Array API Specification

Aaron Meurer[‡][*]

Abstract—The array API standard (https://data-apis.org/array-api/) is a common specification for Python array libraries, such as NumPy, PyTorch, CuPy, Dask, and JAX.

This standard will make it straightforward for array-consuming libraries, like scikit-learn and SciPy, to write code that uniformly supports all of these libraries. This will allow, for instance, running the same code on the CPU and GPU.

This proceedings paper will cover the scope of the array API standard, supporting tooling which includes a library-independent test suite and compatibility layer, what work has been completed so far, and the plans going forward.

Index Terms—Python, Arrays, Tensors, NumPy, CuPy, PyTorch, JAX, Dask

Introduction

TODO: Need more for the intro here, including a motivating example.

There are three primary stakeholders involved in Python code making use of arrays: array libraries, array library consumers, and end users. *Array libraries* are Python libraries that implement an array object and a namespace that conforms to the array API standard. Examples of array libraries are NumPy, CuPy, and PyTorch. *Array library consumers* are libraries that implement functionality on top of array libraries. Examples of array library consumers are SciPy and scikit-learn. *End users* are people such as scientists, data scientists, machine learning practitioners, as well as other higher level libraries, which make use of array libraries and array consuming libraries to solve problems with their data.

In the present paradigm, array library consuming codes are written against a single array library (typically NumPy). Using the algorithms they provide with other array libraries is impossible. This is because, firstly, the array library is hard-coded into the functions with things like `np.<function>`, where `np` is `numpy`. Secondly, even if `np` could be swapped out with a different array library, different libraries provide different APIs, so the code would be unlikely to run without modification.

However, if we examine the three stakeholders, we see that each stakeholder adds its own set of strengths to the ecosystem. Array libraries provide an array object and corresponding functions that are optimized against a certain set of use-cases and hardware. Array consumer libraries provide useful implementations of higher level algorithms. End users provide the actual data and define the problem to be solved. The current paradigm is misaligned, as end users are the ones who are most suitable to choose the array library that best fits their needs. They may prefer a battle-tested, highly portable library like NumPy, or a library that has been optimized for deep learning workflows like PyTorch, or a library that can scale to multiple machines like Dask. But if they also want to make use of a high level array consumer library, that choice of array library will be forced on them by whatever array library it is implemented against.

The array API specification corrects this misalignment by specifying a uniform API for array libraries to provide. Array consumer libraries can then be written against this one uniform API, allowing their functionality to work with arrays from any conforming array library. End users are then able to chose their array library without that choice restricting their choices of array consumer libraries. The usability improvement from different array libraries themselves having more consistent APIs and semantics additionally provides a benefit to the whole ecosystem.

Motivating Example

TODO

History of the Consortium

TODO: Distill this blog post https://data-apis.org/blog/announcing_the_consortium/, as well as more recent history like the standard releases.

The Data APIs Consortium was formed in 2020, with the goal of unifying API standards for Python array and dataframe libraries.

Goals and Non-Goals

The array API specification has the following goals:

- Make it possible for array-consuming libraries to start using multiple types of arrays as inputs.
- Enable increased sharing and reuse of code built on top of the core functionality in the API standard.
- For authors of new array libraries, provide a concrete API that can be adopted as is, rather than each author having to decide what to borrow from where and where to deviate.
- Make the learning curve for users less steep when they switch from one array library to another one.

Additionally, the specification has several non-goals:

- Making array libraries identical for the purpose of merging them. Each library will keep having its own particular strength, whether it's offering functionality beyond what's in the standard, performance advantages for a given use case, specific hardware or software environment support, or more.

* Corresponding author: asmeurer@quansight.com
‡ Quansight

- Implement a backend or runtime switching system to be able to switch from one array library to another with a single setting or line of code. This may be feasible, however it's assumed that when an array-consuming library switches from one array type to another, some testing and possibly code adjustment for performance or other reasons may be needed.
- Making it possible to mix multiple array libraries in function calls. Most array libraries do not know about other libraries, and the functions they implement may try to convert "foreign" input, or raise an exception. This behavior is hard to specify. It is better to require the end user to use a single array library that best fits their needs. Note that specification of an interchange protocol is within scope, but interchange between array libraries is only done explicitly in the specification.

Design Principles

The array API standard has been developed with several design principles in mind. The most important principle is that the standard only specifies behavior that is already widely supported by most existing array libraries. The goal is to minimize the number of backwards incompatible changes required for libraries to support the specification. This in particular leaves many things out-of-scope if they are not already supported by all major array libraries.

The standard has been developed based on the following core principles:

- Don't assume any dependency other than Python itself. Different array libraries have independent codebases, and link against varying backend libraries depending on what hardware they support. There is no common array layer, and array libraries do not need to know about each other. Data can be interchanged between libraries using a protocol which does not require a dependency.
- Libraries may implement behaviors beyond what is specified. Except in a few special instances where avoiding bad behavior is desired, the spec does not disallow libraries to implement additional functions, methods, keyword arguments, and allow additional input types. The onus is on array library consumers to ensure they write portable code (the strict minimal `numpy.array_api` module is designed to help here).
- APIs should support accelerators. This means either not specifying behaviors that are difficult to implement performantly or making them optional.
- In a similar vein, APIs should support JIT compilers. For example, the output type of any function should only depend on its input types.
- The API is primarily functional (e.g., `xp.any(x)` instead of `x.any()`). Outside of Python "dunder" operators, there are only a few method defined on the array object. Functional APIs are already preferred for most array libraries, functional code is easier to read, especially for expressions with many mathematical functions and operations, and functions make it clearer that an operation returns a new array rather than mutating the input array in-place, which is avoided in the specification (see the next bullet point).
- Copy-view behavior and mutability is not required. Array libraries may implement mutation but the behavior of in-place mutation with views is not guaranteed by the spec. Operations producing "views" on existing data is considered an implementation detail and should not be relied on for portability across libraries. The `out` keyword is omitted from API definitions.
- No value-based casting. The output data type of any function or operation should depend only on the input data type(s), not the array values.
- No dimension dependent casting. The output data type of any function or operation should function independently of the input array dimensionality. This also means that 0-D arrays are fully supported. Scalars as a separate concept are not specified.
- Functions are generally only added to the specification if they are already implemented by a wide range of array libraries. There are only a few exceptions where the consortium has decided to specify new functions that are not implemented anywhere yet, because none of the existing implementations were satisfactory (for example, a new `isdtype()` function; see the Data Types section below).
- Functions that can easily be implemented in terms of existing standardized functions do not necessarily need to be standardized.
- Functions with data-dependent output shapes are optional, since graph-based libraries like JAX and Dask cannot easily support them. This includes boolean indexing, `nonzero()`, and the `unique_*` functions.
- Type annotations are defined in a basic way in the spec, but libraries may extend them. Input types are designed to be as simple as possible. For example, functions are only required to accept `array` objects. Accepting "array like" types like lists of numbers, as NumPy does, is problematic because it complicates type signatures, and calling `asarray()` at the top of every function adds additional overhead. However, these type signatures are not strict: libraries may choose to accept additional input types outside of those that are specified.
- The accuracy and precision of numerical functions are not specified beyond the basic IEEE 754 rules.

Scope

The scope of the array API specification includes:

- Functionality that needs to be included in an array library for it to adhere to this standard.
- Names of functions, methods, classes and other objects.
- Function signatures, including type annotations.
- Semantics of functions and methods, i.e., expected outputs and dtypes of numerical results.
- Semantics in the presence of `nan`'s, `inf`'s, and empty arrays (i.e. arrays including one or more dimensions of size `0`).
- Casting rules, broadcasting, and indexing.
- Data interchange, i.e., protocols to convert one type of array into another type, potentially sharing memory.
- Device support.

To contrast, the following are considered **out-of-scope** for the array API specification

- Implementations of the standard are out of scope. Members of the consortium have played a role in helping

libraries like NumPy, CuPy, and PyTorch implement the standard, but this work has been done independently of the standard. In particular, the standard is completely independent of any specific implementation and does not make reference to or depend on any given implementation or Python library (the `array-api-compat` library has been produced as a compatibility layer on top of array libraries such as NumPy, CuPy, and PyTorch, but this library is provided only as a helper tool for array consumer libraries. It is not in any way required to make use of the array API).
- Execution semantics are out of scope. This includes single-threaded vs. parallel execution, task scheduling and synchronization, eager vs. delayed evaluation, performance characteristics of a particular implementation of the standard, and other such topics.
- Non-Python API standardization (e.g., Cython or NumPy C APIs).
- Standardization of dtypes not already supported by all existing array libraries is out of scope. This includes bfloat16, extended precision floating point, datetime, string, object and void dtypes.
- The following topics are out of scope: I/O, polynomials, error handling, testing routines, building and packaging related functionality, methods of binding compiled code (e.g., `cffi`, `ctypes`), subclassing of an array class, masked arrays, and missing data.
- NumPy (generalized) universal functions, i.e. ufuncs and gufuncs.
- Behavior for unexpected/invalid input to functions and methods.

For out-of-scope behavior, array libraries are free to implement it or to raise an error. It is up to array consuming libraries to ensure they write portable code that doesn't depend on behaviors outside of the specification. The `numpy.array_api` implementation, discussed below, can be a useful tool for this.

Features

TODO: write an introduction here.

Data Interchange

As discussed in the non-goals section, array libraries are not expected to support mixing arrays from other libraries. Instead, there is an interchange protocol that allows converting an array from one library to another.

To be useful, any such protocol must satisfy some basic requirements:

- Interchange must be specified as a protocol, rather than requiring a specific dependent package. The protocol should describe the memory layout of an array in an implementation-independent manner.
- Support for all dtypes in this API standard (see Data Types below).
- It must be possible to determine on which device the array to be converted resides (see Device Support below). It must be possible to determine on what device the array that is to be converted lives (see Device Support below). A single protocol is preferable to having per-device protocols. With separate per-device protocols it's hard to figure out unambiguous rules for which protocol gets used, and the situation will get more complex over time as TPU's and other accelerators become more widely available.
- The protocol must have zero-copy semantics where possible, making a copy only if needed (e.g. when data is not contiguous in memory).
- There must be both a Python-side and a C-side interface, the latter with a stable C ABI. All prominent existing array libraries are implemented in C/C++, and are released independently from each other. Hence a stable C ABI is required for packages to work well together. The protocol must support low level access to be usable by libraries that use JIT or AOT compilation, and it must be usable from any language.

To satisfy these requirements, DLPack was chosen as the data interchange protocol. DLPack is a standalone protocol with a header-only C implementation that is ABI stable, meaning it can be used from any language. It is designed with multi-device support and supports all the data types specified by the standard. It also has several considerations for high performance. DLPack support has already been added to all the major array libraries, and is the most widely supported interchange protocol across different array libraries.

The array API specifies the following syntax for DLPack support:

- A `.__dlpack__()` method on the array object, which exports the array as a DLPack capsule.
- A `.__dlpack__device__()` method on the array object, which returns the device type and device ID in DLPack format.
- A `from_dlpack()` function, which converts an object with a `__dlpack__` method into an array for the given array library.

Note that `asarray()` also supports the buffer protocol for libraries that already implement it, like NumPy. But the buffer protocol is CPU-only, meaning it is not sufficient for the above requirements.

Device Support

The standard supports specifying what device an array should live on. This is implemented by explicit `device` keywords in creation functions, with the convention that execution takes place on the same device where all argument arrays are allocated. This method of specifying devices was chosen because it is the most granular, despite its potential verbosity. Other methods of specifying devices such as context managers are not included, but may be added in future versions of the spec.

The primary intended usage of device support in the specification is geared towards array consuming libraries. End users who create arrays from a specific array library may use that library's specific syntax for specifying the device relative to their specific hardware configuration. For an array consuming library, the important things they need to be able to do are

- Create new arrays on the same device as an array that's passed in.
- Determine whether two input arrays are present on the same device or not.
- Move an array from one device to another.
- Create output arrays on the same device as the input arrays.

- Pass on a specified device to other library code.

Consequently, the specified device syntax focuses primarily on getting the device of a given array and setting the device to the same device as another array. The specifics of how to specify actual devices are left unspecified. These specifics differ significantly between existing implementations, such as CuPy and PyTorch.

The syntax that is specified is

- A `.device` property on the array object, which returns a device object representing the device the data in the array is stored on. Nothing is specified about the device object other than that it must support basic == equality comparison within the same library.
- A `device=None` keyword for array creation functions, which takes an instance of a device object.
- A `.to_device()` method on the array object to copy an array to a different device.

In other words, the only specified way to access a device object is via the `.device` property of an existing array object. The specifics of how to specify an actual device depends on the actual array library used, and is something that will be done by end users, not array library consumers.

This also means that the following are currently considered out-of-scope for the array API specification:

- Identifying a specific physical or logical device across libraries
- Setting a default device globally
- Stream/queue control
- Distributed allocation
- Memory pinning
- A context manager for device control

All functions should respect explicit `device=` assignment, preserve the device whenever possible, and avoid implicit data transfer between devices.

Functions and Methods

Signatures: All function signatures in the specification make use of PEP 570 positional-only arguments for arguments that are arrays. It should not matter if one library defines `def atan2(y, x): ...`, for instance, and another library defines `def atan2(x1, x2): ...`. With positional-only arguments, the function must be called by passing the arguments by position, like `atan2(a, b)`. The specific name given the arguments by the library becomes separate from the API.

Additionally, most keyword arguments are keyword-only. For example, `ones((3, 3), int64)` is not allowed—it must be called as `ones((3, 3), dtype=int64)`. This makes user code more readable, and future-proofs the API by allowing additional keyword arguments to be added without breaking existing function calls.

All signatures in the specification include type annotations. These type annotations use generic types like `array` and `dtype` type to represent a library's array or dtype objects. These type annotations represent the minimal types that are required to be supported by the specification. A library may choose to accept additional types, although any use of this functionality will be non-portable. Functionally, type annotations serve no purpose other than as documentation. Libraries are not required to implement any sort of runtime type checking, or to actually include such annotations in their own function signatures. The array API specification does not attempt extend type annotation syntax beyond what is already specified by PEPs and supported by popular type checkers such as Mypy. For instance, including dtype or shape information in the annotated type signatures is considered out-of-scope.

Here is an example type signature in the specification

```
def asarray(
    obj: Union[
        array, bool, int, float, complex,
        NestedSequence, SupportsBufferProtocol
    ],
    /,
    *,
    dtype: Optional[dtype] = None,
    device: Optional[device] = None,
    copy: Optional[bool] = None,
) -> array:
    ...
```

Array Methods and Attributes: All relevant Python double underscore (dunder) methods (e.g., __add__, __mul__, etc.) are specified for the array object, so that people can write array code in a natural way using operators. Each dunder method has a corresponding functional form (e.g., __add__ ↔ `xp.add()`). For consistency, this is done even for operators that may seem unnecessary, like `__pos__` ↔ `positive()`. Operators and their corresponding functions behave identically, except that operators accept Python scalars (see "type promotion" below), while functions are only required to accept arrays.

In addition to the standard Python dunder methods, the standard adds a some new dunder methods:

- `x.__array_namespace__()` returns the corresponding array API compliant namespace for the array `x`. This solves the problem of how array consumer libraries determine which namespace to use for a given input. A function that accepts input `x` can call `xp = x.__array_namespace__()` at the top to get the corresponding array API namespace `xp`, whose functions are then used on `x` to compute the result, which will typically be another array from the `xp` library.
- `__dlpack__()` and `__dlpack_device__()` (see the "data interchange" section above).

Functions: Aside from dunder methods, the only methods/attributes defined on the array object are `x.to_device()`, `x.dtype`, `x.device`, `x.mT`, `x.ndim`, `x.shape`, `x.size`, and `x.T`. All other functions in the specification are defined as functions. These functions include

- Elementwise functions. These include functional forms of the Python operators (like `add()`) as well as common numerical functions like `exp()` and `sqrt()`. Elementwise functions do not have any additional keyword arguments.
- Creation functions. This includes standard array creation functions including `ones()`, `linspace`, `arange`, and `full`, as well as the `asarray()` function, which converts "array like" inputs like lists of floats and object supporting the buffer protocol to array objects. Creation functions all include a `dtype` and `device` keywords (see the "Device" section above). The `array` type is not specified anywhere in the spec, since different libraries use different types for their array objects, meaning

asarray() and the other creation functions serve as the effective "array constructor".

- Data type functions are basic functions to manipulate and introspect dtype objects such as finfo(), can_cast(), and result_type(). Notable among these is a new function isdtype(), which is used to test if a dtype is among a set of predefined dtype categories. For example, isdtype(x.dtype, "real floating") returns True if x has a real floating-point dtype like float32 or float64. Such a function did not already exist in a portable way across different array libraries. One existing alternative was the NumPy dtype type hierarchy, but this hierarchy is complex and is not implemented by other array libraries such as PyTorch. The isdtype() function is a rare example where the consortium has specified a completely new function in the array API specification—most of the specified functions are already widely implemented across existing array libraries.
- Linear algebra functions. Only basic manipulation functions like matmul() are required by the specification. Additional linear algebra functions are included in an optional linalg extension (see below).
- Manipulation functions such as reshape(), stack(), and squeeze().
- Reduction functions such as sum(), any(), all(), and mean().
- Four new functions unique_all(), unique_counts(), unique_inverse(), and unique_values(). These are based on the np.unique() function but have been split into separate functions. This is because np.unique() returns a different number of arguments depending on the values of keyword arguments. Functions like this whose output type depends on more than just the input types are hard for JIT compilers to handle, and they are also harder for users to reason about.

Note that the unique_* functions, as well as nonzero() have a data-dependent output shape, which makes them difficult to implement in graph libraries. Therefore, such libraries may choose to not implement these functions.

Data Types: Data types are defined as named dtype objects in the array namespace, e.g., xp.float64. Nothing is specified about what these objects actually are beyond that they should obey basic equality testing. Introspection on these objects can be done with the data type functions (see above).

The following dtypes are defined:

- Boolean: bool.
- Integer: int8, int16, int32, int64, uint8, uint16, uint32, and uint64.
- Real floating-point: float32 and float64.
- Complex floating-point: complex64 and complex128.

These dtypes were chosen because they are the most widely adopted set across existing array libraries. Additional dtypes may be considered for addition in future versions of the standard.

Additionally, a conforming library should have "default" integer and floating-point dtypes, which is consistent across platforms. This is used in contexts where the result data type is otherwise ambiguous, for example, in creation functions when no dtype is specified. This allows libraries to default to 64-bit or 32-bit data types depending on the use-cases they are aiming for. For example, NumPy's default integer and float dtypes are int64 and float64, whereas, PyTorch's defaults are int64 and float32.

See also the "Type Promotion" section below for information on how dtypes combine with each other.

Broadcasting

All elementwise functions and operations that accept more than one array input apply broadcasting rules. The broadcasting rules match the commonly used semantics of NumPy, where a broadcasted shape is constructed from the input shapes by prepending size-1 dimensions and broadcasting size-1 dimensions to otherwise equal non-size-1 dimensions (for example, a shape (3, 1) and a shape (2, 1, 4) array would broadcast to a shape (2, 3, 4) array by virtual repetition of the array along the broadcasted dimensions). Broadcasting rules should be applied independently of the input array data types or values.

Indexing

Arrays should support indexing operations using the standard Python getitem syntax, x[idx]. The indexing semantics defined are based on the common NumPy array indexing semantics, but restricted to a subset that is common across array libraries and does not impose difficulties for array libraries implemented on accelerators. Basic integer and slice indexing is defined as usual, except behavior on out-of-bounds indices is left unspecified. Multiaxis tuple indices are defined, but only specified when all axes are indexed (e.g., if x is 2-dimensional, x[0, :] is defined but x[0] may not be supported). A None index may be used in a multiaxis index to insert size-1 dimensions (xp.newaxis is specified as a shorthand for None). Boolean array indexing (also sometimes called "masking") is specified, but only for instances where the boolean index has the same dimensionality as the indexed array. The result of a boolean array indexing is data-dependent, and thus graph-based libraries may choose to not implement this behavior.

Integer array indexing is not specified, however a basic take() is specified and put() will be added in the 2023 version of the spec.

Note that views are not required in the specification. Libraries may choose to implement indexed arrays as views, but this should be treated as an implementation detail by array consumers. In particular, any mutation behavior that affects more than one array object is considered an implementation detail that should not be relied on for portability.

As with other APIs, extensions of these indexing semantics, e.g., by supporting the full range of NumPy indexing rules, is allowed. Array consumers using these will only need to be aware that their code may not be portable across libraries.

It should be noted that both 0-D arrays (i.e., "scalar" arrays with shape () consisting of a single value), and size-0 arrays (i.e., arrays with 0 in their shape with no values) are fully supported by the specification. The specification does not have any notion of "array scalars" like NumPy's np.float64(0.), only 0-D arrays. Scalars are a NumPy-only thing, and it is unnecessary from the point of view of the specification to have them as a separate concept from 0-D arrays.

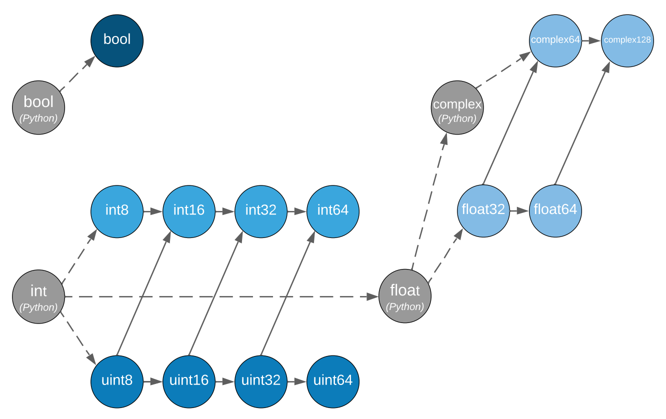

Fig. 1: *The dtypes specified in the spec with required type promotions, including promotions for Python scalars in operators. Cross-kind promotion is not required and is discouraged.*

Type Promotion

Elementwise functions and operators that accept more than one argument perform type promotion on their inputs, if the input dtypes are compatible.

The specification requires that all type promotion should happen independently of the input array values and shapes. This differs from the historical NumPy behavior where type promotion could vary for 0-D arrays depending on their values. For example (in NumPy 1.24):

```
>>> a = np.asarray(0., dtype=np.float64)
>>> b = np.asarray([0.], dtype=np.float32)
>>> (a + b).dtype
dtype('float32')
>>> a2 = np.asarray(1e50, dtype=np.float64)
>>> (a2 + b).dtype
dtype('float64')
```

This behavior is bug prone and confusing to reason about. In the array API specification, any `float32` array and any `float64` array would promote to a `float64` array, regardless of their shapes or values. NumPy is planning to deprecate its value-based casting behavior for NumPy 2.0 (see below).

Additionally, automatic cross-kind casting is not specified. This means that dtypes like `int64` and `float64` are not required to promote together. It also means that functions are not required to accept dtypes that imply a cross-kind cast: for instance floating-point functions like `exp()` or `sin()` are not required to accept integer dtypes, and arithmetic functions and operators like + and $\star$ are not required to accept boolean dtypes. Array libraries are not required to error in these situations, but array consumers should not rely on cross-kind casting in portable code. Cross-kind casting is better done explicitly using the `astype()` function. Automatic cross-kind casting is harder to reason about, can result in loss of precision, and often when it happens it indicates a bug in the user code.

Single argument functions and operators should maintain the same dtype when relevant, for example, if the input to `exp()` is a `float32` array, the output should also be a `float32` array.

For Python operators like + or $\star$, Python scalars are allowed. Python scalars cast to the dtype of the corresponding array's dtype. Cross-kind casting of the scalar is allowed in this specific instance for convenience (for example, `float64_array + 1` is allowed, and is equivalent to `float64_array + asarray(1., dtype=float64)`).

Optional Extensions

In addition to the above required functions, there are two optional extension sub-namespaces. Array libraries may chose to implement or not implement these extensions. These extensions are optional because they typically require linking against a numerical library such as a linear algebra library, and therefore may be difficult for some libraries to implement.

- `linalg` contains basic linear algebra functions, such as `eigh`, `solve`, and `qr`. These functions are designed to support "batching" (i.e., functions that accept matrices also accept stacks of matrices as a single array with more than 2 dimensions). The specification for the `linalg` extension is designed to be implementation agnostic. This means that things like keyword arguments that are specific to backends like LAPACK are omitted from the specified signatures (for example, NumPy's use of `UPLO` in the `eigh()` function). BLAS and LAPACK no longer hold a complete monopoly over linear algebra operations given the existence of specialized accelerated hardware, so these sorts of keywords are an impediment wide implementation across all array libraries.
- `fft` contains functions for performing Fast Fourier transformations.

Current Status of Implementations

Two versions of the array API specification have been released, v2021.12 and v2022.12. v2021.12 was the initial release with all important core array functionality. The v2022.12 release added complex number support to all APIs and the `fft` extension. A v2023 version is in the works, although no significant changes are planned so far. In 2023, most of the work around the array API has focused on implementation and adoption.

Strict Minimal Implementation (`numpy.array_api`)

The experimental `numpy.array_api` submodule is a standalone, strict implementation of the standard. It is not intended to be used by end users, but rather by array consumer libraries to test that their array API usage is portable.

The strictness of `numpy.array_api` means it will raise an exception for code that is not portable, even if it would work in the base `numpy`. For example, here we see that `numpy.array_api.sin(x)` fails for an integral array `x`, because in the array API spec, `sin()` is only required to work with floating-point arrays.

```
>>> import numpy.array_api as xp
<stdin>:1: UserWarning: The numpy.array_api submodule
is still experimental. See NEP 47.
>>> x = xp.asarray([1, 2, 3])
>>> xp.sin(x)
Traceback (most recent call last):
...
TypeError: Only floating-point dtypes are allowed in
sin
```

In order to implement this strictness, `numpy.array_api` employs a separate `Array` object, distinct from `np.ndarray`.

```
>>> a
Array([1, 2, 3], dtype=int64)
```

This makes it difficult to use numpy.array_api alongside normal numpy. For example, if a consumer library wanted to implement the array API for NumPy by using numpy.array_api, they would have to first convert the user's input numpy.ndarray to numpy.array_api.Array, perform the calculation, then convert back. This is in conflict with the fundamental design of the array API specification, which is for array libraries to implement the API and for array consumers to use that API directly in a library agnostic way, without converting between different array libraries.

As such, the numpy.array_api module is only useful as a testing library for array consumers, to check that their code is portable. If code runs in numpy.array_api, it should work in any conforming array API namespace.

array-api-compat

As discussed above, numpy.array_api is not a suitable way for libraries to use numpy in an array API compliant way. However, NumPy, as of 1.24, still has many discrepancies from the array API. A few of the biggest ones are:

- Several elementwise functions are renamed from NumPy. For example, NumPy has arccos(), etc., but the standard uses acos().
- The spec contains some new functions that are not yet included in NumPy. These clean up some messy parts of the NumPy API. These include:
  TODO: How complete do we need to be here?
  - np.unique is replaced with four different unique_* functions so that they always have a consistent return type.
  - np.transpose is renamed to permute_dims.
  - matrix_transpose is a new function that only transposes the last two dimensions of an array.
  - np.norm is replaced with separate matrix_norm and vector_norm functions in the linalg extension.
  - np.trace operates on the first two axes of an array but the spec linalg.trace operates on the last two.

There are plans in NumPy 2.0 to fully adopt the spec, including changing the above behaviors to be spec-compliant. But in order to facilitate adoption, a new library array-api-compat has been written. array-api-compat is a small, pure Python library with no hard dependencies that wraps array libraries to make the spec complaint. Currently NumPy, CuPy, and PyTorch are supported.

array-api-compat is to be used by array consumer libraries like scipy or scikit-learn. The primary usage is like

```
from array_api_compat import array_namespace

def some_array_function(x, y):
    xp = array_api_compat.array_namespace(x, y)

    # Now use xp as the array library namespace
    return xp.mean(x, axis=0) + 2*xp.std(y, axis=0)
```

array_namespace is a wrapper around x.__array_namespace__(), except whenever x is a NumPy, CuPy, or PyTorch array, it returns a wrapped module that has functions that are array API compliant. Unlike numpy.array_api, array_api_compat does not use separate wrapped array objects. So in the above example, the if the input arrays are np.ndarray, the return array will be a np.ndarray, even though xp.mean and xp.std are wrapped functions.

While the long-term goal is for array libraries to be completely array API compliant, array-api-compat allows consumer libraries to use the array API in the shorter term against libraries like NumPy, CuPy, and PyTorch that are "nearly complaint".

array-api-compat has already been successfully used in scikit-learn's LinearDiscriminantAnalysis API (https://github.com/scikit-learn/scikit-learn/pull/22554).

Compliance Testing

The array API specification contains over 200 function and method definitions, each with its own signature and specification for behaviors for things like type promotion, broadcasting, and special case values.

To facilitate adoption by array libraries, as well as to aid in the development of the minimal numpy.array_api implementation, a test suite for the array API has been developed. The array-api-tests test suite is a fully featured test suite that can be run against any array library to check its compliance against the array API specification. The test suite does not depend on any array library—testing against something like NumPy would be circular when it comes time to test NumPy itself. Instead, array-api-tests tests the behavior specified by the spec directly.

When running the tests, the array library is specified using the ARRAY_API_TESTS_MODULE environment variable.

This is done by making use of the hypothesis Python library. The consortium team has upstreamed array API support to hypothesis in the form of the new hypothesis.extra.array_api submodule, which supports generating arrays from any array API compliant library. The test suite uses these hypothesis strategies to generate inputs to tests, which then check the behaviors outlined by the spec automatically. Behavior that is not specified by the spec is not checked by the test suite, for example the exact numeric output of floating-point functions.

Utilizing hypothesis offers several advantages. Firstly, it allows writing tests in a way that more or less corresponds to a direct translation of the spec into code. This is because hypothesis is a property-based testing library, and the behaviors required by the spec are easily written as properties. Secondly, it makes it easy to test all input combinations without missing any corner cases. Hypothesis automatically handles generating "interesting" examples from its strategies. For example, behaviors on 0-D or size-0 arrays are always checked because hypothesis will always generate inputs that match these corner cases. Thirdly, hypothesis automatically shrinks inputs that lead to test failures, producing the minimal input to reproduce the issue. This leads to test failures that are more understandable because they do not incorporate details that are unrelated to the problem. Lastly, because hypothesis generates inputs based on a random seed, a large number of examples can be tested without any additional work. For instance, the test suite can be run with pytest --max-examples=10000 to run each test with 10000 different examples (the default is 100). These things would all be difficult to achieve with an old-fashioned "manual" test suite, where explicit examples are chosen by hand.

The array-api-tests test suite is the first example known to these authors of a full featured Python test suite that runs against multiple different libraries. It has already been invaluable in practice for implementing the minimal numpy.array_api implementation, the `array-api-compat` library, and for finding presidencies from the spec in array libraries including NumPy, CuPy, and PyTorch.

Future Work

The focus of the consortium for 2023 is on implementation and adoption.

NumPy 2.0, which is planned for release in late 2023, will have full array API support. This will include several small breaking changes to bring NumPy inline with the specification. This also includes, NEP 50, which fixes NumPy's type promotion by removing all value-based casting. A NEP for full array API specification support will be announced later this year.

SciPy 2.0, which is also being planned, and will include full support for the array API across the different functions. For end users this means that they can use CuPy arrays or PyTorch tensors instead of NumPy arrays in SciPy functions, and they will just work as expected, performing the calculation with the underlying array library and returning an array from the same library.

Scikit-learn has implemented array API specification support in its `LinearDiscriminantAnalysis` class and plans to add support to more functions.

Work is underway on an array API compliance website. (*TODO*)

There is a similar effort being done by the same Data APIs Consortium to standardize Python dataframe libraries. This work will be discussed in a future paper and conference talk.

TODO: Add references

Conclusion

TODO