822-7