APL Cultivations

Contents

APL Cultivation is the title used for the series of 90-minute live chat lessons given by Adám Brudzewsky in the APL Orchard chat room. The name was first used for lesson 15 at the end of January 2018, but was since applied retroactively to all such lessons.

The first season consisted of 29 weekly sessions running from 18 October 2017 until 16 May 2018, covering most aspects of basic APL programming. Initially, the lessons were not organised, but were given completely impromptu. However, between lessons 2 and 3, Erik Konstantopoulos bookmarked the first two lessons using Stack Exchange's chat conversation bookmarking feature, and thus established the lessons as a numbered series.

Skip to main content

The series continued on 28 November 2019, with more in-depth lessons every two-three weeks. This was sparked by interest among participants of a presentation by Morten Kromberg and Aaron Hsu called Pragmatic Array Oriented Functional Programming, held during Jio talks 2019, after which a series of "APL Hacknights" were to be held in the APL Orchard. However, the audience of the first such event turned out mostly to be people who had *not* been at the Jio talk, and it was decided to fold this new series into a continuation of the previous one. This series ran for 20 sessions until 25 August, 2020.

The following compilation is an attempt to reformat the APL Cultivations into a more accessible format, expand on some of the examples and generally improve the signal-to-noise ratio.

If you find this useful, please consider starring the github repo.

Attribution: APLWiki

Introduction to arrays

The array is APL's fundamental data type. Arrays are collections of scalars (atomic data units). There are a few types of scalars: numbers, characters, and references (refs). References are to such things as namespaces (≈JSON objects), GUI objects (WinForms), HTML Renderers, classes, instances, etc. Let's not worry about all those now.

Characters are denoted by single quotes. `'a'` is a scalar letter `a`. APL doesn't really have strings, just lists (vectors in APL lingo) of characters. In order to write a literal vector (=list) you just write the items next to each other. `'H' 'e' 'y'` will render as `Hey`:

```
'H' 'e' 'y'
```

```
Hey
```

Fortunately, there is a shortcut. APL allows you to write `'Hey'` and it means the same as `'H' 'e' 'y'`:

```
'Hey'
```

Skip to main content

```
Hey
```

So a list of numbers need no decorators whatsoever: `1 2 3`

```
1 2 3
```

```
1 2 3
```

You can also nest items. `'Hey' 'you!'` is a vector of two elements. Each element is itself a vector.

```
'Hey' 'you!'
```

```
 Hey  you!
```

You can also mix data types: `'APL'360` is a two-element vector. The first element is a three-element vector of chars, the second is a scalar number.

```
'APL'360 ⍝ Note: no space required
```

```
 APL  360
```

By the way, in APL, a number is a number. APL converts between internal representations on the fly, so you never have to worry about such conversions. It even takes care of floating point imprecision for you!

`'a'3` is a two element vector. No space needed here, either.

```
'a'3
```

```
a 3
```

You can also use parentheses to delimit vectors:

Skip to main content

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

```
 1 2 3  4 5
```

Question:

> Is there any concept of a “mutable array” in APL?

Nope. You always (appear to) create a new array when modifying an array. However, internally, APL keeps a ref-count and points multiple names to the same memory location if possible. However, all the “reference” types are mutable.

The levels of nesting in APL lingo are called depth. A simple scalar has depth 0. A vector has depth 1. A vector of vectors has depth 2, etc. If the depth is uneven, then we report it as negative. Note that negative numbers in APL are denoted by a high minus (like TI calculators).

You can have 1-element vectors, but you have to “create” them rather than write them. The prefix function `,` (comma) takes an array and makes it into a list. So `,6` is a one-element list.

```
]display ,6        ⍝ Verbose display to demonstrate that ,6 is indeed a vector
]display (,6)1 2
]display (,6)(1 2)
```

```
┌→┐
│6│
└~┘
```

```
┌→────────┐
│ ┌→┐     │
│ │6│ 1 2 │
│ └~┘     │
└∊────────┘
```

```
┌→──────────┐
│ ┌→┐ ┌→──┐ │
│ │6│ │1 2│ │
│ └~┘ └~──┘ │
└∊──────────┘
```

Skip to main content

APL also has a concept of rank. The rank of an array is the number of dimensions in that array. A scalar has rank 0, a vector has rank 1.

However, we can also have a rank 2 array; a matrix, or table. Note that rank ≠ depth. So I can have a matrix where every element is a “string” (i.e. a vector). I can also have a vector of vectors of “strings”.

Rank is always flush. Every row in a matrix must have the same number of columns. Every layer in a 3D block of data must have the same number of rows and columns.

Each APL implementation has a different max number of dimensions. Dyalog allows 15D arrays. If that isn’t enough for you, you may be doing something not quite right. J, which is a dialect of APL (and the mother of Jelly) allows for an unlimited (except by memory) number of dimensions.

Imagine a piece of paper with a grid of letters. So we have rows and columns. Each paper is a page in a book. Each book is numbered on its shelf. The shelves are numbered. There are multiple bookcases next to each other. And there are several such corridors. In rooms next to each other. Each floor has multiple numbered corridors, etc.

The infix function reshape, `⍴` (Greek letter “rho” for reshape), takes a list of dimension lengths as left argument and any data as right argument. It returns a new array with the specified dimensions, filled with the data. If there is too much data, the tail just doesn’t get used. If there is too little, it gets recycled from the beginning.

We can create a 3-row, 4-column table with

```
3 4⍴'abcdefghijkl'
```

```
abcd
efgh
ijkl
```

```
 3 4⍴'abc'     ⍝ insufficient data; keep recycling
```

```
abca
bcab
```

Skip to main content

Most primitive APL functions have both a monadic (one argument) and a dyadic (two arguments) form. It is always clear from context which one is being applied, as all monadic functions are prefix, and all dyadic ones are infix. We already addressed the dyadic `⍴` which was *Reshape*. The monadic `⍴` is Shape. It reports back what the shape is.

```
3 3⍴⎕A
⍴3 3⍴⎕A   ⍝ What is the shape of a 3x3 matrix?
]display ⍴1 2 3 4  ⍝ What is the shape of a vector?
]display ⍴6  ⍝ What is the shape of a scalar?
```

```
ABC
DEF
GHI
```

```
3 3
```

```
┌→┐
│4│
└~┘
```

```
┌⊖┐
│0│
└~┘
```

Note that the shape is always a vector. The shape of a scalar is the empty numeric vector, denoted `⍬`.

Monadic `↑` is mix, which ups the rank (at the cost of one level of depth). We can also lower the rank with split, `↓`, and thereby gain a level of depth.

```
(1 2 3)(4 5 6)(7 8 9)  ⍝ vector
↑(1 2 3)(4 5 6)(7 8 9) ⍝ mix vector to a matrix
3 4⍴⍳12                ⍝ matrix
↓3 4⍴⍳12               ⍝ split matrix to a vector
```

Skip to main content

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

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

```
1  2  3  4
5  6  7  8
9 10 11 12
```

```
  1 2 3 4  5 6 7 8  9 10 11 12
```

There is no primitive for rank, because if you think about it, the rank is the shape (actually, the tally) of the shape. There is, however, a primitive for depth: `≡`

```
≡(1 2 3)(4 5 6)(7 8 9) ⍝ vector of vectors
```

```
2
```

There is a different primitive for count (called tally): `≢` – it looks like a tallying mark.

```
≢7 5 6 3 2
```

```
5
```

Question:

> so...what *is* the difference between rank and depth?

This is important to understand. Depth is the level of nesting. Rank is the number of dimensions.

So now we have discovered monadic `↑`, `↓`, `≡`, `≢`, `⍴` and dyadic `⍴`. Monadic `⍴` always returns a vector. Monadic `≢` always returns a scalar. `≢` on a matrix returns the number of rows. `≢` on a 3D block returns the number of layers, etc. We prefer to call it the tally of "major cells". The concept of major cells is important when it comes to manipulating and comparing arrays

Skip to main content

We already saw how dyadic `ρ` can reshape things. Dyadic `↑` is take. In order to speak about its two arguments easier, we will give them names. The left argument we will call `⍺` as in the leftmost letter of the Greek alphabet, and the right argument we will call `⍵` as in the rightmost letter. In other words, `↑⍵` is monadic `↑` and `⍺↑⍵` is dyadic `↑`.

`⍺↑⍵` takes the `⍺` first major cells from `⍵`:

```
3↑3 1 4 1 5
```

```
3 1 4
```

We can take major cells from the *end* of `⍵` by using a negative `⍺`:

```
¯3↑3 1 4 1 5
```

```
4 1 5
```

APL arrays have something called prototype. The prototype for numbers is 0 and the prototype for chars is a space. The prototype for a mixed-type array is the first element's prototype. More generally, for an array of arrays, the prototype is the first element, but with all numbers made 0 and all chars made spaces. If you take more than there is, APL will pad with this prototype element:

```
10↑1 2 3
¯10↑1 2 3
]display 10↑'Hello'
```

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

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

```
┌→─────────┐
│Hello     │
└──────────┘
```

Skip to main content

Primitive functions

A *primitive function* is a function defined by the language. Outside of the array community, such functions may be called “builtin” or “intrinsic” functions. In APL, each is represented with a single glyph; in other languages, such as those restricted to ASCII characters, they may use multiple characters (“bigraphs” and “trigraphs” are combinations of two and three characters, respectively). Other parts of APL which are written with a single glyph include primitive operators and Quad.

A function is distinct from the glyph used to denote it. Different APLs, or even one APL (using migration level) might use the same glyph for multiple functions, or different glyphs for identical or similar functions. The term “function” can, depending on context, refer either to an ambivalent function which can be applied with one or two arguments, or the monadic or dyadic function obtained by restricting that function to either one or two arguments specifically.

Attribution: APLWiki

`+-×÷*⍟⌹○`

Arithmetic `+-×÷`

Dyadic `+-×÷` are what you expect from mathematics:

```
3+8
4×12
144×11
3-7
```

```
11
```

```
48
```

```
1584
```

Skip to main content

`0÷0` is `1` by default, but you can make all `n÷0` into `0` by setting `⎕DIV←1`:

```
0÷0
```

```
1
```

```
⎕DIV←1
0÷0
⎕DIV←0       ⍝ default setting
```

```
0
```

Reciprocal `÷A`

Question:

> How can we make 0÷0 throw an error?

Multiply with the reciprocal:

```
0×÷0        ⍝ DOMAIN ERROR: Divide by zero
```

```
DOMAIN ERROR: Divide by zero
      0×÷0        ⍝ DOMAIN ERROR: Divide by zero
        ^
```

Monadic `÷` is the reciprocal, i.e. `÷x` is `1÷x`.

Direction `×A`

Monadic `×` is direction, i.e. a complex number which has magnitude 1 but same angle as the argument. For real numbers this means signum (sign).

Skip to main content

```
÷5          ⍝ reciprocal: 1÷5
×12 ¯33 0   ⍝ signum
×32j¯24     ⍝ direction
```

```
0.2
```

```
1 ¯1 0
```

```
0.8J¯0.6
```

Power `*`

Dyadic `*` is power, and the default left argument (i.e. for the monadic form) is e. So, monadic `*` is e-to-the-power-of.

```
2*10        ⍝ ⍺ to the power of ⍵
*1          ⍝ e to the power of ⍵
```

```
1024
```

```
2.718281828
```

Log `⍟`

The inverse of `*` is `⍟`; logarithm. The monadic form is the natural logarithm and the dyadic is left-arg logarithm, so `10⍟n` is `log(n)`:

```
10⍟10000000   ⍝ log(10000000)
```

```
7
```

Skip to main content

Matrix divide `⌹`

`⌹` is matrix division. Give it a coefficients' matrix on the right and it will invert the matrix. If you also put a vector on the left and it will solve your system of equations. If over-determined, it will give you the least squares fit.

For example, in order to solve the following set of simultaneous equations,

$$\begin{aligned} 3x + 2y &= 13 \\ x - y &= 1 \end{aligned}$$

we can use `⌹` like so:

```
13 1 ⌹ 2 2⍴3 2 1 ¯1
```

```
3 2
```

Circular `○`

Monadic `○` multiplies by π:

```
○2          ⍝ 2 times π
```

```
6.283185307
```

Dyadic `○` is circular. It uses an integer left argument to select which trigonometric function to apply. The most common ones are 1, 2 and 3, which are *sin*, *cos* and *tan*. The negative versions `¯1`, `¯2` and `¯3` are *arcsin*, *arccos* and *arctan*.

```
1○○1        ⍝ sin π
2○○1        ⍝ cos π
¯2○2○○1     ⍝ arccos cos π
```

Skip to main content

```
1.224646799E¯16
```

```
¯1
```

```
3.141592654
```

The entire list of `○`'s left arguments is here.

`! ? | ⌈ ⌊ ⊥ ⊤ ⊣ ⊢`

Factorial, binomial `!`

Monadic `!` is factorial. Note that it goes on the left (like all other monadic APL functions) as opposed to mathematics' !.

Dyadic `A!B` is binomial. It is the number of ways to take `A` items from a bag of `B` items, generalised to be the binomial function.

```
!12         ⍝ 12 factorial
2!8         ⍝ how many ways can we select 2 from 8?
```

```
479001600
```

```
28
```

Roll, deal `?`

Monadic `?B` is roll. It returns a random integer among the first `B` integers. `?0` returns a random float between (but not including) 0 and 1:

```
?6 6 6     ⍝ roll three six-sided dice
?0         ⍝ random float between 0-1, excluding 0 and 1
```

Skip to main content

```
4 3 2
```

```
0.04706912049
```

Dyadic `A?B` is deal. It returns a random one of the ways `A!B` counted. I.e. it returns `A` random numbers among the `B` first integers.

```
10?10        ⍝ 1-10 in random order
```

```
4 5 6 10 1 3 2 8 9 7
```

Note that it deals from the set `⍳B`, so it's dependent on your `⎕IO` setting:

```
10?10 ⊣ ⎕IO←0      ⍝ Now we should get 0-9
⎕IO←1              ⍝ Reset ⎕IO to default
```

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

Magnitude, residue `|`

Monadic `|` is magnitude, also called the absolute value, $|x|$:

```
|¯97
|3 5 ¯7 ¯8 7 ¯2
```

```
97
```

```
3 5 7 8 7 2
```

Dyadic `A|B` is residue, also known as the *division remainder* ("mod") when `B` is divided by `A`. Note the reversed order of arguments. "normal" mod is `|⍨`.

Skip to main content

```
1 0 1 0 1 0 1 0 1 0
```

Ceiling, maximum `⌈`

Monadic `⌈` is ceiling, $\lceil x \rceil$,

```
⌈3.14159256
```

```
4
```

Dyadic `A⌈B` is maximum:

```
15⌈23
```

```
23
```

Floor, minimum `⌊`

Monadic `⌊` is floor, and the dyadic is minimum,

```
⌊3.14159256
15⌊23
```

```
3
```

```
15
```

Decode `⊥`

`A⊥B` is decode. It evaluates digits `B` as (mixed) base `A`, e.g,

```
2⊥1 0 1 0 1 0   ⍝ decode binary to decimal
```

```
42
```

Encode `⊤`

`A⊤B`, or [encode](), is the inverse of `⊥`, turning `B` into a list(s) of digits in (mixed) base `A`,

```
24 60 60⊤10000  ⍝ seconds to hour, minutes, seconds
```

```
2 46 40
```

Ten thousand seconds is the same as 2 hours, 46 minutes and 40 seconds.

Left, right `⊣⊢`

Dyadic `⊣` is the left argument unmodified. Monadically, it just returns its sole argument. Dyadic `⊢` is the right argument unmodified. Monadically, it just returns its sole argument.

`=≤<>≥≡≢`

Comparisons `=≤<>≥≡≢`

`=` is comparison (not assignment!) and penetrates all structures, giving a single Boolean (0 or 1) per leaf element. `≠` is the negation of that.

`≤<>≥` work as you'd expect, again penetrating all structure.

`A≡B` is match. It compares the entire arrays `A` and `B` in all respects, even the invisible prototype:

```
''≡⍬   ⍝ does the empty char vector match the empty numeric vector?
```

Skip to main content

```
0
```

`A≢B` is not match, the negation of `A≡B`.

Depth, tally `≡≢`

Monadic `≡B` gives the depth of `B`, which is the amount of nesting. A simple scalar is 0, a vector is 1, a vector of vectors is 2, etc. If the amount of nesting is uneven throughout the array, the result will be negative, and indicate the maximum depth.

`≢B` is the tally of `B`, i.e. how many major cells `B` has. For a scalar, that’s 1. For a vector, it is the number of elements, for a matrix it is the number of rows, for a 3D array it is the number of layers, and so on.

```
≡(1 2 (3 4 5 (6 7 8)))  ⍝ unevenly nested vector
≢1                      ⍝ scalars tally to 1
≢3 2⍴⍳6                 ⍝ matrix tally is the number of rows
```

```
¯3
```

```
1
```

```
3
```

`∨∧⍱⍲↑↓`

OR, GCD `∨`

`∨` is logical OR, and it is Greatest Common Divisor for for other numbers (which happens to fit with *OR* for 0s and 1s):

```
0 1 0 1 ∨ 0 0 1 1    ⍝ logical OR
15 1 2 7 ∨ 35 1 4 0  ⍝ GCD
```

Skip to main content

```
0 1 1 1
```

```
5 1 2 7
```

AND, LCD `∧`

`∧` is logical AND, and it is Lowest Common Multiple for for other numbers (which happens to fit with *AND* for 0s and 1s):

```
0 1 0 1 ∧ 0 0 1 1      # logical AND
15 1 2 7 ∧ 35 1 4 0    # LCM
```

```
0 0 0 1
```

```
105 1 4 0
```

NOR, NAND `∨̃∧̃`

`∨̃` is NOR, and `∧̃` is NAND. They only work on Booleans (arrays with nothing but 1s and 0s). Note that you can use `≠` as *XOR* and `=` as *XNOR* (and you can use `≤` as logical implication. Similarly for the other comparisons.)

```
0 1 0 1 ∨̃ 0 0 1 1 # NOR
0 1 0 1 ≠ 0 0 1 1 # XOR
0 1 0 1 = 0 0 1 1 # XNOR
0 1 0 1 ∧̃ 0 0 1 1 # NAND
```

```
1 0 0 0
```

```
0 1 1 0
```

```
1 0 0 1
```

```
1 1 1 0
```

Skip to main content

Take `↑`

`A↑B` takes from `B`. If `A` is a scalar/one-element-vector, it takes major cells, if it has two two elements, the first element is the number of major cells, and the second the number of semi-major cells, etc.:

```
3 4⍴⎕A              ⍝ original array
2↑3 4⍴⎕A            ⍝ take two major cells (a.k.a rows)
2 3↑3 4⍴⎕A          ⍝ two major, and three semi-major cells
```

```
ABCD
EFGH
IJKL
```

```
ABCD
EFGH
```

```
ABC
EFG
```

If you take more than there is, `↑` will pad with 0s for numeric arguments, and spaces for character arguments:

```
6↑3 1 4
```

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

You may also “overtake” a scalar to any number of dimensions:

```
2 3↑4
```

```
4 0 0
0 0 0
```

Skip to main content

```
¯6↑3 1 4
¯2 ¯3↑4
```

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

```
0 0 0
0 0 4
```

```
3 4⍴⎕A
¯2 ¯2↑3 4⍴⎕A
```

```
ABCD
EFGH
IJKL
```

```
GH
KL
```

Mix `↑`

Monadic `↑` is mix. It trades one level of depth (nesting) into one level of rank.

```
↑(1 2 3)(4 5 6)
```

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

Because rank enforces non-raggedness, monadic `↑` will pad with the prototype element (0 or space) just like dyadic `↑`:

```
↑(1 2 3)(4 5)
```

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

Skip to main content

Drop `↓`

Dyadic `↓` is just like dyadic `↑` except it drops instead of taking:

```
3 4⍴⎕A
1↓3 4⍴⎕A
]display 2 1↓3 4⍴⎕A
```

```
ABCD
EFGH
IJKL
```

```
EFGH
IJKL
```

```
┌→──┐
↓JKL│
└───┘
```

Note that the last result is still a matrix, it just only has one row.

Split `↓`

Monadic `↓` is split. It is the opposite of dyadic `↓` in that it lowers the rank and increases the depth:

```
↓3 4⍴⎕A
```

```
 ABCD  EFGH  IJKL
```

Skip to main content

Enclose `⊂`

Monadic `⊂` encloses its argument. It packages an arbitrary structure into a scalar. Simple scalars cannot be enclosed. We can turn on boxed output with the `]box` user command to illustrate APL's array structure in more detail:

```
]box on -s=max
```

```
┌→─────────────────┐
│Was OFF -style=max│
└──────────────────┘
```

```
v←1 2 3 4
v
⊂v
```

```
┌→──────┐
│1 2 3 4│
└~──────┘
```

```
┌───────────┐
│ ┌→──────┐ │
│ │1 2 3 4│ │
│ └~──────┘ │
└∊──────────┘
```

The little epsilon `∊` in the bottom of the outer box indicates the enclosure.

If we tally an enclosed structure, it should come out as 1:

```
≢⊂v   ⍝ an enclosed vector is a scalar
```

```
1
```

Here’s another example:

```
(3 3⍴⎕A),(3 3⍴⎕A)   ⍝ concatenation of two matrices.
(⊂3 3⍴⎕A),(⊂3 3⍴⎕A) ⍝ concatenation of two enclosed matrices
```

```
┌→─────┐
↓ABCABC│
│DEFDEF│
│GHIGHI│
└──────┘
```

```
┌→────────────┐
│ ┌→──┐ ┌→──┐ │
│ ↓ABC│ ↓ABC│ │
│ │DEF│ │DEF│ │
│ │GHI│ │GHI│ │
│ └───┘ └───┘ │
└∊────────────┘
```

The first gave us a matrix of shape 3 6, the second gave a vector of shape 2.

```
(3 3⍴⎕A),(⊂3 3⍴⎕A) ⍝ concatenation of a matrix and an enclosed matrix
```

```
┌→────────────┐
↓       ┌→──┐ │
│ A B C ↓ABC│ │
│ - - - │DEF│ │
│       │GHI│ │
│       └───┘ │
│       ┌→──┐ │
│ D E F ↓ABC│ │
│ - - - │DEF│ │
│       │GHI│ │
│       └───┘ │
│       ┌→──┐ │
│ G H I ↓ABC│ │
│ - - - │DEF│ │
│       │GHI│ │
│       └───┘ │
└∊────────────┘
```

Skip to main content

Concatenating a scalar to a matrix uses the scalar for each row. Here the entire right-hand matrix was treated as a scalar because it was enclosed.

```
(3 3⍴⎕A),'x'
```

```
┌→───┐
↓ABCx│
│DEFx│
│GHIx│
└────┘
```

So you can (and should) use `⊂` to tell APL how you want the scalar extension (auto-vectorisation) to be applied.

`⊂` is also good for treating text vectors as strings (i.e. in one piece):

```
'aaa' 'bbb' 'ccc' ⍳ 'aaa'
```

```
┌→────┐
│4 4 4│
└~────┘
```

This says that each one of the three right-side 'a's is found in position 4 (i.e. are not) in the left-side list.

```
'aaa' 'bbb' 'ccc' ⍳ ⊂'aaa'
```

```
1
```

This says that 'aaa' is the first string.

Partitioned enclose `⊂`

Dyadic is partitioned enclose. It encloses (hence sharing the symbol) pieces of the right

Skip to main content

```
1 0 0 1 0 1 0 0 0 1 0⊂'Hello World'
```

```
┌→─────────────────────────┐
│ ┌→──┐ ┌→─┐ ┌→───┐ ┌→─┐ │
│ │Hel│ │lo│ │ Wor│ │ld│ │
│ └───┘ └──┘ └────┘ └──┘ │
└∊─────────────────────────┘
```

This works on higher rank arrays, too. It partitions along the last axis:

```
1 0 1 0 0 1 0 1 0 0 0 1 1 ⊂ 2 13⍴⎕A
```

```
┌→─────────────────────────────────────┐
│ ┌→─┐ ┌→──┐ ┌→─┐ ┌→───┐ ┌→┐ ┌→┐ │
│ ↓AB│ ↓CDE│ ↓FG│ ↓HIJK│ ↓L│ ↓M│ │
│ │NO│ │PQR│ │ST│ │UVWX│ │Y│ │Z│ │
│ └──┘ └───┘ └──┘ └────┘ └─┘ └─┘ │
└∊─────────────────────────────────────┘
```

For vectors, `1⊂` is the same as `,¨` which may be useful in trains where you want to have a left argument. For higher rank arrays, `1⊂` cuts into columns:

```
1 ⊂ 2 13⍴⎕A
```

```
┌→────────────────────────────────────────────────────────────┐
│ ┌→┐ ┌→┐ ┌→┐ ┌→┐ ┌→┐ ┌→┐ ┌→┐ ┌→┐ ┌→┐ ┌→┐ ┌→┐ ┌→┐ ┌→┐ │
│ ↓A│ ↓B│ ↓C│ ↓D│ ↓E│ ↓F│ ↓G│ ↓H│ ↓I│ ↓J│ ↓K│ ↓L│ ↓M│ │
│ │N│ │O│ │P│ │Q│ │R│ │S│ │T│ │U│ │V│ │W│ │X│ │Y│ │Z│ │
│ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ └─┘ │
└∊────────────────────────────────────────────────────────────┘
```

You can use brackets to indicate which axis you wish to cut along:

```
1 ⊂[1] 2 13⍴⎕A
1 0 1 1 ⊂[1] 4 3⍴⎕A
```

Skip to main content

```
┌→────────────────────────────────────┐
│ ┌→────────────┐ ┌→────────────┐ │
│ ↓ABCDEFGHIJKLM│ ↓NOPQRSTUVWXYZ│ │
│ └─────────────┘ └─────────────┘ │
└∊────────────────────────────────────┘
```

```
┌→────────────────────┐
│ ┌→──┐ ┌→──┐ ┌→──┐ │
│ ↓ABC│ ↓GHI│ ↓JKL│ │
│ │DEF│ └───┘ └───┘ │
│ └───┘             │
└∊────────────────────┘
```

Note that the left argument need not be the same length as the right argument. If it’s shorter, it’s assumed to consist of zeros to the end:

```
⍳⍣¯1⊢1 5 10
(⍳⍣¯1⊢1 5 10)⊂⎕A ⍝ note: left arg is length 10. right arg is length 26
```

```
┌→──────────────────┐
│1 0 0 0 1 0 0 0 0 1│
└~──────────────────┘
```

```
┌→──────────────────────────────────────┐
│ ┌→───┐ ┌→────┐ ┌→─────────────────┐ │
│ │ABCD│ │EFGHI│ │JKLMNOPQRSTUVWXYZ│ │
│ └────┘ └─────┘ └──────────────────┘ │
└∊──────────────────────────────────────┘
```

Another common use of dyadic `⊂` is to split a vector into its head and tail:

```
1 1⊂1 2 3 4 5 6 7
```

```
┌→────────────────────┐
│ ┌→┐ ┌→──────────┐ │
│ │1│ │2 3 4 5 6 7│ │
│ └~┘ └~──────────┘ │
└∊────────────────────┘
```

The left argument does not have to be a Boolean vector, but can in fact be any simple numeric

Skip to main content

```
1 0 1 0 1 0 1 0 ⊆ 'ABCDEFGH'
1 0 2 0 5 0 2 0 ⊆ 'ABCDEFGH'
1 0 0 2 2 0 0 0 0 0 0 ⊆ 'KenEIverson'
```

```
┌→────────────────────┐
│ ┌→─┐ ┌→─┐ ┌→─┐ ┌→─┐ │
│ │AB│ │CD│ │EF│ │GH│ │
│ └──┘ └──┘ └──┘ └──┘ │
└∊────────────────────┘
```

```
┌→─────────────────────────────────────────────┐
│ ┌→─┐ ┌⊖┐ ┌→─┐ ┌⊖┐ ┌⊖┐ ┌⊖┐ ┌⊖┐ ┌→─┐ ┌⊖┐ ┌→─┐ │
│ │AB│ │ │ │CD│ │ │ │ │ │ │ │ │ │EF│ │ │ │GH│ │
│ └──┘ └─┘ └──┘ └─┘ └─┘ └─┘ └─┘ └──┘ └─┘ └──┘ │
└∊─────────────────────────────────────────────┘
```

```
┌→─────────────────────────────┐
│ ┌→──┐ ┌⊖┐ ┌→┐ ┌⊖┐ ┌→──────┐ │
│ │Ken│ │ │ │E│ │ │ │Iverson│ │
│ └───┘ └─┘ └─┘ └─┘ └───────┘ │
└∊─────────────────────────────┘
```

Here's a practical example. Let's say we have some sorted data, and we'll like to group it by interval.

```
values  ← 3 14 15 35 65 89 92
cutoffs ← 0 20 40 60 80 100
```

We want to end up with `(3 14 15) (,35) ⍬ (,65) (89 92)`. That is, all the numbers in the interval `[0,20)` and in `[20,40)` etc. To get the index of each value's interval, we begin by applying Interval Index:

```
cutoffs ⍸ values
```

```
┌→────────────┐
│1 1 1 2 4 5 5│
└~────────────┘
```

Now, you might think that Key `⌸` could do the trick, but then we'd have to insert all the possible intervals.

Skip to main content

```
(cutoffs⍸values) {⊂1↓⍵}⌸⍥{⍵,⍨⍳≢cutoffs} values
```

```
┌→────────────────────────────────────┐
│ ┌→──────┐ ┌→─┐ ┌⊖┐ ┌→─┐ ┌→────┐ ┌⊖┐ │
│ │3 14 15│ │35│ │0│ │65│ │89 92│ │0│ │
│ └~──────┘ └~─┘ └~┘ └~─┘ └~────┘ └~┘ │
└∊────────────────────────────────────┘
```

However, using partitioned enclose with a non-Boolean left argument, we can craft a much more elegant solution:

```
(1,2-⍨/cutoffs⍸values)⊂values
```

```
┌→────────────────────────────────┐
│ ┌→──────┐ ┌→─┐ ┌⊖┐ ┌→─┐ ┌→────┐ │
│ │3 14 15│ │35│ │0│ │65│ │89 92│ │
│ └~──────┘ └~─┘ └~┘ └~─┘ └~────┘ │
└∊────────────────────────────────┘
```

or, as a train,

```
cutoffs (⊢⊂⍨1,2-⍨/⍸) values
```

```
┌→────────────────────────────────┐
│ ┌→──────┐ ┌→─┐ ┌⊖┐ ┌→─┐ ┌→────┐ │
│ │3 14 15│ │35│ │0│ │65│ │89 92│ │
│ └~──────┘ └~─┘ └~┘ └~─┘ └~────┘ │
└∊────────────────────────────────┘
```

Here's another handy trick. Let's say we want to split a vector at a given set of indices, in other words,

```
mask ← ⎕ ←'KenEIverson'∊⎕A
mask ⊂ 'KenEIverson'
```

Skip to main content

```
┌→────────────────────┐
│1 0 0 1 1 0 0 0 0 0 0│
└~────────────────────┘
```

```
┌→────────────────────┐
│ ┌→──┐ ┌→┐ ┌→──────┐ │
│ │Ken│ │E│ │Iverson│ │
│ └───┘ └─┘ └───────┘ │
└∊────────────────────┘
```

…but if instead of the mask, we started with the indices of the 1s: 1 4 5?

```
(⍸⍣¯1⊢1 4 5)⊂'KenEIverson' ⍝ ⍸ has an inverse!
```

```
┌→────────────────────┐
│ ┌→──┐ ┌→┐ ┌→──────┐ │
│ │Ken│ │E│ │Iverson│ │
│ └───┘ └─┘ └───────┘ │
└∊────────────────────┘
```

Anyway, this shows another extension introduced in 18.0, namely that `⍸⍣¯1` conveniently works, but what you might not notice is that it also shows a further extension of `⊂`. Observe:

```
(⎕←⍸⍣¯1⊢1 4 5)⊂'KenEIverson'
```

```
┌→────────┐
│1 0 0 1 1│
└~────────┘
┌→────────────────────┐
│ ┌→──┐ ┌→┐ ┌→──────┐ │
│ │Ken│ │E│ │Iverson│ │
│ └───┘ └─┘ └───────┘ │
└∊────────────────────┘
```

Note that the length of `⍸⍣¯1⊢1 4 5` doesn't match the length of the string. Until 17.1, the arguments of `⊂` had to have exactly the same length. Now, the left argument can have any length up until 1+the length of the right argument, to allow some empty partitions at the end.

```
1 0 0 1 1 2⊂'Hello'
```

Skip to main content

```
┌→──────────────────────┐
│ ┌→──┐ ┌→┐ ┌→┐ ┌⊖┐ ┌⊖┐ │
│ │Hel│ │l│ │o│ │ │ │ │ │
│ └───┘ └─┘ └─┘ └─┘ └─┘ │
└∊──────────────────────┘
```

So `⊂` will assume that any “missing” elements in its left argument are 0.

Disclose `⊃`

Monadic `⊃` is disclose, which is pretty much the inverse of monadic `⊂`. It discloses a scalar (again, if possible; a simple scalar remains the same). If you use it on a high rank array (i.e. not enclosed), it will give you the first (top left) element:

```
3 3⍴⎕A      ⍝ 3x3 matrix
⊂3 3⍴⎕A     ⍝ enclosed
⊃⊂3 3⍴⎕A    ⍝ disclose enclosed
⊃3 3⍴⎕A     ⍝ dislclose unenclosed gives the first element
```

```
┌→──┐
↓ABC│
│DEF│
│GHI│
└───┘
```

```
┌───────┐
│ ┌→──┐ │
│ ↓ABC│ │
│ │DEF│ │
│ │GHI│ │
│ └───┘ │
└∊──────┘
```

```
┌→──┐
↓ABC│
│DEF│
│GHI│
└───┘
```

```
A
-
```

Skip to main content

The last feature (“first”) means that you can combine it with reverses etc, to get corner elements:

```
⊃⌽3 3⍴⎕A ⍝ top right
⊃⊖3 3⍴⎕A ⍝ bottom left
```

```
C
-
```

```
G
-
```

You can use it with `¨` (each) to get initials:

```
⊃¨'Kenneth' 'Eugene' 'Iverson'
```

```
┌→──┐
│KEI│
└───┘
```

Pick `⊃`

Dyadic `⊃` is pick. It digs into nested arrays. Every scalar on its left is the index of an element in subsequent layers of nestedness:

```
(⊂2 3)⊃3 3⍴⎕A
2 3 1⊃(1 2 3)(4 5 (6 7 8))
```

```
F
-
```

```
6
```

Skip to main content

```
2 2⍴(1 2)(3 4)(5 6)(7 8)
(1 2) 2⊃2 2⍴(1 2)(3 4)(5 6)(7 8)
```

```
┌→────────────┐
↓ ┌→──┐ ┌→──┐ │
│ │1 2│ │3 4│ │
│ └~──┘ └~──┘ │
│ ┌→──┐ ┌→──┐ │
│ │5 6│ │7 8│ │
│ └~──┘ └~──┘ │
└∊────────────┘
```

```
4
```

In the last statement, the first index is 1 2, which picks the element (3 4), and the second index is 2, which picks the 4.

Nest `⊆`

Monadic `⊆` is called nest because it guarantees you that the result is nested (non-simple). `(1 2)(3 4 5)` is already nested, and `⊆` won't do anything:

```
(1 2)(3 4 5)
⊆(1 2)(3 4 5)
```

```
┌→──────────────┐
│ ┌→──┐ ┌→────┐ │
│ │1 2│ │3 4 5│ │
│ └~──┘ └~────┘ │
└∊──────────────┘
```

```
┌→──────────────┐
│ ┌→──┐ ┌→────┐ │
│ │1 2│ │3 4 5│ │
│ └~──┘ └~────┘ │
└∊──────────────┘
```

`1 2 3` is not nested, so `⊆` will nest it:

Skip to main content

```
1 2 3
≤1 2 3
```

```
┌→────┐
│1 2 3│
└~────┘
```

```
┌─────────┐
│ ┌→────┐ │
│ │1 2 3│ │
│ └~────┘ │
└∊────────┘
```

Works on higher rank too, of course:

```
2 3ρ⊂'abc'
≤2 3ρ⊂'abc' ⍝ already nested, so no-op
```

```
┌→──────────────────┐
↓ ┌→──┐ ┌→──┐ ┌→──┐ │
│ │abc│ │abc│ │abc│ │
│ └───┘ └───┘ └───┘ │
│ ┌→──┐ ┌→──┐ ┌→──┐ │
│ │abc│ │abc│ │abc│ │
│ └───┘ └───┘ └───┘ │
└∊──────────────────┘
```

```
┌→──────────────────┐
↓ ┌→──┐ ┌→──┐ ┌→──┐ │
│ │abc│ │abc│ │abc│ │
│ └───┘ └───┘ └───┘ │
│ ┌→──┐ ┌→──┐ ┌→──┐ │
│ │abc│ │abc│ │abc│ │
│ └───┘ └───┘ └───┘ │
└∊──────────────────┘
```

```
2 3ρ'abc'   ⍝ not nested
≤2 3ρ'abc'  ⍝ nested
```

Skip to main content

```
┌→──┐
↓abc│
│abc│
└───┘
```

```
┌───────┐
│ ┌→──┐ │
│ ↓abc│ │
│ │abc│ │
│ └───┘ │
└∊──────┘
```

Partition `⊆`

Dyadic `⊆` is called partition (`⊂` and `⊆` originate with different APL dialects, but Dyalog APL features both). To distinguish between them, we call `⊂` *partitioned enclose* and `⊆` just *partition*, but it doesn't say much.

Dyadic `⊆` works similarly to dyadic `⊂`, but with different rules for the left argument. The left argument is non-negative integer instead of Boolean, and new partitions begin whenever an element is higher than its neighbour on the left. Also, elements indicated by 0s are dropped completely:

```
1 0 0 1 1 3 2 2 5 5 0⊆'Hello World'
```

```
┌→───────────────────┐
│ ┌→┐ ┌→─┐ ┌→──┐ ┌→─┐ │
│ │H│ │lo│ │ Wo│ │rl│ │
│ └─┘ └──┘ └───┘ └──┘ │
└∊───────────────────┘
```

`1⊆array` is the same as `,⊂array` but uses a single dyadic function instead of two monadic ones, i.e. great for trains.

Materialise `⌷`

Monadic `⌷` is materialise. It is almost the same as monadic `⊢` (i.e. identity). However, it will

Skip to main content

effect it turns collections into vectors of items.

```
]dinput
:Class cl
    :Property Default thing
    :Access Public Shared
        ∇ r←get
          r←3 1 4 1 4
        ∇
    :EndProperty
:EndClass
```

```
cl.thing
⊢cl
⌷cl
```

```
┌→────────┐
│3 1 4 1 4│
└~────────┘
```

```
#.cl
```

```
┌→────────┐
│3 1 4 1 4│
└~────────┘
```

Index `⌷`

Dyadic `⌷` is index. It is similar to pick, dyadic `⊃`, but works its way into the rank instead of the depth. On a 3D array, the first element selects layer, the second row, the third column:

```
2 3 4⍴⎕A
2⌷2 3 4⍴⎕A
2 1⌷2 3 4⍴⎕A
2 1 3⌷2 3 4⍴⎕A
```

Skip to main content

```
┌┌→───┐
↓↓ABCD│
││EFGH│
││IJKL│
││    │
││MNOP│
││QRST│
││UVWX│
└└────┘
```

```
┌→───┐
↓MNOP│
│QRST│
│UVWX│
└────┘
```

```
┌→───┐
│MNOP│
└────┘
```

```
O
-
```

Each element of the left argument may be may be any simple array:

```
(⊂1 1)⌷2 3 4⍴⎕A
2 (1 3)⌷2 3 4⍴⎕A ⍝ first and third row of second layer
(1 2)1 3⌷2 3 4⍴⎕A ⍝ third char of first row of layers 1 and 2
(1 2)(2 3)⌷2 3 4⍴⎕A ⍝ rows 2 and 3 of each of layers 1 and 2
```

Skip to main content

```
┌┌→───┐
↓↓ABCD│
││EFGH│
││IJKL│
││    │
││ABCD│
││EFGH│
││IJKL│
└└────┘
```

```
┌→───┐
↓MNOP│
│UVWX│
└────┘
```

```
┌→─┐
│CO│
└──┘
```

```
┌┌→───┐
↓↓EFGH│
││IJKL│
││    │
││QRST│
││UVWX│
└└────┘
```

Grade up/down `⍋⍒`

Next up is `⍋`, called grade up. Monadic `⍋` takes a simple (non-nested) array and returns the indices of the major cells reordered so that they would order the array.

Easiest to understand with an example:

```
⍋3 1 4 1 5
```

```
┌→────────┐
│2 4 1 3 5│
└~────────┘
```

This means that the second element (1) is the smallest, then the fourth (1), then the first (3), etc.

Skip to main content

```
3 1 4 1 5[⍋3 1 4 1 5]
```

```
┌→────────┐
│1 1 3 4 5│
└~────────┘
```

It works on high-rank arrays too:

```
3 2⍴2 7 1 8 2 8
⍋3 2⍴2 7 1 8 2 8
```

```
┌→──┐
↓2 7│
│1 8│
│2 8│
└~──┘
```

```
┌→────┐
│2 1 3│
└~────┘
```

So the first is row 2 `(1 8)` then row 1 `(2 7)` then row 3 `(2 8)`. It works on characters too, where it grades in Unicode point order:

```
5 2⍴'HelloWorld'
⍋5 2⍴'HelloWorld'
(5 2⍴'HelloWorld')[⍋5 2⍴'HelloWorld';]
```

Skip to main content

```
┌→─┐
↓He│
│ll│
│oW│
│or│
│ld│
└──┘
```

```
┌→────────┐
│1 5 2 3 4│
└~────────┘
```

```
┌→─┐
↓He│
│ld│
│ll│
│oW│
│or│
└──┘
```

```
4 2 2⍴'Hello World PPCG'
⍋4 2 2⍴'Hello World PPCG'   ⍝ layer grade up
```

```
┌┌→─┐
↓↓He│
││ll│
││  │
││o │
││Wo│
││  │
││rl│
││d │
││  │
││PP│
││CG│
└└──┘
```

```
┌→──────┐
│1 4 2 3│
└~──────┘
```

Layer 1, layer 4, layer 2, layer 3:

```
{⍵[⍋⍵;;]}4 2 2⍴'Hello World PPCG'
```

Skip to main content

```
┌┌→──┐
↓↓He│
││ll│
││  │
││PP│
││CG│
││  │
││o │
││Wo│
││  │
││rl│
││d │
└└───┘
```

`⍋⍋` is the cardinal numbers:

```
⍋'PPCG'
⍋⍋'PPCG'
```

```
┌→──────┐
│3 4 1 2│
└~──────┘
```

```
┌→──────┐
│3 4 1 2│
└~──────┘
```

So P is the third, P is the fourth, C is the first, and G is the second. Applying `⍋` to a permutation inverts it (swaps between cardinal and grade). Another way to think about it is that `⍋` is the indices of cells in the order that would sort them. `⍋⍋` is the position each will take when sorted. If you think about it hard, you'll see why ⍋ swaps back and forth between these two.

Here's an example where the grade and the cardinals differ:

```
⍋'random'
⍋⍋'random'
⍋⍋⍋'random'  ⍝ grading the cardinals takes us back to grade
```

Skip to main content

```
┌→──────────┐
│2 4 6 3 5 1│
└~──────────┘
```

```
┌→──────────┐
│6 1 4 2 5 3│
└~──────────┘
```

```
┌→──────────┐
│2 4 6 3 5 1│
└~──────────┘
```

`⍋` once is what order the elements would be in when sorted and `⍋` twice is the indices that each element would go to.

Dyadic `⍋` is for character arrays only, and it grades as if the left argument was the alphabet:

```
{⍵['aeioubcdfghjklmnpqrstvwxyz'⍋⍵]}'helloworld'
```

```
┌→─────────┐
│eoodhlllrw│
└──────────┘
```

If characters are missing from the alphabet, they will be considered after the alphabet, and equivalent:

```
'abcdefgh'⍋'hawl'
```

```
┌→──────┐
│2 1 3 4│
└~──────┘
```

Dyadic `⍋` can also use multiple levels of sorting:

```
⍉↑'aeiou' 'bcdfghjklmnpqrstvwxyz'
```

Skip to main content

```
┌→─┐
↓ab│
│ec│
│id│
│of│
│ug│
│ h│
│ j│
│ k│
│ l│
│ m│
│ n│
│ p│
│ q│
│ r│
│ s│
│ t│
│ v│
│ w│
│ x│
│ y│
│ z│
└──┘
```

This 2D “alphabet” means that all vowels should come before all consonants, and only if otherwise the same, the vertical order will be considered.

```
{⍵[(⍉↑'aeiou' 'bcdfghjklmnpqrstvwxyz')⍋⍵]}'helloworld'
```

```
┌→─────────┐
│eoodhlllrw│
└──────────┘
```

This sorted all vowels before all consonants, and only then did it sort the vowels and the consonants. You can have up to 15 levels of sorting using this. If a letter occurs more than once, then its first occurrence rules. This is useful to fill gaps in (e.g.) columns of unequal height.

There is also `⍒`, which is grade down, which follows the pattern of `⍋`, but sorts the other way.

Skip to main content

`⍳ ⍸ ∊ ⍷ ∪ ≠ ∩ ~`

Index generator `⍳`

Monadic `⍳` is the index generator. `⍳a` generates an array of shape a where the elements are the indices for that element:

```
⍳10
⍳2 4
```

```
1 2 3 4 5 6 7 8 9 10
```

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

Any bets on what `⍳0` gives?

```
]display ⍳0
```

```
┌⊖┐
│0│
└~┘
```

The empty numeric list. What about `⍳0 0`?

```
]display ⍳0 0
```

```
┌⊖──────┐
⌽ ┌→──┐ │
│ │0 0│ │
│ └~──┘ │
└∊──────┘
```

Skip to main content

Index-of `⍳`

The dyadic version `A⍳B` is index-of. It finds the first occurrence of the major cells of `B` in the major cells of `A`:

```
'hello'⍳'l'
'hello'⍳'lo'
```

```
3
```

```
3 5
```

If a cell is not a member, it will return a number one higher than the number of elements:

```
'hello'⍳'x'
```

```
6
```

```
(3 2⍴'abcdef')⍳(2 2⍴'cdxy')
```

```
2 4
```

So the "cd" row is the second one, and the "xy" row is not there. This behaviour for elements that are not there is really useful for supplying a "default":

```
'First' 'Second' 'Third' 'Missing'['abc'⍳'cdab']
```

```
 Third  Missing  First  Second
```

Skip to main content

Where `⍸`

Monadic `⍸` is where. It just takes a simple array and returns the list of non-zero indices.

```
⍸0 1 0 1 1
```

```
2 4 5
```

```
⊢m←2 3⍴0 1 0 1 1 0
⍸m
```

```
0 1 0
1 1 0
```

```
 1 2  2 1  2 2
```

If the argument array is not Boolean, the values are taken to mean the repeat count for each index:

```
⍸2 3⍴0 2 0 2 2 0
```

```
 1 2  1 2  2 1  2 1  2 2  2 2
```

A code golf trick: sum a Boolean array with `≢⍸` instead of `+/`,

```
≢⍸2 3⍴0 1 0 1 1 0
```

```
3
```

Skip to main content

Interval index `⍸`

Dyadic `⍸` is interval index. It takes a list of sorted arrays on the left, and for each array on the right, tells which “gap” (interval) it belongs.

```
1 10 100 1000⍸0 500 2000 3 10
```

```
0 3 4 1 2
```

So 0 is in interval number 0 (that is, before 1–10). 500 is in interval 3, which is 100–1000, etc. And as you can see from 10, it is in interval 2; 10–100. So intervals are [min,max). For higher rank arrays, it works like *grade*, i.e. on major cells.

Membership `∊`

Dyadic `∊` is membership. For each scalar in the left argument, return a Boolean if it is a member of the right argument:

```
'aeiou'∊'Hello World'
```

```
0 1 0 1 0
```

Question:

> Does APL have an “insert at index” command? As in, given an array, an index and a value, insert value at the index in the array. Example: [1, 2, 4, 5], 2, 3 => [1, 2, 3, 4, 5]

There are a couple of approaches:

```
∊(⊂,∘3)@2⊢1 2 4 5
```

Skip to main content

```
1 2 3 4 5
```

This appended a 3 to the 2, then flattened. You flatten with monadic `∊` which is the function we're up to. A more traditional and better performing approach would be:

```
{3@(1+2)⊢⍵\⍨1+2=⍳≢⍵}1 2 4 5
```

```
1 2 3 4 5
```

but we have not covered the `\` function yet.

Enlist `∊`

`∊` is enlist:

```
⊢m←(⍳3)(2 2⍴⍳4)
∊m
```

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

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

Find `⍷`

Next up is `⍷` which is (as of yet) only dyadic. `⍷` is find. It returns a Boolean array of the right argument's shape with a 1 at the "top left" corner of occurrences of the left argument in the right argument:

```
'ss'⍷'Mississippi'
```

```
0 0 1 0 0 1 0 0 0 0 0
```

Skip to main content

The ones here indicate the left “s” wherever “ss” begins. It also works for overlaps,

```
'aba'⍷'alababa'
```

```
0 0 1 0 1 0 0
```

and for higher-rank arrays:

```
2 2⍴0 1 0
3 3⍴0 1 1 0
(2 2⍴0 1 0)⍷(3 3⍴0 1 1 0)
```

```
0 1
0 0
```

```
0 1 1
0 0 1
1 0 0
```

```
1 0 0
0 1 0
0 0 0
```

and also for nested arrays, too:

```
'aa' 'bbb'⍷'c' 'aa' 'bbb' 'dddd' 'aa' 'aa' 'bbb'
```

```
0 1 0 0 0 1 0
```

Quiz using `⍷`: Determine if A is a prefix of B.

▶ Click for quiz answer

How about: Is A a suffix of B?

▶ Click for quiz answer

Skip to main content

Union `∪`

Next function is dyadic `∪`. It is basically union of multi-sets. However, it is symmetrical in a way you can often use to your advantage:

```
'abcc'∪'cda'
'cda'∪'abcc'
```

```
abccd
```

```
cdab
```

It preserves duplicates from the left argument, while only adding the items from the right necessary to make the result contain all elements from both. It will add duplicate elements from the right if they are not in the left, though:

```
'abcc'∪'cdda'
```

```
abccdd
```

Unique `∪`

The monadic `∪` is unique. It simply removes duplicates:

```
∪'mississippi'
```

```
misp
```

Skip to main content

Unique mask `≠`

Monadic `≠` is unique mask. It returns a Boolean vector which, when used as left argument to `⌿` and with the original argument as right argument, returns the same as `∪` would on the original argument:

```
∪'mississippi'
≠'mississippi'
{(≠⍵)⌿⍵}'mississippi'
```

```
misp
```

```
1 1 1 0 0 0 0 0 1 0 0
```

```
misp
```

We'll cover this in greater depth in a later chapter.

Intersection `∩`

Dyadic `∩` is, of course, intersection, again asymmetric:

```
'abcc'∩'cda'
'cda'∩'abcc'
```

```
acc
```

```
ca
```

It removes elements from the left which are not present in the right. Duplicates in the right do not matter.

Skip to main content

Without `~`

The last multi-set function is dyadic `~` which is without or *except*. It simply removes from the left whatever is on the right. Note that it can take even high-rank right arguments.

```
'Mississippi'~'pss'
```

```
Miiii
```

NOT `~`

Monadic `~` is logical NOT, simply swapping `1→0` and `0→1`:

```
(3 3⍴0 1 1 0) (~3 3⍴0 1 1 0)
```

```
 0 1 1  1 0 0
 0 0 1  1 1 0
 1 0 0  0 1 1
```

`/\⌿⍀,⍪`

Replicate `/`

Next up is `/`. When what's on its left is an array rather than a function it instead acts like a function, which makes it unusual. We cover the operator case of `/` elsewhere, e.g. `+/` for sum.

As a function, `/` is called replicate. It replicates each element on the right to as many copies as indicated by the corresponding element on the left:

```
1 1 2 1 2 1 2 1/'Misisipi'
```

Skip to main content

```
Mississippi
```

A more common usage is with a Boolean left argument, where it then acts as a filter:

```
1 0 1 1 0 0 1 0 1 1 1/'Hello World'
```

```
HllWrld
```

It has one more trick: if you use a negative number, then it replaces the corresponding element with that many prototypes (spaces for characters and zeros for numbers):

```
1 1 ¯1 1 1/'Hello'
```

```
He lo
```

You can also use a single scalar to “empty” an array:

```
0/'abc'
1/'abc'
```

```

abc
```

`1/x` can also be used to ensure that `x` has at least one dimension (it ravels scalars, leaving all other arrays untouched):

```
⍴1/8   ⍝ Scalar becomes vector, rank 1
⍴1/8 8 ⍝ Higher ranks remain untouched
```

```
1

2
```

Skip to main content

Expand `\`

`/` has a cousin, `\`, which, when used as a function, is called expand.

Positive numbers on the left also replicate like with `/` but negative numbers insert that many prototypical elements at that position:

```
1 1 ¯1 1 1 1\1 2 3 4 5
```

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

You can use `0` instead of `¯1` which makes it convenient to use Boolean left arguments.

We can now begin to see how we can insert into an array. Let’s go back to the problem of inserting 3 in between 2 and 4 in the list 1 2 4 5. Our method was:

Get the indices of the elements:

```
⍳≢1 2 4 5
```

```
1 2 3 4
```

Look where the index is 2:

```
2=⍳≢1 2 4 5
```

```
0 1 0 0
```

That’s where we want to expand:

```
1+2=⍳≢1 2 4 5
```

Skip to main content

```
1 2 1 1
```

Use `\` to perform the expansion:

```
(1+2=⍳≢1 2 4 5)\1 2 4 5
```

```
1 2 2 4 5
```

Replace the extra 2 with our desired element:

```
3@(1+2)⊢(1+2=⍳≢1 2 4 5)\1 2 4 5
```

```
1 2 3 4 5
```

Just like the operators `/` and `\` each have a sibling, `⌿` and `⍀` which do the same thing but along the first axis (i.e. on the major cells) so to with the functions `/` and `\`:

```
(1 0 1/3 3⍴⎕A) (1 0 1⌿3 3⍴⎕A)
(1 ¯2 1 1\3 3⍴⎕A) (1 ¯2 1 1⍀3 3⍴⎕A)
```

```
 AC  ABC
 DF  GHI
 GI


 A  BC  ABC
 D  EF
 G  HI
        DEF
        GHI
```

Ravel `,`

Monadic `,` ravels. It takes all the scalars of an array and makes a single vector (list) out of them. This includes a scalar, so `,3` is a one-element vector:

Skip to main content

```
3 3⍴⎕A
,3 3⍴⎕A
```

```
ABC
DEF
GHI
```

```
ABCDEFGHI
```

Question:

> Isn't that the same as monadic `∊`?

It is not. For example,

```
∊3 3⍴⎕A
∊3 3 3⍴⍳27
```

```
ABCDEFGHI
```

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27
```

The difference is that `∊` will take all the data and make it a simple vector. `,` will take all the scalars and make it a (potentially nested) vector:

```
∊2 2⍴'abc' 'def' 'ghi' 'jkl'
,2 2⍴'abc' 'def' 'ghi' 'jkl'
```

```
abcdefghijkl
```

```
 abc  def  ghi  jkl
```

`∊` is the same as recursive application of `⊃,/`.

Skip to main content

Catenate `,`

Which brings us to dyadic `,`, catenate, which is simply concatenation:

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

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

`,` can also get specified an axis upon which to act:

```
(2 3⍴⎕A),[1](2 3⍴⍳6)
(2 3⍴⎕A),[2](2 3⍴⍳6)
```

```
A B C
D E F
1 2 3
4 5 6
```

```
ABC 1 2 3
DEF 4 5 6
```

You can even use fractional axes to specify that you want to concatenate along a new inserted axis between the next lower and higher integer axes:

```
(2 3⍴⎕A),[0.5](2 3⍴⍳6) ⍝ 3D array
(2 3⍴⎕A),[1.5](2 3⍴⍳6) ⍝ 3D array
```

Skip to main content

```
A B C
D E F

1 2 3
4 5 6
```

```
A B C
1 2 3

D E F
4 5 6
```

This works for the monadic form too:

```
,[0.5]2 3⍴⎕A
⍴,[0.5]2 3⍴⎕A
,[1.5]2 3⍴⎕A
⍴,[1.5]2 3⍴⎕A
```

```
ABC
DEF
```

```
1 2 3
```

```
ABC

DEF
```

```
2 1 3
```

Catenate first `⍪`

Then we have `⍪`. The dyadic `⍪` is a synonym for `,[1]`, and it's sometimes referred to as catenate first:

```
(2 3⍴⎕A),[1](2 3⍴⍳6)
(2 3⍴⎕A)⍪(2 3⍴⍳6)
```

Skip to main content

```
A B C
D E F
1 2 3
4 5 6
```

```
A B C
D E F
1 2 3
4 5 6
```

Table `⍪`

Monadic `⍪` is called table as it ensures that the result is a table. It ravels the major cells of an array and makes each one of them into a row (i.e. a major cell) of a matrix:

```
2 3 4⍴⎕A
⍪2 3 4⍴⎕A
```

```
ABCD
EFGH
IJKL

MNOP
QRST
UVWX
```

```
ABCDEFGHIJKL
MNOPQRSTUVWX
```

That is, monadic `⍪` is just a synonym for `,⍤¯1` (except for scalars). To be universal, we'd need to say `{,⍤¯1⊢1/⍵}`.

`⍴⌽⊖⍉⍎⍕`

Reshape `⍴`

We've met ⍴ (Greek Rho) in passing before. Let's cover it in more depth. `⍴` is maybe the most

Skip to main content

rank) arrays. Note that `⍴` is not *actually* the Greek Rho in Unicode. Dyalog APL only uses the special Unicode APL Rho.

The Greek letter Rho is has the sound of the letter R, and stands for reshape. The right argument of `⍴` is used in ravel order to fill an array with the dimensions given by the left argument. The left argument must therefore be a vector (list) of dimension lengths (although for ease of use, we do allow a scalar instead of a one-element vector). Another way to look at it is that the left argument of `⍴` is the index of the last element in the resulting array (if you stick to the default `⎕IO` of 1). If you omit the shape (left argument) then the current shape is returned.

```
3⍴'a'
3⍴'ab'
3⍴'abcd'
2 3⍴'abc'
```

```
aaa
```

```
aba
```

```
abc
```

```
abc
abc
```

That’s two rows and three columns. The order of the left argument is the number of major cells first and of “leaf” cells last.

```
3 4 5∘.+10 20 30 40
⍴3 4 5∘.+10 20 30 40
```

```
13 23 33 43
14 24 34 44
15 25 35 45
```

```
3 4
```

A scalar doesn’t have any dimensions, so the corresponding left argument is `⍬` (or `0⍴0`):

```
⍬⍴3 4 5∘.+10 20 30 40
```

```
13
```

If one or more dimensions are 0, then the array doesn't have any elements, but it is still there. If it has rank 2 or higher, then it has an empty default display. If an array has no elements, then `⍴` will uses its prototype to fill any array it needs to form:

```
2 3⍴⍬
```

```
0 0 0
0 0 0
```

Recall that `⍬` is just `0⍴0` so it being simple and numeric, its prototype is 0.

Reverse `⌽`

Monadic `⌽` is reverse. It reverses the leaf rank-1 sub-arrays of an array. For a matrix, it means reversing each row:

```
2 4⍴⍳8
⌽2 4⍴⍳8
```

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

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

For a vector, it simply means reversing the vector:

```
⎕A
⌽⎕A
```

Skip to main content

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
ZYXWVUTSRQPONMLKJIHGFEDCBA
```

Of course, it doesn't affect scalars.

Reverse first `⊖`

`⌽` has a sibling, just like `/` and `\` have `⌿` and `⍀`, namely reverse first, `⊖`, which I usually call "Flip". `⊖` reverses the order of major cells, which for a matrix means reversing the order of the rows, i.e. flipping it upside down:

```
⊖2 4⍴⍳8
```

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

For vectors, it is the same as `⌽` and again it does nothing to scalars. For a 3D array, it reverses the order of layers:

```
4 2 3⍴⎕A
⊖4 2 3⍴⎕A
```

Skip to main content

```
ABC
DEF

GHI
JKL

MNO
PQR

STU
VWX


STU
VWX

MNO
PQR

GHI
JKL

ABC
DEF
```

Dyadic `⌽` and `⊖` do rotations instead of reversals:

```
3⊖⎕A
1⊖4 2 3⍴⎕A
```

```
DEFGHIJKLMNOPQRSTUVWXYZABC


GHI
JKL

MNO
PQR

STU
VWX

ABC
DEF
```

Negative rotation amounts just rotate to the other way:

Skip to main content

```
XYZABCDEFGHIJKLMNOPQRSTUVW
```

Here is a cool feature of `⌽` and `⊖`: If you give them a vector of rotation amounts, they get distributed on the relevant cells:

```
3 4⍴⎕A
1 0 2⌽3 4⍴⎕A
1 0 ¯1 0⊖3 4⍴⎕A
```

```
ABCD
EFGH
IJKL
```

```
BCDA
EFGH
KLIJ
```

```
EBKD
IFCH
AJGL
```

Transpose `⍉`

`⌽` and `⊖` also have a cousin named `⍉` (Transpose). The monadic function does not reverse the major cells or the rank 1 cells, but rather reverses the order of the indices. For matrices this is normal transposing:

```
3 4⍴⎕A
⍉3 4⍴⎕A
```

```
ABCD
EFGH
IJKL
```

```
AEI
BFJ
CGK
DHL
```

Skip to main content

For arrays of rank higher than 2 it helps to think of the shape as being reversed:

```
⍉2 3 4⍴⎕A
```

```
AM
EQ
IU

BN
FR
JV

CO
GS
KW

DP
HT
LX
```

If you look carefully, you can see that the runs like ABCD which originally spanned rows are now spanning layers. Look at the top left corner of each new layer. So, too, are the layers now spanning rows. Look how the top left of the layers, A and M are now next to each other in a row. Whilst the column AEI is still a column, because reversing the shape 2 3 4 (layers, rows, columns) gives 4 3 2 (columns, rows, layers) so the runs spanning rows are in the same position, still spanning rows.

Now you know how to reverse the order of axes, but what if you want an entirely *new* order? That's what dyadic `⍉` does. The left argument is the indices of the axes in the desired order. Therefore, if we reverse the indices of the rank, it is the same as monadic transpose:

```
3 2 1⍉2 3 4⍴⎕A
```

Skip to main content

```
AM
EQ
IU

BN
FR
JV

CO
GS
KW

DP
HT
LX
```

Now we can keep the layers and only reverse (i.e. transpose) columns/rows:

```
1 3 2⍉2 3 4⍴⎕A
```

```
AEI
BFJ
CGK
DHL

MQU
NRV
OSW
PTX
```

Here is a very cool thing: You can duplicate indices in the left argument. If so, APL will merge the indicated axes, taking only the elements that have equal indices along those two axes. This is the diagonal or diagonal plane, or diagonal 3D array (!), etc.

```
3 4⍴⎕A
1 1⍉3 4⍴⎕A
1 1 1⍉2 3 4⍴⎕A
1 1 2⍉2 3 4⍴⎕A
```

Skip to main content

```
ABCD
EFGH
IJKL
```

```
AFK
```

```
AR
```

```
ABCD
QRST
```

Here the layers and rows got merged, i.e. 1st row of 1st layer and 2nd row of 2nd layer, while the columns stayed as is.

```
1 2 1⍉2 3 4⍴⎕A
```

```
AEI
NRV
```

Here we merged layers and columns, i.e. 1st column of 1st layer and second column of 2nd layer. Dyadic `⍉` is pretty advanced and quite rarely used, but when you need it (and can figure out the correct left argument — experiment!) it is really handy.

Here's an example. Given a multiplication table, what were the numbers that generated it?

```
3 3⍴9 6 12 6 4 8 12 8 16 ⍝ A multiplication table
```

```
 9 6 12
 6 4  8
12 8 16
```

In this case, the answer is `3 2 4`:

```
∘.×⍨3 2 4
```

Skip to main content

```
  9 6 12
  6 4  8
 12 8 16
```

We can ‘reverse engineer’ this by finding the square root of the diagonal elements:

```
1 1⍉3 3⍴9 6 12 6 4 8 12 8 16       ⍝ main diagonal
0.5*⍨1 1⍉3 3⍴9 6 12 6 4 8 12 8 16
```

```
9 4 16
```

```
3 2 4
```

Execute `⍎`

Execute, `⍎`, evaluates a string representing a line of APL. This can be any valid APL expression, including functions and multiple statements:

```
⍎'2+3'
2(⍎'+')3
⍎'a←2 ⋄ a←a+3 ⋄ a'
```

```
5
```

```
5
```

```
5
```

The result of `⍎` is the result of the last statement, if that has a result. If it doesn’t (e.g. it is an empty statement or has a leading `{}`), then `⍎` doesn’t have a result either. The result of `⍎` can be a monadic operator:

```
≢(⍎'¨')'abc' 'defg'
```

```
3 4
```

`⍎` has all the features of a line of APL. You can run your entire program from `⍎`. Indeed, when a workspace is loaded, APL automatically does `⍎⎕LX` to bootstrap your application. This is what causes the greeting message when you load a workspace like dfns.

Dyadic `⍎` is exactly like the monadic, but executes the expression in the namespace named in the left argument.

```
0 0⍴a←'base'
ns←⎕NS⍬
ns.a←'sub'
⍎'a'
'ns'⍎'a'
```

```
base
```

```
sub
```

Here we first set `a` to `'base'` in `#` (the root namespace), then we created the empty namespace `ns`, populated it there, then evaluated `a` here (in `#`) and then in `ns`. In other words, monadic `⍎` is the same as dyadic `⍎` but with the default left argument of `⎕THIS` (this current namespace).

Nowadays, we usually “dot into” namespaces to evaluate there:

```
0 0⍴a←'base'
ns←⎕NS⍬
ns.a←'sub'
⍎'a'
ns.⍎'a'
```

```
base
```

```
sub
```

Same as before, but here we used the “value” of `⍎` *inside* `ns` instead of `⍎`’s value here.

Skip to main content

Format `⍕`

Format, `⍕`, is really quite simple. It returns a simple character vector or matrix which displays exactly as if its argument had been displayed:

```
]display 1 2 3 4    ⍝ numeric vector
≢1 2 3 4

]display ⍕1 2 3 4  ⍝ convert to character vector
≢⍕1 2 3 4
```

```
┌→──────┐
│1 2 3 4│
└~──────┘
```

```
4
```

```
┌→──────┐
│1 2 3 4│
└───────┘
```

```
7
```

If you give `⍕` a left argument, it will display numeric values with that many decimals, rounding 5 up:

```
4⍕2÷3     ⍝ character vector of 2÷3 rounded to 4 dp
4⍕1 2 3÷3
```

```
 0.6667
```

```
 0.3333 0.6667 1.0000
```

If you give it two values as left argument, it will use the first as “field width” and the second as the number of decimal places:

```
20 4⍕1 2 3÷3
```

```
              0.3333              0.6667              1.0000
```

You can also use twice as many elements on the left as there are leaf cells on the right, and it will pair each two on the left to each one on the right:

```
10 4 20 0 15 1⍕1 2 3÷3
```

```
    0.3333                   1             1.0
```

User-defined functions

In APL, a function can be applied to data, that is, arrays. Note that “arrays” include scalars: a scalar is an array of rank 0.

There are three distinct types of functions, and several ways to create them. The types of functions are *tacit*, *dfns*, and *tradfns*.

Tacit and one-liner dfns can easiest be created by using simple assignment, like we do with arrays:

```
avg←+⌿÷1⌈≢           ⍝ a tacit function
avg 7 6 2 9 6 3 4 5
```

```
5.25
```

```
avg←{(+⌿⍵)÷1⌈≢⍵}     ⍝ a dfn version of the same function
avg 7 6 2 9 6 3 4 5
```

```
5.25
```

You can’t have multi-line tacit functions, although tacit functions may consist of other multi-line non-tacit functions.

Skip to main content

To create a multi-line dfn or tradfn called `foo`, the easiest way is to type `)ed foo` in the session (the REPL). The editor will open with the first line pre-populated with the name `foo`. You can then start extending the function, e.g. to say (the `]dinput` thing is only required in a Jupyter notebook cell when entering multi-line functions, not in the session):

```
]dinput
r←foo nums
r←'Here are your numbers: ',⍕nums
```

Then press Esc to close and save your function in the workspace (the working container – you still need to save your workspace to disk later). The above is a tradfn. A tradfn is good for doing many things, one after another, things that may not necessarily be directly connected. The first line is a header line. It tells APL what the syntax is for that function. In our case, it says that `foo` has a result which will be referenced in the code as `r`, and it takes a single argument (which must be on the right) called `nums`. You can find the full model syntax for the header line here.

Multi-line dfns look like this:

```
]dinput
foo←{
    'Here are your numbers: ',⍕⍵
}
```

In a dfn, the right argument is always called `⍵` and the result is not named, rather, the first statement which is not an assignment (or after a true guard – we can come back to that) is the result.

If a dfn needs a left argument (all dyadic APL functions are infix) it can be referenced with `⍺`.

Both tradfns and dfns can be made *shy*. It means that the function by default does not cause implicit display of its result, but the result can still be captured by any code on its left.

A tradfn can be made shy by enclosing its result name in curly braces:

```
]dinput
{r}←foo nums
r←'Here are your numbers: ',⍕nums
```

Skip to main content

```
foo 1 2 3 4          ⍝ No output
vals←foo 1 2 3 4     ⍝ Capture return
vals
```

```
Here are your numbers: 1 2 3 4
```

A dfn can be made shy by letting the last statement be after a *guard*, and have an assignment:

```
]dinput
foo←{
    1:a←'Here are your numbers: ',⍕⍵
}
```

```
foo 4 5 6 7          ⍝ No output
vals←foo 4 5 6 7     ⍝ Capture return
vals
```

```
Here are your numbers: 4 5 6 7
```

In most circumstances, you should avoid using shyness, though. It can be confusing.

A *guard* is a dfn-specific feature. It consists of a statement (a condition) which must evaluate to a Boolean (i.e. 0 or 1), followed by a colon (:) followed by the result of the function if the condition is true.

```
istrue←{⍵:'true' ⋄ 'false'}  ⍝ a guard statement

istrue 1
istrue 0
```

```
true
```

```
false
```

`⋄` and line breaks are equivalent in almost all cases. One difference is that when you trace through a function, you can only execute one line at a time, even if it has multiple statements separated by

Skip to main content

always execute completely and quit. If an error happens, the stack is cut back to their caller. This is actually useful, to prevent your program from stopping in a bad state.

Tacit, tradfns and dfns end up being different, even though their outwards behaviour may be identical. They have different detailed *name classification*. `⎕NC` (Name Classification) is a system function which takes one or more names and tells you something about them:

```
]dinput
{r}←foo nums                      ⍝ example tradfn
r←'Here are your numbers: ',⍕nums
```

```
avg←+⌿÷1⌈≢                        ⍝ example tacit
istrue←{⍵:'true' ⋄ 'false'}       ⍝ example dfn
```

```
⎕NC 'foo' 'avg' 'istrue'
```

```
3.1 3.3 3.2
```

APL distinguishes between two types of functions when it comes to applying to data: *scalar functions* and *mixed functions*.

Scalar functions penetrate the entire structure of the given arrays, all the way until the simple scalars; hence the name. Mixed functions apply to some larger structures, sometimes only regarding one argument, while the other is treated as scalars.

Examples of scalar functions are the arithmetic functions; `+ - × ÷ ⌈ ⌊` etc. Scalar functions also have something called scalar extension: not only do the functions “pair up” the data, like how `1 2 3+10 20 30` gives `11 22 33`, but they also distribute scalars to all the elements of the other argument, e.g. how `1+10 20 30` gives `11 21 31`.

```
1 2 3+10 20 30
1+10 20 30
```

Skip to main content

```
11 22 33
```

```
11 21 31
```

This is useful, because it means you can enclose pieces of your data to tell APL that something should be distributed. This also lets us see the benefit of having both rank and depth.

E.g. `⍺∊⍵` looks if each element of `⍺` is a member of `⍵`.

```
'hello'∊'CodeGolf'
'hello'∊'Code' 'Golf'
(⊂'hello')∊'Code' 'Golf'
```

```
0 1 1 1 1
```

```
0 0 0 0 0
```

```
0
```

The first example looks whether each element of `'hello'` is a member of `'CodeGolf'`. The second looks whether each element of `'hello'` is a member of the list of words. Of course, there are no single letters in the list of words. The last example looks whether the word `'hello'` is in the list on the right.

Sometimes, this isn’t enough, though. Sometimes you want to apply your function is a non-standard way. This is where operators come in. APL operators (higher-order functions) take one or two functions as operands and apply them in a specific way.

For example, `¨` (called each) is a monadic operator which applies its operand function to each element of the argument(s). Take, for example, the monadic function `≢` which tallies the length of its argument:

```
≢'Code' 'Golf'
≢¨'Code' 'Golf'
```

```
2
```

```
4 4
```

So while `¨` digs into an array, [rank](), `⍤`, applies the function to sub-arrays of a specific rank. For example, `≢⍤1` applies *Tally* to rank 1; that is vectors, thus finding the length of each row (they are of course the same, as all rows in a matrix must be equal length, but you get the idea):

```
⊢A←2 4⍴'CodeGolf'
(≢⍤1) A
```

```
Code
Golf
```

```
4 4
```

You can also define your own operators. There are only two types; *dops* and *tradops*. There are no tacit operators in APL. Tradops are much like tradfns. The only real difference is the header line. So while a tradfn header can look like `result←function arg`, a tradop header can look like `result←(fn operator)arg`. This tells APL that operator takes a single function `fn` as operand, and the resulting combined function is monadic (takes just the right-argument `arg`.

In a dop, much like a dfn, the arguments and operands have fixed names, and the result is the first non-assignment. The dops' name of its left (or only) operand is `⍺⍺` and the right operand is `⍵⍵`. The arguments are `⍺` and `⍵` just like in a dfn.

For example, we can create a dop `twice` which applies the left argument with the operand two times:

```
twice←{⍺ ⍺⍺ ⍺ ⍺⍺ ⍵}
2+twice 5
```

```
9
```

Note that this is different than defining `plustwice←{⍺+⍺+⍵}`, because the operator can be applied

Skip to main content

My favourite defined operator is `under`:

```
under←{(⍵⍵⍣¯1) (⍵⍵ ⍺) ⍺⍺ (⍵⍵ ⍵)}
```

Any guesses as to what it does?

`⍣` is another operator, which applies the function on its left as many times as indicated by its right operand. This also shows that operands may be both functions and arrays, the syntax is the same. `⍺⍺` and `⍵⍵` may each be a function or an array. `f⍣¯1` means apply `f` negative one time, i.e. apply the *inverse* of `f`. The inverse of `⍟` (log) is `*` (power).

Power is to multiplication what multiplication is to addition, so `×under⍟` is power. `*under⍟` is tetration.

Tacit programming

Tacit programming is programming without (direct) reference to the argument(s). Of course, you still need to get the data somehow, but the idea is that a function refers to the result of that function when applied to the argument(s) instead of just referring to itself. When you actually need to refer to an argument, you still need to apply a function to it, but since you want nothing done to the data, you'll need an identity function. Dyalog APL gives you `⊣` and `⊢` which are left and right identity, respectively. This may seem trivial, but becomes very important later.

Next, we need to understand how a train (sequence) of functions is applied to the argument(s). Since APL functions can be called monadically or dyadically (niladic functions cannot directly be used in trains), there needs to be some rules. We also need a way to specify if we want any subsequent functions to be applied to the result of the previous functions, or on the argument(s) anew.

3-trains

Let's begin with 3-trains, or `f g h`. They tend to be the simplest to understand. In the following, we'll call the left and right arguments `A` and `B` respectively. First up is the (albeit slightly more complicated) dyadic case, as the monadic case follows very easily from the dyadic one.

Skip to main content

Evaluating `A (f g h) B` from the right, we first have `h` which represents `A h B`. Then we move on to `g` which will evaluate to `f g (A h B)`. So we need to evaluate `f` first. `f` behaves just like `h`, in that it refers to `A f B`. Finally, `g` can be evaluated as `(A f B) g (A h B)`.

Note that there is no confusion between this last non-tacit (or *explicit*) expression and a train. You can always tell the difference between explicit and tacit APL by looking at the rightmost token. If it is an array, it is explicit, otherwise it is tacit. Conversely, this also means that you need to separate a train from any data you want to apply it to, either by naming it in a separate statement, or by parenthesising it. Getting confused regarding this is a very common mistake.

Going back to our `f g h` train, what happens in the monadic case? The dyadic was `(A f B) g (A h B)`, and the monadic is exactly the same, but with the `A`s removed: `(f B) g (h B)`. This applies universally to all trains: The parsing is identical for monadic and dyadic calls; the functions that would address the left argument are just called monadically. This also means that `⊣` refers to the right argument when the train is called monadically.

3-trains are known as *forks* because their structure resembles a fork (like a rail switch) in that the middle function "connects" to the two sides. We can use the interpreter to help us display a visual representation of a fork:

```
]box on -t=tree    ⍝ Enable tree-display for tacit functions
+×÷                ⍝ A fork
```

```
Was OFF -trains=tree
```

```
┌─┼─┐
+ × ÷
```

2-trains

This leads us to 2-trains. Consider `f g h` again — `(A f B) g (A h B)`. What if there was no `f`? I.e. we just have `g h`. Since `g` would address its left argument, but there isn't any, it is just called monadically, i.e. `A (g h) B` is `g (A h B)`. This is known as an *atop* because the `g` is evaluated atop (i.e. on the top of) the result of `h`'s application.

Skip to main content

4-trains

Let’s look at 4-trains. (1-trains are simply single functions.) Consider `f g h j`. We begin from the right and grab up to three functions, i.e. `g h j`. Those are evaluated as before. Let’s call the result `H`. Now we have `f H`. Really, `f` would have taken a left argument, but there isn’t any, so it is just applied to `H` monadically. In total, `f g h j` is `f (g h j)` or to be explicit, `A (f g h j) B` is `f ((A g B) h (A j B))`.

One little exception which fits right in: The left side of the 3-train (left tine of a fork) may be a constant (i.e. not a function that is applied to the argument(s). It is then treated as if there had been a function there which gave that result. Here’s an illustration: `A (42 g h) B` is just like `A ({42} g h) B` where `{42}` is an ambivalent function which returns a constant value. So it all becomes `42 g (A h B)` or `(A{42}B) g (A h B)` if you want.

Note that you cannot have a 2-train with a constant left side, like `42 f`. Neither can you have a middle tine be a constant, like `f 42 g`. Nor can you have a right hand side be a constant, as that would make your code explicit, as per above. So what if you *need* a constant right-tine? For example, for a “divide-by-42” function? `⊢÷42` won’t work (it’ll give you the identity of the reciprocal of 42). Then you need to supply the constant as a left tine, and swap the arguments of the middle tine, using the `⍨` (Commute) operator: `42÷⍨⊢`.

5-trains

Finally, let’s have a look at a 5-train, which completes the pattern. `f g h j k`: again we begin from the right and take three functions. Now we have `f g (h j k)`. `h j k` evaluates as a normal 3-train, and its result (let’s call it `J`) becomes the right argument of `g`, so `f g J`. Then the pattern just repeats. A 4-train is an atop of a fork, and a 5-train is a fork of a fork, and a 6-train is an atop of a fork of a fork, etc.

Tacit rules

Let’s look at some handy identities.

- Because `(f g) B` is `f (g B)` then if `g` is `⊢` then `(f g)` is just `f`

Skip to main content

- Because `A (f g h) B` is `(A f B) g (A h B)` then if `f` is `⊣` and `h` is `⊢`, then `(f g h)` is just `g`.
- Because `A (f g h) B` is `(A f B) g (A h B)` then if `f` is `⊢` and `h` is `⊣`, then `(f g h)` is just `g⍨`.

We could, of course, make many more such identities, but I'm sure you get the idea, so just one more:

- Because `(f g) B` is `f g B` and `f∘g B` is also `f g B`, we can substitute `(f g)` with `f∘g` in monadic cases.

Converting dfns to tacit

OK, let's look at the dfn given here:

```
f←{(,⍨⍴⍵↑⍨×⍨)⌈.5*⍨≢⍵}
```

Note that converting to tacit form doesn't always make the code shorter. This is just for the exercise. We can begin by substituting `⊢` for every `⍵` (the right argument). That gives us `(,⍨⍴⊢↑⍨×⍨)⌈.5*⍨≢⊢` which won't work because of how trains are evaluated, so let's fully parenthesise it:

```
(,⍨⍴⊢↑⍨×⍨)(⌈(.5*⍨(≢⊢)))
```

Note that the left parenthesis is already a train, but this still doesn't work, because that train used the constant `⍵`, which we've substituted with a `⊢`. But `⊢` inside the train refers to the train's own right argument, and we want the original right argument. So we need to "feed" the left train the unadulterated argument:

```
⊢(,⍨⍴(⊢⊣)↑⍨×⍨)(⌈(.5*⍨(≢⊢)))
```

But now we get another issue: the functions in that train assumed the train was called monadically. That's not the case any more, so let's insert some tacks to use the correct arguments:

Skip to main content

OK, that was the left side. Now for the right side. `(≢⊢)` becomes just `≢` as per above identity, and the rightmost parenthesis isn't needed:

```
⊢ ((,⍨⊢)ρ⊣↑⍨(×⍨⊢)) (⌈.5*⍨≢)
```

Now we can see that `⌈` is applied monadically to its right argument, so we can glue to to the left train instead:

```
⊢ ((,⍨⊢)ρ⊣↑⍨(×⍨⊢))∘⌈ (.5*⍨≢)
```

Of course, we can remove that rightmost parenthesis too:

```
⊢ ((,⍨⊢)ρ⊣↑⍨(×⍨⊢))∘⌈ .5*⍨≢
```

That's it. But we can do a little better. Note that `,⍨` and `×⍨` are "selfies". It should be obvious that `f⍨ X` is the same as `X f X` (no matter if `X` is a function or a constant), so we can just substitute that:

```
⊢ ((⊢,⊢)ρ⊣↑⍨(⊢×⊢))∘⌈ .5*⍨≢
```

Now we can remove final unneeded parenthesis and the whitespace:

```
⊢((⊢,⊢)ρ⊣↑⍨⊢×⊢)∘⌈.5*⍨≢
```

There you go. Totally unreadable, but it looks cool!

```
⊢((⊢,⊢)ρ⊣↑⍨⊢×⊢)∘⌈.5*⍨≢
```

```
       ┌─┼────────┐
       ⊢ ∘    ┌───┼─┐
      ┌──┴──┐ 0.5 ⍨ ≢
  ┌───┼───┐ ⌈   ┌─┘
┌─┼─┐ ρ ┌─┼───┐ *
⊢ , ⊢   ⊣ ⍨ ┌─┼─┐
        ┌─┘ ⊢ × ⊢
```

Skip to main content

Let’s do one more: Moris Zucca’s dfn `{⊃⍵[(⍳⍴⍵)~⍵⍳⍵]}`.

Right away we can spot an issue: you can’t use bracket indexing in a train, but luckily there is a functional alternative in the `⌷` primitive. So, first let’s substitute that in:

```
{⊃⍵⌷⍨⊂(⍳⍴⍵)~⍵⍳⍵}
```

Now, let’s do our `⍵→⊢` substitution:

```
⊃⊢⌷⍨⊂(⍳⍴⊢)~⊢⍳⊢
```

Just a couple of things to fix in this one: `⍳⍴⊢` won’t work, and `⊂` is called monadically, but we can easily fix those:

```
⊃⊢⌷⍨(⊂((⍳⍴)⊢)~⊢⍳⊢)
```

Now we’ve got an `f ⊢` case in `(⍳⍴)⊢`, so we’ll simplify as per the identity above:

```
⊃⊢⌷⍨(⊂(⍳⍴)~⊢⍳⊢)
```

Since `(⍳⍴)` is called monadically, we can use `⍳∘⍴`:

```
⊃⊢⌷⍨(⊂⍳∘⍴~⊢⍳⊢)
```

Note that the rightmost `⍳` uses the same left and right argument, so it is a selfie: `⍳⍨`

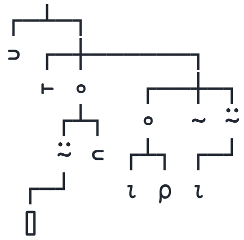
```
⊃⊢⌷⍨(⊂⍳∘⍴~⍳⍨)
```

Finally, `⊂` is called monadically, so we can glue it to `⌷⍨`:

```
⊃⊢⌷⍨∘⊂⍳∘⍴~⍳⍨
```

```
⊃⊢⌷⍨∘⊂⍳∘⍴~⍳⍨
```

Skip to main content



Here's another. My dfn `{ω⊆⍨(⍴ω)↑⍺/+\⍺}`. This one is fun. Let's start with substitution:

```
⊢⊆⍨(⍴⊢)↑⊣/+\⊣
```

OK, on the right we have a monadic `+` so we'll need to parenthesise it:

```
⊢⊆⍨(⍴⊢)↑⊣/(+\⊣)
```

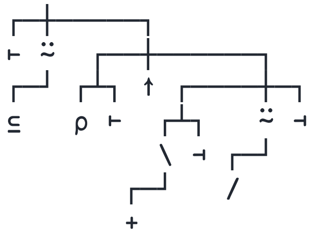
But now note that `/` is used as a function. However, it prefers to be an operator, i.e. doing `⊣` reduction instead of `⊣` replication. To force it into function mode, we need to make it the operand of an operator (since operators cannot be operands). We can use the trick that `f⍨⍨` is the same as `f` (in dyadic cases):

```
⊢⊆⍨(⍴⊢)↑⊣(/⍨⍨)(+\⊣)
```

But since we're anyway swapping arguments (twice) we may as well just swap the actual `⊣` and `(+\⊣)` instead:

```
⊢⊆⍨(⍴⊢)↑(+\⊣)(/⍨)⊣
```

```
⊢⊆⍨(⍴⊢)↑(+\⊣)(/⍨)⊣
```



Skip to main content

Tradfns

Tradfns are the original way to write your own functions in APL. Tradfns are procedural in style, unlike dfns, which are functional.

The basic structure of a tradfn is:

```
∇ header line
function body
∇
```

Function body

Control structures

Let’s consider the body first. We have available to us the full set of control structures from procedural languages. All such key words begin with a colon, `:`, for example `If … :EndIf`. Lines with such keywords must begin with the keyword, and have nothing else on them, although parameters (like a condition) are considered parenthesised expressions. For example,

```
∇ Ex ;i;j;k
  :For i j k :In 'abc'(1 2 3)'ABC'
      ⎕←i j k
  :EndFor
∇
```

```
Ex
```

```
abc
1 2 3
ABC
```

This assigns `(i j k)←'abc'` during the first loop, then `(i j k)←1 2 3`, etc. `:For` can also

Skip to main content

```
∇ Ex ;i;j;k
  :For i j k :InEach 'abc'(1 2 3)'ABC'
      ⎕←i j k
  :EndFor
∇
```

```
Ex
```

```
a 1 A
b 2 B
c 3 C
```

Any unpacking is possible, for example:

```
∇ Ex ;i;j;k
  :For i(j k) :InEach (⍳3)('aA' 'bB' 'cC')
      ⎕←i j k
  :EndFor
∇
```

```
Ex
```

```
1 aA
2 bB
3 cC
```

`:If`, of course, has `:Else`, but also `:ElseIf`. While `∧` and `∨` are normal arithmetic functions, it is allowed to write one or more `:AndIfs` or `:OrIfs` which *will* shortcut. A quite common pattern used to check if a variable exists and then, for example, set it to a default value if it doesn't:

```
∇ Ex ;state
  :If 0=⎕NC'state'
      state←42
  :EndIf
  ⎕←state
∇
```

Skip to main content

```
42
```

Ambivalence

While dfns are always ambivalent (though `⍺` will give value error if called monadically and there's no `⍺←` statement), Dyalog tradfns have to be explicitly declared ambivalent in the header: `∇result←{lAarg} FnName rArg`. Then one can test for `⎕NC'lArg'`, but there's also a faster way: 900⌶ which ignores its argument and returns whether the function was called monadically:

```
∇ res←{lArg} Ambiv rArg
  :If 900⌶⍬
      lArg←42
  :EndIf
  res←lArg ⍝ Return the left argument
∇
```

```
Ambiv 'hello'
99 Ambiv 'world'
```

```
42
```

```
99
```

Note that `900⌶` only works for tradfns, although dfns don't need it so much since they have `⍺←`.

Advanced control structures

`:If` and `:While` should feel familiar, but the `:Select` statement warrants specification:

```
:Select expression
:Case value
:CaseList values
:Else
:EndSelect
```

No need "break" like in C's `switch` statement. It jumps to the end when reaching the next case

Skip to main content

The conditional loops are a bit interesting in that you can piece them together as you want. You can begin with either `:While condition` (which checks before it starts) or `:Repeat` which doesn't check. You can end with either `:EndWhile/:EndRepeat` (which don't check anything) or `:Until` condition (which does). In other words, you can match `:While` with `:Until`. `:While` and `:Until` can also be followed by one or more `:AndIfs` or `:OrIfs`.

You can even insert statements between :`If/:ElseIf/:While/:Until` and `:AndIf/:OrIf`, but this can be hard to read. For example, consider the following:

```
∇ r←Foo val;b
  b←1
  :If 10<val
      b←2
  :AndIf 100>val
      r←b,val
  :Else
      r←val,b
  :EndIf
∇
```

```
Foo 5
Foo 50
Foo 500
```

```
5 1


2 50


500 2
```

The `:AndIf` and `:OrIf` allows you to build up Boolean expressions that have the same kind of short-circuiting behaviour as that found in mainstream languages, but with the added option of statements between them. Whilst this can be confusing to read, it has its place, for example, where you have some costly set-up code required in order to evaluate one of the expressions making up a boolean condition in an if-statement. You can do work that needs to be prepared so we're ready to do the next check. For example,

```
:If ⎕NEXISTS file
```

Skip to main content

```
    Process¨content
:EndIf
```

That sort of thing would be painful to write in as a dfn.

You can do the same with loops, too:

```
∇ r←Foo val
  r←val
  :Repeat
      r+←?5
  :Until r>11
  :OrIf r=9
∇
```

```
Foo 1
Foo ¯100
```

```
12
```

```
9
```

When looping, you can also continue with the next iteration without finishing this one, by stating `:Continue` and you can quit the loop immediately with `:Leave`:

```
∇ r←Foo
  r←0
  :While 1
      r+←1
      :If r>10
          :Leave      ⍝ Like 'break' in C or Python
      :EndIf
  :EndWhile
∇
```

```
Foo
```

```
11
```

Skip to main content

Non-flow structures

There’s actually another couple of interesting structures, which aren’t really flow control per se. `:Section…:EndSection` is like `:If 1` which is useful for organising your code, and they don’t need a comment symbol on their right. You can put any text there. The `:Section` itself provides no actual visible functionality.

```
∇ r←Foo arg
  r←arg
  :Section We can group code that belongs together in sections
      :If r>10
          ⎕←'Greater than 10'
      :EndIf
  :EndSection
∇
```

```
Foo 4
Foo 15
```

```
4


Greater than 10
15
```

`:Trap` takes one or more error numbers exactly like dfns’ error guards. Then the main code, and then `:Case` or `:CaseList` with error numbers. You can also/instead use `:Else` for all (other) errors.

Tradfns can also do advanced stuff that dfns can’t do. If you write `:Implements trigger var` then the function gets called every time var is changed in that namespace.

```
∇ r←Foo
  :Implements trigger var
  ⎕←'var changed!'
∇
```

Skip to main content

```
var changed!
```

If you want a callback on *all* variable changes, you can use `*` instead of a name. You can also use `var1,var2` to only react to those. `:Implements` is just a declaration, not a structure.

The header

There can be up to four parts of the header:

- result
- calling syntax
- locals
- comment

Result

The result is optional and must be terminated by `←` if present. It contains the result name or a parenthesised list of space-separated names.

If one needs to return a vector of various values, then using a name list is nice, because one can assign to each name separately, and only upon return are they collected together:

```
∇(vertices results)←…
  vertices←…
  results←
```

Fun fact: a name can occur multiple places in the header, including in a single name list, so you can actually write somewhat useful function without any body, just a header. For example, `∇(x x)←dup x` makes two of its argument. And `(x y)←x juxtapose y` is the same as `{α ω}`.

The result can also be made “shy”, like a dfn that ends with an assignment `{shh←42}`. This is done by putting the name or the name list in braces. For example, `∇{shh}←Shy shh` will silence its argument, but the value can still be coerced out.

If the result variable name is a function, then the function will return that function! Behold:

Skip to main content

```
∇ Fn←PlusMinus
 :If 1=?2
     Fn←+
 :Else
     Fn←-
 :EndIf
∇
```

Then `3 PlusMinus 4` will give either `¯1` or `7`, each time it is run, it is random.

```
3 PlusMinus 4
3 PlusMinus 4
3 PlusMinus 4
```

```
¯1

¯1

¯1
```

Calling syntax

The calling syntax of the header is always be present. It is basically an image of how the function needs to be called. For example, a monadic function would have `FunctionName argumentName`. A dyadic function would have `leftArg FnName rightArg`. The right argument can also be a name list like the result. In that case, APL will refuse to call the function with anything but a vector argument of the correct length. This is pretty neat for "type" checking. A tradfn can be made ambivalent by putting braces around the left argument name, as we discussed before. The left and right arguments are not allowed to be the same, but multiple names in the right argument can be the same (last will prevail) which is convenient if you're writing a function that needs to take multiple arguments, some of which it doesn't need, for example, `∇ foo(important _ critical _ _)`.

A tradfn can be also be *niladic*, unlike a dfn. Then the syntax part is just the function name. This is usually used for returning caches, bootstrapping, constants, etc. Another useful thing is for a niladic tradfn is to return a derived function, since that allows you to use the editor on it, and also to

Skip to main content

```
∇ f←Avg
  sum←+⌿
  count←1⌈≢
  f←sum÷count
∇
```

So, about operators. The "central" part of the syntax declaration for an operator needs to be parenthesised. It then has two names for a monadic operator `(Operand OPERATOR)` or three names for a dyadic operator `(Operand1 OPERATOR Operand2)`. Outside the parenthesis there must be a name or namelist on the right for the right argument(s), and optionally an optionally optional left argument on the left. In other words, that is either no left argument or yes a left argument or a braced left argument.

Now we can also understand why allowing a left argument namelist would make it really hard to understand what the header stood for: things like `(a b)(c d)` and `(a b c)d e` would certainly be tougher to parse for humans. In practice, if multiple "arguments" are needed, people tend to use multiple right arguments. Of course, you can always unpack any array into any structure, not just a simple list.

As opposed to dfns, tradfns do not auto-localise. This means that it is important that you do so by declaring all your locals. After the syntax part, one can write one or more names, each prefixed by `;` to localise them. There's no need to localise other names that occur in the header. They're all local. The only exception is the function/operator's own name. If you really want to reuse that name, you can localise it explicitly. As a relatively new feature (17.0), you can continue localising names up until you have any actual code (so comments and empty lines are fine):

```
∇foo;local
;more;locals
⍝ finally:
;last;ones
```

Finally, the header line allows a comment. Nothing fancy there. Just a comment :-)

So in summary:

`∇{(result1 result2)}←{left}(Op1 OP Op2)(right args);local;local2 ⍝ comment`

Skip to main content

System functions

The name *System Function* is informally applied to all built-in names which begin with the quad symbol (`⎕`), even if they are actually operators, variables, or constants. We'll cover these roughly in the order presented here.

System functions are things that are not really part of the core language, but have been wrapped into items which conform with normal APL syntax. You can therefore use system functions together with normal APL functions and operators. However, note that many system functions are "shy", meaning that they suppress implicit display of their result, and some even do this selectively.

Behaviour, session

There are several system functions that control behavioural aspects of the interpreter and the session itself.

Comparison tolerance `⎕CT`

To deal with inexactness in floating point representation, we have `⎕CT`, which is Comparison Tolerance. Some APL primitives have implicit arguments, i.e. arguments which are given as values to (semi) global variables instead of on the right or left.

`⎕CT` is a tiny value:

```
⎕CT
```

```
1E¯14
```

Two floating point numbers `X` and `Y` are considered to be equal if `(|X-Y)≤⎕CT×(|X)⌈|Y`:

```
1=1+1e¯15
```

Skip to main content

```
1
```

You can set `⎕CT` within reasonable limits (you can't make two unequal ints the same), so you can just set it to something else if you need to modify (or even disable) this behaviour:

```
⎕CT←1E¯10     ⍝ More tolerant
1=1+1e¯11

⎕CT←0         ⍝ Disable comparison tolerance
1=1+1e¯15

⎕CT←1E¯14     ⍝ Reset to default
```

```
1
```

```
0
```

If you use 128-bit decimal floats (we'll get back to that), you can instead use `⎕DCT`, [Decimal Comparison Tolerance]().

Division method `⎕DIV`

Some of you may be uncomfortable with the default divide by zero behaviour:

```
0÷0
```

```
1
```

Dyalog has this thing called `⎕DIV`, [Division method](), which, when you set it to 1, lets all divisions by 0 give 0:

```
⎕DIV←1
0 0 3 3÷0 3 0 3
⎕DIV←0
```

Skip to main content

```
0 0 0 1
```

If you want to error on division by zero, just use `×∘÷` instead of `÷` under the default `⎕DIV←0`.

Index origin `⎕IO`

There is an old debate on whether to begin indexing with 0 or with 1. APL lets you choose by setting the Index Origin, `⎕IO`:

```
⍳4 ⊣ ⎕IO←0
⍳4 ⊣ ⎕IO←1
```

```
0 1 2 3
```

```
1 2 3 4
```

Note that using `⎕IO←0` means you have to accept negative indices in some cases:

```
3 4 5⍳2
```

```
0
```

```
⎕IO←0
3 4 5⍳2
⎕IO←1
```

```
¯1
```

Also note that these system variables can be localised. So if your dfn sets `⎕IO` it only applies to that function (and its children), but does not permanently affect the environment:

```
⎕IO,({⎕IO←0 ⋄ ⎕IO}⍬),⎕IO
```

Skip to main content

```
1 0 1
```

Print precision `⎕PP`

By default, APL prints 10 significant digits in floats. You can select how many to show by setting `⎕PP`, Print Precision:

```
⎕PP←3
÷7
⎕PP←10
÷7
```

```
0.143
```

```
0.1428571429
```

This affects `⍕`, too:

```
≢⍕÷7
```

```
12
```

```
≢⍕÷7 ⊣ ⎕PP←3
```

```
5
```

In other words, how many characters are needed to represent a seventh using that precision?

Now we can also get *more* precision:

```
○1 ⊣ ⎕PP←17 ⍝ π
```

Skip to main content

```
⎕PP←10 ⍝ Set back to default value
```

Floating-point representation `⎕FR`

What if we want even *more* decimal places in our π from above? Bumping the print precision higher doesn’t work:

```
○1 ⊣ ⎕PP←34
```

```
3.141592653589793
```

The system simply doesn’t keep that much precision. For this we need to set `⎕FR`, Floating-point Representation. By default it is 645, meaning 64-bit binary. We can set it to 1287, meaning 128-bit decimal:

```
○1 ⊣ ⎕PP←34 ⊣ ⎕FR←1287
⎕FR←645
```

```
3.141592653589793238462643383279503
```

Recall also that you can set decimal comparison tolerance with `⎕DCT`.

Random link `⎕RL`

Random link, `⎕RL`, lets you set a seed value for random numbers so you can reproduce the same random numbers again. It also lets you choose which method to use for calculating the next random number based on the seed.

`⎕RL` is a two element array, but as opposed to normal arrays, you cannot modify `⎕RL` in-place; you have to assign to the entire array at once. The first element is the seed; an integer in the range `1` to `¯2+2*31`. You can also use `0` to auto-randomise, or `⍬` to optimise by not keeping track of the seed.

Skip to main content

- 0=Lehmer
- 1=Mersenne
- 2=ask the OS.

If you ask the OS, you can’t provide a seed, so you have to use `⍬`:

```
?0 ⊣ ⎕RL ⍬ 2
```

```
0.1611569607474366
```

When asking our OS we get a different result each time:

```
?0 ⊣ ⎕RL ⍬ 2
```

```
0.2894394027399608
```

Let’s use Mersenne (the default) with a specific seed instead:

```
?0 ⊣ ⎕RL←42 1
?0 ⊣ ⎕RL←42 1   ⍝ Start the sequence at the same place
```

```
0.001953378319754839
```

```
0.001953378319754839
```

Account info `⎕AI`

Account info, `⎕AI`, isn’t very interesting these days, except you can use `⎕AI[3]` as an absolute counter of milliseconds since the beginning of the session. This is useful to avoid having to deal with roll-overs when timing stuff.

How long does it take to wait a second?

Skip to main content

```
a←3⊃⎕AI
⎕DL 1      ⍝ Sleep for 1s
a-⍨3⊃⎕AI
```

```
1282
```

Account name `⎕AN`

`⎕AN` is the account name, which for me is

```
⎕AN
```

```
jeremy
```

Clear workspace `⎕CLEAR`

Clear workspace, `⎕CLEAR`, is a special constant, which when referenced will clear the workspace just like `)clear` does. This means you can use it in code.

Copy workspace `⎕CY`

Copy workspace, `⎕CY`, is a function which copies from a workspace file to the current workspace. You give it the name of a workspace file as right argument, and optionally a name list on the left of items to copy. By default, it will copy everything.

```
'iotag'⎕CY'dfns'   ⍝ Copy the iotag function from the dfns workspace
¯5 iotag 5
```

```
¯5 ¯4 ¯3 ¯2 ¯1 0 1 2 3 4 5
```

Skip to main content

Delay `⎕DL`

`⎕DL` is delay as you saw before. It takes a number (floats are fine) of seconds and (shyly) returns the number of seconds actually used. `⎕DL` guarantees a delay of *at least* what you specified:

```
⎕←⎕DL 1
```

```
1.017188
```

Load `⎕LOAD`

You may have already used `)LOAD`. ⎕LOAD is basically the same, but in a function form. Give it the name of a workspace to load.

Off `⎕OFF`

⎕OFF is similar to `⎕CLEAR` in that referencing its value causes the workspace to be closed, but it also terminates APL. `⎕OFF` has a special syntax though. If you put a value immediately to its right, that will become APL's exit code.

Save `⎕SAVE`

⎕SAVE is similar to `)SAVE` in that it saves the current workspace to disk. However, `⎕SAVE` has a trick up its sleeve. If you use `⎕SAVE` under program control, you can then use `⎕LOAD` on the generated workspace file, and execution will continue where the `⎕SAVE` happened, with `⎕SAVE` giving the result 0. This allows you to write applications where the user can close the application and then resume the left-off state when opening the application again.

Time stamp `⎕TS`

`⎕TS` is time stamp, which returns the current system time as a 7-element vector; year, month, day,

Skip to main content

```
⎕TS
```

```
2024 3 5 9 59 17 838
```

When dealing with times and dates, there is also the date-time system function, `⎕DT`, which can convert between pretty much any date and time formats around.

Constants, tools and utils

In this section we'll cover some system constants and utility functions.

Alphabetic chars `⎕A`

`⎕A` is the uppercase English alphabet :

```
⎕A
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

There is no built-in for the lowercase alphabet, but you can get it with the case convert system function, `⎕C`:

```
⎕C⎕A
```

```
abcdefghijklmnopqrstuvwxyz
```

Digits `⎕D`

`⎕D` has the digits:

```
⎕D
```

Skip to main content

```
0123456789
```

Null item `⎕NULL`

`⎕NULL` is a [scalar null](#) value. It isn't really used much in APL itself, but you can meet it e.g. when importing spreadsheets where it represents empty cells. Note that it is *not* JSON `null`, which is represented as `⊂'null'` to match true and false being `⊂'true'` and `⊂'false'`. Note also that `⎕NULL` equals itself. These three (`⎕A ⎕D ⎕NULL`) are system *constants*; you can't assign to them.

Win/unix command `⎕CMD ⎕SH`

`⎕CMD` and `⎕SH` are identical, but the first feels more natural to Windows users while the second feels more natural to UNIX users. Pressing `f1` on them will give you the help appropriate for that OS. They are used to call the OS command processor:

```
⎕SH'ls /'
```

```
Applications
Library
System
Users
Volumes
bin
cores
dev
etc
home
opt
private
sbin
tmp
usr
var
```

Comma separated values `⎕CSV`

`⎕CSV` will import and export [Comma/Character Separated Values](#).

Skip to main content

```
⎕CSV '"abc","def",3' 'S'
```

```
 abc  def  3
```

It has a ton of options for almost anything you could want, including import and export directly to and from text files.

Data representation `⎕DR`

`⎕DR` is Data Representation. Monadically, it will tell you how an array is represented internally, and dyadically, it allows you to convert between data types:

```
⎕DR 42
```

```
83
```

Dyalog APL data type codes have two parts, the 1's place and the rest. The 1's place tells you which kind of data it is, the rest tells you how many bits are used to store it, with one exception: pointers are always 326 even on 64 bit systems. The number 42 gave us 83, where 3 means integer and 8 means 8-bit.

Dyalog APL has single-bit Boolean arrays, so they are type 11 where the rightmost 1 means Boolean, and the leftmost 1 means 1-bit.

```
⎕DR 1 0 1 1 1 0
```

```
11
```

Dyadic `⎕DR` lets you convert between types:

```
11⎕DR 42
```

Skip to main content

```
0 0 1 0 1 0 1 0
```

This takes the memory which was used to represent 42 and interprets it as if it was a Boolean array. You can also combine two steps of `⎕DR` into one. A two-element left argument will interpret the right argument as that type, then convert it to the type given by the second element of the left argument.

Format `⎕FMT`

`⎕FMT` is ForMaT. It is like a beefed up version of `⍕`. `⍕` retains the rank of its argument (except for numeric scalars becoming character vectors). `⎕FMT` always returns a matrix. Also, `⍕` treats control characters as normal characters, while `⎕FMT` will resolve them:

```
str←⎕←'abc',(⎕UCS 8),'def' ⍝ 8 is backspace
⍴⎕←⍕str                    ⍝ ⍕ treats backspace as any other char
⍴⎕←⎕FMT str                ⍝ ⎕FMT resolves it
```

```
abcdef
```

```
abcdef
7
```

```
abdef
1 5
```

You see that the `'c'` really was erased by the backspace.

Dyadic `⎕FMT` gives you access to a whole new language, namely a formatting specification language. We won't go though all the details here (see docs!), but here's a taste:

```
'I3,F5.2' ⎕FMT 2 4⍴⍳8
```

```
  1 2.00  3 4.00
  5 6.00  7 8.00
```

Skip to main content

The formatting string `I3,F5.2` means that each row should first have an integer, then a float which uses five characters in width and has 2 decimals, then this formatting is cycled as much as needed for all the columns (here twice).

Import/export JSON `⎕JSON`

`⎕JSON` imports/exports JSON. It works for both arrays and objects:

```
⎕JSON'[[42,null],"hello"]'
```

```
 42   null    hello
```

```
⊢ns←⎕JSON'{"abc":42,"de":null,"f":"hello"}'
ns.(abc f)
```

```
#.[JSON object]
```

```
42  hello
```

We can also export from APL to JSON:

```
⎕JSON ('abc' 1 2 3) 4 5
```

```
[["abc",1,2,3],4,5]
```

Just be aware that if you want to convert an APL string to JSON, you need use the left argument to specify whether you want import (0) or export (1).

You can also tell `⎕JSON` that you want your JSON fully white-spaced:

```
⎕JSON⍠'Compact'0⊢('abc' 1 2 3)4 5
```

Skip to main content

```
[
  [
    "abc",
    1,
    2,
    3
  ],
  4,
  5
]
```

Finally, whilst you can import any JSON object, not every APL namespace can be exported. For example, a namespace with APL functions cannot be converted to JSON. Again, `⎕JSON` has some more advanced options — see the docs. `⎕JSON` is fully compliant with JSON, though, but we do allow some leniency which allows you to create some JavaScript objects which are not valid JSON. For example,

```
⎕JSON 'hello' (⊂'world')
```

```
["hello",world]
```

We opted for a generalised system for strings without quotes, rather than special casing `null`. The I-beam that preceded `⎕JSON` did in fact use `⎕NULL`. By using enclosed strings, we can losslessly roundtrip. However, If you DO want to use APL's `⎕NULL`, you can specify this using the `Null` variant to `⎕JSON`:

```
j←⎕JSON⍠'Null' ⎕NULL⊢'{"name": null}'
j.name
j.name = ⎕NULL
```

```
[Null]
```

```
1
```

The JSON format doesn't support arrays of higher rank, only lists-of-lists. This means that not all APL constructs can be converted to JSON directly, for example:

Skip to main content

```
⎕JSON 2 3⍴⍳6 ⍝ DOMAIN ERROR
```

```
DOMAIN ERROR: JSON export: the right argument cannot be converted
      ⎕JSON 2 3⍴⍳6 ⍝ DOMAIN ERROR
      ^
```

However, when speaking with the world outside, we probably want our matrices to be converted to lists of lists. For this, we have the `HighRank` variant option:

```
⎕JSON⍠'HighRank' 'Split' ⊢ 2 3⍴⍳6
```

```
[[1,2,3],[4,5,6]]
```

This works universally, also recursing into namespaces:

```
mat←⍳2 3
cube←2 2 2⍴2
⎕JSON⍠'HighRank' 'Split'⎕NS'mat' 'cube'
```

```
{"cube":[[[2,2],[2,2]],[[2,2],[2,2]]],"mat":[[[1,1],[1,2],[1,3]],[[2,1],[2,2],[2,3]]
```

Another thing that `⎕JSON` can now do is to understand and create JSON5:

```
(ns←⎕JSON⍠'Dialect' 'JSON5'⊢'{noQuotes: [0xdecaf,0xC0FFEE] /* comment */}').noQuote
⎕JSON⍠'Dialect' 'JSON5'⊢ns
```

```
912559 12648430
```

```
{noQuotes:[912559,12648430]}
```

Maybe most importantly, JSON5 allows trailing commas in lists and objects:

```
⎕JSON⍠'Dialect' 'JSON5'⍠'Compact'0⍳3
```

Skip to main content

```
[
  1,
  2,
  3,
]
```

Compare with

```
⎕JSON⍠'Dialect' 'JSON'⍠'Compact'0⍳3
```

```
[
  1,
  2,
  3
]
```

Map file `⎕MAP`

`⎕MAP` is a function we'll only mention and not demonstrate (see the docs). It basically allows you to use a file as an array instead of keeping the array in memory. Very useful.

Unicode convert `⎕UCS`

This brings us to Unicode Convert, `⎕UCS`, which in its monadic form flips characters and their Unicode code points:

```
⎕UCS 954 945 955 951 956 941 961 945
```

```
καλημέρα
```

The dyadic form takes a left argument specifying an encoding scheme and converts to and from byte values rather than code points:

```
'UTF-8' ⎕UCS 206 179 206 181 206 185 206 177 32 207 131 206 191 207 133
```

Skip to main content

```
 γεια σου
```

Verify and fix input `⎕VFI`

`⎕VFI` is Verify and Fix Input. It takes a string and returns two lists. It cuts the string into space separated fields. Then it attempts to convert each field to a number. If it succeeds then the corresponding element of the left result list is 1 (else 0) and the corresponding element of the right list is the number (else 0).

```
⎕VFI '123 four 42'
```

```
 1 0 1  123 0 42
```

You can also specify one or more valid field separators as left argument:

```
';/'⎕VFI '123 four,42 5/2/4'
```

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

Here `123 four` were grouped because space is not a separator anymore, and so it is an invalid number. So too with `42 5`. Only `2` and `4` were valid. You can get just the valid numbers with:

```
//';/'⎕VFI '123 four,42 5/2/4'
```

```
 2 4
```

XML convert `⎕XML`

`⎕XML` is converts to and from XML, but the corresponding APL format is rather involved. We usually just use ⎕XML to verify that some XML is valid or to normalise whitespace:

Skip to main content

```
⎕XML⍣2 ⊢ '<xml><document id="001">An introduction to XML</document></xml>'
```

```
<xml>
  <document id="001">An introduction to XML</document>
</xml>
```

Case conversion `⎕C`

`⎕C` provides various handy case conversion operations for strings. The left argument, if given, currently has to be a single simple scalar integer, 1 or ¯1 or ¯3:

- 1 does upper-casing
- ¯1 does lower-casing
- ¯3 does case normalisation

For ASCII, and most European languages, there's no difference between lowercasing and normalising case. However, some languages have multiple forms of a single letter. Normalising makes all those forms the same, so they can be compared easily. For example, Greek has two lowercase forms of Σ: σ and ς. Even Latin script (like in English and German) used to use a medial form of S: ſ. Note that it does *not* "de-diacriticize": á and a are still seen as different. Nor does it do decomposition or other length-changing normalisation. The constants 2 and ¯2 and ¯4 are reserved for length-changing mapping (upper/lower) and folding (normalisation) in the future.

Here's an example: given a character vector, uppercase the first character.

```
'hello, world!' → 'Hello, world!'
```

```
1⎕C@1⊢'hello, world!'
```

```
Hello, world!
```

Next up: a better (still not perfect) palindrome checker. Given a string without diacritics, but which may have spaces, determine if it is a palindrome. Examples:

Skip to main content

```
'race car' → 1
'Σοφος' → 1
'hello' → 0
'Νιψον ανομηματα μη μοναν οψιν' → 1
```

```
((⊢≡⌽)¯3⎕C~∘' ')¨ 'race car' 'Σοφος' 'hello' 'Νιψον ανομηματα μη μοναν οψιν'
```

```
1 1 0 1
```

Here's a trick too: monadic `⎕C` is the same as `¯3∘⎕C`.

Date-time conversions `⎕DT`

`⎕DT` provides a wealth of date-time conversions. It allows you to convert any numeric representation of a date-time into any other representation. You can use it to glue together two 3rd-party systems that otherwise can't easily communicate.

```
20 ⎕DT 44053.674 ⍝ Dyalog to Unix time
```

```
1597162233
```

Dyalog's basic representation of a moment is the number of days since 1899-12-31. The advantage of Dyalog's system (which was actually the original one) is that you can then find the day-of-week with `7|⌊`:

```
7|⌊44053.674
```

```
2
```

0: Sunday, 1: Monday, etc.

Does anyone use some software that has its own date format? Answer: yes, you all do. APL does. It has the 7-element vector `⎕TS` for the current Time Stamp.

Skip to main content

```
¯1 ⎕DT 44053.674 ⍝ to ⎕TS
```

```
  2020 8 11 16 10 33 600
```

The left argument tells `⎕DT` what you want to convert to. The numbers are largely arbitrary, but not entirely so. Positive codes indicate a scalar format (one number per date-time) and negative numbers indicate a vector format (multiple numbers per date-time). Also, the number divided by 10 and floored indicates the family. So we had 2(0) for UNIX and 4(0) for applications (Excel). The last element of `⎕TS` is the milliseconds. We can get more precision in the `⎕TS`-style result by using `¯2` for microseconds and `¯3` for nanoseconds:

```
¯2 ⎕DT 44053.674
¯3 ⎕DT 44053.674
```

```
  2020 8 11 16 10 33 600000

  2020 8 11 16 10 33 600000000
```

Notice also that vector formats are enclosed. This allows `⎕DT` to handle arrays of dates:

```
¯1 ⎕DT 44053+⍳3
```

```
  2020 8 12 0 0 0 0  2020 8 13 0 0 0 0  2020 8 14 0 0 0 0
```

There are many of these codes; we won't cover them all here, but they are readily available in the documentation. What you do need to know is how to convert from one of these formats. Until now, we've just used the Dyalog day number. That's the default for simple scalars in the right argument. The default for enclosed vectors is the `⎕TS` format (`¯1`). If your input is anything else, you need to give `⎕DT` a two-element left argument. The first element is the input type, and the second is the output type.

For example, this converts an ISO year, week of year, day of week to `⎕TS`-style:

Skip to main content

```
2020 9 30 0 0 0 0
```

Another example: given two ISO-style dates (as a 2 element vector of Y,M,D vectors), compute the inclusive number of days between them. E.g. `(2020 6 25)(2020 08 10)` should give 47. `(2020 08 10)(2020 6 25)` should also give 47. `(2020 08 10)(2020 08 10)` should give 1.

```
diff ← {1+|-/1⎕DTω}
diff (2020 6 25)(2020 08 10)
diff (2020 08 10)(2020 6 25)
diff (2020 08 10)(2020 08 10)
```

```
47
```

```
47
```

```
1
```

Format Date-Time `1200⌶`

Above we covered how to convert between different numerical date-time representations. What about converting a numeric date-time representation to text? For that we can use the Format Date-Time I-beam function, `1200⌶`.

When you want to convert a numeric date-time to text, the first step is always to convert it to a Dyalog day number. After that, you can use `1200⌶` to convert that to text. It takes a left argument which is a format pattern.

```
'YYYY DD MM hhmm'(1200⌶)1⎕DT⊂2020 08 11 11 32
```

```
2020 11 08 1132
```

The system in the pattern for `1200⌶` is that numeric parts of the date are uppercase, while parts of the time are lowercase. You can use a single character for a variable-width pattern, or multi-character for a 0-padded pattern. If instead you want space-padding, use an underscore as the

Skip to main content

```
'YYYY-DD-MM@hh:mm' 'YYYY-D-M@h:m' 'YYYY-_D-_M@_h:_m'(1200⌶)¨1⎕DT⊂2020 8 11 1 3
```

```
 2020-11-08@01:03  2020-11-8@1:3  2020-11- 8@ 1: 3
```

`t` is for 12-hour. `h` is for 24-hour. Furthermore, the format also allows for casing and languages other than English:

```
'YYYY MMM D "at" h:mm'(1200⌶)1⎕DT⊂2020 8 1 4 30
'YYYY Mmm D' 'YYYY mmm D'(1200⌶)¨1⎕DT⊂2020 8 1
'__fr__YYYY Mmmm D'(1200⌶)1⎕DT⊂2020 8 1
```

```
2020 AUG 1 at 4:30
```

```
 2020 Aug 1  2020 aug 1
```

```
2020 Août 1
```

Like `⎕DT`, `1200⌶` has lots of options, including custom languages. Have a look at the documentation.

Code management, I/O, dates, errors

The next category has tools to deal with user defined functions.

Attributes `⎕AT`

User defined functions can have various attributes. For example, they can be niladic/monadic/dyadic/ambivalent, and they of course have an author and a time when they were written. To access this info, we have the attributes system function, `⎕AT`:

```
⎕AT '⎕SE.Dyalog.Utils.formatText'
```

Skip to main content

The first part, `1 ¯2 0`, means that `1`: has an result (which is implicitly printed), `¯2`: it is ambivalent (the left argument is optional) and `0`: it is not an operator. The next part is a timestamp, in `⎕TS` form. The third element is the lock state, with `0` for *unlocked*: APL allows you to lock your code so others cannot inspect and/or suspend it. The last element is the username of whoever last established the function, meaning who most recently made it into an actual function from a text source. It wouldn't update if the function was copied from a different workspace.

For various practical and/or historical reasons, there are a few different functions that let us inspect code under program control. A user in an interactive session can of course just use the editor.

All these system functions have names in the pattern `⎕xR` where x is a single letter.

Canonical representation `⎕CR`

The simplest one is `⎕CR`, character/canonical representation. It returns a matrix:

```
⎕CR '⎕SE.Dyalog.Utils.formatText'
```

```
text←{vals}formatText text;cr;pw;right;hang;first;lead;left
   ⍝ Format text according to specifications (see ]format -?)
:If 900⌶⍬ ⋄ vals←0 ⋄ :EndIf
text←{(+/∨\' '≠⌽⍵)↑¨↓⍵}∘⎕FMT⍣(1=≡text)⊢text ⍝ convert everything to VTV
text←↑,/(⊂''),(⊂vals)formatPar¨text
```

From this you can see on the first line that the function has a result (text) and that the left argument (vals) is optional (it is in braces).

Nested representation `⎕NR`

However, sometimes it is more practical to get the code as a vector of vectors (list of strings), e.g. to extract a single line. For that we have `⎕NR`, nested representation:

```
⊃⎕NR '⎕SE.Dyalog.Utils.formatText' ⍝ first line
```

```
text←{vals}formatText text;cr;pw;right;hang;first;lead;left
```

Visual representation `⎕VR`

Finally, you may want a single string (with newlines) with all the decorations: `⎕VR`, vector/visual Representation:

```
⎕VR '⎕SE.Dyalog.Utils.formatText'
```

```
     ∇ text←{vals}formatText text;cr;pw;right;hang;first;lead;left
[1]       ⍝ Format text according to specifications (see ]format -?)
[2]     :If 900⌶⍬ ⋄ vals←0 ⋄ :EndIf
[3]     text←{(+/∨\' '≠⌽⍵)↑¨↓⍵}∘⎕FMT⍤(1=≡text)⊢text ⍝ convert everything to VTV
[4]     text←↑,/(⊂''),(⊂vals)formatPar¨text
     ∇
```

Fix `⎕FX`

These three forms are all valid arguments to the function `⎕FX`, Fix, which will establish a function according to the code given (or return an index of the first line which was problematic):

```
3 plus 4 ⊣ ⎕FX 'r←a plus b' 'r←a+b'
```

```
7
```

Here `⎕FX` established the function plus (and returned its name, but we ignored that in favour of 4) and then we used the function right away.

As you may recall, tradfns and dfns can easily define dfns in their code, but they cannot easily define tradfns. `⎕FX` lets you dynamically define tradfns should you want to do so.

`⎕FX` works for dfns too:

```
3 plus 4 ⊣ ⎕FX 'plus←{' '⍺+⍵' '}'
```

Skip to main content

References `⎕REFS`

Remember the `formatText` function? It looks complex. Let’s get some order by listing all the identifiers that it uses. Enter References, `⎕REFS`:

```
⎕REFS '⎕SE.Dyalog.Utils.formatText'
```

```
 cr
 first
 formatPar
 formatText
 hang
 lead
 left
 pw
 right
 text
 vals
```

Stop, trace `⎕STOP` `⎕TRACE`

In the editor, you can set breakpoints (stops) and trace points (output function name, line number and value). You can also do this under program control using `⎕STOP` and `⎕TRACE`, we cannot demo this in a non-interactive environment. The syntax is simple, though. `linenumbers ⎕STOP 'fnname'` to set, and omit the left argument to query. Same for `⎕TRACE`.

I/O `⎕ ⍞`

You can explicitly request output using `⎕←` or `⍞←`. `⎕←` means print to `STDOUT` (with trailing newline) and `⍞←` means print to `STDERR` (without trailing newline). However, you can also use these two symbols for input. `←⍞` means read a line from `STDIN`, and `←⎕` means get a value from `STDIN`. See character input/output.

`⎕` will take an APL expression and evaluate it. If you give it an expression without a value, it will keep prompting until you give in (or enter `→` to abort). Expressions evaluated in `⎕` are not encapsulated, so side-effects will persist (e.g. removing your program!)

Skip to main content

Response time limit `⎕RTL`

For normal `⍞` input, you can also set a response time limit in seconds: `⎕RTL←10` gives the user 10 seconds to respond before a `TIMEOUT` error is thrown. You can trap this with a dfns error guard `{1006::}` or a tradfn `:Trap 1006`.

Enqueue event `⎕NQ`

Enqueue event, `⎕NQ`, is mostly used for GUI programming, but there is one other nifty thing you can use it for. The `Calendar` and `DateTimePicker` have two methods (functions) called `DateToIDN` and `IDNToDate`. But the root object (`#`, or the APL session itself) also has these methods. These convert between the `⎕TS` format (Y M D h m s ms) and a International Day Number (as a float, so it includes the time). These are great for date and time calculations. Two days from now:

```
3↑2⎕NQ#'IDNTODate',2+2⎕NQ#'DateToIDN'⎕TS
```

```
2024 3 7
```

Don’t worry much about the syntax. `⎕NQ` needs 2 as left argument (for this job) and then the `#` says to look in the root object. At the end is the timestamp/IDN, either appended (`,`) or juxtaposed. You can also use it to get the weekday:

```
4⊃2⎕NQ#'IDNTODate',2⎕NQ#'DateToIDN'⎕TS
```

```
1
```

0 is Monday.

Skip to main content

Read file `⎕NGET`

Dyalog APL has two sets of file handling system functions. One is intended to make it easy to work with Unicode files, the other gives low level control. There are lots of options, but the basic functionality is as follows. To read the contents of a Unicode file, use `⊃⎕NGET 'filename'`. This will normalise line breaks to `LF` (`⎕UCS 10`). If you'd rather have a list of lines, use `⊃⎕NGET 'filename' 1` instead. This will autodetect encoding and line break style, and should "just work" for almost all files. See docs if you want more fine-grained control.

Write file `⎕NPUT`

Similarly, you can put content into a file with `(⊂content) ⎕NPUT 'filename'`. If you want to overwrite any existing file, use `(⊂content) ⎕NPUT 'filename' 1`. Content may be either a simple character vector (string) or a "VTV" (vector of character vectors, i.e. a list of strings). Again, more fine-grained control is available.

Other file system functions `⎕MKDIR` `⎕NDELETE` `⎕NINFO`

There are also some nice utilities which make it easy to perform some of the most common file operations. You might wonder why not just use `⎕SH/⎕CMD` to ask the OS to do it for you? Because various OSs need various commands and syntax. These system functions will let you write truly cross-platform code.

`⎕MKDIR` and `⎕NDELETE` do what you'd think.

`⎕NINFO` gives you file listings' info like you'd get from `ls/dir`, but in a nice array format, perfect for further APL processing.

```
⍉↑1 0 6⎕NINFO⍠1⊢'/*'
```

Skip to main content

```
 1  /home             0
 1  /usr              0
 1  /bin              0
 1  /sbin             0
 2  /.file            1
 1  /etc              0
 1  /var              0
 1  /Library          0
 1  /System           0
 0  /.VolumeIcon.icns 1
 1  /private          0
 1  /.vol             1
 1  /Users            0
 1  /Applications     0
 1  /opt              0
 1  /dev              0
 1  /Volumes          0
 1  /tmp              0
 1  /cores            0
```

The first column (indicated by the 1 in the left argument) is the type; 1=directory, 2=file. The second column (0) is the name. The third column (6) is Boolean for whether that item is hidden or not. The `⍠1` indicates that the right argument contains wildcards. Otherwise it would look for a file which had actual question marks and/or stars in its name (normally a bad idea, but at least APL can handle it).

Event number `⎕EN`

In a dfn, you can trap errors with error guards `{errornums::result if error ⋄ try this}` and in tradfns with `:Trap errornums ⋄ try this ⋄ :Case errornum` etc. But what are those error numbers? The easiest way to find out is to cause the error and then check event number, `⎕EN`, which is a variable that you cannot set directly. It contains the error number of the most recent error.

```
2{0::⎕EN ⋄ ⍺÷⍵}5
```

```
0.4
```

This catches all errors and returns the error number (or the result of the division if no error happened).

Skip to main content

```
2{0::⎕EN ⋄ α÷ω}0
```

```
11
```

Error 11 is `DOMAIN ERROR` (due to division by zero).

Event message `⎕EM`

`⎕EM` is a function which takes an error number and gives you the corresponding event message for that event number (`⎕EN`):

```
{0::⎕EM ⎕EN ⋄ α÷ω}5
```

```
VALUE ERROR
```

Diagnostic message `⎕DM`

`⎕DM` (diagnostic message) is a vector of three character vectors; a canonical form of what you see in the session when an error happens:

```
{0::↑⎕DM ⋄ α÷ω}5
```

```
VALUE ERROR
      {0::↑⎕DM ⋄ α÷ω}5
                 ∧
```

Extended diagnostic message `⎕DMX`

`⎕DMX` is a namespace (an object) which has Diagnostic Message (Extended). It has a neat display form with more info:

Skip to main content

```
 EM       DOMAIN ERROR
 Message  Divide by zero
```

We can use `⎕JSON` to display all its contents:

```
2{0::⎕JSON⍠'Compact'0⊢⎕DMX ⋄ ⍺÷⍵}0
```

```
{
  "Category": "General",
  "DM": [
    "DOMAIN ERROR",
    "      2{0::⎕JSON⍠'Compact' 0⊢⎕DMX ⋄ ⍺÷⍵}0",
    "                                      ^"
  ],
  "EM": "DOMAIN ERROR",
  "EN": 11,
  "ENX": 1,
  "HelpURL": "https://help.dyalog.com/dmx/18.2/General/1",
  "InternalLocation": [
    "scald.cpp",
    405
  ],
  "Message": "Divide by zero",
  "OSError": [
    0,
    0,
    ""
  ],
  "Vendor": "Dyalog"
}
```

So this error was thrown on line 387 of `scald.cpp`.

Stack and workspace info

Let's continue with other things which deal with functions and other items under program control.

Latent expression `⎕LX`

If you want to have an application start without having the user enter a command (for example, a function name) to boot it, you can assign an expression to `⎕LX` (Latent eXpression) and then save

Skip to main content

your workspace with `⎕SAVE`. When the workspace is loaded (including from the command line) APL will do `⍎⎕LX`. This is what happens when you load the various workspaces supplied with APL.

```
)load dfns
≢⎕LX
⎕LX
```

```
/Applications/Dyalog-18.2.app/Contents/Resources/Dyalog/ws/dfns.dws saved Wed Apr  6

An assortment of D Functions and Operators.

      tree #                      ⍝ Workspace map.
      ↑¯10↑↓attrib ⎕nl 3 4        ⍝ What's new?
      ⍕notes find 'Word'          ⍝ Apropos "Word".
      ⎕ed'notes.contents'         ⍝ Workspace overview.



236


'
An assortment of D Functions and Operators.

      tree #                      ⍝ Workspace map.
      ↑¯10↑↓attrib ⎕nl 3 4        ⍝ What''s new?
      ⍕notes find ''Word''          ⍝ Apropos "Word".
      ⎕ed''notes.contents''         ⍝ Workspace overview.
'
```

Name classification `⎕NC`

Since APL does not enforce a naming scheme (although you might want to adopt one), you may wonder what a certain name is. `⎕NC` (Name Classification) to the rescue! Each type of item has a number. 2 is variable, 3 is function, 4 is operator, 9 is object.

```
⎕CY'dfns'   ⍝ Copy the dfns workspace silently
var←42
⎕NC ↑'blah' '123' 'var' 'to' 'notes'
```

```
0 ¯1 2 3 9
```

Skip to main content

`0` is undefined (but valid name). `¯1` is invalid name. `1` is really rare these days. It is a line label, and can only occur while a tradfn/tradop is running or suspended:

```
∇tradfn
label:
⎕NC↑'label' 'label2' 'label3'
label2:
∇
```

```
tradfn
```

```
1 1 0
```

Sometimes you want even more info. If the argument to `⎕NC` is nested, then the values get a decimal which mean: .1=traditional, .2=field/direct, .3=property/tacit, .4=class, .5=interface, .6=external class, .7=external interface.

```
⎕CY'dfns'    ⍝ Copy the dfns workspace silently
var←42
⎕NC 'blah' '123' 'var' 'to' 'notes'
```

```
0 ¯1 2.1 3.2 9.1
```

Name list `⎕NL`

Using those same codes, you can also use `⎕NL` (Name List) to enquire which items of those name classifications are visible. For example, here are all of the dfns workspace's operators:

```
⎕CY'dfns'
⎕NL 4
```

Skip to main content

Cut
Depth
H
UndoRedo
_fk
acc
alt
and
ascan
ascana
at
avl
bags
big
bsearch
bt
case
cf
cond
cxdraw
dft
do
each
else
file
fk
fk_
fnarray
foldl
for
invr
kcell
limit
lof
logic
ltrav
mdf
memo
nats
of
or
perv
pow
pred
profile
rats
ratsum
ravt
redblack
repl
roman
rows
sam
saw

Skip to main content

```
tc
ticks
time
traj
trav
until
vof
vwise
while
```

You can also specify decimals to get just those specific things. You can get just things beginning with specific letters, too, by giving a list of letters as left argument:

```
⎕CY'dfns'
'b' ⎕NL 4.2
```

```
bags
big
bsearch
bt
```

If you'd rather have a VTV (vector of text vectors, i.e. a list of strings), then use negative numbers. APLers often use this shortcut to list everything:

```
⎕CY'dfns'
10↑⎕NL-⍳9    ⍝ Truncated for display purposes; contains 300+ items...
```

```
 APLVersion  ActivateApp  Caption  ChildList  Cholesky  Coord  CursorObj  Cut  DDE
```

Expunge `⎕EX`

If you find that the name you want to use is unavailable, you may want to EXpunge its current value with `⎕EX`:

```
⎕NC'var' ⊣ ⎕EX 'var' ⊣ var←42
```

```
0
```

Skip to main content

There we created, removed, and enquired about the name `var`.

Shadow `⎕SHADOW`

If you only want to use an already used name temporarily, then you can use `⎕SHADOW` instead of `⎕EX`. The name will then be freed up for your use until the current function terminates. Note that shadowing happens automatically in dfns and dops when you just do regular assignments. In a dfn, `var←42` really means `⎕SHADOW 'var' ⋄ var←42`.

Be careful using `⎕SHADOW` though. It is much better to localise your variables in the function header by putting `;varName` at the end of the header.

State indicator `⎕SI`

Let's say you've built a bunch of functions that call each other, and then you run it, and it stops due to some bug. Now you need some situational awareness. You already know that `⎕NL` will let you check which names are defined, and `⎕NC` what type of things they are. `⎕SI` (State Indicator) will give you a list of function names on the stack:

```
foo←{goo ⍵}
goo←{moo ⍵}
moo←{⎕SI}
foo⍬
```

```
 moo  goo  foo
```

Line count `⎕LC`

`⎕LC` (Line Count) will give you a list of corresponding line numbers where each function in `⎕SI` is holding:

```
]dinput
foo←{
    goo ⍵
```

```
]dinput
goo←{


    moo ω
}
```

```
]dinput
moo←{

⎕LC}
```

```
foo ⍬
```

```
2 3 1
```

Size `⎕SIZE`

If you get a `WS FULL` error, you may want to check how much memory is being used to represent a variable. Use `⎕SIZE`:

```
nums←⍳100 100
⎕SIZE'nums'
```

```
480040
```

Workspace available `⎕WA`

You might also need to know how much [workspace available] (`⎕WA`) you have:

```
nums←⍳100 100
⎕SIZE'nums'
⎕WA
```

Skip to main content

```
480040
```

```
243415792
```

Screen dimensions `⎕SD`

`⎕SD` is the Screen Dimensions, which for a Jupyter kernel is something fairly arbitrary:

```
⎕SD
```

```
24 80
```

Regular expressions `⎕R` `⎕S`

`⎕R` and `⎕S` are Dyalog's regex operators; and take note that they are *operators*, not functions. Occasionally, their operator syntax has unexpected consequences, so it is important to remember this. They are dyadic operators. The left operand is always a character scalar, vector, or vector of such. The right operand may also be any of those, but can also be a function (any type; tacit, dfn or trad), and `⎕S` can also take an integer scalar or vector as right operand.

They then derive an ambivalent function which is can be named or applied to text. Some of their behaviour can be modified with the `⍠` operator, but since operators can only take functions (or arrays) as operands, `⍠` will be acting on the derived function, not on `⎕R` or `⎕S` themselves. This may sound trivial, but you have to remember that you cannot make a case insensitive (more about that later) version of `⎕S` with `MyRegexMachine←⎕S⍠1`, only `MyRegexMachine←'something'⎕S'something else'⍠1`.

Basic use

Final note before we really start: The regex flavour is PCRE, which is well documented, so we won't go too much into details about it. It is summarised here and described in detail here.

Skip to main content

`⎕R` (Replace) changes text in-place and returns the entire amended argument. `⎕S` only returns the amended match(es). In most other aspects, they are identical, so when we speak of one, it applies to the other unless otherwise noted.

OK, the basic example is:

```
'and' ⎕R 'or' ⊢ 'Programming Puzzles and Code Golf'  ⍝ Replace 'and' with 'or'
```

```
Programming Puzzles or Code Golf
```

However, the operands are not just simple text vectors, but rather regexes. For the left operand, that’s just regular PCRE to find a match, but the right argument uses something that very much feels like regex, but in fact is a Dyalog-invented notation to indicate what you want the match replaced by.

The first such notational symbol is `&` which means the match itself; in other words, no change:

```
'(.)\1' ⎕S '&' ⊢ 'Programming Puzzles and Code Golf'  ⍝ Match repeated pairs
```

```
  mm  zz
```

The left operand is just PCRE: `.` is any char, the parens is a *capture group*, which gives it a number, and `\1` is a reference to the first such group. It matches any sequence of two identical characters after each other.

A `%` in the right operand means the entire container (line or document) which contained the match:

```
'(.)\1' ⎕S '%' ⊢ 'Programming' 'Puzzles' 'and' 'Code' 'Golf'
```

```
  Programming  Puzzles
```

So this returned a list of all lines which contained double letters.

Skip to main content

The transformation string in depth

We've earlier talked about how simple APL's "string" (i.e. character vector) model is. The only special character is the quote which you need to double. There's no escaping, rather you have to use `…',(⎕UCS nn),'…`.

However, in the transformation string (that's what the right operand is called), you may also use some common escapes: `\n` and `\r` for newline and carriage return, and `\x{nn}` for any other Unicode character, where `nn` is in hex. Moreover, as `&` and `\` are special, you'll have to escape them too with a prefix backslash.

You may of course mix and match transformation strings as you please:

```
'(.)\1' ⎕S '"%" has "&"' ⊢ 'Programming' 'Puzzles' 'and' 'Code' 'Golf'
```

```
 "Programming" has "mm"  "Puzzles" has "zz"
```

You can also refer to the numbered capture groups with `\N` (or `\(NN)` for two-digit numbers):

```
'(.)\1' ⎕S '"%" has two "\1"s' ⊢ 'Programming' 'Puzzles' 'and' 'Code' 'Golf'
```

```
 "Programming" has two "m"s  "Puzzles" has two "z"s
```

Finally, you can fold to upper or lowercase by inserting `u` or `l` immediately after the backslash (adding a backslash to `&` and `%`):

```
'(.)\1' ⎕S '"\u%" has 2 "\u1"s' ⊢ 'Programming' 'Puzzles' 'and' 'Code' 'Golf'
```

```
 "PROGRAMMING" has 2 "M"s  "PUZZLES" has 2 "Z"s
```

This means that you can also use `⎕R` to just fold case (like `⎕C`):

```
'.'⎕R'\u&'⊢'Programming Puzzles and Code Golf'
```

```
PROGRAMMING PUZZLES AND CODE GOLF
```

In addition to using these text-based codes, `⎕S` can also use a few numeric codes which then return numeric results.

`0` is the offset from the start of the input of the start of the match:

```
'(.)\1'⎕S 0⊢'Programming Puzzles and Code Golf'
```

```
6 14
```

The above means that `mm` and `zz` begin 6 and 14 characters offset from the left. Notice that these are *offsets*, not indices, so they are as indices in origin 0 (`⎕IO←0`).

`1` is the length of the match:

```
'\w+' ⎕S 1 ⊢ 'Programming Puzzles and Code Golf' ⍝ Length of each word
```

```
11 7 3 4 4
```

`\w` is any word character, and `+` means one or more, so this matches whole words, and the result is a list of word lengths.

Question:

> Is there a way to get how many uppercased characters there are in a string?

You can e.g. match all uppercase letters and then tally the result:

```
≢'[[:upper:]]' ⎕S 0 ⊢ 'Programming Puzzles and Code Golf'   ⍝ POSIX character class
≢'[A-Z]' ⎕S 0 ⊢ 'Programming Puzzles and Code Golf'         ⍝ Ranged character clas
```

Skip to main content

```
4

4

4
```

`2` is the number of the block which had the match:

```
'(.)\1' ⎕S 2 ⊢ 'Programming' 'Puzzles' 'and' 'Code' 'Golf'
```

```
0 1
```

So we can see that only strings 0 and 1 had double-letters (again, always origin 0.)

Simultaneous patterns

The last one, `3`, is the pattern number, which brings us to an amazing feature of `⎕R` and `⎕S`: multiple simultaneous patterns:

```
'(.)\1' 'P' ⎕S 3 ⊢ 'Programming Puzzles and Code Golf'
```

```
1 0 1 0
```

Again, the patterns are numbered in origin 0, so first we find a double-letter (`mm`), then a `P`, then a double-letter (`zz`) and then a `P`. The amazing thing about the multiple patterns is that `⎕R` and `⎕S` step through the input letter by letter, and for each letter they look whether each pattern (from left to right) begins there.

You can of course also have multiple transformation patterns. This means that you can use a pattern to exclude from other patterns by placing the exclusion first, and replacing with the match (`&`):

```
' ' '\w' ⎕R (,¨'&' '_') ⊢ 'Programming Puzzles and Code Golf'
```

Skip to main content

```
___________ _______ ___ ____ ____
```

This replaced spaces with themselves, and word characters with underscores.

```
(,¨' ' '.') ⎕R (,¨'&' '_') ⊢ 'Programming Puzzles and Code Golf'
```

```
___________ _______ ___ ____ ____
```

But here, we replaced spaces with themselves, and then *any character* – including spaces – with underscores.

The vectorisation also works differently for numeric and text operands. Text goes pairwise, while numbers return the entire list for each. You can have one transformation string for each matching string, or a single transformation string for all the matching strings:

```
(,¨'aeiou') ⎕R (,¨'AEIOU') ⊢ 'Programming Puzzles and Code Golf'
(,¨'aeiou') ⎕R '_' ⊢ 'Programming Puzzles and Code Golf'
```

```
PrOgrAmmIng PUzzlEs And COdE GOlf
```

```
Pr_gr_mm_ng P_zzl_s _nd C_d_ G_lf
```

But of course, you can't have multiple transformation strings for a single matching string:

```
'o'⎕R(,¨'AEIOU')⊢'Programming Puzzles and Code Golf'  ⍝ LENGTH ERROR
```

```
LENGTH ERROR: Invalid transformation format
      'o'⎕R(,¨'AEIOU')⊢'Programming Puzzles and Code Golf'  ⍝ LENGTH ERROR
         ^
```

Variants

We mentioned earlier that you can use *variant*, `⍠`. The most commonly used option is case

Skip to main content

`⍠'IC' 1` (Insensitive Case); `⍠1` is enough:

```
'g'⎕R'_'⍠1⊢'Programming Puzzles and Code Golf'
```

```
Pro_rammin_ Puzzles and Code _olf
```

Notice that `g` matched both upper and lowercase Gs.

Another cool option is for `⎕S` only: `⍠'OM' 1` (Overlapping Matches):

```
'[^aeiou]{3}'⎕S'&'⊢'Programming Puzzles and Code Golf'  ⍝ Non-overlapping matches
```

```
 ng   zzl  nd
```

`[^aeiou]` is a *negated character group*, which means NOT any of these letters and `{3}` means exactly three of such.

```
'[^aeiou]{3}'⎕S'&'⍠'OM'1⊢'Programming Puzzles and Code Golf' ⍝ Overlapping matches
```

```
 ng   g P  zzl  nd   d C
```

Notice how this matched `g P` even though its first two letters were already found in the first match. `⎕R` cannot allow overlapping matches because that may lead to infinite substitution looping: `'x' ⎕R 'xx'⍠'OM' 1` would loop forever. In `xyz` it would first replace `x` with `xx` to get `xxyz` then continue at the next character, which also matches, and makes `xxxyz`, etc.

Function operand

Arguably the most powerful feature of them all is the fact that the right operand may be any monadic (or ambivalent) function. The right argument (which may of course be ignored) will be a namespace with a few members. This namespace survives between matches for the entire time that the current `⎕R/⎕S` call is ongoing, so you further populate the namespace and so use it to convey information from earlier matches to later matches. The only names that are reserved (i.e.

Skip to main content

- `Block` – same as `%`
- `BlockNum` – same as `2`
- `Pattern` – the literal pattern which matched (i.e. not the match itself)
- `PatternNum` – the origin `0` number of the above
- `Match` – same as `&`
- `Offsets` – first element is same as `0` but has additional elements corresponding to capture groups
- `Lengths` – first element is same as `1` but has additional elements corresponding to capture groups
- `ReplaceMode` – `0` for `⎕S` and `1` for `⎕R`
- `TextOnly` – Boolean whether the result of the function must be a character vector (i.e. for `⎕R`) or can be anything (i.e. for `⎕S`).

The function can then do any computation necessary to determine its result, so you could even have it prompt the user for whether to replace this match or not (i.e. when implementing a “Replace All” button in an editor). This of course renders `⎕R` and `⎕S` as powerful as Dyalog APL as a whole – they are both supersets and subsets of Dyalog APL!

Primitive operators

Operators take *operands*, which may be functions, and derive a function. You can think of APL’s operators as higher-order functions.

Reduce `/` `⌿`

The first operator is `/`, called reduce. It is a monadic operator which derives an ambivalent function. An ambivalent function is one which can be called either monadically or dyadically. For example, `-` is ambivalent. Monadically, it is negate; dyadically, it is subtraction.

`+/` is a derived ambivalent function. The monadic function is plus-reduction (i.e. sum) and the dyadic function is windowed sum, as in sliding windows of size `⍺` (shorthand for “left argument”).

Skip to main content

```
+/3 1 4 1 5
2 +/ 3 1 4 1 5
3 +/ 3 1 4 1 5
```

```
14
```

```
4 5 5 6
```

```
8 6 10
```

Question:

> What does `3-/` do? Subtraction isn't associative.

```
3-/ 1 2 3 4 5
```

```
2 3 4
```

As functions in APL are right-associative, `-/⍵` (this is a shorthand which means the monadic form of `-/`) is alternating sum.

```
1 - (2 - 3)
```

```
2
```

`f/⍵` is called *reduce* because it reduces the rank of its argument by 1. For example, if we apply it to a matrix, we'll get back a vector, even if the function we provide does not "combine" its arguments.

```
{'(',⍺,⍵,')'}/'Hello'
```

```
 (H(e(l(lo))))
```

Skip to main content

Here, the function we gave concatenates its arguments and parentheses. With output boxing turned on, it is clear to see that there is a space in front of the leftmost `(`. Without output boxing, that space is still there, but you may have to look a bit more carefully in order to notice it. This is APLs way to indicate that the array (a character vector) is enclosed. In other words, it returned `⊂'(H(e(l(lo))))'`.

```
(⊂'(H(e(l(lo))))') ≡ {'(',⍺,⍵,')'}/'Hello'
```

```
1
```

We can also apply reductions to higher-rank arrays:

```
3 4⍴⍳12
+/3 4⍴⍳12
```

```
1  2  3  4
5  6  7  8
9 10 11 12
```

```
10 26 42
```

Notice how the rank went down from 2 to 1 (i.e. matrix to vector). Reductions lower the rank. `N f/` is called N-wise reduce, and does not lower the rank. Notice that `/` goes along the trailing axis, i.e. the it reduced the rows of the matrix. It has a twin, `⌿`, which goes along the first axis, i.e. the columns of a matrix.

```
+⌿3 4⍴⍳12
```

```
15 18 21 24
```

If you have higher-rank arrays, you can reduce along any axis with a bracket axis specification:

```
2 3 4⍴⍳24
(+⌿2 3 4⍴⍳24)(+/[2]2 3 4⍴⍳24)(+/2 3 4⍴⍳24)
```

Skip to main content

```
  1  2  3  4
  5  6  7  8
  9 10 11 12

 13 14 15 16
 17 18 19 20
 21 22 23 24


  14 16 18 20  15 18 21 24  10 26 42
  22 24 26 28  51 54 57 60  58 74 90
  30 32 34 36
```

Note that `f/[1]` is the same thing as `f⌿`.

Scan `\` `⍀`

While `/` is reduction, `\` is cumulative reduction, known as scan:

```
+\3 1 4 1 5
```

```
3 4 8 9 14
```

`/`'s cousin `\` of course has a twin, too; `⍀`, behaving analogously.

Each `¨`

The next operator is `¨` which is called each for a good reason. `f¨⍵` applies the function `f` monadically to each element of `⍵`. `⍺ f¨ ⍵` applies `f` between the paired-up elements of `⍺` and `⍵`.

```
1 2 3 , 4 5 6
1 2 3 ,¨ 4 5 6
1 2 3 ,¨ (10 20)(30 40)(5 6)
```

Skip to main content

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

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

```
 1 10 20  2 30 40  3 5 6
```

Most arithmetic functions are “scalar” meaning they penetrate to the very leaves of the arrays. `¨` is meaningless for scalar functions.

```
3+¨3 1 4 1 5 9 2 6 5  ⍝ works; but pointless
3+3 1 4 1 5 9 2 6 5   ⍝ scalar function + is pervasive
```

```
6 4 7 4 8 12 5 9 8
```

```
6 4 7 4 8 12 5 9 8
```

Power `⍣`

`⍣` is the power operator. `f⍣n` applies the function `f` n times.

```
2×3
2×2×3
2×2×2×3
2(×⍣3)3
```

```
6
```

```
12
```

```
24
```

```
24
```

It did the multiplication 3 times. We need parentheses here to separate the two 3s to make sure the

Skip to main content

above, `α (×⍣3) ω` therefore is `α × α × α × ω`. Operators are never ambivalent. Their derived functions can be, but they are either monadic or dyadic. `⍣` is dyadic. `/` and `¨` are monadic. The result of `×⍣3` is a new function which takes arrays as arguments.

`f⍣≡` is the fixpoint of `f`.

```
0.5×⍣≡1
```

```
0
```

If you keep halving 1 you end up with 0. `0.5×⍣≡` means keep multiplying 0.5 with the argument until it stops changing. The power operator can take a custom right function operator, too. See the documentation.

Commute `f ⍨`

Commute, `⍨`, is a monadic operator taking a dyadic function and deriving an ambivalent function. `α f⍨ ω` is `ω f α`. `f⍨ ω` is `ω f ω`. We sometimes informally refer to `⍨` as “selfie” when the derived function is used monadically, because that’s what it does, and it looks like a selfie (photo) too. `⍨` seems very simple, but it has some neat applications. Monadic `+⍨` is double. Monadic `×⍨` is square.

Constant `A ⍨`

With an array operator, `⍨` is constant. It always returns the operator array. It might not be immediately obvious when this is useful. Consider the following examples:

```
neg←{-@(α⍨)ω}
1 0 1 neg ⍳3
```

```
¯1 2 ¯3
```

Here we have a function that uses a Boolean left argument to indicate where to apply negation in

Skip to main content

cumbersome, for example:

```
1 0 1 {a←⍺⋄-@{a}⍵} ⍳3
```

```
¯1 2 ¯3
```

We could use `⍸` to expand the left argument into indices, but that introduces an unnecessary inefficiency we avoid when using *constant*:

```
1 0 1 {-@(⍸⍺)⊢⍵} ⍳3
```

```
¯1 2 ¯3
```

Another usecase is when you want to use one array's structure as a model, but use a particular element instead:

```
(?3)⍨¨'this is a string'
```

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

The alternatives, again, tend to be either more cumbersome, or inefficient

```
(⍴'this is a string')⍴?3
```

```
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

Note that we can't do

```
{?3}¨'this is a string'  ⍝ Not the same thing!
```

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

Skip to main content

Beside/atop `∘`

The Beside operator, `∘`. `∘` comes from function composition, like how `f(g(x))` can be written `f∘g(x)` in mathematics. So, too, in APL, if `f` and `g` are functions, then `f∘g x` is the same as `f g x` (APL doesn't need parentheses for function application). This alone is, of course, not very interesting. However, APL also has dyadic (infix) functions: `A f∘g B` is `A f g B`.

Both of these are very important when writing tacit APL code. For example, if we want to write a function which adds its left argument to the reciprocal (monadic `÷`) of its right argument, it can be written as `f ← +∘÷`.

The golden ratio (phi) can be calculated with the continued fraction

$$\phi = 1 + \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{\ddots}}}}$$

So ϕ is `1+÷1+÷1+÷…`. We can insert the same function between elements of a list with the `/` operator, for example,

```
+/1 1 2 3
```

```
7
```

In our case, we want to insert `…+÷…`, but that isn't a single function. However, we can use `+∘÷`:

```
+∘÷/1 1 1 1 1 1 1 1 1 1
```

```
1.618181818
```

`X⍴Y` reshapes `Y` into shape `X`:

```
+∘÷/1000⍴1  ⍝ A good approximation of phi.
```

Skip to main content

```
1.618033989
```

`∘` also allows you to compose with argumnents. `g←f∘A` where `f` is a dyadic function and `A` an array (any data) gives `g`, a new monadic function which calculates `x f A`. Similarly, `g←A∘f` makes `g` a function which calculates `A f x`.

For *operators* you can “curry” their right operand. So `WithTwo←∘2` is a new monadic operator which can in turn modify a dyadic function to become monadic (using 2 as its right argument). For example, `+ WithTwo 3` will give 5.

```
WithTwo←∘2
+ WithTwo 3
```

```
5
```

This is especially useful with the `f⍣n` power operator which applies its `f` operand function `n` times. `twice←⍣2` is an operator which applies a function twice. For example, `2+twice 3` is 7.

```
twice←⍣2
2+twice 3
```

```
7
```

`inv←⍣¯1` is an operator which will apply a function -1 times, i.e. applies the inverse of that function.

Question:

> Do all functions have inverses?

No, but surprisingly many do. If you derive new functions tacitly using only operators and invertible functions, then the resulting function can also (generally) be inverted automatically. Even structural functions can be inverted:

```
('x'∘ ⍣¯1) 'x'∘  'abc'
```

Skip to main content

```
abc
```

So what happened there is that we applied the function `'x'∘`, which prepends the letter `x`, and then we applied its inverse, which removes an `x` from the left side. The specific function `'x'∘`, is not hardcoded. Instead the interpreter has a bunch of rules which lets it determine the inverse of various compositions.

That's really all there is to say about `∘`. However, a warning is in place: `(f g)Y` is the same as `f∘g Y` which may fool you into thinking that `X(f g)Y` is the same as `X f∘g Y`. However, they are not the same!

A nice golfing trick using `∘` is having the left operand be `⊢`. This allows using a monadic function on the right argument while ignoring the left argument.

At `@`

The at operator, `@`, does exactly what it says. What's on its left gets done at the position indicated by its right operand.

```
('X'@2 5) 'Hello'
```

```
HXllX
```

So we put an `X` at positions 2 and 5 (APL is 1-indexed by default – you can change to 0-indexing if you want). We can also give an array which matches the selected elements:

```
('XY'@2 5) 'Hello'
```

```
HXllY
```

So far, we've only used `@` to substitute elements. We can also use it to modify them:

```
(-@2 5)10 20 30 40 50 60
```

Skip to main content

```
10 ¯20 30 40 ¯50 60
```

Here we applied the monadic function `-` (negate) at positions 2 and 5. We can do the same with a dyadic function, too:

```
7(+@2 5)10 20 30 40 50 60
```

```
10 27 30 40 57 60
```

So far, we have been using an array right operand. If we use a function right operand it gets applied to the right argument, and the result must be a Boolean mask instead of a list of indices.

```
⎕A ⍝ uppercase alphabet
'x'@(∊∘⎕A)'Hello World'
```

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
xello xorld
```

`∊` is membership, so the derived function `∊∘⎕A` gives a Boolean for where elements of the right (and only) argument are members of the uppercase alphabet:

```
(∊∘⎕A)'Hello World'
```

```
1 0 0 0 0 0 1 0 0 0 0
```

which is then used as mask by `@` to determine where to substitute with `x`. See, for example Goto the Nth Page which uses `@` twice.

I-beam `⌶`

I-beam, `⌶`, is a special monadic operator (although it follows normal APL syntax) which uses a

Skip to main content

Note that although documentation is provided for `⌶` functions, any service provided this way should be considered as “experimental” and subject to change – without notice – from one release to the next.

One example is Format Date-Time, `1200⌶`, which formats *Dyalog Date Numbers* according to a set of pattern rules.

```
'%ISO%'(1200⌶)1⎕DT'J'
```

```
2024-03-05T10:03:28
```

Stencil `⌺`

Next up is stencil (as in stencil code), `⌺`. The symbol is supposed to evoke the picture of a stencil over a paper. Stencil is useful for Game of Life and related problems. It is a dyadic operator which derives a monadic function. The left operand must be a function and the right operand must be an array.

The right operand specifies what neighbourhoods to apply to. For example, in Game of Life, the neighbourhoods are 3-by-3 sub-matrices centred on each element in the input array. The operand gets called dyadically. The right argument is a neighbourhood and the left is information about whether he neighbourhood overlaps an edge of the original argument world.

To see how it works, we’ll use `{⊂⍵}` as left operand. It just encloses the neighbourhood so we can see it. As right operand we use `3 3`, i.e. the neighbourhood size:

```
4 6⍴⎕A ⍝ our argument
({⊂⍵}⌺3 3) 4 6⍴⎕A
```

Skip to main content

```
ABCDEF
GHIJKL
MNOPQR
STUVWX
```

```
  AB  ABC  BCD  CDE  DEF  EF
  GH  GHI  HIJ  IJK  JKL  KL
  AB  ABC  BCD  CDE  DEF  EF
  GH  GHI  HIJ  IJK  JKL  KL
  MN  MNO  NOP  OPQ  PQR  QR
  GH  GHI  HIJ  IJK  JKL  KL
  MN  MNO  NOP  OPQ  PQR  QR
  ST  STU  TUV  UVW  VWX  WX
  MN  MNO  NOP  OPQ  PQR  QR
  ST  STU  TUV  UVW  VWX  WX
```

Here you see that we returned a 4-by-6 matrix of neighbourhoods. Notice that all the neighbourhoods are 3-by-3, even at the edges. They were padded with spaces.

The padding was done sometimes on top, sometimes on left, sometimes on right, and sometimes on the bottom. The information about that is in the left argument (`⍺`) of the operand function:

```
({⊂⍺}⌺3 3) 4 6⍴⎕A
```

```
 1 1   1 0   1 0   1 0   1 0   1 ¯1
 0 1   0 0   0 0   0 0   0 0   0 ¯1
 0 1   0 0   0 0   0 0   0 0   0 ¯1
 ¯1 1  ¯1 0  ¯1 0  ¯1 0  ¯1 0  ¯1 ¯1
```

Each cell contains two elements, one for rows, and one for columns. Positive indicates left/top. Negative is right/bottom. The magnitude indicates how many rows/columns were padded.

This fits nicely with the dyadic `↓` drop primitive, which takes the number of rows,columns as left argument to drop from the right argument:

```
({⊂⍺↓⍵}⌺3 3) 4 6⍴⎕A
```

Skip to main content

```
AB   ABC   BCD   CDE   DEF   EF
GH   GHI   HIJ   IJK   JKL   KL
AB   ABC   BCD   CDE   DEF   EF
GH   GHI   HIJ   IJK   JKL   KL
MN   MNO   NOP   OPQ   PQR   QR
GH   GHI   HIJ   IJK   JKL   KL
MN   MNO   NOP   OPQ   PQR   QR
ST   STU   TUV   UVW   VWX   WX
MN   MNO   NOP   OPQ   PQR   QR
ST   STU   TUV   UVW   VWX   WX
```

As you can see, the padding was removed.

Another example. Here you can see that on the far left and right, we have to pad two columns to get a 5-wide neighbourhood centred on the first column:

```
({⊂⍺}⌺3 5) 4 6⍴⎕A
```

```
 1 2    1 1    1 0    1 0    1 ¯1    1 ¯2
 0 2    0 1    0 0    0 0    0 ¯1    0 ¯2
 0 2    0 1    0 0    0 0    0 ¯1    0 ¯2
¯1 2   ¯1 1   ¯1 0   ¯1 0   ¯1 ¯1   ¯1 ¯2
```

Now, let’s try implementing *Game of Life*.

Here are the rules:

- A cell will stay alive with 2 or 3 neighbours.
- It will become alive with 3 neighbours.
- It will die with fewer than 2 or more than 3 neighbours.

Let’s make a world:

```
4 5⍴0 0 1 0 0 1 0
```

```
0 0 1 0 0
1 0 0 0 1
0 0 1 0 0
0 1 0 0 1
```

Skip to main content

The 1s indicate live cells while the 0s indicate dead cells. Let’s look at our neighbourhoods:

```
({⊂⍵}⌺3 3) 4 5⍴0 0 1 0 0 1 0
```

```
 0 0 0  0 0 0  0 0 0  0 0 0  0 0 0
 0 0 0  0 0 1  0 1 0  1 0 0  0 0 0
 0 1 0  1 0 0  0 0 0  0 0 1  0 1 0
 0 0 0  0 0 1  0 1 0  1 0 0  0 0 0
 0 1 0  1 0 0  0 0 0  0 0 1  0 1 0
 0 0 0  0 0 1  0 1 0  1 0 0  0 0 0
 0 1 0  1 0 0  0 0 0  0 0 1  0 1 0
 0 0 0  0 0 1  0 1 0  1 0 0  0 0 0
 0 0 1  0 1 0  1 0 0  0 0 1  0 1 0
 0 0 0  0 0 1  0 1 0  1 0 0  0 0 0
 0 0 1  0 1 0  1 0 0  0 0 1  0 1 0
 0 0 0  0 0 0  0 0 0  0 0 0  0 0 0
```

We can get the number of neighbours by summing. So we make a list with ravel, `,`, and use `+/` to sum:

```
({+/,⍵}⌺3 3) 4 5⍴0 0 1 0 0 1 0
```

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

We also need to know what the current value is. That is the 5th value in the ravelled neighbourhood:

```
({5⌷,⍵}⌺3 3) 4 5⍴0 0 1 0 0 1 0
```

```
0 0 1 0 0
1 0 0 0 1
0 0 1 0 0
0 1 0 0 1
```

Now we can say that `self←5⌷,⍵` and `total←+/,⍵`:

Skip to main content

```
({self←5⌷,⍵ ⋄ total←+/,⍵ ⋄ ⊂self total}⌺3 3) 4 5⍴0 0 1 0 0 1 0
```

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

Here we have the self and the total for each cell.

The logic is that in the next generation the cell is alive if itself was alive and had 2–3 neighbours (3 or 4 total, including self), or if it was dead and had 3 neighbours. That is

```
(self ∧ (total∊3 4)) ∨ ((~self) ∧ (total=3))
```

Let’s plug that in:

```
({self←5⌷,⍵ ⋄ total←+/,⍵ ⋄ (self ∧ (total∊3 4)) ∨ ((~self) ∧ (total=3))}⌺3 3) 4 5⍴0
```

```
0 0 0 0 0
0 1 0 1 0
0 1 0 1 0
0 0 0 0 0
```

This can be shortened considerably, if we so wished. For a detailed walk-though of the shortest possible *Game of Life* using stencil, see the webinar on dyalog.tv.

`⌺` can do a further trick, too. If the right operand is a matrix, then the second row indicates the step size. By default it is 1 in every dimension. Consider the following:

```
({⊂⍵}⌺(2 2⍴3))7 7⍴⎕A
```

Skip to main content

```
 AB  CDE  FG
 HI  JKL  MN
 OP  QRS  TU
 VW  XYZ  AB
 CD  EFG  HI
 JK  LMN  OP
 QR  STU  VW
```

Here we used a 2-by-2 matrix of all 3s. In other words, we get 3-by-3 neighbourhoods going over 3 rows and 3 columns. Thus, we “chop” the argument, with no overlaps. We can also use even sizes, in which case every “space” between elements (rather than elements themselves) gets to be the centre of a neighbourhood:

```
({⊂⍵}⌺(2 2⍴2))6 6⍴⎕A
({⊂⍵}⌺(2 4))4 6⍴⎕A
```

```
AB  CD  EF
GH  IJ  KL
MN  OP  QR
ST  UV  WX
YZ  AB  CD
EF  GH  IJ
```

```
 ABC  ABCD  BCDE  CDEF  DEF
 GHI  GHIJ  HIJK  IJKL  JKL
 GHI  GHIJ  HIJK  IJKL  JKL
 MNO  MNOP  NOPQ  OPQR  PQR
 MNO  MNOP  NOPQ  OPQR  PQR
 STU  STUV  TUVW  UVWX  VWX
```

Key `⌸`

`⌸` is key, a monadic operator deriving an ambivalent function (i.e. monadic or dyadic depending on usage). The lone operand must be a function, and it gets called dyadically in a manner not too different from stencil’s left operand.

Let’s do the monadic derived function first, i.e. `(f⌸) data`.

Skip to main content

Key will group identical major cells of the data together and call the operand `f` with the unique element as left argument, and the indices of that element in the data as right argument:

```
{⊂⍺⍵}⌸'Mississippi'
```

```
  M 1    i 2 5 8 11    s 3 4 6 7    p 9 10
```

This tells us that “M” is at index 1, “i” at 2 5 8 11, etc. It is very common to use `≢` to tally the indices:

```
{⍺,≢⍵}⌸'Mississippi'
```

```
M 1
i 4
s 4
p 2
```

which gives us the count of each unique element. We can, for example, use this to remove elements which only occur once. We first use key to make a Boolean vector for each unique element:

```
{1≠≢⍵}⌸'Mississippi'
```

```
0 1 1 1
```

Monadic `∪` gives us the unique elements:

```
∪'Mississippi'
```

```
Misp
```

We can use `/` to filter one by the other:

Skip to main content

```
0 1 1 1/'Misp'
```

```
isp
```

Putting it all together, we get

```
{({1≠≢⍵}⌸⍵)/∪⍵}'Mississippi' ⍝ Unique elements occurring more than once
```

```
isp
```

`⌸` works on higher rank arrays, too (matrices, 3D blocks, etc.), where it will use the major cells (rows for matrices, layers for 3D blocks…) as “items”.

```
5 3⍴'AAAABCAAAABBAAA'
{⍺⍵}⌸ 5 3⍴'AAAABCAAAABBAAA'
```

```
AAA
ABC
AAA
ABB
AAA
```

```
 AAA  1 3 5
 ABC  2
 ABB  4
```

Dyadic *key* then. Behold:

```
'Mississippi' {⊂⍺⍵}⌸ ⍳11
'Mississippi' {⊂⍺⍵}⌸ 'ABCDEFGHIJK'
```

```
 M 1   i 2 5 8 11   s 3 4 6 7   p 9 10
```

```
 M A   i BEHK   s CDFG   p IJ
```

Skip to main content

Instead of returning the indices of the unique elements (of the right – and only – argument), it returns the elements of the right *corresponding* to the unique elements of the left.

Atop `f⍤g`

Atop has been assigned `function⍤function`, thus sharing the symbol with the rank operator's `function⍤array`. You should be familiar with the 2-train, which is also called "atop": `(f g)Y` and `X(f g)Y`. Maybe you've even been burned by `f∘g Y` being an atop, but `X f∘g Y` *not* being an atop. Well, the atop operator is what you would expect, i.e. `f⍤g Y` is exactly like `f∘g Y` but `X f⍤g Y` is `f X g Y` or `X (f g) Y`. We strongly recommend transitioning to use `⍤` in places where you've hitherto used monadic `f∘g`: it will prevent (at least one potential cause of) frustration should you ever decide to add a left argument to your code.

Let's say you define a function that returns the magnitude of reciprocal:

```
|∘÷ ¯4
|∘÷ ¯5
```

```
0.25
```

```
0.2
```

(This could be written without the `∘`, but this is a very simple function useful for illustration purposes.)

Now you get a feature request that the function should take a left argument which is a numerator (instead of the default 1).

```
2 |∘÷ ¯4
2 |∘÷ ¯5
```

```
1.75
```

```
1.8
```

Skip to main content

```
|⍤÷ ¯4
|⍤÷ ¯5
2 |⍤÷ ¯4
2 |⍤÷ ¯5
```

```
0.25
```

```
0.2
```

```
0.5
```

```
0.4
```

One way to look at `f∘g` vs `f⍤g` is that, when given a left argument, `∘` gives it to the left-hand function and `⍤` gives it to the right-hand function. Other than that, they are equivalent. Another way to look at `f∘g` vs `f⍤g` is simply choosing order of the first two tokens in the equivalent explicit expression: `X f∘g Y` computes `X f g Y` and `X f⍤g Y` computes `f X g Y`. So we're simply swapping `X` and `f`.

Then there's the classic problem with slashes, especially in tacit programming. If you've ever tried using replicate/compress in a train, you'll have bumped into the fact that slashes prefer being operators over being functions. This means that `{(5<⍵)/⍵}` doesn't convert to `(5<⊢)/⊢`.

While it may not be obvious at first sight, if we define `f←5<⊢` it might become clearer that `f/⊢` isn't at all what we want. Now, there's an axiom in APL that an operator cannot be an operand. (Shh, don't mention `∘.f`). This means that if a slash ends up in a situation where it has to be an operand, it will resort to being a function. You may even have noticed that constructs like `⊢(/⍨)5<⊢` work fine, though `⊢/⍨5<⊢` doesn't. This is because the `/` in isolation with the `⍨` is forced to become the operand of `⍨`. But since operators bind from the left, `⊢/` binds first, and so `⊢/⍨5<⊢` becomes `(⊢/)⍨5<⊢` or `(5<⊢)⊢/(5<⊢)` which is usually not what you want.

So, `⍤` to the rescue. If `⍤` (or any dyadic operator) is found to the immediate left of a slash, then clearly the dyadic operator cannot be the operand of the slash, `⍤` being a dyadic operator itself, and it can't be part of the function on the left, since it requires a right-operand, too. Therefore, the slash is forced to become a function. So `-⍤/` is the negation of the replicate:

Skip to main content

```
¯10 ¯30 ¯30
```

It is easy to think then that “oh, this is an atop, so I should be able to do this with parentheses too; `(f g)`” but that’d be a mistake: `(-/)` is just a normal minus-reduction. Since `f⍤g` is “atop” and `(f g)` is “atop”, you might think they are interchangeable.

Another mistake is to think: “if a slash is an operand, it’ll be a function” and then think that `/∘⊢` would work like `⊢⍤/` by pre-processing the right argument with a no-op rather than post-processing the result with a no-op.

Let’s say we have a two-element vector of a mask and some data, and you want to “apply” the mask to the data… Perhaps your instinct is to try

```
apply ← //
```

but that gives an enclosed result, which isn’t what we want:

```
]display apply (1 0 1)'abc'
```

```
┌────┐
│┌→─┐│
││ac││
│└──┘│
└∊───┘
```

Instead, we can do something like

```
apply ← ⊃⊢⍤//
]display apply (1 0 1)'abc'
```

```
┌→─┐
│ac│
└──┘
```

In fact, once `⊢⍤/` becomes a common pattern, you can actually help the reader of your code by using `⊢⍤/` so they don’t have to consider if your slash is Replicate or Reduce. For example, if your

Skip to main content

code says `z←x/y` it might not be obvious what's going on. If you instead write `z←x⊢⍤/y` your reader knows exactly what you're doing.

Another example. Given a string, replace every character with two copies of itself prefixed and suffixed by a space. For example, `'abc'` becomes `' aa  bb  cc '`. Yes, you can do this with regex. Please don't.

```
(⊢ ⊢⍤⍀ ⍨0 2 0⍴⍨3×≢)'abc'
```

```
 aa  bb  cc 
```

Over `f⍥g`

Recall how `f∘g` preprocesses the right argument of `f` using `g`. One way to look at Over is simply as preprocessing all arguments of `f` using `g`. All, as in both or the only. So again `f⍥g Y` is the same as `f⍥g Y` and `f∘g Y`. The difference is, again, when we do a dyadic application. While `X f∘g Y` is `X f(g Y)` we have `X f⍥g Y` be `(g X)f(g Y)`. This may seem like an overly involved operator, but really, the pattern of preprocessing both arguments comes up a lot. Once you start looking for it, you'll see it all over.

For example, a dyadic function computing the sum of absolute values of its arguments:

```
¯1 ¯2 3 4 +⍥| 2 3 ¯8 5
```

```
3 5 11 9
```

Given arguments which are vectors, which one has the smallest maximum? Return `¯1` if the left argument has the smallest maximum, `1` if the right one has, or `0` if they are equal.

```
¯1 ¯2 3 4 ×⍤-⍥(⌈/) 2 3 ¯8 5
```

```
¯1
```

Skip to main content

Beautiful use of both Atop and Over. You can, of course, omit the `⍤` here, unless used inline. OK, how about this: write an alternative to replicate which can take arguments of equal shape, both with rank greater than 1, and replicates the corresponding elements. Since the result might otherwise be ragged, you have to return a vector.

```
      (2 3⍴⍳6) YourFunction 2 3⍴⎕A
ABBCCCDDDDEEEEEFFFFFF
```

```
(2 3⍴⍳6) ⊢⍤/⍥, 2 3⍴⎕A
```

```
ABBCCCDDDDEEEEEFFFFFF
```

Also, in this case, you don't *need* `⊢⍤`, but it is good for clarity, and necessary if used inline in a train. A golfing tip regarding `⍥`: you can sometimes use it to pre-process the left argument, when it is a no-op on the right. For example, `1≡⍥,≡,⍴`, only ravels the left argument, since the right argument already is a vector.

Deep dives

In this chapter, we'll dive deeper into some of the more complex functions and operators, showing how they are used in practice.

Rank in depth `⍤`

Rank, `⍤`, is a dyadic operator which takes a function on its left, and on the right it takes a specification of which sub-arrays we want to apply that function to.

Simple usage of `⍤` is specifying which rank subcells we want a function to apply to, and for dyadic usage, which subcells of the left argument should be paired up with which subcells of the right argument. Let's say we have the vector `'ab'` and the matrix `3 4⍴⍳12`. We want to prepend 'ab' only the beginning of every row in the matrix:

```
(3 2⍴'ab'),(3 4⍴⍳12)
```

Skip to main content

```
ab 1  2  3  4
ab 5  6  7  8
ab 9 10 11 12
```

But here, we did so by reshaping `'ab'` until it became big enough to cover all rows. How do we do this *without* reshaping `'ab'`, just using `⍤`?

```
'ab',⍤1⊢3 4⍴⍳12
```

```
ab 1  2  3  4
ab 5  6  7  8
ab 9 10 11 12
```

Here we treat `'ab'` as a cell and prepend it to every row of `3 4⍴⍳12`. Let's say instead we have a 3D array, and we want to put a single character from `'ab'` on each row:

```
⊢arr←2 2 4⍴⍳16
(2 2⍴'ab'),arr
```

```
 1  2  3  4
 5  6  7  8

 9 10 11 12
13 14 15 16


a  1  2  3  4
b  5  6  7  8

a  9 10 11 12
b 13 14 15 16
```

We can also do the same with rank, pairing up `'ab'` with a *matrix*, not a row. When we concatenate a vector with a matrix, the vector becomes a new column:

```
'ab',⍤2⊢2 2 4⍴⍳16
```

Skip to main content

```
a   1   2   3   4
b   5   6   7   8

a   9  10  11  12
b  13  14  15  16
```

Now consider `'ABCD'` and the following matrix:

```
2 4⍴⍳8
```

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

We want to produce the following,

```
2 4 2⍴'A'1'B'2'C'3'D'4'A'5'B'6'C'7'D'8
```

```
A 1
B 2
C 3
D 4

A 5
B 6
C 7
D 8
```

We can see that each layer is each letter of the character vector paired up with each digit, each row in turn. So, for the first row of the matrix, we want:

```
'ABCD',⍤0⊢1 2 3 4
```

```
A 1
B 2
C 3
D 4
```

We now want to apply this process for each of the rows. “For each row” is just `⍤1`, and, yes, we

Skip to main content

```
'ABCD',⍤0⍤1⊢2 4⍴⍳8
```

```
A 1
B 2
C 3
D 4

A 5
B 6
C 7
D 8
```

Here is another example. Let's say we're constructing a lunch menu card. We have three "fillings" and four "containers". We want to pair up all combinations of fillings and containers, thereby adding a trailing axis of length 2, so we get a rank 3 result:

```
↑'beef' 'fish' 'veggie'∘.{⍺⍵}'sandwich' 'patties' 'platter' 'wrap'
```

```
 beef    sandwich
 beef    patties
 beef    platter
 beef    wrap

 fish    sandwich
 fish    patties
 fish    platter
 fish    wrap

 veggie  sandwich
 veggie  patties
 veggie  platter
 veggie  wrap
```

Following the reasoning above, we can achieve the same thing with rank, using:

```
'beef' 'fish' 'veggie',⍤0⍤0 1⊢'sandwich' 'patties' 'platter' 'wrap'
```

Skip to main content

```
beef    sandwich
beef    patties
beef    platter
beef    wrap

fish    sandwich
fish    patties
fish    platter
fish    wrap

veggie  sandwich
veggie  patties
veggie  platter
veggie  wrap
```

We take each single item from the left argument, and whole right argument, which is `⍤0 1`, and then each single left, with each single right, which is `⍤0 0` (or just `⍤0`). The inner application is the single-single, so it needs to be closest to the function `,`.

Also, remember that `⍤` will not open your enclosures. It always operates on the elements of your arrays.

Time for another example. How can we swap the arguments to outer product just using rank (so no `⍨` or `⍉`)? In other words, go from this:

```
1 2 3∘.×1 2 3 4 5
```

```
1 2 3  4  5
2 4 6  8 10
3 6 9 12 15
```

to this:

```
1 2 3 4 5∘.×1 2 3
```

```
1  2  3
2  4  6
3  6  9
4  8 12
5 10 15
```

Skip to main content

The first thing to note is that we can express the starting product as "each element to the left times the whole thing on the right":

```
1 2 3×⍤0 1⊢1 2 3 4 5
```

```
1 2 3  4  5
2 4 6  8 10
3 6 9 12 15
```

The reversed argument order then becomes "the whole thing on the left times each element to the right", or simply the reversed rank:

```
1 2 3×⍤1 0⊢1 2 3 4 5
```

```
1  2  3
2  4  6
3  6  9
4  8 12
5 10 15
```

A really useful function (let's call it "Sane Indexing" or "Select") is to select the major cells of the right argument as indexed by the left argument. For example, `2 3 1 2 Select 'abcdef'` would give `'bcab'`. Squad indexing, `⌷`, only lets you choose a *single* major cell. Can we define `Select` in terms of `⌷` with the help or rank? We need to pair each element from the left argument with the whole of the right argument, whatever rank it may be:

```
Select←⌷⍤0 99
```

```
2 3 1 2 Select 'abcdef'
```

```
bcab
```

We could, in fact, have used any number greater than Dyalog's max rank (15) to represent the full rank of the argument, but 99 has come to be used for this purpose. It is actually fairly common to specify a negative number, which means that the target rank is that number subtracted from the argument rank. So `f⍤¯1 ¯2` is the same as

```
{⍺ f⍤(¯1+≢⍴⍺)(¯2+≢⍴⍵)⊢⍵}
```

You can also mix-and-match positive and negative ranks.

Power in depth: `f⍣k`

When the power operator, `⍣`, is given an integer as the right operator, it is a very simple: `(f⍣k)Y` is simply `f f f … f f Y`. In its dyadic form, it uses the left argument unchanged every time: `X(f⍣k)Y` is `X f X f X f … X f X f Y`.

The only thing to look out for is that the count (`k`) must be separated from the argument, either by naming, or with parenthesis, or by a monadic function (often `⊢`). Note that `k` may be 0, which can be used for “branch-less” conditionals, like replacing one value with another on a condition:

```
3⊣⍣('a'='b')⊢4
3⊣⍣('b'='b')⊢4
```

```
4
```

```
3
```

In the same vein, you can also use it to perform an action conditionally:

```
{⎕←'yup1'}⍣('b'='b')⊢4
{⎕←'yup2'}⍣('a'='b')⊢4
'done'
```

```
yup1
```

```
4
```

Skip to main content

However, `⍣k` can be quite limited. For example, it doesn’t give you the intermediary results. If we need the intermediate results, we could try something like this:

```
2{list,←⍺×⍵}⍣5⊢list←10
list
```

```
10 20 40 80 160 320
```

However, this approach has a subtle problem. Behold:

```
⎕←{list,←⊂⍵}⍣3⊢list←⊂'Yes'
list
```

```
    Yes

 Yes   Yes     Yes       Yes
```

The problem here is that the argument and all results must be scalar. Observe:

```
⎕←2{list,←⍺×⍵}⍣5⊢list←10 11
list
```

```
320 352

10 11 20 22 40 44 80 88 160 176 320 352
```

We can resolve this by either disclosing it after the concatenation `{⊃list,←⊂⍺×⍵}` or use a “concatenate-the-enclosed” function for the modified assignment:

```
⎕←2{list,∘⊂←⍺×⍵}⍣5⊃list←⊂10 11
list
```

Skip to main content

```
320 352
```

```
 10 11  20 22  40 44  80 88  160 176  320 352
```

Now we can write an operator that works like `⍣` but returns all the intermediaries:

```
Pow←{α←⊢ ⋄ r⊣α αα{r,∘⊂←α αα ω}⍣ωω⊃r←⊂ω}
2×Pow 5⊢10 11
```

```
 10 11  20 22  40 44  80 88  160 176  320 352
```

Going back to `2{list,∘⊂←α×ω}⍣5⊃list←⊂10 11`, let’s study that in more detail. First we add the original input as a scalar: `list←⊂10 11`. However, later, with `list,∘⊂←` we only use the enclose as part of the amendment of list. The pass-through of an assignment is always whatever is on the right of `←`, which is why we don’t need to disclose. We could have written `⊃list,←⊂`, too.

In the operator version, the first thing is `α←⊢`. In a dfn and dop, this is a special statement which is only executed if the function is called monadically:

```
{α←⎕←'hello' ⋄ α ω}'world'
'hi'{α←⎕←'hello' ⋄ α ω}'world'
```

```
hello
 hello  world
```

```
 hi  world
```

Note that the side effect of printing ‘hello’ only happened in the monadic case.

`α←⊢` literally assigns the function `⊢` to `α`. So, while normally `α` and `ω` are arrays, `α` can be a function in this special case. It works with any function, not just `⊢`, too:

```
{α←! ⋄ α+ω}4    ⍝ works with any function!
2{α←! ⋄ α+ω}4
```

Skip to main content

```
24
```

```
6
```

This is a convenient way to write ambivalent functions. The inner function is simply the expression we came up with before: `{r,∘⊂←⍺ ⍺⍺ ⍵}⍣⍵⍵`. However, since the function we're actually applying doesn't have a name, we have to pass it in as `⍺⍺`, so the operand to `⍣` is actually another operator. That's why it has the `⍺⍺` of the outer operator on its left, to pass in the function:

```
Pow←{⍺←⊢ ⋄ r⊣⍺ ⍺⍺{r,∘⊂←⍺ ⍺⍺ ⍵}⍣⍵⍵⊃r←⊂⍵}
```

We could also have named it, and used the name:

```
Pow2←{⍺←⊢ ⋄ f←⍺⍺ ⋄ r⊣⍺ {r,∘⊂←⍺ f ⍵}⍣⍵⍵⊃r←⊂⍵}
2×Pow2 5⊢10 11
```

```
 10 11  20 22  40 44  80 88  160 176  320 352
```

A couple of more things worth mentioning about `⍣k`. The inverse `⍣¯1` is quite nifty, and can make things easy that are otherwise complicated. Maybe the most well-known example is `⊥⍣¯1`. The problem is that to convert a number to a given base, `⊤` requires you to tell it how many digits in that base you want. For example,

```
2 2 2 2 2 2⊤10 ⍝ 10 in 6-bit binary
```

```
0 0 1 0 1 0
```

However, the other way, `⊥` just reuses a single base for all digits:

```
2⊥0 0 1 0 1 0
```

```
10
```

Skip to main content

This means that the *inverse* of `⊥` also reuses a single base for “all” digits (that is, as many as needed):

```
2⊥⍣¯1⊢10
```

```
1 0 1 0
```

`⍣` can also invert non-trivial functions:

```
celsius2farenheit←32+1.8∘×
celsius2farenheit 20
celsius2farenheit⍣¯1⊢ 68
```

```
68
```

```
20
```

It also works with non-numeric things:

```
'a',⍣¯2⊢'aaaaa'
```

```
aaa
```

Here, we did the inverse of prepending “a” twice. That is, we removed two “a”s. If we try to give it something that doesn’t begin with two “a”s, we get an error:

```
'a',⍣¯2⊢'abaaa'   ⍝ DOMAIN ERROR
```

```
DOMAIN ERROR
      'a',⍣¯2⊢'abaaa'   ⍝ DOMAIN ERROR
         ^
```

Finally, let’s introduce the concept of “Under”. Sometimes, we want to perform an action while the

Skip to main content

we perform surgery under anaesthesia, and drive under the influence (don't!). `⍣` can make this very readable by defining the temporary action as an invertible function: `Temp⍣¯1⊢Main Temp argument`. We can define such an operator:

```
Under←{⍵⍵⍣¯1 ⍺⍺ ⍵⍵ ⍵}
+/Under⍟3 4 ⍝ multiplication is summation under logarithm
```

```
12
```

If you know the `@` operator, it can be used in combination:

```
'_'@2⊢'hello' ⍝ put an underscore *at* position 2
```

```
h_llo
```

```
'_'@2Under⌽'hello' ⍝ put an underscore *at* position 2 while reversed, that is, 2nd
```

```
hel_o
```

Power in depth: `f⍣g`

`⍣` with a *function* right operand is conceptually simple, but has some gotchas to be aware of. For this section, we'll call the left operand `f` and the right operand `g`, that is, we're applying `f⍣g`. When the derived function is used dyadically, it is just as if it was used monadically with the left argument bound to `f`. That is, `X f⍣g Y` is exactly the same as `X∘f⍣g Y`, so we only need to discuss the monadic case. The high-level view is that `f⍣g` applies `f` until `f g ⊢`.

Now, what exactly does that mean?

We start by applying `f Y` and its result is used as left argument to `g`. The right argument to `g` is the original `Y`. `g` must then return `0` or `1`. If `g` returns 1 it means we're done, and the result will be the newly found value, `f Y`. If `g` returns 0 then we conceptually set `Y←f Y` and start over. For

Skip to main content

example, we can find a “fix-point” by having `g←=`. If we take 10 and divide it by 2 over and over until it doesn’t change any more, we’ll end up with… 0:

```
2÷⍨⍣=10
```

```
0
```

The power (no pun) of `⍣` is of course that you can use *any* functions as operands. You also don’t have to use both arguments of `g`. Often, you just want to repeat an action until a condition on the generated value is fulfilled.

Let’s say we want to use `⍣` to find the first power of 2 larger than 100. That is, double 1 until it exceeds 100. Remember that the newly generated value (the one we’re interested in) is the left argument of `g`. If you use the right argument of `g`, you’ll have applied `f` one more time than needed because your stop condition hinges on the previous value, but the current value has already advanced one more step.

```
2×⍣{100<⍺}1
```

```
128
```

Another example. Given a string, keep dropping characters from the front until it is a palindrome.

```
IsPal←⊢≡⌽
Palify←{1↓⍣{IsPal ⍺}⍣(~IsPal ⍵)⊢⍵}
Palify¨'otto' 'risotto'
```

```
 otto  otto
```

Here, `1↓⍣{IsPal ⍺}` is the same as what we’ve done before, but we only apply it if the argument isn’t already a palindrome. The “if” is expressed with `⍣` and an array right-operand.

`⍣` can be your friend when you want to test each one of a set until you find a good one, without having to test all of them. You can also use it to loop indefinitely until some outside condition tells

Skip to main content

you to stop. In that case, you'd use neither of the arguments of `g`. Sometimes, you don't care about the argument(s) to `f` either, you just need a dummy argument to get the loop going.

For example, here is an expression to collect lines of text from the user until they enter a blank line:

```
¯1↓text⊣{text,∘⊂←⍞}⍣{''≡⍺}text←0⍴⊂''
```

And here is one that neither uses the arguments of `f`, nor of `g`; output random numbers `1…10` until we roll a 6:

```
{}{⎕←?10}⍣{6=?6}⍬
```

```
9
5
5
1
2
9
5
1
4
2
```

It doesn't output the condition roll, just some random number each time. Here is one that keeps rolling until it gets a 6:

```
{}{⎕←?10}⍣{6=⍺}⍬
```

```
5
8
4
8
2
8
2
2
4
6
```

Here's a trick using f⍣g. Sometimes, we can have a nested list of lists of lists, for example, because

Skip to main content

this to a proper multi-dimensional array.

For example, we get the JSON data `[[[5,22,13,18],[9,19,16,11],[4,2,12,20]],[[8,6,17,1],[10,24,15,14],[21,23,7,3]]]` which we can convert to an APL array:

```
⎕JSON'[[[5,22,13,18],[9,19,16,11],[4,2,12,20]],[[8,6,17,1],[10,24,15,14],[21,23,7,3
```

```
 5 22 13 18   9 19 16 11   4 2 12 20     8 6 17 1   10 24 15 14   21 23 7 3
```

But we want a 2-by-3-by-4 array. How would we do this in a general fashion, without querying the depth?

```
↑⍣≡⎕JSON'[[[5,22,13,18],[9,19,16,11],[4,2,12,20]],[[8,6,17,1],[10,24,15,14],[21,23,
```

```
 5 22 13 18
 9 19 16 11
 4  2 12 20

 8  6 17  1
10 24 15 14
21 23  7  3
```

So `↑⍣≡` is a neat idiomatic expression which is worth remembering. The other way, converting a high-rank array to lists of lists isn't as neat, because you can keep applying `↓` and it will just add more nesting. What can we come up with for that? Since `↓` starts at the "bottom", we can just keep going until we have a vector. However, if we know we'll get one enclosure too much, we can just disclose once when done.

```
⊃↓⍣{0≡≢⍴⍺}2 3 4⍴⍳24
```

```
 1 2 3 4   5 6 7 8   9 10 11 12     13 14 15 16   17 18 19 20   21 22 23 24
```

Skip to main content

Decode in depth `⊥`

Let's begin with a basic understand of what a number system really means. When we write `123`, what we really mean is

```
+/1 2 3×100 10 1
```

```
123
```

But why `100 10 1`? You might say that's `10*2 1 0`, but another way to look at it is `⌽×\1,2⍴10`. The `1` here is the "seed" or initial value for our running product. Now we can see a way to generalise this. Instead of `2⍴10` we could choose two different numbers, say 60 and 24. This gives us `⌽×\1 60 24` or `1440 60 1`. This would be a days-hours-minutes system, 1 day being 1440 minutes. So, if we have 1 day, 2 hours, 3 minutes, how many minutes do we have?

```
+/1 2 3×1440 60 1
```

```
1563
```

This brings us to what `⊥` does. It takes a mixed-radix spec as left argument, and evaluates how many of the smallest unit a given "number" (expressed as a vector of "digits") corresponds to.

```
0 24 60⊥1 2 3
```

```
1563
```

Note the difference in the spec between the `+/×` method and the `⊥` method. We don't have to specify the unit (which'll always be 1 anyway) on the little end, but instead, we pad with a 0 on the big end. The 0 is ignored, and could actually be any value. The only reason it's needed at all is to match the length of the right argument.

Skip to main content

```
10⊥1 2 3  ⍝ base ten
2⊥1 0 1   ⍝ binary
```

```
123
```

```
5
```

So `⊥` is really a kind of fanciful cover for `+/×` or actually `+.×`, the latter explaining why `⊥` takes a transposed argument.

```
10 10 10 ⊥ ⍉2 3⍴1 2 3  3 2 1
100 10 1+.×⍉2 3⍴1 2 3  3 2 1
```

```
123 321
```

```
123 321
```

We can model `⊥` as:

```
10 10 1 {(⌽×\⌽⍺)+.×⍵} ⍉2 3⍴1 2 3  3 2 1
24 60 1 {(⌽×\⌽⍺)+.×⍵} 1 2 3
```

```
123 321
```

```
1563
```

Because `⊥` has a specific definition rather than being some specialised type-dependent utility, it can be used for some unusual tricks that have little apparent connection to base-conversion. One that has achieved some fame is `⊥⍨` on a Boolean vector. Let's analyse what it does.

Let's say we have the vector `1 0 1 1 1`. `⍨` will cause the vector to be used both a base specification and as the count for each "type" place ("hundreds", "tens", ones). So we have `1 0 1 1 1⊥1 0 1 1 1`. Remember, this really means:

Skip to main content

```
+/(⌽×\⌽1,⍨1↓1 0 1 1 1)×1 0 1 1 1
⊥⍨1 0 1 1 1
```

```
3
```

```
3
```

That’s why `⊥⍨` is “count trailing 1s”. Conceptually, we add 1s from the right (though each is multiplied by increasing powers of 1 — all `1*n` being always 1 of course), until a 0 causes everything after that to become 0 (`n×0` being always 0 of course). Finally, we sum.

Another trick, often used in tacit APL, is `1⊥something`. Let’s analyse that one. The first thing we can recognise here is that the 1 will be expanded to match the length of the right argument, so say `1⊥3 1 4` really means `1 1 1⊥3 1 4`. This is simply:

```
+/(⌽×\⌽1,⍨1↓1 1 1)×3 1 4
1⊥3 1 4
```

```
8
```

```
8
```

`×\` applied to a vector of 1s, is still “1”. That’s the multiplicative identity, which means that `1⊥` is equivalent to `+/`. But remember the transposing when dealing with multi-dimensional arguments, and you’ll soon realise that it is actually `+⌿`. Let’s look at that. Notice that the two numbers 271 and 314 are represented in base 10 as:

```
⍉2 3⍴2 7 1  3 1 4
```

```
2 3
7 1
1 4
```

Why? Because then we can do:

Skip to main content

```
100 10 1+.×⍉2 3⍴2 7 1  3 1 4
```

```
271 314
```

which is the same thing as:

```
+⌿100 10 1×⍤0 1⍉2 3⍴2 7 1  3 1 4
```

```
271 314
```

Or, in other words, we multiply each row by its place weight (big endian) and then sum vertically. Then, if the weight is a constant 1, we have a simple vertical summation, or `+⌿`.

Another trick, also sometimes used in tacit APL is `0⊥something`. Let's analyse that one. First, the left argument is extended to match the shape of the right argument: `0⊥314` is the same as `0 0 0⊥3 1 4`. Again, recall that this is the same as

```
(⌽×\⌽1,⍨1↓0 0 0)×3 1 4
```

```
0 0 4
```

Summing that gives us 4; the last element of the vector:

```
+/(⌽×\⌽1,⍨1↓0 0 0)×3 1 4
0⊥3 1 4
```

```
4
```

```
4
```

What happens if we apply this to a higher-rank array? If we examine the rank, we can see it returns the last major cell of its argument:

Skip to main content

```
⊢m←3 3⍴9?9
⊢c←0⊥m
⊃⍴c
```

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

```
5 9 3
```

```
3
```

Since we’re returning the last major cell unmodified, it is the same as `⊢⌿`.

Encode in depth `⊤`

`⊤` is known as “Encode” or “Represent”. It takes a number (or multiple numbers, in the same way as with `⊥`) as right argument and generates a representation in the (mixed) number base(s) given in the left argument. As a memory aid, we can call it N-code (“encode”) to remember that it is typed with APL+n (while `⊥` is clearly a “base”, and indeed evaluates numbers in custom bases, B for base; type it with APL+b).

As we saw previously, `⊥` is quite simple. In a way, it is a fancier `+.×`: it just gives the given “digits” weights, and sums the result. The weights being determined from the reverse cumulative product of the left argument (and there’s some transposing going on too). `⊤` is much more complex, computationally speaking, but not really conceptually, where it is basically the inverse operation. One way to explain it is to show how `⊤` constructs its result. As a simple example, let’s take:

```
0 7 24 60⊤12345
```

```
1 1 13 45
```

The `0 7 24 60` here represents a number system with 60 of the basic units in the next larger unit, 24 of those larger units in the next larger, etc. It could, for example, be 60 minutes in an hour, 24 hours in a day, and 7 days in a week. The 0 means that there are no larger units, and we’ll keep

Skip to main content

making cash change: there's nothing larger than a $500 unit, so even if we have to pay a million, we'll have to use lots of $500s.

What are our weights?

```
⌽×\⌽1,⍨1↓0 7 24 60
```

```
10080 1440 60 1
```

That is, there's 1 minute in a minute, 60 minutes in an hour, 1440 minutes in a day, and 10080 minutes in a week. We can check the result that `⊤` gave us by using these weights:

```
1 1 13 45+.×⌽×\⌽1,⍨1↓0 7 24 60
```

```
12345
```

Yup, that worked.

How did `⊤` get the result then? Let's do it step-by-step, building our result from the right. The first unit rolls over at 60, so we can find how many of the smallest units (here, minutes) we need in order to get the exact total value by applying division remainder:

```
60|12345
```

```
45
```

There's our right-most "digit". Let's put that aside in our result. How many minutes are left?

```
12345-45
```

```
12300
```

The next unit (the hours) consist roll over at 24 hours of 60 minutes each. Any multiple of 24 hours

Skip to main content

counted in hours:

```
(24×60)|12300
```

```
780
```

This is how many *minutes* we want counted as hours. How many hours is that, though?

```
60÷⍨(24×60)|12300
```

```
13
```

There’s the second-from-right element of our result. Let’s prepend it to get a preliminary result of `13 45`. We’ve used 780 minutes this time around. How much do we have left (which will be counted in days and maybe weeks)?

```
12300-780
```

```
11520
```

Next up are days, which we’ll use to fill until we have a value that can be counted in whole weeks. A week, of course, being how many minutes?

```
7×24×60
```

```
10080
```

So we need the division remainder when divided by that.

```
(7×24×60)|11520
```

Skip to main content

That's how many days (stated in minutes) we have. How many actual days does that add up to?

```
(24×60)÷⍨(7×24×60)|11520
```

```
1
```

That's the next value in our result, giving us `1 13 45`. How much is left now?

```
11520-1440
```

```
10080
```

Which you might recognise as a single week (expressed in minutes), giving us another 1 in our result: `1 1 13 45`.

Now for the classic question. Why doesn't this work for making change?

```
4 2.5 2 5⊤42 ⍝ 4 quarters in a dollar, 2.5 dimes in a quarter, 2 nickels in a dime,
```

```
1 1.5 0 2
```

I can't pay 42 pence as 1 quarter, 1.5 dimes, and 2 pennies! Sure, mathematically, it'd work, but I'm not sure the US mint will be too excited if I start chopping dimes in half.

So what happened here? Let's walk through the process again. We start by finding what the remainder is, which we'll have to pay in pennies:

```
5|42
```

```
2
```

That leaves 40 pence. Since 2 nickels go into a dime, we do a mod-10 to find how many nickels we need:

Skip to main content

```
(2×5)|40
```

```
0
```

None, of course. So we still have 40p or ¢40 if you want. Continuing on, how many dimes? The dimes roll over at 2.5:

```
(2.5×2×5)|40
```

```
15
```

So only 15 pence will need to be paid in dimes. Herein lies our problem. That's of course 1.5 dimes. Hence our result. Left over is `40-15`, that is, 25 pence, enough for a single quarter. Actually, proper change-making with arbitrarily valued coins is a weakly NP-hard problem. Look at the total amount as a knapsack you need to fill. You only have items of fixed volume to put in there. There's no obvious way to see exactly how to fill the bag fully. However, mints are careful to only issue pieces in such a way that a greedy algorithm works.

`⊤` gives you the possibility of running a custom counter or odometer which rolls over eventually. Think of the case `24 60 60⊤`. If it didn't "chop" (mod, really), there'd be no way to know what the next digit value would be. So what `2⊤13` means is a base-2 odometer with a single digit display, rolling over whenever the value exceeds 1. Now, you could complain that this equates 2 and ,2. You'd be right. There probably never any reason to use a scalar as left argument for `⊤`. If you want mod, use `|`.

The only difference between `⊤` and `|` for scalar left arguments, is comparison tolerance (`⎕CT`), which `|` cares about, but ⊤ ignores. But if you want `⎕CT←0`, you should set it explicitly rather than obscuring your code with `⊤` and a scalar left argument.

Let's look at some neat tricks with `⊤`. You can use 0 1⊤ to split a number into its integer part and fractional part:

```
0 1⊤3.14
0 1⊤3.14 2.7 100.23
```

Skip to main content

```
3 0.14
```

```
3     2    100
0.14 0.7    0.23
```

You can use `⊤` to split “packed integers”:

```
0 100 100⊤20200326
```

```
2020 3 26
```

A golfing trick is getting `0 0⍴0` (an empty numeric matrix):

```
(0 0⍴0) ≡ ⍬⊤⍬
```

```
1
```

If fact, you can “silence” anything by making the leading axis have length `0` using `⍬⊤`:

```
⍬⊤2 3⍴⎕A
```

If you have a multi-dimensional array, but want the Nth element without having to ravel the array, how do you find the multi-dimensional index of the sought element? Consider

```
4 5⍴⎕A
```

```
ABCDE
FGHIJ
KLMNO
PQRST
```

Using 0-based indexing, this is very simple:

Skip to main content

```
⎕IO←0
4 5⊤13
(4 5⊤13)⌷4 5⍴⎕A
13⊃,4 5⍴⎕A
```

```
2 3
```

```
N
```

```
N
```

We need `⎕IO←0` because of how `⊤` works. It does a mod (`|`) all the time. When we "roll over" from one row to the next, we end up in position 0.

Variant in depth `⍠`

Variant, `⍠`, is a dyadic operator, but it is quite unlike all other operators in APL. Syntactically, it is normal though. It always takes a function (monadic or dyadic) on its left, and always takes an array on its right. Although it is usually called Variant, you can also call it *Option*. In fact, it has a system operator synonym, `⎕OPT`.

Variant is special in that it sets options in an invisible set of options. You can't access this set directly, only observe modified behaviour in the operand function, because the operand function will check this set to know what to do.

This also means that, uniquely, the operand function will "know" that it is being called as an operand of `⍠`. Usually, functions can't really detect (easily) who called them. The left operand (the function) must be one of a fixed set of system functions (or functions derived from system operators).

The right operand must be one of:

- a scalar (this one is known as the principal option)
- a 2-element key-value pair
- a vector of 2-element key-value pairs.

Skip to main content

The scalar operand is only allowed if a default key exists, in which case it is equivalent to 'DefaultKey' value. Let's take an example. You might know about the system function to convert to and from JSON:

```
⎕JSON⍳3
```

```
[1,2,3]
```

We can use `⍠` with the key `'Compact'` to change the white-space behaviour of `⎕JSON`. In essence, `⍠` sets the `Compact` setting to the corresponding value (0 or 1 in this case):

```
⎕JSON⍠'Compact' 0 ⍳3
```

```
[
  1,
  2,
  3
]
```

There are other options too. Typically, `⎕JSON` will convert a JavaScript `null` to an APL enclosed string `⊂'null'`:

```
(⊂'null') ≡ ⎕JSON'null'
```

```
1
```

However, if you instead want it to convert it to an object-type null, `⎕NULL` you can tell it so:

```
⎕NULL ≡ ⎕JSON⍠'Null' ⎕NULL ⊢ 'null'
```

```
1
```

Notice the `⊢`. Whenever a dyadic operator has an array right operand, it will strand together with

Skip to main content

Another option for `⎕JSON` is to convert JSON into an APL matrix that describes the JSON, rather than attempting to actually convert to an equivalent APL structure:

```
⎕JSON⍠'Format' 'M' ⊢ '[1,null,"hello"]'
```

```
0            2
1         1  3
1     null   5
1     hello  4
```

The exact details of this *M*atrix Format isn't important here, though. You can check out the docs. Now that we know about a couple of options, we can look at how to specify multiple options. We can create a "dictionary" of key-value pairs:

```
⎕JSON⍠('Format' 'M')('Null' ⎕NULL) ⊢ '[1,null,"hello"]'
```

```
0            2
1         1  3
1    [Null]  5
1     hello  4
```

Notice how we both got a matrix, and the `null` became `[Null]` (the text representation of `⎕NULL`) rather than an enclosed `'null'`. We can also use `⍠` twice:

```
⎕JSON⍠'Format' 'M'⍠'Null' ⎕NULL ⊢ '[1,null,"hello"]'
```

```
0            2
1         1  3
1    [Null]  5
1     hello  4
```

If we check the docs for `⎕JSON`, we'll see that `'Format'` is the principal option, which means we can specify it as a scalar:

```
⎕JSON⍠'M'⍠'Null' ⎕NULL ⊢ '[1,null,"hello"]'
```

Skip to main content

```
0            2
1         1  3
1    [Null]  5
1     hello  4
```

What happens if we set the same option twice with different values? The rightmost one takes precedence. There are two ways you can think of it, both leading to that same conclusion:

1. `⍠` (like any operator) modifies its operand function. For simplicity, lets say we have two monadic operators applied acting on a function, `f op1 op2`, `op2` gets to modify the derived function `f op1`. That is, the rightmost has the final say.
2. When we evaluate, we first have to process the inner derived function's operator (as in the previous point), which sets the hidden option. Then we proceed to the outer operator, which in turn overwrites the state. Only then is the function allowed to run, picking up the setting set by the rightmost (outer) operator.

Variant is also used with `⎕R` and its sibling `⎕S`. If you're not familiar with `⎕R`: Briefly, it is a dyadic operator, Replacing occurrences of its left operand with its right operand, in the right argument:

```
's'⎕R'S' ⊢ 'mississippi'
```

```
miSSiSSippi
```

This replaces all lowercase s with uppercase S. Let's say we only want to replace the first 2. We can set the Match Limit to 2. The option key to use for this is `'ML'`.

```
's'⎕R'S'⍠'ML'2 ⊢ 'mississippi'
```

```
miSSissippi
```

`⎕R` is an operator. It takes two operands, in our case 's' and 'S', and derives a new function. It is this derived function that `⍠` needs to act upon by taking it as its left operand. So the order is `FunctionToBeModified ⍠ options ⊢ argument`. Alternatively, we can parenthesise: `(FunctionToBeModified ⍠ options) argument`.

Skip to main content

```
('s'⎕R'S'⍠'ML'2) 'mississippi'
```

```
miSSissippi
```

Naming a derived monadic operator:

```
ReplaceWithS←⎕R'S'
's'ReplaceWithS 'mississippi'
```

```
miSSiSSippi
```

This also means we can name the combination of `⍠` with one or more options.

```
OnlyTwo←⍠'ML'2
's'⎕R'S'OnlyTwo 'mississippi'
```

```
miSSissippi
```

We can even do both:

```
ReplaceWithS←⎕R'S'
OnlyTwo←⍠'ML'2
's'ReplaceWithS OnlyTwo 'mississippi'
```

```
miSSissippi
```

A really common thing with regexes is wanting case insensitivity. That is `'IC'1` (Ignore Case), but it is also the principal option:

```
'ss'⎕R'__'⍠1⊢'MISSissippi'
```

```
MI__i__ippi
```

Skip to main content

But it only works if that is the only setting you're changing. Though, you can always use `⍠` twice:

```
's'⎕R'_'⍠'ML'3⍠1⊢'MISSissippi'
```

```
MI__i_sippi
```

Here is another example where we use `⍠` on `⎕R` to do something entirely unrelated to regular expressions. Sometimes, your input can be of various forms and you need to normalise it. Say you get some text, but it could be a character scalar, a character vector, a vector of character vectors, an enclosed character vector, or even a character vector with literal newlines. So we want to normalise all of these to become a vector of character vectors.

```
VecOfVecs←''⎕R''⍠'ResultText' 'Nested'
VecOfVecs 'a'
VecOfVecs 'abc'
VecOfVecs 'abc' 'def'
VecOfVecs ⊂'abc'
VecOfVecs 'abc',(⎕UCS 10),'def'
```

```
 a

 abc

 abc  def

 abc

 abc  def
```

Note that Dyalog often adds additional options to existing system functions based on customer demand. Case in point, in version 18.0, options were added to `⎕JSON` to automatically split high-rank arrays so they can be represented as JSON, and an option to process and generate JSON5. And for `⎕R`/`⎕S`, options to turn regexes off so you can do literal replacements without worrying about having to escape characters that have special meaning in PCRE.

One more usage of `⍠` that isn't really related to this, and we can't demonstate it easily here, either.

called method. However, .NET methods can be overloaded (different code depending on the type of the argument), and then APL can't know which one you want. You can use `⍠` with the method and the option `'OverloadTypes'` to choose. The value has to be a .NET data type, e.g. `Double` or `Int32`. This option is the principal option too, so the calling can be done simply with `MyDotNetMethod⍠Double ⊢ argument`. If the method takes multiple arguments, you can specify a vector of types: `MyDotNetMethod⍠(⊂Double Int32) ⊢ argument`

Notice two things:

1. The types are not quoted names, they are scalar references to the .NET types. 1.   When specifying a vector of types, it must be enclosed, as the principal option must be a scalar.

Unique mask in depth `≠`

Let's have a look at monadic `≠`, called called unique mask or *nub sieve*. Note that it isn't particularly related to the dyadic form (unequal). Instead, it relates to unique, `∪`. *Unique* returns a subset of the major cells of its argument. *Unique mask* returns a Boolean vector which, when used as left argument to `⌿` and with the original argument as right argument, returns the same as *unique* would on the original argument:

```
∪'mississippi'
≠'mississippi'
{(≠⍵)⌿⍵}'mississippi'
```

```
misp
```

```
1 1 1 0 0 0 0 0 1 0 0
```

```
misp
```

Why might we need such a function? Compared with `∪Y`, you can use the results from `≠Y` to filter *other* arrays, or indeed to do other computations. It is as if `∪Y` already applied their implied information before you had a chance to use that info for what you wanted. It's also worth noting that the result of `≠Y` is much more light-weight than `∪Y`, in that it only ever has one bit per major cell, while `∪Y` could end up duplicating a lot of data.

Skip to main content

```
m←⎕←(≠∧∊∘'aeiou')t←'hello world'
m/t
```

```
0 1 0 0 1 0 0 0 0 0 0
```

```
eo
```

which gives the unique vowels. Of course, in this case, you could equally well write `'aeiou'∩'hello world'` but this example is to illustrate the concept.

Here's another example. Given some text (simple character vector) `t`, return a matrix so that the first instance of each occurring character is "underlined". Here's one approach:

```
F1 ← ↑⊢,⍥⊂'-'\⍨≠
```

```
F1 'mississippi'
F1 'hello world'
```

```
mississippi
---     -
```

```
hello world
--- --- - -
```

Here's another,

```
F2 ← {⎕IO←0⋄↑⍵(' _'[≠⍵])}
```

```
F2 'mississippi'
F2 'hello world'
```

Skip to main content

```
mississippi
---     -

hello world
--- --- - -
```

A final example: given a vector, return the set of elements which have duplicates, preserving order:

```
F3 ← (∪∩{⍵/⍨~≠⍵}) ⍝ h/t @bubbler
```

```
F3 'mississippi'
F3 'hello world'
```

```
isp

lo
```

Domino in depth ⌹

Matrix inversion, `⌹`, is often called *domino* due to its symbol which isn't really a domino (`🁫`) at all, but rather a division sign in a quad, the latter representing division/inversion (`÷`). You're of course familiar with the `÷` primitive. Perhaps you also know that matrix multiplication is `+.×` but that we don't have a corresponding operator for matrix division. You *can* actually use `+.×⍣¯1` for matrix division, but since `⍣` wasn't always around (and certainly not `⍣¯1`) and for notational ease, `⌹` provides this functionality too. Matrix inversion, what is that? Well, for a square matrix A, its inverse A^{-1} is a matrix such that when the two are multiplied together, the result is the identity matrix:

$$AA^{-1} = A^{-1}A = I$$

Let's keep two easy-to-remember matrices at hand:

```
⎕←E←2 2⍴2 7 1 8
⎕←P←2 2⍴3 1 4 1
```

Skip to main content

```
2 7
1 8
```

```
3 1
4 1
```

If we invert `P` we get:

```
⌹P
```

```
¯1  1
 4 ¯3
```

And indeed:

```
P+.×⌹P
```

```
1 0
0 1
```

In mathematics, matrix division as a notation isn't usually used. Instead, mathematicians use multiplication by an inverse. However, the analogy with `×` and `÷` *is* pretty obvious, so APL defines `A⌹B` as `(⌹B)+.×A` just like `a÷b` is `(÷b)×a` (remember though that matrix multiplication isn't commutative!):

```
E⌹P
(⌹P)+.×E
```

```
¯1 1
 5 4
```

```
¯1 1
 5 4
```

So far, there's nothing much controversial here. However, `⌹` isn't just for matrices. You can use it

Skip to main content

```
2 7⌹3 1
```

```
1.3
```

What does this mean? Well, following the above reasoning, we can perhaps see that the following should be equivalent:

```
(⌹3 1)+.×2 7
```

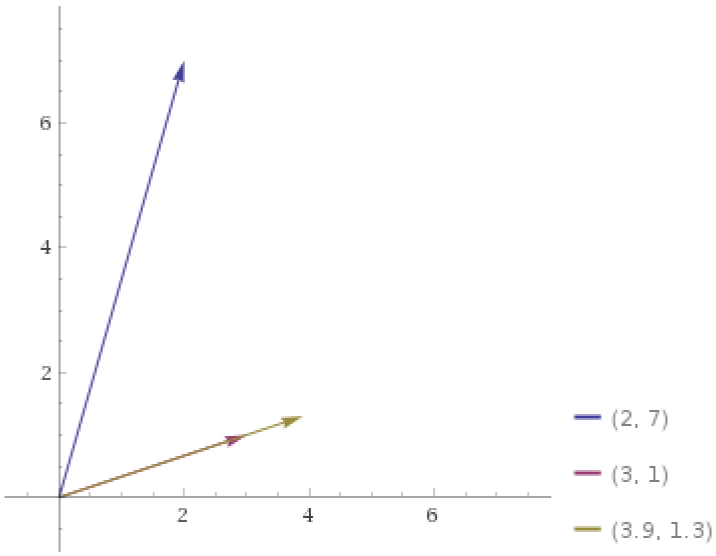
```
1.3
```

and that `⌹ v` represents the vector divided by the square of its norm:

```
⌹3 1
```

```
0.3 0.1
```

Another way to think of it is that `2 7⌹3 1` is the “length” of the component of `2 7` in the `3 1`-direction.



In other words, if we project `2 7` perpendicularly to the extension of `3 1` we hit a point on `3 1`’s extension which is `1.3×3 1` from `0 0`.

Skip to main content

to make sure to catch division-by-zero errors.

A common usage for `⌹` is to solve equation systems. Consider

\begin{array}{rcrcr} 2x &+& 7y &=& 12 \ x &+& 8y &=& 15 \end{array}

We can represent this as a matrix (our `E`) on the left of the equal signs and as a vector (`12 15`) on the right.

```
12 15⌹E
```

```
¯1 2
```

This says `x←¯1` and `y←2`. Let's check the result:

```
2 7+.×¯1 2
1 8+.×¯1 2
```

```
12
```

```
15
```

OK, remember how we found `x y≡¯1 2` with `12 15⌹E`? It follows that if we add `x` and `y` we should get 1:

```
12 15 1⌹E⍪1 1
```

```
¯1 2
```

which simply means that

\begin{array}{rcrcr} 2x &+& 7y &=& 12 \ x &+& 8y &=& 15 \ x &+& y &=& 1\end{array}

But what if we tell APL that the last sum *doesn't* equal 1?

Skip to main content

```
(x y)←⎕←12 15 1.1⌹E⍪1 1
```

```
¯0.9412903226 1.989032258
```

What nonsense *is* this? It doesn't even fulfil any of the equations:

```
2 7+.×x y
1 8+.×x y
1 1+.×x y
```

```
12.04064516
```

```
14.97096774
```

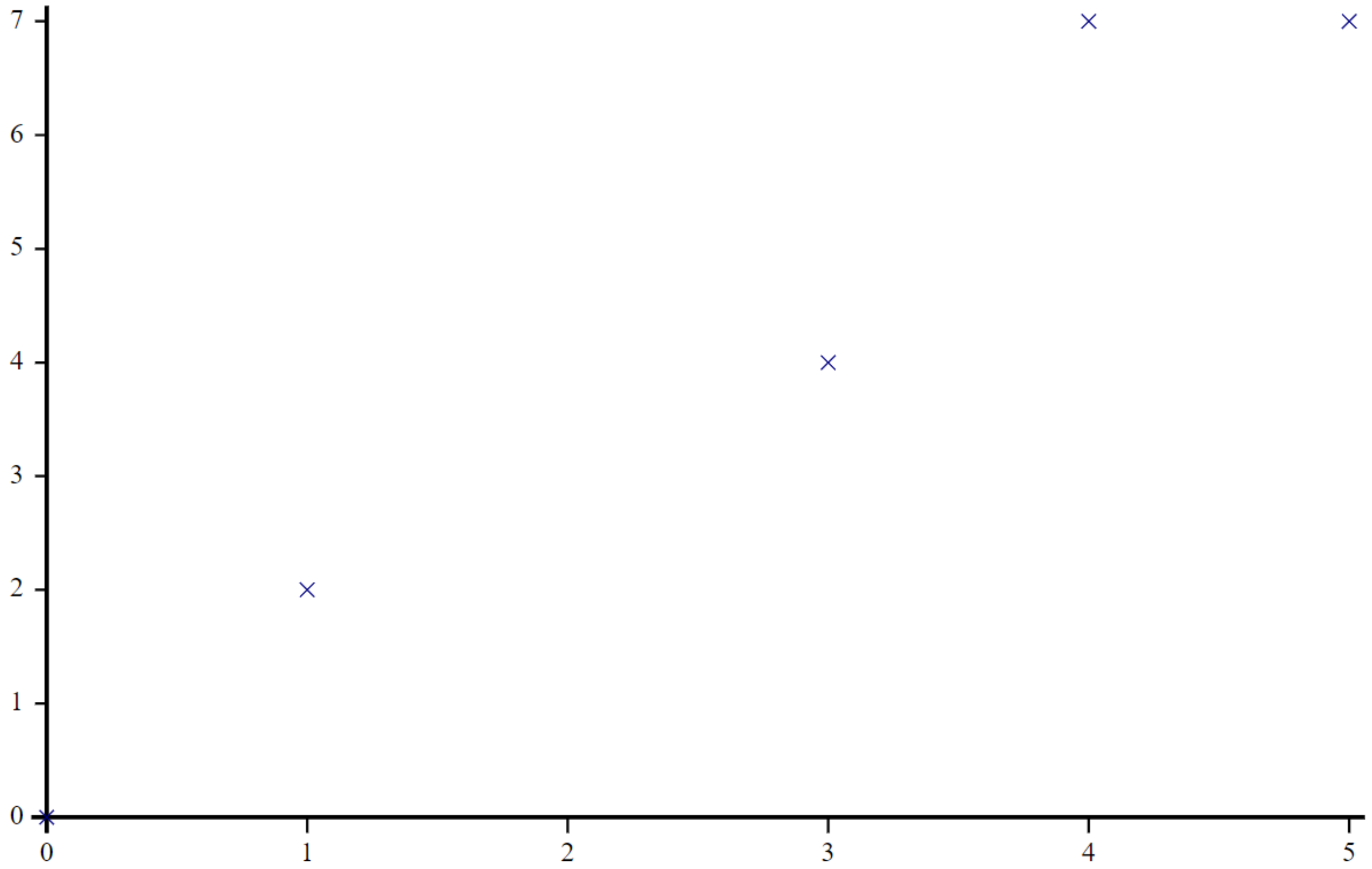
```
1.047741935
```

But as you can see, it is pretty close. This is an over-determined system, so APL found the solution that fits best. It defines "best" by a very common method called the least squares fit, which can also be used to make other kinds of fits. What it means is that it tries to minimise the squares of the "errors". In a sense, it smoothes the errors out, which means we can use it for smooth curve-fitting too.

Unfortunately, we won't have the scope to go through many possibilities here, but you can see a few uses if you search APLcart for `⌹ fit`. Let's just take the very first one from there: `⊢⌹1,∘⍪⊣`. Let's say we have

```
x ← 0 1 3 4 5
y ← 0 2 4 7 7
```

Skip to main content



```
x(⊢⌹1,∘⍪⊣)y
```

```
0.2209302326 1.453488372
```

This means the best linear fit is $y(x) = 0.22093x + 1.45349$

Object-oriented APL

Dyalog has rich support for object-oriented programming in APL. If you are familiar with C# or Java OOP you'll find this very familiar. Dyalog APL is an official (i.e. listed by Microsoft as a) .NET language and the object orientation is well aligned with C#. This also means that it's easy to call Dyalog from other .NET languages.

Skip to main content

Namespaces

Let’s create the simplest APL type of object Dyalog APL has, the namespace. APLWiki has a good intro. A namespace is like a container for other APL items (functions, variables, and namespaces). It is very much like a JSON object.

One way to create a new empty namespace is using the system function `⎕NS`. For now, we’ll just use a dummy right argument; `⍬`. To assign into a namespace we use the dot-notation: `namespace.name←value`. Same goes when we want to query the value.

```
b←a←⎕NS ⍬
a.var←52
b.var←42
a.var b.var
```

```
42 42
```

We created the namespace `a`. Then we used its value to set `b`, then we set `var` inside `a` and inside `b` to two different values, but when we queried the two values they had become the same (the latter). This is because APL objects are mutable. Another way to look at it is that the value isn’t really the namespace itself, but rather a *reference* to a single object we created with a single call to `⎕NS`.

```
b a←⎕NS¨ ⍬ ⍬
a.var←52
b.var←42
a.var b.var
```

```
52 42
```

Here we called `⎕NS` twice, once on each of the two `⍬`s. And so `b` and `a` refer to two different objects. Also note that there is no assignment arrow between `b` and `a`, but don’t be fooled:

Skip to main content

```
b a←⎕NS ⍬
a.var←52
b.var←42
a.var b.var
```

```
42 42
```

The last `42 42` result is of course (!) because of APL’s scalar extension (vectorisation/mapping/…). Refs are scalar values, and so the scalar was distributed to both names, just like `b a←42` would have done.

You can also put functions inside a namespace:

```
ns←⎕NS ⍬
ns.fn←{'hello' ⍵}
ns.fn 'world'
```

```
 hello  world
```

All APL built-ins exist (separately!) in every namespace.

```
a b←⎕NS¨⍬⍬
a.⎕IO←0
b.⎕IO←1
a.⍳ 5
b.⍳ 5
```

```
0 1 2 3 4
```

```
1 2 3 4 5
```

Here is another way to create a namespace:

```
ns←⎕JSON '{"a":52, "b":42}'
ns.a
ns.b
```

Skip to main content

```
52
```

```
42
```

We can, of course, also use `⎕JSON` to visualise (simple) APL objects:

```
a←⎕NS ⍬ ⋄ a.(x y z)←1 2 'Brian'
⎕JSON a
```

```
{"x":1,"y":2,"z":"Brian"}
```

Namespaces are great ways to organise you code and data. But sometimes you need a better overview of the namespace content, or you want to put tradfns there (in an easy manner) or even put some comments in. To help you manage larger namespaces and especially *code* in namespaces, you can have a scripted namespace. The script is a simple text document which gets “fixed” into a namespace, much like the JSON text got converted to an APL object. This uses a syntax similar to the tradfn control structures, namely `:Namespace … :EndNamespace`:

```
⎕FIX ':Namespace a' 'var←42' ':EndNamespace'
a.var
```

```
42
```

Of course, the `⎕FIX` usage is even more cumbersome (except possibly when you need to define namespaces under program control), but in an interactive APL session, you can enter `)ed ⍟nyns` to open the editor with a new namespace script. In a Jupyter notebook cell you can create a scripted namespace using `]dinput`:

```
]dinput
:Namespace b
var←43
:EndNamespace
```

```
b.var
```

Skip to main content

```
43
```

Here’s a scripted namespace with a few things in it; a variable, a dfn, and a tradfn:

```
]dinput
:Namespace ns
    var←42
    dfn←{
        'the argument:' ⍵
    }
    ∇ r←tradfn x
      r←?x
    ∇
:EndNamespace
```

```
var←77
⎕NL -⍳9
ns.⎕NL -⍳9
var ns.var
```

```
 a  b  ns  var

 dfn  tradfn  var

77 42
```

We first ask for the Name List in `#` (the root namespace) and again inside `ns` and then we retrieve the value of `#.var` and `ns.var`.

By the way, from inside a namespace, you can access the parent namespace with `##` and its parent with `##.##` etc. `#` doesn’t have a parent though, so `#.##` is the same as `#`. This of course implies that you can nest namespaces. And indeed, you can even do so inside a script:

Skip to main content

```
]dinput
:Namespace ns
    variable←42
    dfn←{
        'the arguments:'⍺ ⍵
    }
    :Namespace inner
        ∇ r←tradfn x
          r←?x
        ∇
    :EndNamespace
:EndNamespace
```

```
ns.inner.tradfn 3
```

```
3
```

Objects and classes

Next up is a special case of a namespace called a class. Remember: All APL objects are namespaces. The ones we just call “namespaces” are the most general ones with no restrictive rules. Classes can hide stuff from the outside onlooker. Adhering to a set of rules, they can be used to create other objects (instances). All this should be familiar to you if you’ve done any OOP (object oriented programming), e.g. in C#, Java or Python.

Remember that we can tell the editor to begin a new namespace with `)ed ⍟myns`? We can begin editing a new class with `)ed ○myclass`. We could also create a new empty namespace with `⎕NS`. We can’t do that with classes as they need some meta-information.

Fundamental to a class it that it restricts which of its members can be “seen” from the outside. By default fields (i.e. variables) and methods (i.e. functions) are “private”, but we can make them “public” so that they can be seen. This is convenient to implement black-box things and create layers of abstraction (for those that like such). Another feature of fields and methods is whether they are “shared” among all the instances, or whether a separate method/field belongs to each instance. By default, they belong to the instances.

So what is an instance?

Skip to main content

An instance is a new object which is based on a class, which is then its base class. Instances inherit all methods and fields from their base class, but they may either each have their own or share one (which is then considered as if it remains in the base class).

Let's see some code:

```
]dinput
:Class cl
    :field f←'f'
    :field public fp←'fp'
    :field shared fs←'fs'
    :field public shared fps←'fps'
    ∇ r←look
      :Access public shared
      r←⎕NL -⍳9
    ∇
:EndClass
```

The above is a script for a class called `cl`. You can see that it has four fields and one method (function). The first field, `f`, has all the defaults, i.e. it is private, and for each instance. The second field, `fp` can be seen from outside each instance. The third, `fs`, is private, but shared among all instances (and their base class, `cl`). The last field, `fps` is both visible to the outside public, and also shared. The method, `look`, is public and shared, just like the field `fps`.

So, if, from outside `cl`, we try looking into `cl`, which members can we see? We won't be able to see `fp` because it is `instance`, not `shared`. So since `cl` is not an instance, it won't show `fp`. We can verify this:

```
cl.⎕NL -⍳9
```

```
 fps  look
```

Now, let's step into `cl` and have a look from inside. We do that by running `cl.look`. As you can see, `look` just returns the list of members that it can see.

```
cl.look
```

Skip to main content

Note that `cl` is in there, just like a function can "see" itself:

```
f←{⎕NL-⍳9}
f ⍬
```

```
 cl  f
```

Everything that's shared (i.e. non-"instance") can be seen, and also the class itself. This is useful if you work with a class and need to inspect what's going on inside. You can just trace into any public function, and then leave the system suspended. Now you can work from inside the class. When you're done, just execute `→0` to quit the function.

Let's try to create our first instance of `cl`. We do that using the system function `⎕NEW`. It takes `cl` as right argument and returns an instance:

```
inst←⎕NEW cl
inst.⎕NL -⍳9
```

```
 fp  fps  look
```

If you expected `fs`, then it is shared alright, but remember that we're on the outside. It isn't public. We can see `look`, because it is public (and shared too, but that doesn't matter here). We can also see `fp` which we couldn't see before, because it is an instance field. But now we do have an instance, and as it is public too, we can see it.

Now, let's run `inst.look`. What do we get?

```
inst.look
```

```
 cl  fps  fs  look
```

The reason `look` cannot see `f` is because `f` isn't public. But we're *inside*, you say? Yes, but inside what? Remember that `look` is a *shared* method. This means that it resides in `cl`, not in `inst`. And from inside `cl`, the private fields of `inst` are invisible. To prove this, we can make a

Skip to main content

```
]dinput
:Class cl
    :field f←'f'
    :field public fp←'fp'
    :field shared fs←'fs'
    :field public shared fps←'fps'
    ∇ r←look
      :Access public                 ⍝ Note: no longer 'shared'!
      r←⎕NL -⍳9
    ∇
:EndClass
```

```
inst.look
```

```
 cl  f  fp  fps  fs  look
```

The only difference here is that `look` is now an instance method. This means that we can no longer do `cl.look`.

Constructors and destructors

When you create an instance of a class using `⎕NEW BaseClass`, you may want to supply some parameters. For this, we use a special type of method (function) called a constructor.

Constructors

When you create a new instance, a *constructor* (if one exists in the class script) will be called.

```
]dinput
:Class cl1
    ∇ Ctor
      :Access public
      :Implements constructor
      ⎕←'Hi!'
    ∇
:EndClass
```

Skip to main content

```
Hi!
```

We defined a tradfn called `Ctor` (it could be called anything, though) and declared it available from the outside (it must be, as you can't be inside yet when you're just creating the instance). As you saw, creating an instance with `⎕NEW` ran the constructor.

Here's a slightly modified version where the constructor sets a field value:

```
]dinput
:Class cl2
    :Field public value
    ∇ Ctor x
      :Access public
      :Implements constructor
      value←x
    ∇
:EndClass
```

Here we have an uninitialised field (`value`) and a monadic constructor (`Ctor`) which sets the field upon construction:

```
inst←⎕NEW cl2 42
inst.value
```

```
42
```

A class can have multiple constructors, too:

```
]dinput
:Class cl3
    ∇ None
      :Access public
      :Implements constructor
      ⎕←'No arguments.'
    ∇
    ∇ One x
      :Access public
      :Implements constructor
      ⎕←'1 argument:'x
    ∇
:EndClass
```

Here is a class with two constructors. APL will call the appropriate one (niladic or monadic):

```
instA←⎕NEW cl3 42
instB←⎕NEW cl3
```

```
  1 argument:  42
```

```
 No arguments.
```

Or why not a class with *three* constructors:

```
]dinput
:Class cl4
    ∇ None
      :Access public
      :Implements constructor
      ⎕←'No arguments.'
    ∇
    ∇ One x
      :Access public
      :Implements constructor
      ⎕←'1 argument:'x
    ∇
    ∇ Two(a b)
      :Access public
      :Implements constructor
      ⎕←'2 arguments'a b
    ∇
:EndClass
```

```
instA←⎕NEW cl4
instB←⎕NEW cl4 42
instC←⎕NEW cl4 (42 3.1415)
```

```
 No arguments.
```

```
  1 argument:  42
```

```
  2 arguments  42 3.1415
```

Skip to main content

So APL calls the right constructor based on the number of arguments (if you've provided several). Another approach is to make a fancy constructor that handles everything:

```
]dinput
:Class cl5
    :Field public name←'baby'
    :Field public age←0
    ∇ Ctor x
      :Access public
      :Implements constructor
      :If ' '=⊃0ρx
          name←x
      :Else
          age←x
      :EndIf
    ∇
:EndClass
```

```
instA←⎕NEW cl5 42
instA.(name age)
instB←⎕NEW cl5 'Charlie'
instB.(name age)
```

```
 baby  42
```

```
 Charlie  0
```

One final example of multiple constructors:

Skip to main content

```
]dinput
:Class cl6
    ∇ None
      :Access public
      :Implements constructor
      ⎕←'No arguments.'
    ∇
    ∇ One x
      :Access public
      :Implements constructor
      ⎕←'1 argument:'x
    ∇
    ∇ Two(a b)
      :Access public
      :Implements constructor
      ⎕←'2 arguments'a b

    ∇
    ∇ Three(a b c)
      :Access public
      :Implements constructor
      ⎕←'3 arguments:'a b c
    ∇
:EndClass
```

We have 0–3. What happens when I call this with more than 3? Let’s see, shall we?

```
instA←⎕NEW cl6
instB←⎕NEW cl6 42
instC←⎕NEW cl6 (2.7 3.1 42)
instD←⎕NEW cl6 (1 2 3 4 5 6 7)
```

```
No arguments.


 1 argument:  42


 3 arguments:  2.7 3.1 42


 1 argument:  1 2 3 4 5 6 7
```

In other words, APL tries to match the specific number of arguments, but if there is no exact match, it passes the array as a single argument to the constructor that takes one argument.

Skip to main content

Destructors

Sometimes when an instance ceases to exist, you want to do some clean-up. For example, when a webserver is closed, you might want to free ports and write a message to the log, etc. This functionality is handled by a destructor.

```
]dinput
:Class cl7
    ∇ Hi
      :Access public
      :Implements constructor
      ⎕←'Hello there!'
    ∇

    ∇ Bye
      :Implements destructor
      ⎕←'See you later!'
    ∇
:EndClass
```

```
inst←⎕NEW cl7
2+3            ⍝ do some work
⎕EX 'inst'     ⍝ Expunge
2×3
```

```
Hello there!

5

See you later!

6
```

Properties

So far, classes have acted pretty much like restricted namespaces. Properties act much like fields/variables, but allow us to take special action when they are set or used.

Skip to main content

Get and Set

Have a look at this code:

```
]dinput
:Class Person

    :Field public name←'-'

    Upper←1∘⎕C
    Lower←⎕C

    :Property Name
    :Access Public
        ∇ text←Get
          :If '-'≡name
              text←'I don''t have a name!'
          :Else
              text←'Hi, my name is ',name,'!'
          :EndIf
        ∇
        ∇ Set text
          name←(Upper 1↑text.NewValue),(Lower 1↓text.NewValue)
        ∇
    :EndProperty

:EndClass
```

`Upper` and `Lower` are two functions (methods) which just uppercase and lowercase. Then we have a block which defines the property `Name`. It doesn't matter that it only has casing difference from the name field, but it is convenient to remember their connection. The way properties work is that they have 1–3 specially named functions. Here, `Name` has `Set` and `Get`. The `Get` and `Set` functions have to be named thus, but you may case them as you want, to fit with whatever coding style you choose. The third one is called `Shape`, but it only applies to a special kind of properties which we won't cover.

`Name` will be treated as a public (due to the `:Access` declaration) field, but instead of directly setting a variable, the `Set` function will be called whenever one uses assignment syntax for `Name`. However, `Set` doesn't just get the new value as argument. Rather, it gets a namespace with some members (you'll see later why). The important member here is `NewValue`, as you can see.

Skip to main content

In the code abobe, `:Field` initialises `name` to be a dash. `Get` will check whether `name` is a dash or not, and respond accordingly. `Set` will accept a character vector and make sure the casing is right (upper initial, rest lower) before assigning to `name`. Let's see if it works:

```
p←⎕NEW Person
p.name
p.Name
p.Name←'anTON'
p.Name
p.name
```

```
-
```

```
I don't have a name!
```

```
Hi, my name is Anton!
```

```
Anton
```

Multiple properties and Default

A class can have more than one property. Let's have a look at a fancier version:

Skip to main content

```
]dinput
:Class Person

    :Field age←0
    :Field name←'-'

    :Property Age
    :Access Public
        ∇ num←get
          num←⌊age
        ∇
    :EndProperty

    ∇ Grow amount
      :Access Public
      age+←amount
    ∇

    Upper←1∘⎕C
    Lower←⎕C

    :Property Default Name
    :Access Public
        ∇ text←Get
          :If '-'≡name
              text←'I don''t have a name!'
          :Else
              text←'Hi, my name is ',name,'!'
          :EndIf
        ∇
        ∇ Set text
          name←(Upper⊃text.NewValue),Lower 1↓text.NewValue
        ∇
    :EndProperty

:EndClass
```

There are three changes here. The most obvious one is the `Age` property and the complementary method `Grow`. The third change is the `Default` declaration for the `Name` property. Normally, objects are passed by reference while arrays are passed by value. But the monadic `⌷` called Materialise has the ability to transform references into values. So if a method has a `Default` property, then monadic `⌷` will yield this property.

Let’s look at those changes in action:

Skip to main content

```
p←⎕NEW Person
p.Name←'BRUNO'
p.Age
p.Grow 3.6
p.Age
p.Grow 0.6
p.Age
⌷p
```

```
0


3


4


Hi, my name is Bruno!
```

On the topic of monadic `⌷`, if you apply it on .NET collections, it materialises the collection’s items, returning an array of the .NET items that the collection consisted of. You can of course make your class have that same behaviour by setting the default property appropriately.

Generic properties

Sometimes a class needs a few properties that have the same or similar getter and setter. Instead of repeating yourself, Dyalog APL lets you collapse the code into a single `:Property` block:

```
]dinput
:Class Person

    :field heightVal
    :field weightVal
    :field ageVal←0

    :Property height,weight,age
    :Access public
        ∇ r←Get x
          r←⌊⍎x.Name,'Val'
        ∇
    :endproperty

:EndClass
```

Skip to main content

```
JUPYTER NOTEBOOK: Input through ⎕ is not supported
```

Notice the comma-separated “name list”. You can also see why the argument to `Get` needs to be a namespace: so that we can determine which property was requested. Here’s a complete listing of the `Person` class:

Skip to main content

```
]dinput
:Class Person

    :field heightVal
    :field weightVal
    :field ageVal←0

    ∇ Birth(h w)
      :Access public
      :Implements constructor
      (heightVal weightVal)←h w
    ∇

    :Property height,weight,age
    :Access public
        ∇ r←Get x
          r←⌊⍎x.Name,'Val'
        ∇
    :endproperty

    ∇ Grow cm
      :Access public
      heightVal+←cm
    ∇

    ∇ Gain kg
      :Access public
      weightVal+←kg
    ∇

    ∇ Lose kg
      :Access public
      weightVal-←kg
    ∇

    ∇ Age y
      :Access public
      ageVal+←y
    ∇

    :property BMI
    :access public
        ∇ bmi←Get
          bmi←⌊0.5+weightVal÷×⍨heightVal÷100
        ∇
    :endproperty

:EndClass
```

Skip to main content

```
p←⎕NEW Person (50 3)
p.Gain 0.7
p.weight
p.Grow 2.5
p.Gain 0.4
p.weight
p.BMI
```

```
3
4
15
```

Display form

The normal display of an object is with a namespace path and object name or class name/"namespace" in brackets. Not very useful:

```
⎕NS ⍬
```

```
#.[Namespace]
```

However, the system function `⎕DF` (Display Form) allows you to change this to any character array:

```
ns←⎕NS ⍬
ns.⎕DF 2 2⍴'yo'
ns
```

```
yo
yo
```

This is similar in spirit to Python's "dunder" method __repr__(). Of course, having a static display form like that isn't much fun. Here is a better usage:

Skip to main content

```
]dinput
:Class Person

    ∇ Birth
        :Implements constructor
        :Access public
        ⎕DF 'baby'
    ∇

    Upper←1∘⎕C
    Lower←⎕C

    :Property Name
    :Access Public
        ∇ text←Get
          :If 0=⎕NC'name'
              text←'I don''t have a name!'
          :Else
              text←'Hi, my name is ',name,'!'
          :EndIf
        ∇
        ∇ Set text
          name←(Upper⊃text.NewValue),Lower 1↓text.NewValue
          ⎕DF name
        ∇
    :EndProperty

:EndClass
```

Now we have a constructor which sets up the initial display form. And every time the `Name` property is `Set`, the display form is updated.

```
p←⎕NEW Person
p
p.Name
p.Name←'anTON'
p.Name
p
```

```
baby
I don't have a name!
Hi, my name is Anton!
Anton
```

As we now know, objects are passed by reference. This means that if we just try to grab the object value, we get a ref rather than the display form, even if the display form is what shows in the

Skip to main content

it: if you have a numeric array, how would you get the character array display form? Well, Format (`⍕`) is APL's "ToString". So `⍕object` will give you whatever argument has been fed to `⎕DF`:

```
p               ⍝ Still a reference, even if it displays as a character array
⎕NC 'p'         ⍝ Name class 9: object
1⎕C⍕p           ⍝ ⍕p is the actual ⎕DF: we can for example upcase it
```

```
Anton
9
ANTON
```

Overtaking objects

Another cool thing you can do is *overtaking*. Remember how APL pads with the a fill element if there are not enough elements to go?

```
10↑3 1 4    ⍝ Overtake a list of ints pads with 0
```

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

If a class has a niladic constructor, then overtaking an instance will create siblings (i.e. new instances of the same class) using the niladic constructor:

```
]dinput
:Class Person

    ∇ Birth
        :Implements constructor
        :Access public
        'I''m an orphan!'
    ∇

    ∇ Naming name
        :Implements constructor
        :Access public
        'I was born with the name ',name
    ∇

:EndClass
```

```
p←⎕NEW Person 'Joe'
```

Skip to main content

```
I was born with the name Joe
I'm an orphan!
I'm an orphan!
 #.[Person]  #.[Person]  #.[Person]
```

Advanced properties

You can also have a `:property numbered` which acts like a normal property, but if you use indices to set or get, those functions are called with a namespace that has an `Indexers` member to tell the function which elements are being asked for.

Remember the `Shape` function of a property we mentioned briefly before? This means that a property can have any (pretend) shape. So when `Get` or `Set` are called, the argument has a member called `IndexersSpecified` which is a Boolean vector indicating which dimensions are being addressed. You can use this, for example, to implement sparse arrays.

You can also have a `:Property keyed` which instead of numeric indices can use any arrays as keys. It is then up to the `Set` and `Get` functions to handle these. Typically you'd want to use character vectors as keys. For such properties you must use indexing, as APL cannot know how many "elements" there are. You can use this to implement dictionary objects.

Inheritance and interfaces

Inheritance

A fundamental idea in OOP is that you can make a more sophisticated object based on a simpler or more general object. For this we have derived or "child" classes. Notice the difference between an instance and a derived class. The instance also inherits from class, but it is fundamentally of the same nature as its sibling instances. A derived class is a new class that you can make instances of. They inherit the members of the base class (although the derived class's code may overwrite base members), but cal also have additional features. Instances of a derived class are also instances of the base class.

Here's an example:

Skip to main content

```
]dinput
:Class Person          ⍝ Person base class
    :field heightVal
    :field weightVal
    :field ageVal←0

    ∇ Birth(h w)
      :Access public
      :Implements constructor
      (heightVal weightVal)←h w
    ∇

    :Property height,weight,age
    :Access public
        ∇ r←Get x
          r←⌊⍎x.Name,'Val'
        ∇
    :endproperty

    ∇ Grow cm
      :Access public
      heightVal+←cm
    ∇

    ∇ Gain kg
      :Access public
      weightVal+←kg
    ∇

    ∇ Lose kg
      :Access public
      weightVal-←kg
    ∇

    ∇ Age y
      :Access public
      ageVal+←y
    ∇

    :property BMI
    :access public
        ∇ bmi←Get
          bmi←⌊0.5+weightVal÷×⍨heightVal÷100
        ∇
    :endproperty

:EndClass
```

Skip to main content

```
]dinput
:Class American: Person
    :field public ssn
    ∇ Birth(w h)
      :Access public
      :Implements constructor :base w h
      ssn←1↓∊('-'@1∘⍕¨⊢+¯1+?)1000 100 10000
    ∇
:EndClass
```

So in the `:Class` header line, we have an additional colon (`:`) and the name of the base class. An `American` is really just another `Person`, but with a social security number. The social security number is given at birth, so we have a constructor that sets `ssn`. But we can't just replace the constructor of the `Person` class, because it performs some important stuff too, namely initialising the weight and height.

Notice the `:base` in the constructor declaration. It tells APL to call the constructor of the base class. `w h` is used to propagate the constructor arguments to the base constructor. In this case, we wrote `w h` out for clarity, but it could also just have said `Birth args … :base args`. APL would have made sure to find the right base constructor (for 2 arguments), and would have thrown an error if the user didn't supply exactly two arguments.

Of course, you can also have a base class that doesn't need any arguments to construct, but a derived class that does need arguments. In such a case, you'd have a monadic derived class constructor, with the line `:Implements constructor :base`. And, of course, you can have the opposite too, and differing number of args, etc. Mix and match as you see fit.

We can extend our classes further:

```
]dinput
:Class NorthAmerican : Person

    :field public language←'English'

    ∇ Birth args
      :Access public
      :Implements constructor :base args
    ∇

:EndClass
```

Skip to main content

```
]dinput
:Class American : NorthAmerican

    :field public ssn

    ∇ Birth(w h)
      :Access public
      :Implements constructor :base w h
      ssn←1↓∊('-'@1∘⍕¨⊢+¯1+?)1000 100 10000
    ∇

:EndClass
```

```
]dinput
:Class Canadian : NorthAmerican

    :field public sin

    ∇ Birth(w h)
      :Access public
      :Implements constructor :base w h
      sin←1↓∊('-'@1∘⍕¨⊢+¯1+?)3⍴1000
    ∇

:EndClass
```

```
]dinput
:Class Swede : Person

    :field public pin
    :field public language←'Swedish'

    ∇ Birth(w h)
      :Access public
      :Implements constructor :base w h
      pin←(2↓⍕100⊥3↑⎕TS),'-'@1⍕10000+¯1+?10000
    ∇

:EndClass
```

So here we have `Americans` and `Canadians` being derived from `NorthAmerican` which is a type of `Person` (yes, really). Each "level" adds its features to the final class's instances.

If you deal with a lot of such derivations, you may want to know the hierarchy of a certain class or instance. Monadic `⎕CLASS` gives you a vector of refs beginning with the class and ending with the

Skip to main content

most basic class. You may also want to know the opposite: which instances does this class have? Monadic `⎕INSTANCES` gives you a vector of refs to all the instances of the given class.

```
c1 c2 c3←{⎕NEW Canadian ⍵}¨(3 50)(4 55)(6 60)
⎕CLASS c1
(c1 c2 c3).⎕DF 'Albert' 'Bert' 'Charlie'
a1←⎕NEW American (7.5 47)
a1.⎕DF 'Dave'
s1←⎕NEW Swede (5 70)
s1.⎕DF 'Erik'
⎕CLASS s1
⎕INSTANCES Person
⎕INSTANCES NorthAmerican
```

```
   #.Canadian    #.NorthAmerican    #.Person

   #.Swede    #.Person

 Albert  Erik  Bert  Charlie  Dave

 Albert  Bert  Charlie  Dave
```

There's another nice system function when dealing with classes (and other scripted objects); `⎕SRC` (SouRCe):

```
↑⎕SRC cl⊣(⎕FIX':class cl' '∇r←SetDF x' ':access public shared' '⎕DF x' 'r←1' '∇' ':
```

```
:class cl
∇r←SetDF x
:access public shared
⎕DF x
r←1
∇
:endclass
```

Interfaces

A Dyalog interface is a script (unsurprisingly `:Interface…:EndInterface`) which defines some

Skip to main content

help ensure a harmonised API.

Consider, for example, the following:

```
]dinput
:Interface FishBehaviour
∇ R←Swim ⍝ Returns description of swimming capability
∇
:EndInterface ⍝ FishBehaviour
```

Note that there isn't any code in `Swim`. It is just a stub for the actual class to fill in. Interfaces can also have multiple such stubs:

```
]dinput
:Interface BirdBehaviour
∇ R←Fly ⍝ Returns description of flying capability
∇

∇ R←Lay ⍝ Returns description of egg-laying behaviour
∇

∇ R←Sing ⍝ Returns description of bird-song
∇
:EndInterface ⍝ BirdBehaviour
```

Now we can define a class with a base class, which implements these methods:

```
]dinput
:Class Penguin: Animal,BirdBehaviour,FishBehaviour
    ∇ R←NoCanFly
      :Implements Method BirdBehaviour.Fly
      R←'Although I am a bird, I cannot fly'
    ∇
    ∇ R←LayOneEgg
      :Implements Method BirdBehaviour.Lay
      R←'I lay one egg every year'
    ∇
    ∇ R←Croak
      :Implements Method BirdBehaviour.Sing
      R←'Croak, Croak!'
    ∇
    ∇ R←Dive
      :Implements Method FishBehaviour.Swim
      R←'I can dive and swim like a fish'
    ∇
```

Skip to main content

A derived class can only have a single base class, but you can use these interfaces to have something resembling multiple inheritance. Notice the `:Class` line. `Animal` is the base class, whereas methods and properties from `BirdBehaviour` and `FishBehaviour` are included in the `Penguin` class.

Advanced OO techniques

Overriding methods

Overridable methods then. Dyalog borrows this terminology from Visual Basic. In C# and Java, they are referred to as “virtual methods”.

If a derived class defines a method that has the same name as a base class method, then that shadows the base class method (although the base class method remains callable with `⎕BASE.MyMethod`). However, if the derived class’ code calls a base class method which in turn calls a function by a name that has been defined both in the base class, and in the derived class, then it is the base class version that gets run. This is of course because the code that calls already is running in the base class. If in such a situation you want the derived class’ method to be called, then you need to *override* the base class method.

In order to do so, two conditions must be met:

1. The base class method must declare itself to be overridable.
2. The derived class method must declare that it is overriding the base class method.

Let’s look at an example. Here is a base class:

```
]dinput
:Class Base
    ∇ r←O
      :Access Public Overridable
      r←'O in Base'
    ∇

    ∇ r←F
      :Access Public
      r←'F in Base'
    ∇

    ∇ r←Caller
      :Access Public
      r←O F
    ∇
:EndClass
```

We have three methods. The two single letter methods just report when they're called. `O` says it is overridable, `F` doesn't. Then there is a `Caller` method which just calls the two single-letter methods.

Here is the companion derived class:

```
]dinput
:Class Derived : Base
    ∇ r←O
      :Access Public Override
      r←'O in Derived'
    ∇

    ∇ r←F
      :Access Public
      r←'F in Derived'
    ∇
:EndClass
```

`O` overrides the base `O`, but `F` doesn't. If we call `Caller` from (an instance of) `Derived`, it will of course execute in the base class, but since `O` has been overridden, it will call the `O` in `Derived`, while `F` will just call the `F` in Base.

```
I←⎕NEW Derived
I.Caller
```

Skip to main content

```
 O in Derived  F in Base
```

Keyed properties

Let's have a more in-depth look at properties, starting with keyed properties. Normally, indexing is for numbers only, e.g. `vector[3 1 4]` and `matrix[3;1 4]` etc. Sometimes you want an array-like thing where individual parts are identified by "keys" (usually character vectors).

For example, instead of referring to the individual columns of a database, you could refer to them by column name. Instead of having to look up each customer ID to find its current row in the database, you'd want to refer to the rows by "name", e.g. the customer ID. Keyed properties allow you to do so, but of course, you have to write the look-up code below the covers, in the property's code.

As you can imagine, the possibilities are endless, but here is a general keyed property skeleton which tells you what the APL code sees:

```
]dinput
:Class ClassK
    :Property Keyed K
    :Access public shared
        ∇ r←Get x
          ⎕←Show x
          →
        ∇

        ∇ Set x
          ⎕←Show x
        ∇
    :EndProperty
    Show←{⍵.(↑{⍵(⍎⍵)}¨⎕NL-⍳9)}
:EndClass
```

You may remember the argument to the setter function from our first treatment of properties. It wasn't very interesting then, but now it is of course critical. Also, note that the getter function now takes an argument. This is because we cannot just return the value of the property; we need to return the correct particular value using the keys.

For now, each function just calls `Show` which is a little, hacky, function that creates a visual also has `→` to force quit instead of actually returning something. This is to avoid having to generate some data which conforms to the shape of the request.

```
ClassK.K[⊂'Abe']←⊂2 5⍴⎕A
ClassK.K['Abe' 'Bob']←3 14
ClassK.K['Abe' 'Bob';'Name' 'Age']←2 2⍴'Abraham' 3  'Robert' 14
```

```
Indexers             Abe
IndexersSpecified  1
Name               K
NewValue            ABCDE
                    FGHIJ


Indexers             Abe  Bob
IndexersSpecified  1
Name               K
NewValue           3 14


Indexers             Abe  Bob    Name  Age
IndexersSpecified  1 1
Name               K
NewValue            Abraham   3
                    Robert   14
```

Notice that keyed properties do not have any particular rank. The first two assignments treat `K` like it's a vector, while the last one treats it as a matrix. APL does check that the indexers and the new values conform according to the rules of scalar extension.

Getting is exactly the same, except that the argument namespace does not have a `NewValue` member:

```
ClassK.K[⊂'Abe']
ClassK.K['Abe' 'Bob']
ClassK.K['Abe' 'Bob';'Name' 'Age']
```

Skip to main content

```
Indexers             Abe
IndexersSpecified  1
Name               K


Indexers             Abe  Bob
IndexersSpecified  1
Name               K


Indexers             Abe  Bob    Name  Age
IndexersSpecified  1 1
Name               K
```

```
]dinput
:Class Database
    :Field public DB←0 3⍴'' '' 0
    :Property Keyed K
    :Access public
        ∇ r←Get x
          (id col)←x.Indexers
          :If id∊DB[;1]
              r←DB[DB[;1]⍳id;'id' 'name' 'age'⍳col]
          :Else
              ⎕SIGNAL 6 ⍝ value error
          :EndIf
        ∇

        ∇ Set x;id;col
          (id col)←x.Indexers
          :If id∊DB[;1]
              DB[DB[;1]⍳id;'id' 'name' 'age'⍳col]←x.NewValue
          :Else
              DB⍪←id,x.NewValue
          :EndIf
        ∇
    :EndProperty
    Show←{⍵.(↑{⍵(⍎⍵)}¨⎕NL-⍳9)}
:EndClass
```

```
i←⎕NEW Database
i.K[⊂'Dave';'name' 'age']←'David' 31
i.K[⊂'Ernie';'name' 'age']←'Ernie' 28
i.K[⊂'Dave';'name' 'age']
```

```
David  31
```

Skip to main content

Numbered properties

A numbered property behaves like an array (conceptually a vector) which is only ever partially accessed and set (one element at a time) via indices. Here's an example:

```
]dinput
:Class ClassN
    :Property Numbered N
    :Access public shared
        ∇ r←Get x
          ⎕←Show x
          →
        ∇

        ∇ Set x
          ⎕←Show x
        ∇

        ∇ r←Shape
          r←2 3
        ∇
    :EndProperty
    Show←{⍵.(↑{⍵(⍎⍵)}¨⎕NL-⍳9)}
:EndClass
```

```
ClassN.N[1;2 3]←'ab'
ClassN.N[1;2 3]
```

```
Indexers   1 2
Name      N
NewValue  a
Indexers   1 3
Name      N
NewValue  b


Indexers   1 2
Name      N
```

It looks very much like our first keyed example, but there is an additional `Shape` function which allows APL to know what this imaginary array looks like. Also, note that the setter (and for that sake the getter) gets called once for each element that needs to be set (or retrieved).

Skip to main content

Using this, you implement a sparse array in much the same way as we did the database. Basically, you'd make a 2-column table of indices and values, and then look up any requested index in the first column to find the corresponding value in the right column. When setting, we'd again look whether the index is already used, and then overwrite that, or if not found, add an entry to our "database". This index lookup can be made very performant by means of a hashed array, `1500⌶`.

Complex numbers

Instead of `a+bi` or `a+b×i`, APL uses `aJb` for scalar atomic complex numbers. In other words, `3+4i` is `3J4` and *i* is `0J1`. The arithmetic functions support complex mathematics where sensible. Of special interest are monadic `+` and `|` and the circular functions `k○Y`. Monadic `+` is the complex conjugate, that is, `a+bi → a-bi`.

```
2J4*0.5
```

```
1.79890744J1.111785941
```

We can combine a real and imaginary parts with `re+0J1×im` but since the complex numbers are atomic (simple scalars) we need a way to split them. For this we have `9○Y` and `11○Y` which would be `Re(Y)` and `Im(Y)` in traditional notation. You might think it odd that we have numbered functions (like the trigonometric functions; sine and cosine are `1○Y` and `2○Y`) but it can actually be really neat because `○` is a scalar function.

Let's say we have a vector of complex numbers `Nv←2J3 0J1 10` then how might we get a 2-row matrix with one row for the real parts and one row for the complex part?

```
Nv←2J3 0J1 10
9 11∘.○Nv
```

```
2  0  10
3  1   0
```

Now, if we have an array `N←2 2⍴2J3 0J1 10 0` and want a two-element vector where each element has the same shape as `N` but the first has the real parts and the second the imaginary

Skip to main content

```
N←2 2⍴2J3 0J1 10 0
9 11○⊂N
```

```
  2 0   3 1
 10 0   0 0
```

The solution can be either a tacit function `(9 11○⊂)N` or the expression `9 11○⊂N` though the outcome is equivalent. First we enclose `N` which makes it a scalar. Then we pair that scalar with a vector `9 11` as arguments to a scalar function, `○`. This makes APL do a scalar extension: `9 11○(⍴9 11)⍴⊂N` or `9 11○N N` or `(9○N)(11○N)`.

Now, if you're familiar with the trigonometric functions, you'll know that negating the left argument of `○` gives you the inverse function. For example, `sin` is `1○Y` and `arcsin` is `¯1○Y`. So `11○Y` extracts the imaginary part into a real number. `¯11○Y` will "put back" a real number into its imaginary place:

```
¯11○3
```

```
0J3
```

Of course, it can't restore the real part, as that was discarded. So... given our 2-element real-and-complex vector from above, how can we reconstitute our original `N`? In other words, how can we convert `(2 0)(3 1)` back to `2J3 0J1`?

```
⊃+/¯9 ¯11○(2 0)(3 1)
```

```
2J3 0J1
```

If the argument is a matrix, we can use

```
N←2 2⍴2J3 0J1 10 0
m←9 11○⊂N
⊃¯9 ¯11+.○9 11○⊂m
```

Skip to main content

```
  2 0   3 1
 10 0   0 0
```

If you deal with complex numbers a lot, you might want to define `J←{⍺+0j1×⍵}` which will then allow you to write `a J b` to form `aJb`, and so `⊃J/vec` for this challenge.

Complex numbers are not just for hard-core mathematicians. Sometimes they are convenient to use as simple scalar 2D coordinates, where the real part represents offset along one axis, and the imaginary part along the other. One benefit in doing so is that some formulas become vastly simpler with this representation. Let's say we have two points in 2D space, `(a,b)` and `(x,y)`, and we want to compute the distance between them. The traditional approach is something like this:

```
(a b x y)←4 6 1 2
0.5*⍨+/2*⍨a b-x y
```

```
5
```

Let's rewrite it given `(u v)←4j6 1j2`:

```
(u v)←4j6 1j2
|u-v
```

```
5
```

This lends itself nicely to a 2-train:

```
Dist←|-
u Dist v
```

```
5
```

Now imagine you need to represent some vectors in 2D space. `3j3` would point north-east. We can now rotate the pointer 90 degrees counter-clockwise, with `0j1×3j3`:

Skip to main content

```
0J1×3J3
```

```
¯3J3
```

Now it points north-west instead. Using `0J¯1×` will rotate clockwise instead. Also, multiplication by `¯1` (which is `0J1*2`) and so rotation by 180 degrees, giving us the oppositely pointed vector, and further multiplication by `0J1` (i.e. to `0J1*3`) is 270 degrees. This means we can get the four corners with `3j3×0j1*⍳4`. Similarly, we can get the four cardinal directions with `3J0×0J1*⍳4`:

```
3J3×0J1*⍳4
3×0J1*⍳4
```

```
¯3J3 ¯3J¯3 3J¯3 3J3
```

```
0J3 ¯3 0J¯3 3
```

Some more cheatsheet about vectors: `+v` is reflection by x-axis, `+-v` is by y-axis, `|v` is length, `×v` is unit vector in that direction, `k××v` is vector of length `k` in that direction. If you want to scale vector `v` with scaling factor `k`, do `k×v`, and to rotate vector `v` by the angle of vector `w`, do `v××w`.

You can represent a number of “moves” in 2D space as complex vectors, say `moves←1j2 0j3 ¯1j0`. This means move 1 right and 2 up, then 3 right, then 1 down. Given such a moves sequence, where do we end up?

```
moves←1j2 0j3 ¯1j0
+/moves
```

```
0J5
```

What points did we pass through?

```
+\moves
```

Skip to main content

```
1J2 1J5 0J5
```

Although we may want to say

```
+\0,moves
```

```
0 1J2 1J5 0J5
```

to include the origin.

Conversely, given a set of points, what is the corresponding moves sequence?

```
2-⍨/0 1J2 1J5 0J5
```

```
1J2 0J3 ¯1
```

Sometimes, it is convenient to deal with the angle (upwards from due east) and magnitude (pointer length) instead of the “coordinates”. We can already get the magnitude (absolute value) with `|Y` but the angle (or phase) is `12○Y`. Side note: the `12○Y` is the one called `atan2()` in other languages. Remember how convenient it was to use the scalar `○` function with a 2-element left argument `9 11`. For that same reason, `|Y` exists as `○` argument, which is 10 of course. So `10 12○Y` gives you the magnitude and phase. Of course, we can use `10 12∘.○Y` and `10 12○⊂Y` like before.

How about the other way, if we have an angle and magnitude and want to combine them into a single complex number? Remember how we used `{α+¯11○ω}` before. This is then `{α×¯12○ω}` (or, if you prefer, `¯10 ¯12×.○Y`).

Counting words, faster

As a practical application, let’s consider the counting of words in a string. There are many ways to do that, but I’ll show you how an array oriented approach can give tremendous speed-ups. But first we have to generate some test data. Since actual letters don’t matter, we’ll just have a text

Skip to main content

programmer would, of course, jump to regular expressions. As we've seen, Dyalog APL has a really powerful support for the PCRE-flavour of regular expressions built-in:

```
≢'[^,]+'⎕S 3⊢',YYY,,YYYYYY,,XXXXXX,YYYYYYXXYYY,YYYXXYYYXX,XX,XXYYYXXXX,YYY'
```

```
8
```

`⎕S` is an operator which takes the regex on its left and what to return for each match on its right. 3 is a special code meaning the pattern number, which is just 0 because we only have one regex. Then we tally (count) that with `≢` and we're done.

Another approach is to just split on the delimiter: a good job for `⊆` here. If you give it a Boolean mask as left argument, it isolates runs corresponding to runs of 1s, discarding the elements corresponding to 0s:

```
',' ≠',YYY,,YYYYYY,,XXXXXX,YYYYYYXXYYY,YYYXXYYYXX,XX,XXYYYXXXX,YYY'      ⍝ non-delim
','(≠⊆⊢)',YYY,,YYYYYY,,XXXXXX,YYYYYYXXYYY,YYYXXYYYXX,XX,XXYYYXXXX,YYY'  ⍝ groups co
```

```
0 1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1

  YYY  YYYYYY  XXXXXX  YYYYYYXXYYY  YYYXXYYYXX  XX  XXYYYXXXX  YYY
```

Read `≠⊆⊢` as "the difference partitions the right argument". What remains is to count the partitions:

```
','(≢≠⊆⊢)',YYY,,YYYYYY,,XXXXXX,YYYYYYXXYYY,YYYXXYYYXX,XX,XXYYYXXXX,YYY'
```

```
8
```

This solution has an issue, but before we get to that, let's compare the performance of the "pure" APL solution to the regex solution.

```
cmpx '≢''[^,]+''⎕S 3⊢t' 's(≢≠⊆⊢)t' ⊣ s←',' ⊣ ⎕←≢t←',XY'[/⍨?1e6⍴3] ⊣ 'cmpx'⎕CY'dfns'
```

Skip to main content

```
1997880
  ≢'[^,]+'⎕S 3⊢t → 5.4E¯2 |    0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  s(≢≠⊆⊢)t        → 3.2E¯2 | -42% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

On about 2 million characters we’re saving two thirds of the running time by using the split and count approach over regex. Quite a bit faster, but there is more scope: it is problematic that we split the array to count the pieces, as this has to make a new (pointer!) array.

So our issue is that we need to ignore multiple spaces. We actually need to do *edge detection*. If we have a text, say `,YYY,,YYYYYY,,` we want to see whenever we go from a non-space to a space (or the opposite). The only gotcha is at the end, if there are no trailing spaces, we will miss the last word. APL has the “find” function `⍷`:

```
'ss'⍷'mississippi'
```

```
0 0 1 0 0 1 0 0 0 0 0
```

It indicates the beginning of its left argument (“the top-left corner”) in its right argument. So now we can create an is-space mask, and look for `0 1`.

```
','=',YYY,,YYYYYY,,'          ⍝ is-space mask
0 1⍷','=',YYY,,YYYYYY,,'      ⍝ locate star points for 0 1 patterns
+/0 1⍷','=',YYY,,YYYYYY,,'    ⍝ count them
```

```
1 0 0 0 1 1 0 0 0 0 0 0 1 1
```

```
0 0 0 1 0 0 0 0 0 0 0 1 0 0
```

```
2
```

However, it counts wrong here:

```
+/0 1⍷','=',YYY,,YYYYYY'
```

Skip to main content

So we need to add a “space” to the end.

```
','{+/0 1⍷1,⍨α=ω}',YYY,,YYYYYY'
```

```
2
```

How do they stack up, speed-wise?

```
cmpx '≢''[^,]+''⎕S 3⊢t' 's(≢≠⊆⊢)t' 's{+/0 1⍷1,⍨α=ω}t'⊣ s←',' ⊣ ⎕←≢t←',XY'[/⍨?1e6⍴3]
```

```
2000210
  ≢'[^,]+'⎕S 3⊢t    → 5.4E¯2 |   0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  s(≢≠⊆⊢)t          → 3.2E¯2 | -42% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  s{+/0 1⍷1,⍨α=ω}t → 2.4E¯2 | -57% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

Unfortunately, that’s a bit slower, but perhaps we can think of another way to exploit our idea to look for the 0 to 1 transition? Since we’re looking for `0 1`, we can just insert `<` between elements, using a windowed reduction:

```
','{+/2</1,⍨α=ω}',YYY,,YYYYYY'
```

```
2
```

As before, we append a 1 *after* we’ve calculated our binary mask. We could, of course, also have written that as

```
','{+/2</α=ω,α}',YYY,,YYYYYY'
```

```
2
```

which is adding an extra ‘,’ at the end, before calculating the mask. When we concatenate the space to the string, APL has to create a copy of the whole string with one additional byte at the end, which is costlier than appending a 1 to a bit-Boolean array as we did in the first version.

Skip to main content

Note also that `2</` takes care of duplicate spaces. What about performance?

```
cmpx '≢''[^,]+''⎕S 3⊢t' 's(≢≠⊆⊢)t' 's{+/0 1⍷α=ω,α}t' 's{+/2</1,⍨α=ω}t'⊣ s←',' ⊣ ⎕←≢
```

```
2000759
  ≢'[^,]+'⎕S 3⊢t  → 5.4E¯2 |    0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  s(≢≠⊆⊢)t        → 3.2E¯2 |  -42% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  s{+/0 1⍷α=ω,α}t → 2.4E¯2 |  -56% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  s{+/2</1,⍨α=ω}t → 1.7E¯4 | -100%
```

That's a crushing improvement! Let's zoom in a bit, by removing the slower versions from the comparison:

```
cmpx 's(≢≠⊆⊢)t' 's{+/2</(α=ω),1}t'⊣ s←',' ⊣ ⎕←≢t←',XY'[/⍨?1e6⍴3]
```

```
1999750
  s(≢≠⊆⊢)t         → 3.2E¯2 |    0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
  s{+/2</(α=ω),1}t → 1.7E¯4 | -100%
```

Lookup without replacement

"Lookup without replacement" is a very old (and famous) programming problem in the APL world. You can see a thorough investigation of this in video form, too.

Consider two vectors,

```
L←⎕←'abacba'
R←⎕←'baabaac'
```

```
abacba
```

```
baabaac
```

Dyadic iota `⍳` lets us find the first index of occurrence of the elements in `R` in `L`:

Skip to main content

```
L⍳R
```

```
2 1 1 2 1 1 4
```

However, what if we wanted the first `b` in `R` to “consume” the first `b` in `L` so that the second `b` in `R` would have to contend with the index of the *second* `b` in `L`? That is, we want some function which gives `2 1 3 5 6 7 4`. You could call it “iota without replacement”.

Let’s begin by labeling the elements so we can see what goes where:

```
'a1' 'b1' 'a2' 'c1' 'b2' 'a3' ⍳ 'b1' 'a1' 'a2' 'b2' 'a3' 'a4' 'c1'
```

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

As we numbered the `a`s (which otherwise all match each other) and the `b`s, the right pairs get matched up. If you recall the chapter about `⍋`, you may also recall what `⍋⍋` does. While `⍋` gives use the indices that will sort, `⍋⍋` gives us the positions that each element will occupy in the sorted result.

```
↑L(L⍳L)(⍋⍋L⍳L)⊣L
```

```
a b a c b a
1 2 1 4 2 1
1 4 2 6 5 3
```

The first line is the data and the second is the indices of the first occurrences (in other words, all identical items will get the same index). The third line is the position that each will occupy when sorted. That means that identical elements get consecutive positions.

For example, you can see that the first `b` gets 4 (because there are 3 `a`s) and the second gets 5. This almost solves the problem.

However, there are a couple of issues:

Skip to main content

2. The two arrays must have equally many of each unique element
3. The unique elements must initially occur in the same order

Why these conditions?

1. is because otherwise the purely numeric “labels” will match the wrong things.
2. is because otherwise one element’s “label” will be paired up with the label of a different value element of the other array.
3. is because otherwise identical “labels” numbers refer to two entirely different things, and so the matching won’t give a meaningful result.

But if these conditions are met, we get the right result:

```
L←'abacba'
R←'aaabcb'
(⍋⍋L⍳L)⍳⍋⍋R⍳R
```

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

The first `a` in `R` gets paired with the element in position 1 of `L`, and the second `a` in `R` goes with the element in position 3, and the third goes with the last element of `L`.

Let’s have a stab at how we can ensure that all conditions are eliminated, and then we’ll have our solution. Since we’re going to look up elements of `R` in `L` anyway, we can use indices into `L` (that is `L⍳R`) instead of the lookup of `R` into itself (`R⍳R`) This ensures that elements of `R` are labelled with “`L`’s labelling system”.

```
L←'abacba'
R←'bcabaa'
↑L(L⍳L)(⍋⍋L⍳L)
↑R(L⍳R)(⍋⍋L⍳R)
```

Skip to main content

```
a b a c b a
1 2 1 4 2 1
1 4 2 6 5 3
```

```
b c a b a a
2 4 1 2 1 1
4 6 1 5 2 3
```

The first line (of each group) is the data, the second line is the first-positions of that data in `L`. The third is the progressive labeling of that. Now you can see that the first `a` is labeled 1 for both `L` and `R` and the first `b` is labeled 4 for both `L` and `R`.

```
(⍋⍋L⍳L)⍳(⍋⍋L⍳R)
```

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

We now have that the first `b` of `R` takes out element 2 of `L`, and the `c` takes out element 4 of `L` and so on. But this still requires both sides to have the same set of elements and equally many of each element. How can we ensure that there are equally many of each unique element on each side? Well, if you think about it, `L,R` and `R,L` must necessarily have the same set in equal proportions. But this also gives us way more elements than we need. We'll take care of that later.

```
(⍋⍋L⍳L,R)⍳(⍋⍋L⍳R,L)
```

```
2 4 1 5 3 6 9 7 11 8 10 12
```

Note that this sequence begins with what we want, and now we have equal proportions, so we've eliminated issue 2. We just need to reshape (or take) to chop the unneeded elements:

```
((⍴L)⍴⍋⍋L⍳L,R)⍳((⍴R)⍴⍋⍋L⍳R,L)
```

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

Now it works even though we have a `d` in `R` which doesn't occur in `L`. In accordance with the

Skip to main content

chopped the left list of labels to the length of `L`, that’s what we get.

```
L←'abacba'
R←'bcdabaaaaa'
((⍴L)⍴⍋⍋L⍳L,R)⍳((⍴R)⍴⍋⍋L⍳R,L)
```

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

And so, we’ve taken care of issue 1 (different sets of elements). This algorithm can also be adapted to use with any-rank arrays by using `≢` instead of monadic `⍴` and `↑` instead of dyadic `⍴` and `⍪` instead of `,`. Let’s have a look back at what we did. Consider:

```
↑(L R)←'abacba' 'baabaac'
```

```
abacba
baabaac
```

We then labeled the elements:

```
↑('a1' 'b8' 'a2' 'c12' 'b9' 'a3')('b8' 'a1' 'a2' 'b9' 'a3' 'a4' 'c12')
```

```
 a1  b8  a2  c12  b9  a3
 b8  a1  a2  b9   a3  a4  c12
```

And looked those labels up:

```
('a1' 'b8' 'a2' 'c12' 'b9' 'a3') ⍳ ('b8' 'a1' 'a2' 'b9' 'a3' 'a4' 'c12')
```

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

But actually, we don’t need the original values (the letters); the numeric labels are enough:

```
(1 8 2 12 9 3) ⍳ (8 1 2 9 3 4 12)
```

Skip to main content

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

And how did we get those labels?

```
↑(L R)←'abacba' 'baabaac'
(⍴L)⍴⍋⍋L⍳L,R
(⍴R)⍴⍋⍋L⍳R,L
```

```
abacba
baabaac
```

```
1 8 2 12 9 3
```

```
8 1 2 9 3 4 12
```

So now we can define our function:

```
pdi ← {((⍴⍺)⍴⍋⍋⍺⍳⍺,⍵)⍳(⍴⍵)⍴⍋⍋⍺⍳⍵,⍺} ⍝ Progressive Dyadic Iota
```

```
'abacba' pdi 'bcabaa'
```

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

Here’s an example. We want to fill a plane with multiple classes, using first-come, first-serve. We may want to ask: for each customer, will they fit on the plane? Say we have a plane like ‘11bbbpeepee’, where 1 is first class, b is business, p is economy plus (extra legroom at emergency exits), and e is regular economy. We now have a bunch of customers coming to buy seats: ‘1bbbpppeeeee’. That’s one 1st class customer, three business people, three want more legroom, and a load of regular people.

```
'11bbbpeepee' pdi '1bbbpppeeeee'
```

```
1 3 4 5 6 9 12 7 8 10 11 12
```

Skip to main content

Being that the plane only has 11 seats, we can see that one plus and one economy will not fit (indicated by the 12s), but we just want a Boolean, not the actual seating. Progressive dyadic iota (or iota without replacement) asks “For each element, where would it go in the remaining elements?” Now we need to ask “For each element, does it fit in (i.e. is it in) the remaining elements?”.

“is it in” is APL’s `∊`. Just note that the arguments of `∊` and `⍳` are “reversed” in that the array we look up in is on the left for `⍳` and on the right for `∊`, so we just swap the parts of our function and substitute `∊` for the middle `⍳`:

```
pde ← {((⍴⍵)⍴⍋⍋⍺⍳⍵,⍺)∊((⍴⍺)⍴⍋⍋⍺⍳⍺,⍵)} ⍝ Progressive Dyadic Epsilon
```

```
'11bbbpeepee' pde '1bbbpppeeeee'
```

```
1 1 1 1 1 1 0 1 1 1 1 0
```

Alternatively, we could just call the function with swapped arguments:

```
'1bbbpppeeeee' {((⍴⍺)⍴⍋⍋⍺⍳⍺,⍵)∊(⍴⍵)⍴⍋⍋⍺⍳⍵,⍺} '11bbbpeepee'
```

```
1 1 1 1 1 1 0 1 1 1 1 0
```

This function is “membership without replacement”, or “progressive dyadic epsilon”. Did you notice the pattern? We are taking two functions and modifying them in a consistent manner. This calls for an operator!

```
WithoutReplacement←{((⍴⍺)⍴⍋⍋⍺⍳⍺,⍵)⍺⍺(⍴⍵)⍴⍋⍋⍺⍳⍵,⍺}
```

```
↑ (p c)←'11bbbpeepee' '1bbbpppeeeee'
p ⍳WithoutReplacement c
p ∊WithoutReplacement c
```

Skip to main content

```
11bbbpeepee
1bbbpppeeeee
```

```
1 3 4 5 6 9 12 7 8 10 11 12
```

```
1 0 1 1 1 1 1 1 1 1 1
```

Notice how the APL code reads much like normal English.

User commands

We’ve used the user command `]RunTime` to compare the speed of two otherwise equivalent expressions elsewhere. You may also have encountered system commands like)save,)clear and)off. The system commands are an integral part of the interpreter (and have been so for a very long time). That is, for Dyalog APL, they are written in C.

System commands are not APL functions, but rather a way to directly interact with the system. Thus, they do not follow APL syntax. Instead, they act more like commands on a command line. That’s why they’re called commands. Sometimes, this non-syntactic way is really useful tor day-to-day stuff, and you’d want that for your APL code as well. This is where *user commands* come in. They have exactly the same syntax model as system commands, they just begin with a `]` instead of a `)`.

The only thing built into the interpreter is that whenever it sees a line in the session beginning with] it takes the rest of that line and calls `⎕SE.UCMD` with the line as a character vector argument. Dyalog APL comes pre-installed with a “user command processor”, i.e. a function `⎕SE.UCMD` which takes care of the rest. The default user command system is tightly integrated with `SALT`, but you could write your own drop-in, should you with to do so. Dyalog APL also comes loaded with more than 100 pre-defined user commands, some are simple and complex. All are written in APL, and you can change them as you see fit.

```
] -?
```

Skip to main content

```
97 commands:

 ARRAY          Compare  Edit
 CALC           Factors  FromHex  PivotTable  ToHex
 DEVOPS         DBuild  DTest
 EXPERIMENTAL   Get
 FILE           CD  Collect  Compare  Edit  Find  Open  Replace  Split  ToLarge  ToQu
 FN             Align  Calls  Compare  Defs  DInput  Latest  ReorderLocals
 LINK           Add  Break  Create  Export  Expunge  GetFileName  GetItemName  Import
 NS             ScriptUpdate  Summary  Xref
 OUTPUT         Box  Boxing  Disp  Display  Find  Format  HTML  Layout  Plot  Repr  R
 PERFORMANCE    Profile  RunTime  SpaceNeeded
 SALT           Boot  Clean  Compare  List  Load  Refresh  RemoveVersions  Save  Set
 TOOLS          ADoc  APLCart  Calendar  Config  Demo  Help  Version
 TRANSFER       In  Out
 UCMD           UDebug  ULoad  UMonitor  UNew  UReset  USetup  UVersion
 WS             Check  Compare  Document  FindRefs  FnsLike  Locate  Map  Names  Name

]             ⍝ for general user command help
] -??         ⍝ for brief info on each command
]grp -?       ⍝ for info on the "GRP" group
]grp.cmd -?   ⍝ for info on the "Cmd" command of the "GRP" group
```

At this point, we should mention that all these user commands have a whole host of options which you can specify with various arguments or modifiers. It would be too much to go into details about it all, but you can always get documentation about any user command with `]cmdname -?`, for example:

```
]calls -?
```

```
]FN.Calls

Produce the calling tree of a function in a class/namespace/scriptfile
    ]Calls <function> [<namespace>]

]Calls -?? ⍝ for more information and examples
```

Now that we are talking about the special syntax of user commands, the command processor has another few tricks up its sleeve.

Skip to main content

If for some reason you want to capture the result of a user command, you can do so with `]varname←cmdname`. If you want to silence a user command, you can do that with `]←cmdname`.

Remember that we said everything after the `]` is passed as argument to `⎕SE.UCMD`? That means that you can even call user command under program control: `⎕SE.UCMD 'cmdname'`. Anything else you'd write on the line just goes inside that character vector.

Let's have a look at some of the available user commands.

]CD

There are simple things like `]cd`:

```
]cd
```

```
/Users/jeremy/repos/apl-cultivations/cultivations/contents
```

`]cd`, in its niladic form, shows the interpreter's current working directory. You can set the current working directory, too, by

```
]cd /Users/stefan/work/notebooks
]cd /Users/stefan/work/notebooks/cultivations/contents
```

```
* Command Execution Failed: Unable to change directory: /Users/stefan/work/notebooks
```

```
* Command Execution Failed: Unable to change directory: /Users/stefan/work/notebooks
```

Note that when you set the current working directory this way, `]cd` will echo back the directory it changed *from*, not the one it changed *to*.

]DInput

If you've ever wanted to enter or paste a multi-line statement into the session, you can use

Skip to main content

What is a multi-line statement? Remember that you don't have to assign dfns before you use them; you can insert them inline. And dfns may have multiple lines. Effectively, you then have a single multi-line statement. Now, as soon as you press Enter in the session, you code will be executed, and if it has any un-closed braces, e.g. `2+{a←⍳10` it will fail. However, if you enter `]dinput` you will get a new prompt indicated by a dot `·` and then you can begin entering (or pasting) multi-line statements. `]dinput` will keep track of your brace-nesting level and indicate it with more dots.

You can also just type `]dinput f←` and then paste a multi-line dfn there, beginning on that line. That'll define it in the workspace.

Another important use for `]dinput` is when you write multi-line functions in a Jupyter notebook cell, as you will have seen already in many places in this book.

]Calls

There are also various code analysis tools, like `]calls`. It will produce a calling tree:

```
]calls  getEnvir  ⎕se.SALTUtils
```

Skip to main content

```
Level 1:  →getEnvir
  F:rlb               F:splitOn           F:splitOn1st        F:GetUnicodeFile    F:

Level 2: getEnvir→UnixFileExists

Level 2: getEnvir→SALTsetFile

Level 2: getEnvir→GetUnicodeFile
⍝ Read a Unicode (UTF-8 or even UCS-2) file
⍝ This version allows excluding specific 1-byte characters before the translation
⍝ This prevents TRANSLATION errors in classic interpreters
  F:condEncl     F:numReplace   F:Special      F:Uxxxx

Level 3: GetUnicodeFile→condEncl

Level 3: GetUnicodeFile→Special

Level 3: GetUnicodeFile→Uxxxx

Level 3: GetUnicodeFile→numReplace
⍝ fromto is the list of lists of numbers to replace
  F:num

Level 4: numReplace→num
  F:isChar

Level 5: num→isChar

Level 2: getEnvir→splitOn1st
⍝ Split on 1st occurrence of any chars in str

Level 2: getEnvir→splitOn

Level 2: getEnvir→rlb
```

This says that the `getEnvir` function in `⎕SE.SALTUtils` calls these six functions, which in turn call the other listed functions, each at its level. This is really useful if you're trying to extract some utility function and need to know its dependencies.

]Settings

A workspace stores information about each function; who was it last modified by, and when. This information can also be saved in script files with `]save` if you turn on "atinfo tracking". You can turn turn that on with `]settings track atinfo`. Then you can list which functions were recently modified: `]Latest 20180501 -by=Fred`

Skip to main content

```
]settings
```

```
 compare        APL
 cmddir         /Users/jeremy/MyUCMDs:/Applications/Dyalog-18.2.app/Contents/Resourc
 debug          0
 editor         notepad
 edprompt       1
 fndels         0
 mapprimitives  1
 newcmd         auto
 track
 varfmt         xml
 workdir        .:/Applications/Dyalog-18.2.app/Contents/Resources/Dyalog/Library/Co
```

These are basically like OS environment variables, but used just by `SALT`. For example, `edprompt` determines if the editor should ask you before writing changes to scripted items back to their source file. `varfmt` determines how `]save` should save variables; as XML or as APL statements that produce the value. `cmddir` tells `SALT` where to look for user commands.

As you can gather, you can just drop your own or downloaded user commands into the mentioned `/MyUCMDs` dir and you're in business. Watch the webinar about how to write your own user commands!

]ReorderLocals

If you've ever written anything moderately complex as a tradfn, you may have been annoyed that, as you edit along, your list of local variables on the header line is not neatly ordered. `]reorderlocals` allows you to sort the header row of all (or some of) the functions currently in the workspace: `]reorderlocals MyFn` or `]reorderlocals F*` or just `]reorderlocals`.

]CopyReg

If you're on Windows, you have a few goodies especially for you. When the time comes to upgrade your Dyalog between major versions, but you've spent a whole year customising the current version to your liking. There is a user command that allows you to easily migrate your settings between versions:

Skip to main content

```
]CopyReg 17u64 -to=18u64
```

does the job (you may need admin privileges, though).

The command processor

At this point, we should mention that all these user commands have a whole host of options which you can specify with various arguments or modifiers. It would be too much to go into details about it all, but you can always get documentation about any user command with `]cmdname -?`:

]Summary

There are also commands that let you get an overview of things:

```
]summary ⎕se.Parser
```

```
Name        Scope   Size  Syntax
Parse       P      17128   r1f
Propagate           2744   r2f
Quotes              2256   r1f
Switch              2616   r2f
deQuote             1512   r1f
fixCase              120   r2f
if                    48   r2f
init        PC     14040   n1f
splitParms          3400   r1f
sqz                10872   r2f
upperCase          10960   r2f
xCut               10648   r2f
```

This analyses the `⎕SE.Parser` class and tells you a little bit about each function. `P` means public, `C` constructor, and the syntax is whether they have a result, number of arguments, and type (function/monadic operator/dyadic operator).

Skip to main content

]XRef

`]xref` will produce a cross reference of all items in a namespace, which ones call or reference which, how they do so (global/local) and what type they all are.

]Box

You may already know about `]box`. It is, for example, responsible for that nice boxed output you can see on TryAPL. You can turn that on and off, and decide exactly how you want it to display things with the user command. For now, let's just see what the current settings are in this notebook:

```
]box ?
```

```
]Box OFF -style=min -view=min -trains=box -fns=off
```

]Rows

There is a lesser known, but very useful, companion to `]box` called `]rows`. Probably, by now, you've entered a statement that caused way too much output, so your session would just scroll and scroll. Right? Well, the `]rows` user command can protect you against that but limiting output to the current height and width of your window.

```
]rows ?
```

```
]Rows OFF -style=long -fold=off -fns=off -dots=·
```

So if you do `]rows on -fold=3` it will cut any output four lines before the bottom of your screen, insert a row of dots (or whichever character you choose, e.g. `]rows on -fold=3 -dots=~`) and then display the last three lines of the output. It will then also (by default) not wrap lines that are too

Skip to main content

long, but rather will cause them to continue beyond the right edge of the screen (scroll horizontally to see it). Again, see `]box -?` and `]rows -?` for the full details.

]Disp,]Display

If you prefer boxing off during normal work, but want to display some results boxed here and there, you can use `]disp` and `]display` for that. `]disp` is much like `]box -style=mid` and `]display` is like `]box -style=max`. As you saw above, the notebook uses `-style=min`, but that doesn't always give you enough information:

```
2 3⍴'' (⍳3) (0 0⍴0) 'a'
```

```
   1 2 3
a         1 2 3
```

OK, we've go three empty (or are they filled with spaces?) elements. But what are they *really*?

```
]display 2 3⍴'' (⍳3) (0 0⍴0) 'a'
```

```
┌→────────────────────┐
↓ ┌⊖┐ ┌→────┐ ┌⊖┐     │
│ │ │ │1 2 3│ ⌽0│     │
│ └─┘ └~────┘ └~┘     │
│     ┌⊖┐     ┌→────┐ │
│ a   │ │     │1 2 3│ │
│ -   └─┘     └~────┘ │
└∊────────────────────┘
```

Now we can see what exactly each thing is; we've got two empty character vectors and one 0-by-0 numeric matrix. We can also see that the `a` is a scalar, and the `1 2 3`s are vectors (not e.g. one-row matrices).

]ADoc

If you comment your code using markdown, you can use `]adoc` to automatically generate some

Skip to main content

that has comments and syntax information gleaned from your code.

]Calendar

For a quick calendar, do:

```
]Calendar
```

```
     March 2024
Su Mo Tu We Th Fr Sa
                1  2
 3  4  5  6  7  8  9
10 11 12 13 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
31
```

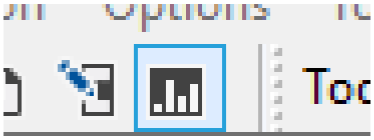
You can also specify a year or a month and a year, for example:

```
]Calendar January 1969
```

```
    January 1969
Su Mo Tu We Th Fr Sa
          1  2  3  4
 5  6  7  8  9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30 31
```

]Chart (Windows only)

If you are on Windows, you'll have a handful more user commands than if not. Perhaps the coolest of them is the Chart Wizard. It has a button in the IDE:



Skip to main content

But it is also available as a user command. Try e.g. `]chart (⍳50)×↓|1○(500÷⍨⍳50)∘.×⍳50`. If you're not on Windows, you can still generate charts using SharpPlot (for which `]chart` is just a GUI). Here's some example code for that, and the chapter on plotting in this book.

]Version

If you ever run into trouble with your APL system, you may want to know the version numbers of various parts and dependencies of your APL system:

```
]version -extended
```

```
  Dyalog  18.2.45505 64-bit Unicode, BuildID 50b14a3f
          /Applications/Dyalog-18.2.app/Contents/Resources/Dyalog/lib/htmlrenderer.dy
  OS      Darwin 23.2.0 Darwin Kernel Version 23.2.0: Wed Nov 15 21:53:18 PST 2023; r
  CPUs    10
  Link    3.0.19
  SALT    2.9
  UCMD    2.51
  .NET    (unavailable)
  WS      18.2
  Conga   Version: 3.4.1612
          loaded from: /Applications/Dyalog-18.2.app/Contents/Resources/Dyalog/lib/co
          Copyright 2002-2022 Dyalog Ltd. GnuTLS 3.6.15
          Copyright (c) 2000-2021 Free Software Foundation, Inc.  Copyright 2002-2022
  SQAPL   (unavailable)
```

]UVersion

If you're having trouble with a user command, you can get the version number of it with:

```
]uversion calendar
```

```
 framework:  2.51
 command:    ]TOOLS.Calendar
 source:     /Applications/Dyalog-18.2.app/Contents/Resources/Dyalog/SALT/spice/jsuti
 version:    1.18
 revision:   1574
 commit:     2019 01 29 Adam: Help
```

Skip to main content

]Compare

There is actually a whole family user commands, all called `Compare`. They are in the groups `SALT`, `WS`, `ARRAY`, `FN`, and `FILE`. You can use them to compare two similar items, just may have done file diffs, but here you can do them on various things related to APL. For example, `]WS.Compare path1/ws1 path2/ws2` compares two workspaces, and `]NS.Compare #.ns1 #.ns2` compares two namespaces. Of course, if your items are stored in script files, you could use your favourite diff tool, but it probably doesn’t have any understanding of the APL code involved.

]Document

If you want a “hardcopy” of your workspace or part of it, you can use `]document` to list all items, describe what they are, and show how they look if typed into the session.

]FindRefs

If you work with a lot of objects, especially if they point to each other, you may find `]findrefs` useful. It will follow all pointers (refs) and report everything. For example,

```
A ← ⎕NS '' ⋄ B←C←D←A
V ← 0 C 2 99

]findrefs
```

```
#: followed 6 pointers to reach a total of 2 "refs"

 Name
 #
 #.B+4 more
```

]Names

Skip to main content

shell-style *glob* expression:

```
⎕CY 'salt'
]names 3.1 -filter=*l*
```

```
3.1: GetUnicodeFile PutUTF8File SALTsetFile disableSALT enableSALT regClose regGetHa
```

The `-filter=` option can also take full regexes,

```
]names 3.1 -filter=/.*\d.*/ ⍝ Tradfn names containing a digit
```

```
3.1: PutUTF8File
```

Note that the regex pattern is implicitly anchored to the beginning and end, so your pattern must match the whole name.

]Map

A really cool user command is `]map` which draws a tree view of your workspace or (if given an argument) a specific namespace:

```
]map ⎕SE.Dyalog
```

Skip to main content

```
⎕SE.Dyalog
·   ∇ Serial
·   Array
·   ·   ~ DEBUG sysVars
·   ·   ∇ Deserialise DeserialiseQA L QA RoundtripQA Serialise SerialiseQA ∆NS ∆NSin
·   ·   ∘ Ed Inline Is
·   ·   serialise
·   ·   ·   ~ cc cr cs di ec nl oc or os qu sp
·   ·   ·   ∇ Any0 Basic Char Clean Empty Esc HiRank Join Lead0 Mat Nested Ns Null N
·   ·   ·   ∘ _Paren_ _Sub_
·   Callbacks
·   ·   ~ loaded
·   ·   ∇ BootSALT FontChange LoadFonts NJoin SECreate SetBoxButton WSLoaded startup
·   Hooks
·   ·   ∇ Deregister Handle Init Norm Num Register Registered
·   Out
·   ·   ~ OUTSpace allSettings cmds
·   ·   ∇ Dft Filter Init Rows SD SetCallback flipBox pfnops timestamp
·   ·   ∘ Box
·   ·   B
·   ·   ·   ~ fns state style trains view
·   ·   F
·   ·   ·   ~ includequadoutput state stop timestamp
·   ·   L
·   ·   ·   ~ pfkey state
·   ·   R
·   ·   ·   ~ dots fns fold state style
·   SALT
·   ·   ~ List
·   SEEd → ⎕SE.[SessionEditor]
·   Utils
·   ·   ~ APLcartTableCache APLcartTableTime lc uc
·   ·   ∇ APLcart APLcartTable CD CDshy Config ExpandConfig Version View condRavel c
·   ·   ∘ currying nabs
·   ·   SALT_Data → ⎕SE.[Namespace]
·   ·   qa
·   ·   ·   ∇ ExpandConfig
```

The tree structure itself are the nested namespaces, while the lists of names are ordered by type; `~` are variables, `∇` are functions, `∘` are operators. It also displays ref-names and where they point.

]Peek

But perhaps the most powerful user command of them all is `]peek`. It allows you to “peek” into a different workspace, execute an expression there, then come back with the result, all without polluting or modifying neither the current workspace, nor the workspace that was peeked into:

Skip to main content

```
]peek dfns queens 5
```

```
⍟ · · · ·  · ⍟ · · ·
· · ⍟ · ·  · · · · ⍟
· · · · ⍟  · · ⍟ · ·
· ⍟ · · ·  ⍟ · · · ·
· · · ⍟ ·  · · · ⍟ ·
```

How to place five queens on a 5-by-5 chess board without them being able to capture each other, all without even loading any utilities! How about that? :-)

There are, of course, many, many more user commands, and new versions of Dyalog usually adds more.

]APLCart

Version 18.2 of Dyalog added the `]aplcart` user command. APLCart is searchable collection of short APL phrases. It is a goldmine of answers to APL-related questions of the type “How do I do X in APL?” for a surprisingly wide range of X. The `]aplcart` command makes this resource available directly in the session,

```
]aplcart Append scalar to each column of matrix
```

```
X, Y, Z: any type array  M, N: numeric array  I, J: integer array  A, B: Boolean arr
────────────────────────────────────────────────────────────────────────────────────
Xm⍪Ys  ⍝ Append scalar to each column of matrix
────────────────────────────────────────────────────────────────────────────────────
Showing 1 of 1 matches
```

`]aplcart` has a number of options and capabilities. You can, for example, filter results by regular expression:

```
]aplcart /highest|lowest/
```

Skip to main content

```
X, Y, Z: any type array  M, N: numeric array  I, J: integer array  A, B: Boolean arr
──────────────────────────────────────────────────────────────────────────────────────
M∨N                        ⍝ Greatest Common Divisor of M and N
M∧N                        ⍝ Lowest Common Multiple of M and N
⌈/N                        ⍝ Maximum of N
⌊/N                        ⍝ Minimum of N
⍳⍨Nv                       ⍝ Assign ranking based on non-descending scores Nv (ties a
⍸⍨Nv                       ⍝ Assign ranking based on non-descending scores Nv (ties a
Is(⊃>⍥|⌽,)Js               ⍝ Choose the number closer to zero (the left one if tied)
Is(⌈(⊃>⍥|⌽,)⌊)Js           ⍝ Choose the number closer to zero (the positive one if ti
Ms{⍵×⍺÷⍵[⊃⍒|⍵]}Nv          ⍝ Scale Nv so the maximum element is Ms
{s←0 ⋄ ⌈/{s⊢←0⌈s+⍵}¨⍵}Nv   ⍝ Largest sum of any contiguous subvector
──────────────────────────────────────────────────────────────────────────────────────
Showing 10 of 12 matches (-list=<n> to show up to <n>; -list to show all)
```

or ask it to generate the URL to the corresponding search query on the website itself:

```
]aplcart /highest|lowest/ -url
```

```
https://aplcart.info?q=/highest|lowest/
```

A `-b` opens your default web browser on the corresponding results page instead of displaying it in the session.

]Get

18.2 also added the `]Get` command. There is a lot to this (it comes with comprehensive documentation; see `]get -??`), and we'll only skim the surface here. `]Get` provides a unified interface for quickly getting data (and code) into Dyalog from a multitude of different sources, including local files, by URL, or from git repositories. Note that this is intended as a development aid, and not something that should be relied upon during runtime or production.

From `]Get -??`:

> `]Get` is a development tool intended as a one-stop utility for quickly getting bringing resources into the workspace while programming. Do not use at run time, as exact results may vary. Instead, use precisely documented features like `⎕JSON`, `⎕CSV`, `⎕XML`, and `⎕FIX` in combination with loading tools like `⎕NGET`, `HttpCommand`, `⎕SE.Link.Import`, etc.

> `]Get` supports importing directories and the following file extensions (files with any other extensions are imported as character vectors): apla, aplc, aplf, apli, apln, aplo, charlist, charmat, charstring, charvec, class, csv, dcf, dcfg, dws, dyalog, function, interface, js, json, json5, operator, script, tsv, xml, zip

Here's an example of fetching–and decoding–a remote XML-file:

```
]Get raw.githubusercontent.com/Dyalog/MiServer/master/Config/Logger.xml
```

```
#.Logger
```

```
]disp #.Logger
```

```
┌─┬────────────┬────────────────┬───┬─┐
│0│Logger      │                │   │3│
├─┼────────────┼────────────────┼───┼─┤
│1│active      │0               │   │5│
├─┼────────────┼────────────────┼───┼─┤
│1│anonymousIPs│1               │   │5│
├─┼────────────┼────────────────┼───┼─┤
│1│directory   │%SiteRoot%/Logs/│   │5│
├─┼────────────┼────────────────┼───┼─┤
│1│interval    │10              │   │5│
├─┼────────────┼────────────────┼───┼─┤
│1│prefix      │                │   │1│
└─┴────────────┴────────────────┴───┴─┘
```

]Repr

Also making its debut in 18.2 was `]repr`. It takes an APL value and returns (by default) an expression that produces this value.

```
]repr #.Logger
]repr 0 1 1 1 1 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1 1
```

Skip to main content

```
(6 5ρ0 'Logger' '' (0 2ρ⊂'') 3 1 'active' (,'0') (0 2ρ⊂'') 5 1 'anonymousIPs' (,'1')
```

```
(0,(4/1),0 0 1 0 1 1,(8/0),1 1)
```

That's useful enough, but it has a few other handy tricks up its sleeve, too. Perhaps you want to convert a specific APL value to csv?

```
]repr 'ABC'⍪2 3ρ⍳6 -f=csv
```

```
"A","B","C"
1,2,3
4,5,6
```

or maybe you need help with showing the correct parenthesing of a train?

```
]repr +⌿÷1⌈≢
```

```
(+⌿)÷(1⌈≢)
```

Plotting with SharpPlot

This Cultivation was hosted by Nicolas Delcros. Nicolas also gave a presentation on SharpPlot at the Dyalog '13 user conference, and there are several blog posts available on the topic, too.

SharpPlot is a professional charting and typesetting engine that ships with Dyalog APL. If you want to draw graphs or plot functions using APL, SharpPlot has you covered. SharpPlot comes in two versions, firstly a native .NET bundle that can be used through Dyalog's .NET integration, and secondly as a pure APL workspace, referred to as `Causeway`. Whilst they're identical in terms of functionality, the former tends to be faster, but the latter obviously has the advantage of working everywhere Dyalog works, without the need to have access to .NET. We'll be using the `Causeway` approach here.

Let's kick this off with an example! First we need to pull in two functions from the `sharpplot`

```
'InitCauseway' 'View' ⎕CY'sharpplot'
```

```
⎕RL←16807 1   ⍝ fixed seed for random numbers
```

Let’s write a function we can use to generate some data to plot,

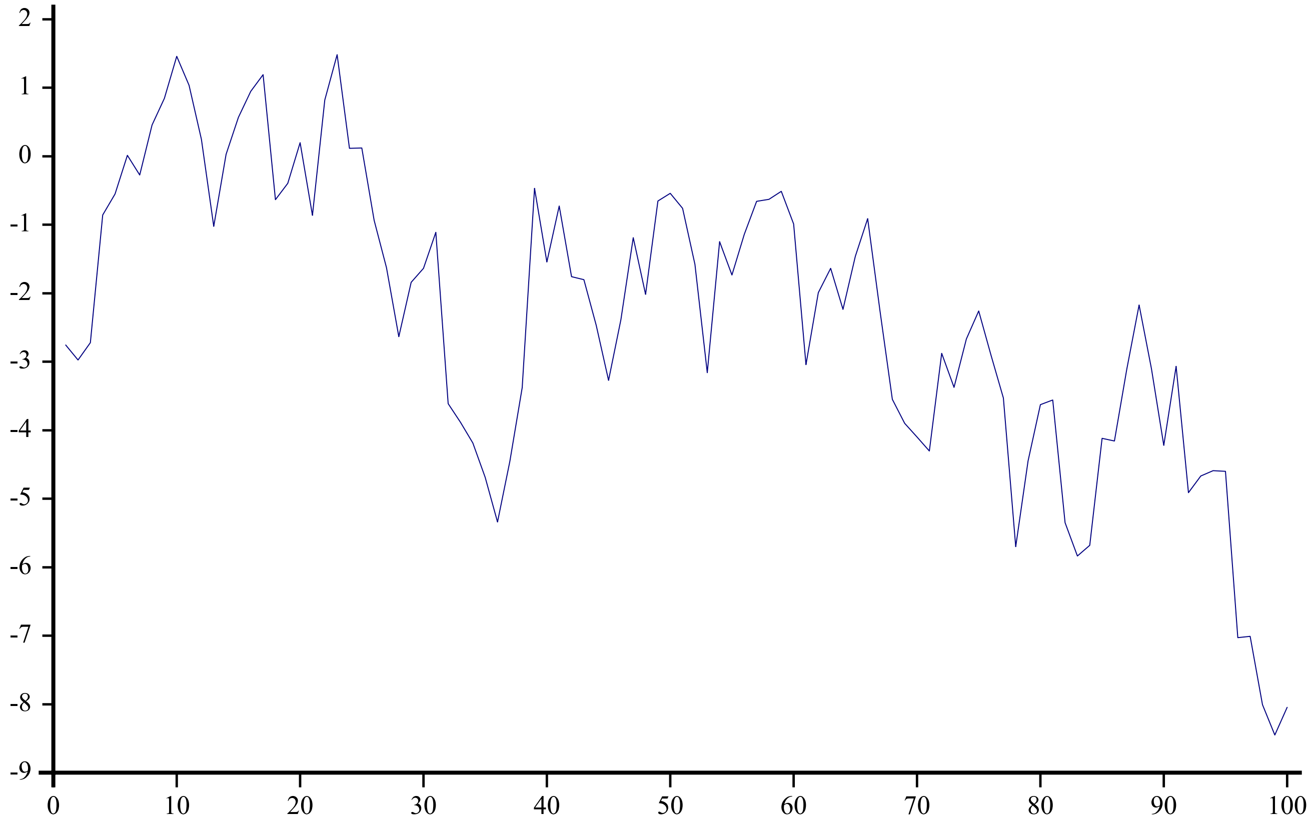
```
]dinput
NormalRandom ← {
  depth ← 1000000000                      ⍝ Randomness depth
  (x y) ← ⊂[1+⍳⍴,⍵](?(2,⍵)⍴depth)÷depth ⍝ Two random variables within ]0;1]
  ((-2×⍟x)*0.5)×1○○2×y                   ⍝ https://en.wikipedia.org/wiki/Box-Muller_
}
```

Now we can draw a SharpPlot line graph:

```
line ← ⊃↓+\NormalRandom 5 100
InitCauseway ⍬
sp ← ⎕NEW Causeway.SharpPlot
sp.DrawLineGraph ⊂line                    ⍝ Single argument must be enclosed

sp.SaveSvg 'plot1.svg' Causeway.SvgMode.FixedAspect ⍝ Write the graph image to disk
```

Skip to main content



Unfortunately, many of the SharpPlot functions don't actually return anything, making them tricky to use inside a dfn. Here's a somewhat hideous workaround for this,

```
]dinput
Plot ← {
    do ← {⍎'⍺⍺ ⍵ ⋄ ⍵' ⋄ ⍺⍺}
    _ ← InitCauseway do ⍬
    sp ← ⎕NEW Causeway.SharpPlot
    _ ← sp.DrawLineGraph do ⍵
    sp
}
```

You may have gathered that what we get returned from this function is a SharpPlot instance, with a bunch of methods and properties. You might draw another line graph, or any other kind of graph, or add some notes, perhaps.

Skip to main content

Ok, let's plot some more involved data. Here we have some personal account-keeping, showing expenditures of different types across a set of dates. A quirk here is that SharpPlot uses so-called "OLE dates" which are one-off to international day numbers (IDN — the number of days since the beginning of 1899-12-31).

```
'date'⎕CY'dfns'
↑(date¨43578+?20ρ100)(('Groceries' 'Entertainment' 'Subscription')[?20ρ3])(20+?20ρ1
```

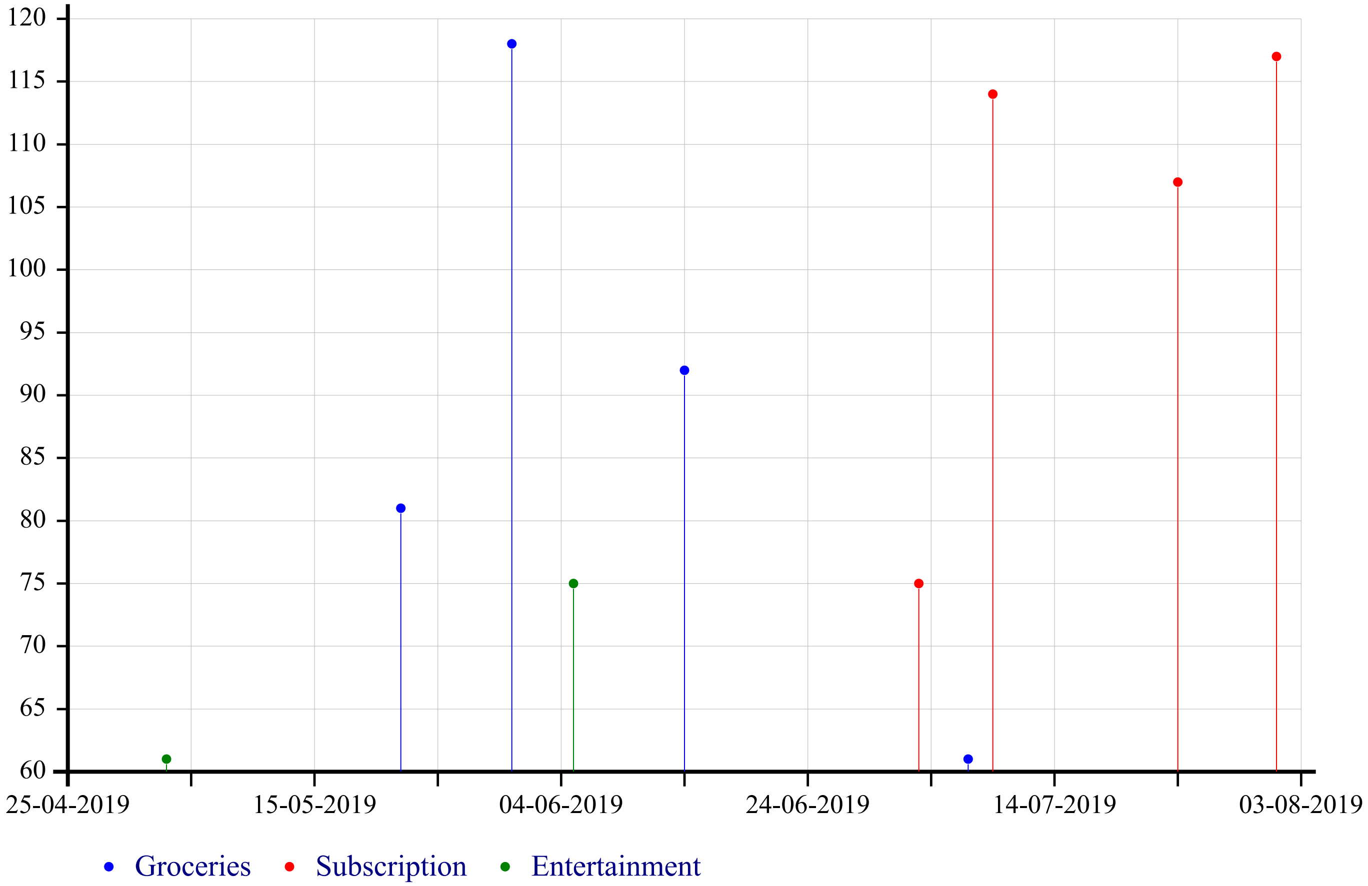
```
2019 5 19 0 0 0 0  2019 7 24 0 0 0 0  2019 8 1 0 0 0 0  2019 7 14 0 0 0 0  2019 7 3
        Groceries      Entertainment     Entertainment          Groceries      Ente
              107                 36               105                 58
```

```
∇ sp ← Budget size;count;dates;oledates;type
  dates ← date¨43578+size?10×size
  type ← 'Groceries' 'Entertainment' 'Subscription'[?sizeρ3]
  count ← 20+?sizeρ100

  oledates ← {1+2 ⎕NQ'.' 'DateToIDN'ω}¨dates
  InitCauseway ⍬
  sp ← ⎕NEW Causeway.SharpPlot
  sp.SplitBy⊂type ⍝ single argument must be enclosed
  sp.ScatterPlotStyle ← Causeway.ScatterPlotStyles.(GridLines+Risers)
  sp.SetColors System.Drawing.Color.(Blue Red Green)
  sp.SetMarkers Causeway.Marker.Bullet
  sp.XAxisStyle ← Causeway.XAxisStyles.(Date)
  sp.XDateFormat ←'dd-MM-yyyy'
  sp.DrawScatterPlot count oledates
∇
```

```
gr ← Budget 10
gr.SaveSvg 'plot02.svg' Causeway.SvgMode.FixedAspect
```

Skip to main content



Here’s a subset of Our World In Data’s dataset on COVID-19. We’ve picked out the data for United States, Canada, United Kingdom, France and Denmark, plotting the new cases per million, and new deaths per million over time, starting from January, 2022. We did a bit of data slicing and date conversion outside APL, detailed here, in order for us to be able to focus mainly on the plotting aspect.

Skip to main content

```
∇ {sp}←OwidCovidData;Causeway;InitCauseway;View;countries_to_plot;csv;data;date;dat
miss ← ¯1E300 ⍝ missing value
csv ← {⎕CSV ⍵ ⍬ 4} '/Users/stefan/work/data/covid_subset2.csv'
dates ← {⍵[⍋⍵]}∪date ← 20 1⎕DT csv[;2]
csv[;2] ← date

locations ← ∪location←csv[;1]
row ← csv[;1 2]⍳↑locations∘.{⍺ ⍵}dates
csv ⍪← (⊂'')(⊂'')miss miss
data ← csv[row;3 4]

fields_to_plot ← 'New cases per million' 'New deaths per million'
countries_to_plot ← 'United States' 'Canada' 'United Kingdom' 'France' 'Denmark'

'InitCauseway' 'View'⎕CY'sharpplot'
InitCauseway ⍬
sp ← ⎕NEW Causeway.SharpPlot
sp.MissingValue ← miss
sp.SetTrellis ≢fields_to_plot

:For field :In ⍳≢fields_to_plot
    sp.NewCell
    sp.Heading ← field⊃fields_to_plot
    sp.MarginBottom ← 70
    sp.SetKeyText ⊂countries_to_plot
    sp.YAxisStyle ← Causeway.YAxisStyles.LogScale
    sp.XAxisStyle ← Causeway.XAxisStyles.(Date+MonthlyTicks)
    sp.XDateFormat ← 'MMM-yy'
    values ← ↓data[;;field]
    sp.DrawLineGraph values dates
    sp.DrawKey ⍬
:EndFor
∇
```

```
InitCauseway ⍬
cov ← OwidCovidData
cov.SaveSvg 'plot03.svg' Causeway.SvgMode.FixedAspect
```

```
FILE NAME ERROR: /Users/stefan/work/data/covid_subset2.csv: Unable to open file ("N
OwidCovidData[2] csv←{⎕CSV ⍵ ⍬ 4}'/Users/stefan/work/data/covid_subset2.csv'
                      ∧
```

```
VALUE ERROR: Undefined name: cov
      cov.SaveSvg'plot03.svg'Causeway.SvgMode.FixedAspect
      ∧
```

Skip to main content

plot3

Array programming techniques

There are a few things one can do to make APL look more… APL. What really characterises “classic” code is control structures and especially loops. Modern APL has control structures, too, and loops can easily be done with `¨`. So those are really the features you want to *avoid*.

Try to think of differentiation between cases in terms of any of:

- Boolean masks
- Mathematical relationships
- Commonality between cases

FizzBuzz

Maybe FizzBuzz would be a good example. The classic approach (other than “I don’t think that’s possible”!) is a loop. Possibly two loops, an outer one for N and an inner one for the 3, 5 list. Instead, let’s try processing the entire list `⍳35` at once, using any one or more of the above.

To start off, we can find which numbers are divisible by 3 or 5 with an outer product:

```
]rows on ⍝ don't wrap output cells
```

```
Was OFF
```

```
mask←⎕←0=3 5∘.|⍳35
```

```
0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0
0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
```

which gives us a nice mask for when we need Fizz and when we need Buzz, but when do we need the number itself? Let’s create an additional row in the mask array that holds 1 if neither of the Fizz

Skip to main content

```
(ṽ⌿⍪⊢)mask
```

```
1 1 0 1 0 0 1 1 0 0 1 0 1 1 0 1 1 0 1 0 0 1 1 0 0 1 0 1 1 0 1 1 0 1 0
0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0
0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1
```

So far, everything has been pretty clean. Things will start to get dirty now because FizzBuzz essentially is a mixed-type problem, but we can still try to stick with Array operations until the very end.

We can zero out unwanted numbers by multiplying the mask with the numbers,

```
(⍳35)×@1⊢(ṽ⌿⍪⊢)mask
```

```
1 2 0 4 0 0 7 8 0 0 11 0 13 14 0 16 17 0 19 0 0 22 23 0 0 26 0 28 29 0 31 32 0 34 0
0 0 1 0 0 1 0 0 1 0  0 1  0  0 1  0  0 1  0 0 1  0  0 1 0  0 1  0  0 1  0  0 1  0 0
0 0 0 0 1 0 0 0 0 1  0 0  0  0 1  0  0 0  0 1 0  0  0 0 1  0 0  0  0 1  0  0 0  0 1
```

If we split that up a bit, we end up with

```
nums←⍳35
mat←(ṽ⌿⍪⊢)0=3 5∘.|nums
mat×@1⍨←nums
```

The next step is to replace all 1s in row 2 with 'Fizz', and the 1s in row 3 with 'Buzz':

```
mat←(⊂'Fizz')@⊢@2⊢mat
mat←(⊂'Buzz')@⊢@3⊢mat
```

```
mat
```

```
1 2     0  4     0     0  7 8     0     0  11     0  13 14     0  16 17     0  19
0 0  Fizz  0     0  Fizz  0 0  Fizz     0   0  Fizz   0  0  Fizz   0  0  Fizz   0
0 0     0  0  Buzz     0  0 0     0  Buzz   0     0   0  0  Buzz   0  0     0   0  B
```

Skip to main content

```
]dinput
FizzBuzz←{
    nums←⍳⍵
    mat←(⍱⌿⍪⊢)0=3 5∘.|nums
    mat×@1⍨←nums
    mat←(⊂'Fizz')@⊢@2⊢mat
    mat←(⊂'Buzz')@⊢@3⊢mat
    mat←(⊂⍬)@(∊∘0)mat
    ,⌿mat
}
```

```
FizzBuzz 35
```

```
 1  2  Fizz  4  Buzz  Fizz  7  8  Fizz  Buzz  11  Fizz  13  14  FizzBuzz  16  17  Fi
```

This isn’t, perhaps, how you should implement FizzBuzz in an industrial context, and it does do things that impact performance, but it is a pretty good demonstration of applying the array approach to a traditionally loopy problem.

Justify it

Let’s do another example: take a character matrix and justify it without looping over the lines. This means distributing the trailing spaces into the existing word separations.

For example,

```
In publishing and graphic design,
Lorem ipsum is a placeholder text
commonly used to demonstrate the visual form
of a document or a typeface
without relying on meaningful content.
```

becomes

```
In    publishing    and    graphic   design,
Lorem    ipsum    is    a    placeholder    text
commonly used to demonstrate the visual form
of      a      document     or     a     typeface
```

Skip to main content

This isn't a particularly difficult problem for a single line, but if we enforce treating the contiguous ravelled data in one go, it becomes a bit more tricky. So, let's say we have `t` as the above 5-by-44 matrix. It follows that our result must also be a 5-by-44 matrix.

There are two obvious approaches. One is to move some spaces from the end of the lines to the middle by reordering elements. The other is to determine for each space how many copies if it we need (0 to remove it, 1 to keep it, and more to extend it). Let's go with the latter method.

```
t←(5 44⍴'In publishing and graphic design,           Lorem ipsum is a placeholder t
```

```
t
```

```
In publishing and graphic design,
Lorem ipsum is a placeholder text
commonly used to demonstrate the visual form
of a document or a typeface
without relying on meaningful content.
```

The first step is identifying spaces. Luckily, scalar extension allows use to do `spaces←' '=t`:

```
spaces←⎕←' '=t
```

```
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0
0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 1
```

How might we use that to create a mask (Boolean matrix) for the characters we want to *keep*?

```
keep←⎕←~⌽∧\⌽spaces
```

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0
```

Skip to main content

Next we need to get the number of trailing spaces on each line,

```
cols←⊃⌽⍴keep
trail←⎕←cols-+/keep
```

```
 11 11 0 17 6
```

Since we need to distribute extra width over inner spaces, we need to know how many inner spaces each line has, so we can divide the trailing width by that.

```
inner←⎕←trail-⍨+/spaces
```

```
 4 5 6 5 4
```

We now need to distribute the extra spaces over the inner spaces, noting that they may not be evenly distributable. We can just take the floor throughout, and the strategically add 1 here and there, preferably as evenly distributed as possible. We could start at the beginning and add one to each interspace until we're “fully adjusted”. If you look at the example above, that's what we did:

```
In□□□□publishing□□□□and□□□□graphic□□□design,
```

The first three have 4 and the last one has 3. How might we determine the number of spaces that need one extra space? Well, it's the remainder of dividing total needed spaces by how many spaces we have. For example, if we need to have 14 spaces and only have 5 spots then it'd be 4. We can express this as:

```
mod←⎕←inner|trail
```

```
 3 1 0 2 2
```

The base extension per line is

```
div←⎕←⌊trail÷inner
```

```
2 2 0 3 1
```

Now we can create a mask for spaces that need an extra space:

```
extra←⎕←spaces×mod≥⍤0 1+\spaces×keep
```

```
0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Now we’re ready to put the parts together to get a replication factor for each character.

```
replication←⎕←keep+extra+div(×⍤0 1)spaces×keep
```

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

and, finally, we can apply the transformation:

```
(⍴ t)⍴(,replication)/,t
```

```
In    publishing    and    graphic   design,
Lorem    ipsum   is   a   placeholder   text
commonly used to demonstrate the visual form
of     a     document    or    a    typeface
without   relying   on  meaningful  content.
```

Combine it all into a dfn, and we get:

Skip to main content

```
]dinput
Justify ← {
    spaces ← ' '=ω
    keep ← ~⌽∧\⌽spaces
    trail ← +/~keep
    inner ← |trail - +/spaces
    mod ← inner|trail
    div ← ⌊trail÷inner
    extra ← spaces×mod(≥⍤0 1)+\ spaces×keep
    replication ← keep+extra+div(×⍤0 1)spaces×keep
    (ρ ω)ρ(,replication)/,ω
}
```

```
Justify t
```

```
In    publishing    and    graphic   design,
Lorem    ipsum   is   a   placeholder   text
commonly used to demonstrate the visual form
of     a     document    or    a    typeface
without   relying   on  meaningful  content.
```

Function application

Some operators apply (to) their operands in intricate ways. How do you get a clearer picture of what they actually do? Let's take outer product `∘.f` as an example.

```
10 20 30∘.×1 2 3 4
```

```
10 20 30  40
20 40 60  80
30 60 90 120
```

Sure, ok, but what actually happened? It may seem simple, but what about:

```
(3 2ρ10×ι6)∘.×(2 4ρι8)
```

Skip to main content

```
 10  20  30  40
 50  60  70  80

 20  40  60  80
100 120 140 160


 30  60  90 120
150 180 210 240

 40  80 120 160
200 240 280 320


 50 100 150 200
250 300 350 400

 60 120 180 240
300 360 420 480
```

What exactly got paired up with what? Here’s a trick you can use to analyse derived functions, that is both functions modified by operators and all tacit functions in general. Let’s replace the function (the operand) with a function which doesn’t actually do the computation, but rather tells us what the computation would be:

```
10 20 30{'(',⍺,'×',⍵,')'}1 2 3
```

```
( 10 20 30 × 1 2 3 )
```

`×` is scalar. We can model that too:

```
10 20 30{⍺{'(',⍺,'×',⍵,')'}¨⍵}1 2 3
```

```
 ( 10 × 1 )  ( 20 × 2 )  ( 30 × 3 )
```

```
(3 2⍴10×⍳6)∘.{⍺{'(',⍺,'×',⍵,')'}¨⍵}(2 4⍴⍳8)
```

Skip to main content

```
( 10 × 1 )    ( 10 × 2 )    ( 10 × 3 )    ( 10 × 4 )
( 10 × 5 )    ( 10 × 6 )    ( 10 × 7 )    ( 10 × 8 )

( 20 × 1 )    ( 20 × 2 )    ( 20 × 3 )    ( 20 × 4 )
( 20 × 5 )    ( 20 × 6 )    ( 20 × 7 )    ( 20 × 8 )


( 30 × 1 )    ( 30 × 2 )    ( 30 × 3 )    ( 30 × 4 )
( 30 × 5 )    ( 30 × 6 )    ( 30 × 7 )    ( 30 × 8 )

( 40 × 1 )    ( 40 × 2 )    ( 40 × 3 )    ( 40 × 4 )
( 40 × 5 )    ( 40 × 6 )    ( 40 × 7 )    ( 40 × 8 )


( 50 × 1 )    ( 50 × 2 )    ( 50 × 3 )    ( 50 × 4 )
( 50 × 5 )    ( 50 × 6 )    ( 50 × 7 )    ( 50 × 8 )

( 60 × 1 )    ( 60 × 2 )    ( 60 × 3 )    ( 60 × 4 )
( 60 × 5 )    ( 60 × 6 )    ( 60 × 7 )    ( 60 × 8 )
```

Now we can see what’s going on! Even better if we use indices as arguments:

```
({⊂'α[',(⍕1⊃ω),';',(⍕2⊃ω),']'}¨⍳2 3)∘.{α{'(',α,'×',ω,')'}¨ω}({⊂'ω[',(⍕1⊃ω),';',(⍕2⊃
```

```
(α[1;1]×ω[1;1])   (α[1;1]×ω[1;2])   (α[1;1]×ω[1;3])   (α[1;1]×ω[1;4])
(α[1;1]×ω[2;1])   (α[1;1]×ω[2;2])   (α[1;1]×ω[2;3])   (α[1;1]×ω[2;4])

(α[1;2]×ω[1;1])   (α[1;2]×ω[1;2])   (α[1;2]×ω[1;3])   (α[1;2]×ω[1;4])
(α[1;2]×ω[2;1])   (α[1;2]×ω[2;2])   (α[1;2]×ω[2;3])   (α[1;2]×ω[2;4])

(α[1;3]×ω[1;1])   (α[1;3]×ω[1;2])   (α[1;3]×ω[1;3])   (α[1;3]×ω[1;4])
(α[1;3]×ω[2;1])   (α[1;3]×ω[2;2])   (α[1;3]×ω[2;3])   (α[1;3]×ω[2;4])


(α[2;1]×ω[1;1])   (α[2;1]×ω[1;2])   (α[2;1]×ω[1;3])   (α[2;1]×ω[1;4])
(α[2;1]×ω[2;1])   (α[2;1]×ω[2;2])   (α[2;1]×ω[2;3])   (α[2;1]×ω[2;4])

(α[2;2]×ω[1;1])   (α[2;2]×ω[1;2])   (α[2;2]×ω[1;3])   (α[2;2]×ω[1;4])
(α[2;2]×ω[2;1])   (α[2;2]×ω[2;2])   (α[2;2]×ω[2;3])   (α[2;2]×ω[2;4])

(α[2;3]×ω[1;1])   (α[2;3]×ω[1;2])   (α[2;3]×ω[1;3])   (α[2;3]×ω[1;4])
(α[2;3]×ω[2;1])   (α[2;3]×ω[2;2])   (α[2;3]×ω[2;3])   (α[2;3]×ω[2;4])
```

We can make this an “eXplanation” operator:

```
X←{f←αα ⋄ α←⊢ ⋄ '(',α,(⎕CR'f'),ω,')'}
```

Skip to main content

How does it work? First it captures its operand `⍺⍺` as `f`, then it makes `⍺` into identity which is a common trick to make ambivalent functions. Finally, it strings together the left arg, the function character representation, and the right arg.

```
'abc'∘.(×X)'DEF'
```

```
(a×D)  (a×E)  (a×F)
(b×D)  (b×E)  (b×F)
(c×D)  (c×E)  (c×F)
```

OK, now that we have a grip on `∘.f`, let's look at `f.g`.

```
'abc'(+X).(×X)'DEF'
```

```
((a×D)+((b×E)+(c×F)))
```

The result is enclosed which shows us that if the arguments are vectors (as in this case) then the result is a scalar. What happens with higher-rank arguments?

```
'abc'(+X).(×X)(3 2⍴'DEFGHI')
```

```
((a×D)+((b×F)+(c×H)))  ((a×E)+((b×G)+(c×I)))
```

The left argument was a 3-element vector and the right argument a 3-by-2 matrix. We can see how the left argument cells were distributed to the right argument cells.

```
(2 3⍴'abcdef')(+X).(×X)3 2⍴'DEFGHI'
```

```
((a×D)+((b×F)+(c×H)))  ((a×E)+((b×G)+(c×I)))
((d×D)+((e×F)+(f×H)))  ((d×E)+((e×G)+(f×I)))
```

OK, now it is getting more interesting. The left arg was `2 3⍴` and the right was `3 2⍴`. The result became `2 2⍴`. In fact, the rule is that `f.g` removes the last axis of the left argument and the first

Skip to main content

axis of the right argument, so the result has the shape `(¯1↓⍴⍺),(1↓⍴⍵)`. So if the left arg is shape `2 4 3` and the right arg is `3 5 1` the result should be shape `2 4 5 1`:

```
⍴(2 4 3⍴0)+.×(3 5 1⍴0)
```

```
2 4 5 1
```

Let’s return to `∘.f` for a moment. What is the rule about the shape of the result of *that*?

```
⍴(2 4 3⍴0)∘.×(3 5 1⍴0)
```

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

So the shape of `∘.f` is `(⍴⍺),(⍴⍵)`. `∘.f` and `f.g` are definitely related! In fact, Iverson suggested that the slightly anomalous `∘` in `∘.f` be replaced with a number that indicates how many axes to combine. This way `0.f` would be `∘.f`. However, there is a more general alternative: the rank operator, `⍤`. This powerful operator is one many struggle with. Let’s explore it! Let’s use a slightly modified version of `X`:

```
X←{f←⍺⍺ ⋄ ⍺←'' ⋄ '(',(⍕⍺(⎕CR'f')⍵),')'}
```

```
(⊂X)2 3 4⍴⎕A
```

```
(   ⊂  ABCD )
(      EFGH )
(      IJKL )
(           )
(      MNOP )
(      QRST )
(      UVWX )
```

This just shows enclosing the rank-3 alphabet.

```
(⊂X)⍤¯1⊢2 3 4⍴⎕A
```

Skip to main content

```
(    ⊂   ABCD  )
(        EFGH  )
(        IJKL  )

(    ⊂   MNOP  )
(        QRST  )
(        UVWX  )
```

Let’s begin with negative rank, which is often what you really want. `f⍤¯N ⊢ B` applies the function to cells of rank `(≢⍴B)-N`. So in this case the array had rank 3, and the function was applied to sub-arrays of rank 3-1, that is 2, that is, matrices.

```
(⊂X)⍤¯2⊢2 3 4⍴⎕A
```

```
(    ⊂   ABCD  )
(    ⊂   EFGH  )
(    ⊂   IJKL  )

(    ⊂   MNOP  )
(    ⊂   QRST  )
(    ⊂   UVWX  )
```

Here, the function was applied to sub-arrays of rank 3-2, that is 1, i.e. vectors. Now lets try positive rank.

```
(⊂X)⍤1⊢2 3 4⍴⎕A
```

```
(    ⊂   ABCD  )
(    ⊂   EFGH  )
(    ⊂   IJKL  )

(    ⊂   MNOP  )
(    ⊂   QRST  )
(    ⊂   UVWX  )
```

`f⍤N` applies the function to sub-arrays of rank `N`. So `f⍤1` digs in until it finds vectors.

```
(⊂X)⍤2⊢2 3 4⍴⎕A
```

Skip to main content

```
(   ⊂  ABCD )
(      EFGH )
(      IJKL )

(   ⊂  MNOP )
(      QRST )
(      UVWX )
```

So, too, does `⍤2` apply the function to matrices. What about `⍤0`? It applies the function to sub-arrays of rank 0, i.e. scalars. `⊂` obviously isn't a useful function on scalars, but some functions are, for example, `∊`. Consider the following nested array:

```
m←⎕←2 2⍴(2 3⍴⎕A)(3 2⍴⎕A)(2 2⍴⎕A)(3 3⍴⎕A)
```

```
 ABC  AB
 DEF  CD
      EF
 AB   ABC
 CD   DEF
      GHI
```

It has four scalars. We can apply `∊` on each scalar:

```
∊⍤0⊢m
```

```
ABCDEF
ABCDEF

ABCD
ABCDEFGHI
```

Notice the description: *on each*. In general, `⍤0` is the same as `¨`:

```
∊¨m
```

```
 ABCDEF  ABCDEF
 ABCD    ABCDEFGHI
```

Skip to main content

```
↑∊¨m
```

```
ABCDEF
ABCDEF

ABCD
ABCDEFGHI
```

```
⊂∘∊⍤0⊢m
```

```
 ABCDEF  ABCDEF
 ABCD    ABCDEFGHI
```

Actually, rank can do more than just that, in a powerful way that `¨` cannot compare to. The derived function can be applied dyadically.

```
(⎕C 2 3 4⍴⎕A)(,X)⍤1⊢2 3 4⍴⎕A
```

```
( abcd  ,  ABCD )
( efgh  ,  EFGH )
( ijkl  ,  IJKL )

( mnop  ,  MNOP )
( qrst  ,  QRST )
( uvwx  ,  UVWX )
```

Here, we're concatenating the rank-1 sub-arrays of the arguments. Let's use *different* ranks for the left and right arguments!

```
(⎕C 2 2⍴⎕A)(,X)⍤1 2⊢2 2 2⍴⎕A
```

```
( ab  ,  AB )
(        CD )

( cd  ,  EF )
(        GH )
```

Skip to main content

```
(⎕C 2 2⍴⎕A),⍤1 2⊢2 2 2⍴⎕A
```

```
aAB
bCD

cEF
dGH
```

We can express the outer product in terms of rank.

```
(⎕C 2 2⍴⎕A)∘.(,X)3 2⍴⎕A
```

```
 (a , A) (a , B)
 (a , C) (a , D)
 (a , E) (a , F)

 (b , A) (b , B)
 (b , C) (b , D)
 (b , E) (b , F)


 (c , A) (c , B)
 (c , C) (c , D)
 (c , E) (c , F)

 (d , A) (d , B)
 (d , C) (d , D)
 (d , E) (d , F)
```

Note how each scalar in `⍺` got paired up with the *entire* `⍵`. In other words, we need the left rank to be 0 and the right rank to be infinite. But since Dyalog APL only allows arrays of up to rank 15, that is enough (15 = ∞ for very small values of ∞).

`⍤N` can also take a three-element `N`. That's only useful for ambivalent functions. It then means that if the derived function is applied monadically, it gets applied to sub-arrays of rank `N[1]` and if it is applied dyadically, it is applied to sub-arrays of rank `N[2]` of `⍺` and of `N[3]` of `⍵`.

```
(⊂X)⍤1 2 0⊢2 2⍴⎕A
```

```
(    ⊂    AB )
```

Skip to main content

That is, applies to rank-1 sub-arrays.

```
(⎕C 2 2⍴⎕A)(⊂X)⍤1 2 0⊢2 2⍴⎕A
```

```
( ab  ⊂ A)
( cd     )

( ab  ⊂ B)
( cd     )


( ab  ⊂ C)
( cd     )

( ab  ⊂ D)
( cd     )
```

That is, applies to rank-2s of `⍺` (which happens to be the entire array here) and rank-0s of `⍵`.

Finally, let's explore how `f∘g` works. Let's again use a slightly modified X:

```
X←{f←⍺⍺ ⋄ ⍺←'' ⋄ ∊'('⍺(⎕CR'f')⍵')'}
```

```
(,X)∘(⊂X)'⍵' ⋄ ⎕←'⍺'(,X)∘(⊂X)'⍵'
```

```
(,(⊂⍵))
(⍺,(⊂⍵))
```

Here is an example of how we can use this to analyse more complex trains, like this CamelCase splitter:

```
(⊢⊂⍨∊∘⎕A)'CamelCaseRocks'
```

```
 Camel  Case  Rocks
```

The `⍨` isn't necessary, but it is in there for illustration purposes.

Skip to main content

```
(⊢X⊂X⍨∊∘⎕A X)'ω'
```

```
((∊∘ABCDEFGHIJKLMNOPQRSTUVWXYZω)⊂(⊢ω))
```

So now we can see how `⍨` works and how `ω` is distributed to the outer functions. Here’s an even more complex train, which splits on any number of delimiters:

```
' ,;'(⊢⊆⍨∘~∊⍨)'some delimiters;in,use'
```

```
 some  delimiters  in  use
```

```
'α'((⊢X)(⊆X)⍨∘(~X)(∊X)⍨)'ω'
```

```
((~(ω∊α))⊆(α⊢ω))
```

Now we just have to note the obvious that `α⊢ω` is `ω`. This should also explain why `⊣` and `⊢` can get you the arguments when in a train.

Condition controlled loops

How do you write APL code for “do-while” type problems? Well, modern APL *does* actually have `:While-:EndWhile` and `:Repeat-:Until` constructs. But we have other options: like the `⍣` operator, and recursion, which isn’t bad in APL, as you can use the optimised tail-recursion.

Power `⍣`

About `⍣`, it is important to note that it always applies its left operand at least once. Let’s take a very simple (pun intended) example. Let’s say we have an array like `⊂⊂⊂⊂2 2⍴'ok'`. We want to disclose it until it is simple. If we do `⊃⍣≡` we’ll end up with ‘o’.

Another common pitfall is to use `ω` in the right operand (the one that answers “are we done?”)

Skip to main content

```
⊃⍣{1≥|≡⍺} ⊂⊂⊂⊂2 2⍴'ok'
```

```
ok
ok
```

The problem is that our input might have 0 levels of nesting; then we fail:

```
⊃⍣{1≥|≡⍺} 2 2⍴'ok'
```

```
o
```

This is because `⊃` is being applied once before we even ask if we're done. If instead we move the test inside the left operand we get:

```
{1≥|≡⍵:⍵ ⋄ ⊃⍵}⍣≡ 2 2⍴'ok'
```

```
ok
ok
```

The left operand will become a no-op when we're done. In fact, we can even use the power operator instead of the guard!

```
{⊃⍣(1<|≡⍵)⊢⍵}⍣≡⊂⊂⊂⊂2 2⍴'ok'
{⊃⍣(1<|≡⍵)⊢⍵}⍣≡ 2 2⍴'ok'
```

```
ok
ok
```

```
ok
ok
```

Of course, you don't have to write everything inline. You could use a separate function for the main processing. In your left operand, you can of course place your done-condition at the top or at the bottom, or anywhere else. But let's say instead that we don't want the condition to be based on the

Skip to main content

data processed. Rather, we want to periodically read an outside value to decide whether to continue or not.

You can try this in your local APL:

```
done←0 ⋄ {⎕←ω⊣⎕dl 5}⍣{done}&'work'
```

It will run in the background, printing “work” every 5 seconds. Of course, it didn’t need to be a single value in `{done}`. It could be an entire function that figures out if we’re done based on a bunch of stuff.

Recursion `∇`

Recursion can be done simply by calling the function name. Dfns can also call themselves using `∇`. The benefit of `∇` is that you can rename the function or leave it anonymous. We should also mention `∇∇`. If you are writing your operators, you might want the operator’s code to “use” itself. You do that with `∇∇`. Inside such a dop, you can also use `∇` as a shortcut for `αα∇∇` or `αα∇∇ωω` depending on operator valence.

Other than this, it is actually much the same as with `⍣`: Establish the stop condition with a guard (or a control structure in a tradfn), and do the work otherwise.

The important thing is that APL detects when the final result will be used unmodified as the result of the previous iteration. Let’s say we wanted the beginning number of the 7-long sequence: `{16=+/'2'=⍕ω:ω⋄⊃∇1+ω}⍳7`. Now APL has to keep track of where came from so we can apply that final `⊃`. Can we detect a tail call? Yes. You can try this:

```
{⎕←≢⎕SI ⋄ 16=+/'2'=⍕ω:ω ⋄ ∇ 1+ω}2000+⍳7
```

It starts searching at 2000 to prevent output flooding. `⎕SI` is the State Indicator, or stack. Every time around the loop, we count the frames on the stack and print that. It’ll print 1 every time, because the stack “forgets” about the previous call every time.

However, if you try it with the `⊃`, then:

Skip to main content

```
{⎕←≢⎕SI ⋄ 16=+/'2'=⍕⍵:⍵ ⋄ ⊃ ∇ 1+⍵}2000+⍳7
```

you should be able to observe the stack frames increasing.

Let’s try implementing `Fib n` (which returns the n first Fibonacci numbers) using `⍣` and recursion. We can factor out the fundamental Fibonacci operation, which sums the last two elements of a vector, and tacks on the result:

```
Δ←{⍵,+/¯2↑⍵}  ⍝ Fundamental Fibonacci function
```

Using this, we can write a neatly tail-recursive Fibonacci function – note that all processing is to the right of the `∇`:

```
{⍺≤≢⍵:⍺↑⍵ ⋄ ⍺∇Δ⍵}∘1⊢ 10 ⍝ Tail-recursive
```

```
1 1 2 3 5 8 13 21 34 55
```

And here’s a clever application of the power operator:

```
{⍵↑Δ⍣⍵⊢1 1} 10 ⍝ append 1 1, n times
```

```
1 1 2 3 5 8 13 21 34 55
```