Introduction

> A language that doesn't affect the way you think about programming is not worth knowing. *–Alan Perlis*

Who is this for?

I wrote this to be the book I would have wanted to read when I started to learn APL. An introduction to APL for an experienced practitioner from a different programming language or two. We all learn in different ways, and I prefer the fundamental concepts laid bare first, and then learn by example.

I came to APL after discovering a file of solutions to the Advent of Code 2015 challenge in K, an APL derivative. That's around 100 lines of actual code, and whilst I didn't understand any of it, I kept looking at it, trying to figure out which of the 50 problems (well, 49) this was a solution to. Each of my Python solutions typically ran to 50-100 lines+ for the bulk of the problems.

Turns out it was the whole lot. That blew my mind.

What is APL?

APL is an array language, and one of the oldest programming languages still in use today, next to FORTRAN, Lisp and COBOL. APL uses its own curious-looking symbols, like ⍋⍒⍕⌈∗○≡⍬, rather than reserved words written out in English like most other languages, like C or Python. As a language, APL sits at a very high level of abstraction, making it well suited to ultra-concise formulations of algorithms.

APL is a language that time is only now beginning to catch up with. Modern processors sport dedicated vector-oriented instructions and APL presents a high degree of mechanical sympathy ideally suited to SIMD instructions and by often being completely branchless in nature. APL, and its more punk rock little sister, K, really *fly*. APL can offer unprecedented programmer efficiency, as well as all-out execution speed.

But isn't APL dead? APL is alive and well.

Why should I learn APL?

You will have your own motivations for wanting to learn APL. If you're searching for the hottest thing on the market right now, to land a job at a FAANG or pad your resumé with the most marketable skills, there are many other programming languages that will offer a better return: Go, Java, Rust… APL doesn't make a show in the TIOBE index in the top 100 "most popular" programming languages. Apparently, COBOL, Logo and AWK are all more "popular" than APL, at the time of writing. That's not to say there aren't extremely well-paid jobs around for engineers with skills in the "Iverson-family" languages (APL, J, K, Q etc), especially in finance. There are.

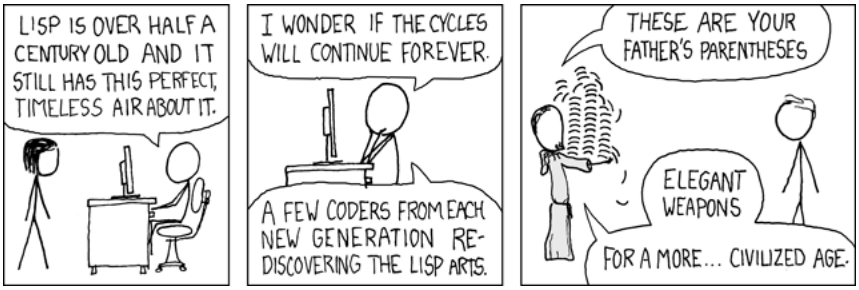
However, learning APL will reward you in other ways. Perhaps the old trope about "expanding your mind" is tired, but if your background is "C-like" – C, C++, Java, Python etc – you will be exposed to a new way of solving problems. In fact, learning new programming languages that come from a different paradigm is a force multiplier for your brain.

Talking about Lisp, but equally applicable to APL, Eric S. Raymond says in his essay How to become a hacker:

> Lisp is worth learning for a different reason — the profound enlightenment experience you will have when you finally get it. That experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.

On the topic of Lisp, one of my favorite xkcd strips talks about Lisp's parentheses as 'elegant weapons for a more civilized age', paraphrasing Star Wars. It could also have been said about APL.

Contents



APL was conceived as a *notation for thought*. It actually took years before it even had an actual implementation as a programming language. APL is a very natural way to express algorithms. Ken Iverson, the creator of APL, was awarded the ACM Turing Award in 1979 for his work on APL, and his accompanying paper, Notation as a Tool of Thought is somewhat of a computer science milestone. Dyalog, a leading commercial APL solutions provider, make a reference to this in their corporate tag-line: "The Tool of Thought for Software Solutions".

If you persist, you may discover an added bonus: code you write in *other* languages suddenly becomes more concise and efficient, too.

...but it's unreadable!

Every time APL gets a mention on Hacker News (it happens regularly), someone chirps up in the comments that "APL is unreadable", so let's deal with that upfront.

```
crt←{m|ω+.×α(⊣×⊢|∘⊃{0=ω:1 0 · (ω∇ω|α)+.×0 1,¯1,-⌊α÷ω})¨≍α÷≍m←×/α} ⍝ From APL Cart
```

There it is – perfectly readable: squiggle, squiggle, greek letter, weird symbol....

The key is that readability is in the eye of the beholder. To an APL programmer, the above implementation of the Chinese Remainder Theorem is perfectly readable. Claiming that something is unreadable because *you* can't read it is intellectually dishonest. I don't speak, or indeed read, Japanese, but that doesn't make Japanese in any way unreadable. As master haiku smith, Bashō (松尾 金作), exclaimed:

夏草や　兵どもが　夢の跡

The summer grasses. // All that remains // Of warriors' dreams.

What is a fairer observation is that APL *doesn't look anything like* the other programming languages you know, and yes, learning APL sure presents a different kind of challenge. If you're a seasoned (say) Python programmer, and you decide to pick up Ruby, you can reasonably expect to follow code written by others from day 1, and learn enough syntax within a day or two to write code yourself. Sure, it takes a bit longer to find your way around the standard library and learn how to write idiomatic Ruby, but still. If you've learned a few languages like that, picking up another represents known and quantifiable effort.

Let's be honest: there *will* be more initial friction when learning APL – for starters, your keyboard doesn't even have the squiggly symbols! Having said that, APL is a *tiny* language – there is negligible amounts of syntax to pick up, and believe it or not, actually quite easy to learn at a superficial level, once you get past the practical barriers.

What takes longer is to grasp the APL *way* – how to wield it idiomatically, if you like. Learning how to solve problems the data-parallel way, no loops, no branching, is the key to writing APL that rocks.

Once you get used to it, APL is *more* readable than more verbose languages [*citation needed*]. It strips away all the fluff, leaving only the algorithmic intent. John Earnest, creator of oK, wrote a humorous blog post on the topic, and Aaron Hsu's presentation on APL patterns and anti-patterns make some of the same points in a more serious vein.

The last example from John Earnest's blog post poses the hypothetical question – which is more readable, the JavaScript

```
let max = list[0];
for (let i=0; i<list.length; i++) {
    max = Math.max(list[i], max);
}
```

or the APL (although John used the corresponding K version)

```
⌈/
```

Don't I need a lot of mathematics?

No, not really, no.

Whilst APL can be traced back to Iverson's attempts at fixing some of the shortcomings of traditional mathematical notation, you don't have to be a math-wiz to understand, or benefit from, APL. If you're familiar with sums and products, or if you know another programming language, that's all you need. If you *do* have a background in maths, especially in linear algebra, you'll no doubt recognize some concepts along the way.

A note on our APL subset

Dyalog's APL dialect, which we'll chiefly work with here, has been around for a long time, and carries with it a lot of history in the shape of backwards compatibility. APL has also evolved a lot over the years. Dyalog's APL is really two rather different languages rolled into one: *traditional* style, which is procedural, and the newer *dfn* (pronouced *DEE-fun*) style, which is functional(ish, but let's not quibble). For the purposes of this exercise, we're going to pretend that the traditional style doesn't exist.

When we claim that there are no if-statements or loops in APL, how come one can write

```
∇ FizzBuzz end;i
:For i :In ⍳end
    :If 0=15|i
        ⎕←'FizzBuzz'
    :ElseIf 0=3|i
        ⎕←'Fizz'
    :ElseIf 0=5|i
        ⎕←'Buzz'
    :Else
        ⎕←i
    :EndIf
:EndFor
∇
```

which certainly looks both iffy and loopy – and dare I say even moderately readable, almost Pascal-chic? This is an example of an APL trad-function, which we from now onwards will pretend doesn't exist. This isn't a value judgment so much as a practical one in the context of this book. We will not cover the traditional style, nor Dyalog's object orientation extensions, and for simplicity's sake, when we from now on talk about APL, take that to mean "our subset of APL".

As an historical aside, the "newer" dfn style first appeared in Dyalog 8.1, from early 1997, according to John Scholes. So not *that* new.

Not covered

Specifically, we're not going to cover tradfns, OO, inverses, the trigonometric functions, complex numbers, variant, I-beams, spawn, threads, isolates, format, most of the ⎕-fns, .NET integration and probably many more.

Is terser better?

APL's *raison d'être* is to cut out the noise, to become an extension of thought. A lofty aim, for sure. APL code can be exceptionally compact; no other general-purpose programming language (apart from its siblings) gets close. No bit of code we write here will likely be longer than a few lines, and anecdotally, from personal experience, the difference runs to perhaps an order of magnitude compared with Python.

Is terser better? Opinions differ here. APLers, only half-jokingly, say they never scroll. Arthur Whitney, the creator of K, reputedly *hates* scrolling, even when writing in C:

> The K binary weighs in at about 50Kb. Someone asked about the interpreter source code. A frown flickered across the face of our visitor from Microsoft: what could be interesting about that? "The source is currently 264 lines of C," said Arthur. I thought I heard a sotto voce "that's not possible." Arthur showed us how he had arranged his source code in five files so that he could edit any one of them without scrolling. "Hate scrolling," he mumbled.

(from Vector)

Being able to see your whole implementation on a single page of code cuts down on context switching, and at least anecdotally makes mistakes easier to spot.

Other resources

If you're interested in the history of APL, a good read is Kromberg and Hui's weighty paper APL since 1978. Morten and Roger do all their own stunts.

J Software maintains a rich collection of APL-related publications.

Vector is the journal of the British APL Association.

A comprehensive, but now a bit outdated, reference is Legrand's door stopper Mastering Dyalog APL, all 800+ pages of it. It covers a lot of ground that we're not touching upon here. Work is afoot to update MDAPL to cover a more recent version of Dyalog.

As you get a bit further along your path to APL mastery, the APL Cart indexed idiom collection is an *amazing* resource.

The chat room APL Orchard on Stack Exchange features some of the sharpest minds on the array programming circuit. It's a welcoming place for newbies, too, and every now and then hosts so called *cultivations* – impromptu interactive lessons. A list of old cultivations is here. Drop in and say 'hi' – chances are that you will be given a private APL intro lesson (in public).

The APL Wiki is a rich and growing source of knowledge.

Dyalog's own developer documentation is a reference - rich depth, but not the easiest place to learn new things from. Dyalog TV hosts Dyalog's back catalog of webinars and lots of conference presentations with a lot of varied nuggets. There is also the Dyalog forums although a bit low-volume.

Dyalog's docs for the dfns workspace is a treasure trove of APL exotica. It's what passes as a "standard library" for Dyalog, and some of it is useful for everyday coding (like segs and iotag), and other stuff perhaps less so – but you can learn a lot by looking at its implementations to see what "real APL" looks like.

Speaking of "real APL", Dyalog has a lot of public code in their github, which is a great resource.

Ok, I'm convinced, how do I get started?

There are a couple of options. To get started right now, you can head to TryAPL, which is a web version of Dyalog APL's latest and greatest release. That should work for most of the stuff we'll do here.

> **Note**
>
> Dyalog APL is a commercial product, not open source. Fortunately, they offer a generous non-commercial free to use clause in their licensing terms, so you can download and install Dyalog APL locally without cost, as long as you're not making money from it.

Dyalog consists of the APL interpreter itself, and an IDE. You'll also need to install a font with the APL glyphs (as the "squiggles" are actually called) and a keyboard layout, depending on your platform. I am using Dyalog version 18 on MacOS, which comes with an IDE called RIDE. On Windows you get a different IDE.

There is also GNU APL, and whilst there are similarities, its dialect is a bit different. GNU APL was made to be a *libre* software reimplementation of IBM's APL2, and Dyalog has moved on quite a bit from this. The examples we present here are unlikely to all function as-is in GNU APL. Other options are ngn/apl and dzaima/apl, both hobbyist implementations that roughly implement the subset we're interested in.

Dyalog also has an excellent Jupyter kernel that allows you to use APL in the Jupyter notebook interface – which indeed is how this book is written.

Assuming you've managed to get the APL keyboard layout installed, you'll be hunting and pecking for a while. It takes a few days to learn where the most useful glyphs reside. The Dyalog IDE has the *language bar* at the top which you can also use – and hovering over a glyph will indicate where it resides on the keyboard. Here's what my RIDE shows in terms of keyboard layout:

Are you ready? Let's begin.

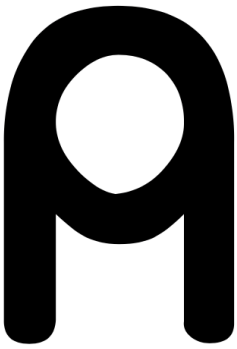
Our first tentative steps

The first thing we'll do is to specify our *index origin*. In most programming languages in use today, arrays are indexed starting from 0, with a few notable exceptions (for example Julia and Lua). In APL, the index origin is a configurable option. Leaving aside for a moment as to if this is a good idea or not, it's one of those things that sooner or later will catch you out. We'll try to always be explicit. If, when trying to run code someone else wrote, you're faced with an `INDEX ERROR`, it's worth changing the index origin to see if that makes a difference.

Dyalog's default index origin is 1, but we'll set it to 0 for the least possible surprise. We won't use anything where this actually matters for a while, but let's get used to it from the beginning.

```
⎕IO ← 0 ⍝ Index origin gets 0
```

We pronounce this as "quad-eye-oh gets zero". The left-arrow, `←`, called *Gets*, is the symbol for assignment. The green text following the `⍝` symbol is a comment. APL's comment symbol is called *Lamp* – and a bit of an *aide-memoire* – comments should illuminate the code. On my keyboard, it lives on the comma-key, and is a useful first APL key sequence to memorise. Does it look like a lamp?



Maybe an LED? Or a kerosene lamp, depending on how long you've been doing APL for. Anyway - a comment.

```
⍝ This line does nothing!
```

The usual arithmetic operations work as expected, don't they?

```
2*5 ⍝ Wait... wot?
```

```
32
```

Ok, that may have come as a surprise. It turns out that `*` is the symbol for exponentiation, not multiplication. If we want to multiply, we need to use `×` (which lives on the `–` key):

```
2×5
```

```
10
```

Ok, what about division?

```
6/2 ⍝....?
```

```
2 2 2 2 2 2
```

Nope. The slash does something decidedly not-division. We'll get back to that one later. Division is `÷`, not `/`:

```
20÷4 ⍝ Division is ÷, not /
```

```
5
```

Negative numbers are not written with the traditional minus-sign, but with a special glyph, the *High minus*: `¯`

```
¯87 ⍝ Note: not -87
```

```
¯87
```

Here's another surprise:

```
2×5+7
```

```
24
```

In any other language, or indeed on a calculator, that would have given 17. So what's going on here? We're taught that multiplication goes before addition since year 1 in school. We've stumbled on one of the things in mathematical notation that motivated Iverson to create APL: in mathematics, operator precedence is kind of awkward and irregular.

In APL, everything evaluates strictly right to left. If you want to bend this evaluation order, you'll need parentheses.

```
(2×5)+7
```

```
17
```

We can assign a value to a variable using *Gets* (`←`) that we met earlier:

```
a ← (2×5)+7 ⍝ a gets 17.
```

```
a ⍝ Just evaluating the variable returns its value
```

```
17
```

```
⊢a ← 42 ⍝ a gets 42, and please evaluate it so I can see the value
```

```
42
```

The glyph `⊢` is called *Right tack*, and is a function referred to as *Same*. It simply returns its argument. Used like this it allows us to show the result of an expression even if the expression itself wouldn't normally return a result. *Right tack* might seem like a pretty pointless function, but we'll put it to good use later in different contexts. To "print" the value of a variable we can also use *Quad gets*, `⎕ ←`:

```
⎕ ← a
```

```
42
```

As a mnemonic, you can think of the *Quad* (⎕) glyph representing your computer screen (or a piece of printer paper) in this context. And for the avoidance of any doubt, the *Quad* glyph really *is* a little rectangle, and not the symbol used to show a character set mismatch on badly formed web pages.

APL variable names follow mostly similar rules to other programming languages – any combination of letters (upper, and lower case), digits (apart from the first character), and the underscore symbol (_):

```
⎕ ← fiftySeven ← 57
⎕ ← five38 ← 538
⎕ ← My_Variable_Name ← 'hello world'
```

```
57
538
hello world
```

One convention you'll occasionally see is the _ variable, which is sometimes used to denote a value we don't care about. There is nothing special about a variable called _ – it's just a convention.

```
(a b _ d) ← 3 1 4 1 ⍝ Don't care about the third value
a b d
```

```
3 1 1
```

There are two additional characters you can use in names, Δ and ∆, if you feel the need to add extra spice in the lives of others reading your code. An advantage of APL's use of glyphs instead of reserved words is that it's impossible to create names that clash with internal or reserved words in APL.

You can of course use whatever naming convention that pleases you, but Adám from Dyalog has written up an informal style guide that's useful reading. As APL lends itself to a terse style, long, rambling, Java-esque variable names are frowned upon. Here are some opinionated home truths from the Carlisle Group, too, to consider on your APL journey.

Valence

Valence (sometimes called *arity*, too) refers to the way that functions expect arguments. In APL, we talk about *monadic* and *dyadic* functions and operators (and occasionally *niladic*, too).

There is a central tenet when writing about programming that for every mention of the word monad you lose half your readership. We can blame Team Haskell for that. Fear not – we're staying right out of category theory. Whilst APL has monadic functions and operators, the term predates Haskell by some margin, and really refers to something different altogether. Hey, Haskell, make up your own words, will you? Just kidding, we're huge Haskell fans.

APL functions can take arguments on either side, just like you're perfectly used to with arithmetic operators in many other languages (not Lisps):

```
1 + 1 ⍝ Infix function application, dyadic +
```

```
2
```

In a Lispy language, function application is always monadic, even for arithmetic, so you'd write

```
(+ 1 1)
```

When we say that a function is `dyadic` in APL, all we mean is that the function takes both a left, and a right argument, like the `+` example above. A `monadic` function, conversely, is a function that only takes a right argument, for example, *Factorial*, `!`:

```
!7 ⍝ Monadic ! is factorial: 1×2×3×4×5×6×7
```

```
5040
```

Some APL functions can be either monadic or dyadic, and in many cases these have different meanings – an endless source of ~~confusion~~ joy for the beginner. For example:

```
7×3 ⍝ Dyadic × is multiplication
×¯7 ⍝ Monadic × is 'direction', or signum (the sign, in our case). Thanks.
```

```
21
¯1
```

> **Tip**
>
> In the Dyalog IDE, if you hover over a glyph in the language bar, it will show brief examples of both its monadic and dyadic uses, where applicable. Some claim you can learn the whole language by hovering over this bar.

It's arrays all the way down

> There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult. –*Tony Hoare*

Dyalog APL is an *array language*. Chances are that you have not come across this programming language paradigm before. In APL, the array is the only data structure that's available to us. In terms of terminology, this can get a bit confusing. Both in mathematics and other programming languages there are many different words used to refer to what might ultimately end up implemented as an array in APL – vector, matrix, list, string etc. We'll use some of these interchangeably where the problem domain makes this convenient: the APL character array `'hello world'` is most likely intended to be a string.

But first things first: we promised to be explicit about our index origin, so let's set that to zero upfront:

```
⎕IO ← 0
```

A large part of programming in APL becomes a matter of data wrangling. Instead of looping and branching, you use simple primitives to transform your data, perhaps from a big vector into a multi-dimensional array which you then reduce back down along a different axis. As a simple example, say we have a nested vector, and we want to know the sum of all the first elements.

```
⍝ ← data ← (1 2 3 4) (2 5 8 6) (8 6 2 3) (8 7 6 1)
```

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

In naïve Python, we'd say something like

```
def sumfirst(d):
    total = 0
    for item in d:
        total += item[0]
    return total
```

The APL version turns the vector into a 2D array, *sum-reduces* (we'll explain properly what this means later) along the leading axis and picks the first value:

```
⊃+⌿↑data
```

```
19
```

Wow. Ok, let's take that apart. Starting from the right, first step is to "trade depth for rank" – turning the nested vector into a 2D array:

```
↑data
```

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

Now we sum each column, *reducing* the dimensionality by 1, turning our matrix into a vector:

```
+⌿↑data
```

```
19 20 19 14
```

Finally, we pick the first value:

```
⊃+⌿↑data
```

Let's get in shape

The first concept we need to understand is that data is defined in terms of its *shape*. The shape defines the dimensionality of our data, and we can investigate that using the APL function `⍴` – the Greek letter "rho":

```
⍴5
```

Oh, ok – nothing? That was rather uninteresting, right? Let's ask Dyalog to help us a bit by displaying output with borders indicating shape, type and structure, and try that again:

```
⍝ Some visualisation help, please.
]box on -style=max
```

```
⍴5
```

```
┌⊖┐
│0│
└~┘
```

Well, at least that's something to go on. That particular output is worth committing to memory. It's the empty vector, which actually has its own symbol called *Zilde*, from zero-tilde, a `⍬` overstruck by `~` – `⍬`.

> **Note**
>
> Expressions that start with a close square bracket, like `]box` are called *user commands*. We'll encounter a few of those throughout this book. Typically, user commands will give you a help text if you pass them a `-??`. Give that a go with `]box -??` – there is a lot of useful information there.

We can glean a lot of useful information from the boxed output. Here, the `⊖` symbol at the top of the box means that the vector's axis is of length 0. The `~` at the bottom means that the vector is *non-nested*, and the 0 in the middle is the empty vector's *prototype element* – we can for now think of this as the vector's *type*. So we have an *empty numerical vector*.

Running with `]box -style=max` gets very noisy after a while, so let's dial that back a bit:

```
]box on -style=min
```

```
Was ON -style=max
```

We can always get verbose boxing by using the command `]DISPLAY` before any expression.

We can check that the shape of 5 is indeed *Zilde* by saying:

```
⍬≡⍴5 ⍝ Does zilde match shape of 5?
```

```
1
```

So it would seem. What does *that* mean, then? Well, it means that the scalar `5` has "no shape" – it has no dimensional extent. It is a point, which makes a degree of sense.

Let's instead consider a vector – in other words, something that *has* dimensional extent. Here's a vector:

```
]DISPLAY 8 5 2 3 ⍝ A vector, yay
```

```
┌→────┐
│8 5 2 3│
└~────┘
```

We made a vector by simply listing a bunch of scalars with spaces between them. Our verbose visualisation shows the vector as a box with an arrow on top. "Arrow on top" is a notation borrowed from mathematics to denote vectors.

So, what shape does a vector have?

```
]DISPLAY ⍴ 8 5 2 3 ⍝ What's the shape of my vector?
```

```
┌→┐
│4│
└~┘
```

The shape is a *vector* with a single element, the integer 4. We suspected already that shape is a vector. A scalar's shape is an empty vector, and a vector's shape is a vector containing a single integer representing the number of elements.

Ok, let's try a matrix – or a 2D array, disregarding for the moment the APL mechanics of its creation:

```
m ← ↑(1 2 3 4)(5 6 7 8)(9 10 11 12)
]DISPLAY m
```

```
┌→──────┐
↓1  2  3  4 │
│5  6  7  8 │
│9 10 11 12 │
└~──────┘
```

That gives us a 3×4 matrix called `m`, holding the integers 1-12. APL helps us by now showing two arrows on the surrounding box. What is its shape?

```
⍴m
```

```
3 4
```

The shape vector now has *two* components – as our matrix has two axes – the first axis, y, of length 3, and the second, x, of length 4. In other words, the *shape of the shape* tells us something about our array: its dimensionality, or its number of axes:

```
]DISPLAY ⍴⍴m
```

```
┌→┐
│2│
└~┘
```

The *shape-of-shape* thing has a name: *rank*, although it's usually thought of as the *length* of the shape vector, as it's more convenient to have it as a scalar, rather than as a 1-element vector:

```
≠⍴m ⍝ Rank
```

```
2
```

Rank and reshape ⍴

Rank is a central concept in APL. APL's functions are said to be rank-polymorphic, meaning that operations extend seamlessly to any-rank arguments where this is possible.

For example:

```
1 + 1          ⍝ Scalar + scalar
1 + 1 2 3      ⍝ Scalar + vector
1 2 3 + 9 3 2  ⍝ Vector + vector
```

```
2
2 3 4
10 5 5
```

This is an extraordinarily powerful concept, and one of the reasons why APL leads to such compact code. We'll return to this time and again.

When used dyadically, the *Shape* (⍴) function *reshapes* data. A typical use for this is to take data from some raw format and use ⍴ to bend it into the shape we want. Here's an example:

```
]DISPLAY 2 4⍴⍳8 ⍝ Reshape vector to 2×4 matrix
```

```
┌→──────┐
↓0 1 2 3 │
│4 5 6 7 │
└~──────┘
```

A couple of things to note here. First, the *iota* primitive `⍳`, which is a function called *Index generator*. It's what might be called range in other languages. It makes a sequence of consecutive integers, starting at the current *index origin*, `⎕IO`, which we set to zero in the beginning of this chapter, as you may recall.

In the example above, it creates a vector of 8 consecutive integers, starting with zero: `0 1 2 3 4 5 6 7`. We then *reshape* this into a 2×4 array.

Here's another example:

```
]DISPLAY v ← 8 6 2 9      ⍝ Vector of length 4
]DISPLAY m ← 1 4⍴v        ⍝ Reshape vector to 1×4 matrix
```

```
┌→──────┐
│8 6 2 9│
└~──────┘
┌→──────┐
↓8 6 2 9│
└~──────┘
```

In case the difference isn't obvious, let's look at the ranks:

```
≠⍴v
≠⍴m

v≡m ⍝ Does v match m?
```

```
1
2
0
```

This is an issue that bites everyone at some stage of their APL journey. We can see from the rank that `v` is a *vector* – rank 1, but `m` is a 1×4 matrix – rank 2, even if they look the same. The visualisation shows this with two arrows on the box for `m`, but only one arrow on the box for `v`. If we switch off the verbose visualisation, it gets *much* trickier to spot:

```
v
m

v≡m ⍝ ?????
```

```
8 6 2 9
8 6 2 9
0
```

We can conclude that higher arrays is a first-order concept in APL. A 2D array in APL is *not* a list-of-lists, as it would be in, for example, Python.

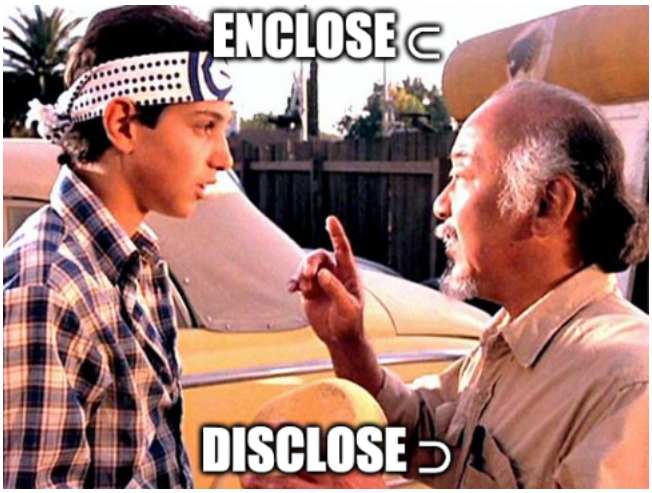
The other thing that is worth re-highlighting is how the array axes are enumerated in APL. We saw from a previous example that the shape of a 2D array lists y before x in the shape vector:

```
      m ← 2 4⍴0 1 2 3 4 5 6 7 ⍝ y=2, x=4
⍴m
```

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

This idea extends to higher dimensions, too. A 3D matrix would have its shape vector represent z, y and x axes in that order. A 57D matrix would.. you get the point.

Enclose, disclose ⊂⊃



Arrays in Dyalog APL are always collections of scalars, regardless of rank. However, we can create arbitrarily complex scalars by a process known as *enclosing*. This means putting something in a "box". It looks like so:

```
      v ← 1 2 3
⊂v          ⍝ Enclose the vector 1 2 3
v≡⊂v        ⍝ Does the vector v match the enclosed v? Of course not!
```

```
1 2 3
```

```
┌─────┐
│1 2 3│
└─────┘
```

```
0
```

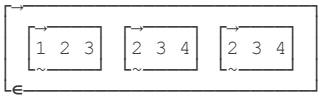
We can check that enclosing creates a scalar by inspecting the rank:

```
≠⍴v  ⍝ Rank of vector should be 1
≠⍴⊂v ⍝ Rank of scalar should be 0
```

```
1
0
```

This actually allows us to create a Python-style “list of lists” – or in APL terms, a vector of enclosed vectors:

```
]DISPLAY v ← (1 2 3) (2 3 4) (2 3 4)
```



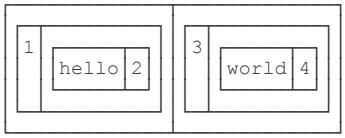
See the little epsilon ∈ in the lower edge of the box? It indicates that the elements of the vector are enclosed. We’ll discuss indexing in depth later on, but for now, let’s get a cell out of that vector:

```
v[1] ⍝ Get position 1 -- we get back an enclosed vector
```

```
┌─────┐
│2 3 4│
└─────┘
```

We can create arbitrary levels of nesting:

```
(1 ('hello' 2)) (3 ('world' 4))
```



> **Tip**
>
> Non-simple, a.k.a nested, arrays can be detrimental to APL performance!
>
> It’s best to bear this in mind from the beginning: if you can, choose multiple simple vectors, rather than a single vector of complex scalars.

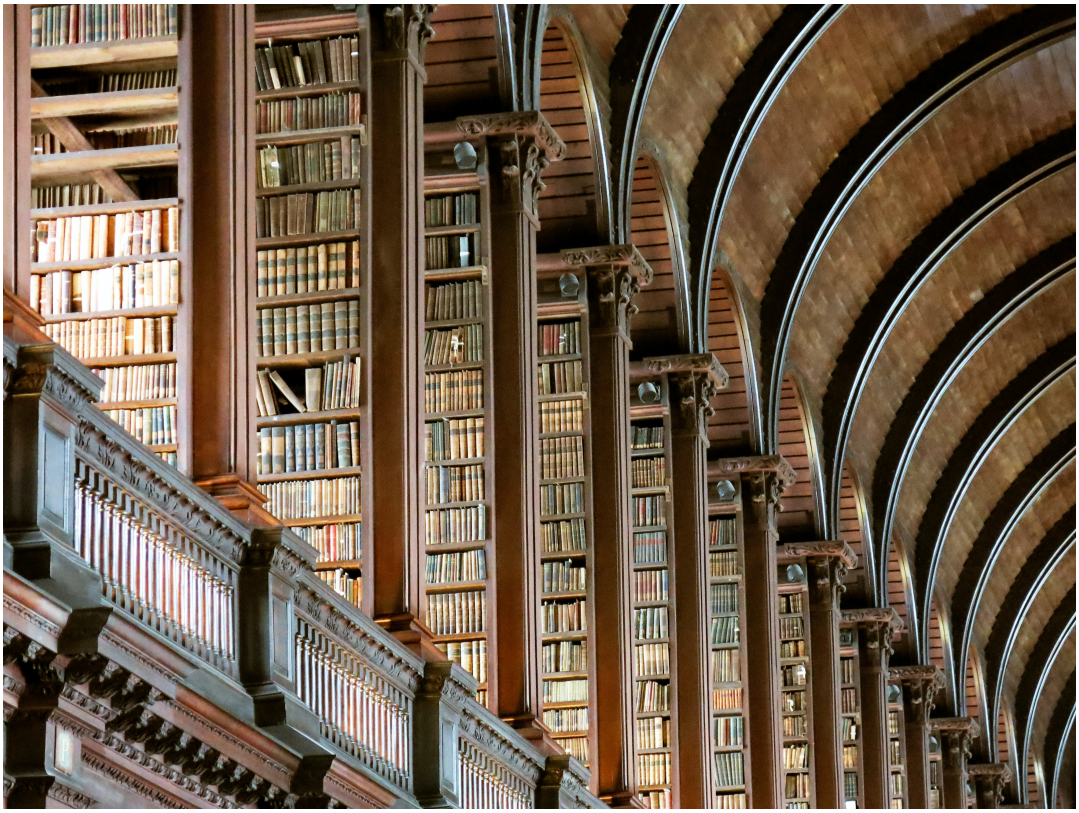


Photo by Jonathan Singer on Unsplash

Indexing

> Elegance is not a dispensable luxury but a factor that decides between success and failure. –*Edsger Dijkstra*

There are several ways of indexing into arrays and vectors. Some might even say "too many". A good start point is to read up on indexing in Dyalog's documentation, and Richard Park's webinar on the topic is extremely helpful, too.

But as before, we begin by setting our *index origin*, extra important as we're about to discuss indexing. Whilst we're at it, let's also ensure we have output boxing turned on.

```
⎕IO ← 0
```

```
]box on
```

Anyway – back to indexing. Crucially, elements of vectors and matrices are always scalars, but a scalar can be an enclosed vector or matrix. Indexing with `[]` or `⌷` returns the *box*, not the element *in* the box. However, if the element is a simple scalar, it's the same thing.

Let's look at *bracket indexing* first.

Bracket indexing `[ ]`

Bracket indexing is similar to how C-like languages index into arrays:

```
⍝ ← v ← 9 2 6 3 5 8 7 4 0 1
v[5]    ⍝ Grab the cell at index 5
```

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

However, unlike C (and its ilk), the indexing expression can be a vector, or even a higher-rank array:

```
v[5 2]  ⍝ Grab the cells at indices 5 and 2
```

```
8 6
```

We can mutate the vector or array via a bracket index, too:

```
v[3] ← ¯1
v
```

```
9 2 6 ¯1 5 8 7 4 0 1
```

Of course, this being APL, this idea extends to any shape of array, either by separating the axes by semi-colon:

```
]DISPLAY m ← 3 3⍴4 1 6 5 2 9 7 8 3 ⍝ a 3×3 matrix
m[1;1]  ⍝ Row 1, col 1
m[1;]   ⍝ Row 1
m[;1]   ⍝ Col 1
```

```
┌→──────┐
↓4 1 6  │
 5 2 9  │
 7 8 3  │
└~──────┘
2
5 2 9
1 2 8
```

or by supplying one or more *enclosed vectors*, each with shape equal to the *rank of the array*:

```
m[⊂1 1] ⍝ Centre
```

```
2
```

```
m[(0 0)(1 1)(2 2)] ⍝ Three points along the main diagonal
```

```
4 2 3
```

As indicated above, bracket indexing references *cells*, not the *values* enclosed in the cells. For numeric or character scalars, there is no difference, but the difference is clear for a nested array:

```
□ ← m ← 3 3⍴(1 2 3)(3 2 1)(4 5 6)(5 3 1)(5 6 8)(7 1 2)(4 3 9)(3 7 6)(4 5 1)
```

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

```
]DISPLAY m[1;1] ⍝ Note the returned enclosure.
```

```
┌→──┐
│5 6 8│
└~──┘
└∊──┘
```

Functional indexing ⌷

Whilst bracket-style indexing feels immediately familiar for those of us coming from a different programming language tradition, it is somewhat frowned upon amongst APLers. One reason for this is that it doesn't follow APL's normal strict right-to-left evaluation order as the indexing expression always must be evaluated first. As a consequence, it just stands out a bit: it's neither a monadic or dyadic function call. Another reason is that bracket indexing doesn't work in *tacit* functions, a topic we'll cover in a later chapter.

There is an alternative native indexing method: *functional*, or *Squad indexing*. *Squad*, "squashed quad", is the glyph ⌷. It can be seen mnemonically as the two square indexing brackets pushed together. *Squad* fixes some of the issues surrounding the bracket indexing method above (but introduces some new ones, too). As *Squad* is a normal dyadic function, it behaves just like any other of APL's dyadic functions:

```
□ ← m ← 3 3⍴(1 2 3)(3 2 1)(4 5 6)(5 3 1)(5 6 8)(7 1 2)(4 3 9)(3 7 6)(4 5 1)
```

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

```
1⌷m       ⍝ Row 1
```

5 3 1	5 6 8	7 1 2

```
1 1⌷m     ⍝ Cell 1 1
```

5 6 8

```
(⊂1 2)⌷m  ⍝ Rows 1 and 2
```

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

However, selecting cells from other axes than the first requires you to specify the axes explicitly with square brackets, which arguably looks a bit clumsy. Note that this isn't a bracket index, even though it looks like one. For example, here's how we select cell 2 from axis 1 (i.e. third column):

```
2⌷[1]m
```

4 5 6	7 1 2	4 5 1

or we could avoid the bracketed axis specification by picking the row from the matrix's *Transpose*, ⍉:

```
2⌷⍉m
```

4 5 6	7 1 2	4 5 1

Squad index does not let you mutate the array.

Another issue with *Squad* is that it flips the conventions established by the bracket indexing method. Let's return to a couple of our examples from the bracket indexing section, and compare those with how you'd achieve the same thing with *Squad*:

```
n ← 3 3⍴4 1 6 5 2 9 7 8 3
```

```
n[⊂1 1] ⍝ Centre
```

```
2
```

```
n[(0 0)(1 1)(2 2)] ⍝ Three points along the main diagonal
```

```
4 2 3
```

Squad's indexing expression, unlike that of bracket indexing's, specifies the coordinate for each axis in turn:

```
1 1⌷n ⍝ Centre
```

```
2
```

If we enclose the indexing expression we pick major cells, which arguably "feels" odd compared with how bracket indexing behaves:

```
(⊂1 1)⌷n ⍝ Repeat row 1
```

```
5 2 9
5 2 9
```

So, how do we choose the three diagonal cells with *Squad*? With great difficulty, as it turns out. For this we need *Sane indexing*, up next.

Sane indexing

Some APLers are unhappy with *Squad*'s semantics, and have proposed yet another mechanism, called *Sane indexing* or [*Select*](). It's not yet built into Dyalog, but it can be defined as:

```
I←⌷⍨∘⊃⍨⍤0 99 ⍝ Sane indexing
```

For the purposes of this explanation, it matters less how that incantation hangs together (we'll return to how this works in the section on the [*Rank*]() operator, `⍤`, later), but it does have a set of nice properties for the user.

Compare and contrast *Squad* and *Sane indexing*:

```
⎕ ← m ← 3 3⍴(1 2 3)(3 2 1)(4 5 6)(5 3 1)(5 6 8)(7 1 2)(4 3 9)(3 7 6)(4 5 1)
```

```
┌─────┬─────┬─────┐
│1 2 3│3 2 1│4 5 6│
├─────┼─────┼─────┤
│5 3 1│5 6 8│7 1 2│
├─────┼─────┼─────┤
│4 3 9│3 7 6│4 5 1│
└─────┴─────┴─────┘
```

Index with a vector:

```
1 2I m   ⍝ Sane:  select leading axis cells 1 and 2, or m[1 2;]
1 2⌷ m   ⍝ Squad: select m[⊂1 2]
```

```
┌─────┬─────┬─────┐
│5 3 1│5 6 8│7 1 2│
├─────┼─────┼─────┤
│4 3 9│3 7 6│4 5 1│
└─────┴─────┴─────┘
┌─────┐
│7 1 2│
└─────┘
```

Index with an *enclosed* vector:

```
(⊂1 2)I m  ⍝ Sane:  select m[⊂1 2]
(⊂1 2)⌷ m  ⍝ Squad: select m[1 2;]
```

```
┌─────┐
│7 1 2│
└─────┘
┌─────┬─────┬─────┐
│5 3 1│5 6 8│7 1 2│
├─────┼─────┼─────┤
│4 3 9│3 7 6│4 5 1│
└─────┴─────┴─────┘
```

So you can think of *Sane indexing* as *Squad*, but closer to the behaviour of the bracket indexing expression. We can finally select a bunch of cells by index:

```
(0 0)(1 2)(2 2)I m ⍝ Multiple cells by index, like m[(0 0)(1 2)(2 2)]
```

```
┌─────┬─────┬─────┐
│1 2 3│7 1 2│4 5 1│
└─────┴─────┴─────┘
```

Boolean indexing: compress

But wait! There's more to APL indexing. In fact, much of APL's expressive power comes from its central application of bit-Boolean arrays, and it's typically highly optimised. It's a concept you don't often see in non-array languages, but you may have been exposed to limited forms of it from bolt-on array libraries such as Python's [NumPy](). Similar functionality can be achieved using a [filter]() function taking a predicate in other languages.

The core idea is actually quite simple: select cells from an array by using a Boolean array as the indexing method, where a 1 means "yes, this one" and a 0 means "nope, not this one". We use [*Compress*]() to do this in APL, one of the several things represented by a forward slash `/`.

```
data   ← 0 1 2 3 4 5 6 7 8 9
select ← 0 0 1 0 1 1 0 1 1 0 ⍝ Select elements 2, 4, 5, 7 and 8
select/data
```

```
2 4 5 7 8
```

Compress is really a special case of the *Replicate* function, where the left argument is a Boolean vector. However, we can view the left argument more generally as a specification of how many times we should pick each element. In the compression case, that's either 1 or 0. In the more general case we need not constrain ourselves to binary – we can pick *any* number:

```
select ← 1 3 0 0 5 0 7 0 0 1
select/data
```

```
0 1 1 1 4 4 4 4 4 6 6 6 6 6 6 6 9
```

Replicate and *Compress* apply along the given axis in higher-rank arrays, either via *Replicate first* (⌿) or by specifiying the axis explicitly with the *bracket axis notation*, `/[axis]`:

```
m ← 3 3⍴9?9
]DISPLAY m
```

```
┌→──────┐
↓3 0 5  │
 4 1 8  │
 6 7 2  │
└~──────┘
```

```
select ← 0 1 0
```

```
select⌿m ⍝ Replicate first
```

```
4 1 8
```

```
select/m ⍝ Replicate
```

```
0
1
7
```

Pick ⊃

Yet another way to index into arrays is to use *Pick*. *Pick* eh… picks *elements*, not boxes, which often comes in handy. A monadic *Pick* picks the first element.

```
□ ← m ← 3 3⍴(1 2 3)(3 2 1)(4 5 6)(5 3 1)(5 6 8)(7 1 2)(4 3 9)(3 7 6)(4 5 1)
```

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

Element at 1;1 - note, no box:

```
(⊂1 1)⊃m
```

```
5 6 8
```

First element - note, no box:

```
⊃m
```

```
1 2 3
```

Reach indexing

Reach indexing is how you access elements of nested arrays. Note that nested arrays carry with them performance penalties and are best avoided if at all possible.

```
⍝ ← G ← 2 3⍴('Adam' 1)('Bob' 2)('Carl' 3)('Danni' 4)('Eve' 5)('Frank' 6)
G[⊂(0 1)0] ⍝ First element of the vector nested at ⊂0 1 of G
G[((0 0)0)((1 2)1)]
```

```
┌──────┬──────┬──────┐
│Adam 1│Bob 2 │Carl 3│
├──────┼──────┼──────┤
│Danni 4│Eve 5│Frank 6│
└──────┴──────┴──────┘
```

```
┌───┐
│Bob│
└───┘
```

```
┌─────┐
│Adam 6│
└─────┘
```

Assignable indexing expressions

As we saw above, bracket indexing is *assignable*, meaning that we can mutate the array. It is not the only assignable indexing expression in APL. The full list of *selective assignment functions* is available from the Dyalog documentation. It's worth studying this manual page, as it unlocks quite a few crafty ways of getting data into arrays.

For example, we can change the diagonal of a matrix by assigning directly to a dyadic *Transpose* by noting that `0 0⍉m` is the main diagonal of the matrix `m`:

```
]DISPLAY m ← 3 3⍴9?9
(0 0⍉m) ← ¯1 ¯1 ¯1 ⍝ 0 0⍉m is the main diagonal.
]DISPLAY m
```

```
┌→──────┐
↓1 2 3  │
│4 8 0  │
│6 5 7  │
└~──────┘
```

```
┌→──────┐
↓¯1  2  3│
│ 4 ¯1  0│
│ 6  5 ¯1│
└~──────┘
```

Indeed, we can even assign via Boolean indexing expressions, which might not be immediately obvious:

```
data   ← 0 1 2 3 4 5 6 7 8 9
select ← 0 0 1 0 1 1 0 1 1 0

(select/data) ← ¯1 ¯1 ¯1 ¯1 ¯1
data
```

```
0 1 ¯1 3 ¯1 ¯1 6 ¯1 ¯1 9
```

Perhaps even less obvious is assigning to *Take*:

```
⍝ ← s ← 'This is a string'
(2↑s) ← '**'
s
```

```
This is a string
**is is a string
```

...or even *Compress each*:

```
s←'This' 'is' ('a') 'string' 'without' 'is.'
((s='i')/¨s)←'*'
s
```

```
┌──────┬──┬─┬─────┬──────┬──┐
│Th*s  │*s│a│str*ng│w*thout│*s.│
└──────┴──┴─┴─────┴──────┴──┘
```

Glyphiary

> Are you quite sure that all those bells and whistles, all those wonderful facilities of your so called powerful programming languages, belong to the solution set rather than the problem set? –*Edsger Dijkstra*

Learning what each glyph does is an unavoidable chunk of time investment. However, there are some mnemonic cues sometimes based on where they sit on the keyboard, or that related functions sometimes have glyphs that are visually similar. Other times all bets are off: here's looking at you, `/`…

We're not going to cover them all. Learn them a few at a time as the need arises. Use the language bar in RIDE. But let's run through some of the immediately handy ones.

But first, the usual dance:

```
⎕IO ← 0             ⍝ Index origin is zero
]box on -style=max  ⍝ Show boxes at max verbosity
]rows on            ⍝ Don't wrap long output lines
```

Let's have a random matrix for our demonstration purposes. We've met *Shape* (`⍴`) already, but we'll get dyadic `?` – called *Deal* – for free. It gives us a random selection of numbers from a set, without replacement:

```
⍝ ← mat ← 3 4⍴12?12 ⍝ Ladies and gentlemen: our matrix
```

```
┌→──────┐
↓3  0  5 1│
 7  9  8 6│
 2 10 11 4│
 └~──────┘
```

Tally, Depth, Match: ≠≡

Tally, monadic `≠`, gives the number of major cells in an array, kind of like Python's len():

```
≠7 5 1 2 9
≠'Hello world'
≠mat
```

```
5

11

3
```

Pretty straight-forward. Monadic *Equal underbar*, `≡` is *Depth* – the max level of nesting:

```
≡1 2 3 4
≡(1 2)(3 4)
≡((1 2)(2 3))((4 5)(6 7))
≡(1 2)(3 4)3
```

```
1

2

3

¯2
```

The last case, giving `¯2`, means that the max depth is 2, but that not all cells are at the same depth. Depth is not rank. Say it with me: depth is not rank, depth is not rank, depth is not rank…

Turning to the dyadic forms, `≡` is *Match*, and with a pleasing visual symmetry, `≠` is *Not match*. We can think of *Match* being "deep equals for arrays": same rank, same order, same depth, every element the same:

```
1 2 3 4 ≡ 1 2 3 4 5
1 2 3 4 ≡ 1 2 3 4
1 2 3 4 ≡ 4 1 2 3
1 2 3 4 ≡ 1 4⍴1 2 3 4
```

```
0
1
0
0
```

Transpose, Reverse and Rotate: ⍉⊖⌽

Three glyphs used to change arrays around are *Transpose* (⍉), *Reverse* (⊖) and *Rotate* (⌽). They all look like a circle overstruck with a line. For example:

```
⍉mat ⍝ Transpose
⌽mat ⍝ Reverse
⊖mat ⍝ Reverse first
```

```
┌→──────┐
↓3 7  2 │
 0 9 10 │
 5 8 11 │
 1 6  4 │
└~──────┘
```

```
┌→──────┐
↓1  5  0 3 │
 6  8  9 7 │
 4 11 10 2 │
└~──────┘
```

```
┌→──────┐
↓2 10 11 4 │
 7  9  8 6 │
 3  0  5 1 │
└~──────┘
```

The two reverse glyphs mirror the issue we've seen with *Replicate* (/) vs *Replicate first* (⌿) – if you can, use the -first versions (the leading axis versions), and if you want to apply them along other axes, use either *Rank* ⍤ or the `[axis]` notation:

```
⊖⍤1⊢mat ⍝ Apply reverse-first to second axis using Rank
⊖[1]mat ⍝ Apply reverse-first to second axis bracket-axis
```

```
┌→──────┐
↓1  5  0 3 │
 6  8  9 7 │
 4 11 10 2 │
└~──────┘
```

```
┌→──────┐
↓1  5  0 3 │
 6  8  9 7 │
 4 11 10 2 │
└~──────┘
```

Both *Transpose* and *Reverse* can be applied dyadically, too, which presents us with a slight conundrum: the dyadic form of *Transpose* requires a deeper understanding of APL that we don't yet have – we'll push that one to its own chapter later on.

Dyadic ⊖ is actually *Rotate first*:

```
1 2 ¯1 0⊖mat
```

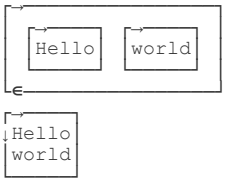
```
┌→──────┐
↓7 10 11 1 │
 2  0  5 6 │
 3  9  8 4 │
└~──────┘
```

Here, the left argument vector specifies the per-column magnitude and direction of the rotation.

Mix, Split, Take and Drop: ↓↑

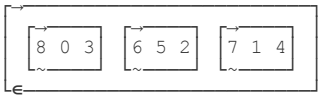
Mix ↑ raises the rank by 1. Easiest to visualise as a means of turning a nested vector into a matrix (but works for any rank):

```
      ← v ← 'Hello' 'world'
      ↑v
```



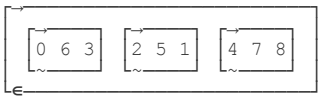
Split ↓, unsurprisingly, goes the other way; reducing rank:

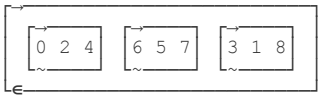
```
      ← m ← 3 3⍴9?9
      ↓m
```



Mix and *Split*, when combined with *Transpose*, make for a bit of a power-combo, ↓⍉↑, occasionally dubbed *Remix*, or *Zip*:

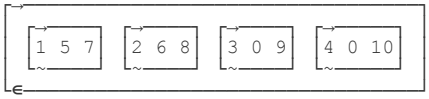
```
      ← v ← (0 6 3)(2 5 1)(4 7 8)
      ↓⍉↑v
```





Whilst it's tempting to think of *Remix* as the zip() found in, for example, Python, note that it most likely behaves differently to what you're used to:

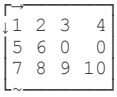
```
      ↓⍉↑(1 2 3 4)(5 6)(7 8 9 10)
```




```
Python 3.9.0 (default, Nov 15 2020, 06:25:35)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> l = [[1,2,3,4],[5,6],[7,8,9,10]]
>>> list(zip(*l))
[(1, 5, 7), (2, 6, 8)]
```

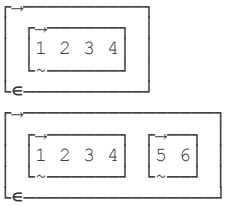
APL abhors ragged arrays and will inject the "prototype" element for whatever the element type is to ensure that all cells are the same size – Python has no concept of array as such, and so abandons play if an element can't be filled. Mixing a vector with cells of unequal numbers of elements in each cell will show us what happens:

```
      ↑(1 2 3 4)(5 6)(7 8 9 10)
```



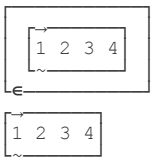
Dyadically, *Mix* and *Split* become *Take* and *Drop*. *Take* … takes cells:

```
1↑(1 2 3 4)(5 6)(7 8 9 10) ⍝ Take 1
2↑(1 2 3 4)(5 6)(7 8 9 10) ⍝ Take 2
```



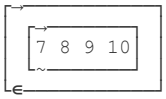
Note carefully the fact that *Take* returns *cells*, not elements, even if you take 1. Recalling the indexing chapter, *Take* 1 is equivalent to *Squad* 0, not *Pick* 0:

```
0⌷(1 2 3 4)(5 6)(7 8 9 10) ⍝ Squad 0 returns a cell
0⊃(1 2 3 4)(5 6)(7 8 9 10) ⍝ Pick 0 returns an element
```



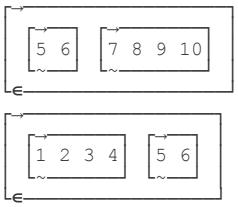
We can also use negative numbers to take from the rear:

```
¯1↑(1 2 3 4)(5 6)(7 8 9 10) ⍝ Take 1 from the back
```



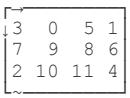
Drop does what we hopefully expect:

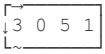
```
1↓(1 2 3 4)(5 6)(7 8 9 10)  ⍝ Drop 1 from the front
¯1↓(1 2 3 4)(5 6)(7 8 9 10) ⍝ Drop 1 from the back
```

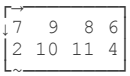


Take and *Drop* work on any rank array:

```
mat
1↑mat ⍝ Take first cell
1↓mat ⍝ Drop first cell
```







Index generator and Index of: ⍳

Iota, ⍳, called *Index generator* (or *Interval*) when used monadically and *Index of* when used dyadically is one to figure out early. Note that there is another glyph that looks similar, *iota underbar*, ⍸, that does something entirely different, so don't confuse the two!

The monadic case generates an integer interval, starting from the currently set ⎕IO (0 in our case, remember):

```
⍳10
```

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

Thinking of this as a monadic function taking a shape vector, this generalises to more complex shapes:

```
⍳3 4
```

```
┌→──────────────────────┐
↓ ┌→──┐ ┌→──┐ ┌→──┐ ┌→──┐ │
  │0 0│ │0 1│ │0 2│ │0 3│ │
  └~──┘ └~──┘ └~──┘ └~──┘ │
  ┌→──┐ ┌→──┐ ┌→──┐ ┌→──┐ │
  │1 0│ │1 1│ │1 2│ │1 3│ │
  └~──┘ └~──┘ └~──┘ └~──┘ │
  ┌→──┐ ┌→──┐ ┌→──┐ ┌→──┐ │
  │2 0│ │2 1│ │2 2│ │2 3│ │
  └~──┘ └~──┘ └~──┘ └~──┘ │
└∊──────────────────────┘
```

In other words, *iota* generates all possible *indices* into an array with the shape of its argument.

In the dyadic form, *iota* becomes *Index of*, another useful thing to know. *Index of* tells us the index of the first occurrence of an element:

```
'Hello world'⍳'o'
```

```
4
```

The right argument can have any shape, but the left argument is usually a vector.

```
'Hello world'⍳'od'
```

```
┌→──┐
│4 10│
└~──┘
```

A nifty feature is that if the right element isn't found, the returned index is `1+≢⍺` – one more than the length of the left argument. This can be used to provide a default match for items not found:

```
⎕ ← staff ← 'Adam' 'Bob' 'Charlotte'
lookup ← staff,⊂'Not found'
lookup[staff⍳'Bob' 'David']
```

```
┌→──────────────────┐
│ ┌→──┐ ┌→─┐ ┌→─────┐ │
│ │Adam│ │Bob│ │Charlotte│ │
│ └~──┘ └~─┘ └~─────┘ │
└∊──────────────────┘
┌→──────────┐
│ ┌→─┐ ┌→─────┐ │
│ │Bob│ │Not found│ │
│ └~─┘ └~─────┘ │
└∊──────────┘
```

Ravel, Catenate, Enlist, Member: `,⍪∈`

Ravel, monadic `,` and *Enlist*, monadic `∈`, do related things: *Ravel* creates a vector of the major cells of its argument, and *Enlist* creates a vector of the *elements* of its argument. For non-nested arrays, there is no difference:

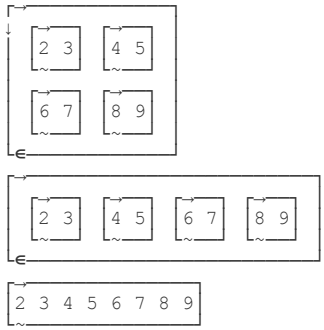
```
simple ← 3 4⍴3 0 5 1 7 9 8 6 2 10 11 4
,simple ⍝ Ravel
∈simple ⍝ Enlist
```

```
┌→──────────────┐
│3 0 5 1 7 9 8 6 2 10 11 4│
└~──────────────┘
```

```
┌→──────────────┐
│3 0 5 1 7 9 8 6 2 10 11 4│
└~──────────────┘
```

For a nested array, the difference is clearer:

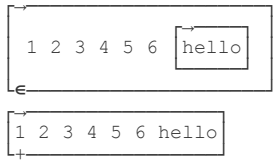
```
⍝ ← nested ← ↑((2 3)(4 5))((6 7)(8 9))
,nested ⍝ Ravel
∈nested ⍝ Enlist
```



In their dyadic guises, `,` becomes *Catenate*, and `∈` becomes *Membership*.

Catenate merges its left and right arguments:

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

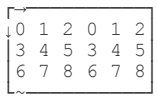


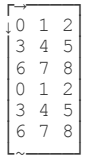
The distinction above is perhaps not obvious - and without `]box on` they would look identical. In the first case, *Catenate*'s right argument is a nested vector, whereas in the second case, it's a simple character vector.

Note that *Catenate* is trailling axis. There is a leading axis version, too, `⍪`, called *Laminate* – or, perhaps more logically – *Catenate first*.

We can catenate higher-rank arrays, too:

```
(3 3⍴⍳9),(3 3⍴⍳9) ⍝ Catenate-last (new cols)
(3 3⍴⍳9)⍪(3 3⍴⍳9) ⍝ Catenate-first/laminate (new rows)
```





Dyadic `∈` is *Membership*, another handy glyph in your arsenal:

```
'l'∈'Hello world'
```

```
1
```

It's not unlike Python's `in`:

```
>>> 'l' in 'Hello world'
True
```

at least at a superficial level. The APL version extends naturally to higher-rank arrays:

```
'lo w'∊'Hello world'
```

```
┌→──────┐
│1 1 1 1│
└~──────┘
```

whereas Python would see that as *is substring*:

```
>>> 'lo w' in 'Hello world'
True
```

You can, of course, get a similar substring behaviour in APL, too, but you need a different approach:

```
'lo'(⍳⍷)'Hello world' ⍝ Index of start of substring
```

```
┌→┐
│3│
└~┘
```

but we need a bit more flesh on our APL bones before we're ready for that – see the Finding things section later!

Selfie, Commute, Constant: ⍨

A firm favourite, the *Selfie* (although it's really called *Commute*), mirroring the confused look of the APL neophyte: ⍨. A monadic operator, the selfie commutes the left and right arguments of its operand function. At first, this seems beyond pointless – worse, in fact: it seems to offer nothing but added, deliberate obfuscation:

```
□ ← v ← 6 9 5 2 0 9
v↑⍨1     ⍝ Take 1 but commute arguments ⍨
```

```
┌→──────────┐
│6 9 5 2 0 9│
└~──────────┘
```

```
┌→┐
│6│
└~┘
```

As it turns out, it has its legitimate uses. Consider the consequences of APL's right to left evaluation order. If you have a dyadic function application with a complex expression to the *left*, you're forced to introduce parenthesis to ensure that the left side is fully evaluated before it is passed to the function. The commute operator, by shifting the complex expression to the *right* side, avoids this.

Compare the following two equivalent forms (disregarding for the moment what they mean):

```
{⍵⊂⍨1,2≠/⍵}
{(1,2≠/⍵)⊂⍵}
```

```
{⍵⊂⍨1,2≠/⍵}
{(1,2≠/⍵)⊂⍵}
```

So you could say "big deal, *one* glyph fewer to type", and you'd have a point. But the main advantage is that, by commuting, we preserve the right-to-left evaluation order. By having parenthesised part of the expression, we have an unnatural evaluation order. With a few well-placed selfies we can read the expression as Ken intended.

As with anything, there's a balance to be struck here. For the learner, selfies do make expressions harder, not easier, to read. The process of "flipping selfied expressions" occasionally helps when trying to deconstruct something someone else wrote.

If we give no left argument to the dyadic function derived by *Selfie* we echo the right argument to its left:

```
=⍨1 2 3 4 5
```

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

which is the same as saying:

```
1 2 3 4 5 = 1 2 3 4 5
```

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

If APL didn't already have a *Tally* function built in (≢) we could make one as the sum-reduction of self-equals to count the number of elements in a vector, say:

```
tally ← +/=⍨
```

```
tally 1 2 3 4 5
```

```
5
```

If *Selfie* is given an *array* operand, it becomes *Constant*: it always returns its operand:

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

```
1
```

```
1
```

```
1
```

```
┌→──────┐
│1 1 1 1│
└~──────┘
```

You can be excused for thinking that this is pretty pointless, but you'd be surprised how handy it can be. For example, you often find yourself wanting to create a vector of all the same value matching the shape of some other array:

```
□ ← m ← 3 3⍴9?9
5⍨¨m ⍝ Make an array that looks like m, but will all elements 5
```

```
┌→──────┐
↓0 6 3  │
│2 5 1  │
│4 7 8  │
└~──────┘
```

```
┌→──────┐
↓5 5 5  │
│5 5 5  │
│5 5 5  │
└~──────┘
```

There are many other ways we could achieve the same thing, for example

```
5⍴⍨⍴m
```

```
┌→──────┐
↓5 5 5  │
│5 5 5  │
│5 5 5  │
└~──────┘
```

but if you pronounce `5⍨¨m` as "constant 5 for each element in `m`" it matches the problem description nicely.

Unique, Union, Intersection, Without: ∪∩~

We have the full complement of set operations at our disposal. Starting with *Unique*, monadic `∪`, it does exactly what it says on the tin:

```
∪1 1 2 2 3 3 4 4 5 5 6 6
∪'hello world'
```

```
┌→
└~ 1 2 3 4 5 6
```

```
┌→
└~ helo wrd
```

In its dyadic version, ∪ becomes *Union*:

```
1 1 2 3 4 ∪ 1 2 5 6
```

```
┌→
└~ 1 1 2 3 4 5 6
```

Note that the arguments aren't proper sets. The above says "take ALL elements in the left argument, and add any element from the right which isn't already present".

Intersection, dyadic ∩, works similarly:

```
1 1 2 3 4 ∩ 1 2 5 6
```

```
┌→
└~ 1 1 2
```

For each element to the left, keep it if it's also in the right.

Dyadic ~ is *Without* – set difference:

```
1 1 2 3 4 5 ~ 1 3 5
```

```
┌→
└~ 2 4
```

The monadic ~ is Boolean *Not*:

```
~1 0 1 1 0 0 1
```

```
┌→
└~ 0 1 0 0 1 1 0
```

Grade up/down: ⍋⍒

To me, it's a Christmas tree and a carrot, but these twins are called *Grade up* (⍋) and *Grade down* (⍒). They are APL's very clever mechanisms for ordering arrays. To sort an array, we do:

```
□ ← data ← 110 109 204 40 105 201 2 208 160 143 213 31 21 317 132 242 164 176 67 18 75
89 18 7 20
data[⍋data]
```

```
┌→
└~ 110 109 204 40 105 201 2 208 160 143 213 31 21 317 132 242 164 176 67 18 75 89 18 7 20
```

```
┌→
└~ 2 7 18 18 20 21 31 40 67 75 89 105 109 110 132 143 160 164 176 201 204 208 213 242 317
```

So what does the *Grade-up* actually do? Let's have a look:

```
⍋data
```

```
┌→
└~ 6 23 19 22 24 12 11 3 18 20 21 4 1 0 14 9 8 16 17 5 2 7 10 15 13
```

Grading an array (up or down) produces a set of *indices*, not values. Consider the first element in the grade array. It says: the smallest element is to be found at index 6. The second-smallest is at index 23. The third smallest at index 19 etc.

At first blush, this seems like a roundabout way to sort something. First generate an indexing expression, then select elements according to this indexing expression. However, doing it this way – separating the determining of the order from the reordering of elements – has a number of advantages, chiefly that we can as easily apply the ordering to another array,

not just the one that we generated the ordering from.

In any sort of data processing or analysis, this crops up all the time: give the customer names, ordered by contract date. Sort the keys based on the values. That sort of thing. You can also answer questions such as *where* is the smallest value?

```
⍝ ← minidx ← ⊃⍋data ⍝ Index of smallest value: first element of Grade-up
data[minidx]
```

```
6
```

```
2
```

Direct functions and operators

> I made up the term "object-oriented," and I can tell you I did not have C++ in mind. –*Alan Kay*

Until now, we've mainly used APL as a toolkit for array manipulation using its built-in primitives. Sure, it goes a long way, but to fully take advantage of it, we also need to be able to create our own functions and operators. Fortunately, the syntax – if we can even call it that – for user-defined functions, is super-simple. As someone said on the APL Orchard chat room, all you need to do is "slap curly braces around your code, and you're done". It's not *quite* that simple, but not too far off.

The definition of a *dfn* (direct function) is enclosed in a pair of curly braces. It's what's known as an anonymous function, or a lambda, in other languages, meaning that there is no syntactic sugar for naming a function beyond ordinary assignment using *Gets*, `←`:

```
name ← {
   ⍝ expressions
}
```

First a bit of unavoidable yak shaving: if a dfn fits entirely on a single line, you can type it out directly in the RIDE IDE. So far, so expected. If your dfn extends across multiple lines, you can't (at the time of writing) just type it in. Dyalog is working on enhancing this.

Instead, the most convenient way to enter such a function is to say

```
)ed name
```

if you want to create a function called `name`. RIDE will open its function editor and let you enter your code there. Once you want to test your function, you first need to save it, which in APL-speak is called "to fix your function". The default way (and you're unlikely to ever discover this by yourself) to fix (save) your function is to hit `esc`. Yes, really. Fortunately – and I recommend you do so right now – RIDE allows you to re-bind keystrokes by clicking on the little keyboard symbol in the top-right corner.

Locate the row that says "Fix the current function" and map that to the key combo that is `save` on your platform. And whilst you're there, make a note of the keys binding to "Forward" and "Backward" – they default to `ctrl-shift-enter` and `ctrl-shift-backspace`. In the REPL, these mean forward/back in the history of executed lines, like the arrow keys in the shell. The arrow keys instead move up and down spatially, in the *output*, still, to me an incomprehensible design decision. You might also want to rebind "Strong interrupt" to `ctrl-c` while you're at it.

The other thing to note is that in RIDE there is no visual cue that an editor window contains unsaved changes. This is sure to bite you sooner or later.

In a Jupyter notebook, like here, entering functions might feel a bit more familiar: just type them into a cell. The only quirk is that in order to enter a multi-line dfn, you need to start the cell with `]dinput` (which isn't needed in a RIDE edit window). You *can* actually use the `]dinput` way of entering a multi-line function in the interpreter REPL, too, but (in my opinion) it's rather cumbersome unless you're just cutting and pasting from somewhere else. In the upcoming (at the time of writing) version 18.1 there is experimental support for being able to type in multi-line functions directly in the REPL, similar to how that works in Python's REPL.

First, our now familiar prelude:

```
⎕IO ← 0
]box on
]rows on
```

We're also going to make some assertions, so let's have a helper function for that, courtesy of APL legend Roger Hui (and not speculating on its inner workings for now):

```
assert ← {⍺ ← 'assertion failure' ⋄ 0∊⍵: ⍺ ⎕signal 8 ⋄ shy ← 0}
```

Left and right arguments: ⍺, ⍵

```
]dinput
MyFirstFunction ← {
    ⍝ Add left and right
    ⍺+⍵
}
```

```
32 MyFirstFunction 98
```

```
130
```

The first thing to note is the glyphs ⍺ and ⍵; the Greek letters alpha and omega. Alpha, the first letter of the Greek alphabet, is bound to the function's left argument, and omega, the last letter of the Greek alphabet, is bound to the function's right argument.

The second thing to learn is that APL expressions are separated by newline (or the diamond glyph, ⋄ which we'll get to later). We could rewrite the above function to use an intermediate variable:

```
]dinput
Sum ← {
    ⍝ Add left and right
    total ← ⍺+⍵
    total
}
```

```
32 Sum 98
```

```
130
```

The third thing is that there is no "return" statement. The function returns the first non-assigned value. It's worth letting this sink in: a function returns at the first point you do anything that isn't an assignment. In the function above, the return point is us just stating the variable `total`.

Default left argument

A useful feature is that you can set a default value for the left argument, ⍺. Compare:

```
{⍺ ← ¯99 ⋄ ⍺+⍵} 99
57 {⍺ ← ¯99 ⋄ ⍺+⍵} 99
```

```
0
156
```

This way a function can behave in different ways depending on if it is called monadically or dyadically. If you give a value to ⍺ with ←, then ⍺ will have this value if you didn't pass a left argument to the function. On the other hand, if you *do* pass a left argument, the "alpha gets" line has no effect. When defaulting the left argument this way, note that this only works for the *first* time you give a value to alpha, which is perhaps obvious when you think of it:

```
{⍺ ← ¯99 ⋄ ⍺ ← ¯999999 ⋄ ⍺+⍵} 99
```

```
0
```

Setting a default value for the left argument is a useful tool if you write recursive dfns, which typically accumulate their result on the left argument. This way you can call the function monadically to start, but subsequent iterations can use the left argument to build up the result. We'll talk more about recursion later in the chapter on iteration techniques, but here's a taster to demonstrate the default left argument technique. The glyph *Del* (∇) is a reference to the innermost function:

```
]dinput
sum ← {
    α ← 0       ⍝ Initialise the accumulator
    0=≢ω:α      ⍝ If right arg empty vector, return accumulator, see below!
    (α+⊃ω)∇1↓ω  ⍝ Add head to accumulator, recurse over tail
}
```

```
sum ⍳10
100 sum ⍳10
```

```
45
145
```

Alternation

What about alternation? We got a brief view of a conditional return in the example just above – a *guard statement*. Recall that there is no if-statement for us. A *guard*, defined with a colon, says, if the expression to the left of the colon is true, return from the function with the value to the right of the colon. A contrived example:

```
]dinput
Palinish ← {
    rev ← ⌽ω ⍝ Reverse the right arg
    rev≡ω: 1 ⍝ If right arg matches its own reverse, return 1
    0        ⍝ Else, return 0
}
```

```
Palinish 1 2 3 2 1
Palinish 1 2 3 4 5
Palinish 3
```

```
1
0
1
```

For the avoidance of doubt, we could of course have written that as

```
{ω≡⌽ω} 1 2 3 2 1 ⍝ Anonymous (unnamed) version
```

```
1
```

Note that execution flow does *not* carry on after a guard expression, even if that was an assignment. This requires some thought when writing code that needs to conditionally "do something" and then carry on, as illustrated in the following meaningless Python-like snippet:

```
def foo(arg):
    fum = 57
    fee = 8
    if flerp(arg) < 47:
        fee = 92
        fum = flumm(arg)
    return fee + fum
```

Actually, we can turn that into APL trivially, using only what we already know:

```
foo ← {47>flerp ω: 92+flumm ω ⋄ 57+8} ⍝ Note: diamond separator
```

Let's investigate some possible patterns for achieving a flow where execution carries on following an "if-then-else" branch. On way is to use an anonymous function, in essence like Python's

```
a = 42 if answer else -99
```

```
]dinput
foo ← {
    answer ← ω
    a ← {ω:42 ⋄ ¯99} answer
    ⍝ ...execution follows here
    ⍝ do something with a
}
```

In this case, as `answer` is a boolean, we could have avoided the inner function entirely by simply picking the values from a 2-element vector (note: need `⎕IO←0` for this to work):

```
]dinput
foo ← {⎕IO←0
    answer ← ⍵
    a ← answer⊃¯99 42
    ⍝ ...execution follows here
    ⍝ do something with a
}
```

With the first technique (the anonymous function) we have scope to extend the work done in each branch by making the function more complex, but ultimately it still returns a value. What if we need to modify multiple values? We could of course return an array of things, that's a perfectly valid approach.

Name scoping rules in Dyalog are a mixture between dynamic and lexical scope. Here's what Dyalog's docs say about it:

> When an inner (nested) dfn refers to a name, the interpreter searches for it by looking outwards through enclosing dfns, rather than searching back along the state indicator.

Lexical scope is almost certainly "what you expect". However, as Dyalog has no dedicated syntax for declaring a variable beyond giving it a value, if you need to modify a variable not introduced in the innermost dfn, we need a way to indicate this.

Consider the following:

```
]dinput
foo ← {
    a ← 45
    _ ← {a←¯99}⍬
    a
}
```

```
foo ⍬ ⍝ Note: 45, not ¯99
```

```
45
```

The innermost function creates a *new* variable with lexical scope, with the name *a*. So how *do* you modify state defined outside a function? This is where *modified assignment* comes in.

Modified Assignment

Modified assignment should feel familiar to any C or Python programmers amongst you:

```
# Modified assignment, Python-style
a = 45
a += 45 # a is now 90
```

In APL, we can also modify an existing variable's value via a function, for example:

```
]dinput
foo ← {
    a ← 45
    _ ← {a +← 45}⍬
    a
}
```

```
⎕ ← r ← foo ⍬
assert r=90
```

```
90
```

The supremely useful, but perhaps non-intuitive kicker: you can use modified assignment with *Right tack* (⊢) to *set* values:

```
]dinput
foo ← {
    a ← 45
    _ ← {a ⊢← ¯99}⍬
    a
}
```

```
⍝ ← r ← foo 0
assert r=¯99
```

```
¯99
```

We've already met a bunch of selectable assignment expressions when we talked about indexing. We can use all of those here. For example, we can mutate cells in a matrix using bracket indexing:

```
]dinput
foo ← {
    a ← 3 3⍴1 ⍝ 3×3 matrix of all 1
    _ ← {a[1;1] ← 0}⍬
    a
}
```

```
⍝ ← r ← foo 0
assert r≡3 3⍴1 1 1 1 0 1 1 1 1
```

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

Direct operators

We've met some of APL's built-in operators already, like *Reduce* and *Selfie*. Operators are perhaps the closest APL gets to the functional programming paradigm. An *operator* is a function that returns a derived function. An operator can take other functions as its *operands*.

Consider *Reduce*, `/`. It's a *monadic* operator that returns a derived function that can be called both monadically and dyadically:

```
2 (+/) ⍳10 ⍝ Parentheses not required, added for illustrative purposes
```

```
1 3 5 7 9 11 13 15 17
```

Here we see the monadic reduction operator, its sole operand being the plus function. The derived function it returns, plus-reduce, is called dyadically, with 2 to its left and `⍳10` to its right. In this case, a windowed reduction. We could have said

```
sumred ← +/
2 sumred ⍳10
```

```
1 3 5 7 9 11 13 15 17
```

to emphasize the fact that the operator really returns a function.

Dyalog lets us write our own operators, too, in a very similar way to how we write dfns. We call our own operators *direct operators*, henceforth just *dops*. Operators can be powerful, but the need for them actually rarely arises in practice. The reason for this is that the built-in operators already let you apply your own functions in various ways.

A few things to note with operators:

- In a *dop*, `⍺⍺` represents the left operand, and `⍵⍵` is the right operand.
- Unlike a monadic *dfn*, which takes a *right* argument, a monadic *dop* takes a *left* argument, like we just saw in the case of reduce.
- The operator can refer to itself for recursion using `∇∇`, analogously to the dfn's `∇`.

Here's an example monadic dop from the dfns workspace, a left-to-right version of reduce:

```
]dinput
foldl ← {                   ⍝ Fold (reduce) from the left.
    ⍺ ← ⊃0⍴⍵                ⍝ Default initial value for accumulator
    ↑⍺⍺⍨/(⌽⍵),⊂⍺
}
```

This implements `foldl` in terms of the standard `foldr` by reversing the list and appending an accumulator element.

```
+foldl ⍳10
+/⍳10
-foldl ⍳10
-/⍳10
```

```
 45
 45
¯45
¯5
```

This version takes an optional left argument that can be used to initialise the accumulator:

```
99 +foldl ⍳10
```

```
 144
```

Whilst there isn't anything overly complex about writing your own operators as such, we probably won't need to use many of them for the rest of this book, but at least now you have seen them.

Iteration

> As long as I'm alive, APL will never be used in Munich –*Fritz Bauer*
> Nor in Holland –*Edsger Dijkstra (as told by Alan Perlis)*

We started all this by claiming that there are no loops in APL. This is of course not entirely true: there are plenty of ways of achieving iteration, some of which are more efficient than others.

In order to get the best possible performance out of APL, it's worth seeking data-parallel algorithms, typically employing Boolean masks. However, it's not always possible, or sometimes performance matters less than code complexity, and a more iterative solution can be both clearer and fast enough.

We have at least four, maybe five different kinds of iteration mechanisms at our disposal: *Each* (¨), *Reduce* (⌿), *Del* (∇) and *Power* (⍣). The fifth is that of *scalar pervasion*, which can either be seen as a way of achieving iteration, or as a way of *avoiding* iteration, depending on your point of view. Wait, is it six? Maybe we should count *Rank* (⍤), too? *Rank* deserves its own separate section! Oh, and *Scan* (⍀), don't forget *Scan*!

Let's introduce them.

```
⎕IO ← 0
]box on
]rows on
assert ← {⍺ ← 'assertion failure' ⋄ 0∊⍵: ⍺ ⎕signal 8 ⋄ shy ← 0}
```

Each (a.k.a map): ¨

Most languages nowadays have a `map` construct. In fact, it's occasionally touted – erroneously – as sufficient evidence that a language is "functional" if it has a map function.

Perhaps you've seen Python's somewhat cumbersome version of map:

```
>>> list(map(lambda x: x*x, [1, 2, 3, 4, 5, 6, 7, 8, 9])) # Square elements
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

which in Python corresponds to something like

```
def mymap(func, iterable):
    for item in iterable:
        yield func(item)
```

Of course, seasoned Pythonistas would rightly frown at the above and instead recommend a *list comprehension*.

A map takes a function and applies it to every element in an array, creating a result array of the same length as its argument.

In APL, the glyph for map – referred to as *Each* – is two high dots: ¨.

```
×⍨¨1+⍳9  ⍝ Square elements via each (but see below!)
```

```
 1 4 9 16 25 36 49 64 81
```

This wasn't actually a great example, this is one of those cases where using a scalar function already does the job for us:

```
×⍨1+⍳9  ⍝ Square elements via scalar pervasion
```

```
1 4 9 16 25 36 49 64 81
```

Scalar pervasion means that functions in certain cases already know how to penetrate arrays.

Let's try another. Given a nested vector, what are the lengths of the elements?

```
□ ← V ← (1 2 3 4)(1 2)(3 4 5 6 7)(,2)(5 4 3 2 1)
≢¨V ⍝ Tally-each
```

```
┌─────────┬───┬─────────┬─┬─────────┐
│1 2 3 4  │1 2│3 4 5 6 7│2│5 4 3 2 1│
└─────────┴───┴─────────┴─┴─────────┘
4 2 5 1 5
```

If you're used to a different language, *Each* has a seductive quality, as it maps conceptually onto constructs you already know how to use. Beware though that in APL it's often inefficient, and that there are usually better alternatives.

Reduce (a.k.a foldr): ⌿, /

Most languages today sport some kind of variety of *fold* or *reduce*, regardless of the level of "functional" they claim to be.

In APL, *Reduce* is a central feature, which somewhat unhelpfully hijacks the glyph also used by *Compress/Replicate*, `/`. This is one, albeit not the main, reason why we prefer to use its close cousin, *Reduce first*, `⌿`, where we can. Reduction has a bit of an unfair reputation of being hard to understand. Guido reportedly hates `reduce()` in Python so much that it was demoted down to the dusty functools module:

> So now reduce(). This is actually the one I've always hated most, because, apart from a few examples involving + or *, almost every time I see a reduce() call with a non-trivial function argument, I need to grab pen and paper to diagram what's actually being fed into that function before I understand what the reduce() is supposed to do. So in my mind, the applicability of reduce() is pretty much limited to associative operators, and in all other cases it's better to write out the accumulation loop explicitly. –*Guido van Rossum*

Despite what Guido thinks, reduction is actually a pretty simple idea, and APL may even have been the first programming language with reduce in it (some Lispers disagree). Think of the operation of summing a bunch of numbers – this is an example of a reduction.

In APL, *Reduce* applies a function to elements in an array, producing a result which is rank-reduced by 1. In other words, reducing a vector (rank 1) produces a scalar (rank 0). In the example of `+`, summing the elements of a vector obviously produces a scalar: the total.

```
+⌿1 2 3 4 5 6 7 8 9 ⍝ sum-reduce-first integers 1-9
```

```
45
```

Simplifying a bit, we can think of *Reduce first* as an operator that injects its left operand function in the gaps between elements of the argument:

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

```
45
```

When using *Reduce* in APL, you need to take extra care to ensure that it works with its strict right to left evaluation order. A reduce is also called a *fold* in other languages (Lisp, Erlang etc), and APL's *Reduce* is a so-called *foldr* – it reduces right to left, which makes sense for APL, but occasionally less sense for the programmer.

Again, it can help writing it out in long-hand to see what's going on:

```
-⌿1 2 3 4 5 6 7 8 9 ⍝ difference-reduction -- take care: right to left fold!
```

```
5
```

If that was the result you expected, you're well on your way to mastery. Inject the operand between items:

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

```
5
```

Reduction is especially useful when working with higher-rank arrays. *Reduce first* is called so because it reduces along the *first* axis. So a sum-reduce-first of a rank 2 integer array will sum its columns to produce a *vector* (rank 1) of the columnar sums:

```
⍝ ← m ← 3 3⍴9?9
+⌿m
```

```
3 0 5
4 1 8
6 7 2
13 8 15
```

If we wanted to sum-reduce along the rows, we can either use `/` (which for historical reasons does just that):

```
+/m
```

```
8 13 15
```

or we can explicitly tell `⌿` to apply along a different axis, using bracket axis:

```
+⌿[1]m
```

```
8 13 15
```

For consistency, it's best to prefer operators and functions that default to applying to the leading axis where possible. The fact that APL, unlike J, has a mixture is an unhelpful side-effect of backwards compatibility.

Windowed reduction

Reduce has a few more handy tricks up its sleeve. Instead of reducing the whole argument array, we can employ a sliding window. This lets us compute a set of reductions over shorter stretches of the data. The derived function returned by the reduction operators can be called dyadically, specifying as the left argument the size of the sliding window.

For example, to calculate the sum of each element in a vector with its subsequent element, we employ a reduction with a sliding window of size 2:

```
2+⌿1 2 3 4 5 6 7 8 9
```

```
3 5 7 9 11 13 15 17
```

Scan: `\, ⍀`

Scan/Scan first blurrs the distinction between *Each* and *Reduce*. In right hands, it can be a true APL super power, but beware that scans tend to be slow: most scans run to $O(n^2)$, although the interpreter can optimise some to the O(n) you perhaps expected.

Scan is just like *Reduce*, but instead returns every intermediate state, not just the end state. A sum-reduce of a vector of numbers returns the sum total. A sum-scan of the same vector returns the running sums:

```
+⌿1 2 3 4 5 6 7 8 9 ⍝ Sum-reduce first
+⍀1 2 3 4 5 6 7 8 9 ⍝ Sum-scan first
```

```
45
1 3 6 10 15 21 28 36 45
```

One way we can think of scan is that it's the amalgamation of all possible calls to *Reduce* with the same operand, taking in increasing lengths of the argument array. In the case above:

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

```
1
3
6
10
15
21
28
36
45
```

As with *Reduce*, it's worth re-emphasizing that *Scan* still is evaluated right-to-left, as with everything else APL, no matter how much you'd prefer it to run left to right instead. You can, of course, roll your own scan-left if you really need it.

Power: ⍣

One of my favourite glyphs! It looks like a happy starfish! The *Power* operator is… powerful. Conceptually it should be easy to grasp, but there are some aspects that take time to understand. Formally, it's defined as a function repeatedly applied to the output of itself, until some stopping criterion is fulfilled. If you pass it an integer as its right operand, it's basically a for-loop:

```
f ← { ... }               ⍝ some function or other
f f f f f f f f argvector ⍝ repeatedly apply function to itself, eh 8 times
f⍣8 ⊢ argvector           ⍝ power-8
```

If you give it a function as the right operand, it can be used as a while-loop. One example is to find a function's *fixed point*:

```
2÷⍨⍣=10 ⍝ Divide by 2 until we reach a fixed point
```

```
0
```

Here the right operand function is *equals* =. This says: repeatedly apply the left operand (2÷⍨) until two subsequent applications return the same value.

We can explicitly refer to the left and right arguments of the right operand function. The left argument, ⍺, refers to the result of the function application to the right argument, ⍵.

Keep generating random numbers between 1 and 10 until we get a 6:

```
{⎕ ← ?10}⍣{6=⍺} 0 ⍝ Keep generating random numbers between 1 and 10 until we get a 6
```

```
1991487931157990989988905581516
6
```

The *Quad-quote-gets* (⎕⍞←) combo prints values without newlines. The final result is also returned, the expected 6.

Del: ∇

Dyalog has a most excellent, concise and efficient recursion operator, ∇. It allows you to express recursive algorithms in a natural, almost Lisp-like fashion. The interpreter has a very good TCO implementation.

Let's start with making our own version of sum-reduce, this time without actually using the *Reduce* operator.

```
]dinput
Sum ← {
    ⍺ ← 0          ⍝ Left arg defaults to 0 if not given
    0=≢⍵: ⍺        ⍝ If right arg is empty, return left arg
    (⍺+⊃⍵)∇1↓⍵     ⍝ Add head to acc, recur over tail
}
```

```
⍝ ← mysum ← Sum 1 2 3 4 5 6 7 8 9
assert mysum=+/1 2 3 4 5 6 7 8 9
```

```
45
```

Yup, that seems to work; good.

The glyph *Del* (∇) is a reference to the current innermost dfn. If your dfn has a name, you can substitute it for the actual function name. In our case, the last line could equally well have been written:

```
(⍺+⊃⍵)Sum 1↓⍵
```

However, using the glyph has a number of advantages: it's more concise, immune to function name changes, and works equally well for anonymous dfns.

Our `Sum` dfn follows a common pattern: we accumulate something as the left argument, and decrease the right argument, either by magnitude, or as in this case, by dropping items off the front of a vector.

The recursion termination guard,

```
0=≢⍵: ⍺
```

simply states that, if the right argument is empty, we should return our accumulator. The recursive call itself is:

- Add the head of the right argument to the accumulator.
- Recur with the updated accumulator as the new left argument and with the tail as the right argument.

If the last thing the function does is a function call, and this includes *Del*, this is what's called a *tail call* which the Dyalog interpreter can handle without the addition of an extra stack frame. If you're using *Del*, strive to make all your recursive functions tail calls – avoid making your function do any work on the value you get back from the recurring line – for example, note that `1+⍺∇⍵` isn't a tail call, as the last thing that happens is the `1+...` not the `⍺∇⍵`.

For a fractionally more involved example, let's write our own *sum-scan*.

```
]dinput
Sscan ← {
    ⍺ ← 0              ⍝ Left arg defaults to 0 if not given
    0=≢⍵: ⍺            ⍝ If right arg is empty, return left arg
    (⍺,⊃⍵+⊃¯1↑⍺)∇1↓⍵   ⍝ Append the sum of the head and the last element of acc and
recur on tail
}
```

```
⎕ ← myscan ← Sscan 1 2 3 4 5 6 7 8 9
assert myscan≡+⍀1 2 3 4 5 6 7 8 9
```

```
1 3 6 10 15 21 28 36 45
```

No discourse on recursion is complete without mentioning the Fibonacci sequence. You know which one I mean – every number is the sum of its two direct predecessors:

```
0 1 1 2 3 5 8 13 21 34 ⍝ etc, something about rabbits
```

Here's one possible formulation where the right argument is the Fibonacci ordinal.

```
]dinput
Fib ← { ⍝ Tail-recursive Fibonacci.
    ⍺ ← 0 1
    ⍵=0: ⊃⍺
    (1↓⍺,+/⍺)∇⍵-1
}
```

```
Fib¨⍳10 ⍝ The 10 first Fibonacci numbers
```

```
0 1 1 2 3 5 8 13 21 34
```

The pattern is still the same: set a default for the accumulator, `⍺`. Terminate on some condition on `⍵`, returning a function of the accumulator. Modify the accumulator, given the head of the right argument, and recur on the tail.

The guts of the function is the last line. To the right, we decrease the right argument – this is our loop counter if you like. To the left is our accumulator, which basically is a sliding window of size 2 over the Fib sequence. We append the sum of the two numbers, and drop the first, and recur over the tail.

Here's a pretty neat implementation of the Quicksort algorithm:

```
]dinput
Quicksort ← {
    1≥≢⍵: ⍵
    S ← {⍺≠⍨⍺ ⍺⍺ ⍵}
    ⍵((∇<S),=S,(∇>S))⍵[⍺⍺?≢⍵]
}
```

Here `ω⌷⍨?≠ω` is the pivot element, picked at random, and the `S` operator partitions its left argument array based on its left operand function and the pivot element to the right. The whole idea of quicksort is pretty clearly visible in the tacit fork `(∇<S),=S,(∇>S)` – elements *less than* the pivot, the pivot, elements *greater than* the pivot, recursively applied.

```
Quicksort ⎕←20?20
```

```
 3 10  4 15  9 11  0  7  6 14 13  5  2 19 18 12  8  1 17 16
 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
```

Another example is binary search: locate an element in an array that is known to be sorted. Here's a function to do that:

```
]dinput
bsearch ← {⎕IO←0
    _bs_ ← {                    ⍝ Operator: α,ω – lower,upper index. αα – item, ωω – array
        α>ω: 0                  ⍝ If lower index has moved past upper, item's not present
        mid ← ⌈0.5×α+ω          ⍝ New midpoint
        αα=mid⊃ωω: mid          ⍝ Check if item is at the new midpoint
        αα<mid⊃ωω: αα∇¯1+mid    ⍝ Drill into lower half
        ωω∇1+mid                ⍝ Upper half
    }
    0 (α _bs_ (,ω)) ¯1+≠,ω
}
```

```
5 bsearch 0 2 3 5 8 12 75
5 bsearch 0 2 3 5 8 12
5 bsearch 5 5
5 bsearch 5
]display 1 bsearch 0 2 3 5 8 12
```

```
3
3
1
0
┌⊖┐
│0│
└~┘
```

Performance considerations

The pattern we've used in some of the examples above,

```
⍝ some stuff
0=≠ω:α
head ← 1↑ω
tail ← 1↓ω
(head f α)∇tail
```

has a …sting in the tail, especially if you come from a functional language where that pattern is the expectation, like Racket, Erlang or Clojure. In such languages, vectors/lists are either implemented as linked lists, slices, or are immutable, meaning that dropping an element from the front is an `O(1)` operation. In APL, like in Python, that's an `O(n)` operation. When combined with recursion, this can be crushing for performance. Here's an example.

The Knuth-Morris-Pratt algorithm is an efficient string search algorithm. In one of its forms it pre-calculates a prefix table, which is a metric of how well a string matches against shifts of itself. The brilliant.org article linked to above has this written out in Python as:

```
def prefix(p):
    # https://brilliant.org/wiki/knuth-morris-pratt-algorithm/
    m=len(p)
    pi=[0]*m
    j=0
    for i in range(1,m):
        while j>=0 and p[j]!=p[i]:
            if j-1>=0:
                j=pi[j-1]
            else:
                j=-1
        j+=1
        pi[i]=j
    return pi
```

Here's an example running that:

```
Stefans-MacBook-Pro:~ stefan$ python
Python 3.8.5 (default, Sep 17 2020, 11:24:17)
[Clang 11.0.3 (clang-1103.0.32.62)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from kmp import prefix
>>> prefix('CAGCATGGTATCACAGCAGAG')
[0, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 1, 2, 1, 2, 3, 4, 5, 3, 0, 0]
>>>
```

We can write that as an APL dfn like so:

```
]dinput
prefix1 ← {⎕IO←0
    p ← ⍵
    pi ← 0⍴⍨≢⍵
    j ← 0
    {
        0=≢⍵: pi
        i ← ⊃⍵ ⍝ head
        pi[i] ← j⊢←1+{⍵<0:⍵⋄p[⍵]=p[i]:⍵⋄0≤⍵-1:∇pi[⍵-1]⋄¯1} j ⍝ while j>=0 and p[j] !=
p[i]
        ∇1↓⍵ ⍝ tail
    } 1+⍳¯1+≢⍵ ⍝ for i in range(1, m)
}
```

and we'd hope it produces the same result:

```
prefix1 'CAGCATGGTATCACAGCAGAG'
```

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

So that's two nested "loops", both nicely tail recursive, and probably similar to how you'd construct it in Clojure or Racket. However, as the argument string grows, the performance tanks. We can illustrate this by tweaking it a tiny bit to avoid the reallocation of the argument to the recursive call in the outer loop:

```
]dinput
prefix2 ← {⎕IO←0
    p ← ⍵
    pi ← 0⍴⍨≢⍵
    j ← 0
    0 {
        ⍺=≢⍵: pi
        i ← ⍺⊃⍵ ⍝ Note: pick ⍺, not first
        pi[i] ← j⊢←1+{⍵<0:⍵ ⋄ p[⍵]=p[i]:⍵ ⋄ 0≤⍵-1:∇ pi[⍵-1] ⋄ ¯1} j
        (⍺+1)∇ ⍵ ⍝ Note: no tail!
    } 1+⍳¯1+≢⍵
}
```

Instead of taking the tail of ⍵, we pass the current index as ⍺. Let's see if that works before we proceed:

```
prefix2 'CAGCATGGTATCACAGCAGAG'
```

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

To demonstrate the difference, let's compare performance on a long string. Here's one taken from Project Rosalind:

```
data ← ⊃⊃⎕NGET'../kmp.txt'1 ⍝ From http://rosalind.info/problems/kmp/
≢data ⍝ LONG STRING!
```

```
99972
```

```
'cmpx'⎕CY'dfns' ⍝ Load `cmpx` – comparative benchmarking
```

```
cmpx 'prefix1 data' 'prefix2 data'
```

```
  prefix1 data → 8.9E¯1 |   0% ████████████████████████████████████████
  prefix2 data → 2.1E¯1 | -77% ████████
```

Quite a staggering difference for such an innocuous change, perhaps.

The binary search implementation we concluded the previous section with already 'does the right thing' – it doesn't cut off the data array on each iteration. So it should be fast, right? Searching for a number amongst a hundred thousand or so *must* be faster than the APL primitive function that examines every element looking for a match. Surely…? In `Algorithms 101`

they taught you that `O(log n)` always beats `O(n)`!

Except when it doesn't:

```
data ← ⍳100000 ⍝ A loooot of numbers
cmpx 'data ⍳ 17777' '17777 bsearch data' ⍝ Look for the number 17777
```

```
  data ⍳ 17777       → 2.3E¯6 |    0% ⎕⎕⎕⎕⎕
  17777 bsearch data → 1.8E¯5 | +697% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

Ouch… a lesson for the APL neophyte here. Your intuition for what's efficient and what isn't is almost certainly wrong. Key point: simple APL primitives on simple arrays are always faster than anything you can write in APL. Only ever iterate if there are no other alternatives.

Exercise for the reader: as the data grows, sooner or later the `bsearch` function will win out. How large does the array need to be for that to happen?

Scalar pervasion

We touched briefly on *scalar pervasion* above, but it's an important topic, so let's dive in a little bit deeper. It's worth reading what Dyalog has to say on the topic in the docs.

The idea is that a certain class of functions is *pervasive*. This means that a function that operates on a scalar will operate on scalars at any level of nesting if applied to an array. Recall that the level of nesting is *not* the same as the rank in APL.

Consider a nested vector that contains both numbers and other vectors of numbers:

```
⎕ ← nesty ← (1 2 3 (3 4 (5 6)) 7)
```

```
┌─┬─┬─┬─────────┬─┐
│1│2│3│┌─┬─┬───┐│7│
│ │ │ ││3│4│5 6││ │
│ │ │ │└─┴─┴───┘│ │
└─┴─┴─┴─────────┴─┘
```

As an obvious and simple example, let's say we want to negate all the numbers:

```
-nesty
```

```
┌──┬──┬──┬──────────────┬──┐
│¯1│¯2│¯3│┌──┬──┬─────┐│¯7│
│  │  │  ││¯3│¯4│¯5 ¯6││  │
│  │  │  │└──┴──┴─────┘│  │
└──┴──┴──┴──────────────┴──┘
```

Where it can be applied, scalar pervasion is an efficient operation in Dyalog APL. It works for dyads, too:

```
(1 2) 3 + 4 (5 6)
(1 1⍴5) - 1 (2 3)
```

```
┌───┬───┐
│5 6│8 9│
└───┴───┘
┌─┬───┐
│4│3 2│
└─┴───┘
```

The Key operator: `⌸`

> Overemphasis of efficiency leads to an unfortunate circularity in design: for reasons of efficiency early programming languages reflected the characteristics of the early computers, and each generation of computers reflects the needs of the programming languages of the preceding generation. –*Kenneth E. Iverson*

```
⎕IO ← 0
]box on
]rows on
```

The monadic operator *Key* (`⌸`) groups things. It can, for example, be used to generate histograms, or if you are so inclined, you can think of it as similar to SQL's GROUP BY clause.

Other resources on *Key*:

- Dyalog docs
- APL Orchard cultivation (also covering *Stencil*)

The function derived by *Key* is ambivalent (can be called either monadically or dyadically). The operand function can be any dyadic function returning a value.

Let's look at the derived function, when called monadically:

```
{αω}⌸ 'bill' 'bob' 'bill' 'eric' 'bill' 'bob' 'eric' 'sue'
```

bill	0 2 4
bob	1 5
eric	3 6
sue	7

In this case, the operator function is called with each unique element from the argument array in turn as the left argument, and a vector of indices where they occur. If we wanted this to be a histogram, all we need to to is:

```
{α (≠ω)}⌸ 'bill' 'bob' 'bill' 'eric' 'bill' 'bob' 'eric' 'sue'
```

bill	3
bob	2
eric	2
sue	1

In the dyadic form, *Key* takes each unique element to the left, and groups the corresponding elements from the right. In other words, the following two formulations do the same thing:

```
names ← 'bill' 'bob' 'bill' 'eric' 'bill' 'bob' 'eric' 'sue'
{αω}⌸ names
names {αω}⌸ ⍳≠names
```

bill	0 2 4
bob	1 5
eric	3 6
sue	7

bill	0 2 4
bob	1 5
eric	3 6
sue	7

Let's look at a slightly more involved example. Here are the Rugby Union Gallagher English Premiership results for Jan, 2021, home fixtures only. The 0-0 games were COVID cancellations.

```
results ← 'London Irish,31–22' 'Wasps,17–49' 'Gloucester,26–31' 'Worcester Warriors,17–
21' 'Bristol,48–3' 'Leicester Tigers,15–25' 'Harlequins,27–27' 'Newcastle Falcons,22–
10' 'Exeter Chiefs,7–20' 'Northampton Saints,0–0' 'Bath,44–52' 'Sale,20–13' 'London
Irish,0–0' 'Leicester Tigers,36–31' 'Wasps,34–5' 'Gloucester,19–22' 'Bristol,29–17'
'Worcester Warriors,0–0'
```

We'll do some quick and dirty slicing and dicing to separate team names from results.

```
table ← ⍎CSV ('–'⎕R','⊢results)''4
teams ← ⊣/table
scores ← table[;1 2]
```

which gives us

```
3 6⍴teams
```

London Irish	Wasps	Gloucester	Worcester Warriors	Bristol	Leicester Tigers
Harlequins	Newcastle Falcons	Exeter Chiefs	Northampton Saints	Bath	Sale
London Irish	Leicester Tigers	Wasps	Gloucester	Bristol	Worcester Warriors

and

```
3 6⍴↓scores
```

31 22	17 49	26 31	17 21	48 3	15 25
27 27	22 10	7 20	0 0	44 52	20 13
0 0	36 31	34 5	19 22	29 17	0 0

To illustrate the similarity with SQL's `GROUP BY`, by using dyadic *Key*, we can group the scores under each team:

```
teams {⍺⍵}⌸ scores
```

London Irish	31 22 0 0
Wasps	17 49 34 5
Gloucester	26 31 19 22
Worcester Warriors	17 21 0 0
Bristol	48 3 29 17
Leicester Tigers	15 25 36 31
Harlequins	27 27
Newcastle Falcons	22 10
Exeter Chiefs	7 20
Northampton Saints	0 0
Bath	44 52
Sale	20 13

One more example. Given a vector, return the most frequent element, or if there are several, return all of those:

```
{ω⌷⍨⊂0⍨⍨⊢/0,⊢⌸ω} 'Mississippi' ⍝ Most frequent from APLCart
```

```
is
```

Ouch.

A lot of stuff there we haven't seen yet, and rather 'golfy'. Let's unpick it, and see how far we get. From the right we first have our *Key*:

```
⊢⌸ 'Mississippi'
```

```
0 0 0  0
1 4 7 10
2 3 5  6
8 9 0  0
```

That's just the indices of each unique element in turn, as if we'd written

```
{ω}⌸ 'Mississippi'
```

```
0 0 0  0
1 4 7 10
2 3 5  6
8 9 0  0
```

We then prepend a 0 column

```
0,⊢⌸ 'Mississippi'
```

```
0 0 0 0  0
0 1 4 7 10
0 2 3 5  6
0 8 9 0  0
```

Why is this? It doesn't seem to be necessary?

```
{ω⌷⍨⍋⊂0,⍨⍨⊢/⊢⌸ω} 'Mississippi'
```

```
is
```

This is just guarding against a special case – should the function be passed an empty argument, we don't want to error:

```
{ω⌷⍨⍋⊂0,⍨⍨⊢/⊢⌸ω} '' ⍝ Note: DOMAIN ERROR
```

```
DOMAIN ERROR
      {ω⌷⍨⍋⊂0,⍨⍨⊢/⊢⌸ω}'' ⍝ Note: DOMAIN ERROR
                ∧
```

whereas with the prepended zeros it does the right thing:

```
{ω⌷⍨⍋⊂0,⍨⍨⊢/0,⊢⌸ω} ''
```

Anyway. After prepending a zero-col, we pick the *last* column:

```
⊢/0,⊢⌸ 'Mississippi'
```

```
0 10 6 0
```

This is how we can find the most frequent element(s) without any sorting – the vector(s) of indices of the most frequent elements will be the longest, and hence the non-zero elements in the last column will be corresponding to the last occurrence of the most frequent elements. We need to exclude the zeros first:

```
0⍨⍨⊢/⊢⌸ 'Mississippi'
```

```
10 6
```

and then enclose and index:

```
{ω⌷⍨⍋⊂0,⍨⍨⊢/0,⊢⌸ω} 'Mississippi'
```

```
is
```

Pretty cool, eh? But there is one gotcha here –

```
{ω⌷⍨⍋⊂0,⍨⍨⊢/0,⊢⌸ω}'abc'
```

```
bc
```

If the argument array contains only unique cells, we get the wrong result if we run under `⎕IO←0`. Set it to 1 and it works:

```
{⎕IO←1⋄ω⌷⍨⍋⊂0,⍨⍨⊢/0,⊢⌸ω}'abc'
```

```
abc
```

This is, of course, to do with the excluding of zeros from the last column. Unfortunately, as we rely on 0 being the fill item for the array, so sadly we can't prepend, say, a negative number instead of 0 for ⎕IO-independence:

```
{ω⌷⍨⍋⊂¯1,⍨⍨⊢/¯1,⊢⌸ω}'abc' ⍝ Whilst this works....
{ω⌷⍨⍋⊂¯1,⍨⍨⊢/¯1,⊢⌸ω}'Mississippi' ⍝ This breaks!
```

```
abc
MisM
```

Here are some drills on *Key*, suggested by Roger Hui in his APL Exercises. Try to work out in your head what the answers are before you reveal:

```
x ← 'Supercalifragilisticexpialidocious'

{αω}⌸x
{α (≠ω)}⌸x
{≠ω}⌸x
{α}⌸x
```

```
x ← 'abcdefghijk'
y ← 10+11 2⍴⍳22

x{αω}⌸y
x{α (≠ω)}⌸y
x{≠ω}⌸y
x{α}⌸y
x{α(+/,ω)}⌸y
x{α(⌈/,ω)}⌸y
x{α(⌊/,ω)}⌸y
```

The At operator: @

> Once you understand how to write a program get someone else to write it. –*Alan Perlis*

Let's look at the powerful immutable array update operator, *At* (@). By "immutable", in this context, we mean non-destructive – it creates a new array, rather than mutating in place. Immutability is a good thing.

The dyadic @ operator has a lot of moving parts, and we won't cover all possible combinations here.

References for @:

- APL Cart has loads of examples
- The APL Cultivations covered it in Lesson 4 (amongst other things)
- Dyalog docs

Our usual prelude:

```
⎕IO ← 0
]box on
]rows on
```

@ can take both functions and arrays as either operand. In its simplest form, the left operand specifies values, and the right operand indices:

```
0@1 2 3⊢1 1 1 1 1 1
```

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

Recall from the introduction to operators that an operator returns a derived function. We inserted a *Right tack* to make clear the distinction – to stop stranding – between the operator's right operand and the derived function's argument. We could equally well have written:

```
(0@1 2 3) 1 1 1 1 1 1
```

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

We can think of the right operand as defining a selection criteria. For example, it can be a *function* that when called with the derived function's argument returns a Boolean array:

```
'*'@{0 1 1 1 0 0 0 0 0 0 0}'Hello world' ⍝ Right operand: function returning Boolean array
```

```
H***o world
```

The left operand is either an array providing the new values for every value selected by the right argument, or a function which will be called with each selected element to produce the replacement value. For example, let's add 5 to every even element:

```
{5+⍵}@{0=2|⍵} ⍳12
```

```
5 1 7 3 9 5 11 7 13 9 15 11
```

In this case, the right operand is the function `{0=2|⍵}`, which is called as such:

```
{0=2|⍵} ⍳12 ⍝ Return Boolean mask indicating the even numbers
```

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

The selected numbers are then:

```
({0=2|⍵} ⍳12)/⍳12 ⍝ Compress
```

```
0 2 4 6 8 10
```

which are passed to the left operand:

```
{5+⍵} 0 2 4 6 8 10 ⍝ Left operand applied to elements selected by right operand
```

```
5 7 9 11 13 15
```

Finally, @ will make the substitution.

Let's write our own naive, overly verbose, partial implementation of @ for illustrative purposes.

```
]dinput
_At_ ← { ⍝ Partial @ - left and right operands must be functions, and the right arg a
vector
    mutator ← ⍺⍺                    ⍝ Left operand -- function only
    selector ← ⍵⍵                   ⍝ Right operand -- function only
    data ← ⍵                        ⍝ Derived function right argument -- vector only.
Note: copy
    mask ← selector data
    selection ← mask/data           ⍝ Compress the data according to boolean mask
    newvals ← mutator selection     ⍝ Mutate our selection
    (mask/data) ← newvals           ⍝ Replace with the mutated values in copy
    data                            ⍝ Return result
}
```

```
{5+⍵}_At_{0=2|⍵} ⍳12
```

```
5 1 7 3 9 5 11 7 13 9 15 11
```

We can shrink that down a bit without any real loss of clarity:

```
]dinput
_At_ ← { ⍝ Partial @ - left and right operands must be functions, and the right arg a
vector
    data ← ⍵
    (mask/data) ← ⍺⍺ (mask←⍵⍵ ⍵)/⍵
    data
}
```

```
{5+⍵}_At_{0=2|⍵} ⍳12
```

```
5 1 7 3 9 5 11 7 13 9 15 11
```

We used leading and trailing underscores when naming our operator to indicate visually that it's a dyad, as suggested by Adám Brudzewsky's unofficial style guide.

Higher rank: choose and reach

The real @ has many more tricks up its sleeve, of course. It can also be applied to arrays of any rank, not just vectors.

If we look at Dyalog's specification of @ we have

```
R←{X}(f@g)Y
```

If g is a simple vector, it chooses *major cells* in Y. If g is nested, it specifies indices for *choose* or *reach* indexing. What does that mean, then? One way to think of it is that the g vector behaves just as if it had been inserted between square brackets.

The first case is straightforward: the major cells of an array are those given by the first axis of the shape. In the case of a 2D matrix, its rows:

```
-@0 2 ⊢ 3 3⍴1 2 3 4 5 6 7 8 9 ⍝ Negate rows 0 and 2: major cells
```

```
¯1 ¯2 ¯3
 4  5  6
¯7 ¯8 ¯9
```

If we want to access individual elements we can use *choose indexing* – which is a nested vector of indices:

```
-@(0 0)(2 2) ⊢ 3 3⍴1 2 3 4 5 6 7 8 9 ⍝ Negate corners of main diagonal by choose
indexing
```

```
¯1 2  3
 4 5 ¯6
 7 8 ¯9
```

We can also access elements that are deeply nested using *reach indexing*, which we met briefly in the indexing section:

```
□ ← G ← 2 3⍴('Adam' 1)('Bob' 2)('Carl' 3)('Danni' 4)('Eve' 5)('Frank' 6)
'***' ¯999@((0 0)0)((1 2)1)⊢G
```

Adam	1	Bob	2	Carl	3
Danni	4	Eve	5	Frank	6

***	1	Bob	2	Carl	3
Danni	4	Eve	5	Frank	¯999

Examples

Here's a random collection of handy @ expressions. Many more on APLCart, too.

Replace dashes with spaces

```
' '@(=⍤'-') 'Hello-World-One-Two'
```

```
Hello World One Two
```

Pad an array with zeros

```
{⍵@(1+⍳⍴⍵)⊢0⍴⍨2+⍴⍵} 2 2⍴1 1 1 1
```

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

Array search and replace

```
]DISPLAY array ← ?10 3⍴10
]DISPLAY 1 2 3 4 {⍺⍺(⍵⍵⍨⍤⊂⍳)@(∊⍤⍺⍺)⍵} 0 10 100 1000 ⊢ array
```

```
┌→──┐
↓3 2 3│
 9 0 4│
 2 1 9│
 9 1 4│
 8 7 9│
 3 1 1│
 5 7 9│
 9 0 9│
 8 9 9│
 8 8 9│
└~──┘
```

```
┌→──────┐
↓100 10  100│
   9  0 1000│
  10  0    9│
   9  0 1000│
   8  7    9│
 100  0    0│
   5  7    9│
   9  0    9│
   8  9    9│
   8  8    9│
└~──────┘
```

Boolean alternating selection

Here's a creative use of @ that Adám Brudzewsky posted on APL Orchard.

Given two arrays A, B and a boolean filter C, select items from A where C is false and from B where C is true:

```
A ← 1 2 3 4
B ← 11 22 33 44
C ← 0 1 0 1
(C/⍤⌶,B)@{C}A
```

```
1 22 3 44
```

If that looks puzzling (we haven't met the ⍤⌶ operator yet!), for vectors, it's just

```
(C/B)@{C}A
```

```
1 22 3 44
```

All that says is: replace the true spots (as defined by C) in A with the true spots (as defined by C) from B, which we could also do as an assignable indexing expression if we don't mind mutating A:

```
(C/A) ← C/B
```

```
A
```

```
1 22 3 44
```

The Rank/Atop operator: ⍤

> An algorithm must be seen to be believed. –*Don Knuth*

The somewhat startled-looking *Rank* operator, ⍤, has a reputation for being one of the harder ones to grasp. It sits on the 'J' key, a handy mnemonic, as it was an invention borrowed from the J-language. Well, it makes sense to *me*.

If ⍤ is given a right operand *function* it becomes *Atop*. We'll talk about that after we've covered the *Rank* version.

Other resources on *Rank* and *Atop*:

- APL Wiki
- Dyalog docs Rank, Atop
- Webinars: Rank Operator, Advanced Rank, Rank and Dyadic Transpose
- APL Orchard cultivations: Basic Rank, Advanced Rank, Atop and Over
- Roger Hui: Rank Operator: An Idea Worth ~~Stealing~~ Borrowing

```
⎕IO ← 0
]box on
]rows on
```

In essence, the simple use of *Rank* drills into the argument array and applies its operand to sub-cells of a specified rank only.

Consider the following rank 3 array:

```
]DISPLAY m ← 3 3 3⍴27?27
≢⍴m
```

```
┌┌→──────┐
↓↓ 3  0 21 │
 │19 15  9 │
 │24  6 17 │
 │         │
 │26  8 20 │
 │ 4 14 12 │
 │11 23  5 │
 │         │
 │18  7  2 │
 │13 16  1 │
 │22 25 10 │
 └└~──────┘
3
```

Let's say we want to drop the first row of each layer of this array. In this case, we could use a bracket-axis specification to drop:

```
1↓[1]m ⍝ Apply drop ↓ to axis 1, i.e. the y-axis, with ⎕IO←0
```

```
19 15  9
24  6 17

 4 14 12
11 23  5

13 16  1
22 25 10
```

However, this is now ⎕IO dependent, and the bracket-axis spec has some other undesirable properties, including the fact that it cannot be applied to user-defined functions.

The *Rank* operator allows us to solve this in a different way:

```
1↓⍤2⊢m
```

```
19 15  9
24  6 17

 4 14 12
11 23  5

13 16  1
22 25 10
```

You can think of *Rank* in this case being closer to the problem statement: drop the first major cell of each *sub-cell* that has a rank one less than the argument. Or: in our 3D array, apply the drop to each of its constituent 2D arrays.

Note that *Rank* doesn't equal *depth*, and this is part of the reason why the *Rank* operator can be confusing. So given a complex vector that contains a variety of scalars and higher-rank arrays, you can't use the *Rank* operator to selectively apply a function to, say, just the vectors.

```
⍝ ← V ← (1 2 3)(2 2⍴1 1 1 1)(1)
```

```
┌─────┬─────┬─┐
│1 2 3│1 1  │1│
│     │1 1  │ │
└─────┴─────┴─┘
```

```
≢⍤1⊢V ⍝ NOTE: this is wrong!
```

```
3
```

In this case, V is of course already a rank 1 array (i.e. a vector). But wait, there is more confusion to be had! The right operand doesn't even have to be a scalar at all.

We already saw that if the right operand of ⍤ is a scalar, we are specifying which rank sub cells we want a function to apply to. For dyadic usage, we instead specify which sub-cells of the left argument should be paired up with which sub-cells of the right argument. It takes a while to wrap your head around this.

Here's an example:

```
1 2 3,⍤0 1 ⊢ 'hello'
```

```
1 hello
2 hello
3 hello
```

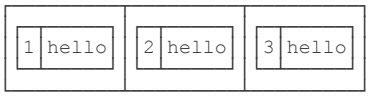
This says that the *left* argument of the derived function should be paired at rank 0 (that is each element) with the *right* argument at rank 1 (that is the whole vector in this case). If this feels reminiscent of an outer product, you're not wrong.

In other words, *Rank pairs up* the arguments of the function it derives. We can examine which elements would be paired up by using the following incantation in place of our function:

```
rankpairs ← ⊂⍤,⍥⊂
```

if you know what that means, you probably don't need to read this tutorial. If not, don't worry – treat it as a debug statement:

```
1 2 3 rankpairs⍤0 1 ⊢ 'hello'
```



In the section on array indexing earlier, we introduced the concept of *Sane Indexing* without any further explanation on how it was working. Here it is again, as a tacit function:

```
I←⌷⍨⍥⊃⍨⍤0 99 ⍝ Sane indexing, a.k.a select
```

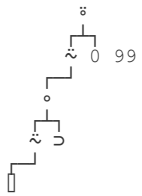
Now that we understand a bit about *Rank*, we can at least make a start in taking that apart. So we have a *Rank* 0 99 to the right, stating that we should combine the left argument at rank 0 (scalars) with the right argument at "rank 99". Rank 99 means "full rank", as Dyalog has a max rank of 99. So we should combine "each scalar" to the left with "the whole thing, at whatever rank it happens to be" to the right.

The actual indexing is done by the leading *Squad* (⌷). Let's de-selfie the expression and make it a dfn. Although we haven't covered tacit functions yet, we can have the interpreter help us out a bit to see how that expression actually parses:

```
]box on -trains=tree ⍝ Visualise tacit functions as a parse tree
```

```
Was ON -trains=tree
```

```
⌷⍨⍥⊃⍨⍤0 99
```



If we separate out the *Rank* expression, after a bit of head-scratching, we arrive at

```
sane←{(⊃⍺)⌷⍵}
```

```
3 sane 'abcdefg'
```

```
d
```

```
1 2 3 sane⍤0 99⊢'abcdefg'
```

```
bcd
```

Hopefully, we can now see a bit clearer how it works – we simply pass the disclosed left argument vector to *Squad*, and *Rank* does the appropriate combination of left and right: each left element, to the whole (full-rank) of the right.

Here's another example. Given two integer arrays of the same shape, can we replicate (or compress) the rows of one from the other? We might be excused if we think that we can replicate-reduce, but this doesn't work:

```
]DISPLAY A ← 3 4⍴2 1 1 1 1 2 1 1 1 1 2 1
]DISPLAY B ← 3 4⍴1 2 3 4 1 2 3 4 1 2 3 4
```

```
┌→──────┐
↓2 1 1 1│
 1 2 1 1│
 1 1 2 1│
 ~      │
└~──────┘
```

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

(note: all expressions in the next cell will generate syntax or rank errors)

```
A⌾//B     ⍝ Nope... SYNTAX ERROR
{A/⍵}/B   ⍝ Nope... RANK ERROR
A⌾{α/⍵}/B ⍝ Nope... SYNTAX ERROR
```

```
SYNTAX ERROR: The function does not take a left argument
      A⌾//B     ⍝ Nope... SYNTAX ERROR
        ∧
```

```
RANK ERROR
      {A/⍵}/B   ⍝ Nope... RANK ERROR
        ∧
```

```
SYNTAX ERROR: The function does not take a left argument
      A⌾{α/⍵}/B ⍝ Nope... SYNTAX ERROR
            ∧
```

Instead, we need to reach for *Rank*:

```
A/⍤1⊢B
```

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

If we spell it out in plain words, perhaps the use of *Rank* suggests itself: combine each major cell on the left with the corresponding major cell on the right, and apply the dyadic function "replicate". Every time you think "combine each major cell on the left…", you should think *Rank*.

As we've seen elsewhere, APL has a somewhat unholy mix of leading and trailing axis operators, mainly for historical reasons. *Rank* allows us to stick to the leading axis versions [Kromberg and Hui]:

Leading axis	**Trailling axis**	**Operation**
(⊖⍤1)⍵	⌽⍵	reverse
α(⊖⍤1)⍵	α⌽⍵	rotate (scalar α)
(f⌿⍤1)⍵	f/⍵	reduce
(f⍀⍤1)⍵	f\⍵	scan

Rank drills

In his paper APL Excercises, Roger Hui sets out the following drills for *Rank*. See if you can predict the results before revealing the answers:

```
,t←2 3 4⍴⍳24
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

```
,⍥3⊢t
]DISPLAY ,⍥2⊢t
]DISPLAY ,⍥1⊢t
```

```
⊂⍥3⊢t
⊂⍥2⊢t
⊂⍥1⊢t
```

```
↑ ,⍨ ⊂⍥3⊢t
]DISPLAY ↑ ,⍨ ⊂⍥2⊢t
]DISPLAY ↑ ,⍨ ⊂⍥1⊢t
```

```
]DISPLAY +⌿⍥3⊢t
]DISPLAY +⌿⍥2⊢t
]DISPLAY +⌿⍥1⊢t
```

```
]DISPLAY ⊖⍥3⊢t
]DISPLAY ⊖⍥2⊢t
]DISPLAY ⊖⍥1⊢t
```

If you got all those right, well done – Roger's paper has got many more for you to pit your wits against.

Atop

As we mentioned above, from version 18 of Dyalog, ⍥ can also be *Atop* if given a right operand function instead of array. We'll see more of different kinds of *Atop* later in the tacit chapter, but for now it's a functional composition rule, stating that (for the dyadic case),

```
X f⍥g Y ← f X g Y
```

It is similar enough to *Jot*, ∘, to be confusing. Whilst monadic ∘ is an atop, dyadic ∘ is a *beside*. Behold:

```
f∘g X ← f(gX) ← fgX
f⍥g X ← f(gX) ← fgX
X f∘g Y ← X f g Y
X f⍥g Y ← f X g Y
```

Another way to think about it is that in the dyadic case, the difference between ∘ and ⍥ is which operand function (f or g) gets the left argument. In the monadic case, *Atop* is equivalent for ∘ and ⍥, so it's worth using ⍥ for both monadic and dyadic atops for consistency.

Here is a simple example of a dyadic *Atop*,

```
1 2 3 -⍥+ 4 5 6 ⍝ Negate the sum of two vectors
```

```
¯5 ¯7 ¯9
```

which, following the `X f⍥g Y ← f X g Y` rule, is the same as

```
-1 2 3+4 5 6 ⍝ Negate the sum of two vectors
```

```
¯5 ¯7 ¯9
```

or, as a tacit formulation (which we'll cover in more detail later),

```
1 2 3 (-+) 4 5 6 ⍝ Negate the sum of two vectors
```

```
¯5 ¯7 ¯9
```

The Stencil operator: ⌺

> Every time I see some piece of medical research saying that caffeine is good for you, I high-five myself. Because I'm going to live forever. –*Linus Torvalds*

Are you familiar with Conway's *Game of Life*? If not, it's a cellular automaton with a deceptively simple set of rules:

1. Any live cell with two or three live neighbours survives.
2. Any dead cell with three live neighbours becomes a live cell.
3. All other live cells die in the next generation.

yet *Game of Life* gives rise to extraordinary complex patterns. It's fun to implement – try it in your favourite language! Some good ones can be found on Rosettacode.

```
⎕IO ← 0
]box on
]rows on
```

Game of Life is a bit of an APL party trick, to the extent that it may be the only application of APL that people have heard of after seeing a famous video of John Scholes'.

Here's the current champion, all 17 symbols of it:

```
gol ← {≠⌸ω}⌺3 3⍷¨3+0,¨⊢ ⍝ Generate the next generation
```

To run it, we first need an array with some live cells in it. Here's an 8×8 petri dish containing a "glider":

```
glider ← 3 3⍴0 0 1 1 0 1 0 1 1
dish ← 8 8⍴0
dish[(⊂3 3)+⌸glider] ← 1
dish
```

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

The glider pattern is called so because it glides across the dish. Here's a full cycle:

```
1 4⍴(gol⍣1⊢dish)(gol⍣2⊢dish)(gol⍣3⊢dish)(gol⍣4⊢dish)
```

```
┌─────────────────┬─────────────────┬─────────────────┬─────────────────┐
│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  │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 1 0 0  │0 0 0 0 0 0 0 0  │0 0 0 0 0 0 0 0  │
│0 0 0 0 0 1 1 0  │0 0 0 0 0 0 1 0  │0 0 0 0 1 0 1 0  │0 0 0 0 0 0 1 0  │
│0 0 0 0 1 1 0 0  │0 0 0 0 1 1 1 0  │0 0 0 0 0 1 1 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 1 0 0  │0 0 0 0 0 1 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  │
└─────────────────┴─────────────────┴─────────────────┴─────────────────┘
```

If you'd like to have the above Game of Life implementation dissected in detail, head over to dyalog.tv.

Anyway – this APL *Game of Life* implementation is primarily a demonstration of the power and versatility of the *Stencil* operator, ⌺. In simple terms, the dyadic *Stencil* operator lets you specify a sliding window of a rank of your choice to the right and a function operating over this sliding window to the left. It makes it trivial to implement image processing filters or convolution. In essence, for the 2D case, it's four nested for-loops, the outermost two moving the window, and the innermost two visiting every data point within the window.

This is obviously a good fit for *Game of Life*: the rules are defined within a 3×3 neighbourhood, so all we need to do is count the 1s in the windows that *Stencil* delivers:

```
{≠⌸ω} glider
```

Let's look at *Stencil* in a bit more detail. We can illustrate the sliding window using enclose as the left operand and a size 2 window over a vector:

```
⊂⍤⊢⌺2⊢⍳10
```

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

But now look what happens if we increase the window size to 3:

```
⊂⍤⊢⌺3⊢⍳10
```

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

The first and last windows now "overhang" the edge, filling the overhang with zeros. The way to think of it is that each data point in the argument array must line up with the *centre* of the window.

Stencil has a few more tricks up its sleeve. The right operand can be an array, where the first row defines the shape of the window, and the second row defines the *step size*. Again, using the vector above, we can say that we want a size 3 window, but instead of moving it one step (the default), we want to shift it three steps each time:

```
⊂⍤⊢⌺(↕3 3)⊢⍳10
```

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

We still have an odd-size window, and the first is centered over the first item as before. The step size, 3, dictates how far the center of the window moves, so the next window is centered over item 3 etc.

With an even size we can create non-overlapping windows that don't overhang,

```
⊂⍤⊢⌺(↕2 2)⊢⍳10
```

0 1	2 3	4 5	6 7	8 9

and, of course, even-sized, non-overlapping windows that *do* overhang, too, if that's what we want:

```
⊂⍤⊢⌺(↕4 4)⊢⍳10
```

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

The left argument

The left operand function is actually called with a left argument; a vector indicating the number of fill arguments per axis: positive for the "before", and negative for the "after".

```
{⊂⍺ ⍵}⌺3 3⊢3 5⍴⎕A ⍝ Show the left argument vector
```

1 1 AB FG	1 0 ABC FGH	1 0 BCD GHI	1 0 CDE HIJ	1 ¯1 DE IJ
0 1 AB FG KL	0 0 ABC FGH KLM	0 0 BCD GHI LMN	0 0 CDE HIJ MNO	0 ¯1 DE IJ NO
¯1 1 FG KL	¯1 0 FGH KLM	¯1 0 GHI LMN	¯1 0 HIJ MNO	¯1 ¯1 IJ NO

Another way to think of it is as what you need to drop to get rid of all the fillers:

```
{⊂α↓ω}⌺3 3⊢3 5⍴⎕A ⍝ Drop all fillers
```

AB FG	ABC FGH	BCD GHI	CDE HIJ	DE IJ
AB FG KL	ABC FGH KLM	BCD GHI LMN	CDE HIJ MNO	DE IJ NO
FG KL	FGH KLM	GHI LMN	HIJ MNO	IJ NO

The Över operator: ⍥

> Last week I was listening to a podcast on Hanselminutes, with Robert Martin talking about the SOLID principles… They all sounded to me like extremely bureaucratic programming that came from the mind of somebody that has not written a lot of code, frankly. –*Joel Spolsky*

If you thought *rank* looked startled, ⍤, wait until you meet his big sister, over: ⍥, *nom de guerre* for *Circle Diaresis*. `⍥over` was introduced to Dyalog in version 18. Over is a function composition rule, similar to those that govern tacit programming. Once you "see" this pattern, you'll see it everywhere. Let's take a look.

`⍥over` complements jot ∘ and the atop version of ⍤ in that it specifies how a set of functions should be applied to common arguments. If we compare `f∘g` and `f⍤g`, 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 the same. For the monadic case, these are all the same:

- `f∘g Y`
- `f⍤g Y`
- `f⍥g Y`

In the dyadic case, look at the order of the leftmost two tokens:

- `X f∘g Y` ⟶ `X f g Y`
- `X f⍤g Y` ⟶ `f X g Y`

and `⍥over` completes this by

- `X f⍥g Y` ⟶ `(g X)f(g Y)`

Clear as mud? But this really is a common pattern. For example, let's say we want to know if one vector is longer than another, we would previously have written:

```
X ← ⍳7
Y ← ⍳9
(≢X)>(≢Y)
```

```
done

Rebuilding user command cache... done
0
```

Using `⍥over` we can instead now write

```
X >⍥≢ Y
```

```
0
```

In words: first apply the right function to both arguments, then apply the left between the results. And of course, either side can be a user-defined function, too. Which vector have the most even numbers?

```
X >⍥{+/0=2|⍵} Y
```

```
0
```

Dyadic transpose: A⍉B

> The development of mathematics toward greater precision has led, as is well known, to the formalization of large tracts of it, so that one can prove any theorem using nothing but a few mechanical rules… One might therefore conjecture that these axioms and rules of inference are sufficient to decide any mathematical question that can at all be formally expressed in these systems. It will be shown below that this is not the case, that on the contrary there are in the two systems mentioned relatively simple problems in the theory of integers that cannot be decided on the basis of the axioms. –*Kurt Gödel*

```
⎕IO ← 0
]box on -style=max
]rows on
assert←{⍺←'assertion failure' ⋄ 0∊⍵:⍺ ⎕signal 8 ⋄ shy←0}
```

Mastery of the dyadic form of *Transpose* (`A⍉B`) – together with *Rank* (`⍤`) – is considered the mark of the accomplished array programmer. Let's see if it's as hard to grasp as its reputation suggests.

Some other resources on dyadic transpose:

- APLWiki
- Dyalog docs
- Webinar: The Rank Operator and Dyadic Transpose

Most programmers have probably had at least some exposure to linear algebra, and so have internalised the concept of the transpose of a matrix: the x-axis becomes the y-axis, and vice-versa. We've alreay met the monadic form:

```
⍝ ← B ← 3 4⍴9 4 2 7 2 5 4 7 8 6 1 2 6 8 2 9
⍉B
```

```
┌→──────┐
↓9 4 2 7│
 2 5 4 7│
 8 6 1 2│
└~──────┘
```

```
┌→────┐
↓9 2 8│
 4 5 6│
 2 4 1│
 7 7 2│
└~────┘
```

We can see what was the x-axis is now the y-axis as we said. If we generalise this a bit, transpose is an operation that *reorders* axes. It just happens to be the case that in the rank-2 case, there is only one other possible ordering.

In the dyadic form of transpose, we are explicit about the new axis order:

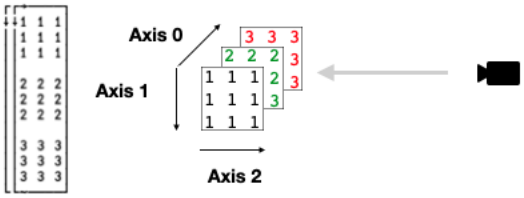
```
1 0⍉B ⍝ Same result as the monadic form: x-is-y
```

```
┌→────┐
↓9 2 8│
 4 5 6│
 2 4 1│
 7 7 2│
└~────┘
```

So far, so obvious. However… once the rank increases, it becomes harder to visualise perhaps what's going on. One tip is to not think of the array itself having its axes pulled and moved, but to instead think of the *observer* (you, or "the camera") moving around the array.

Consider rank-3:

```
m ← 3 3 3⍴1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3
```



Imagine if we want to rearrange this array such that the new view is that indicated by the camera symbol, to see the array as three slices each of:

```
3 3⍴1 2 3
```

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

We'd need to rotate the scene 90 degrees clockwise around the current axis 1. So axis 1 stays the same, but axes 0 and 2 changes position:

```
2 1 0⍉m
```

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

When rank goes above 3, it becomes tricker to visualise this way, but instead, think about what each resulting major cell should look like – typically, when you need dyadic transpose, you have an idea what shape each cell should take, and you can usually work backwards from there to deduce the correct axis ordering.

But perhaps the main question remains – what is this all useful for? A *great deal*, as it turns out. One helpful clue is that *Transpose*, including its dyadic form, is one of the functions allowing for modified assignment. We talked about that in the section on indexing earlier. By assigning to the dyadic transpose of an array we can "fill" it from a data source that has the elements laid out in a different arrangement.

We can use the same example data as above to illustrate this. Let's say that our data is a ravel of recurring 1, 2, 3 to give 27 elements, but we want it organised as our original matrix we used above, with the major cells being `3 3⍴9⍳1`, `3 3⍴9⍳2` and `3 3⍴9⍳3` respectively

```
□ ← data ← ∊9/⊂1 2 3
mat ← 3 3 3⍴0              ⍝ Empty matrix
```

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

We can now assign through the dyadic *Transpose* to fill our empty matrix in a different axis order:

```
(2 1 0⍉mat) ← 3 3 3⍴data
mat
```

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

Let's look at a higher-rank example. Given a 6×6 matrix, partition it into four non-overlapping partitions of size 3×3. Can we achieve this using dyadic *Transpose*? Here's our matrix:

```
□ ← mat ← 6 6⍴⍳36
```

```
┌→
↓ 0  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 32 33 34 35
└~
```

Let's reshape this into an array of shape 2 3 2 3 (rank 4), like so:

```
□ ← r4 ← 2 3 2 3⍴mat
```

```
┌┌┌→
↓↓↓ 0  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 32
│ 33 34 35
└└└~
```

The first cell has turned the first three rows of our original matrix into a rank 3 array where the cells are shape 2 3, essentially "folding" each row in half:

```
0⌷r4
```

```
┌┌→
↓↓ 0  1  2
│ 3  4  5
│
│  6  7  8
│  9 10 11
│
│ 12 13 14
│ 15 16 17
└└~
```

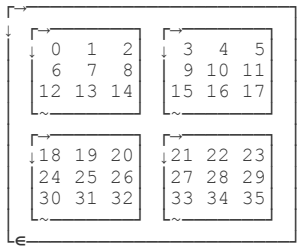
We can now look to reorder the axes. After a bit of thought we can say that we need the shape to be 2 2 3 3 before we have a go at partitioning. If we think of the 3×3 at the end as given by the task at hand (our partition sizes), we know that we'll ultimately need two "rows" and two "cols" of those. So we currently have a shape of 2 3 2 3, so we flip the middle two axes:

```
□ ← reorder ← 0 2 1 3⍉r4
```

```
┌┌┌→
↓↓↓ 0  1  2
│││ 6  7  8
│12 13 14
│
│  3  4  5
│  9 10 11
│ 15 16 17
│
│
│ 18 19 20
│ 24 25 26
│ 30 31 32
│
│ 21 22 23
│ 27 28 29
│ 33 34 35
└└└~
```

That looks right: each 3×3 cell is correct. We just need to enclose each rank-2 cell:

```
⊂⍤2⊢reorder
```



Translating RNA into Protein

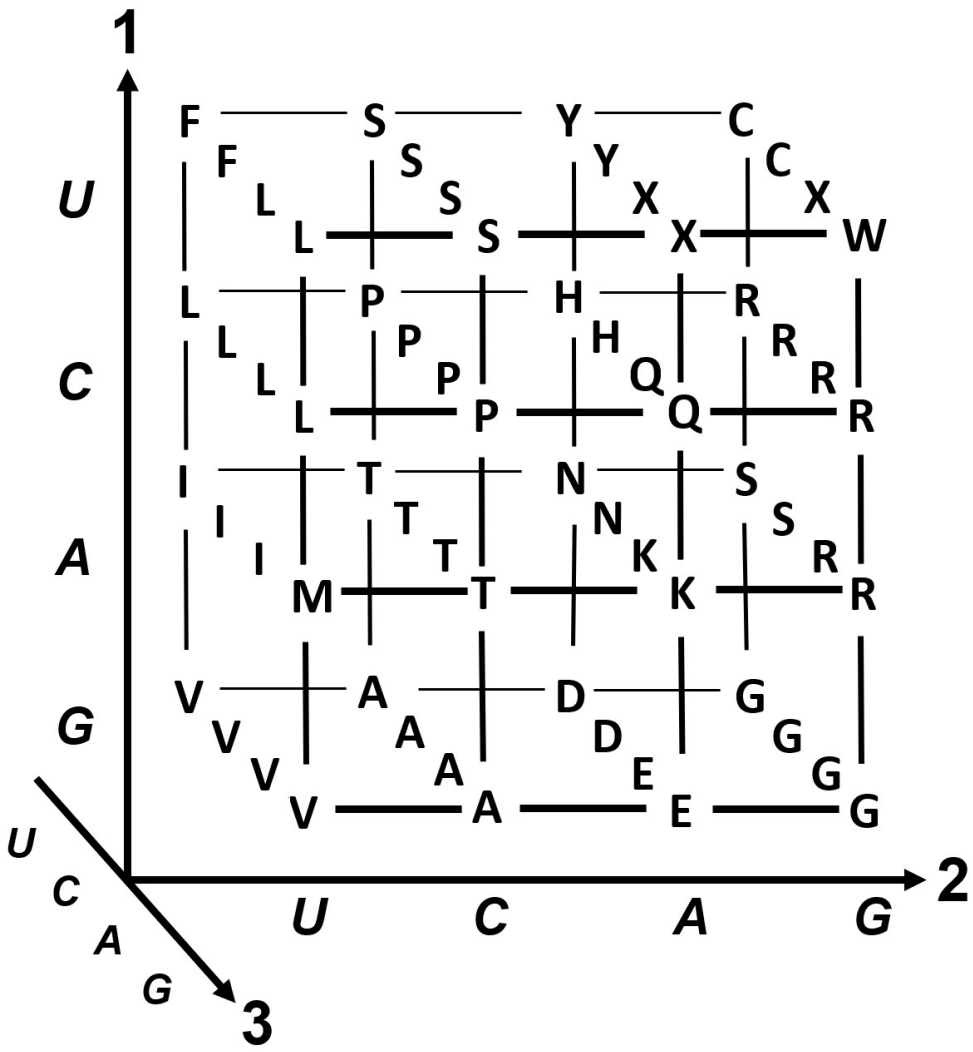
Here's an example from Project Rosalind – a great problem collection in the field of bioinformatics: http://rosalind.info/problems/prot/

The ask is to take an mRNA sequence and encode specific "codons" into the amino acid alphabet. No, I don't know what that means either, but it doesn't matter: we have a sequence of letters, which when grouped into triplets define a point in a 4×4×4 matrix.

The "codon table" is given as:

```
UUU F      CUU L      AUU I      GUU V
UUC F      CUC L      AUC I      GUC V
UUA L      CUA L      AUA I      GUA V
UUG L      CUG L      AUG M      GUG V
UCU S      CCU P      ACU T      GCU A
UCC S      CCC P      ACC T      GCC A
UCA S      CCA P      ACA T      GCA A
UCG S      CCG P      ACG T      GCG A
UAU Y      CAU H      AAU N      GAU D
UAC Y      CAC H      AAC N      GAC D
UAA Stop   CAA Q      AAA K      GAA E
UAG Stop   CAG Q      AAG K      GAG E
UGU C      CGU R      AGU S      GGU G
UGC C      CGC R      AGC S      GGC G
UGA Stop   CGA R      AGA R      GGA G
UGG W      CGG R      AGG R      GGG G
```

and you'd be excused for not seeing the 4×4×4 matrix immediately popping out at you. However, the question links to a Wikipedia page which has the following helpful illustration:



So let's take the values from the table given and make the above 3D representation from it, and then solve this Rosalind problem. The "Stop" codons is basically just the "EOL" – we can call that '.' for simplicity. Picking the values and reshaping those to 4×4×4:

```
4 4 4⍴'FLIVFLIVLLIVLLMVSPTASPTASPTASPTAYHNDYHND.QKE.QKECRSGCRSG.RRGWRRG'
```

```
┌→──┐
↓↓FLIV│
│FLIV│
│LLIV│
│LLMV│
│    │
│SPTA│
│SPTA│
│SPTA│
│SPTA│
│    │
│YHND│
│YHND│
│.QKE│
│.QKE│
│    │
│CRSG│
│CRSG│
│.RRG│
│WRRG│
└────┘
```

Now let's figure out the axis order. Looking at the table (and graph), the coordinate order is U, C, A, G. Let's look at the first few entries in the table again:

```
UUU F      CUU L      AUU I      GUU V
UUC F      CUC L      AUC I      GUC V
UUA L      CUA L      AUA I      GUA V
UUG L      CUG L      AUG M      GUG V
```

We can see that when the leading coordinate changes we get `FFLL`, `LLLL`, `IIIM`, `VVVV`, so our current axis ordering isn't correct. Looking at the graph, we can see that this corresponds to placing the "camera" to the left, looking right – our first axis is probably where it should be. Let's try to flip the other two:

```
0 2 1⍉4 4 4⍴'FLIVFLIVLLIVLLMVSPTASPTASPTASPTAYHNDYHND.QKE.QKECRSGCRSG.RRGWRRG'
```

```
┌┌→┐
↓↓FFLL
 ││LLLL
 ││IIIM
 ││VVVV
 ││
 ││SSSS
 ││PPPP
 ││TTTT
 ││AAAA
 ││
 ││YY..
 ││HHQQ
 ││NNKK
 ││DDEE
 ││
 ││CC.W
 ││RRRR
 ││SSRR
 ││GGGG
 └└─┘
```

Hmm, the first row of the first cell *is* correct, but not the rest, but if the first axis is correct we have no options left, right? Let's try a monadic transpose, too:

```
0 2 1⍉⍉4 4 4⍴'FLIVFLIVLLIVLLMVSPTASPTASPTASPTAYHNDYHND.QKE.QKECRSGCRSG.RRGWRRG'
```

```
┌┌→┐
↓↓FFLL
 ││SSSS
 ││YY..
 ││CC.W
 ││
 ││LLLL
 ││PPPP
 ││HHQQ
 ││RRRR
 ││
 ││IIIM
 ││TTTT
 ││NNKK
 ││SSRR
 ││
 ││VVVV
 ││AAAA
 ││DDEE
 ││GGGG
 └└─┘
```

That looks correct. Now we can solve this problem:

```
codon ← 0 2 1⍉⍉4 4 4⍴'FLIVFLIVLLIVLLMVSPTASPTASPTASPTAYHNDYHND.QKE.QKECRSGCRSG.RRGWRRG'
```

```
]dinput
prot ← {
    enc ← 0 1 2 3['UCAG'⍳⍵]        ⍝ Convert RNA string from UCAG to 0123
    ¯1↓codon[⊂⍤1⊢(3÷⍨≠enc) 3⍴enc]  ⍝ Group into triplets and index into codon table,
and drop Stop.
}
```

We can try this on the example given in the question:

```
test ← 'AUGGCCAUGGCGCCCAGAACUGAGAUCAAUAGUACCCGUAUUAACGGGUGA'
⍝ ← r ← prot test
assert 'MAMAPRTEINSTRING'≡r
```

```
┌→┐
MAMAPRTEINSTRING
```

Best figures in $MONTH

One more? Let's say we have a team of sellers who report their results monthly. We have a set of historical data going back a number of years. Which seller produced the best results in a particular month in a particular year?

Here are the monthly figures for our twenty five sellers going back ten years (as a random selection, like all sales figures are):

```
⎕IO ← 0
sales ← ?10 25 12⍴250
```

Our array has the axes ordered year, seller id, month. So how can we pick the best performing seller in (say) April, year 6?

Most APLers would probably reach for bracket indexing, and rightly so – it feels natural:

```
year ← 6
month ← 3
sales[year;;month] ⍝ Sales figures for month/year
```

```
┌→──────────────────────────────────────────────────────────────────────────────┐
│120 73 153 208 22 152 223 111 47 169 145 154 190 161 158 72 221 164 155 214 8 1 55 36 234 │
└~──────────────────────────────────────────────────────────────────────────────┘
```

But we can also dyadic *Transpose* here to avoid the bracket indexing. We have a couple of options – we want to select sellers results over a specific month in a specific year, so one approach is to make the year the first axis, followed by the month and then seller id: first go to the filing cabinet for year 6. Pull out the April drawer. Rifle through the folders of sales results and pick the one with the largest total.

```
year month⍉0 2 1⍉sales
```

```
┌→──────────────────────────────────────────────────────────────────────────────┐
│120 73 153 208 22 152 223 111 47 169 145 154 190 161 158 72 221 164 155 214 8 1 55 36 234 │
└~──────────────────────────────────────────────────────────────────────────────┘
```

Picking the best is the first item in the grade-down:

```
⊢employee_of_the_month←⊃⍒year month⍉0 2 1⍉sales
```

```
24
```

We could have achieved the same thing with axes reversed, too:

```
month year⍉1 2 0⍉sales
```

```
┌→──────────────────────────────────────────────────────────────────────────────┐
│120 73 153 208 22 152 223 111 47 169 145 154 190 161 158 72 221 164 155 214 8 1 55 36 234 │
└~──────────────────────────────────────────────────────────────────────────────┘
```

The filling order

When we shape a matrix in APL, the "filling order" follows the axis order. When this filling order isn't what you need, that's another indication that you may want to consider dyadic transpose. Consider an example: let's say you have a vector of integers

```
⊢data←⍳45
```

```
┌→──────────────────────────────────────────────────────────────────────────────────────────────┐
│0 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 32 33 34 35 36 37 38 39 40 41 │
└~──────────────────────────────────────────────────────────────────────────────────────────────┘
```

where each group of 9 has 3 sub-groups that we'd like to separate out. Without dyadic transpose, perhaps we'd try something like this:

```
⍝ ← m←(9÷⍨≢data) 9⍴data
↑(m[;0 1 2])(m[;3 4 5])(m[;6 7 8])
```

```
 0  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 32 33 34 35
36 37 38 39 40 41 42 43 44
```

```
 0  1  2
 9 10 11
18 19 20
27 28 29
36 37 38

 3  4  5
12 13 14
21 22 23
30 31 32
39 40 41

 6  7  8
15 16 17
24 25 26
33 34 35
42 43 44
```

Looking at the fill order, we can see that the result fills cell 0, row 0, cell 1, row 0, cell 2, row 0, cell 0, row 1 etc, instead of following the axis order, which would have looked like

```
3 5 3⍴data
```

```
 0  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 32
33 34 35
36 37 38
39 40 41
42 43 44
```

We know that we need to end up with the shape 5 3 3, and that we want to grab triplets in order from the data. So we first need to reshape the data to give us triplets in fill order as the rows its major cells:

```
5 3 3⍴data
```

```
 0  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 32
33 34 35

36 37 38
39 40 41
42 43 44
```

We know the desired shape is 3-5-3, so our only option is:

```
1 0 2⍉5 3 3⍴data
```

```
┌┌─┐
↓↓ 0  1  2
│  9 10 11
│18 19 20
│27 28 29
│36 37 38
│
│ 3  4  5
│12 13 14
│21 22 23
│30 31 32
│39 40 41
│
│ 6  7  8
│15 16 17
│24 25 26
│33 34 35
│42 43 44
└~────────
```

There is an additional hard-won lesson here. Let's look at the shapes:

```
⍴5 3 3⍴data
```

```
┌─┐
5 3 3
└~┘
```

No surprises there, right. So we reorder the first two axes, to get a shape of 3-5-3:

```
⍴1 0 2⍉5 3 3⍴data
```

```
┌─┐
3 5 3
└~┘
```

The last two axes have the same dimensions, so we could equally well use the 2-0-1 order and maintain the 3-5-3 shape, right? Wrong:

```
⍴2 0 1⍉5 3 3⍴data ⍝ SURPRISE!
```

```
┌─┐
3 3 5
└~┘
```

This goes to the heart of the matter why dyadic *Transpose* is difficult to grasp. Morten Kromberg offered the following explanation on the APL Orchard:

> Many people are confused by dyadic *Transpose* because the "intuitive" interpretation of the left argument is that it gives the order that you want to select dimensions of the right argument for the result, when in fact it gives the NEW position of each of the dimensions (he says nervously, hoping he got that right after 40 years of practice). –*Morten Kromberg*

In other words, the left argument 2 0 1 does *not* mean "take the old axis 2 and make that the new axis 0, take the old axis 0 and make that the new axis 1 and finally, take the old axis 1 and make that the new axis 2".

Instead it means:

- The old axis 0 is now axis 2
- The old axis 1 is now axis 0
- The old axis 2 is now axis 1

Perhaps this chapter should have started, rather than ended, with this.

Encode decode: ⊤⊥

> Teaching peers is one of the best ways to develop mastery. –*Jeff Atwood*

Here's some of APL's secret sauce, not commonly encountered in other languages: *Encode*, ⊤ and *Decode*, ⊥. *Encode* and *Decode* provide efficient basis conversion across potentially mixed radixes.

Other resources:

- Dyalog docs *Encode*, *Decode*
- Cultivations Lesson 6, Lesson 37, Lesson 38
- APL Wiki *Encode*, *Decode*

```
⎕IO ← 0 ⍝ You know the drill
```

The canonical example is to convert numbers between bases, for example, converting a base-10 number to 8-bit binary:

```
(8⍴2)⊤54
```

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

…and back to base-10:

```
2⊥0 0 1 1 0 1 1 0
```

```
54
```

I vowed not to mention magic inverses, but these few are too damned useful to leave out. Convert a base-10 number to binary, using the least number of bits:

```
2∘⊥⍣¯1 ⊢ 54
```

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

…and as a consequence, split a number into its constituent digits:

```
10∘⊥⍣¯1 ⊢ 677398723 ⍝ Number to digit vector
```

```
6 7 7 3 9 8 7 2 3
```

Those were all fixed radix. An example of *mixed* radix is converting between seconds and days, hours, mins and seconds, e.g "how many days, hours, mins and seconds is 10000 seconds"?

```
1 24 60 60⊤10000
```

```
0 2 46 40
```

and, conversely, how many seconds in 1 day (and night)?

```
1 24 60 60⊥1 0 0 0
```

```
86400
```

Here's an example of *Decode* and *Encode*, borrowed from Mathematica's documentation for its corresponding *MixedRadix* function.

A Roman legion was made of 10 *cohorts*, a cohort of 6 *centuries*, a century of 10 *contuberniae*, and a contubernia of 8 soldiers.

```
units ← 'legion' 'cohort' 'century' 'contubernia' 'soldier'
bases ← 10 10 6 10 8
```

Given 16,894 soldiers, how are they organized?

```
↑units (bases⊤16894)
```

legion	cohort	century	contubernia	soldier
3	5	1	1	6

Going the other way, how many soldiers are there in a legion?

```
bases⊥1 0 0 0 0
```

```
4800
```

There are some less obvious uses, too. For example, we can use 1⊥ to sum vectors:

```
1⊥⍳10
```

```
45
```

The utility of that may not be obvious, but it comes very handy when writing tacit code which we'll cover in depth later, but here's a taster – given a vector where all elements are the same, bar one, what's the index of the "odd one out"?

```
(1⍳⍨1⊥∘.≠⍨) 243 243 243 243 251 243 243 ⍝ Index of "odd one out"
```

```
4
```

The enigmatic `⊥⍨` counts trailing 1s:

```
⊥⍨0 1 0 0 1 1 0 1 1 1 1 1
```

```
5
```

We can also use *Encode* and *Decode* between indices of an array and its ravel:

```
⎕ ← m ← 3 5 ⍴15?15
⎕ ← rav ← ,m
```

```
 3 0 5 12  4
 9 8 6  2 10
11 7 1 14 13
3 0 5 12 4 9 8 6 2 10 11 7 1 14 13
```

Given `index ← 1 4` into the above array, what's the corresponding index into the ravel vector?

```
index ← 1 4
⎕ ← k ← (⍴m)⊥index ⍝ 2D to 1D
m[⊂index]
k⊃rav
```

```
9
10
10
```

And the reverse:

```
⎕ ← i j←(⍴m)⊤k ⍝ 1D to 2D
```

```
1 4
```

Products

> There is no point in being precise if you do not even know what you are talking about. –*John von Neumann*

```
⎕IO ← 0
]box on
]rows on
```

We'll talk about outer and inner products in APL, two powerful features. Other resources on these topics:

- Mastering Dyalog APL has an excellent section on products, from around page 387.
- Dyalog docs on outer product and inner product
- Marshall Lochbaum's talk on outer products

Outer product

Let's look at APL's dyadic *Outer product* operator `∘.` - and by "product" in this case we don't mean multiplication, but as in outer product in the linear algebra sense. The canonical example of an outer product is to build a "times table":

```
∘.×⍨⍳10
```

```
 0  0  0  0  0  0  0  0  0  0
 0  1  2  3  4  5  6  7  8  9
 0  2  4  6  8 10 12 14 16 18
 0  3  6  9 12 15 18 21 24 27
 0  4  8 12 16 20 24 28 32 36
 0  5 10 15 20 25 30 35 40 45
 0  6 12 18 24 30 36 42 48 54
 0  7 14 21 28 35 42 49 56 63
 0  8 16 24 32 40 48 56 64 72
 0  9 18 27 36 45 54 63 72 81
```

To all intents and purposes, that's a nested for-loop which we could write out in Python along the lines of:

```python
for j in range(10):
    for i in range(10):
        print(f"{i*j} ", end="")
    print()
```

apart from the fact that in APL, outer products are vectorised and operate on the whole arrays.

The right side operand can be any dyadic function, including user-defined ones. For example:

```
∘.<⍨⍳10
```

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

The eagle-eyed reader might interject that we can achieve the same thing with *Rank*:

```
(⍳10) <⍤0 1 ⊢ ⍳10
```

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

If you realised that unprompted: well done – this is a deep insight.

> **Tip**
>
> Outer product can be defined in terms of Rank:
>
> ```
> ∘.f → f⍨⍤0 99
> ```

The performance characteristics will differ though, for example

```
prod ← ∘.×
rank ← ×⍤0 1
x←⍳1000
]runtime -c "x prod x" "x rank x"
```

```
x prod x → 5.0E¯4 |   0% ████████████████████████████
x rank x → 7.7E¯4 | +55% ████████████████████████████████████████████
```

Here's an expression using an outer product that picks out the prime numbers between 0 and 20. Can you figure out how it works?

```
(2=+⌿0=x∘.|x)/x←⍳20
```

```
2 3 5 7 11 13 17 19
```

Inner product

The *Inner product*, `f.g` is perhaps less obvious. An example of an inner product is actual matrix multiplication:

```
]DISPLAY A ← 3 4⍴3 2 0 8 11 7 5 1 4 9 6 10
]DISPLAY B ← 4 3⍴8 10 11 2 4 6 5 1 7 9 3 0
]DISPLAY A +.× B
```

```
┌→──────┐
↓ 3 2 0  8│
│11 7 5  1│
│ 4 9 6 10│
└~──────┘
```

```
┌→──────┐
↓8 10 11│
│2  4  6│
│5  1  7│
│9  3  0│
└~──────┘
```

```
┌→──────────┐
↓100  62  45│
│136 146 198│
│170 112 140│
└~──────────┘
```

If you know how to multiply matrices, you'll understand what the inner product operator does: consider the top left element of the result, 100. How did we get that? First we multiply each number in the first row of A with the corresponding element in the first col of B:

```
⍝ ← row0col0prod ← A[0;]×B[;0]
```

```
24 4 0 72
```

Then sum it:

```
+/row0col0prod
```

```
100
```

Next element:

```
+/A[0;]×B[;1]
```

```
62
```

We can apply inner product to many common problems where the impulse is to "first apply one function, then reduce another over the result". For example, given two strings of equal lengths, how many letters are the same? You might try first:

```
+/'GATTACA' = 'TATTCAG' ⍝ Equal-then-sum-reduce
```

```
3
```

but this fits the inner product pattern nicely:

```
'GATTACA' +.= 'TATTCAG'
```

```
3
```

Calculating expected offspring

Here's a problem from Project Rosalind, a bioinformatics problem collection, Calculating Expected Offspring.

We're given six integers, corresponding to the number of couples in a population possessing each genotype pairing for a given factor. In order, the six given integers represent the number of couples having the following genotypes:

1. AA-AA
2. AA-Aa
3. AA-aa
4. Aa-Aa
5. Aa-aa

6. aa-aa

If each couple have two offspring, what is the expected number of offspring displaying the dominant phenotype (A) in the next generation?

Let's say we're given the couples distribution as

```
data ← 19083 17341 19657 16896 16197 18256
```

I'm no biologist, but the way this works is that for each of the 6 possible genotype pairings we can see the probability of an offspring displaying the dominant phenotype, which is 1, 1, 1, 0.75, 0.5 and 0 respectively.

```
prob ← 1 1 1 0.75 0.5 0
```

We need to multiply the data with the corresponding probabilities, and sum that up, and finally multiply by 2, as we have two offspring per couple. This can be neatly formulated as an inner product:

```
2×data+.×prob ⍝ Same as 2× +/ data×prob
```

```
153703
```

eXplanation operator

A handy trick when figuring out products is to use the "eXplanation" operator, that Adám Brudzewsky demonstrated in one of the Cultivations:

```
X ← {f←⍺⍺ ⋄ ⍺←⊢ ⋄ '(',⍺,(⎕CR'f'),⍵,')'} ⍝ The product eXplanation operator
```

Using this we can visualise how Dyalog will calculate each element in a product – inner and outer. For example, given

```
]DISPLAY A ← 2 3⍴6?20
]DISPLAY B ← 3 2⍴6?20
]DISPLAY A +.× B
```

```
┌→──────┐
↓ 3  2 19 │
│16 11  4 │
└~──────┘
```

```
┌→──┐
↓18 17 │
│ 9  7 │
│ 3  1 │
└~──┘
```

```
┌→────┐
↓129  84 │
│399 353 │
└~────┘
```

We can show how this was derived with X:

```
A +X.(×X) B
```

((3 × 18)+((2 × 9)+(19 × 3)))	((3 × 17)+((2 × 7)+(19 × 1)))
((16 × 18)+((11 × 9)+(4 × 3)))	((16 × 17)+((11 × 7)+(4 × 1)))

Trainspotting

> Simplicity and elegance are unpopular because they require hard work and discipline to achieve and education to be appreciated. – *Edsger Dijkstra*

Tacit is the *third* way to write code in Dyalog APL, after dfns, and *traditional* (which we're not covering). The tacit style, also sometimes called *point-free*, was taken from the J language, which originated the concept. Tacit code can be made super-terse, and is best reserved for short snippets only. APL *idioms* – short, efficient, and ultra-optimised bits of code, are often expressed in tacit. You'll find that pretty much everything on APLCart is written in tacit, and as you start working on improving your Code Golf handicap, tacit is an essential skill.

We can look at tacit programming in APL as a little embedded DSL for functional composition, complete with its own grammar. The rules of this grammar are actually quite simple, but learning how to read and write tacit functions takes a lot of practice, even for those well-versed in the rest of APL.

References for tacit programming:

- APL Wiki
- Cultivation
- Dyalog webinar
- dfns page on dfn to tacit translation

The word *tacit* means implicit, and refers to a function where there is no explicit mention of its arguments. In a tacit function, then, you'll see no `⍺` and `⍵` as you would in a dfn. Instead a set of rules decide how the components of a tacit function interact with the arguments.

This time, in our prelude, we'll use the `-trains=tree -fns=on` arguments to `]box` which helps with showing the structure of tacit functions.

```
⎕IO ← 0
]box on -style=min -trains=tree -fns=on
]rows on
assert←{⍺←'assertion failure' ⋄ 0∊⍵:⍺ ⎕signal 8 ⋄ shy←0}
```

The tacit rules: atop

A tacit function is wrapped in parentheses, `( ... )`, and is governed by a set of rules. Let's begin with the easiest one, the *monadic atop*. An *atop*, also known as a *2-train*, is a combination of two functions:

```
(f g)Y ← f g Y
```

This states that the tacit function `(f g)`, comprising the monadic functions `f` and `g`, simply corresponds to the sequence `f g` applied to an argument array `Y`. Well, that seems… obvious? We can try it:

```
⍝ ← Y ← 3 3⍴⍳9
(-⌽)Y ⍝ Tacit
-⌽Y   ⍝ Explicit "beside"
```

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

Yes, it's understandable if you're scratching your head, wondering "what's the point of that then?". It pays to think of it as a means of creating a derived function. Doing

```
-⌽Y
```

is *two* operations, whereas

```
(-⌽)Y
```

is the application of a single, derived function.

But the main purpose of the monadic atop rule is to serve as a building block in more complex tacit functions, as we shall discover shortly.

Let's move on to the dyadic version of atop instead:

```
X (f g) Y ← f X g Y
```

Here, the dyadic tacit function `(f g)` combines the *monadic* function `f` and the dyadic function `g`. Let's say we want to tally the elements that are left if you remove all elements from array A from those in B:

```
B ← 1 2 3 4 5 6 7
A ← 3 2 6
≢ B ~ A
```

```
4
```

There we can see the `f X g Y` pattern. Let's try the tacit version, a dyadic *atop*:

```
B (≢~) A
```

```
4
```

There's also a monadic version of the atop rule:

```
(f g)Y → f g Y
```

Again, hard to see the utility yet, perhaps.

The tacit rules: fork

Let's keep going! Here's our first *fork* (also known as a 3-train) – a tacit combination of *three* functions:

```
X (f g h) Y → (X f Y) g (X h Y)
```

This is perhaps the simplest tacit rule that one can see applications for. Consider the three-way compare (sometimes called "spaceship") operator, `a <=> b` that some languages (like Perl or Ruby) offer. It returns a negative value if a < b, 0 if a == b and a positive value if a > b. APL has no such thing, but we can cook up a tacit function using the fork rule:

```
cmp ← >-<
```

```
1 cmp 2
1 cmp 1
2 cmp 1
```

```
¯1
0
1
```

Let's look at the rule application to see why it works:

```
(a > b) - (a < b) ⍝ using the fork rule: (X f Y) g (X h Y)
```

Once you learn to see this pattern, it really crops up frequently. Why is it called a "fork"? Well, if we have specified `]box on` with `-trains=tree` we can ask the interpreter to show us its parse of the tacit function:

```
cmp
```

```
┌┼┐
> - <
```

Very much a fork, or trident. As an aside, you can actually achieve the same thing with one symbol fewer:

```
cmp ← ×- ⍝ Try it!
```

Let's look at a few more examples. A common problem is to split a string into substrings, dividing on some separator character. Here's one way you can achieve that, using a dfn:

```
Split ← {ω⊆⍨~α≠ω}
```

```
' ' Split 'A common problem is to split a string into substrings'
```

A	common	problem	is	to	split	a	string	into	substrings

Let's rework that into a tacit formulation instead, taking it step by step. First, let's flip that selfie and put in some parentheses for emphasis:

```
Split ← {(α≠ω) ⊆ (ω)}
```

So that *almost* matches the fork pattern `(X f Y) g (X h Y)`, but not quite. The last part makes no reference to `α`. Let's address that with a tack-trick:

```
Split ← {(α≠ω) ⊆ (α⊢ω)}
```

To recap, the tacks (`⊣⊢`) are functions that just return the value they're "pointing" to:

```
2⊢3
⊢5
2⊣3
```

```
3
5
2
```

They are very useful when writing tacit functions, as we've just seen. We now have a dfn body that matches the fork rule, and we can write it tacitly as:

```
Split ← ≠⊆⊢
```

```
' ' Split 'It should still work the same way'
```

It	should	still	work	the	same	way

Ok, let's try another one, this time going the other way. Here's a fork that scales a vector of numbers such that its components sum to 1:

```
UnitSum ← ⊢÷+/
```

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

```
0.047619 0.0952381 0.142857 0.190476 0.238095 0.285714
```

What is the corresponding explicit dfn?

The complication here is that we now have an operator involved: a reduction. Recall that operators take one (or two) function(s) and return a *derived function* that can then be applied to arguments in turn. So let's think of the sum-reduction as a single function. Using spaces to make this clearer we get:

```
UnitSum ← ⊢ ÷ +/
```

We know already that the fork itself is monadic, meaning that the `f` and the `h` functions must both be monadic, and the `g` function dyadic. The tack becomes just `ω` and the sum-reduction we just need to give an argument:

```
UnitSum ← {ω÷+/ω}
```

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

```
0.047619 0.0952381 0.142857 0.190476 0.238095 0.285714
```

That last example – a monadic fork – can be formalised as:

```
(f g h)Y → (f Y) g (h Y)
```

There are, in fact, two more fork-varieties, beyond the two we've already seen. They differ in how the consitutent functions are applied (monadically or dyadically):

```
X(Z g h)Y ← Z g X h Y ⍝ h dyad, g dyad, derived dyad
(X g h)Y  ← X g h Y   ⍝ h modad, g dyad, derived monad
```

Summary: forks and atops

For completeness, then – here are all the rules of the tacit grammar in one place:

Tacit	Type	Explicit
`X(Z g h)Y`	Fork	`Z g X h Y`
`X(f g h)Y`	Fork	`(X f Y) g (X h Y)`
`(X g h)Y`	Fork	`X g h Y`
`(f g h)Y`	Fork	`(f Y) g (h Y)`
`X(f g)Y`	Atop	`f X g Y`
`(f g)Y`	Atop	`f g Y`

Forks atop forks

This is where the real fun begins. We can string together longer tacit functions by combining forks and atops. Some consideration should be given to comprehensibility here. Long stretches of tacit code requires more effort to understand than the corresponding explicit formulation.

As our tacit rules indicate, barring the presence of parentheses, a combination of three functions is a fork, and two functions is an atop. If we have *more* than three functions, we start reading from the right, making groups of threes and twos, and combine *those* using the fork and atop rules.

From Dyalog's docs:

> ...in the absence of parentheses, a sequence of an odd number of functions resolves to a 3-train (fork) and an even-numbered sequence resolves to a 2-train (atop)

So how long is too long a train? It depends on the "carriages". The example given in Dyalog's docs above is a good point:

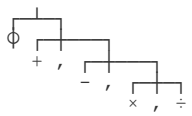
```
6 (⌽+,-,×,÷) 2
```

```
3 12 4 8
```

That's 8 constituent functions. We'll prise it apart in a bit, but looking at it, it's pretty clear what the intention is: find the sum, difference, product and ratio of two numbers, and reverse the list.

As always, let's look at the parse:

```
⌽+,-,×,÷
```



So reading right to left, down to up, we have three forks and an atop. Let's resolve them from the right:

```
f1 ← ×,÷ ⍝ ...fork, (6×2),(6÷2)
```

```
□ ← r ← 6 (⌽+,-,f1) 2 ⍝ does it still work?
assert r≡6 (⌽+,-,×,÷) 2
```

```
3 12 4 8
```

The next fork is now

```
f2 ← -,f1 ⍝ ...fork: (6-2),f1 → (6-2),(6×2),(6÷2)
```

```
6 (⍛+,f2) 2
```

```
3 12 4 8
```

and the final fork is more of the same:

```
f3 ← +,f2 ⍝ (6+2),(6-2),(6×2),(6÷2)
```

```
6 (⍛f3) 2
```

```
3 12 4 8
```

And finally, the atop, which moves the left argument to between the functions and removes the brackets:

```
⍛ 6 f3 2
```

```
3 12 4 8
```

or, fully explicit:

```
6 {⍛(⍺+⍵),(⍺-⍵),(⍺×⍵),⍺÷⍵} 2
```

```
3 12 4 8
```

Complicating factors

Using the method outlined above, you can usually untangle reasonably-sized trains that others have composed:

- Use `]box on -style=mid -trains=tree -fns=on` to visualise the parse tree for the train
- Follow the pattern indicated by the tree, and resolve forks and atops (R→L) using The Rules

However, certain factors make this more complicated:

- Mid-train parentheses to alter order of precedence
- Binding strengths – operators bind tighter than functions
- Jots & tacks, if overused.

A handy tacit function is finding the min and max of an array, which we can use to demonstrate the operator binding:

```
minmax ← ⌊/,⌈/
minmax 7 2 3 8 0 9 12
```

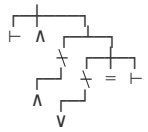
```
0 12
```

This is a single fork:

```
minmax
```

So we can see that we need to start by letting the reduces bind first to form the derived functions that make up the fork. Ok, that wasn't too bad. What about this one?

```
□ ← ones ← ⊢∧(∧⍀∨⍀=⊢) ⍝ Hat tip to Adám Brudzewski for the suggestion
```



Top marks if you can say what that even does. Before we get into that, we can see that the parenthesis have affected the "groups of three from the right" parse. We're now looking at a fork-atop-fork, two operators and two tacks. All non-specific religious holidays came at once.

This function preserves the first uninterrupted sequence of 1s in a vector:

```
ones 0 0 1 1 1 0 0 0 1 0 1 1 1 0
```

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

Let's untangle the parse tree, bottom-up, right-left. The first fork is

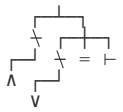
```
□ ← f1 ← ∨⍀ = ⊢ ⍝ ...monadic fork, giving a dfn equivalent: {(∨⍀ω)=ω}
```

```
□ ← r ← (⊢∧(∧⍀f1)) 0 0 1 1 1 0 0 0 1 0 1 1 1 0
assert r≡ones 0 0 1 1 1 0 0 0 1 0 1 1 1 0 ⍝ still works!
```

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

Now let's tackle the atop, which is just applying the and-scan to the fork:

```
□ ← a1 ← ∧⍀f1
```



```
(⊢∧a1) 0 0 1 1 1 0 0 0 1 0 1 1 1 0
```

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

and there we have a monadic fork:

```
{ω∧a1 ω} 0 0 1 1 1 0 0 0 1 0 1 1 1 0
```

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

We can now fully untangle it if we want:

```
{ω∧∧⍀(∨⍀ω)=ω} 0 0 1 1 1 0 0 0 1 0 1 1 1 0
```

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

So now we can compare the tacit and explicit forms of the same function:

```
tacit ← ⊢∧(∧⍀∨⍀=⊢)
expl  ← {ω∧∧⍀(∨⍀ω)=ω}
```

Which one is "better"? Yes, the tacit formulation is shorter by three glyphs. *Readability* is a slippery concept which depends on the skill and experience of the reader, but at least to *this* reader, without going through the above deconstruction exercise, I could not have told you how the tacit function worked. In the explicit formulation – to *my* eyes at least – the algorithm is more visible.

Let's work through a couple more, back and forth. If you haven't already, do check out Richard Park's most excellent webinar on the topic, too.

> **Note**
>
> Working through examples this way, whilst somewhat tedious, is the only way to learn. Eventually it clicks. If this all feels a bit dry, skip to the next chapter and keep coming back here in small doses.

Here's a tacit phrase which groups pairs of elements based on the first:

```
group ← ↓⊃⍤,∘⊂⌸⊢/⍤ ⍝ Group pairs based on first element
group □←(5 3)(5 6)(7 5)(4 7)(1 8)(2 4)(1 2)
```

```
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
│5 3  │5 6  │7 5  │4 7  │1 8  │2 4  │1 2  │     │     │     │     │     │
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘
┌─────────┬─────┬─────┬─────────┬─────┐
│┌─────┐  │┌─┐  │┌─┐  │┌─────┐  │┌─┐  │
│└─────┘  │└─┘  │└─┘  │└─────┘  │└─┘  │
└─────────┴─────┴─────┴─────────┴─────┘
```

Let's look at the parse tree:

```
group
```

```
↓
 ∪  ∘  ⊢/
 ,  ⊂
```

Yikes… but actually, it looks worse than it is – it's a single fork, with an atop. The sprinkling of operators make it look more complicated than it is. So, bit by bit as indicated by the tree, starting with the *Key* (⌸):

```
key ← ⍺←,∘⊂⌸
```

```
 ⌸
 ∘
 ,  ⊂
```

Looks correct. If we substitute into the original definition, we get

```
↓⊃¨key⊢/¨
```

```
↓
 ∪  ∘  ⊢/
 ,  ⊂
```

Yep. Still the same. Now for the left and right tines:

```
discloseAll ← ⊃¨
mergeLasts ← ⊢/¨
↓discloseAll key mergeLasts
```

```
↓
 ∪  ∘  ⊢/
 ,  ⊂
```

We can now translate to a dfn:

```
{↓(discloseAll ⍵) key (mergeLasts ⍵)}  (5 3)(5 6)(7 5)(4 7)(1 8)(2 4)(1 2)
```

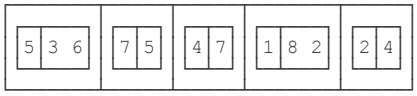
```
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
│5 3  │5 6  │7 5  │4 7  │1 8  │2 4  │1 2  │     │     │     │     │     │
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘
┌─────────┬─────┬─────┬─────────┬─────┐
│┌─────┐  │┌─┐  │┌─┐  │┌─────┐  │┌─┐  │
│└─────┘  │└─┘  │└─┘  │└─────┘  │└─┘  │
└─────────┴─────┴─────┴─────────┴─────┘
```

```
{↓(⊃¨⍵) ,∘⊂⌸ ⊢/¨⍵} (5 3)(5 6)(7 5)(4 7)(1 8)(2 4)(1 2)
```

```
┌─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┬─────┐
│5 3  │5 6  │7 5  │4 7  │1 8  │2 4  │1 2  │     │     │     │     │     │
└─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┴─────┘
┌─────────┬─────┬─────┬─────────┬─────┐
│┌─────┐  │┌─┐  │┌─┐  │┌─────┐  │┌─┐  │
│└─────┘  │└─┘  │└─┘  │└─────┘  │└─┘  │
└─────────┴─────┴─────┴─────────┴─────┘
```

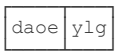
The operand to *Key* is a derived function. We could expand that, too, if we want to be purist:

```
{↓(⊃¨⍵) {⍺,⊂⍵}⌸ ⊢/¨⍵} (5 3)(5 6)(7 5)(4 7)(1 8)(2 4)(1 2)
```



Here's another one. What does this even do?

```
2 (|∘⌽∘≢⊢∘⊂⌸⊢) 'dyaloge'
```



Another thing we can do that may or may not be useful is to ask the interpreter to give us a parethesised expression instead of the parse tree.

```
]box on -style=mid -trains=parens -fns=on
```

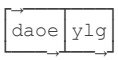
```
Was ON -style=min -trains=tree -fns=on
```

```
|∘⌽∘≢⊢∘⊂⌸⊢
```

```
((|∘⌽)∘≡/)((⊢∘⊂)⌸)⊢
```

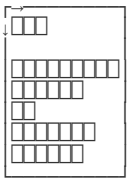
Some say that's an easier start than the tree. Take your pick. Following the steps above you should land on:

```
2 {(⍺|⌽≠⍵) {⊂⍵}⌸ ⍵} 'dyaloge'
```



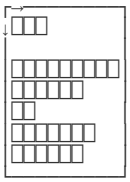
This one draws a bar chart:

```
]DISPLAY (↑⍴¨∘'□')3 0 9 6 2 7 6
```



which is just

```
]DISPLAY ↑{⍵⍴'□'}¨3 0 9 6 2 7 6
```



Let's try a couple of harder ones. This one removes leading, trailing and repeated stretches of ⍺ from ⍵:

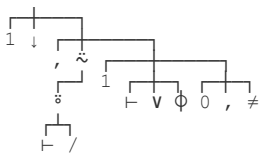
```
trim ← 1↓,⊢⍱⍀/≁1(⊢v⌽)0,≠
' ' trim '        hello     world     '
```

```
hello world
```

```
]box on -style=mid -trains=tree -fns=on ⍝ I'm a tree guy!
```

```
Was ON -style=mid -trains=parens -fns=on
```

```
trim
```



No fewer than *five* forks! The other thing to note is the use of ⍤ here which in fact is *not Rank*. When ⍤ is used with a right *function* operand, it becomes *Atop*, not *Rank* (which takes a right *array* operand).

The tacit phrase ⊢⍤f is a trick to force f to be parsed as a function, rather than as an operator for glyphs that can be either. In our case, ⊢⍤/⍨ is just {ω/α}. With that we may be able to skip a few intermediate steps:

```
f ← {1{ω∨α⍱ω}α{0,α≠ω}ω}
g ← {ω/α}
trimdfn ← {1↓(α,ω)gαfω}
```

```
' ' trimdfn '        hello     world     '
```

```
hello world
```

Expanding that out we can make a few simplifications, arriving at the resaonably compact

```
trim ← {1↓(x∨1⍱x←0,α≠ω)/α,ω}
```

```
' ' trim '        hello     world     '
```

```
hello world
```

How about going the other way? We've done some already. A pretty comprehensive guide to the mechanical process of translating a dfn to its tacit equivalent can be found in the docs for the dfns workspace, mentioned at the top of this chapter.

Here's a noddy dfn that takes a string consisting of a leading letter and some digits, and returns a 2-element vector with the letter and the number:

```
{(1↑ω),⍎1↓ω} 'X1234'
```

```
X 1234
```

There's clearly a fork in there, but also an atop. This should pose little difficulty, right? Change the braces for parantheses, and omegas for right-tack, but we need to delineate the eval call:

```
((1↑⊢),(⍎(1↓⊢))) 'X1234'
```

```
X 1234
```

We can do more here. Both left and right tines of the fork share a 1. We can make that a left argument, so in our case a *left* tack:

```
1 ((⊣↑⊢),(⍎(⊣↓⊢))) 'X1234'
```

```
X 1234
```

but the pattern ⊣f⊢ is just f, which also lets us remove some parentheses:

```
1 (↑,(⍎↓)) 'X1234'
```

```
X 1234
```

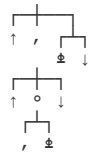
We can actually remove the inner parantheses, too, by a bit of sleight of hand:

```
1 (↑,∘⍎↓) 'X1234'
```

```
X 1234
```

but these two trains – whilst doing the same thing – parse differently:

```
↑,(⍎↓)
↑,∘⍎↓
```



In other words, we made a derived function with jot: `,∘⍤`. That turned out well: the tacit version is shorter than the explicit, and no less readable.

A few more?

This dfn takes a rational to the right and tries to compute a vector representing the corresponding fraction:

```
{(1∧⍵)÷1,⍵} 0.75
```

```
3 4
```

A fork, with both tines also being forks.

```
f1 ← {1∧⍵}
f2 ← {1,⍵}
{(f1 ⍵) ÷ f2 ⍵} 0.75
```

```
3 4
```

We can convert the tine forks directly by swapping `⍵` for `⊢` and `{}` for `()`:

```
((1∧⊢)÷1,⊢) 0.75
```

```
3 4
```

This one shifts a vector of numbers the specified number of steps to the left, padding with zeros on the right:

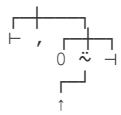
```
3 {⍺↓⍵,⍺↑0} ⍳10
```

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

Looks easy, right? Tempting to to jump to the conclusion that this is a fork on *Catenate*, `f,g` – it's not.

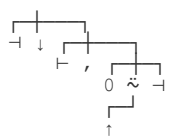
The complication is the take at the end, which has an array right argument which isn't allowed in a train: we need to *Commute* that one. So starting from the right, we have a dyadic function as `⍺↑0`, preceded by `⍵,`. So that's our first fork of the form `⊢,f`:

```
⊢,0↑⍨⊣
```



Before that we have `⍺↑` which together with what we already did also makes a fork, this time `⊣↑g`:

```
 ← shift ← ⊣↓⊢,0↑⍨⊣
```

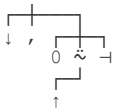


```
3 shift ⍳10
```

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

Note that again it's tempting to say that `⊣f⊢` is just `f` like we did earlier, but that makes the same slip as we started with – they don't belong to one fork:

```
↓,0↑⍨⊣
```



Shuffle the array

The LeetCode problem 1470 was posed as a chat mini challenge on APL Orchard to produce a tacit solution, and some elegant solutions were presented. Let's look at that problem.

The task is to take a vector of length 2n, and essentially zip the first half with the second, and flatten:

```
Input: nums = [1,2,3,4,4,3,2,1], n = 4
Output: [1,4,2,3,3,2,4,1]
```

A dfn formulation is pretty straight-forward:

```
4 {,⍉2⍴⍵} 1 2 3 4 4 3 2 1
```

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

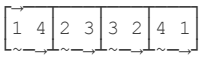
which is reshape to `2 n`, *Transpose*, *Ravel* – in fact, it's kind of how we said it: "zip the first half with the second, and flatten". However, that particular dfn doesn't lend itself nicely to a tacit formulation. APL Orchard user @rak1507 proposed the following gem:

```
4 (∊↑,¨↓) 1 2 3 4 4 3 2 1
```

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

Remarkably clever, but kinda obvious (once you've seen it). This relies on *Catenate each* `,¨` as the (almost) "zip" instead of `,⍉`:

```
1 2 3 4 ,¨ 4 3 2 1
```

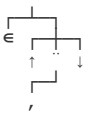


The actual formulation is the tacit version of

```
4 {∊(⍺↑⍵),¨(⍺↓⍵)} 1 2 3 4 4 3 2 1
```

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

```
(∊↑,¨↓)
```



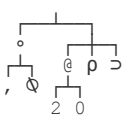
Nice. But wait, there's more. Adám's version takes a wholly different approach, and do note the way the function takes its arguments, which is key to how this works:

```
(,∘⍉2@0⍴⊃) (1 2 3 4 4 3 2 1) 4
```

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

Not obvious how that works, right? Let's look at the parse tree:

```
,∘⍉2@0⍴⊃
```



The interesting bit is the fork `2@0⍴⊃`, so let's look at that:

```
2@0⍴⊃
```

```
  ┌┼┐
  @ ⍴ ⊃
 ┌┘
 2 0
```

```
(2@0⍴⊃) (1 2 3 4 4 3 2 1) 4
```

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

Aha. Yes, there's the "fold in half" – but how *does that work*? The middle tine is clearly the reshape, and the right tine picks the first element, which makes sense, but what of the left tine?

```
2@0 ⊢ (1 2 3 4 4 3 2 1) 4
```

```
2 4
```

Wow. Did the penny drop? I think that qualifies as a code-golfer's trick shot.

Other suggestions were:

```
((⍋≠⍴(⍳.5×≠))⊃¨⊂) 1 2 3 4 4 3 2 1 ⍝ Author: @Razetime -- no need for the length
(⊢⌷⍨∘⊂∘⍋∘⍋2|⍳∘≠)  1 2 3 4 4 3 2 1 ⍝ Author: @rak1507  -- no need for the length
```

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

Feel free to untangle those at your leisure.

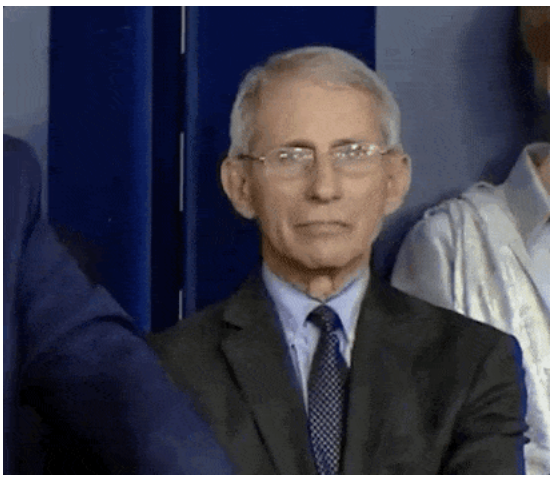
Finding things

> Computer languages of the future will be more concerned with goals and less with procedures specified by the programmer. –*Marvin Minsky*

Just like with indexing, there seems to be a baffling array of ways to locate items in arrays in APL. As stated in The Zen of Python:

> There should be one– and preferably only one –obvious way to do it.

Aha. About that…



```
⎕IO ← 0
]box on
]rows on
```

Equality =

So how do you locate where an element resides in an array in APL? Well, the obvious way to do it is to exploit equality and scalar extension. Where is the number 2?

```
]DISPLAY ↑(2=data)(data ← 4 1 22 20 16 10 25 7 18 11 15 2 12 23 9 17 6 14 21 19 3 8 5
13 24)
```

```
┌→──────────────────────────────────────────────────────────────────────────────┐
↓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             │
│4 1 22 20 16 10 25 7 18 11 15 2 12 23 9 17 6 14 21 19 3 8 5 13 24             │
└~──────────────────────────────────────────────────────────────────────────────┘
```

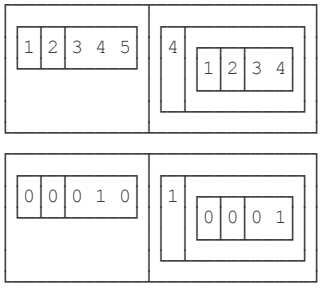
That's surely sufficiently *Zen* in the *Zen of Python* sense. Hands up every item that equals 2. We can find out the actual index, too, by using *Iota underbar*, ⍳, which in its monadic form is appropriately enough called *Where*:

```
⍳2=data ⍝ Where is data = 2?
```

```
11
```

As we should expect by now, scalar extension lets scalar functions penetrate any level of nesting:

```
⎕ ← nested ← (1 2 (3 4 5))(4 (1 2(3 4)))
4=nested
```



Match ≡

Match we've already met. It is like equality but for non-scalar things. You can be forgiven to think that if you look for a particular vector in a vector-of-vectors, you should be able to use equality, but that doesn't work:

```
strings ← 'aaa' 'bbb' 'ccc' 'ddd' 'ccc' 'aaa' 'ccc' 'bbb'
'ccc'=strings  ⍝ LENGTH ERROR!
```

```
LENGTH ERROR
      'ccc'=strings  ⍝ LENGTH ERROR!
           ∧
```

Instead we need either *Jot match each* (∘≡¨) or *Enclose match each*:

```
⍳'ccc'∘≡¨strings
⍳(⊂'ccc')≡¨strings
```

```
2 4 6
2 4 6
```

Index of A⍳B

Another way we can find the index of things is via *Index of*, dyadic ⍳, which we have actually met before:

```
data⍳2
```

```
11
```

Dyadic *iota* finds the *first* index - or 1 plus the last index if not found. This has the nice feature that we can feed it an array to the right:

```
data⍳2 3
```

```
11 20
```

…which, if we want to use equality requires a bit more dexterity – but this approach would also spot multiples:

```
⍸v≠2 3 ∘.= data
```

```
11 20
```

What about if we crank the rank a bit?

```
□ ← mat ← 4 4⍴2 15 16 14 11 9 12 10 13 1 7 5 4 8 6 3
```

```
 2 15 16 14
11  9 12 10
13  1  7  5
 4  8  6  3
```

```
□ ← mask ← 7=mat
⍸mask
```

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

```
┌→┐
↓2 2│
└~┘
```

Gotta love APL.

Find ⍷

There is of course also a glyph that's called *Find* (⍷) which locates the start-points of subsequences:

```
'ana' ⍷ 'banana'
```

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

That combines nicely with ⍸ as a tacit atop to give a bit of an idiom to commit to memory:

```
substr ← ⍸⍷
'ana' substr 'banana'
```

```
1 3
```

We can use find to locate arrays-in-arrays of higher ranks, too:

```
]DISPLAY needle ← 2 2⍴0 1 1 0
]DISPLAY haystack ← 4 4⍴0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0
```

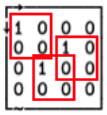
```
┌→┐
↓0 1│
│1 0│
└~┘
```

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

```
needle ⍷ haystack
```

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

The 1s represent the top-left corners of the 'needle' array in 'haystack':



and we can pick out the actual coordinates using the same idiom we used for `substring` above:

```
needle (⍳⍷) haystack
```

0 0	1 2	2 1

Regular expessions

Dyalog supports regular expressions through the most excellent PCRE engine. Dyalog's docs on the topic can be found here, and an APL Orchard cultivation was dedicated to the topic, too. How regexes themselves work is beyond the scope of this book, but an exceptional reference that belongs on every programmer's bookshelf is Jeffrey Friedl's Mastering Regular Expressions.

Let's take a brief look at how regexes are integrated in Dyalog.

Two system operators, ⎕S and ⎕R, implement regex search and replace respectively. They're both dyadic operators, taking regular expression(s) to the left and transformation(s) to the right. The derived function can be applied to text data.

Here's a simple example. Using a transformation string of &, ⎕S returns a vector of what was matched:

```
'll.\sw'⎕S'&' ⊢ 'hello world well  worn'
```

llo w	ll w

The left operand can be a nested vector of regexes:

```
'he' 'wo' 'll'⎕S'&' ⊢ 'hello world well  worn'
```

he	ll	wo	ll	wo

The right operand can be a function, too. This is where it gets a bit ~~hairy~~ flexible. We could have written the above as:

```
'he' 'wo' 'll'⎕S{⍵.Match}'hello world well  worn'
```

he	ll	wo	ll	wo

In other words, the right operand function gets passed a *namespace* representing the match at that point. In this case, the `⍵.Match` holds "what the current regex matched" - the same as the magic transformation short-hand `'&'`.

For capture groups, we have numbered references `'\1', '\2'` etc, as per Perl:

```
'(a{2,}|b{2,})'⎕S'\1' ⊢'aaababababbbbaaaa'
```

aaa	bbbbb	aaaa

When using a function operand, we have the opportunity to apply the full might of APL to the matches. Let's find stretches of 2 or longer of a or b, and turn them to upper-case (using *Case convert*, ⎕C):

```
'(a{2,}|b{2,})'⎕S{1⎕C ⍵.Match}'aaababababbbbaaaa'
```

AAA	BBBBB	AAAA

The regex replacement operator, ⎕R, operates much in the same way, but here the right operand represents a substitution into the right argument:

```
'a(.)a'⎕R'A\1A'⊢'abababadabaaba' ⍝ \1 is the first capture group
```

```
AbAbAbAdAbAAbA
```

A handy trick is to split strings based on a regular expression, removing the separators in the process. This is tricker than you might guess in Dyalog. Here's what APL Cart suggests:

```
rsplit ← {(⊢/¨r)↓¨ω⊂⍨(⍳≠ω)∈1+⊃¨r←(⍺,'|^')⎕S 0 1⊢ω} ⍝ APL Cart
```

```
'\d+' rsplit 'aaaa6666bbb1cccc999eee87'
```

aaa6	bb1	ccc9	ee8	

Note the empty segment at the end.

Overlapping matches

In normal operations, a regex "consumes" what it matches, and any susbequent matches will start where the previous one ended. For example, if we want to capture pairs of letters starting with an a:

```
'a.'⎕S'&'⊢'abaac' ⍝ Won't return ab aa ac
```

ab	aa

which won't capture the last pair ac as it overlaps with the previous match aa. If we want to capture potentially overlapping matches, we have two options. Option 1 is the time-honoured technique borrowed from Perl, using a capture group inside a *zero-width lookahead assertion*:

```
'(?=(a.))'⎕S'\1'⊢'abaac'
```

ab	aa	ac

Zero-width lookaheads (and lookbehinds) work just like normal patterns, except that they don't *consume* what they match.

Option 2 is to tell the regex engine that we want to allow overlapping matches via a *Variant* (⍠) setting to ⎕S:

```
'a.'⎕S'&'⍠'OM'1⊢'abaac'
```

ab	aa	ac

This is both elegant and clear: variant OM is *Overlapping Matches*. See the docs for more details on the various options that can be enabled with *Variant*.

Partitions

> It's harder to read code than to write it. –*Joel Spolsky*

```
]box on
```

We often need to group or select items in an array based on some criterion – perhaps stretches of items that are equal, or perhaps we have a boolean mask indicating which stretches of elements should be joined up.

Other resources:

- Dyalog docs Partition, Partitioned enclose
- APLWiki
- Cultivations Lesson 7, Lesson 49

> **Note**
>
> This is a feature that different APL dialects treat differently. Dyalog's ⎕ML setting allows a degree of conformity if you need it: if ⎕ML≥3, the symbol ⊂ means the same as ⊆. We'll use the default Dyalog approach throughout.

Partition ⊆

The glyph *Partition*, ⊆, selects groups elements based on a vector where new selections are started where the corresponding element is larger than its predecessor. In its simplest form it's like *Compress*, but enclosing elements corresponding to stretches of 1s. Compare:

```
1 1 0 1 0 0 0 0 1 1 0 1⊆⍳12 ⍝ Partition
1 1 0 1 0 0 0 0 1 1 0 1/⍳12 ⍝ Compress
```

```
┌─────┬─┬─────┬──┐
│1 2 4│9│10 12│  │
└─────┴─┴─────┴──┘
1 2 4 9 10 12
```

This can be supremely handy: apply some predicate to an array to give a boolean vector. Use enclose to get the matching elements, grouped. As we noted above, the left argument array doesn't have to be Boolean, just integer. That gives us a bit more flexibility in our mapping.

Partitioned enclose A⊂B

Partitioned enclose, A⊂B, groups items based on a Boolean mask where enclosures start on 1. For example:

```
1 0 1 0 1 0 1 0 1 0⊂⍳10
```

```
┌───┬───┬───┬───┬─────┐
│1 2│3 4│5 6│7 8│9 10 │
└───┴───┴───┴───┴─────┘
```

We can use this to group stretches of elements which are equal:

```
{ω⊂⍨1,2≠/ω} 1 1 1 1 1 2 2 1 4 4 4 4 1 1 2 2
```

```
┌─────────┬───┬─────────┬───┬───┐
│1 1 1 1 1│2 2│1 4 4 4 4│1 1│2 2│
└─────────┴───┴─────────┴───┴───┘
```

We can apply the same technique to group based on sign:

```
{ω⊂⍨1,2≠/×ω} 0 0 1 2 3 0 ¯1 ¯7 ¯8 0 0 1 2 3
```

```
┌───┬─────┬─┬──────┬───┬─────┐
│0 0│1 2 3│0│¯1 ¯7 ¯8│0 0│1 2 3│
└───┴─────┴─┴──────┴───┴─────┘
```

In the above two examples we generate the mask by a windowed reduction of size 2:

```
{2≠/×ω} 0 0 1 2 3 0 ¯1 ¯7 ¯8 0 0 1 2 3
```

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

but we also need to prepend a 1, as we want the first partition to start at the beginning.

Note that the left argument doesn't have to be the same length as the right. If it's shorter, it will be padded with zeros, allowing us a convenient way to chop a vector into a head and tail:

```
1 1⊂⍳10
```

```
┌─┬─────────┐
│1│2 3 4 5 6 7 8 9 10│
└─┴─────────┘
```

Non-boolean left ⊂

From v18 of Dyalog, the left argument is no longer restricted to a Boolean array, which allows us to generate empty partitions:

```
1 0 0 2 0 0 3 0 1 ⊂ ⍳20
```

```
┌─────┬─────┬┬┬─────────────────────────────────┐
│1 2 3│4 5 6│││7 8│9 10 11 12 13 14 15 16 17 18 19 20│
└─────┴─────┴┴┴─────────────────────────────────┘
```

We can think of each number as specifying how many partitions beyond its neighbor to the left it is. So 0 means same partition, 1 means new partition, 2 means "skip one", 3 means "skip two" etc.

Error handling

> Controlling complexity is the essence of computer programming. –*Brian Kernighan*

Other resources on the topic:

- Dyalog docs: Error guards, Error codes, Signal event
- Dyalog webinars: Part 1, Part 2, Part 3, Part 4

By now you will have encountered a range of errors where Dyalog takes objection to something you wrote, like for example

```
vec ← ⍳10
50⊃vec      ⍝ INDEX ERROR
```

```
INDEX ERROR
      50⊃vec      ⍝ INDEX ERROR
        ∧
```

If you're used to a Java or Python stack trace, the errors thrown by Dyalog will appear spartan, but after a bit of experience, they're actually *blissfully* spartan, not unlike APL itself. As an aside, the K language is even more brutal in its simplicity here.

So how do you deal with errors? There is no `try-catch-throw` exception mechanism to draw on. What we have instead is the concept of an *error guard*:

```
]dinput
pick50 ← {
    3::'out of range' ⍝ Error guard, catching INDEX ERROR
    50⊃⍵
}
```

```
pick50 ⍳10
```

```
out of range
```

An error guard takes an array of error numbers to the left, and an expression to be returned to the right, separated by two colons. Simple, right? Wrong.

Here's a rather contrived function that prepends a 1 to its left argument and then multiplies it with the inverse of its right argument. We want to protect against division by zero, in which case we just want to return the 1 prepended to the left argument, like so:

```
]dinput
f ← {
    nums←1 ⍺
    0=⍵:nums
    nums×÷⍵
}
```

```
4 f 0
4 f 2
```

```
1 4
0.5 2
```

This seems like a job for an error guard. Division by zero in this case will throw a `DOMAIN ERROR`, which has a code of 11.

Try to predict what happens before revealing the actual result:

```
]dinput
fbug ← { ⍝ Can you spot the bug?
    11::nums
    nums←1 ⍺
    nums×÷⍵
}
```

```
4 fbug 0
```

What happens is that the local environment is unwound *before* the error-guard's body is evaluated. To get the behavior we wanted we'd need to do this:

```
]dinput
fnew ← { ⍝ Now correct
    nums←1 ⍺
    11::nums
    nums×÷⍵
}
```

```
4 fnew 0
```

```
1 4
```

The other subtlety is that following the *setting* of an error guard, subsequent calls disable tail-call optimization:

```
]dinput
g ← {
    3::'out of range' ⍝ Error guard, catching INDEX ERROR
    h ⍺⊃⍵ ⍝ No longer a tail call, due to presense of error guard
}
```

This can be worked around by localizing the error guard in its own dfn:

```
]dinput
g ← {
    val ← ⍺ {3::'out of range' ⋄ ⍺⊃⍵} ⍵
    h val ⍝ Tail call
}
```

The mapping of numeric error code to error message can be found in Dyalog's documentation, referenced above, or via the system function ⎕EM:

```
⎕EM⍳10 ⍝ ...and the rest
```

WS FULL	SYNTAX ERROR	INDEX ERROR	RANK ERROR	LENGTH ERROR	VALUE ERROR	FORMAT ERROR	ERROR 8	ERROR 9	LIMIT ERROR

Error guards can also be stacked if you want to take different actions for different kinds of errors:

```
]dinput
g ← {
    3::'handle index error'
    4::'handle rank error'
    11::'handle domain error'
    ⍝ error-prone function body here
}
```

or if you want a catch-all for the errors you think you might encounter, the guard takes a vector to the left:

```
]dinput
g ← {
    3 4 11::'it all went pear-shaped in some way'
    ⍝ error-prone function body here
}
```

You can also (sort of) throw "exceptions". You may have noticed the definition for `assert` I've used here and there already:

```
assert ← {⍺←'assertion failure' ⋄ 0∊⍵:⍺ ⎕SIGNAL 8 ⋄ shy←0}
```

```
assert 0=1
```

```
assertion failure
      assert 0=1
      ∧
```

The ⎕SIGNAL ambivalent system function *throws* an error of the number given by its right argument, presenting as the corresponding message from the vector given by ⎕EM if no left argument is given, or by the left argument if given, as is the case in the `assert` function above. In other words, we could for example claim that the "workspace is full" by throwing error 1, somewhat duplicitously:

```
⎕SIGNAL 1
```

```
WS FULL
      ⎕SIGNAL 1
      ∧
```

In the `assert` function we make use of an error code that isn't used by Dyalog normally, error 8:

```
⎕SIGNAL 8
```

```
ERROR 8
      ⎕SIGNAL 8
      ∧
```

The APL Way

> Every reader should ask himself periodically "Toward what end, toward what end?"—but do not ask it too often lest you pass up the fun of programming for the constipation of bittersweet philosophy. –*Alan Perlis*

Up until now, we've skirted around one of the main advantages of APL – array-oriented, or *data-parallel* programming. This feels awkward and unnatural at first, but finding data-parallel approaches to problems is a skill that makes for efficient solutions in other languages, too, not just APL, and libraries such as Python's NumPy encourages such solutions (it was inspired by APL, by the way).

Note

In this chapter, we'll be making some comparisons between data-parallel APL and "loop & branch" implementations in Python. We chose Python because its syntax is clean and understandable by a large proportion of programmers from other languages, too. In case it's not immediately obvious, no effort has been made to find optimal Python solutions here; indeed, quite the opposite. View the Python examples as pseudocode illustrations of the algorithms, and yes, we're fully aware that one can string together elegant, efficient Python solutions using iterator algebra and comprehensions.

A few pointers – Richard Park gave a series of webinars on Thinking in APL that you should check out, and Adám Brudzewski gave several interactive Cultivations dedicated to the topic, Lesson 39 - Array programming techniques and Lesson 42 - Array coding style in depth, too.

```
⎕IO ← 0
]box on -s=min
]rows on
assert←{⍺←'assertion failure' ⋄ 0∊⍵:⍺ ⎕signal 8 ⋄ shy←0}
```

The power of the Array

Several factors super-charge array-oriented programming in APL:

1. Many operations on non-nested arrays are fast; *really* fast, taking advantage of data parallelism in the processor (SIMD).
2. By operating on whole arrays, we can – with a bit of practice – avoid branches, thus avoiding processor branch-prediction misses.

3. Arrays in APL are implemented frugally when it comes to memory, especially Boolean arrays.

The helicopter pitch for the simplest case might go something like this. Here's a naive "loop and branch" way to pick all elements matching some predicate in Python:

```
def select(f, array):
    """Loop & branch"""
    result = []
    for element in array:
        if f(element):
            result.append(element)
    return result
```

and here's a data-parallel version:

```
elems ← f array ⍝ Boolean mask
elems/array     ⍝ Compress; i.e. pick elemets at 1s
```

Let's say we want to find the odd numbers in a sequence:

```
where ← 2|data ← ⍳20 ⍝ mod-2 for the whole array at once
where/data
```

```
1 3 5 7 9 11 13 15 17 19
```

In the APL version, there is no loop, and no branch. The data is simple, and most likely allocated contiguously, and the CPU will be able to process several items in parallel for every cycle without any branch-prediction misses. Of course, with only 20 elements, algorithmic performance is a largely academic concern.

We can think of many problems that can be solved using the pattern of a scalar selection function applied to an array. One example, given by Richard Park in his webinar mentioned above, is to pick out all vowels from a string:

```
'aeiou' {(⍵∈⍺)/⍵} 'pick out all vowels from a string'
```

```
iouaoeoai
```

…or for train spotters:

```
'aeiou' (∊⍨⊢⍤⍳⍤≢/⊢) 'pick out all vowels from a string'
```

```
iouaoeoai
```

It's the same idea as the 'odd numbers' above: create a Boolean mask using set membership (dyadic ∊) and compress.

As an aside, and for the avoidance of any doubt, the *actual* APL way would be to just intersect the string with the vowels:

```
'aeiou' ∩⍨ 'pick out all vowels from a string'
```

```
iouaoeoai
```

Shorter, faster, array-ier, nicer – but not the point we wanted to make.

Luhn's Algorithm

For something meatier, let's look at implementing Luhn's algorithm, an error-detecting code used for validating credit card numbers. This was used to illustrate array-oriented concepts by Dyalog CTO, Morten Kromberg, at a presentation given at JIO. We'll follow Morten's clear explanation of this algorithm:

1. Split the credit card number into a *body* and the last digit, a *checksum*

```
Card number: 7 9 9 2 7 3 9 8 7 1 3
Body:        7 9 9 2 7 3 9 8 7 1    Checksum: 3
```

2. Multiply every other digit in the body by 2, starting from the last digit with a 2

```
Body:        7  9 9 2 7 3 9  8 7 1
Weights:     1  2 1 2 1 2 1  2 1 2
           ×_____________________
Products:    7 18 9 4 7 6 9 16 7 2
```

3. Separate any numbers greater than 9 into their individual digits, and sum columns

```
Tens:        0 1 0 0 0 0 0 1 0 0
Units:       7 8 9 4 7 6 9 6 7 2
           +_________________
Sum:         7 9 9 4 7 6 9 7 7 2
```

4. Sum the sums

```
Sum:         7+9+9+4+7+6+9+7+7+2 = 67
```

5. Sum the *digits* of this, modulo 10:

```
Result:      (6+7)%10 = 3
```

If the result equals the checksum, the card number is valid.

Here's one way this could be implemented in Python:

```python
def luhn(cardnum):
    checksum = 0
    parity = len(cardnum) % 2
    for index in range(len(cardnum)-1, -1, -1):
        digit = cardnum[index]
        if (index + 1) % 2 != parity:
            digit *= 2
            if digit > 9:
                digit -= 9
        checksum += digit

    return checksum % 10 == 0
```

It may or may not be clear that the Python implementation follows the algorithm description. We step through the card number backwards making it easier to ensure that we get the factors right, regardless of the number of digits.

Now let's implement this in APL. We can follow the algorithm description very closely.

1: Split the credit card number into a *body* and the last digit, a *checksum*

```
card ← 7 9 9 2 7 3 9 8 7 1 3

 ← body ← (count←¯1+≢card)↑card
 ← check ← ⊢/card
```

```
7 9 9 2 7 3 9 8 7 1
3
```

2: Multiply every other digit in the body by 2, starting from the last digit with a 2

```
 ← weights ← countρ(2|count)⍋1 2
 ← products ← body×weights
```

```
1 2 1 2 1 2 1 2 1 2
7 18 9 4 7 6 9 16 7 2
```

3, 4. Separate any numbers greater than 9 into their individual digits, and sum

```
 ← digits ← 0 10⊤products ⍝ We can think of `0 10⊤` as divmod
 ← sum ← +⌿,digits        ⍝ By raveling, we save doing sum the sum of cols
```

```
0 1 0 0 0 0 0 1 0 0
7 8 9 4 7 6 9 6 7 2
67
```

5. Digit sum, mod 10

```
10|-sum ⍲ Magic
```

```
3
```

If we put all that together, we arrive at Morten's solution:

```
]dinput
luhn ← {
    check ← ⊢/⍵
    body ← (count←¯1+≢⍵)↑⍵
    weights ← count⍴(2|count)⍋1 2
    digits ← 0 10⊤body×weights
    check=10|-+/,digits
}
```

```
assert luhn 7 9 9 2 7 3 9 8 7 1 3
```

The Luhn algorithm was obviously chosen as an ideal fit for a data-parallel implementation. A few things stand out: firstly, the APL implementation follows the algorithm definition very closely, much closer than the Python version. Granted – and this is an interesting experiment – you can take the APL approach and implement a Python version following a similar pattern, albeit without array operations. Secondly, the solution is completely free or loops and branches. As Morten also observes, over 10 digits, there is no meaningful performance implication here, but it's not hard to imagine what would happen over a million or more digits.

Balancing the Scales

This is Problem 8, Phase II from the 2020 Dyalog problem solving competition. You can see the problem statement here. Obviously, there are spoilers to follow – if you want to have a crack at it yourself, stop reading here.

Our task is to partition a set of numbers into two groups of equal sum if this is possible, or return `0` if not. This is a well-known NP-hard problem, called The Partition Problem, and as such has no fast, always correct solutions for the general case. The problem statement indicates that we only need to consider a set of 20 numbers or fewer, which is a bit of a hint on what kind of solution is expected.

This problem, in common with many other NP problems, also has a plethora of interesting heuristic solutions: faster algorithms that whilst not guaranteed to always find the optimal solution will either get close, or be correct for a significant subset of the problem domain in a fraction of the time the exhaustive search would take. We'll look at one of these approaches, too, although it probably wasn't what Dyalog wanted in terms of the competition (I might be wrong; I didn't take part at the time).

So given that we have a defined upper bound, and that Dyalog wants the solution to be optimal for all inputs up to this bound, and we know this problem to be NP-hard, we're looking at an exhaustive search of some sort. We can also guess that given this is a Dyalog competition problem, whilst the algorithm might be crude, we should be able to exploit APL's fast array processing to implement it efficiently.

The algorithm (if we can call it that) is simply to try every possible combination in which the numbers can be put into two separate piles and check if the partition sum equals half the total sum. If we have `n` items in our set, that corresponds to considering all numbers up to `¯1+2*n` in binary (recall that `*` is exponentiation, not multiplication), and the two sets correspond to those matching the 0 bits and 1 bits respectively. For example, if we have the numbers 1, 1 and 2 we need to consider the following partitions (although we can skip the first and last):

```
(0: 0 0 0)
 1: 0 0 1
 2: 0 1 0
 3: 0 1 1
 4: 1 0 0
 5: 1 0 1
 6: 1 1 0
(7: 1 1 1)
```

In this case, the solutions would be either 0 0 1, or its inverse, 1 1 0, corresponding to the partitioning `(1 1)(,2)`.

So what are we dealing with here, in terms of the search space?

```
¯1+2*20
```

```
1048575
```

Just over a million or so different partitions to check, which should be manageable. The search space doubles in size for each additional item, which is something alluded to in the old fable about the inventor of Chess:

```
e←¯1+2∘*⍳e¨21+⍳9
```

```
2097151 4194303 8388607 16777215 33554431 67108863 134217727 268435455 536870911
```

To generate all binary combinations for n bits, we can use the following incantation (for n=3):

```
⍉2∘⊥⍤¯1⍳2*3
```

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

Given a Boolean vector, how do we turn that into the two corresponding sets of numbers? There are several ways you can try, for example compress:

```
1 1 0 {(⍺/⍵)(⍵/⍨~⍺)} 1 1 2 ⍝ compress and compress ~
1 1 0 {(⍵~⍵)(v←⍺/⍵)} 1 1 2 ⍝ compress and set difference
```

1	1	2

2	1	1

but we can exploit the fact that we're really interested in the *sums* of the sets. The sum of the items corresponding to the set bits is really just the bit pattern times the input set, summed:

```
+/1 1 0×1 1 2
```

```
2
```

which we by now hopefully should recognize as an inner product:

```
1 1 0 +.× 1 1 2
```

```
2
```

And of course, inner product can be applied to an array, too:

```
patterns ← ⍉2∘⊥⍤¯1⍳2*3
patterns +.× 1 1 2
```

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

We can now draw up a plan for this.

1. Our target partition sum is the total divided by two.
2. Generate the bit patterns up to ≠⍵.
3. Calculate the sums of the numbers corresponding to the set bits in each pattern
4. Find the first point where the partition sum is equal to the target
5. Return the corresponding partitioning

This translates readily to APL:

```
]dinput
Balance ← {
    total ← +/⍵
    2|total: 0              ⍝ Sum must be divisible by 2
    psum ← total÷2          ⍝ Our target partition sum
    bitp ← ⍉2∘⊥⍤¯1⍳2*≠⍵     ⍝ All possible bit patterns up to ≠⍵
    idx ← ⍳<\psum=bitp+.×⍵ ⍝ First index of partition sum = target
    0≡idx: 0                ⍝ If we have no 1s, there is no solution
    part ← idx⌷bitp         ⍝ Partition corresponding to solution index
    (part/⍵)(⍵/⍨~part)      ⍝ Compress input by solution pattern and inverse
}
```

```
Balance 1 1 2
```

2	1 1

and on the full 20-bits:

```
Balance 10 81 98 27 28 5 1 46 63 99 25 39 84 87 76 85 78 64 41 93
```

99 84 87 76 85 41 93	10 81 98 27 28 5 1 46 63 25 39 78 64

To a Python programmer, this approach must seem *incredibly* counter-intuitive, borderline criminally wasteful. Despite the fact that we only want the first solution (which occurs around bit pattern 1,500ish), we go through the *entire* search space. In a "loop and branch" language, we'd try each candidate pattern in turn, and break once we find the first solution.

But the seemingly naive APL solution is quick, even though it will process 2-3 orders of magnitude more patterns than would a scalar solution:

```
]runtime "Balance 10 81 98 27 28 5 1 46 63 99 25 39 84 87 76 85 78 64 41 93"
```

```
* Benchmarking "Balance 10 81 98 27 28 5 1 46 63 99 25 39 84 87 76 85 78 64 41 93"
```

	(ms)
CPU (avg):	28
Elapsed:	29

So what's going on here? Well, two things. Firstly, we're hitting the most optimal core of Dyalog APL, its handling of Boolean vectors, using only primitive functions. Secondly, APL is able to vectorise our operations, taking advantage of SIMD operations in the processor.

For completeness, let's see how a scalar solution would perform. We can, for example, take a tail-call Scheme-like approach:

```
]dinput
BalanceScalar ← {⎕IO←0
    total ← +/⍵
    2|total: ⍬                ⍝ Sum must be divisible by 2
    psum ← total÷2            ⍝ Our target partition sum
    data ← ⍵
    bitp ← ↓⍉2∘⊥⍣¯1⍳2*≢⍵     ⍝ Pre-compute the bit patterns
    {                         ⍝ Try one sum after the other, halt on first solution
        0=⍵: ⍬
        patt ← ⍵⊃bitp
        psum=patt+.×data: (patt/data)(data/⍨~patt) ⍝ Exit on first solution found
        ∇¯1+⍵
    } ¯1+≢bitp
}
```

```
BalanceScalar 1 1 2
```

1 1	2

Ok, let's compare:

```
'cmpx'⎕CY'dfns'
```

```
d←10 81 98 27 28 5 1 46 63 99 25 39 84 87 76 85 78 64 41 93
cmpx 'Balance d' 'BalanceScalar d'
```

```
  Balance d       → 2.7E¯2 |   0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
* BalanceScalar d → 4.0E¯2 | +46% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
```

Ouch. That difference is pretty stark.

Dyalog can no longer vectorise, and performance slips considerably. If you take home anything from this chapter, this is the thing to remember.

However, the brute-force algorithm is of course still crude, even if APL can be quick about it at the specified scale. Add a few more elements, and the unforgiving `O(2^N)` complexity would soon crush us.

As I mentioned earlier, there are a number of heuristic algorithms that have a much more benign complexity, at the cost of not being able to guarantee optimal solutions in all cases. One such algorithm is called the Karmarkar-Karp, after its inventors, and has a complexity of `O(N log N)`, but is not guaranteed to find the optimal solution, even if such a solution does exist.

In short, the KK algorithm maintains a priority queue of the remaining numbers, and picks the two largest numbers in each iteration, replacing them with their difference until a single number remains – this represents the final difference between the two partitions. We can construct the corresponding partitions via backtracking. A thorough analysis of this algorithm can be found here.

```
]dinput
KarmarkarKarp ← {
    sort ← {ω[;⍋ω[0;]]}
    (pairs last) ← θ {
        2>≢⍉ω:α (1 0⌷ω)
        (i x)(j y) ← ↓⍉ω[;0 1]
        (α,⊂x y)∇sort (2↓⍤1⊢ω),(i-j) x
    } sort ↑(-ω)(ω)

    last θ {
        (a b) ← α
        0=≢ω: α
        pair ← ⊃¯1↑ω
        a∊⍨⊃pair: (a (b,1⊃pair))∇¯1↓ω
        ((a,1⊃pair) b)∇¯1↓ω
    } pairs
}
```

In our test case, the KK does return an optimal solution:

```
KarmarkarKarp 10 81 98 27 28 5 1 46 63 99 25 39 84 87 76 85 78 64 41 93
```

```
┌──────────────────────────────┬─────────────────────────────┐
│41 85 99 5 93 10 63 27 64 78│1 25 28 76 81 39 46 84 87 98│
└──────────────────────────────┴─────────────────────────────┘
```

```
cmpx 'Balance d' 'KarmarkarKarp d'
```

```
  Balance d         → 3.7E¯2 |    0% ⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕⎕
* KarmarkarKarp d → 1.4E¯4 | -100%
```

and, unsurprisingly, although completely scalar, it wins hands down. However, it's still a heuristic:

```
Balance 4 5 6 7 8
KarmarkarKarp 4 5 6 7 8
```

```
┌───┬─────┐
│7 8│4 5 6│
└───┴─────┘
┌─────┬───┐
│4 5 7│6 8│
└─────┴───┘
```

A thing to note in the KK implementation above: it simply sorts the array at each iteration, instead of using a heap queue. As an exercise for the interested reader, try re-implementing it using a heap. However, and for similar reasons as we saw earlier, you will probably find that APL is likely faster at sorting a simple array than the complexity cost of maintaining a more nested data structure.

Merge

Ok, we're on a roll. Problem 6 from the same year was a spin on the old "mail merge" concept: given a template file, expand templates with values from a corresponding merge file. For example, given a template of

```
Hello @firstname@,
You have been selected to test drive a new @carmake@!
```

with a merge file of

```
{
    "firstname": "Bob"
    "carmake": "Aston Martin"
}
```

the final output should be

```
Hello Bob,
You have been selected to test drive a new Aston Martin!
```

A literal `@` is represented by `@@` in the template, and templates with no corresponding match in the merge file should be expanded to `???`.

We'll try two different solutions. For the first approach, it's hard not to think of regexes here – replace "templates" matching a defined pattern with corresponding values.

Let's look at the two data files first:

```
template ← '/Users/stefan/template.txt'
⊃⎕NGET template
```

```
@salutation@ @firstname@ @lastname@;
Congratulations!  You have won a @prize@ worth over £@value@!
@firstname@, please come to our office to pick up your @prize@.
Please feel free to contact us at info@@contest.com.
Your email address in our domain is @firstname@@@contest.com
```

```
merge ← '/Users/stefan/merge1.json'
⊃⎕NGET merge
```

```
{
    "firstname":"Drake",
    "lastname":"Mallard",
    "prize":"yoyo",
    "value":100
}
```

So our patterns are defined by the keys in the JSON, plus the `@@` for literal `@` and any remaining templates that have no corresponding entry in the merge file, along the lines of `@[^@]+@`. We'll start with reading the JSON data into a namespace and pulling out the keys and values:

```
mrg←⎕JSON⊃⎕NGET merge
keys←mrg.⎕NL¯2
vals←mrg.(⍎¨'#.keys')
↑keys vals
```

firstname	lastname	prize	value
Drake	Mallard	yoyo	100

and our patterns and replacements then become:

```
↑(({'@',(⍕ω),'@'}¨keys), (⊂'@@'), ⊂'@[^@]+@') ((⍕¨vals), (⊂,'@'), ⊂'???')
```

@firstname@	@lastname@	@prize@	@value@	@@	@[^@]+@
Drake	Mallard	yoyo	100	@	???

Putting it all together we get:

```
]dinput
Merge ← {
    mrg←⎕JSON⊃⎕NGET ⍺
    keys←mrg.⎕NL¯2
    vals←mrg.(⍎¨'#.keys')

    (({'@',(⍕ω),'@'}¨keys), (⊂'@@'), ⊂'@[^@]+@')⎕R((⍕¨vals), (⊂,'@'), ⊂'???')⊢⊃⎕NGET ω
}
```

```
merge Merge template
```

```
??? Drake Mallard;
Congratulations!  You have won a yoyo worth over £100!
Drake, please come to our office to pick up your yoyo.
Please feel free to contact us at info@contest.com.
Your email address in our domain is Drake@contest.com
```

All well and good, but not very “array-oriented” now, is it? Let’s remedy that, and forget about cheaty regexes.

If we partition the data such that each partition begins with `@` we get this:

```
'@' (=⊂⊢) 'aaaa @bbb@ ccc @@ @ddd@' ⍝ Partitions starting at @
```

@bbb	@ ccc	@	@	@ddd	@

in fact, let’s drop the leading `@`, too:

```
□ ← tmpls←'@'(1↓¨=⊂⊢)'aaaa @bbb@ ccc @@ @ddd@'
```

bbb	ccc			ddd	

What we have now is a nested vector where every other element is a template.

Our replacement values will again come from a namespace

```
(mrg←□NS⍬).(bbb ccc ddd) ← 'bees' 'cees' 'dees'
```

and we’ll make a little helper function to look up value by key, with the added functionality to return `@` for `` and `???` for non-present keys:

```
]dinput
val ← {
    0=≢⍵: ,'@'
    ~(⊂⍵)∈mrg.□NL¯2: '???'
    ⍎mrg⍱⍵
}
```

```
val¨'bbb' 'ddd' 'ccc' '' 'hubba'
```

bees	dees	cees	@	???

We can use Dyalog’s handy `@` operator to make the replacements. Recall that the `@` operator can take a right operand function which must return a Boolean vector, which in our case should select every other element, starting with the first:

```
val¨@{1 0 1 0 1 0}⊢tmpls
```

bees	ccc	@		dees	

Finally, we’d need to tack on anything prior to the first `@` and enlist.

```
]dinput
Merge ← {
    mrg ← □JSON⊃□NGET ⍺
    templ ← ⊃□NGET ⍵
    first ← templ⍳'@'
    first>≢templ: templ    ⍝ No templates at all
    prefix ← first↑templ
    val ← {
        0=≢⍵: ,'@'
        ~(⊂⍵)∈mrg.□NL¯2: '???'
        ⍎mrg⍱⍵
    }
    ∈prefix,val¨@{1 0⍴⍨≢⍵}'@'(1↓¨=⊂⊢)templ
}
```

```
merge Merge template
```

```
??? Drake Mallard;
Congratulations!  You have won a yoyo worth over £100!
Drake, please come to our office to pick up your yoyo.
Please feel free to contact us at info@contest.com.
Your email address in our domain is Drake@contest.com
```

Right-align a block of text

Let's work through a more comprehensive problem. Here's a tweaked version of one of the Phase 1 tasks from the Dyalog Problem Solving Competition, 2021:

Write a function that:

- has a right argument `T` which is a character scalar, vector or a vector of character vectors
- has a left argument `W` which is a positive integer specifying the width of the result
- returns a right-aligned character array of shape `((2=≡T)/≠T),W`.
- if an element of T has length greater than `W`, truncate it after `W` characters.

The challenge here is not resorting to 'eaching' the rows, or employing some creative regexing. So let's treat this as an exercise in array-oriented problem solving.

We're given a couple of examples of how the solution should behave for arrays of different ranks:

```
⍝ 6 Align '⍦'
'     ⍦'

⍝ 10 Align 'Parade'
'    Parade'

⍝ 8 Align 'Longer Phrase' 'APL' 'Parade'
3 8⍴'Longer P     APL  Parade'
```

```
     ⍦
    Parade
Longer P
     APL
  Parade
```

Note

The last example above is different from the way the problem was published. For extra points, solve it as it was presented in the competition:

```
3 8⍴'r Phrase     APL  Parade'
```

A hint is given in the problem statement on where we could start: we're told what the shape should be of the resulting array: `((2=≡T)/≠T),W`. Given the result shape, we can try the following approach:

Figure out the result shape, and squeeze the data into this shape, truncating if we need to. Locate trailing spaces. Prepend a space on each line, and then replicate by the number of trailing spaces.

We're given the shape. In order for this to work for character scalars, a character vector, or vector of character vectors, we need to operate on the ravel of the right argument when finding the shape:

```
6 {⍺,⍴(2=≡,⍵)/≠,⍵} '⍋'
10 {⍺,⍴(2=≡,⍵)/≠,⍵} 'Parade'
8 {⍺,⍴(2=≡,⍵)/≠,⍵} 'Longer Phrase' 'APL' 'Parade'
```

```
6
10
3 8
```

To fit the data into that shape, we need to *Mix* the ravel (`↑,⍵`) to increase the rank and then *Take* `⍺` elements, rank 1 (along vectors), and then reshape. Let's explore what this means:

```
data ← 'Parade'
⊢sh ← 10 {α,⍨(2=≡,ω)/≢,ω} data ⍝ Shape
↑,data                          ⍝ Mix ravel (no change for a char vector)
10↑⍤1⊢↑,data                    ⍝ (over)take, rank-1
```

```
10
Parade
Parade
```

More interesting to consider is the nested vector case:

```
data ← 'Longer Phrase' 'APL' 'Parade'
□ ← sh ← 8 {α,⍨(2=≡,ω)/≢,ω} data ⍝ Shape
↑,data                            ⍝ Mix ravel
□ ← mat ← 8↑⍤1⊢↑,data             ⍝ (over)take, rank 1
```

```
3 8
Longer Phrase
APL
Parade
Longer P
APL
Parade
```

So far, so array-oriented! Now we need to find trailing spaces. The equal sign makes for an excellent search function! Where are the spaces?

```
' '=mat
```

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

All well and good, but we need the *trailing* spaces. A handy technique here is to *and-scan*, which is one of those APLisms that is so obvious (once someone pointed it out to you, you found it on APLCart, or you're a genius). It keeps all 1s from the beginning, and zeros everything after the first zero:

```
∧\1 1 1 0 0 0 1 0 1 0
```

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

but that looks at *leading* not trailing spaces, so we need to flip, and-scan, flip back:

```
⌽∧\⌽' '=mat   ⍝ We're keeping on arraying
```

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

Conceptually, we'll add extra spaces at the beginning of each line, the same number as any trailing spaces. In practice, we'll add a single space, and then use a replication vector to clone it the relevant number of times. Recall:

```
□ ← data ← ' ABCDEFGHIK'         ⍝ Note: leading space
□ ← repl ← 5 1 1 1 1 1 1 1 1 1 1 ⍝ Five spaces, one of everything else
repl/data
```

```
 ABCDEFGHIK
5 1 1 1 1 1 1 1 1 1 1
     ABCDEFGHIK
```

```
trailing ← ⌽∧\⌽' '=mat     ⍝ Find trailing spaces, as per above
□ ← spaced ← ' ',mat        ⍝ Add a space as the first char of each row
□ ← keepers ← ~trailing     ⍝ Everything not a trailing space
□ ← padding ← +/trailing    ⍝ Amount of leading padding to be inserted per row
□ ← repl ← padding,keepers ⍝ Number of replications per character
```

```
 Longer P
 APL
 Parade
1 1 1 1 1 1 1 1
1 1 1 0 0 0 0 0
1 1 1 1 1 1 0 0
0 5 2
0 1 1 1 1 1 1 1 1
5 1 1 1 0 0 0 0 0
2 1 1 1 1 1 1 0 0
```

Now what remains is to do the replication and ensure that everything gets the shape it should have.

```
repl/⍤1⊢spaced ⍝ Replicate along vectors
```

```
Longer P
     APL
  Parade
```

That's it! Putting it all together, we get the following:

```
]dinput
Align ← {
    shape ← ⍺,≢(2=≡,⍵)/≢,⍵  ⍝ Shape, as specified
    mat ← ⍺↑⍤1⊢↑,⍵          ⍝ Truncate rank 1
    trailing ← ⌽∧\⌽' '=mat  ⍝ Find trailing spaces
    spaced ← ' ',mat        ⍝ Add a space as the first char of each row
    keepers ← ~trailing     ⍝ Everything not a trailing space
    padding ← +/trailing    ⍝ Amount of leading padding to be inserted per row
    repl ← padding,keepers  ⍝ Number of replications per character
    repl/⍤1⊢spaced          ⍝ Replicate rank 1
}
```

```
6 Align '⍋'
10 Align 'Parade'
8 Align 'Longer Phrase' 'APL' 'Parade'
8 Align 'K' 'E' 'Iverson'
```

```
     ⍋
    Parade
Longer P
     APL
  Parade
       K
       E
 Iverson
```

Pretty cool, eh? As an exercise, now shrink that to a one-liner suitable for submitting as a competition entry.

Shall we do one more?

FizzBuzz

Adám Brudzewski used the classic FizzBuzz problem – the darling of technical interviewers everywhere – as an illustration in one of his Cultivation lessons, Lesson 39 - Array programming techniques.

Wikipedia has the following to say about FizzBuzz:

> Fizz buzz is a group word game for children to teach them about division. Players take turns to count incrementally, replacing any number divisible by three with the word "fizz", and any number divisible by five with the word "buzz".

but most programmers will be familiar with it as a task frequently set as a technical challenge in job interviews. In fact, we've seen an APL version of this already. Right at the beginning it was used as an illustration of a *trad function*.

Here's a "loopy" version as an APL dfn, for context:

```
{∊(3↑(0=3 5|⍵)∪1)/'Fizz' 'Buzz'⍵}¨1+⍳16
```

1	2	Fizz	4	Buzz	Fizz	7	8	Fizz	Buzz	11	Fizz	13	14	FizzBuzz	16

Can we do without the each, given what we've already learnt? Let's ponder what an array solution might look like.

Let's begin by creating a Boolean mask showing the positions of the numbers which are divisible by 3 (the "fizzes") and those divisible by 5 (the "buzzes"). We can do that as an outer product:

```
data ← 1+⍳16
□ ← fizzbuzz ← 0=3 5∘.|data
```

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

So the columns that have only zeros in them represent the “numbers” - the “non-fizzbuzzy” bits. Let’s add a new row to our matrix to make this clear:

```
⍝ ← mat ← (⍲⌿⍪⊢)fizzbuzz  ⍝ Tacit for {(⍲⌿⍵)⍪⍵} -- column-wise logical NOR as the new first row
```

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

If we now multiply the first row by our input numbers, we have the numbers that aren’t “fizzbuzzy”

```
mat×@0⍤←data ⍝ Check that out! Modified assignment
mat
```

```
1 2 0 4 0 0 7 8 0 0 11 0 13 14 0 16
0 0 1 0 0 1 0 0 1 0  0 1  0  0 1  0
0 0 0 0 1 0 0 0 0 1  0 0  0  0 1  0
```

Now we substitute all 1s for “fizz” in the second row, and for “buzz” in the third row:

```
mat[↓1,⍤0 0⊢⍳1⍴⍨≢mat]←⊂'fizz'
mat[↓2,⍤0 0⊢⍳1⍴⍨≢mat]←⊂'buzz'
mat
```

1	2	0	4	0	0	7	8	0	0	11	0	13	14	0	16
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

Merge columns, remove zeros:

```
0⍤⍲⌿⍤¯1⊢⍪mat
```

1	2	fizz	4	buzz	fizz	7	8	fizz	buzz	11	fizz	13	14	fizzbuzz	16

Nice. So in this case, is the longer array solution better than the clever loopy one we showed in the beginning? Well, the intention was to demonstrate how to approach solutions in a data-parallel way, but the problem was of a magnitude that probably made this unnecessary. But practice makes perfect.

Namespaces ⎕NS

> The utility of a language as a tool of thought increases with the range of topics it can treat, but decreases with the amount of vocabulary and the complexity of grammatical rules which the user must keep in mind. Economy of notation is therefore important. *–Kenneth E. Iverson*

A namespace is a way to group data and code into a hierarchy. Dyalog describes namespaces like so:

> Namespace is a (class 9) object in Dyalog APL. Namespaces are analogous to nested workspaces.

No, that doesn’t mean anything to me either. In fact, APL namespaces are similar in spirit to those found in C++:

> Namespaces provide a method for preventing name conflicts in large projects. Symbols declared inside a namespace block are placed in a named scope that prevents them from being mistaken for identically-named symbols in other scopes.

Here’s an anonymous namespace:

```
obj ← ⎕NS⍬
```

We can assign values to variables inside this namespace:

```
obj.(name cost id)  ← 'widget' 55.0 'widg443'
```

```
obj.(name cost id)
```

widget	55	widg443

Names inside a namespace can hold any value that names can hold *outside* a namespace, including functions:

```
obj.sum←+/
```

```
obj.sum 1 2 3 4 5
```

```
15
```

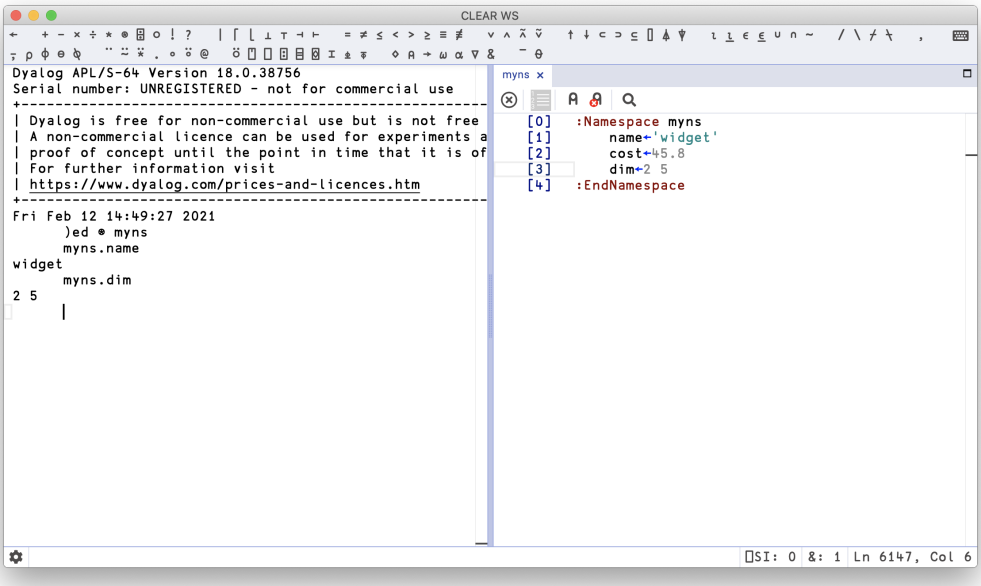
There is a handy user command, `]map` that gives a tree outline of a namespace that's worth adding to your repertoire:

```
]map obj
```

```
#.[Namespace]
·   ~ cost id name
·   ∇ sum
```

Scripted namespaces

A nifty feature is that we can compose a namespace as a *script*. In RIDE, you need to say `)ed ⍟ mynamespace` in order to work with a scripted namespace. Yes, of course that's a good use for the *Logarithm* glyph. It looks like so:



In this way, the namespace can be a convenient way to organise your code. And in case it wasn't obvious, it's actually a way in which you can have many multi-line dfns in the same editing window – or even text file. It even works as intended in the Jupyter notebook:

```
]dinput
:Namespace myns
    f ← {
        α+ω
    }

    g ← {
        ⍝ g fun
        αω
    }

    h ← {
        s ← '\d+'⎕R'D'⊢ω
        α g s
    }
:EndNamespace
```

```
'nodigits' myns.h 'abd556jashgd8879'
```

nodigits	abdDjashgdD

Wait, this is starting to look like a dict!

We can get tantalisingly close to having a namespace function as a dict. In order to list the names of variables contained in a namespace, we have the namelist, ⎕NL, system function.

```
⍝ ← names ← obj.⎕NL 2
```

```
cost
id
name
```

The 2 there lets ⎕NL know that we want an array back.

```
⍴names
```

```
3 4
```

If we feed it ¯2 instead we get a nested vector instead:

```
obj.⎕NL ¯2
```

cost	id	name

The right argument to ⎕NL is a name class, allowing us to select based on what kind something contained in the namespace is. Many of the name classes are concerned with bits of Dyalog that are out of scope for this book.

We can get the *values* of variables by evaluating their names:

```
obj⍎'cost'
obj.(⍎¨⎕NL ¯2)
```

```
55
```

55	widg443	widget

Almost, but not quite, a dict:

```
_set←{⍎'⍺⍺.',⍺,'←⍵'⍪⍺⍺}
keys←{⍵.⎕NL ¯2}
vals←{⍵.(⍎¨⎕NL ¯2)}
```

```
'hello' (obj _set) 'world'
```

```
obj⍎'hello'
keys obj
vals obj
```

```
world
```

cost	hello	id	name

55	world	widg443	widget

but this approach won't let you use anything but character vectors as keys, and the keys must also be valid APL names. There are also some performance constraints if the number of items in a namespace grow large.

A note on mutability

Namespaces in Dyalog are reference types, and mutable. This allows you to bypass some scope-related barriers that may have been erected for very good reasons, so *caveat emptor*. For example, we can mutate a namespace even if it's passed as the *left* argument, which is an error for normal arrays:

```
ns ← ⎕NS⍬
ns.key ← 45
ns {⍺.key ← 99 ⋄ ⍵} 'hello' ⍝ Mutation through ⍺...
ns.key
```

```
hello
99
```

It also means that there is no need for modified assignment through a tack, ⊢← if we want to set a value in a namespace not in our immediate scope. All of this is either very useful, or very dangerous, depending on your particular view point. The reality is that it's both useful, but also increases the risk of foot-gun incidents.

Arrays of namespaces

Certain namespace operations extend into arrays of namespaces, almost like scalar extension, which might not be obvious at all, but supremely useful. For example, we can assign to a specific field across the whole array:

```
⎕IO←0
nsarray←⎕NS¨6⍴⊂⍬
nsarray.name←'adam' 'bob' 'charlotte' 'dave' 'erica' 'fred'
nsarray.pid←3 7 87 32 32 9
nsarray.items←(1 2 3)(2 3)(9 32 23)(9 8 7 6 5 4)(8 7)(,1)
```

```
nsarray[2].name
```

```
charlotte
```

and, as a consequence, pick a field from all array elements:

```
nsarray.name
```

adam	bob	charlotte	dave	erica	fred

We can also execute functions on fields if we enclose in parentheses:

```
nsarray.(≢items)
```

```
3 2 3 6 2 1
```

or pick multiple fields, too:

```
nsarray.(name pid)
```

adam	3	bob	7	charlotte	87	dave	32	erica	32	fred	9

If you were jealous of KDB+'s neat SQL integration, we can even knock up a simple query DSL for accessing data represented as arrays of namespaces, as Adám showed in a post in APL Orchard:

```
From←{⍵⍎¨⊂⍺}
Has←{⍺.⎕NC⊂⊂⍵}
In←{⍺∘(∨/⍷)¨⍵}
Where←/⍨
```

```
'pid' From nsarray Where 'dave' In nsarray.name
```

```
32
```

Who needs SQL now? We can select on the existence of a particular field using the `Has` helper:

```
nsarray[2 5].title←'manager' 'pointy-haired boss'
```

```
'name' From nsarray Where nsarray Has 'title'
```

charlotte	fred

Left argument

We can also pass a left argument to ⎕NS, which names the namespace, *including nested names*:

```
'a'⎕NS⍬
a
```

```
#.a
```

```
'top.middle.bottom'⎕NS⍬
```

```
]map top
```

```
#.top
·   middle
·   ·   bottom
```

Dealing with real data

> You want it in one line? Does it have to fit in 80 columns? –*Larry Wall*

Amazingly, we got all the way to here without once mentioning how to get data in (or out) with less than typing it in. Nor have we dealt with "real" data much. Real data is usually messy. Dyalog APL has a rich set of routines to deal with data, either as bytes on disk, or formatted as JSON or CSV (or XML, but let's keep pretending that XML was never actually a thing), or fetching it from a database, or as an HTTP-request. Some of this stuff we won't cover – if you need it, you're probably capable enough as an APLer to figure it out yourself.

Some other resources to consult:

- Docs on ⎕NGET, ⎕CSV, ⎕JSON and (if you must) ⎕XML
- Dyalog webinar on http, ⎕JSON and ⎕XML

```
⎕IO ← 0
]box on
]rows on
```

Reading text files: ⎕NGET

The system function ⎕NGET reads text files from disk:

```
⊃⎕NGET'/Users/stefan/work/dyalog/file.txt'1
```

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

We can note two things from that:

1. We disclose the result (actually, we pick the first item)
2. There is a 1 after the file name

Let's explore what happens if we don't do these things.

```
⎕NGET'/Users/stefan/work/dyalog/file.txt'1
```

1 8 6 9 5 9 5 9 | 6 3 4 2 8 1 8 2 | 8 0 7 5 5 1 2 3 | 6 5 2 9 8 2 6 1	UTF-8-NOBOM	10

We can see that when we read a file with ⎕NGET we get back a vector with several elements, the first is the data itself, and the second contains information about the file's encoding. The last one is Dyalog's opinion of what the file's newline separator is. This will vary across operating systems.

What happens if we leave out the 1 at the end?

```
⎕NGET'/Users/stefan/work/dyalog/file.txt'
```

```
┌─────────────────┬──────────────┬──┐
│1 8 6 9 5 9 5 9  │UTF-8-NOBOM   │10│
│6 3 4 2 8 1 8 2  │              │  │
│8 0 7 5 5 1 2 3  │              │  │
│6 5 2 9 8 2 6 1  │              │  │
└─────────────────┴──────────────┴──┘
```

Leaving out the 1 (or indeed instead passing the default value of 0) returns a single character vector containing the data:

```
⍴⊃⎕NGET'/Users/stefan/work/dyalog/file.txt'
```

```
64
```

This is probably not what you want in most cases when processing a text file. Note that we've read lines of *characters*. But we can convert them to numbers:

```
⍎⍤1⊢↑⊃⎕NGET'/Users/stefan/work/dyalog/file.txt'1 ⍝ Note health warning below!
```

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

We mixed the vector into an array of rank 2, then applied the *Hydrant*, ⍎ at rank 1 (vectors). *Hydrant* is a function called *Execute*, and it takes a string and evaluates it as if it was a little APL program, similar to how eval() works in Python and Perl:

```
⍎ '1 8 6 7 5 9 5 9'
```

```
1 8 6 7 5 9 5 9
```

If you have ever seen code written in PHP, you know why evaluating strings requires a degree of caution unless you're certain of the provenance. Perl has a taint mode to stop bleed-over from untrusted sources, and Python papers over the issue by decree, calling `eval()` un-Pythonic, and Guido-hated.

> **Tip**
>
> Dyalog provides more industry-strength mechanisms for converting strings to numbers in a safe manner, like verify-fix-input, ⎕VFI

Anyway, enough of the yak shaving already.

Reading CSV: ⎕CSV

CSV, or *comma-separated values*, is a data format for tabular data. It's hairier than one might think to write a correct CSV parser. Fortunately, Dyalog provides one for us as the system function ⎕CSV. It can be used to convert data both to and from the CSV format. The ⎕CSV function has one extremely handy feature: it converts numbers for us, avoiding the hydrant-dance we resorted to above.

Consider the following decidedly not-CSV-looking data. It's a section from the data given in Day 2 in the Advent of Code competition from 2020. Let's assume we've already read from disk using ⎕NGET:

```
data ← '3-6 s: ssdssssss' '17-19 f: cnffsfffzhfnsffttms' '8-11 c: tzvtwncnwvwttp' '8-10
r: rwrrtrvttrrrr' '1-2 p: zhpjph' '4-6 l: pldnxv'
```

Whilst it's not CSV, it *does* look tabular. One handy use of ⎕CSV is to rely on it to make a table from data, and convert columns of numbers. In our case, we separate each item by converting each "non-word" character to a comma, and then ask ⎕CSV to do the conversion:

```
⎕CSV('\W+'⎕R','⊢data)''4
```

3	6	s	ssdsssss
17	19	f	cnffsfffzhfnsfftms
8	11	c	tzvtwncnwvwttp
8	10	r	rwrrtrvttrrrr
1	2	p	zhpjph
4	6	l	pldnxv

We called ⎕CSV monadically, with an argument vector containing three items. The first, in our case, is obviously the data itself. The second is called the *data description*, which is used to specify the encoding where necessary. Dyalog's docs tells us:

> If omitted or empty, the file encoding is deduced

The 4 at the end of that is the *column specifier*, and this is what Dyalog's docs have to say about it:

> 4: The field is to be interpreted numeric data but invalid numeric data is tolerated. Empty fields and fields which cannot be converted to numeric values are returned instead as character data.

We can read files containing CSV data directly with ⎕CSV, in which case the first element of the argument vector should be a character vector containing the file name. We can also use ⎕CSV to *write* .csv files – consult Dyalog's docs for details.

Reading JSON: ⎕JSON

Dyalog also has a function for reading (and writing) JSON, appropriately enough called ⎕JSON. The JSON data interchange format crops up everywhere: usually in REST-like APIs as the payload, as config files, as the document format in document-oriented databases. The name JSON is short for *JavaScript Object Notation*, and as one can expect, dealing with JSON in JavaScript is both trivial and convenient. Python's native dictionary and list types - through luck, mostly - map pretty closely to JSON, too, making working with JSON in Python pretty simple.

In APL, we have a wealth of different data types: array. JSON doesn't do arrays of rank > 1. This makes a recursively defined format depending on dictionaries, like JSON, potentially more awkward to deal with. In the specific case of Dyalog APL, we do have the *namespace* which can be persuaded to act a bit like a dictionary.

> **Warning**
>
> Dyalog notes:
>
> JSON supports a limited number of data types and there is not a direct correspondence between JSON and APL data structures. In particular:
>
> - JSON does not support arrays with rank > 1.
> - The JSON standard includes Boolean values true and false which are distinct from numeric values 1 and 0, and have no direct APL equivalent.
> - The JSON5 standard includes numeric constants Infinity, -Infinity, NaN and -NaN which have no direct APL equivalent.
> - JSON objects are named and these might not be valid names in APL.

Whilst the ⎕JSON function does a good job given these constraints, working with JSON data (IMHO) in Dyalog always end up feeling a smidge gritty compared with JavaScript or Python.

Let's look at some examples:

```
json ← '{"key": 1, "list": [1, 2, 3, {"colour": "red", "shape": "oblong", "number": 5},
[98, 43, 77]]}'
```

Yep, that's JSON alright. Let's convert that to APL as vectors and namespaces:

```
⍎ ← data ← ⎕JSON json
```

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

Ok, that went through without complaints, and we can see that Dyalog (correctly) thinks that the top level is "object". We can now get at the innards of this by attribute names and indices:

```
data.key
data.list
data.list[3].shape
```

```
1
```

1	2	3	#.[JSON object].[JSON object]	98 43 77

```
oblong
```

Well, that looks perfectly smooth, right? But the unsmooth bit here is that in a JSON dict, the keys are *strings*, not object instance attributes which is what they end up as in our namespace. If we're reading an unknown bit of JSON data where we don't already know what the layout is, how do we, for example, process each key's value in turn if the keys aren't known to us?

We can list all keys in a namespace using the following construct, where ⎕NL is the *Name list* and the ¯2 magic number means that we want a vector back:

```
data.⎕NL¯2
```

key	list

To get the corresponding values, we need to *evaluate* each such key:

```
data.(⍎¨⎕NL¯2)
```

1	
	1 2 3 #.[JSON object].[JSON object] 98 43 77

Let's say that we want to sum all numbers in the JSON. Here's one way we can achieve that:

```
]dinput
Values ← {
    keys ← ⍵.⎕NL¯2
    values ← keys {0=≠⍺:⍬⋄⍵.(⍎¨⍺)} ⍵
    0=≠⍵.⎕NL¯9:values          ⍝ No nested namespaces: done
    values,∊∇¨⍵.(⍎¨⎕NL¯9)      ⍝ Also expand any namespaces we found as values
}
```

```
]dinput
JSONSum ← {
    {(1=2|⎕DR⍵)⍵}⍵:⍵                      ⍝ Are we a number?
    (⍬≡⊢)⍵:0                               ⍝ Are we a string?
    {(326=⎕DR⍵)∧(0=≡)⍵} ⍵:+/∇Values ⍵      ⍝ Are we an object?
    +/∊∇¨⍵                                 ⍝ We're a list
}
```

```
JSONSum data
```

```
230
```

The bits to find the type of each field in JSONSum are all things you can just look up on APLCart: monadic ⎕DR means *Data representation*, and you just need to feed it the right magic number.

There is a lot more to say about the ⎕JSON function that we won't go into here (we'll see a bit more of it in the next chapter). It can also convert JSON to a pure array format instead of using namespaces, and it can also be used to convert APL arrays *to* JSON. The interested reader will have plenty to go at in the Dyalog docs, and Morten's webinar signposted at the top of this chapter.

HttpCommand

> Ignorance more frequently begets confidence than does knowledge: it is those who know little, not those who know much, who so positively assert that this or that problem will never be solved by science. –*Charles Darwin*

```
⎕IO ← 0
]box on
]rows on
```

Ah yes, the web. I'm sure you've heard of it. Dyalog has a nifty http client library built in, called `HttpCommand`. In order to make us of it, we first need to load it up:

```
hc ← ⎕SE.SALT.Load'HttpCommand'
```

This loads the `HttpCommand` class, calling it `hc`. We could also have used the Dyalog user command `]load HttpCommand`, which loads it as `HttpCommand` – but who wants to type all that? The above approach is also usable programmatically.

Kanye.rest

There is a handy web service delivering random Kanye West quotations we can put to good use, `kanye.rest`, to demonstrate this. Kanye as a Service?

```
⍝ ← resp ← hc.Get 'https://api.kanye.rest/'
```

```
[rc: 0 | msg: "" | HTTP Status: 200 "OK" | ⍴Data: 25]
```

What did we get back? Let's look in the headers to start with:

```
{(⍵[;0]∊⊂'Content-Type')⌿⍵} resp.Headers
```

Content-Type	application/json

We know JSON; good. Let's unpack that.

```
body ← ⎕JSON resp.Data
```

We can use the handy user command `]map` to show what the inside of a namespace looks like (and yes, there is no way you'd ever discover its existence without being told):

```
]map body  ⍝ list the fields
```

```
#.[JSON object]
·   ~ quote
```

So the payload we're interested in is in the `quote` field of the `body` namespace:

```
body.quote
```

```
Manga all day
```

If we already know we're dealing with JSON, there is a handy shortcut, called `GetJSON`:

```
(hc.GetJSON 'GET' 'https://api.kanye.rest/').Data.quote
```

```
There are people sleeping in parking lots
```

`GetJSON` will do a couple of things for us behind the scenes. It will unpack the JSON body of the http response. If we're POSTing to the url, it will also treat a parameter namespace as the JSON body of the *request*; we'll look at that in more detail below. One last pearl of wisdom from Kanye:

```
(hc.GetJSON 'GET' 'https://api.kanye.rest/').Data.quote
```

```
I feel calm but energized
```

Thanks for that, Kanye. Here's the seminal Gold Digger, feat. Jamie Foxx. If you're easily offended by colorful language, that's probably not a choon for you.

A more complex API

Anyway. Back to APL. Let's look at a more complex API to examine a large data set. The Cloudant database `https://skruger.cloudant.com/airaccidents` contains a large data set drawn from the FAA, listing air accident reports. Cloudant is a db as a service running the open source CouchDB database, a JSON-over-HTTP distributed document store. Let's pull a few documents from it and see what they look like.

We're going to hit the `_all_docs` endpoint, but as this is a large database, we only want to fetch a few documents. In order to do so, we pass the parameters `limit=3` and `include_docs=true` on the URL.

```
url ← 'https://skruger.cloudant.com/airaccidents/_all_docs'
(params←⎕NS⍬).(include_docs limit) ← 'true' 3
```

```
resp ← hc.Get url params
```

We know it's going to be JSON, as everything in CouchDB is JSON.

```
body ← ⎕JSON resp.Data
]map body
```

```
#.[JSON object]
·   ~ offset rows total_rows
```

For the `_all_docs` API endpoint, the data is returned under `rows`, and if we set the `include_docs` parameter, each of those entries will have a `doc` field, containing the document itself. Let's look at the first one.

```
]map body.rows[0].doc
```

```
#.[JSON object].[JSON object].[JSON object]
·   ~ Country Latitude Location Longitude Make Model Schedule _id _rev ΔAccidentΔ32ΔNumber ΔAircraftΔ32ΔCategory ΔAi
```

Yuck. What *is* that? So the JSON field names aren't valid APL names, meaning Dyalog had to mangle them when converting to namespaces. We *can* read them like that if we want to, for example

```
body.rows[0].doc.ΔEventΔ32ΔDate
```

```
10/10/1982
```

but it sure hurts the eyes. A handy trick if we want to quickly peer into a nested JSON namespace thing is to… turn it back into JSON, but nicer:

```
1(⎕JSON⍠'Compact' 0)body.rows[0].doc
```

```
{
  "Country": "United States",
  "Latitude": "",
  "Location": "1/4NM S. OF PEO, OR",
  "Longitude": "",
  "Make": "BELLANCA",
  "Model": "7GCBC",
  "Schedule": "",
  "_id": "42788bb50806d9cc7770d46953000c99",
  "_rev": "1-0f7d34e40be0c3d4db805cf52c4adf42",
  "Accident Number": "SEA83FYM01",
  "Aircraft Category": "Airplane",
  "Aircraft Damage": "Substantial",
  "Airport Code": "",
  "Airport Name": "",
  "Air Carrier": "",
  "Amateur Built": "No",
  "Broad Phase of Flight": "CLIMB",
  "Engine Type": "Reciprocating",
  "Event Date": "10/10/1982",
  "Event Id": "20020917X05139",
  "FAR Description": "Part 91: General Aviation",
  "Injury Severity": "Fatal(1)",
  "Investigation Type": "Accident",
  "Number of Engines": "1",
  "Publication Date": "10/10/1983",
  "Purpose of Flight": "Personal",
  "Registration Number": "N57457",
  "Report Status": "Probable Cause",
  "Total Fatal Injuries": "1",
  "Total Minor Injuries": "1",
  "Total Serious Injuries": "0",
  "Total Uninjured": "0",
  "Weather Condition": "VMC"
}
```


As before, we could have used `GetJSON` instead, and we can utilize the scalar extension behavior of arrays of namespaces to pick out all the embedded docs in the `rows` field of the CouchDB `_all_docs` response.

A quirk with `GetJSON` if you're used to, say, Python's `requests` library, is that it will encode any parameters given as JSON and pass those in the request body, which isn't going to work against the CouchDB API, so we need to tag on the parameters on the URL ourselves first:

```
(hc.GetJSON 'GET' (url,'?include_docs=true&limit=3')).Data.rows.doc
```

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

The other option is to convert the data to an array instead:

```
□ ← body ← □JSON⍠'M' ⊢ resp.Data
```

0			1
1	total_rows	68389	3
1	offset	0	3
1	rows		2
2			1
3	id	42788bb50806d9cc7770d46953000c99	4
3	key	42788bb50806d9cc7770d46953000c99	4
3	value		1
4	rev	1-0f7d34e40be0c3d4db805cf52c4adf42	4
3	doc		1
4	_id	42788bb50806d9cc7770d46953000c99	4
4	_rev	1-0f7d34e40be0c3d4db805cf52c4adf42	4
4	Air Carrier		4
4	Aircraft Category	Airplane	4
4	Injury Severity	Fatal(1)	4
4	Event Id	20020917X05139	4
4	Country	United States	4
4	Airport Name		4
4	Total Fatal Injuries	1	4
4	Total Minor Injuries	1	4
4	Total Uninjured	0	4
4	Latitude		4
4	Amateur Built	No	4
4	Event Date	10/10/1982	4
4	Report Status	Probable Cause	4
4	Airport Code		4
4	FAR Description	Part 91: General Aviation	4
4	Schedule		4
4	Publication Date	10/10/1983	4
4	Longitude		4
4	Make	BELLANCA	4
4	Model	7GCBC	4
4	Engine Type	Reciprocating	4
4	Total Serious Injuries	0	4
4	Purpose of Flight	Personal	4
4	Broad Phase of Flight	CLIMB	4
4	Weather Condition	VMC	4
4	Aircraft Damage	Substantial	4
4	Accident Number	SEA83FYM01	4
4	Location	1/4NM S. OF PEO, OR	4
4	Registration Number	N57457	4
4	Number of Engines	1	4
4	Investigation Type	Accident	4
2			1
3	id	42788bb50806d9cc7770d469530012b9	4
3	key	42788bb50806d9cc7770d469530012b9	4

3	value		1
4	rev	1-4aae09af2230b2cbb6ffec5fa2767e56	4
3	doc		1
4	_id	42788bb50806d9cc7770d469530012b9	4
4	_rev	1-4aae09af2230b2cbb6ffec5fa2767e56	4
4	Air Carrier		4
4	Aircraft Category	Airplane	4
4	Injury Severity	Non-Fatal	4
4	Event Id	20020917X05078	4
4	Country	United States	4
4	Airport Name	CHEMUNG CO.	4
4	Total Fatal Injuries	0	4
4	Total Minor Injuries	0	4
4	Total Uninjured	2	4
4	Latitude		4
4	Amateur Built	No	4
4	Event Date	10/10/1982	4
4	Report Status	Probable Cause	4
4	Airport Code	ELM	4
4	FAR Description	Part 91: General Aviation	4
4	Schedule		4
4	Publication Date	10/10/1983	4
4	Longitude		4
4	Make	GLOBE	4
4	Model	SWIFT GC-1B	4
4	Engine Type	Reciprocating	4
4	Total Serious Injuries	0	4
4	Purpose of Flight	Personal	4
4	Broad Phase of Flight	APPROACH	4
4	Weather Condition	VMC	4
4	Aircraft Damage	Substantial	4
4	Accident Number	NYC83LA007	4
4	Location	ELMIRA, NY	4
4	Registration Number	N50BS	4
4	Number of Engines	1	4
4	Investigation Type	Accident	4
2			1
3	id	42788bb50806d9cc7770d46953001c8c	4
3	key	42788bb50806d9cc7770d46953001c8c	4
3	value		1
4	rev	1-0b3960a8c809e5b33b0cfb122c2fb7bb	4
3	doc		1
4	_id	42788bb50806d9cc7770d46953001c8c	4
4	_rev	1-0b3960a8c809e5b33b0cfb122c2fb7bb	4
4	Air Carrier		4
4	Aircraft Category	Airplane	4
4	Injury Severity	Non-Fatal	4

4	Event Id	20020917X05079	4
4	Country	United States	4
4	Airport Name	SULLIVAN CO. INT'L	4
4	Total Fatal Injuries	0	4
4	Total Minor Injuries	0	4
4	Total Uninjured	3	4
4	Latitude		4
4	Amateur Built	No	4
4	Event Date	10/10/1982	4
4	Report Status	Probable Cause	4
4	Airport Code	MSV	4
4	FAR Description	Part 91: General Aviation	4
4	Schedule		4
4	Publication Date	10/10/1983	4
4	Longitude		4
4	Make	CESSNA	4
4	Model	172P	4
4	Engine Type	Reciprocating	4
4	Total Serious Injuries	0	4
4	Purpose of Flight	Personal	4
4	Broad Phase of Flight	MANEUVERING	4
4	Weather Condition	VMC	4
4	Aircraft Damage	Substantial	4
4	Accident Number	NYC83LA008	4
4	Location	GRAHAMSVILLE, NY	4
4	Registration Number	N54177	4
4	Number of Engines	1	4
4	Investigation Type	Accident	4

Which is actually quite suitable for this data – the documents are completely flat. The documents themselves are at "depth 4", as indicated by the first column.

The database has a few handy indexes, too, which in CouchDB-speak is called *views*. Let's look at a couple of those. The first view allows us to fetch documents based on the make of aircraft. Here's the first entry in the index where the make of the plane involved was a Cessna:

```
url ← 'https://skruger.cloudant.com/airaccidents/_design/make/_view/by-make'
(params ← ⎕NS⍬).(limit reduce key) ← 1 'false' '"Cessna"'
resp ← hc.Get url params
⎕ ← body ← ⎕JSON⍠'M' ⊢ resp.Data
```

0			1
1	total_rows	68387	3
1	offset	13905	3
1	rows		2
2			1
3	id	5b97c6d78b17b37ceff620baf9657693	4
3	key	Cessna	4
3	value	1	3

This is a materialised view, keyed on make. The view iteself contains no particularly interesting information beyond the document id and the "value" 1. We can fetch this document by the id:

```
url ← 'https://skruger.cloudant.com/airaccidents/5b97c6d78b17b37ceff620baf9657693'
resp ← hc.Get url
⍬ ← body ← ⎕JSON⍠'M' ⊢ resp.Data
```

0			1
1	_id	5b97c6d78b17b37ceff620baf9657693	4
1	_rev	1-8d2e3c5096718be0236ef51ff7e0f153	4
1	Air Carrier		4
1	Aircraft Category		4
1	Injury Severity	Non-Fatal	4
1	Event Id	20080611X00825	4
1	Country	United States	4
1	Airport Name	PORTLAND-TROUTDALE	4
1	Total Fatal Injuries		4
1	Total Minor Injuries		4
1	Total Uninjured	1	4
1	Latitude	45.549444	4
1	Amateur Built	No	4
1	Event Date	05/24/2008	4
1	Report Status	Probable Cause	4
1	Airport Code	TTD	4
1	FAR Description		4
1	Schedule		4
1	Publication Date	06/30/2008	4
1	Longitude	-122.401389	4
1	Make	Cessna	4
1	Model	172P	4
1	Engine Type	Reciprocating	4
1	Total Serious Injuries		4
1	Purpose of Flight	Instructional	4
1	Broad Phase of Flight		4
1	Weather Condition	VMC	4
1	Aircraft Damage	Substantial	4
1	Accident Number	LAX08CA156	4
1	Location	Troutdale, OR	4
1	Registration Number	N62348	4
1	Number of Engines	1	4
1	Investigation Type	Accident	4

but perhaps more interesting is that we can do aggregations if we enable the `reduce` part of the view. We can also exploit the CouchDB API a bit further by using a POST instead, noting that we again treat the body and URL parameters separately. Let's say we want to find the accident distribution, per make, for a make subset:

```
url ← 'https://skruger.cloudant.com/airaccidents/_design/make/_view/by-make?group=true'
(params ← ⎕NS⍬).keys ← 'Cessna' 'Boeing' 'Airbus' ⍝ Request body payload; will be JSON-
encoded
body ← (hc.GetJSON 'POST' url params).Data
```

As we're now relying on `GetJSON` to encode our parameter list, we no longer need the ugly double-quotes in our list of keys.

Reductions in CouchDB views are similar to reduces in APL. All we did there was a `+/` over the values in the view, which as we saw earlier was a "1", grouping by key:

```
1(⎕JSON⍠'Compact' 0)body.rows
```

```
[
  {
    "key": "Cessna",
    "value": 7728
  },
  {
    "key": "Boeing",
    "value": 771
  },
  {
    "key": "Airbus",
    "value": 13
  }
]
```

Now we're running the risk of making this about the CouchDB API, but this is quite an interesting data set. I made a video a long time ago about it and how to process the data with map-reduce using CouchDB, and this was all inspired by a very old blog post from Cloudant founder, Mike Miller.

Other useful bits

You can pass a left argument 1 to `HttpCommand`'s functions to inspect what the request would have looked like had it been issued:

```
1 hc.GetJSON 'GET' 'https://api.kanye.rest/'
```

```
GET / HTTP/1.1
Host: api.kanye.rest
Content-Type: application/json
User-Agent: Dyalog/Conga
Accept: */*
Accept-Encoding: gzip, deflate
```

HttpCommand will strip basic auth params passed on the URL and turn them into a header instead:

```
1 hc.Get 'https://username:password@example.com'
```

```
GET / HTTP/1.1
Host: example.com
User-Agent: Dyalog/Conga
Accept: */*
Accept-Encoding: gzip, deflate
Authorization: Basic dXNlcm5hbWU6cGFzc3dvcmQ=
```

Web scraping

The Dyalog student competition in 2020 had a web-scraping problem set, problem 3, from Phase 2, asking us to find all URLs referencing PDF-files off the competition website, https://www.dyalog.com/student-competition.htm. There will be spoilers here, so if you want to have a go yourself, stop reading here.

Still here? The suggestion is that we process the data as XML (*sigh*). Let's grab that page and see what we can find:

```
page ← (_←hc.Get 'https://www.dyalog.com/student-competition.htm').Data
```

```
xml ← ⎕XML page
```

What we get back from `⎕XML` is a matrix with columns for depth, tag, content, attribute and type. We care only about the tag and attribute columns:

```
(tags attributes) ← (⊂1 3)⌷⍉⍪xml
```

To pick out URLS, we need to look at the anchor tags:

```
anchors ← ((,'a')∘≡¨tags)/attributes
```

Each such tag has a set of attribute key-value pairs. Let's grab those:

```
(names vals) ← ↓⍉⊃⍪/anchors
```

Now we can look through the attribute values to find things that end in `.pdf`:

```
pdfs ← vals/⍨{'.pdf'∘≡¯4↑⍵}¨vals
↑3↑pdfs
```

```
uploads/files/student_competition/2020_problems_phase1.pdf
uploads/files/student_competition/2020_problems_phase2.pdf
uploads/files/student_competition/2019_problems_phase1.pdf
```

As we can see, these are all relative URLs. To convert to absolute, we need to extract the base, and prepend that:

```
□ ← base ← ⊃⌽⊃('base'∘≡¨tags)/attributes
```

```
https://www.dyalog.com/
```

```
↑3↑base∘,¨pdfs
```

```
https://www.dyalog.com/uploads/files/student_competition/2020_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2020_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2019_problems_phase1.pdf
```

Putting it all together, we get something like

```
]dinput
PastTasks ← {
    (tags attributes) ← (⊂1 3)⌷↓⍉⊃XML(hc.Get ⍵).Data
    (names vals) ← ↓⍉⊃⍪/((,'a')∘≡¨tags)/attributes ⍝ Names and values of attributes of
anchor tags
    pdfs ← vals/⍨{'.pdf'∘≡¯4↑⍵}¨vals
    base ← ⊃⌽⊃('base'∘≡¨tags)/attributes
    base∘,¨pdfs
}
```

Here's what we get:

```
↑PastTasks 'https://www.dyalog.com/student-competition.htm'
```

```
https://www.dyalog.com/uploads/files/student_competition/2020_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2020_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2019_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2019_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2018_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2018_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2017_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2017_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2016_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2016_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2015_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2015_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2014_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2014_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2013_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2013_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2012_problems.pdf
https://www.dyalog.com/uploads/files/student_competition/2011_problems.pdf
https://www.dyalog.com/uploads/files/student_competition/2010_problems.pdf
https://www.dyalog.com/uploads/files/student_competition/2009_problems.pdf
```

For the sake of completeness, we could equally have solved this with some filthy regexing if we were so inclined:

```
]dinput
PastTasksRE ← {
    body ← (hc.Get ⍵).Data
    pdfs ← '<a href="(.+?\.pdf)"'□S'\1'⊢body
    base ← '<base href="(.+?)"'□S'\1'⊢body
    base,¨pdfs
}
```

```
↑PastTasksRE 'https://www.dyalog.com/student-competition.htm'
```

```
https://www.dyalog.com/uploads/files/student_competition/2020_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2020_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2019_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2019_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2018_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2018_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2017_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2017_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2016_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2016_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2015_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2015_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2014_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2014_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2013_problems_phase1.pdf
https://www.dyalog.com/uploads/files/student_competition/2013_problems_phase2.pdf
https://www.dyalog.com/uploads/files/student_competition/2012_problems.pdf
https://www.dyalog.com/uploads/files/student_competition/2011_problems.pdf
https://www.dyalog.com/uploads/files/student_competition/2010_problems.pdf
https://www.dyalog.com/uploads/files/student_competition/2009_problems.pdf
```

So which one is better? I wrote a bit on this topic on my guest blog post on Dyalog's blog. In this case, as we *know* that the page is valid XML, we can delegate a lot of complexity to the `⎕XML` function, such as different quotes, whitespace etc, which we'd need to be explicit about in anything regexy if we wanted it to be robust. However, regular expressions are hard to beat when looking for complex patterns in textual data. If the page had *not* been correct XML, it would have been a lot harder solving this problem without reaching for regular expressions.

The dfns workspace

> Computers are like Old Testament gods; lots of rules and no mercy. –*Joseph Campbell*

Dyalog ships with a workspace called dfns. It's perhaps best viewed as a set of diverse examples of … stuff. Hard to explain – maybe think of it as John Scholes' scrap book, interesting snippets of APL code that may (or may not) be useful to other APLers. No other language I'm aware of has something like this – a folklore snapshot. The oral tradition of the language, written down.

As a "standard library" it's hopelessly badly organized - so let's not call it that, and it's not the intention anyway. Yet it contains some real gems, and it's outstandingly well documented, most entries accompanied by substantial essays. It contains interpreters for several languages, including Lisp, tons of utility routines, a pretty comprehensive set of functions for manipulating graphs, various non-trivial data structures (avl, splay, redblack) etc etc.

For the learner, perhaps the most productive use of the dfns ws is as inspiration: browsing through it, learning from its examples and well-documented implementations. I thought I'd highlight some of the things I've discovered in there and found useful.

iotag

A very useful little function is iotag, for `iota-generalized`, although I can't help but read it `io-tag`. It fleshes out the built-in `iota`, `⍳`, both in its monadic and dyadic forms. Let's look at a few examples:

```
'iotag' ⎕CY 'dfns' ⍝ Load the iotag function
```

Inclusive range `start` to `end`:

```
10 iotag 20 ⍝ α to ω inclusive range
```

```
10 11 12 13 14 15 16 17 18 19 20
```

…and which can also count down:

```
20 iotag 10
```

```
20 19 18 17 16 15 14 13 12 11 10
```

and is fine with negative numbers, too:

```
¯10 iotag ¯20
¯20 iotag ¯10
```

```
¯10 ¯11 ¯12 ¯13 ¯14 ¯15 ¯16 ¯17 ¯18 ¯19 ¯20
¯20 ¯19 ¯18 ¯17 ¯16 ¯15 ¯14 ¯13 ¯12 ¯11 ¯10
```

We can also specify a step-size:

```
10 iotag 21 2
```

```
10 12 14 16 18 20
```

which doesn't even have to be integer

```
10 iotag 21 0.53
```

```
10 10.53 11.06 11.59 12.12 12.65 13.18 13.71 14.24 14.77 15.3 15.83 16.36 16.89 17.42 17.95 18.48 19.01 19.54 20.07
```

It also understands letters:

```
'Q' iotag 'H'
```

```
QPONMLKJIH
```

It can do many, many more things, also generalizing the `index-of` aspect of *iota*.

segs

The segs function splits a character vector on a set of separator chars. Whilst the implementation is simple enough to memorize – and easy enough to derive – it's a handy tool to have access to.

```
'segs' ⎕CY 'dfns'
```

```
',;: ' segs 'one,two;three four,;:five,six,'
```

one	two	three	four	five	six

There isn't much to this:

```
',;: ' {(~ω∈α)⊆ω} 'one,two;three four,;:five,six,'
```

one	two	three	four	five	six

cmat and pmat

cmat and pmat generates combinations and permutations respectively.

```
'cmat' 'pmat' ⎕CY 'dfns'
```

Starting with `cmat`, what unique ways can we combine three numbers out of 1, 2, 3, 4 and 5?

```
3 cmat 5
```

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

Or 3 out of 5 letters?

```
↓3{ω[α cmat⍴ω]}'abcde'
```

abc	abd	abe	acd	ace	ade	bcd	bce	bde	cde

Note that `cmat` produces results that are lexicographically ordered.

Moving on to `pmat` we generate *permutations* instead – given an integer right argument, produce all the different ways that many items can be ordered.

```
pmat 4
```

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

span, stpath and scc

There is also an excellent set of routines for graph manipulation, path finding and searching. Whilst a full intro to those is beyond the scope I had in mind for this, we can show some simple examples that perhaps can serve as a jump-off point. The documentation is comprehensive and excellent.

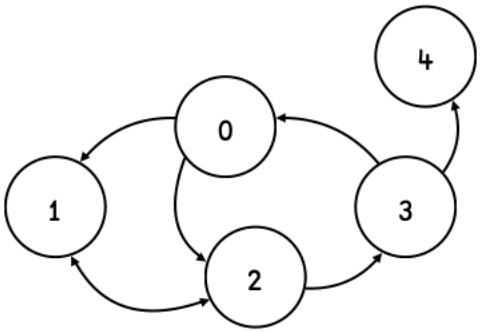
When we deal with graphs, we want to separate the graph *structure* from the content. The graph routines in the dfns ws only deal with graph structure, so it's up to you to relate that back to content. We'll look at how this is done.

So a graph consists of a set of vertices, connected by edges that have a direction. We represent the connectivity as a vector of vectors, where the position is the "vertex id" and the content a list of other vertices reachable from here. So, for example, the graph `g`

```
⎕IO←0
g ← ⎕ ← (1 2)(,2)(1 3)(0 4)(,0)
```

1 2	2	1 3	0 4	

could be visualized like so:



We can easily create a separate vector that holds each vertex's contents if we wanted to, say

```
vertices ← 'adam' 'bob' 'charlotte' 'damian' 'erica'
```

Let's load up a few of the graph functions:

```
'span' 'stpath' 'scc'⎕CY 'dfns'
```

...and a little helper function to show graph structure:

```
show←{↑↓¨(⍳⍴⍵),¨'→',¨⍵}
```

```
show g
```

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

The function span finds the spanning tree of a graph, which is basically removing any cycles whilst not making any previously reachable vertices unreachable. So let's find the (actually, one of potentially several) spanning tree for our graph g, rooted at vertex 0, using the function called span:

```
st ← g span 0
```

Now that we have a spanning tree, we can find shortest paths in the graph. For example, what's the shortest path from adam (vertex 0) to erica (vertex 4)? For this we apply the function stpath, *spanning tree path*:

```
vertices[st stpath 4]
```

adam	charlotte	damian	erica

By pre-calculating the spanning tree, we can do quick path finding to any other node in the graph. Note though that this approach will only find one shortest path, even if several of equal length might exist.

The implementations for span and stpath are worth studying. Note, however, that for many path-finding applications in graphs, as the graph gets bigger there are other algorithms worth considering, like Dijkstra's shortest path, or its heuristic cousin, A*, that would cut the search space.

Strongly Connected Components

The function scc implements Tarjan's algorithm for finding the strongly connected components of a graph. The strongly connected components are groups of interconnected vertices, so for example a graph defined by the following adjacency index vector

```
⎕IO ← 1
 ← graph ← (,2) (1 8) 0 (,10) (,9) (,10) (,9) (,2) (5 7) (4 6)
```

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

has the following strongly connected components (graph has vertices labeled ⍳10):

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

Note how vertex 3 is not connected to any other vertices and so forms its own connected component. We can use scc directly on the graph adjacency index vector to label each vertex with a connected component index:

```
]display ↑(⍳10) (scc graph)
```

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

In the dfns ws there are also graph algorithms operating on weighted graphs – where each edge has a cost that can differ from 1.

If you're interested in using APL for graphs, there is a complete example provided for route planning in various tube systems of world cities for you to study.

Testing

> Reminds me of the awesome bug report I saw once: 'Everything is broken. Steps to reproduce: do anything. Expected result: it should work'. –*Felipe Knorr Kuhn*


```
⎕IO ← 0
]box on
]rows on
```

APL programmers don't need to test their code, as they always write correct code. Next chapter…

Ahem. Ninja master @ngn offered up the following unit testing framework on APL Orchard:

```
≡ ⍝ usage: expectedoutput ≡ f input.  prints 1 for ok and 0 for failure
```

```
≡
```

Whilst @ngn was (at least partially) joking, he's got a point.

There is no official testing framework blessed by Dyalog. However, APL programmers do of course test their code. We can learn a lot from Dyalog's published source code. Here, for example, is the test suite for the Link package. This is a fairly hefty namespace running to 2k+ LOC. If the code looks unfamiliar, it's because it's written in the tradfn style.

A 'framework'?

No. But let's look at how unit testing is handled in other languages. In Python there are (too) many testing frameworks to choose from, all different. The original unit testing framework, included in Python by default, is the imaginatively named unittest module. `unittest` talks about `test suites` comprising of related `test cases`, controlled by a `runner` and context managed by `fixtures`. There is no reason why we couldn't have something similar in APL if we wanted to build such a thing. But we probably want to find a more APL-y way of doing that.

Here's what I'd want from my testing system:

1. Ability to automatically run all tests, and get a report back on which tests succeeded.
2. Ability to run a single test.
3. Easy way to create more tests and have them be picked up by the test runner.

Here's the first example from the `unittest` docs:

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

To add further tests, we just add methods starting with `test_` to the `TestStringMethods` class, and the `unittest.main()` method will run them for us. So for our first attempt, let's try to replicate that functionality in a Dyalog namespace. Here's our test runner, which simply executes all functions where the name starts with `test_` and produces a little report.

```
:Namespace unittest
    ⎕IO ← 0
    run←{
        tests ← 'test_.+'⎕S'&'⎕NL ¯3
        0=≢tests: 'no tests found'
        ↑{α,('.'/⍨30-≢α),ω⊃'[FAIL]' '[OK]'}⍎↑tests (⍎¨tests,¨⊂' ⍬')
    }
:EndNamespace
```

```
unittest.run⍬
```

```
no tests found
```

Let's make some functions that we can unit test. We can take the ones from the Python example above.

```
upper←1∘⎕C
isupper←{ω≡1⎕Cω}
split←≠⊆⊢          ⍝ Won't complain about a non-string separator, but hey, why should
it?
```

Now we can write the tests themselves. Note that as in this case the functions we're testing are defined outside the `unittest` namespace, so we need to prefix the calls with `#.`. Note how we're using the test framework proposed by @ngn, outlined above :)

```
unittest.test_upper←{'FOO'≡#.upper 'foo'}
```

```
unittest.run⍬
```

```
test_upper.....................[OK]
```

```
unittest.test_isupper←{(#.isupper 'FOO')∧~#.isupper 'Foo'}
```

```
unittest.run⍬
```

```
test_isupper...................[OK]
test_upper.....................[OK]
```

Let's add a test that fails.

```
unittest.test_isupper2←{(#.isupper 'FOO')∧~#.isupper 'BAR'} ⍝ Failing test
```

```
unittest.run⍬
```

```
test_isupper...................[OK]
test_isupper2..................[FAIL]
test_upper.....................[OK]
```

```
unittest.test_split←{'hello' 'world'≡' '⍎'#.split ''hello world''}
```

```
unittest.run⍬
```

```
test_isupper..................[OK]
test_isupper2.................[FAIL]
test_split....................[OK]
test_upper....................[OK]
```

We can of course run single tests trivially:

```
unittest.test_split⍬
```

```
1
```

Nice, simple, and surprisingly useful.

Data-driven testing

Data-driven testing, also known as parametrized testing, is where you provide essentially a table of inputs and expected outputs and let your testing framework run them all. This approach isn't supported out of the box in Python's basic `unittest` module. However, other more fully-featured Python frameworks, such as pytest, do.

Here's how that can look:

```
import pytest

@pytest.mark.parametrize("test_input,expected", [("3+5", 8), ("2+4", 6), ("6*9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Here the decorator `@pytest.mark.parametrize` defines the test function arguments, and then provides a list of tuples. The test runner will then call the test function with the arguments as given by each tuple in turn. Let's see if we can achieve something similar in APL.

This sounds like a job for an operator: we pass a *function* to the test runner, and a vector of 3-"tuples" representing left argument, right argument and expected outcome.

```
_test←{(⊢/↑⍵)≡⍹⍬ ⍬⊢⍺⍺/¯1↓⍹1⊢↑⍵}
```

```
split _test (' ' ('hello world') ('hello' 'world')) (',' ('hello,world') ('hello'
'world'))
```

```
1 1
```

What we need now is a convenient way to specify such parameter sets so they can be picked up by the test runner. We can do this by defining variables named `fn_testdata`, and have that be picked up by our unit testing namespace.

```
split_testdata←(' ' ('hello world') ('hello' 'world')) (',' ('hello,world') ('hello'
'world'))
```

```
split _test split_testdata
```

```
1 1
```

```
:Namespace datatest
    ⎕IO ← 0
    _test←{(⊢/↑⍵)≡⍹⍬ ⍬⊢⍺⍺/¯1↓⍹1⊢↑⍵}
    run←{ ⍝ ⍵ -- ns containing functions to be tested
        params ← '_'(≠⊆⊢)¨'[^_]+_testdata'⎕S'&'⎕NL¯2.1    ⍝ https://aplcart.info/?
q=%E2%8E%95NL#
        ⍬=≢params: 'no test parameter sets found'
        funs ← ⊃⍪¨↓¯1↓⍹1↑params                          ⍝ Corresponding functions
defined?
        testable ← funs/⍨funs∈⍵.⎕NL¯3
        result←⍵⍎¨{(⍺.⍙⍵)_test ⍙⍵,'_testdata'}¨testable    ⍝ Run the tests
        ↑{⍺,('.'/⍨30-≢⍺),'[',(⍕+/⍵),'/',(⍕≢⍵),']'}⌸↑testable result ⍝ Format
    }
:EndNamespace
```

```
datatest.split_testdata←(' ' ('hello world') ('hello' 'world')) (',' ('hello,world')
('hello' 'world')) ⍝ dyadic function
datatest.isupper_testdata←(0 ('FOO') 1) (0 ('Foo') 0) (0 (,'F') 1) ⍝ monadic function
```

```
datatest.run ⎕THIS
```

```
isupper.....................[3/3]
split.......................[2/2]
```

So there we have it. Of course, in a real project you may want a slightly more fleshed out test framework, capable of testing for exceptions etc.

A Dyalog workflow. Or two.

> I don't know how many of you have ever met Dijkstra, but you probably know that arrogance in computer science is measured in nano-Dijkstras. –*Alan Kay*

For the majority of the examples we'll encounter in this book, you can just type them directly into the Dyalog IDE, or TryAPL, and when you understand the point made, scrap them. But as you progress, sooner or later you will want to save your work to disk.

If you have a similar background to me, you'll expect to be able to write your code using an editor of your choice, arranging it into files containing logically related routines, which you then put into a source code revision management system, like git, perhaps triggering a build and test pipeline on every commit via something like jenkins or circle-ci. This is most likely in the neighbourhood of your current workflow, regardless of which language or platform you're using, from ADA to Zig.

Before we get into the practicalities of how you can do this and more, we'll need a slight diversion into the history of APL and Dyalog in particular [Kromb&Hui2020]. Firstly, APL predates all these workflows and build tools. That in itself isn't an issue: so does C. But APL wasn't really conceived as a software engineering tool. During its heyday, middle management and CFOs were slinging APL to generate forecasts and reports, and they *loved* APL as it allowed them to essentially perform tasks that up until then had been in the realm of specialist "batch processing" on the company mainframe. It essentially occupied the niche which is now dominated by Microsoft Excel.

This influenced all APL vendors. Commercial APL systems are full application stacks that make it seductively convenient to package up your work as binary workspaces that can be easily distributed to other APL users without ever having to "leave" the confines of the stack itself. In terms of Dyalog, its tooling and development philosophy supports this world view, as that is what its customers expect: professional Dyalog developers working for billion-dollar financial institutions will be of the opinion that you can pry their binary workspaces from their cold, dead hands. You can build fully-fledged, GUI-driven applications integrating deeply with code bases written in .NET.

Yet, for you, the newcomer, this approach will probably feel a bit alien. Dyalog obviously understand this, and they're walking the tightrope of both delivering what their current paying, long-term customers need, but also a keen eye on what *new* users and potential customers might be looking to. Morten Kromberg, Dyalog CTO, wrote a blog post recently, ostensibly about the APL Orchard chat room, but also touching on some of Dyalog's challenges when dealing with this dichotomy.

Fortunately, and for the avoidance of any doubt, you can *absolutely* work with modern Dyalog (I'm using version 18.0) in a pretty similar way to what you expect, but it requires a little bit of setup to get right, and is not quite there "out of the box" at the time of writing. However, at the time of *reading* some of the kinks may well have been ironed out: better support for Dyalog as a scripting engine, and working with code-as-text is a priority for Dyalog right now.

So let's take a brief look at workspaces and then at the `]LINK` tool that allows you to keep your code in text files that can be versioned.

Other resources:

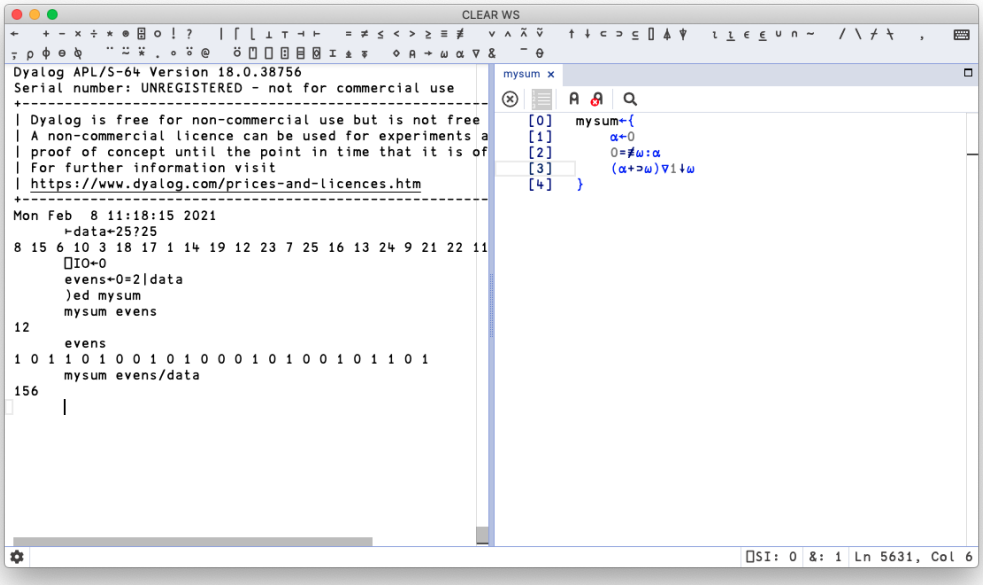
- Dyalog webinar

Saving and loading workspaces

If you open RIDE, you'll see no "Save as.." in the menu, and your OS's default key binding for "Save" does nothing. You might have figured out how to enter multi-line functions, and how to "fix" (somewhere between "save" and "compile") them. Yet, when you "fix" a function, where does it go?

At no point were you asked for a file name. The Excel analogy might help to clarify. Think of a cell containing a formula in Excel – you're not expecting to "save" the formula out to a separate file. Instead, the formula becomes an integral part of the sheet or workbook. We can think of Dyalog workspaces as analoguous to Excel workbooks: a snapshot of values and logic to manipulate them. It has the nice feature that if you distribute a workspace, you can be sure that users of that workspace sees exactly what you intended: it's an exact snapshot.

Let's try this out. In a new RIDE session, let's create a few functions and variables, like so:



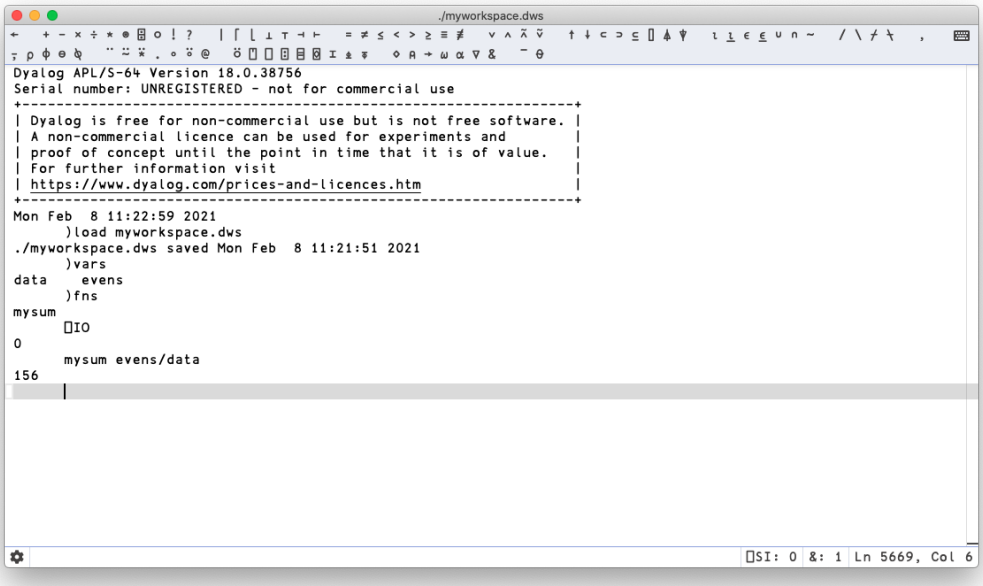
To save the workspace into a workspace file, type

```
)save myworkspace
Cannot perform operation with trace/edit windows open.
```

Ooops, that didn't work. Dyalog can only save a workspace if you close down all edit windows first. So let's do that, and once you've saved your workspace, close down the session and reopen a fresh one.

To get back your workspace, type

```
)load myworkspace
```



We can use the commands `)vars` to see the defined variables, and `)fns` to show the names of any functions contained in the workspace. As we can see, they're all present and correct. If you find yourself using a set of utility functions over and over, they might be usefully combined into a workspace that you can then load up as a unit with a simple `)load` command. As you might expect, the workspace concept is deeply engrained in Dyalog and it comes with rich, sophisticated, thoroughly battle-tested functionality – all of which is overkill for our needs right now. If you want to explore this further, Legrand's book, Mastering Dyalog APL covers it extensively.

Using LINK

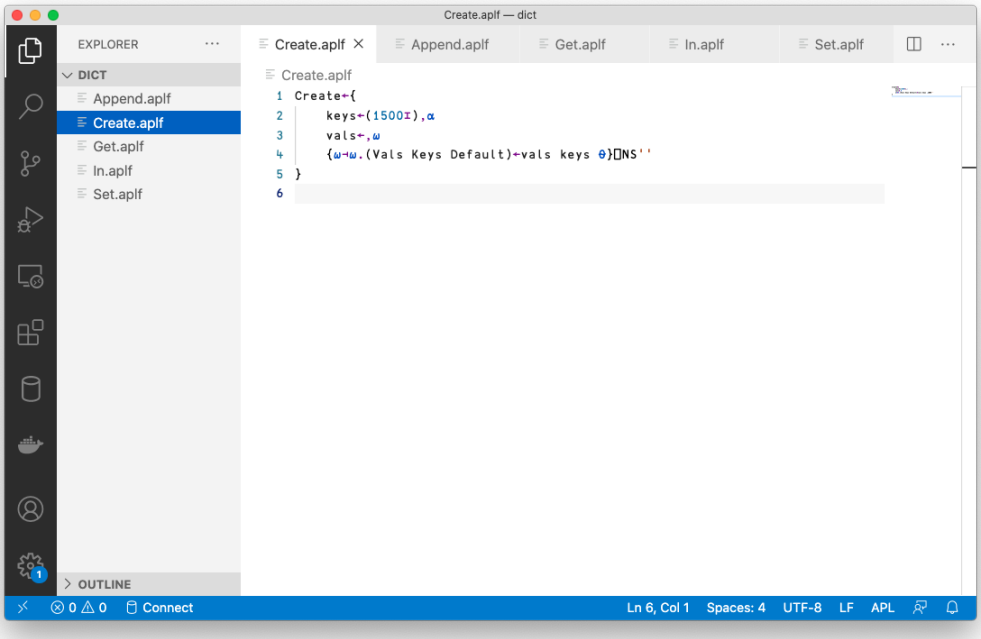
LINK is a Dyalog library that can be used to set up synchronisation between a directory of text files containing APL source code, and namespaces in your current workspace. As you may recall from the chapter on namespaces, one way to think of them is as a grouping mechanism akin to a module or class. Recent Dyalog versions come with LINK bundled, but it's under active development and you can always install the latest version from the above link.

The LINK concept is pretty intuitive: a LINKed directory becomes a namespace, and all its content loaded. If you make a change to a file using an external tool or editor, that change will be reflected inside Dyalog, and if you make a change to the namespace inside Dyalog, these changes are reflected on disk. This means that you can write APL code in Emacs or VS Code, and have your Dyalog session see any changes you make. And – of course – now you can put your code in github with all that this entails in terms of your workflow.

Let's work through a complete example, step by step, by building a "dict" – an array that gives the illusion of being indexable by string. Yes, you're not the only one wishing Dyalog had one of those. Start by creating a new directory called `dict` and make a note of the full path:

```
stefan–% mkdir –p ~/work/dyalog/dict
```

Now, using an editor of your choice, you're going to create five functions, each contained in a file with the `.aplf` extension (for APL function). Here's me editing the first one in VS Code, called `Create.aplf`:



> **Note**
>
> One thing you'll notice when editing APL code in VS Code is that as VS Code also uses the option key for certain commands, it becomes a bit more tricky to type APL glyphs. You can always use the backtick as the APL key for combos that VS Code hijacks.

Here are the functions themselves, leaving aside for the moment how they work (or indeed if they're particularly good):

```
]dinput
Append←{⎕IO←0
    ⍝ Given a vector of keys and a vector of vals append vals
    α∘{
        k←0⊃⍵ ⋄ v←1⊃⍵
        i←α.Keys⍳k
        i=≢α.Keys:1⊣α.Keys,←k⊣α.Vals,←v  ⍝ New key
        α.Vals[i],←v                      ⍝ Append to existing
        0
    }¨↓⍉↑⍵
}
```

```
]dinput
Create←{
    keys←(1500⌶),α
    vals←,⍵
    {⍵⊣⍵.(Vals Keys Default)←vals keys 0}⎕NS''
}
```

```
]dinput
Get←{
    ~⍵ In α:α.Default
    α.Vals[α.Keys⍳⍵]
}
```

```
In←{α∈⍵.Keys}
```

```
]dinput
Set←{⎕IO←0
    ⍝ Upsert ht α given a vector of keys and a vector of vals
    α∘{
        k←0⊃⍵ ⋄ v←1⊃⍵
        i←α.Keys⍳k
        i=≢α.Keys:1⊣α.Keys,←k⊣α.Vals,←v ⍝ New key
        α.Vals[i]←v                      ⍝ Replace existing
        0
    }¨↓⍉↑⍵
}
```

Now open a new Dyalog session, and link this directory:

```
]LINK.Create dict /Users/stefan/work/dyalog/dict
```

You should see the following:

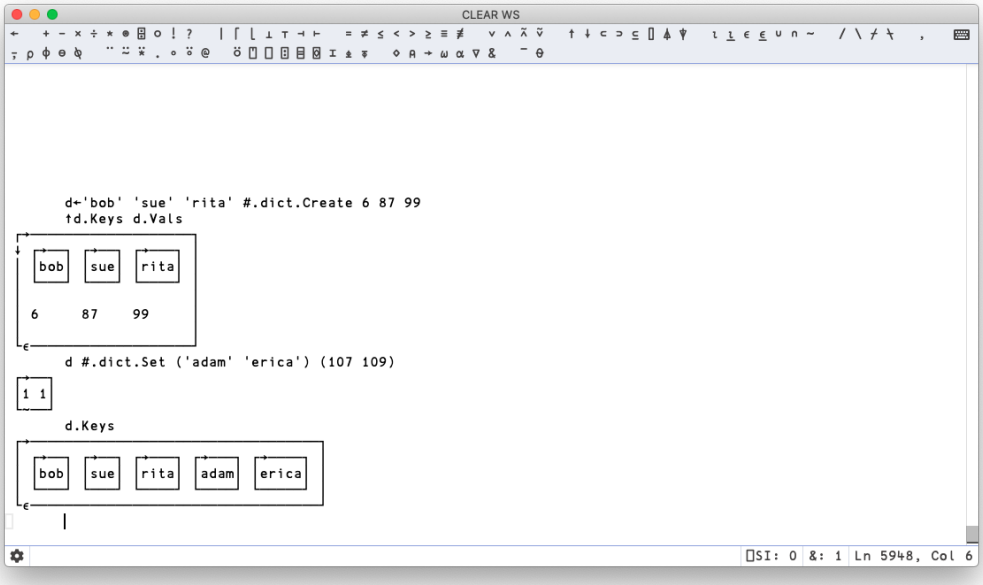
```
Linked: #.dict → /Users/stefan/work/dyalog/dict
```

If you see a pop-up window with an error message about the .NET bridge not being present, this is a known bug you can ignore, see issue 166.

All being well, our session should now be aware of the code we just wrote. We can list the functions that Dyalog now thinks the namespace contains by executing the following obvious and intuitive “list functions in namespace” command:

```
dict.⎕NL -3
Append  Create  Get  In  Set
```

Great! Our functions have clearly been loaded. Let’s see if they work, shall we?



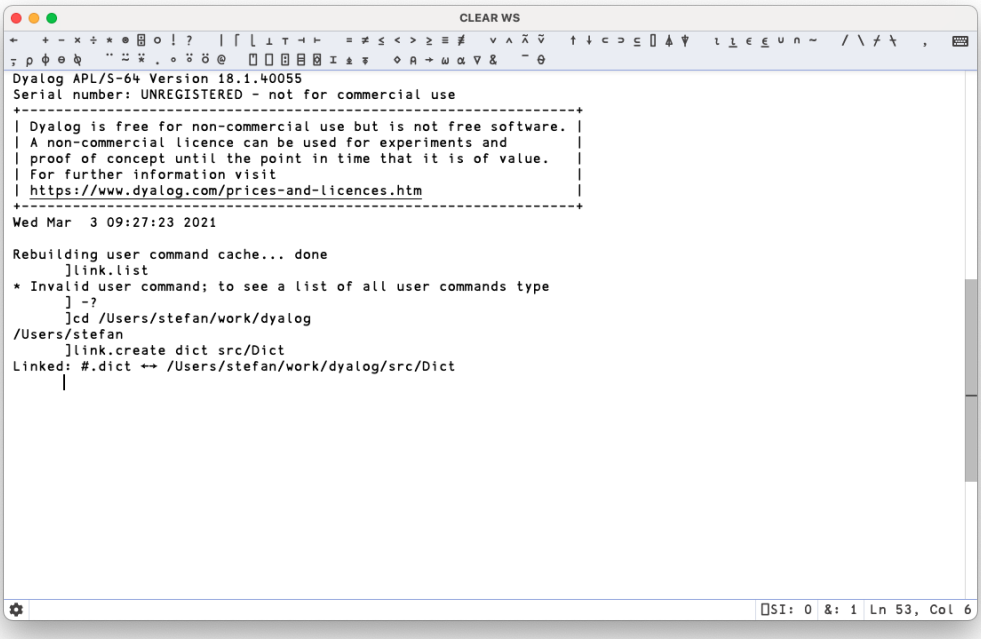
LINK limitations

That works really well so far. Any change we make in RIDE to the `dict` namespace will be reflected on disk, including new functions, or modifications of existing. However, there are a few limitations you need to be aware of here. The eagle-eyed reader may have spotted our link being one-directional only:

```
Linked: #.dict → /Users/stefan/work/dyalog/dict
```

The arrow shows synchronization direction. Bi-directional sync is really what we want, but at the time of writing this is only available under Windows. Do read the docs for LINK.

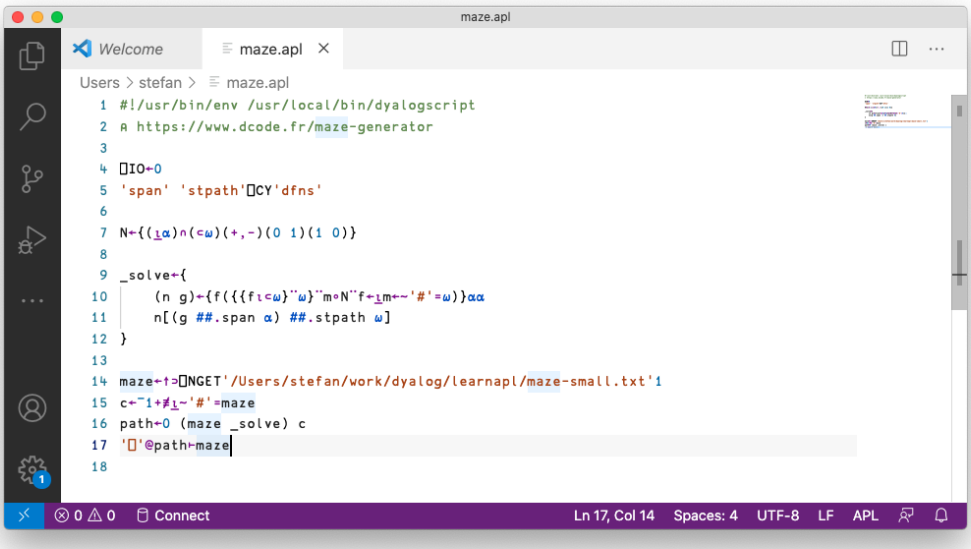
In Dyalog version 18.1 (which at the time of writing isn't formally released yet) you can get bi-directional `]LINK` on non-Windows platforms if you also install Microsoft .NET Core 3.1.



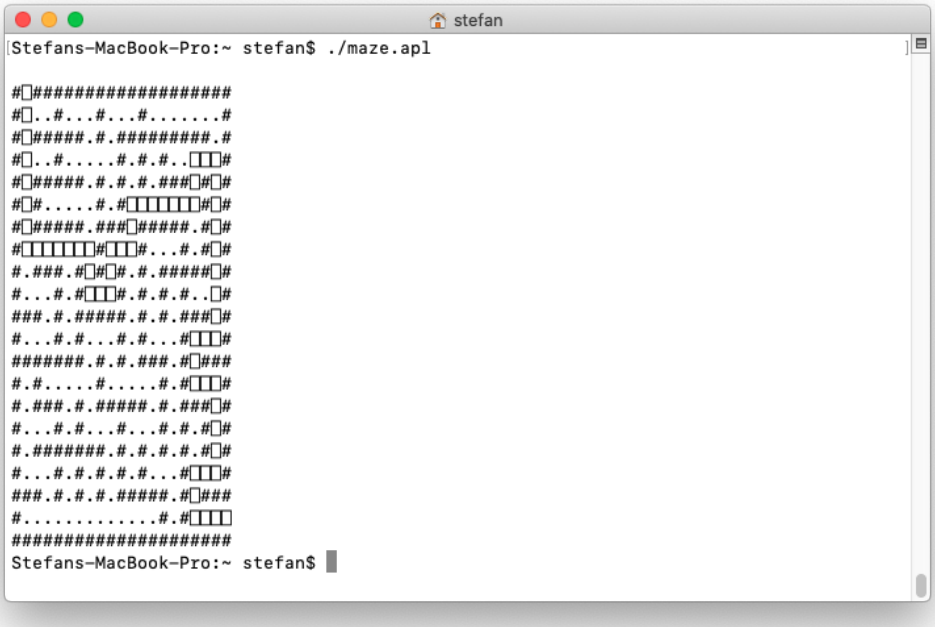
The other thing is that when you're working this way, you'll need to keep each function, namespace, class, or operator in its own separate file, and they all have their separate file extensions, which if you're anything like me is quite a fine granularity, especially since APL functions tend to be pretty short.

Dyalogscript

Also in Dyalog v18.1, a new feature called `dyalogscript` is present. For me, this is a bit of a workflow game changer. `dyalogscript` allows you to execute a text file containing APL code, top to bottom, with function definitions and all, from the command-line, following standard un×x conventions. This means that with this approach you can now use Dyalog for 'quick and dirty' scripts where you otherwise would have reached for Python, and that you can easily share code as files, or via a github gist and have other people run it. For example, I created the following file:



and can run it from the command line:



Jupyter workflow

As I've mentioned before, this book is composed of a set of Jupyter notebooks running the Dyalog kernel. APL is a pretty sharp tool for data analysis and ad-hoc modeling and experimentation, and as such fits neatly into the expectations and workflows that gave rise to Jupyter in the first place. The way I have ended up working with Dyalog APL is 75% Jupyter, 20% exploration and debugging in RIDE and 5% LINK for stuff I really think I might reuse.

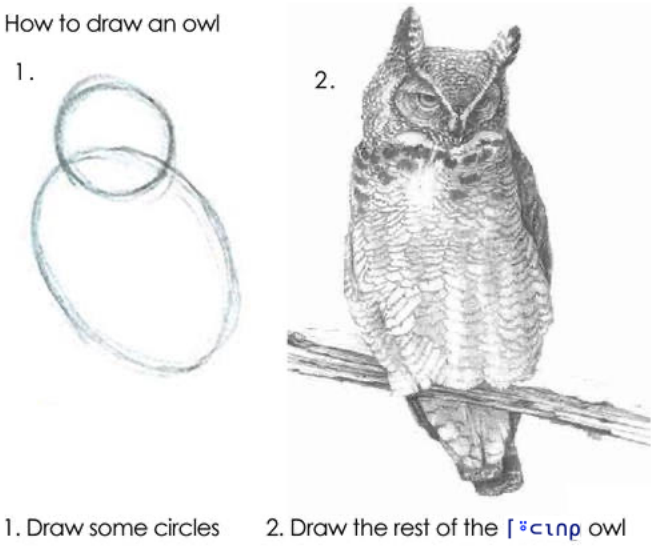
Here's a Dyalog webinar introducing the Jupyter kernel.

What now?

> Beware of Methodologies. They are a great way to bring everyone up to a dismal, but passable, level of performance, but at the same time, they are aggravating to more talented people who chafe at the restrictions that are placed on them. –*Joel Spolsky*

If you stayed with it to here, well done – you're going to Top Gun.

You should have enough skills to make a considerable dent in a problem collection such as Advent of Code, Project Euler, Perl Weekly Challenge or Project Rosalind. What you need now is practice.



A good place to start is the Dyalog Problem Solving Competition, an annual event that's been running for a while. All the old problems are available and they are typically constructed to flatter APL's capabilities.

Advent of Code is *great* – a nice ramp-up of difficulty and a Grand Tour of Computer Science 101. It's my benchmark – I find that starting on Day01 knowing near nothing, by the time I get to Day25 I am reasonably proficient in a new language. So I thought we'd solve a couple of Advent of Code problems here. Obviously, if you don't want to spoil it for yourself, come back here once you've had a go yourself.

```
⎕IO ← 0
⎕PP ← 34
]box on
]rows on
assert←{⍺←'assertion failure' ⋄ 0∊⍵:⍺ ⎕signal 8 ⋄ shy←0}
```

20/3 Toboggan Trajectory

This one is day 3 from 2020. Note that my data will be different from yours - a feature of Advent of Code.

We have a 2D array of dots and hashes. The pattern is repeated indefinitely to the right. The example given is:

```
⍞←data ← 11
11⍴'..##.......#...#...#...#....#..#...#.#...#.#.#...##..#...#.##......#.#.#....#.#....
....##.##...#...#...#...##....#.#..#...#.#'
```

```
..##.......
#...#...#..
.#....#..#.
..#.#...#.#
.#...##..#.
..#.##.....
.#.#.#....#
.#........#
#.##...#...
#...##....#
.#..#...#.#
```

which is then assumed to repeat to the right indefinitely, the first few like so:

```
data,⍪3⊢data
```

```
..##.........##.........##.........##.......
#...#...#..#...#...#..#...#...#..#...#...#..
.#....#..#..#....#..#..#....#..#..#....#..#.
..#.#...#.#..#.#...#.#..#.#...#.#..#.#...#.#
.#...##..#..#...##..#..#...##..#..#...##..#.
..#.##.......#.##.......#.##.......#.##.....
.#.#.#....#.#.#.#....#.#.#.#....#.#.#.#....#
.#........#.#........#.#........#.#........#
#.##...#...#.##...#...#.##...#...#.##...#...
#...##....##...##....##...##....##...##....#
.#..#...#.#.#..#...#.#.#..#...#.#.#..#...#.#
```

The task is to calculate how many # we'd encounter if, starting from the top-left corner, we follow a path which for each iteration moves three steps to the right, and one step down. In this test case, we'd encounter 7, here indicated by quads, □:

```
11
44⍴'0.##.........##.........##.........##.......#..0#...#..#...#..#...#..#...#..#...#.
..#...#....□..#..#....#..#..#....#..#..#....#..#...#.#...#0#..#.#...#.#..#.#...#.#..#.#
...#.#.#...##..#..□...##..#..#...##..#..#...##..#...#.##.......#.□#.......#.##.......#.
##......#.#.#....#.#.#.#.0..#.#.#.#....#.#.#.#....#.#.........#.#.........□.#........#.#.
.......##.##...#...#.##...#...#.□#...#...#.##...#...#...##....##...##....##...#□....##.
..##....#.#..#...#.#.#..#...#.#.#..#...□.#.#..#...#.#'
```

```
0.##.........##.........##.........##.......
#..0#...#..#...#...#..#...#...#..#...#...#..
.#....□..#..#....#..#..#....#..#..#....#..#.
..#.#...#0#..#.#...#.#..#.#...#.#..#.#...#.#
.#...##..#..□...##..#..#...##..#..#...##..#.
..#.##.......#.□#.......#.##.......#.##.....
.#.#.#....#.#.#.#.0..#.#.#.#....#.#.#.#....#
.#........#.#........□.#.........#.#........#
#.##...#...#.##...#...#.□#...#...#.##...#...
#...##....##...##....##...#□....##...##....#
.#..#...#.#.#..#...#.#.#..#...□.#.#..#...#.#
```

We can generate the coordinates for the path directly. The y-coordinate just increases by 1. The x-coordinate increases by 3 until we "fall off" the end of the pattern, after which it resets back to zero. Let's look at this specific case.

If we ignore the wrap-around in x, we can get our coordinates as (assuming the pattern is shape 11 11):

```
↓⍉↑1 3×⊂⍳11 ⍝ Remix to create y x pairs
```

0	0	1	3	2	6	3	9	4	12	5	15	6	18	7	21	8	24	9	27	10	30

The wrap-around is just a mod by the shape of the pattern, in our case 11 11:

```
↓⍉↑11 11|1 3×⊂⍳11
```

0	0	1	3	2	6	3	9	4	1	5	4	6	7	7	10	8	2	9	5	10	8

So we pick out the chars along this vector, look for # and sum:

```
+⌿'#'=data[↓⍉↑11 11|1 3×⊂⍳11]
```

So that works on the test data at least; good. Let's make that generic in the pattern shape and the path offset:

```
data {+⌿'#'=α[↓⍉↑(⍴α)⍴⍵×⊂⌊(≠α)÷⊃⍵]} 1 3
```

```
7
```

At this point we're ready to try it on the real pattern.

```
data ← ↑⊃⎕NGET'../../AoCDyalog/data/2020/day03.txt'1
```

The real pattern is considerably larger, but as we've made our solution independent of pattern shape and path offset, this should not present a problem:

```
⍴data
```

```
323 31
```

```
⎕ ← part1 ← data {+⌿'#'=α[↓⍉↑(⍴α)⍴⍵×⊂⌊(≠α)÷⊃⍵]} 1 3
assert 203=part1
```

```
203
```

That was the correct answer for part 1 (for me – you will get a different number).

For part 2, we're given five slopes, and we need to multiply all results together. Not much left to do:

```
⎕ ← part2 ← ×⌿data∘{+⌿'#'=α[↓⍉↑(⍴α)⍴⍵×⊂⌊(≠α)÷⊃⍵]}¨(1 1)(1 3)(1 5)(1 7)(2 1)
assert 3316272960=part2
```

```
3316272960
```

So that was kind of APL home-court advantage - a nice, array-oriented solution.

16/6 Signals and Noise

This one is day 6 from 2016.

Given a bunch of random-looking strings of letters of equal lengths, we're to find the most commonly occurring letter in each position. We're given the following test set:

```
data ← 'eedadn' 'drvtee' 'eandsr' 'raavrd' 'atevrs' 'tsrnev' 'sdttsa' 'rasrtv'
'nssdts' 'ntnada' 'svetve' 'tesnvt' 'vntsnd' 'vrdear' 'dvrsen' 'enarar'
↑data
```

```
eedadn
drvtee
eandsr
raavrd
atevrs
tsrnev
sdttsa
rasrtv
nssdts
ntnada
svetve
tesnvt
vntsnd
vrdear
dvrsen
enarar
```

Picking out the most commonly occurring letter in each column spells `easter`. "Most commonly occurring" should have us reach for *Key*, ⌸. We remix the input data so that we get a vector of the columns, which we then *Key*:

```
{α,≠⍵}⌸¨↓⍉↑data
```

e 3	e 2	d 2	a 2	d 2	n 2
d 2	r 2	v 1	t 3	e 3	e 2
r 2	a 3	n 2	d 2	s 2	r 3
a 1	t 2	a 2	v 2	r 2	d 2
t 2	s 2	e 2	n 2	t 2	s 2
s 2	d 1	r 2	r 2	v 2	v 2
n 2	v 2	t 2	s 2	n 1	a 2
v 2	n 2	s 3	e 1	a 2	t 1

Now we sort each array based on the last column, and select the element at 0 0 in each, which should be the most frequent letter:

```
∊{0 0⌷ω[⍒ω[;1];]}¨{α,≠ω}⌸¨↓⍉↑data
```

```
easter
```

Looks promising. Let's try that on the real data:

```
data ← ⊃⎕NGET'../../AoCDyalog/data/2016/06.txt'1
```

```
⎕ ← part1 ← ∊{0 0⌷ω[⍒ω[;1];]}¨{α,≠ω}⌸¨↓⍉↑data
assert 'qrqlznrl'≡part1
```

```
qrqlznrl
```

Part 2 - the same, but now pick the *least* frequent. All we need to do is to swap grade down for grade up:

```
⎕ ← part2 ← ∊{0 0⌷ω[⍋ω[;1];]}¨{α,≠ω}⌸¨↓⍉↑data
assert 'kgzdfaon'≡part2
```

```
kgzdfaon
```

If we wanted to be clever, we could make that an operator:

```
_day6 ← {∊αα{0 0⌷ω[ααω[;1];]}¨{α,≠ω}⌸¨↓⍉↑ω}
```

```
⍒_day6 data
⍋_day6 data
```

```
qrqlznrl
kgzdfaon
```

18/10 The Stars Align

Back in 2018, we were treated to this gem on day 10. We have a bunch of "stars" each having a position and a velocity. At some moment in time, these stars will align, forming a message.

We can reasonably guess that the message forms when the bounding box of all the stars is at its smallest. If that *isn't* the case, then this becomes much harder, so let's try this hypothesis first.

Step one is to pick out four numbers, including negatives, from each line of the input data:

```
data ← ↑'¯?\d+'⎕S{⍎⍵.Match}¨⊃⎕NGET'../../AoCDyalog/data/2018/10.txt'1
```

The first two columns are the position, and the last two columns the velocity.

```
pos ← ,/data[;0 1]
vel ← ,/data[;2 3]
```

We want to minimise the bounding box. We can use the sum of the width and height of the bounding box as a single number that we can compare. Here's a function to compute that:

```
bbx ← {+/|-⌿2 2⍴(⌊⌿,⌈⌿)↑⍵} ⍝ Width + height of bounding box
```

So we want to repeatedly apply a function (the adding of the velocity to the position) to the output of itself until we reach a defined stopping condition – when the width plus the height of the bounding box no longer decreases. This sounds like a job for the power operator:

```
msg ← vel⍢{vel+ω}⍣{⍺>⍤bbxω}⊢pos ⍝ Add velocity until bounding box no longer
decreases.
```

That's actually it – but in order to actually *see* the message we need to prune it down to the bounding box, and for extra flair, use the ∗ character, as shown in the question.

```
(w h) ← |-/2 2⍴(⌊/,⌈/)↑msg          ⍝ Width and height of bounding box
(-h+1)(-w+1)↑'∗'@(⍸⍤msg)⊢143 203⍴' ' ⍝ Prune our display, and make stars
```

```
******  *****   *****   *****   *****   *****   ******     **
*       *    *  *    *  *    *  *    *  *    *       *   *  *
*       *    *  *    *  *    *  *    *  *    *       *  *    *
*       *    *  *    *  *    *  *    *  *    *      *   *    *
*****   *****   *****   *****   *****   *****      *    *    *
*       *       *  *    *    *  *  *    *  *      *     ******
*       *       *   *   *    *  *   *   *   *    *      *    *
*       *       *   *   *    *  *   *   *   *   *       *    *
*       *       *    *  *    *  *    *  *    *  *       *    *
*       *       *    *  *****   *    *  *    *  ******  *    *
```

17/25 The Halting Problem

The Christmas Day problem from 2017. We're asked to solve the Halting Problem for a Turing machine. We're given a description of the state transitions – the program, basically. The description for state `A` is:

```
In state A:
  If the current value is 0:
    - Write the value 1.
    - Move one slot to the right.
    - Continue with state B.
  If the current value is 1:
    - Write the value 0.
    - Move one slot to the left.
    - Continue with state C.
```

Compiling the state descriptions into a table, we get to something along the lines of

```
  0    1

A 1RB 0LC

B 1LA 1RD

C 1RA 0LE

D 1RA 0RB

E 1LF 1LC

F 1RD 1RA
```

which we can translate to a neat APL 6×6 matrix:

```
□ ← STATE ← 6 6⍴1 1 1 0 ¯1 2 1 ¯1 0 1 1 3 1 1 0 0 ¯1 4 1 1 0 0 1 1 1 ¯1 5 1 ¯1 2 1 1 3
1 1 0
```

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

We'll need a honking looooong magnetic tape, initially set to all zeros. We don't know the exact length, but we know we are running 12919244 iterations, so let's make the tape that size to start with.

```
TAPE ← 12919244⍴0
```

We also need a state transition function, applying the rules from the `STATE` table, reading from, and writing to `TAPE`:

```
]dinput
stf ← {
    (pos state) ← ⍵
    args ← (0 3[pos⊃TAPE])+⍳3
    (nv m ns) ← STATE[state;args]
    TAPE[pos] ← nv
    (pos+m) ns
}
```

We need to run 12919244 iterations. We'll start in the middle of the tape, and hope that it is long enough:

```
_ ← stf⍣12919244⊢6459622 0  ⍝ Takes ~20s
```

```
assert 4287=⌹←+/TAPE
```

```
4287
```

Statsify All The Words!

Here's a task that I was set many, many years ago for a job interview. The original task sheet is included below, suitably redacted, although I'm sure they have moved on from this by now. They were somewhat arrogant to say "use any programming language" – I only wish I'd known APL by then…

Hi,

We'd like you to write a library that computes some statistics about a text file stored on disk. Your program can be in any language you're comfortable with, but please include instructions about how to use it.

The statistics we'd like you to compute are:

- whitespace delimited word count,
- line count,
- average number of letters per word (to one decimal place) and
- most common letter.

Please send us your program via your [redacted] contact within five working days, and supply instructions that tell us how to compile and use it. Include the text "[redacted]" somewhere in your submission.

We'll check the accuracy of the output; assess the program for style and readability; and also look at how easy it is to extend your program to compute additional statistics. If your program meets the required standard we'll invite you to a phone interview. Please have your code to hand when we talk to you.

Thank you,
[redacted]

So the task is deliberately open-ended and not fully specified; part of the expectation was to show ability to work with ambiguous requirements yadda yadda. So we'll make a note of the assumptions we make. First we need some test data in a file; let's make a few paragraphs of Lorem Ipsum:

```
data←⊃⎕NGET'/Users/stefan/work/dyalog/learnapl/loremipsum.txt'1
```

```
1↑data
```

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam non finibus felis, id imperdiet purus.
```

This file contains Latin words, mixed case, punctuation, whitespace including empty lines. Our first assumption is that we will do all calculations case-insensitively. So we'll pick out the words, and turn them to upper-case. There are a number of ways we could do this, but let's use a regex solution that's UTF-8-safe – we do want to be able to deal with not just English, right?

```
words←⊃(⊢⊆⍨'\W' '.'⎕S 3⍠'UCP'1)1⊃⎕C⊃,/data
```

LOREM	IPSUM	DOLOR	SIT	AMET	CONSECTETUR	ADIPISCING	ELIT	NULLAM	NON	FINIBUS	FELIS	ID	IMPERDIET	PURUS	ALIQUAM	HENDRERI

Is that sufficient? Well, it depends if we expect the file to contain numbers, too:

```
('\W' '.'⎕S 3⍠'UCP'1⊢) 'These are words 55, 56 and 57 in total. Number1nmiddle'
```

These	are	words	55	56	and	57	in	total	Number1nmiddle

The problem statement says nothing about numbers. We could remove those first, like so:

```
('\W' '.'⎕S 3⍠'UCP'1⊢) '\d'⎕R''⊢1⊃⎕C'These are words 55, 56 and 57 in total.
Number1nmiddle'
```

THESE	ARE	WORDS	AND	IN	TOTAL	NUMBERNMIDDLE

```
words←('\W' '.'⎕S 3⍠'UCP'1⊢) '\d'⎕R''⊢1⊃⎕C⊃,/data
```

So we wanted to know the number of words and lines in the text. That's just

```
(≢words)(≢data)
```

```
555 42
```

Next we want to find the most frequent letter. This is a fairly standard *Key* pattern, and here we also assume that if there are several most frequent letters, we should return all of them.

```
letters⍪≂⊂0~≂⊢/0,⊢⌸letters←∈words
```

```
E
```

Finally, we need the average word length, to one decimal place. Let's build a vector of word lengths and average with a neat little train:

```
(+/÷≢)≢¨words
```

```
5.531531531531532
```

Oh, and to one decimal place:

```
⍎1⍕(+/÷≢)≢¨words
```

```
5.5
```

I think we have all the pieces. Let's golf that down to confuse the interviewers properly:

```
{(≢w)(≢⍵)(⍎1⍕(+/÷≢)≢¨w)(l⍪≂⊂0~≂⊢/0,⊢⌸l←∈w←('\W' '.'⎕S 3⍠'UCP'1⊢)'\d'⎕R''⊢1⊃⎕C⊃,/
⍵)}data
```

555	42	5.5	E

If we didn't care about numbers, and we were dealing with only the letters A-Z, we could have written the more compact

```
(≢w)(≢data)(⍎1⍕(+/÷≢)≢¨w)(l⍪≂⊂0~≂⊢/0,⊢⌸l←∈w←(∈∘⎕A⊢)1⊃⎕C⊃,/data)
```

555	42	5.5	E

Now we just have to wait for them to "assess the program for style and readability; and also look at how easy your program is to extend". Once we're done being smug, we could of course have structured that a bit friendlier for people new to APL (what Aaron Hsu calls "the didactic style"):

```
]dinput
stats ← {
    words   ← ('\W' '.'⍠'S' 3⍠'UCP'1⊆⊢) '\d'⍠'R''⊢1⍠C⊃,/⍵ ⍝ Flatten, upcase, remove
numbers, split on "non-word"
    letters ← ∈words                                    ⍝ Join
    common  ← letters⌷⍨⊂0⍨⌸⊢/0,⊢⍪letters                ⍝ Frequency table, picking the
most frequent
    lengths ← ≢¨words                                   ⍝ Vector of length of each word
    ave     ← ⍕1⍕(+/÷≢)lengths                          ⍝ Average, and round to 1
decimal place
    (≢words) (≢⍵) ave common
}
```

```
stats data
```

555	42	5.5	E

`Extensibility` is an amorphous topic, and typically something lauded by Object Orientationists as something that has intrinsic value. Is the above function extensible? Sure! I can add another line to count (say) the number of paragraphs and tag that on the end of the returned vector. However, in an OO language, perhaps you'd created a class that could be extended through inheritance. As APLers, we'd say that this obscures the intent, hiding the functionality away in perhaps other files, instead of having all available within a line or two.

So the question you'd have to ask yourself – would the above effort have gotten you the job?

I think we know the answer to that :) They expected five pages of Java… Anyway, if you aspire to dazzle interviewers, take a leaf out of Aphyr's book.

Too long; didn't read

> Don't be afraid to ask questions. Don't be afraid to ask for help when you need it. I do that every day. Asking for help isn't a sign of weakness, it's a sign of strength. It shows you have the courage to admit when you don't know something, and to learn something new. –*Barack Obama*

I started this from wanting to write down the few things that would have made my learning easier had I known them from the beginning. Many of those things (inevitably) now seem obvious, and this is probably part of the problem I faced at the time: the simple bits that everyone assumes everyone understands are only simple and obvious once you know them. Given that this book ended up creeping in scope, perhaps we should return to that original aim. What are the top-10 things that you really should take the time to understand from day 1, from practical to theoretical?

1. Scalar extension is vital to understand (but also pretty hard to miss).
2. Arrays are not "list of lists": rank is not depth. Make sure you *really* understand the difference.
3. Corollary: a vector of `n` items is not the same as a `1×n` array.
4. Arrays can only contain scalars. This means that everything in an array must have rank 0, and thus higher-rank elements must be enclosed before they can go in an array.
5. Cells are not equal to the element(s) they contain. Understand why pick (⊃) is different from squad (⌷).
6. Enclosures are not shown in output by default. Turn on boxing with `]box on -s=min|mid|max`! This also helps with visualising rank.
7. Understand the leading axis idea and a lot of other things start to make sense.
8. Study the rank (⍤) operator early and don't stop until the penny drops.
9. RIDE: Shift-ctrl-backspace/return moves through your command history.
10. RIDE: Esc 'fixes' a function – change that immediately to command-s!

Final thoughts

Learning APL is a rewarding pursuit. It certainly forced me to reexamine many calcified patterns in my own brain on what programming "should be like". It was frustrating at first, feeling that the process of learning a new language was slower than usual. If you can resist the reaction that this is somehow the fault of APL ("It's unreadable!"), you'll soon conquer that initial small hump on the learning curve.

The other thing that I've found interesting is how many of your established "best practices" have evolved in completely different directions for APL. Aaron Hsu talks about this more eloquently than I ever could in his talk on APL patterns and anti-patterns, but I'd like to highlight the following:

> Good APL follows a set of best practices that directly contradict and conflict with traditional programming wisdom. Indeed, APL design patterns appear as Anti-patterns in most other programming languages. –*Aaron Hsu*

This cognitive dissonance is one reason why some "computer people" *hate* APL. For me it's one of the main draws: two fingers to Uncle Bob, the Gang of Four, and all of their ilk, the buzzkills of the software world.

> Writing APL code the same way you write traditional software only defeats both sanity and efficiency. –*Aaron Hsu*

Idioms, not libraries

When splitting a string on a delimiter is as easy as

```
',' (≠⊆⊢) 'one,two,three'
```

```
Rebuilding user command cache... done
```

one	two	three

there is no value in having a string library that needs importing and a `stringsplit` function. In fact, avoid naming things if you can, and if you must, only name things so close to their use that the definition can be seen without scrolling.

It's kind of the polar opposite of any DRY rule you will have had drilled into you from the beginning. Good APL code reuses short idioms by "copy & paste" so as not to obscure functionality.

Implicit is better than explicit (sometimes)

If you're a Python programmer indoctrinated to believe in the Zen of Python ball and chain, you're probably cringing. In APL, the tacit style allows you to compress more meaning into the glyph count, like, for example the string split example above. This means that short phrases can be made even shorter, and thus become idioms that can be recognized as whole "words". See Aaron Hsu's Co-dfns project for an awe-inspiring masterclass in what this can look like when you take this approach to its logical conclusion.

Economy of representation is a virtue

The fewer keystrokes needed for your implementation, the easier it is to discover bugs by inspection, and the more of the system architecture you can hold in your head. This leads to increased confidence when making changes that these will function as intended in the larger context.

Ken wrote the following:

> The utility of a language as a tool of thought increases with the range of topics it can treat, but decreases with the amount of vocabulary and the complexity of grammatical rules which the user must keep in mind. Economy of notation is therefore important. Economy requires that a large number of ideas be expressible in terms of a relatively small vocabulary. A fundamental scheme for achieving this is the introduction of grammatical rules by which meaningful phrases and sentences can be constructed by combining elements of the vocabulary. –*Ken Iverson*

Somewhere in the mists of time, that idea was forgotten in mainstream languages. Look at your Java, your C++, your Rust, your Go – even your Python, a language which prides itself on its clean syntax. Why *do* you have to type all that stuff to get things done?

Acknowledgements

[Illustration by Razetime on APL Orchard]

I'd like to thank the bunch of mysterious pseudonyms that hang out on the APL Orchard chat room who patiently answered every question I asked. It's the main reason I persisted with APL. In no particular order, an incomplete list: Adám, rikedyp, RGS, ngn, Bubbler, rak1507, Marshall, dzaima, Razetime, Elias Mårtenson, Morten Kromberg and many others. I go under the mysterious pseudonym xpqz – pop in and say hello.
