Mastering Dyalog APL

A Complete Introduction to Dyalog APL

Bernard Legrand

With most grateful acknowledgements to the contributors:

Kim S.	Andreasen
Daniel	Baronet
Gitte	Christensen
Peter	Donnelly
Morten	Kromberg
John	Scholes
Adrian	Smith
Tim JA.	Smith

Published by Dyalog Limited

First Edition November 2009

Dyalog Limited

http://www.dyalog.com

ISBN : 978-0-9564638-0-7

Contents

Introduction - Will You Play APL With Me?

Will You Follow Us?

We would like to have you discover a new land, a land where people who may or may not be specialists in programming can process their data, build computerised applications, and take pleasure in using a programming language which is an extremely elegant and powerful tool of thought.

Beware: Dyalog APL is Addictive!

Among the hundreds of programming languages which have been created, most of them share the same fundamentals, the same basic instruction set, approximately the same functions, and by and large the same methods to control the logic of a program. This greatly influences the way people imagine and build solutions to computing problems. Because the languages are so similar, the solutions are similar. Does it mean that these are the only ways of solving problems? Of course not!

Dyalog APL is there to open doors, windows, and minds, prove that original new methods do exist, and that mathematics is not limited to four basic operations. Using APL will expand and extend the range of mental models that you use to solve problems, but beware:

Once you are hooked on APL, there is a real risk that you will no longer accept the limitations of "traditional" programming languages.

Installation and Keyboard

If you do not have access to a computer with Dyalog APL installed, you should still be able to gain an appreciation of the language from these pages and, we hope, enjoy the experience.

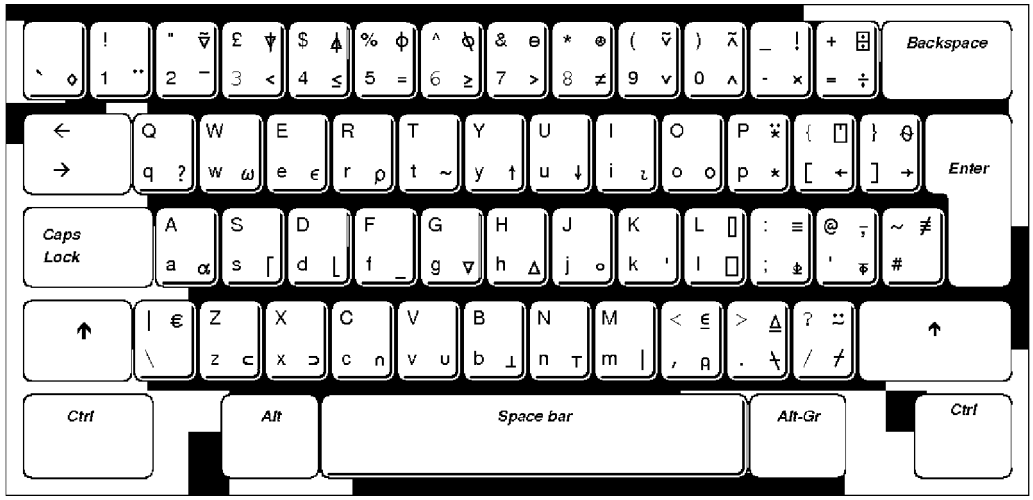
If you have installed Dyalog, not only can you read this book, but you can also experiment on your own computer using the examples below, and invent your own data and calculations.

If you have a copy of Dyalog APL, install it as explained in the User Guide. Just run the installation program and accept all the defaults; there is no need to change anything.

You might like to refer to section A-1 for additional installation hints.

As you will see in the following pages, APL uses special symbols, like ⍬, ⍴, and ↓, which you enter using a special keyboard layout.

You will not need all of the special symbols to read the following pages. The picture below illustrates a cut-down version of the UK keyboard layout, with only the APL symbols that are referred to in this introduction. They are shown on a grey background. The US keyboard is slightly different, but the symbols we need are positioned identically. The full keyboard layout is shown in the User Guide.



Here is how the keyboard is to be used:

- All the standard English letters, numerals and symbols are typed as usual.
- The majority of the APL symbols are obtained by pressing the "*Ctrl*" key, in conjunction with another key. For example, to obtain ⍴, you must press *Ctrl* and R. From now on, this keystroke will be identified as "*Ctrl*+R". On the keyboard layout illustrated above, the symbols that you enter this way are printed at the bottom-right corner of each key.
- Most other APL symbols are obtained by simultaneously pressing "*Ctrl*" and "*Shift*" and then the appropriate key. For example, to obtain ⍬, you must press *Ctrl* and *Shift* and 6. From now on, this keystroke will be identified as "*Ctrl*+*Shift*+6". On the keyboard layout illustrated above, the symbols that you enter this way are printed at the top-right corner of each key.
- A few APL symbols are obtained by pressing the "*Alt Gr*" key (or *Ctrl*+*Alt* if your keyboard does not have an "*Alt Gr*" key), simultaneously with another key, but you don't have to bother about them here: We won't be using them in the examples in this introduction.

- In case you initially have any difficulty with the keyboard, there is a "language bar" on top of the session screen, with all the APL symbols on it. When you need a symbol, you just have to click on it and the symbol will appear wherever your cursor is positioned.

You may notice that some symbols appear twice on the keyboard. This is the case for example for the symbols < = >. These symbols are all part of a normal keyboard (the black ones), but they have been repeated on the APL keyboard, mostly in order to group the symbols used for *comparison functions* together (the red ones). Do not worry: No matter which key you use to produce one of the duplicated symbols, you'll obtain the same result.

Utilities and Data

For most of the examples, you can just type what you read in the following pages, but sometimes you will need some data which we have prepared for you.

This data is contained in a special file (called a WorkSpace) named DyalogTutor_EN.dws which accompanies this book.

If you don't have the file, please refer to section A-1 for instructions on how to download it.

The file can only be opened by Dyalog APL. You can open it as follows:

- In Windows Explorer, double-click on the file's name. APL will be started, and it will then open the file.
- Or, start APL by double clicking on the Dyalog APL icon. Then, using the "File/Open" menu, search for the file and open it.

Once the workspace has been loaded, a welcome message is displayed, and you can check that the workspace contains the variables we shall be using in the following pages:

```
      )vars
Actual Ages AlphLower  AlphUpper  Area  Big  Category etc...
```

You can display the contents of any variable by hovering over its name with the mouse-pointer, by double-clicking on its name, or just by typing its name and pressing the *Enter* key, like this:

```
      Forecast
150 200 100  80  80  80
300 330 360 400 500 520
100 250 350 380 400 450
 50 120 220 300 320 350
```

Now, you are ready, fasten your seatbelts, we're off!

Our First Steps into APL's Magic World

Simple Operations

In APL, what you type starts 6 characters right from the left margin (we say it is "*indented*"), whereas the computer's response begins at the left margin. For additional clarity, in the following pages the characters typed by the user are printed in red, the response given by the computer being in black.

You will notice that in the examples given in the book we very often put a blank space between a symbol and the surrounding names or values. This is in most cases unnecessary; we only do so in order to improve readability. Later on, we will gradually cease to insert the blank spaces in expressions that you should become familiar with along the way.

An expression gets evaluated and the result shown in the session when you press the *Enter* key. Let's try some simple expressions:

```
      27 + 53
80
      1271 - 708
563
      86 ÷ 4              The Divide sign is obtained using  Ctrl+ =
21.5
      59 × 8              The Multiply sign is obtained using  Ctrl+ -
472
```

You can see that APL behaves like any hand-held calculator with, however, a small difference; multiplication is represented by the multiplication symbol (×) which is used in schools in many countries; likewise for division (÷).

In most other computer languages, a star `*` is used for Multiply and `/` for Divide. This is a legacy of the early days of computers, when the character set was limited to the one available on a typewriter. At the time it was decided to use `*` and `/` in place of × and ÷. But it is now possible to display any type of symbol on a screen and on a printer, and this transposition is no longer justifyable. The use of common symbols, which are taught all over the world, aids the understanding of APL by non programmers.

If you are familiar with other programming languages, you may occasionally and erroneously use `*` instead of `×`. Let's see what might happen then:

```
      7 * 3                    ⇦ In APL the star means "Power"
343                              so that 7*3 is equivalent to 7×7×7
```

Variables

As in any programming language, it is possible to create variables. Just choose a name and use the left arrow to assign it a value. In APL a numeric value can consist of a single number, or several numbers separated by at least one blank space. The arrow can be obtained using *Ctrl*+ [

```
      VAT ← 19.6                    ⇦ Read it as: VAT gets 19.6
      Years ← 1952 1943 1986 2007
```

The names are "*case sensitive*". It means that three variables named respectively `VAT`, `Vat`, and `vat`, would be distinct, and may contain different values.

To ask for the contents of a variable, just type its name and press *Enter*, like this:

```
      VAT
19.6
      Years
1952 1943 1986 2007
```

Array Processing

APL is able to operate on two sets of numbers, provided those two sets have the same "shape". For the moment, understand this as "the same number of items". For example, suppose that you have a list of prices of 5 products, and the quantity bought of each:

Prices	5.20	11.50	3.60	4.00	8.45
Quantities	2	1	3	6	2

You can create two variables like this:

```
      Price ← 5.2 11.5 3.6 4 8.45
      Qty   ← 2 1 3 6 2
```

When multiplied together, the variables are multiplied item by item, and produce a result of the same length. That result can be assigned to a new variable.

```
      Costs ← Price × Qty
      Costs
10.4 11.5 10.8 24 16.9
```

This *array processing* capability eliminates most of the "loops" which are common to other programming languages. This remains true even if the data is not a simple list but a multi-dimensional array, of almost any size and number of dimensions.

To make it clear, imagine that a Sales Director makes forecasts for sales of 4 products over the coming 6 months, and assigns them to the variable `Forecast`. At the end of the 6 months, he assigns the real values to the variable `Actual`. Here they are:

```
      Forecast                          Actual
150 200 100  80  80  80          141 188 111  87  82  74
300 330 360 400 500 520          321 306 352 403 497 507
100 250 350 380 400 450          118 283 397 424 411 409
 50 120 220 300 320 350           43  91 187 306 318 363
```

We have not yet explained how you can build such arrays of data, but if you have APL installed, these variables are provided in the Workspace file named "DyalogTutor_EN.dws". Refer to the "*Utilities and Data*" section above to see how you can load the workspace and access the data.

It is clear that the first idea of any Sales Director will be to ask for the differences between what he expected and what he has really got. This can be done easily by typing:

```
      Actual - Forecast
¯9 ¯12  11   7   2  ¯6
21 ¯24  ¯8   3 ¯3 ¯13
18  33  47 44 11 ¯41
¯7 ¯29 ¯33   6 ¯2  13
```

⇦ Note that to distinguish the sign attached to negative values from subtraction, negative values are shown with a high minus sign.

To enter negative values, this high minus sign can be obtained by pressing *Ctrl*+2.

In most traditional programming languages an operation like the one above requires two embedded loops. See what is needed in PASCAL:

```
DO UNTIL I=4
   DO UNTIL J=6
      DIFF(I,J):=ACTUAL(I,J)-FORECAST(I,J)
   END
END.
```

Even if this may seem obvious to a programmer, it is worth noting that most of the code has nothing to do with the user requirement. The only important thing (subtract forecasts from actual values) is hidden behind the detailed workings of the computer program.

To have a calculation done by a machine, one must translate our human wording into something that the computer can understand. With traditional languages, most of that effort is made by the man, to produce a program like the PASCAL example above. The great advantage of APL is that the man has generally much less effort to make, and the machine does the rest.

We have seen that APL will work on two variables of the same shape; it also works if one of the variables is a single item, which is called a *scalar*. If so, the other variable may be of any shape.

For example, if we want to calculate the amount of 19.6% VAT applied to the variable `Price` above, we can type `Price × VAT ÷ 100` (or `VAT × Price ÷ 100` as well), as shown here:

```
      Price × VAT ÷ 100
1.0192 2.254 0.7056 0.784 1.6562   ⇦ This result would require some rounding
                                     but this is not important for now
```

More Symbols

Most programming languages represent only a very small subset of the mathematical functions using symbols (typically +, -, * and /). The creator of APL, Kenneth E. Iverson, chose to include many traditional mathematical symbols in his language, and also added some new symbols to the set that we already know so well.

E.g.: Many functions which in other programming languages are library routines with names like "Maximum" have their own symbols in APL.

The function "*Maximum*" (⌈) returns the greater of two numbers, or of two arrays of numbers compared item by item.

There is also, as one might expect, a symbol for "*Minimum*" (⌊).

```
      75.6 ⌈ 87.3                  ⇦ Maximum (Ctrl+S)
87.3
      11 28 52 14 ⌈ 30 10 50 20    ⇦ Comparison item by item
30 28 52 20
      11 28 52 14 ⌊ 20             ⇦ Minimum (Ctrl+D)
11 20 20 14
```

APL supports about 70 symbols. Since some symbols have more than one meaning one could argue at length about the exact number.

This is nothing to worry about: Some of the symbols are familiar; such as × or > or again ÷ and -, but also ! and a good many others.

Most Symbols Have a Double Meaning

This is not a peculiarity of APL; in algebra we are familiar with the use of symbols as common as the minus sign being used in two different ways.

In the expression	$a = x - y$	the minus sign means subtract
Whereas in	$a = -y$	the minus sign indicates the negation of y, that's different

The first form is called the "*dyadic*" use of the symbol.
The second form is called the "*monadic*" use of the symbol.

It is the same in APL, where most of the symbols can have two meanings.

For example, to find the shape (the dimensions) of an array, one uses the Greek letter Rho (⍴), which can be read "*shape of …*", in its monadic use. It is produced using *Ctrl*+R.

```
      ⍴ Price                          ⇦ Monadic use
5                                        Price has 5 items

      ⍴ Forecast
4 6                                      Forecast has 4 rows of 6 items
```

Used dyadically, the same symbol will organise items into a specified shape. For example, suppose that we want to create the matrix below:

```
25 60
33 47
11 44
53 28
```

We must give the computer two pieces of information:

- First the *shape* to give to the matrix: `4 2` (4 rows of 2 columns)
- Next the *contents* of the matrix: `25 60 33 47 11 44 53 28`

It is the symbol ⍴ (Rho) which makes the connection between the shape and the contents:

```
      Tab ← 4 2 ⍴ 25 60 33 47 11 44 53 28
      Tab
25 60
33 47
11 44
53 28
```

A new variable `Tab` is thereby created, and this is also how the variables `Forecast` and `Actual` above were made.

Conventions

In APL, we give special names to certain shapes of data:

- *Scalar* is used for a single value, a number like `456.18` or a single letter like `'Q'`.
- *Vector* is a plain list of values
  It may be composed of numbers like `Price` and `Qty`,
  or of letters like `'Once upon a time'` within single quotes
- *Matrix* is an array with two dimensions, like `Forecast` or `Tab`
- *Array* is a generic word for any set of values, whatever the number of its dimensions
- *Table* is a common word used for arrays with 2 dimensions (matrices)
- *Cube* is a common word used for arrays with 3 dimensions

Reduction Unifies Traditional Notations

Perhaps you remember the variable `Costs`: `10.4 11.5 10.8 24 16.9`

So what must we do to work out the total? Mathematicians are creative people who long ago devised the symbol $\sum$, always with a pretty collection of indices above and below, which make it complex to understand and to type on a typewriter.

In APL, the operation is written like this:

```
      +/ Costs
73.6
```

Simple isn't it? This gives the total of all the items of the array.

You can read this as "*Plus Reduction*" of the variable `Costs`.

To gain a better understanding of the process:

When we write an instruction such as `+/ 21 45 18 27 11`
- it works as if we had written `21 + 45 + 18 + 27 + 11`
- and we obtain the sum `122`

In fact, it works as if we had "inserted" the symbol + between the values.

But then, if we write `×/ 21 45 18 27 11`
- it is as if we had written `21 × 45 × 18 × 27 × 11`
- so, we get the product `5051970`

Similarly, if we write
- it is as if we had written
- so, we obtain the largest term

```
⌈/ 21 45 18 27 11
21 ⌈ 45 ⌈ 18 ⌈ 27 ⌈ 11
45
```

Reduction, represented by the symbol /, belongs to a special category of symbols called *Operators*. All the other symbols (+ - × ⌈ ⍴ ⍉ ...) are called *Functions* (addition, subtraction, multiplication, maximum, shape, etc.).

The arguments of a function are data (arrays): `Price × Qty`

Whereas at least one of the arguments of an operator is a function: `+/ Qty`

The left argument of *Reduction* can be one of many of the APL symbols, and it can also be the name of a user-defined program. This may give you an idea of the generality and power of the concept.

Dyalog APL contains 10 such powerful operators. If that is not enough, you can even write your own operators, just like you can write your own functions!

Let's Write Our First Programs

Imagine that we want to calculate the average of the following numbers:

```
Val ← 22 37 41 19 54 11 34
```

We must:

- first calculate the sum of the values: `+/ Val` giving `218`
- next calculate the number of values: `⍴ Val` giving `7`
- and finally divide one result by the other

The calculation can be written as the single formula: `(+/Val) ÷ (⍴Val)`

As it is quite likely that we shall often want to make this sort of calculation, it is preferable to store this expression in the form of a program.

In APL we prefer the name *defined function* to the name "program".

Defined functions may be used in the same way as the built-in functions represented by special symbols like + - × ⌈ > ⍴..., which are called *primitive functions*.

To define a simple function like this one, here is the easiest way:

```
Average ← {(+/⍵)÷(⍴⍵)}
```

`Average` is the program name
`⍵` is a generic symbol which represents the array passed on the right.
`⍺` would be the generic symbol for the array passed on the left, if any

The definition of the function is delimited by a set of curly braces { and }. For more complex functions it is also possible to use a text editor, but this is beyond the scope of this short introduction.

Once defined, this function may be invoked in a very simple way:

```
      Average Val                    ⇦ For execution, ⍵ will get the values
31.14285714                            contained in Val
      Average 12 74 56 23
41.25
```

Let us also write two little dyadic functions, the left argument of which is ⍺, and the right is ⍵:

```
      Plus ← {⍺+⍵}
      Times ← {⍺×⍵}
      (3 Plus 6) Times (5 Plus 2)
63
```

As you can see, these functions behave exactly as if we had written `(3+6) × (5+2)`

We said in the preceding section that a user-defined program could be used by the *Reduce* operator; let us try:

```
      Plus/ Val
218                                  ⇦ It works!
```

Indexing

Returning to our vector of numbers `Val`: `22 37 41 19 54 11 34`

In order to extract the 4th item, we just write: `Val[4]`

In many other programming languages one uses parentheses instead of brackets; this is not very different.

What is new is that one can extract several items in one instruction.

```
      Val
22 37 41 19 54 11 34
      Val[2 4 7 1 4]                 ⇦ One may extract the same item twice or more
37 19 34 22 19
```

And of course, in the same way, one may modify one or more items of `Val` using their indexes. Naturally, one must provide as many values as there are items to modify, or a single value for all:

```
      Val[3 5 1] ← 0
      Val
0 37 0 19 0 11 34
      Val[3 5 1] ← 300 77 111
      Val
111 37 300 19 77 11 34    ⇦ You can check that the 3rd item is now 300, the 5th is 77, etc.
```

It is often necessary to extract the first few items from a list of values, for example the first 5. Nothing could be easier:

```
     Val[1 2 3 4 5]
111 37 300 19 77
```

But if one needs to extract the first 500 items from a long list, typing the integers from 1 to 500 would of course be very inconvenient.

This is why APL has been given the symbol `⍳` (*Iota*), which produces the set of the first *n* integers (`⍳` can be obtained using *Ctrl*+I)

Thus, instead of writing `1 2 3 4 5 6 7 8`, it is sufficient to write `⍳8`.

And to extract the first 500 terms of a large vector, one may write: `Big[⍳500]`

We shall discover later an even simpler method.

Calculating Without Writing Programs

The employees of a company are divided into three hierarchical categories, denoted simply 1, 2, and 3. One assigns to two variables the salaries and the categories of these employees; as partly shown here:

```
Salaries   ← 4225 1619 3706 2240 2076 1389 3916 3918 4939 2735 ...
Categories ←    3    1    3    2    2    1    3    3    3    2 ...
```

Do they never want to increase these salaries? (what has our poor world come to!).

A rumour reaches us about their plans: They want a different percentage increase for each category, according to the following scale:

Category	Suggested increase
1	8%
2	5%
3	2%

How much is this going to cost the company?

We create a variable containing the above three rates:

```
      Rates ← 8 5 2 ÷ 100      ⇦ APL allows us to divide three numbers by a single one
      Rates
0.08 0.05 0.02
```

The first employee is in category 3, so the rate that applies to him is:

```
     Rates[3]
0.02
```

It follows that the first 5 employees, being in categories 3 1 3 2 2 respectively, are entitled to the following increases:

```
      Rates[3 1 3 2 2]
0.02 0.08 0.02 0.05 0.05
```

More generally, the rates applied to all of our employees could be obtained like this:

```
      Rates[Categories]
0.02 0.08 0.02 0.05 0.05 0.08 0.02 0.02 0.02 0.05 0.05 0.02 etc.
```

Having the rates, one has just to multiply by the salaries to obtain the individual increases:

```
      Salaries × Rates[Categories]
84.5 129.52 74.12 112 103.8 111.12 78.32 78.36 98.78 136.75 etc.
```

Finally, by adding them all, one will know how much it will cost the company:

```
     +/ Salaries × Rates[Categories]
2177.41
```

You may note that:

- The expression remains valid whatever the number of employees or categories,
- the result has been obtained without writing any program,
- and this expression can be read as the simplest possible English, like this:

Sum the Salaries multiplied by Rates according to Categories

Clever, no?

This illustrates how the expression of a solution in APL can be very close to the way that the solution could be phrased in everyday language. This also shows clearly that the ways of reasoning induced by traditional programming languages are not the only possible ones. This difference and originality, introduced by APL, are among the major features of the language.

Friendly Binary Data

APL makes much use of binary data. It is most often created by means of relational functions like = or >, which give the answer 1 or 0, depending whether the relation is true or not:

```
      Salaries > 3000
1 0 1 0 0 0 1 1 1 0 1 1 0 0 1 1 0 0 0 0
      Actual > Forecast
0 0 1 1 1 0
1 0 0 1 0 0                    ⇦ One can see the favourable results instantly
1 1 1 1 1 0
0 0 0 1 0 1
```

APL offers the conventional mathematical form of the 6 relational functions:

```
< ≤ = ≥ > ≠
```

Naturally one can operate on this binary data using all the functions of Boolean algebra, and moreover, the symbols used are those familiar to mathematicians of all nationalities around the world:

Function *AND*	is represented by the symbol	^	(represented by the word AND in many programming languages)
Function *OR*	is represented by the symbol	∨	(represented by the word OR in these languages)

Thus, if I am looking for people in category 3 whose salary is less than 4000 euros, I can write:

```
      (Categories = 3) ^ (Salaries < 4000)
0 0 1 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 1
```

In fact APL offers all the functions of Boolean algebra, including some perhaps less familiar functions like *NOR* and *NAND* (Not-OR and Not-AND), but they are very useful in finance and electronic automation.

There is, however, no special symbol for the function *Exclusive OR* (often called *XOR*). This is because it is not needed: The function *Not Equal* ≠ gives the same result as *Exclusive OR* when it is used with Boolean values, as you can see below:

```
      0 0 1 1 ≠ 0 1 0 1
0 1 1 0
```

Finally, not only can these binary vectors be used as we have described but also for novel purposes, such as counting and selecting.

Counting

Having found which salaries are less than 2500 euros by means of the following expression:

```
      Salaries < 2500
0 1 0 1 1 1 0 0 0 0 0 0 1 1 0 0 1 0 1 0
```

It is easy to add all the 1s and 0s to calculate how many people earn less than 2500 euros:

```
      +/ Salaries < 2500
8
```

Selection

One can also use the binary vector as a "mask" to select the items corresponding to the binary "1"s from another array:

```
      1 1 0 1 0 0 1 / 23 55 17 46 81 82 83
23 55 46 83
```

The procedure is identical for character data:

```
      1 0 1 0 0 0 0 1 1 / 'Drumstick'
Duck
```

This function, called *Compress*, is particularly useful for extracting the items conforming to a given criterion from a variable. For example, to display the salaries of people in Category 2, one writes:

```
      (Categories = 2) / Salaries
2240 2076 2735 3278 1339 3319      ⇦ Powerful, isn't it?
```

Discovery

To practise our skills some more, let us find in our variable `Val` the positions of numbers greater than 35. Here are the necessary steps:

```
Val       ←      22 37 41 19 54 11 34
Val>35    is      0  1  1  0  1  0  0
⍴Val      is     7
⍳⍴Val     is     1 2 3 4 5 6 7    ⇦ All possible positions
```

Let us compare two of these results

```
Val>35    ⇨      0 1 1 0 1 0 0
⍳⍴Val     ⇨      1 2 3 4 5 6 7
```

You can see that that if you eliminate (using *Compress*) the items which correspond to zeros in order to retain only those corresponding to 1, you easily get the positions required: 2 3 5

Thus the job may be done as follows:

```
      (Val>35) / ⍳⍴Val
2 3 5
```

This expression is applicable in many different situations.

Here is a similar use, but applied to character data: To find the positions of "a" within a phrase; the method is the same.

```
      Phrase ← 'Panama is a canal between Atlantic and Pacific'
      (Phrase = 'a') / ⍳⍴Phrase
2 4 6 11 14 16 30 36 41                ⇦ You can check it!
```

A Touch of Modern Math

Proudly having found all the "a"s, we may wish to find all the vowels.

Alas, although we can write `Phrase = 'a'`, because a vector can be compared with a single value, one cannot write `Phrase = 'aeiouy'`[1], because that would require the item by item comparison of a phrase of 46 letters and "aeiouy" which has only 6.

In other words: You may compare 46 letters with 46 other letters, or compare them with one letter, but not with 6.

So we shall use a new function: ***Membership*** which is represented by the symbol `∈`, also used in mathematics.(`∈` can be obtained by pressing *Ctrl*+E)

The expression `A ∈ B` returns a Boolean result which indicates which items of the variable `A` appear in the variable `B`, wherever they may be. And it works no matter what are the shapes, the dimensions or the type (numeric or character) of `A` and `B`, a pure marvel!

For example:

```
      5 7 2 8 4 9 ∈ 3 4 5 6
1 0 0 0 1 0                        ⇦ Only 5 and 4 are found in 3 4 5 6
      'dandelion' ∈ 'garden'
1 1 1 1 1 0 0 0 1                  ⇦ The letters "lio" do not appear in "garden"
```

[1] "Y" is considered to be a vowel in many European languages.

So in pursuit of our enquiry we shall write:

```
      (Phrase ∈ 'aeiouy') / ⍳⍴Phrase
2 4 6 8 11 14 16 20 23 24 30 33 36 41 43 45
```

One can also use membership between a vector and a matrix, as shown below, assuming that the list of towns is a variable created earlier.

We have represented side by side the variable itself and the result of using *Membership*:

```
      Towns                    Towns ∈ 'aeiouy'
Canberra                   0 1 0 0 1 0 0 1 0 0
Paris                      0 1 0 1 0 0 0 0 0 0
Washington                 0 1 0 0 1 0 0 0 1 0
Moscow                     0 1 0 0 1 0 0 0 0 0
Martigues                  0 1 0 0 1 0 1 1 0 0
Mexico                     0 1 0 1 0 1 0 0 0 0
```

We can reverse the expression, but the result has always the same shape as the left argument:

```
      'aeiouy' ∈ Towns
1 1 1 1 1 0                ⇦ None of the town names contains a "y"
```

A Powerful Search Function

We have harnessed a very useful method to look for the positions of letters or numbers in a vector, but the answer obtained does not provide a one to one correspondence between the search values and the resultant positions:

```
      List ← 15 40 63 18 27 40 33 29 40 88      ⇦ Vector of values
      Where ← 29 63 40 33 50                    ⇦ We want to find these
      (List ∈ Where) / ⍳⍴List                   ⇦ Let's apply our method
2 3 6 7 8 9                                     ⇦ Positions found
```

The positions are correct, but 29 is not in position 2, and 40 is not in position 6.

The question we have answered using the expression above is: "In which positions in `List` do we find a number that also appears somewhere in `Where`?"

If we want to answer the slightly different question: "Where in `List` do we find each number in `Where`?" we need to use a different method.

This new method uses the dyadic form of the symbol ⍳ (*Iota*).

```
      List ← 15 40 63 18 27 40 33 29 40 88      ⇦ Same vector of values
      Where ← 29 63 40 33 50                    ⇦ Where are these?
      List ⍳ Where                              ⇦ New method using dyadic ⍳
8 3 2 7 11                                      ⇦ Positions found
```

It is true that 29, 63, 40 and 33, occur respectively in positions 8, 3, 2 and 7. It's much better!

But, first surprise: The value 40 occurs 3 times in `List`, but only the first one is reported in the result. This is because, by definition, dyadic *Iota* returns only the first occurrence of a given item. If the response for each value sought has to match a position; how may one, looking for 5 numbers, obtain 7 results?

Second surprise: The value 50 is reported as being found in position 11 in a vector comprising only 10 items! This is how the function *IndexOf* (dyadic ⍳) reports that a value is absent.

At first sight this seems a bit surprising, but in fact it is a property which makes this function so generally powerful, as we shall soon see.

An Example

A car manufacturer decides that he will offer his customers a discount on the catalogue price (you can see how this example is imaginary!)

The country has been divided into 100 areas, and the discount rate will depend on the area according to the following table:

Area	Discount
17 50 59 84 89	9 % 8 % 6 % 5 % 4 %
Others	2 %

The problem is to calculate the discount rate that may be claimed for a potential customer who lives in given area `D`; for example `D ← 84`.

Let us begin by creating two variables:

```
Area     ← 17 50 59 84 89
Discount ←  9  8  6  5  4  2
```

Let us see if `84` is in the list of favoured areas:

```
      D ∈ Area
1                                  ⇦ Yes, it's there
      Area ⍳ D
4                                  ⇦ 84 is the 4th item in the list
```