Array manipulations as functional programming

Jim Pivarski

September 19, 2019

Introduction

The central features of an array library like Numpy or Awkward Array simplify if we think of arrays as functions and these features as function composition. A one-dimensional array of `dtype` d (e.g. `int32` or `float64`) can be thought of as a function from integer indexes to members of d. Thus,

$$\texttt{array[i]}$$

becomes

$$\texttt{array} : \mathbb{Z} \to d$$

because given an integer $\texttt{i} \in \mathbb{Z}$, it returns a value in d. In Python, this function is the implementation of the array's `__getitem__` method.

Specified this way, this is a partial function[1]—for some integers, it raises an exception rather than returning a value in d. (Integers greater than or equal to the array's length or less than its negated length, if the array implements Python's negative indexing, are outside the bounds of the array and do not return a value.) It can be made into a total function by restricting the domain to $[0, n)$ where n is the length of the array:

$$\texttt{array} : [0, n) \to d.$$

We can choose $[0, n)$ as the domain and work with total functions or $\mathbb{Z}$ as the domain and work with partial functions—it is a matter of the granularity of the type system. Numpy has a single type `ndarray` for all arrays (effectively untyped), Numba has an array type that depends on the array's dimension, and C++ has a `std::array<dtype, n>` type that depends on the exact size (`n`) of the array, like our functional description above. As we'll see later, a consequence of this specificity is that the return value of some functions will depend on the values given to that function, a feature known as dependent types[2].

In this note, we'll describe arrays as total functions in a dependent type system.

[1] `https://en.wikipedia.org/wiki/Partial_function`
[2] `https://en.wikipedia.org/wiki/Dependent_type`

Multidimensional arrays

Numpy arrays can have arbitrarily many dimensions, referred to as the array's `shape`. The `shape` is a tuple of positive integers specifying the length of each dimension: $(n_1, n_2, \ldots, n_k)$ is a rank-k tensor ($k = 1$ is a vector, $k = 2$ is a matrix, etc.).

To get values of type d from a rank-k array of `dtype` d, we must specify k integers, each in a restricted domain $[0, n_i)$. In Numpy syntax, this is an implicit Python `tuple` between the square brackets:

```
array[i1, i2, ..., ik]
```

In mathematical syntax, we can represent a k-tuple as a cartesian product,

$$[0, n_1) \times [0, n_2) \times \ldots \times [0, n_k)$$

so the function corresponding to this array is

$$\texttt{array} : [0, n_1) \times [0, n_2) \times \ldots \times [0, n_k) \to d.$$

A function with multiple arguments can be replaced with functions of one argument that each return a function, a process known as currying[3]. For example, the function above can be replaced with

$$\texttt{array} : [0, n_1) \to [0, n_2) \to \ldots \to [0, n_k) \to d$$

by noting that

```
array[i1]
```

returns an array of rank $k - 1$ and `dtype` d, which is a function

$$\texttt{array[i1]} : [0, n_2) \to \ldots \to [0, n_k) \to d$$

(and so on, for each dimension). In fact, Numpy's indexing syntax illustrates this clearly:

```
array[i1, i2, i3] == array[i1][i2][i3]
```

for any `i1`, `i2`, `i3` that satisfy a three-dimensional array's domain.

Record arrays

Numpy also has record arrays[4] for arrays of record-like structures (e.g. `struct` in C). In Numpy, the named fields and their types are considered part of the array's `dtype`, but they are accessed through the same square bracket syntax as elements of the array's `shape`:

```
array[i1, i2, i3][fieldname]
```

[3] https://en.wikipedia.org/wiki/Currying

[4] https://docs.scipy.org/doc/numpy/user/basics.rec.html

where `fieldname` is a string, the name of one of the record's fields. (Numpy does not allow the `fieldname` to be uncurried—it must be in a different set of brackets from `i1`, `i2`, `i3`.)

Since record fields are accessed through a similar syntax, let's consider it part of the array's functional type, making no distinction between `shape` elements and field names. For a record type in which string-valued field names s_1, s_2, ..., s_m map to `dtypes` d_1, d_2, ..., d_m, we can write

$$\begin{aligned}\texttt{recarray} : [0, n) \to\ & s_1 \to d_1 \\ & s_2 \to d_2 \\ & \dots \\ & s_m \to d_m\end{aligned}$$

to represent a one-dimensional record array of length n. This is a dependent type because the choice of field name determines the return type of the function.

A multidimensional record array can be described as

$$\begin{aligned}\texttt{recarray} : [0, n_1) \to [0, n_2) \to \dots \to [0, n_k) \to\ & s_1 \to d_1 \\ & s_2 \to d_2 \\ & \dots \\ & s_m \to d_m\end{aligned}$$

or as

$$\begin{aligned}\texttt{recarray} : [0, n_1) \to\ & s_1 \to [0, n_2) \to \dots \to [0, n_k) \to d_1 \\ & s_2 \to [0, n_2) \to \dots \to [0, n_k) \to d_2 \\ & \dots \\ & s_m \to [0, n_2) \to \dots \to [0, n_k) \to d_m\end{aligned}$$

or any other placement of the field name index within the ordered sequence of dimensional indexes. In general, the string indexes (field names) commute with the integer indexes (dimensions). This is evident in Numpy's syntax:

```
recarray[i1][i2][i3][fieldname] == recarray[i1][i2][fieldname][i3]
                                == recarray[i1][fieldname][i2][i3]
                                == recarray[fieldname][i1][i2][i3]
```

It is also evident if the array is arranged as a rectilinear table, in which $[0, n_1)$, $[0, n_2)$, $\dots [0, n_k)$ form a k-dimensional lattice of bounded integers and the field names are an additional dimension, indexed by a finite set of strings with no predefined order. This dimension of category labels is usually called the "columns" and all other dimensions are called "rows." In this picture, rearranging the order of the string index and the integer indexes corresponds to selecting a column before a row, rather than a row before a column.

Vectorized functions

Numpy uses so-called "vectorized" functions or "universal" functions ("ufuncs") for most calculations[5]. (These are not to be confused with vectorized instructions in CPU hardware, but are based on a similar idea.) Any function f that maps scalar `dtypes` d^A to d^B,

$$f : d^A \to d^B,$$

can be lifted to a vectorized function that maps arrays of `dtype` d^A to arrays of `dtype` d^B:

$$\text{ufunc}(f) : \left([0, n) \to d^A\right) \to \left([0, n) \to d^B\right).$$

Note that the `shape` of the array, $[0, n)$ in this case, is the same for the argument type of ufunc(f) as for its return type.

This ufunc functor is a partial application of what would be called "map" in most functional languages[6]. The map functor takes a function and a collection, returning a collection of the same length with the function applied to each element. The ufunc functor only takes a function, and its result is applied to collections (arrays) later.

Since arrays are themselves functions, applying ufunc(f) to an array is a composition[7] of the array with f. Thus, the following is true for any $\texttt{i} \in [0, n)$:

$$\underbrace{\text{ufunc}(f)(\texttt{array})}_{\text{array}}(\texttt{i}) = f(\underbrace{\texttt{array(i)}}_{\text{scalar}})$$

$$\underbrace{\texttt{numpy.vectorize(f)(array)}}_{\text{array}}\texttt{[i]} = \texttt{f(}\underbrace{\texttt{array[i]}}_{\text{scalar}}\texttt{)}.$$

by associativity of function composition. This composition always applies f to the *output* of the array, never the *input* (function composition is not commutative).

$$\begin{aligned} f &: d^A \to d^B \\ \texttt{array} &: [0, n) \to d^A \\ \text{ufunc}(f)(\texttt{array}) = f \circ \texttt{array} &: [0, n) \to d^B. \end{aligned}$$

Using associativity again, we should be able to compose a sequence of scalar functions $f : d^A \to d^B$, $g : d^B \to d^C$, ..., $h : d^Y \to d^Z$ before applying them to the array. If the scalar function can be extracted from a ufunc object, it would be possible to compose

$$\text{ufunc}(f) \circ \text{ufunc}(g) \circ \ldots \circ \text{ufunc}(h)$$

[5] `https://docs.scipy.org/doc/numpy/reference/ufuncs.html`
[6] `https://en.wikipedia.org/wiki/Map_(higher-order_function)`
[7] `https://en.wikipedia.org/wiki/Function_composition`

into a single

$$\text{ufunc}(f \circ g \circ \ldots \circ h) : \left([0,n) \to d^A\right) \to \left([0,n) \to d^Z\right)$$

that can be applied to an array. This is an optimization known as loop fusion[8], and is often faster than making multiple passes over arrays and possibly allocating large temporary arrays between ufuncs. There have been proposals[9] and external libraries[10] to add this feature transparently to Numpy. In principle, it could even be an explicit (user-visible) feature of the ufunc object, but to my knowledge, it has never been implemented as such.

Array slicing

Whereas ufuncs compose scalar functions to the output of an array, slicing composes `index` arrays (which are functions) to the input of an array.

The fact that array slicing is itself composition may not be obvious because of the way that slicing is presented:

```
array[i:j]
```

does not seem to be a composition of two arrays. The first point to make is that all of Numpy's slicing mechanisms—range slices (Python's `slice` operator or `start:stop:step`), `numpy.compress` with boolean arrays, and `numpy.take` with integer arrays—can be rewritten in terms of `numpy.take` with integer arrays:

- A range slice `start:stop:step` can be replaced with an integer sequence

  ```
  range(start, stop, step)
  ```

  (ignoring the effect of negative `start` and `stop`).

- A boolean array `mask` can be replaced with `numpy.nonzero(mask)`.

- An integer array `index` is already an integer array.

As an array, `index` is a function $\texttt{index} : [0,x) \to [0,n)$ that can be composed with $\texttt{array} : [0,n) \to d$ to produce

$$\texttt{array} \circ \texttt{index} : [0,x) \to d$$

Note that `index` is to the right (transforms the *input*) of `array`, whereas ufunc(f) put f to the left (transforms the *output*) of `array`.

Numpy uses the same syntax for this function composition, `array[index]`, as it does for function evaluation, `array[i]`, which is potentially confusing. Let's illustrate this with an extended example.

[8] https://en.wikipedia.org/wiki/Loop_fission_and_fusion

[9] https://numpy.org/doc/1.14/neps/deferred-ufunc-evaluation.html

[10] https://www.weld.rs/weldnumpy

Example

Consider two functions that are defined on all non-negative integers (at least).

```
def f(x):
    return x**2 - 5*x + 10
def g(y):
    return max(0, 2*y - 10) + 3
```

They may be transformed into arrays by sampling `f`, `g`, and `f` ∘ `g` at enough points to avoid edge effects from their finite domains. For `f` and `g` above, 100 points in `g` is enough to accept the entire range of `f` when `f` is sampled at 10 points.

```
F   = numpy.array([f(i) for i in range(10)])      # F is f at 10 elements
G   = numpy.array([g(i) for i in range(100)])     # G is g at 100 elements
GoF = numpy.array([g(f(i)) for i in range(10)])   # GoF is g∘f at 10 elements
```

Now `F` : $[0, 10) \to [4, 46)$, `G` : $[0, 100) \to [3, 191)$, and `GoF` : $[0, 10) \to [3, 85)$.

Indexing `G` by `F` can be expressed with square-bracket syntax or `numpy.take`, and it returns the same result as the sampled composition `GoF`.

```
G[F]               # → [13,  5,  3,  3,  5, 13, 25, 41, 61, 85]
G.take(F)          # → [13,  5,  3,  3,  5, 13, 25, 41, 61, 85]
numpy.take(G, F)   # → [13,  5,  3,  3,  5, 13, 25, 41, 61, 85]

GoF                # → [13,  5,  3,  3,  5, 13, 25, 41, 61, 85]
```

In `GoF`, the functions are composed before being transformed into arrays, and in `G[F]`, the arrays themselves are composed via integer-array indexing.

Function composition is associative, so we should be able to change the order of two integer-array indexings. To demonstrate this, introduce another array, which need not have integer `dtype`.

```
H = numpy.arange(1000)*1.1
```

When we compute `H` indexed by `G` indexed by `F`, it shouldn't matter whether the `H[G]` index is computed first or the `G[F]` index is computed first, and we see that this is the case.

```
H[G][F]            # → [14.3  5.5  3.3  3.3  5.5 14.3 27.5 45.1 67.1 93.5]
H[G[F]]            # → [14.3  5.5  3.3  3.3  5.5 14.3 27.5 45.1 67.1 93.5]
```

Multidimensional slicing

If Numpy's integer-array indexing for multiple dimensions worked the same was as its range-slicing does, then the above would be trivially extensible to any number of dimensions. However, Numpy's integer-array indexing (called "advanced indexing")[11] couples iteration over integer arrays supplied to each of the k slots in a rank-k array.

To work-around this caveat, consider rank-k integer arrays in each of the k slots, in which the integer array in slot i has shape $(1, \ldots, n_i, \ldots, 1)$. For example, a three-dimensional slice

```
array[start1:stop1, start2:stop2, start3:stop3]
```

can be simulated with integer arrays

```
array[numpy.arange(start1, stop1).reshape(-1, 1, 1),
      numpy.arange(start2, stop2).reshape(1, -1, 1),
      numpy.arange(start3, stop3).reshape(1, 1, -1)]
```

because Numpy broadcasts the three integer arrays into a common three-dimensional shape, and the symmetry of these arrays decouples their effects in each dimension.

Beyond rectilinear arrays

Numpy is a library for contiguous, rectangular grids of numbers—within that scope, range-slicing and broadcasting[12] can be effectively computed without modifying or copying array buffers, using stride tricks[13].

However, we often want more general data structures, so Awkward Array extends Numpy by interpreting collections of rectilinear arrays as non-rectilinear arrays. The two most important additions are

- records containing fields of any type and
- arrays of unequal-length subarrays.

(Although union types, nullable types, and cross-referencing are also included in the Awkward Array library, they are less related to this note's focus on arrays as functions.)

[11] https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html
[12] https://docs.scipy.org/doc/numpy/user/basics.broadcasting.html
[13] https://docs.scipy.org/doc/numpy/reference/generated/numpy.lib.stride_tricks.as_strided.html

Non-rectilinear record types

Numpy's record array only allows one dimension of category labels, and all arrays identified by a category label share the same `shape`. This is because the location of each of these arrays is defined by a stride. It also means that Numpy is limited to "arrays of structs."[14]

Suppose we want field s_1 of an array to have shape $[0, n_1)$ and field s_2 to have shape $[0, n_1) \times [0, n_2)$. That is, at each i_1, field s_1 has a scalar and field s_2 has an array. This can be described as the dependent type

$$\begin{aligned} \texttt{array} : [0, n_1) \to & s_1 \to d^A \\ & s_2 \to [0, n_2) \to d^A. \end{aligned}$$

With $\texttt{i1} \in [0, n_1)$ and $\texttt{i2} \in [0, n_2)$,

$$\begin{aligned} \texttt{array[i1][s1]} &\text{ returns } d^A \text{ (a scalar)} \\ \texttt{array[i1][s2]} &\text{ returns } [0, n_2) \to d^A \text{ (an array), and} \\ \texttt{array[i1][s2][i2]} &\text{ returns } d^A \text{ (a scalar).} \end{aligned}$$

Such an array can still be passed into ufuncs because ufuncs compose a function $f : d^A \to d^B$ to the *output* of an array. For example,

$$\begin{aligned} \text{ufunc}(f)(\texttt{array}) : [0, n_1) \to & s_1 \to d^B \\ & s_2 \to [0, n_2) \to d^B. \end{aligned}$$

Such an array can still be sliced because slicing composes an integer array `index` : $[0, x) -> [0, n_1)$ to the *input* of the array. For example,

$$\begin{aligned} \texttt{array}(\texttt{index}) : [0, x) \to & s_1 \to d^A \\ & s_2 \to [0, n_2) \to d^A. \end{aligned}$$

The string-index/integer-index can be *partially* commuted: the domain $\{s_1, s_2\}$ can be moved from the middle of this function type to the left, like so:

$$\begin{aligned} \texttt{array} : & s_1 \to [0, n_1) \to d^A \\ & s_2 \to [0, n_1) \to [0, n_2) \to d^A, \end{aligned}$$

but it cannot be moved to the right, where the type of each field is different.

Similarly, if we had nested records, with s_1 containing `dtype` d and s_2 containing a record with fields t_1 and t_2, our options for commuting indexes would be limited to one possibility:

$$\begin{aligned} \texttt{array} : [0, n_1) \to & s_1 \to d \\ & s_2 \to t_1 \to d \\ & t_2 \to d \end{aligned} \qquad \begin{aligned} \texttt{array} : & s_1 \to [0, n_1) \to d \\ & s_2 \to [0, n_1) \to t_1 \to d \\ & t_2 \to d. \end{aligned}$$

[14] `https://en.wikipedia.org/wiki/AoS_and_SoA`

Non-rectilinear shapes

We can also consider arrays of unequal-length subarrays ("jagged" or "ragged" arrays).

Like records, jagged arrays must be described by a dependent type if the function is to be defined on its whole domain. Just as each value in a record's domain, $\{s_1, s_2\}$, can return a different type, each value in a jagged array's domain can return a different type.

For example, the type of an array like `[[1.1, 2.2, 3.3], [], [4.4, 5.5]]` is

$$\begin{aligned}\texttt{array} : 0 &\to [0, 3) \to d^A \\ 1 &\to [0, 0) \to d^A \\ 2 &\to [0, 2) \to d^A.\end{aligned}$$

The type description grows with the length of the array—while it may be practical to fully enumerate a record's fields, it's not practical to enumerate a large jagged array's type.

Jagged arrays can be passed into ufuncs and sliced for the same reasons as non-rectilinear records: these features are composition to the *output* and *input* of the array, respectively:

$$\begin{aligned}\text{ufunc}(f)(\texttt{array}) : 0 &\to [0, 3) \to d^B \\ 1 &\to [0, 0) \to d^B \\ 2 &\to [0, 2) \to d^B\end{aligned} \qquad \begin{aligned}\texttt{array}(\texttt{index}) : 0 &\to [0, 2) \to d^A \\ 1 &\to [0, 3) \to d^A \\ 2 &\to [0, 3) \to d^A \\ 3 &\to [0, 0) \to d^A\end{aligned}$$

for $f : d^A \to d^B$ and `index = [2, 0, 0, 1]`, for example.

Non-rectilinear record types and non-rectilinear shapes can be combined, and these two generators can already produce data types as general as JSON. (Note that the explicit enumeration of dependent types for each array index allows heterogeneous lists and `null`.)

String-valued field indexes can always commute to the left through a jagged dimension, but it can only commute to the right if domains match for all elements of a jagged dimension. For example, $\{s_1, s_2\}$ can commute through both levels of the following jagged array, but only because it has the same combinations of nested shapes for both s_1 and s_2.

$$\begin{aligned}\texttt{a} : s_1 \to\ & 0 \to [0, 3) \to d^A \\ & 1 \to [0, 0) \to d^A \\ & 2 \to [0, 2) \to d^A \\ s_2 \to\ & 0 \to [0, 3) \to d^B \\ & 1 \to [0, 0) \to d^B \\ & 2 \to [0, 2) \to d^B\end{aligned} \qquad \begin{aligned}\texttt{a}:\ & 0 \to s_1 \to [0, 3) \to d^A \\ & \qquad s_2 \to [0, 3) \to d^B \\ & 1 \to s_1 \to [0, 0) \to d^B \\ & \qquad s_2 \to [0, 0) \to d^B \\ & 2 \to s_1 \to [0, 2) \to d^B \\ & \qquad s_2 \to [0, 2) \to d^B\end{aligned} \qquad \begin{aligned}\texttt{a} : 0 \to [0, 3) \to\ & s_1 \to d^A \\ & s_2 \to d^B \\ 1 \to [0, 0) \to\ & s_1 \to d^A \\ & s_2 \to d^B \\ 2 \to [0, 2) \to\ & s_1 \to d^A \\ & s_2 \to d^B\end{aligned}$$

Conclusions

The reason Awkward Array can make use of Numpy's ufunc and slicing concepts, despite a much more general data model, is because arrays are functions and these two operations correspond to function composition at the *output* or the *input* of the array.

This note does not discuss the implementation details of Numpy or Awkward Array, though their scopes are well drawn by technical considerations. Numpy focuses on rectilinear arrays because stride tricks greatly optimize that domain. Awkward Array is more general, but it cannot use stride tricks on non-rectilinear data. Parts of a data structure must be physically separated in memory to allow generalized reshaping. However, the fact that slices can be explicitly composed with one another before applying them to an array (the associativity of function composition) has been very useful when dealing with separated data: slices do not need to be propagated all the way down a tree of nested structures—the meaning is preserved by lazy evaluation.