

Quantity 2.0

Report by Nathaniel Starkman

Abstract

In this document we outline and consider the design of a new `Quantity` class for Astropy. In contrast to the current limitation for only `NumPy` arrays, this new class will be able to be used with any array library, including NumPy, JAX, CuPy, and others. The new `Quantity` class architecture is also designed for the future: compatible with emerging Python data standards, built for easy interoperability and extensibility, and designed for static typing and associated compiled speedups.

We provide three reference libraries that demonstrate how to build the `Quantity 2.0` class and support `Dask` arrays, to build an Array-API function library that works with JAX arrays in a `Quantity`, and to build a JAX-supporting `Quantity`.

Reference Libraries:

1. <https://github.com/nstarman/units>
2. <https://github.com/GalacticDynamics/array-api-jax-compat>
3. <https://github.com/GalacticDynamics/jax-quantity>

Introduction

The `Quantity` is a core component of the Astropy, and is used throughout the library and by users. The usefulness of `Quantity`, and the units library in general, is such that many other libraries have Astropy as a dependency just to use `Quantity`. One reason for `Quantity`'s popularity is that it deeply interoperable with `numpy.ndarray`, the primary numerical array library in Python, and keeps a focused scope, adding only a unit attribute and making `numpy.ndarray` operations unit-aware.

```
In [1]: import astropy.units as u
import numpy as np

q = u.Quantity(np.array([-1, 0, 1]), unit=u.dimensionless_unscaled)

np.arccos(q).to(u.Unit(np.pi * u.rad))
```

```
Out[1]: [1, 0.5, 0] 3.1415927 rad
```

When `Quantity` was first developed, `numpy.ndarray` was essentially the only numerical array library in Python. For this good reason, `Quantity` is a subclass of `numpy.ndarray`. However, in the years since `Quantity` was first developed, the Python computing landscape has changed. Many numpy-like libraries exist for different use cases: `Dask` for distributed computing, `CuPy` for GPU, `xarray` for labeled data, `JAX` for automatic differentiation and GPU acceleration, `zarr` for compressed arrays, etc. However, since it is a `numpy` subclass, `Quantity` is incompatible with all these libraries, preventing users from choosing between the most suitable array computation library and having units. In other words, users working on larger-than memory data or doing machine learning, among other use cases, struggle to use Astropy. This is a problem; and one that we can solve.

Moreover, now is the best time to solve this problem. A consortium of interested parties - - companies and open-source projects -- have formed the Data APIs consortium to develop common standards for data processing in Python. Their first project is the Array-API, which aims to define a common interface for array libraries. The Array-API is in continual development, but has already gone through several iterations and is seeing widespread adoption. `NumPy` is in the end stages of transitioning to a new `numpy.ndarray` that implements the Array-API they helped develop. This transition in `NumPy` and in the Python data ecosystem more broadly presents the need and opportunity for `Quantity` 2.0.

In this document we outline and consider the design of a new `Quantity` class for Astropy. In contrast to the current limitation of only `numpy` arrays, this new class will be able to be used with any array library, including `NumPy`, `JAX`, `CuPy`, and others.

- In the first section we outline all the requirements for the new `Quantity` class.
- In the second section we discuss elemental building blocks of the new `Quantity` class and the opportunity they present to work with other units libraries to create a more interoperable units ecosystem. This section includes a discussion of the development of the Array-API, which is promising to significantly simplify the work required for `Quantity` to interoperate with array libraries.
- In the third section we discuss the actual construction of the new `Quantity` class.
- In the last section we outline a roadmap for transitioning to the new `Quantity` class.

As in all sections, the three reference libraries provide concrete implementations of the concepts discussed and we refer interested readers to these libraries.

Requirements

The current `Quantity` class is well designed and has a clear scope. However, that

design and that scope were developed with some assumptions that no longer hold true, or that are no longer the best choice given more recent developments in the Python ecosystem.

We list out the requirements for the new `Quantity` then discuss each in turn. The requirements for `Quantity 2.0` are:

1. `Quantity` must be interoperable with many array libraries, without losing any support for `NumPy`.
2. `Quantity` must be easily extensible to new array libraries so that users and developers can easily add support for new libraries as needed.
3. `Quantity` must have a clear API. In particular, this means
 - A. `Quantity` must be easily subclassable, e.g. with minimal abstract methods to override.
 - B. `Quantity` must have minimal "magic", composed instead of explicit blocks with clear semantics that are easy to understand.
 - C. `Quantity` must be deeply introspectable. All layers of `Quantity`, from the array library to the unit operations, must be accessible to users and developers.
4. `Quantity` must be statically typable.
5. `Quantity` must be fast: to import, construct, and use.

1. Interoperability

`Quantity` must be interoperable with many array libraries, without losing any support for `NumPy`. This is the most important requirement and primary motivation for developing `Quantity 2.0`. The current `Quantity` is a subclass of `numpy.ndarray`. Therefore, in its current form, `Quantity` will cast the input array to a `NumPy` array.

```
In [2]: # A NumPy array is kept as a NumPy array
x = np.array([1, 2, 3])
print(type(x), type(u.Quantity(x).value))

<class 'numpy.ndarray'> <class 'numpy.ndarray'>
```

```
In [3]: # A array-like object is cast to a NumPy array.
# This is a desirable behavior.
x = [1, 2, 3]
print(type(x), type(u.Quantity(x).value))

<class 'list'> <class 'numpy.ndarray'>
```

```
In [4]: # However, Array objects are ALSO cast to NumPy arrays.
# This is not desirable.
```

```
import dask.array as da

x = da.ones(3)
print(type(x), type(u.Quantity(x).value))
```

```
<class 'dask.array.core.Array'> <class 'numpy.ndarray'>
```

For `Quantity` 2.0 we want to remove this limitation and allow `Quantity` to be used with any (most) array library. The new `Quantity` must be able to be constructed from an array from any array library, and that `Quantity` will not convert the input array to a `NumPy` array.

```
>>> x = da.ones(3)
>>> print(type(x), type(Quantity2(x).value))
<class 'dask.array.core.Array'> <class 'dask.array.core.Array'>
```

2. Extensibility

`Quantity` must be easily extensible to new array libraries so that users and developers can easily add support for new libraries as needed. As it is currently implemented, `Quantity` is a subclass of `numpy.ndarray` and thus has automatic support for `NumPy`. In `Quantity` 2.0, we want to make it easy to add support for new array libraries, such as `Dask`, `CuPy`, `JAX`, while still maintaining first-class support for `NumPy`. The Consortium for Python Data API Standards's Array-API will make array libraries have a more common interface, but there will still be differences between libraries that will require some work to support. We want to make this work as easy as possible.

Conceptually supporting multiple libraries means that there must be a mechanism to register interfaces connecting `Quantity` to an array library.

e.g. for a hypothetical interface between `Quantity` and `Dask` arrays we might implement:

```
from astropy.units import register_quantity_interface
```

```
register_quantity_interface(dask.array.Array,
QuantityDaskInterface)
```

Then `Quantity` will know how to interact with `Dask` arrays.

```
>>> x = da.ones(3)
>>> q = Quantity(x, u.m)
>>> print(type(x), type(q))
<class 'dask.array.core.Array'> <class 'dask.array.core.Array'>
```

```
>>> q.to_dask_dataframe() # a Dask Array method
Quantity(value=Dask Series Structure:
npartitions=1
```

```
0    int64
2    ...
dtype: int64
Dask Name: from-dask-array, 2 graph layers, unit=Unit("m"))
```

3. Clear API

`Quantity` must have a clear API.

When `Quantity` only supports `NumPy` arrays, it is easy to explain and teach the API. `NumPy` is a well-known and well-documented library, and `Quantity` only adds a few attributes and methods to the `NumPy` arrays. However, when `Quantity` supports multiple array libraries, both the implementation of `NumPy` compatibility and the method of teaching how `Quantity` works will change. We want to make sure that the new `Quantity` is easy to understand and teach. Moreover, with a clear API we want to make sure that the new `Quantity` is easy to subclass and extend.

`Quantity` has a lot of intrinsic "magic", where it just works. Some of the magic is due to the concept of "duck" typing, where objects that common methods and attributes are interchangeable.

```
In [5]: from typing import SupportsAbs
```

```
def func(x: SupportsAbs):
    return abs(x)
```

```
In [6]: x = np.array([1, -2, 3])
print(x, "->", func(x))
```

```
[ 1 -2  3] -> [1 2 3]
```

```
In [7]: dx = da.asarray(x)
print(dx.compute(), "->", func(dx).compute())
```

```
[ 1 -2  3] -> [1 2 3]
```

```
In [8]: q = u.Quantity(x, unit=u.m)
print(q, "->", func(q))
```

```
[ 1. -2.  3.] m -> [1. 2. 3.] m
```

However, much of the magic is how `Quantity` implements method overrides for `NumPy` methods.

```
In [9]: np.abs(q)
```

```
Out[9]: [1, 2, 3] m
```

Each library has its own way of overriding methods, and even where they share semantics, as in the Array-API specification, `Quantity` has to support all the various methods. This leads to a lot of complex code that is hard to understand. Therefore we want to minimize the amount of magic in `Quantity` and implement very explicit blocks with clear semantics that are easy to understand. The "magic" should be in interaction between these clear blocks.

Finally, `Quantity` must be deeply introspectable. This is related to the requirement for minimal magic and explicit blocks. All layers of `Quantity`, from the array library to the unit operations, must be accessible to the user.

5. Static typing

Many array libraries, including `NumPy` and `JAX` support static typing. Thus an intrinsic part of supporting and being interoperable with these libraries is to also support static typing.

Consider a function `func` that takes a `NumPy` array with `float64` dtype elements and returns that array with the elements squared.

```
In [10]: from numpy.typing import NDArray

def round(x: NDArray[np.float64]) -> NDArray[np.float64]:
    return x**2
```

If we want to have a `Quantity` version of `func` that has the same requirements we want to be able to communicate this to the user and the type checker.

```
def ground(x: Quantity[NDArray[np.float64]]) ->
Quantity[NDArray[np.float64]]:
    return x**2
```

As will be discussed later, we will be able to write much more general static types that are compatible with many array libraries, not only `NumPy`.

4. Speed

`Quantity` must be fast: to import, construct, and use. This is an obvious but difficult requirement. A lot of work has been done by Python and array libraries like `NumPy` to make to speed up numerical computations. Adding a units layer around each array operation can drastically slow down computation unless care is taken to make the units layer as fast as possible.

There are multiple ways to make `Quantity` fast. Some are related to the clear API and others to the static typing. We will discuss these in subsequent sections.

Building Blocks

Having outlined the requirements for `Quantity` 2.0, we now discuss the building blocks that will be used to construct the new `Quantity` class.

We will discuss the following building concepts:

1. The Array-API and what this means for implementing method overrides for array libraries.
2. An API for `Quantity` that is built on top of the Array-API. This API will allow users to abstractly describe the `Quantity` library and can enable interoperability with other `Quantity`-vending libraries, like `pint`.
3. Inheritance versus composition. We will discuss the pros and cons of each approach and how they can be combined to make a flexible and extensible `Quantity` class.
4. The `Quantity` class itself. We will discuss the implementation of the new `Quantity` class and how it will be built on top of the Array-API and the `Quantity` API.
5. Thoughts for the units library.

Array-API

The Array-API is a specification for a common interface for array libraries (see https://data-apis.org/array-api/latest/purpose_and_scope.html). With the Array-API users will be able to write code that works with any implementing library.

How does this work? The Array-API specifies that all array objects have a new `__array_namespace__` method that returns an object (generally a module) that contains all the functions that are part of the Array-API.

```
class Array:
    def __array_namespace__(self, ...):
        return array_namespace_module
```

where the `array_namespace_module` contains all the functions that are part of the Array-API.

```
# array_namespace_module.py
```

```
def cos(x):
    ...
```

```
def sin(x):
```

...

...

By calling `__array_namespace__` method of an array object, users can get access to all the Array-API functions.

We start with a simple example using the new NumPy Array-API library.

```
In [11]: # from https://github.com/nstarman/array_api since the Array-API does not
# currently have an installable implementation. This repo provides `Protocol
# objects for static and runtime type checking.
from array_api import Array

def func(x: Array) -> Array:
    xp = x.__array_namespace__()
    return xp.add(xp.abs(x), xp.ones_like(x))
```

Now we can evaluate this function with any array library that implements the Array-API.

```
In [12]: from numpy import array_api as npx

x = npx.asarray([1, -2, 3])
func(x)
```

```
/var/folders/lt/4kxx3wjs4y17_nrgc1psr2fr0000gn/T/ipykernel_73628/125159778.p
y:1: UserWarning: The numpy.array_api submodule is still experimental. See N
EP 47.
    from numpy import array_api as npx
```

```
Out[12]: Array([2, 3, 4], dtype=int64)
```

The function `func` is a simple example and it is hardly surprising that a NumPy array works. A more interesting example is to build a simple Array class of our own and see if it works.

```
In [13]: from dataclasses import dataclass, replace
from types import SimpleNamespace
from typing import Any

@dataclass
class MyArray(Array):
    """A simple wrapper class that implements the Array-API."""
    value: Array

    def __array_namespace__(self, *, api_version: Any = None) -> SimpleNamesp
        return myxp
```



```
# Instead of making a whole new module we can just use a SimpleNamespace
# object and fill it with functions that operate on our custom array type.
# This is for illustration purposes only, in practice you would want to
# make a proper module.
myxp = SimpleNamespace(
    abs=lambda x: replace(x, value=x.value.__array_namespace__().abs(x.value),
    ones_like=lambda x: replace(
        x, value=x.value.__array_namespace__().ones_like(x.value)
    ),
    add=lambda x, y: MyArray(x.value.__array_namespace__().add(x.value, y.value))
)
```

```
In [14]: func(MyArray(x))
```

```
Out[14]: MyArray(value=Array([2, 3, 4], dtype=int64))
```

Excellent! The Array-API is working as expected and we can easily build "universal" functions. The Array-API means that array-consuming libraries, like `SciPy` (except for their `sparse` classes), will be able to adapt their code to work with any array types. All they need to do is to get the correct array namespace and then they can use the functions and pass the namespace around to other functions. This is succinctly demonstrated in `func`.

While this method of building functions works for *array-consuming* libraries, it potentially changes the paradigm for *array-wrapping* libraries. The Array-API is suggested to be *strict* in the input types it accepts. What does this mean? It means that if a function is written to accept an `Array` then it won't work with a `MyArray` object (or `Quantity`). This is counter to the current override mechanisms in `NumPy` which allow for duck-type inputs to `NumPy` functions. We note that the Array-API does not *require* this strictness but for reasons of clarity and speed it is recommended.

```
In [15]: # Showing that the Array-API is different from
try:
    nxp.abs(MyArray(x))
except Exception:
    print("we will need a different approach.")
```

we will need a different approach.

Universal Libraries

As demonstrated above it is easy to write functions that work with any array. Despite this, functions from a library that are wihing the Array-API specification are not necessarily universal since they are generally strict in the input types they accept.

`NumPy` has historically been a universal library, in that any array library can register into `NumPy`'s override mechanisms (NEP 13 & 18) and then `NumPy` can operate on those

objects. NumPy is in the process of transitioning to the Array-API and it is unclear to what extent NumPy will continue to be a universal library.

Currently NumPy's Array-API implementation is not a universal library. This is demonstrated in `ones_like`, where there is no way to dispatch to a different implementation of `ones_like` for a given array library.

<https://github.com/numpy/numpy/blob/7b0c33ef8ca52f1446c39d24f2c4e054802a7a56/nurL288>

```
def ones_like(
    x: Array, /, *, dtype: Optional[Dtype] = None, device:
Optional[Device] = None
) -> Array:
    """
    Array-API compatible wrapper for :py:func:`np.ones_like`
    <numpy.ones_like>`.
```

```
    See its docstring for more information.
    """
```

```
    from ._array_object import Array
```

```
    _check_valid_dtype(dtype)
    if device not in ["cpu", None]:
        raise ValueError(f"Unsupported device {device!r}")
    return Array._new(np.ones_like(x._array, dtype=dtype))
```

Given NumPy's history and commitment to interoperability, we think it highly likely that NumPy will continue to be a universal library. Even if this is the case it is still worthwhile to discuss the implementation of a universal library in the context of the Array-API specification.

The Array-API specification makes implementing a universal library fairly straightforward. In a previous example it was shown how an array-consuming library can use the Array-API to write functions that work with any array.

```
In [16]: def func(x: Array) -> Array:
        xp = x.__array_namespace__()
        return xp.abs(x)
```

A universal library can use the same mechanism to write functions *within* the Array-API specification

```
In [17]: def cos(x: Array) -> Array:
        """Universal cosine function."""
```

```
xp = x.__array_namespace__()  
return xp.cos(x)
```

For the example of `ones_like` we can similarly write:

(https://github.com/nstarman/array_api/blob/a6fc6b5584ab45b24a0c11e292d6755283c16cL168)

```
def ones_like(  
    x: Array, /, *, dtype: DType | None = None, device: Device |  
    None = None  
    ) -> Array:  
    xp = x.__array_namespace__() # get the correct namespace  
    return xp.ones_like(x, dtype=dtype, device=device)
```

This example implementation is adapted from `nstarman/array_api`, which was used earlier to write the static type hints for `Array`. This repository is one of the many reference implementations produced to help develop the `Quantity 2.0` design. We refer the reader to the repository for more examples of how to write universal functions.

The most logical place for this universal library to live is in the Array-API compatibility library (<https://github.com/data-apis/array-api-compat>) maintained by the Consortium for Python Data API Standards. The `array-api-compat` library has submodules to implement support for the Array-API for different array libraries that are not yet compatible with the Array-API. The universal library would be a submodule of `array-api-compat` and can be used by other libraries to abstract over all the array libraries that support the Array-API. We note again that `NumPy` might choose to continue to be a universal library and would implement a similar dispatch mechanism.

Recommendation: Working with ``data-apis/array-api-compat`` and/or ``NumPy`` to implement an universal library.

Quantity Array-API Library

`Quantity` is an array-wrapping library. Let us assume that all the arrays that `Quantity` wraps are Array-API compatible. Then `Quantity` can easily hook into the Array-API style dispatch. This has already been demonstrated with `MyArray`, but we repeat the implementation here.

```
class Quantity:  
    value: Array  
    unit: Unit  
  
    def __array_namespace__(self, ...):  
        return quantity_array_namespace
```

```
# quantity_array_namespace.py
```

```
def cos(x: Quantity) -> Quantity:
    xp = get_wrapped_namespace(x) # e.g. `numpy`, or `jax.numpy`
    value = ... # deal with unit stuff like converting to radians
    return Quantity(xp.cos(value), unit=u.dimensionless_unscaled)

def add(x: Quantity, y: Quantity) -> Quantity:
    xp = get_wrapped_namespace(x, y) # e.g. `numpy`, or
    `jax.numpy`
    return Quantity(xp.add(x.value, y.to_value(x.unit)),
    unit=x.unit)
```

Now a user using the universal library discussed above can use Astropy's `Quantity`.

```
>>> from array_api_compat import universal
>>> from dask.array import asarray
>>> from math import pi

>>> q = Quantity(asarray([0, 0.5, 1]) * pi, unit=u.rad)

>>> universal.cos(q)
Quantity([1, 0, 1], unit=Unit(''))
```

Where `universal.cos` redirects to `quantity_array_namespace.cos`. This handles the units and passes the value computation to `dask`.

The above was an illustrative example only.

For a real example let's turn to the reference library `units` (<https://github.com/nstarman/units>) made in part for this report. The `units` library contains a working implementation of both `Quantity` and associated `array_namespace`. Moreover, `units` implements a registration system for extensions, as discussed in the Extensibility section. In this example we will see how `Quantity` can natively work with `dask` arrays.

```
In [18]: import astropy.units as apyu
import numpy as np
import dask.array as da

# isort: split
import units as u
from units import array_namespace as xp
import array_api as universal
```

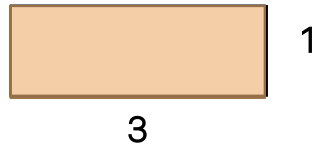
```
In [19]: x = da.from_array(np.array([1, 2, 3]))
```

```
In [20]: q = u.Quantity(x, apyu.deg)
```

```
# Demonstrating that this Quantity preserves Dask Arrays
q.value
```

Out[20]:

	Array	Chunk
Bytes	24 B	24 B
Shape	(3,)	(3,)
Dask graph	1 chunks in 1 graph layer	
Data type	int64 numpy.ndarray	



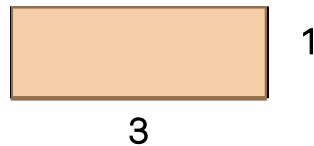
We can use the Quantity specific Array-API namespace...

```
In [21]: print(type(xp.cos(q)))
xp.cos(q)
```

```
<class 'units._quantity.core.Quantity'>
```

Out[21]:

	Array	Chunk
Bytes	24 B	24 B
Shape	(3,)	(3,)
Dask graph	1 chunks in 3 graph layers	
Data type	float64 numpy.ndarray	

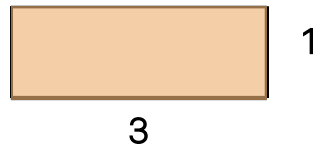


Or the universal library...

```
In [22]: universal.cos(q)
```

Out[22]:

	Array	Chunk
Bytes	24 B	24 B
Shape	(3,)	(3,)
Dask graph	1 chunks in 3 graph layers	
Data type	float64 numpy.ndarray	



Recommendation: Implementing a `units.array_api` module for `Quantity`.

Further Reading

1. The Data-APIs Array-API
2. The author's `array_api` library (https://github.com/nstarman/array_api) that implements static and run-time type hints for the Array-API
3. The author's `units` library (<https://github.com/nstarman/units>) that implements `Quantity` and associated Array-API namespace.

In []:

Quantity API

Since [PEP 544](#) was implemented in Python 3.8, Python can now separate the description of an API from its implementation. This is done using the `typing.Protocol` class. Protocols are essentially abstract base classes that don't require inheritance. Instead, they are used to describe the interface of a class. Any class that implements the interface is considered a subclass of the Protocol and the class' instances are likewise instances of the Protocol. This is called "structural subtyping" or "duck typing".

As an example, consider the following Protocol that describes the interface of an object that has a name and a value.

In [23]: `from typing import Protocol, runtime_checkable`

```
@runtime_checkable
class NamedValue(Protocol):
    """API for Quantity."""

    value: float
    name: str
```

This Protocol can be used to annotate a function that takes a `NamedValue` duck-type as an argument.

In [24]: `def print_value(x: NamedValue) -> None:
 print(f"{x.name}: {x.value}")`

Any class that has a `value` attribute of type `float` and a `name` attribute of type `str` is considered a subclass of `NamedValue` and can be used as an argument to `print_value`.

In [25]: `class NamedValueClass1:
 def __init__(self, name: str, value: float):
 self.name = name
 self.value = value`

```
v = NamedValueClass1("foo", 1.0)

isinstance(v, NamedValue) # True

print_value(NamedValueClass1("foo", 1.0))
```

foo: 1.0

Or

```
In [26]: from typing import NamedTuple

class NamedValueClass2(NamedTuple):
    name: str
    value: float

print_value(NamedValueClass2("foo", 1.0))
```

foo: 1.0

Note again that neither `NamedValueClass1` nor `NamedValueClass2` inherit from `NamedValue`.

Let's consider how we might use this to describe the interface of `Quantity`. The first thing we need are the two attributes of `Quantity`: the `value` and the `unit`. The `value` is a numeric type, and the `unit` is a unit type. We can use `typing.TypeVar` to represent these types. For now we will not constrain the type, e.g. we will not require that the `value` be array-like, or that the `unit` be a unit object. This will be discussed later.

```
In [27]: from typing import TypeVar

# TypeVar is used to represent a generic type.
Value = TypeVar("Value")
Unit = TypeVar("Unit")

class QuantityAPI(Protocol[Value, Unit]):
    """API for Quantity."""

    value: Value
    unit: Unit
```

This API describes a class that has two attributes: `value` and `unit`. The class is generic over the types of these attributes. This means that `QuantityAPI` can be used to describe a class of many different `value / unit` types, depending on the type arguments passed to `QuantityAPI`. For example, `QuantityAPI[int, str]` describes a class with an `int` `value` and a `str` `unit`. One of the benefits of this

approach is that we can use the arguments to `QuantityAPI` to describe how a function handles different data types. For example, consider the following function that takes two `QuantityAPI` objects and returns their sum, but doesn't work for integer datatypes.

```
In [28]: def wierd_add(
          x: QuantityAPI[float | int, Any], y: QuantityAPI[float | int, Any]
        ) -> QuantityAPI[float, Any]:
          """Add two Quantity objects."""
          if isinstance(x.value, int) or isinstance(y.value, int):
              raise TypeError("Can only add Quantity objects with float values.")
          return type(x)(x.value + y.value, unit=x.unit)
```

This is a formal way of saying that the function only works for `QuantityAPI` objects with `float` values.

As described thus far, `QuantityAPI` is generic first over the `value` type and then over the `unit` type. This is not the only way to do this. We could also make `QuantityAPI` generic over the `unit` type first, and then over the `value`. This would look like this:

```
In [29]: class QuantityAPI(Protocol[Unit, Value]): # Note the order of the type argu
          """API for Quantity."""

          value: Value
          unit: Unit
```

The unit-first approach is how `Quantity` is currently implemented. The current implementation is not recognized by static type checkers, which must be fixed. Which should be first, the `value` or the `unit`? Given the way Python's type system works the value-first approach is recommended.

Recommendation: Change `Quantity` to be generic over the value, then the unit.

In Python only `type`s may be used as type parameters. Astropy's units implementation is not at the type-level since units are **instances** of a type (`UnitBase` / `FunctionUnit` /etc.) not a type themselves. Therefore the following annotation is not a valid type hint

```
def stringify_km(unit: u.km) -> str
    return str(unit)
```

But this would be (approximately since `astropy.units` has many base classes).

```
def stringify_km(unit: u.UnitBase) -> str
    return str(unit)
```

Unfortunately this does not allow us to write the specific unit, only that something is an

instance of a unit. The advantage of being able to statically describe unit transformations is that a type checker could verify the full flow of units in a program, ensuring correctness. However, this would only be possible in Astropy by dramatically rewriting the entire library such that all units were types, not instances. Moreover, new types would have to be dynamically created for composite units, which would be a significant performance hit and also not compatible with static type checkers. We do not recommend implementing a type-level units system in Astropy.

How then can we type hint the `unit` attribute of `Quantity`? We can continue to treat the unit type as metadata, e.g. a `typing.Annotation`. Libraries like `nptyping` and `jaxtyping` have developed ways to annotate the shapes of arrays as string type annotations and to check these annotations. In the case of `jaxtyping` the annotations are checked at compile-time, incur no runtime overhead. This is a promising approach for Astropy to take. Metadata arguments (like the unit) to `Annotated` are always the second argument, proceeding the type argument (here the value). Therefore we should preserve this value-first approach in `Quantity`.

Recommendation: Implement a `jaxtyping`-like layer for units and dimension checks.

Recommendation: For backwards compatability add a ``astropy.units.typing.QuantityU`` Generic that fills in ``Any`` for the value and accepts the unit as the first type parameter (like ``numpy.typing.NDArray`` fills in the shape parameter, leaving the dtype type parameter).

As described thus far, `QuantityAPI` cannot be used in `isinstance` or `issubclass` checks. This is because `QuantityAPI` is not runtime-checkable. This can be fixed by adding the `@runtime_checkable` decorator to the class.

```
In [30]: from typing import runtime_checkable, Protocol, TypeVar

Value = TypeVar("Value")
Unit = TypeVar("Unit")

@runtime_checkable
class QuantityAPI(Protocol[Value, Unit]): # Note the order of the type arguments
    """API for Quantity."""

    value: Value
    unit: Unit
```

Next, in the requirements we said that `Quantity` should be immutable. Immutable

classes are much safer than mutable classes, and are easier to type check and optimize, e.g. by `mypyc` compilation or cached computations. To make `QuantityAPI` immutable we can change the type hints to properties.

```
In [31]: @runtime_checkable
class QuantityAPI(Protocol[Value, Unit]):
    """API for Quantity."""

    @property
    def unit(self) -> Unit:
        ...

    @property
    def value(self) -> Value:
        ...
```

Now we have the basic API for `Quantity`. However, this API does not describe the full API of `Quantity`. Let's consider some of the other methods.

Currently `Quantity` has a `to` method that converts the `Quantity` to a new unit.

```
In [32]: from typing import overload

ToUnit = TypeVar("ToUnit")

@runtime_checkable
class QuantityAPI(Protocol[Value, Unit]):
    """API for Quantity."""

    @property
    def unit(self) -> Unit:
        ...

    @property
    def value(self) -> Value:
        ...

    # -----

    def to_unit(self, unit: ToUnit) -> QuantityAPI[Value, ToUnit]:
        """Return the Quantity in the given units."""
        ...

    def to_unit_value(self, unit: ToUnit) -> Value:
        """Return the value of the Quantity in the given units."""
        return self.to_unit(unit).value # default implementation

    # -----
    # Arithmetic. (This is only a subset of the full arithmetic API.)
```

```

def __add__(self, other: QuantityAPI[Value, Unit]) -> QuantityAPI[Value,
    """Add two Quantity objects."""
    ...

@overload # Multiply by a Quantity
def __mul__(self, other: QuantityAPI[Value, Unit]) -> QuantityAPI[Value,
    ...

@overload # Multiply by a non-Quantity
def __mul__(self, other: Value) -> QuantityAPI[Value, Unit]:
    ...

@overload # Multiply by a Unit
def __mul__(self, other: Unit) -> QuantityAPI[Value, Unit]:
    ...

def __mul__(
    self, other: QuantityAPI[Value, Unit] | Value | Unit
) -> QuantityAPI[Value, Unit]:
    """Multiplication."""
    ...

```

We recommend that the basis of the new `Quantity` class be a Protocol that describes the interface of `Quantity`. This API definition is important for three reasons, that will be discussed in turn. First, within Astropy we can type hint and check duck-types of `Quantity`, like `Column`. Second, we can allow users to define their own `Quantity`-like classes that can be used interchangeably with `Quantity`. Third, if we develop this API in collaboration with other units libraries, we can create a common interface that allows for interoperability between units libraries.

Recommendation: Working with other stakeholders, develop a `Quantity` API.

As discussed in the context of the type parameter ordering (value-first versus unit-first) there is not a way to specify, statically, operations on units. For example we can't tell the unit system that multiplying two quantities will result in a new quantity with the units multiplied, only that there's some unit. The `Quantity` API does not solve this problem, but it does provide the opportunity to build the unit-interpreting infrastructure on top of the API and to make it compatible with other units libraries.

Recommendation: Working with other stakeholders, implement the ``jaxtyping``-like layer for units and dimension checks on top of the `Quantity` API.

Composition over Inheritance

We at Astropy want to support other numerical array libraries, without losing first-class

support for NumPy. When `Quantity` was developed and built on NumPy, the best way to interoperate with `numpy.ndarray` was to subclass it. This subclassing approach is the primary cause of the current incompatibility with other array libraries. However, in subsequent years `numpy` has further developed two new protocols for interoperability: `__array_ufunc__` in NEP 13 and `__array_function__` in NEP 18 that allow for any class, regardless of inheritance, to interoperate with `numpy` functions. These developments allow for an alternative approach to `Quantity` that is compatible both with NumPy and other array libraries.

As a demonstration of NEP 13 and 18, consider the following class that implements a `Mixin` for `Quantity` and uses the Array-API to implement `__array_ufunc__` and `__array_function__`. This class demonstrates how composition can be used to implement `Quantity`'s `NumPy` compatibility and how the Array-API fits naturally into this approach.

```
In [33]: from typing import Protocol
from mypy_extensions import trait
from array_api import Array as ArrayAPI

@trait
class NumPyMixin(Protocol): # TODO: proper type hints
    """Mixin for NumPy NEP13,18-style overloading."""

    def __array_ufunc__(
        self: ArrayAPI, ufunc: Any, method: Any, *inputs: Any, **kwargs: Any
    ) -> Any:
        if method != "__call__":
            # TODO: dispatch to something other than `xp`
            return NotImplemented

        xp = self.__array_namespace__()
        func = getattr(xp, ufunc.__name__)
        return func(*inputs, **kwargs, _xp=np) # TODO: allow general xp

    def __array_function__(
        self: ArrayAPI,
        func: Any,
        types: Any,
        args: tuple[Any, ...],
        kwargs: dict[str, Any],
    ) -> Any:
        xp = self.__array_namespace__()
        func = getattr(xp, func.__name__)
        return func(*args, **kwargs, _xp=np) # TODO: allow general xp
```

This `Mixin` class is implemented in the reference library `units`.

```
In [34]: import astropy.units as apyu
```

```
import numpy as np
import dask.array as da

# isort: split
import units as u
from units import array_namespace as xp
```

In [35]: `u.Quantity.mro()`

```
Out[35]: [units._quantity.core.Quantity,
          units._quantity.base.AbstractQuantity,
          units._quantity.base.LegacyQuantityMixin,
          units._quantity.base.NumPyMixin,
          typing.Protocol,
          typing.Generic,
          object]
```


Using the `Mixin`, `NumPy` can operate on `Quantity` objects by calling `__array_ufunc__` and `__array_function__` methods. These methods pass through to `Quantity`'s `__array_namespace__`, which in turn passes through to the `value`'s `__array_namespace__`. In the following example we use `dask` arrays.

```
In [36]: x = da.from_array(np.array([1, 2, 3]))
         q = u.Quantity(x, apyu.deg)

         np.cos(q)
```

Out[36]:

	Array	Chunk
Bytes	24 B	24 B
Shape	(3,)	(3,)
Dask graph	1 chunks in 3 graph layers	
Data type	float64 numpy.ndarray	



Again, the sequence of `cos` calls is `numpy.cos` -> `units.array_namespace.cos` -> `dask.array.cos`.

The alternative approach to inheritance is composition. Instead of `Quantity` being a subclass of `numpy.ndarray`, `Quantity`'s will contain a `numpy.ndarray` as the attribute `value`. Through the `__array_ufunc__` and `__array_function__` protocols, `Quantity` can interoperate with `numpy`. The advantage of composition over inheritance is that it is possible to use the same `Quantity` class with other array library, not just `NumPy`, by allowing the `value` attribute to be an array from any array library. The challenge is getting the `Quantity` class to work with all the different array libraries. In principle inheritance makes this easier, since by Liskov substitution the

subclass should be able to be used anywhere the superclass is used. However, in practice this is not the case -- either a library like `JAX` doesn't work with subclasses or superclass arrays are made as part of an operation, making the operation incompatible. In the end, inheritance does not solve the problems that are inherent to composition but composition does solve the problems that are inherent to inheritance.

Let's explore this further by considering the following grid showing subclasses of `Quantity` on one axis and array libraries on the other axis.

With inheritance the grid elements are all subclasses of `Quantity`. Imagine defining a new subclass of `Quantity`, with inheritance we would actually need to define a new subclass for each supported array library. There is a way to mitigate this problem, by dynamically creating subclasses of `Quantity` for each array type. In Astropy we do this with `astropy.coordinates.SkyOffsetFrame`. While this is a good solution for `SkyOffsetFrame`, which only operates on Astropy's `BaseCoordinateFrame` objects, it is not a good solution for array libraries for a number of reasons.

1. Unlike `BaseCoordinateFrame`, there is a heterogeneity of array libraries. Defining general rules to dynamically produce subclasses of `Quantity` and an array library is difficult, if not impossible.
2. Dynamically creating subclasses of `Quantity` for each array library is not static typing friendly. The type checker will not know about these dynamically created subclasses and will not be able to check them.
3. Too much magic. Dynamically creating subclasses of `Quantity` for each array library is not introspection friendly. The dynamically created subclasses will not be easily discoverable by users or developers.

Instead we consider the grid with composition. In this case we only need to define a new `Quantity` class for each actual subclass of `Quantity`. Due to composition the different arrays are supported through the `value` attribute, not as the class itself. This effectively "orthogonalizes" the grid, where we can solve each axis independently, an $X + Y$ problem, instead of jointly as $X \cdot Y$. Therefore there are a much smaller number of classes to define. Moreover, composition is more compatible with static typing and deep introspection.

Having established that composition will be a more useful paradigm for the `Quantity` class we proceed to work through the required components to make the new `Quantity`. We emphasize that this is an overview only, not a feature-complete implementation. We start by considering the simplest case, which will need to be complexified, but which solves the majority of Astropy's needs.

Consider the class hierarchy for `Quantity` (following the increasingly popular abstract-final pattern, see <https://docs.kidger.site/equinox/pattern/> for more information)

```
In [37]: from abc import ABCMeta
from typing import Generic, Protocol
from typing_extensions import Self
from mypy_extensions import trait
from array_api import Array as ArrayAPI, ArrayAPINamespace

from units import array_namespace
from units.api import Quantity as QuantityAPI

Array = TypeVar("Array", bound=ArrayAPI)
Array_co = TypeVar("Array_co", bound=ArrayAPI, covariant=True)

@trait
class LegacyQuantityMixin(Protocol[Array_co]):
    """Long term support for non-API methods."""

    def to(self: QuantityAPI[Array_co], unit: Unit) -> Self:
        return self.to_unit(unit)

    def to_value(self: QuantityAPI[Array_co], unit: Unit) -> Array_co:
        return self.to_unit_value(unit)

@dataclass(frozen=True)
class AbstractQuantity(
    Generic[Array], LegacyQuantityMixin[Array], NumPyMixin, metaclass=ABCMeta
):
    value: Array
    unit: Unit

    def __getattr__(self, name: str) -> Any:
        """Map to the wrapped value"""
        return getattr(self.value, name)

# =====
# Quantity API

    def to_unit(self, unit: Unit) -> "AbstractQuantity[Array]":
        """Convert to a new unit."""
        # TODO: self.value * self.unit.to(unit) doesn't work for temperature
        # This is for illustration purposes only.
        return replace(self, value=self.value * self.unit.to(unit), unit=unit)

    def to_unit_value(self, unit: Unit) -> Array:
        return self.to_unit(unit).value

# =====
# Array-API
```

```

    def __array_namespace__(self, api_version: str | None = None) -> ArrayAPI:
        return array_namespace

    # --- Arithmetic ---

    def __add__(self, other: "AbstractQuantity[Array]") -> "AbstractQuantity[Array]":
        ...

@dataclass(frozen=True)
class Quantity(AbstractQuantity[Array]):
    value: Array # type: ignore[assignment]
    unit: Unit # type: ignore[assignment]

```

The `LegacyQuantityMixin` allows for indefinite backwards compatibility between `Quantity` 1.0 and `Quantity` 2.0 and easy deprecation if / when we choose. The `NumPyMixin`, introduced previously, connects the Array-API specification of `Quantity` 2.0 to `NumPy`'s overriding mechanisms. `AbstractQuantity` is the base class of the entire `Quantity` 2.0 hierarchy, bringing together the legacy API, the `NumPy` connection, the first-class Array-API support, and of course the new API for `Quantity`. `Quantity` is a concrete class building off `AbstractQuantity`.

```

In [38]: x = da.from_array(np.array([1, 2, 3]))
         q = u.Quantity(x, apyu.deg)

```

```

In [39]: # LegacyQuantityMixin
         q.to(apyu.deg).unit

```

```

Out[39]: Unit("deg")

```

```

In [40]: # Quantity API
         repr(q.to_unit(apyu.deg))

```

```

Out[40]: 'Quantity(value=dask.array<mul, shape=(3,), dtype=float64, chunksize=(3,),
             chunktype=numpy.ndarray>, unit=Unit("deg"))'

```

```

In [41]: # NumPyMixin
         repr(np.cos(q))

```

```

Out[41]: 'Quantity(value=dask.array<cos, shape=(3,), dtype=float64, chunksize=(3,),
             chunktype=numpy.ndarray>, unit=Unit(dimensionless))'

```

```

In [42]: # Array API
         xp = q.__array_namespace__()
         repr(xp.cos(q))

```

```

Out[42]: 'Quantity(value=dask.array<cos, shape=(3,), dtype=float64, chunksize=(3,),
             chunktype=numpy.ndarray>, unit=Unit(dimensionless))'

```

```

In [43]: # Array API universal library

```



```
repr(universal.cos(q))
```

```
Out[43]: 'Quantity(value=dask.array<cos, shape=(3,), dtype=float64, chunksize=(3,),
chunktype=numpy.ndarray>, unit=Unit(dimensionless))'
```

`Angle`, `Distance` and other `Quantity` classes will subclass `AbstractQuantity` in a similar abstract-final pattern.

1. `AbstractQuantity` -> `AbstractPhysicalTypeQuantity` -> `Distance` (perhaps with an `AbstractDistance` if we want to make a `Redshift` class)
2. `AbstractQuantity` -> `AbstractPhysicalTypeQuantity` -> `AbstractAngle` -> `Angle` / `Longitude` / `Latitude`

Looking at `AbstractQuantity` and the above examples we see that `AbstractQuantity` supports all operations within the Array API. If this is the limit to what Astropy wants to support then `Quantity` can be this very simple structure. However, the Array class in each array library has some distinct methods and attributes. As a brief example:

1. Dask has `to_dask_dataframe`
2. JAX has `.at[]`
3. Zarr has `get_orthogonal_selection`.

As currently described `Quantity` does not add unit support to these methods and attributes, passing data directly from the underlying `value` Array.

```
In [44]: # Building a `Quantity` from the above code
Quantity(q.value, q.unit).to_dask_dataframe() # does not support units!
```

```
Out[44]: Dask Series Structure:
npartitions=1
0      int64
2      ...
dtype: int64
Dask Name: from-dask-array, 2 graph layers
```

The `nstarman/units` reference library we have been referring to does support units on this method:

```
In [45]: q.to_dask_dataframe()
```

```
Out[45]: Quantity(value=Dask Series Structure:
npartitions=1
0      int64
2      ...
dtype: int64
Dask Name: from-dask-array, 2 graph layers, unit=Unit("deg"))
```

How? `Quantity` is not a subclass, e.g. `DaskQuantity`, that adds support for `dask`

operations. Instead `Quantity` implements an interface layer between `Quantity` and `value` that adds support for units.

Approximately this is implemented as:

```
In [46]: class Quantity:

    def __init__(self, value, unit):
        self.unit = unit
        self.interface = get_interface(value)

    @property
    def value(self):
        return self.interface.value

    def __getattr__(self, name):
        return getattr(self.interface, name)
```

The `get_interface` function takes the `value` and returns an `AbstractQuantityInterface` object. It is this interface object that supports units for `.to_dask_dataframe()`. In fact the `Interface` objects perform all the computations on the value, with support for the `unit`. The `Quantity` class is a very thin wrapper around the `Interface`. Subclasses of `Quantity`, e.g. `Angle`, are less-thin wrappers around the `Interface`, by including additional computation on, e.g. a `wrap_angle`.

```
In [47]: class Quantity:
    ...

    def __add__(self, other): # thin wrapper
        return self.interface + other

    ...
```

This separation of `Quantity` (and subclasses) and the `Interface` demonstrates how composition can simplify the `Quantity - Array` grid discussed earlier, making the axes independent. When constructing a `Quantity` a user will always get back a `Quantity`. The API is simple, familiar, intuitive, and not magical -- `Quantity` in, `Quantity` out. The Array library object is accessible as `.value`. If a user needs to dig deeper they can access the `interface` object that provides the backend support for the Array library.

```
In [48]: type(q.value)
```

```
Out[48]: dask.array.core.Array
```


```
In [49]: type(q.interface)
```

```
Out[49]: units._quantity.interface.builtin.dask_interface.LegacyDaskArrayInterface
```

```
In [50]: q.interface
```

```
Out[50]:
```

	Array	Chunk
Bytes	24 B	24 B
Shape	(3,)	(3,)
Dask graph	1 chunks in 1 graph layer	
Data type	int64 numpy.ndarray	



The details of how the `Interface` object works and is implemented for `dataclass` classes can be found in the reference implementation library `nstarman/units`.

Recommendation: Using entry-points make a ``quantity_interfaces`` (or some other name) library that registers ``Interface``'s for the most common array libraries.

```
In [ ]:
```

Jax Arrays

In the previous sections we have discussed how the Array-API is changing the paradigm of interoperability from `NumPy`'s NEP 13,18 - style overrides to using the array namespace held on Array objects. With the Array-API `Quantity` will be able to interoperate with any other supporting Array library. This brings us to the issue of `JAX`, which does not support the Array-API and which also does not allow for overrides using either composition or inheritance. `JAX` works with `JAX` arrays. This is not quite true. `JAX` actually has a very powerful system of performing operations on complex Python objects through JAX's `PyTree` framework (<https://jax.readthedocs.io/en/latest/pytrees.html>).

The popular library `Equinox` builds off the `PyTree` framework to auto-enable Python dataclasses to work with `jax` operations that are mapped over the PyTree. The JAX - Python dataclasses integration is the foundation upon which `Equinox` has built an entire ML library, and through `Equinox` on which a growing ecosystem of libraries are being built (lineax, diffrax, etc). One library, `quax`, allows jax array-like classes that are `Equinox` PyTrees to register overrides into JAX's system of primitives. Then by wrapping a function with `quax`'s custom JAX tracer, the wrapped jax functions can

operate on the array-like objects. This sounds complicated, and it is in detail, but the takeaway is that it is relatively simple to create an Array-API conformant library of `quax`-wrapped `jax` functions (`array_api_jax_compat`) that will work on `Quantity` objects!

```
In [51]: import astropy.units as u
import jax
import jax.numpy as jnp

# isort: split
from jax_quantity import Quantity

import array_api_jax_compat as xp

jax.config.update("jax_enable_x64", True)
```

```
In [52]: # Jax Array
x = jnp.array([1, 2, 3], dtype=jnp.float64)
x
```

```
Out[52]: Array([1., 2., 3.], dtype=float64)
```

```
In [53]: # As a Quantity
q = Quantity(x, unit="m")
q
```

```
Out[53]: Quantity(value=f64[3], unit=Unit("m"))
```

```
In [54]: # A universal library for JAX Array-API objects
xp.pow(q, 2)
```

```
Out[54]: Quantity(value=f64[3], unit=Unit("m2"))
```

```
In [55]: # The universal library for any Array-API objects
universal.pow(q, 2)
```

```
Out[55]: Quantity(value=f64[3], unit=Unit("m2"))
```

```
In [56]: # demonstrating jit works!

@jax.jit
def func(x1, x2):
    return xp.divide(xp.pow(x1, 3), x2)
```

```
In [57]: func(q, q).value, func(q, q).unit
```

```
Out[57]: (Array([1., 4., 9.], dtype=float64), Unit("m2"))
```

Two reference libraries were used in this demonstration.

1. `jax_quantity` for a `Quantity` class.
2. `array_api_jax_compat` as the Array-API library for Jax Quantities. Due to the implementation of `jax` and `quax` this library can be universally used for JAX array-like objects.

The only reason the `jax_quantity.Quantity` class exists rather than being an `Interface` object in `nstarman/units` is that `quax` currently only supports `equinox` classes, which `units.Quantity` is not (but `jax_quantity.Quantity` is). `quax` is built on top of `equinox` as part of a vertical chain of integration with JAX. This vertical integration is due to the historical development of both libraries by Patrick Kidger. `quax` relies on `equinox` primarily for the PyTree integration. It is very possible to decouple `quax` from `equinox` and make it applicable to any object that can be transformed to a JAX PyTree, which we can do with `Quantity`, without adding a runtime dependency on JAX

Recommendation: Working with ``quax`` to enable more general PyTree support, independent of ``equinox``. This will allow ``jax_quantity`` to change from vendoring an example ``Quantity`` class to instead a JAX-Quantity interface object.

Roadmap

In the previous sections we outlined the future of the `Quantity` class. From the Array API to a Quantity API, from a composition framework to interface objects, the new `Quantity` will have a clear API, be statically typed and introspectable, and easily extensible to support any Array library. In this short section we will roadmap how Astropy can create and transition to this new `Quantity` class.

1) Develop the Quantity API

The whole process begins with a meeting of stakeholders. There are numerous other Python libraries that enable unit-ful calculations: in particular `pint`, `unyt`, `sympy`. We must create a working group that will create a common Quantity API. Hopefully this working group will continue to collaborate on a Units API and to develop other shared resources, like unit parsers.

The first draft of the Quantity API must establish:

1. The attribute holding the Quantity's value (`value` in Astropy, `magnitude` in `pint`). The author does not have strong opinions on this point: `value` is more intuitive, but more likely to be an attribute on an array object `Quantity` is trying to wrap. `magnitude` has the opposite situation.

2. The attribute holding the Quantity's units (`unit` in Astropy, `unit` and `units` in `pint` , depending on what you need). The author prefers `units` as this matches speech patterns, but has no strong opinions.
3. The method for converting a Quantity from one units to another. The author strongly advocates for `to_unit(s)` (whichever is chosen for point 2). Astropy currently support the more general `to` , but this is significantly more likely to override methods on an array object `Quantity` is trying to wrap.
4. The method for converting a Quantity from one units to another and taking the value / magnitude. The author strongly advocates for `to_unit(s)_value(/magnitude)` .

2) Implement the API on the existing Quantity

The Quantity API should be implemented on Astropy's current Quantity class. The old methods, e.g. `to()` and `to_value()` should NOT be deprecated, but should be changed to just point to their new implementations. In a previous section the `Quantity 2.0` class implemented support for the old API via a `LegacyQuantityMixin` . The same approach can be taken here.

It is important to go over the entire Astropy code base and ensure that the new API is being used instead of the old. In particular, we must get ready for the change from inheritance to composition by not relying on `numpy` -specific functionality like zero-copy `view` operations. Other libraries do not support this and since the goal is for Quantity to work with other Array libraries, then functionality built on Quantity must adapt.

3) Make a `astropy.units.array_api` module

Just as `numpy` has a `numpy.array_api` module for the development of `numpy` 's support for the Array API, `Astropy` should make an `astropy.units.array_api` module that will house the development of `Quantity 2.0` and associated Array-API namespace. This will be clearly labelled in the documentation as being an *experimental* and *unstable* module while it is under development.

The development of `astropy.units.array_api` will hopefully consist mostly of porting over (and improving) selected contents from <https://github.com/nstarman/units/>.

5) Develop interface extension libraries for other array libraries

We don't want to add unnecessary run-time dependencies to Astropy, and there's not

much point cluttering up the repo with files of code gated by import checks. Python's extension point mechanism will allow us to develop interface extensions, like the one shown for `dask`, and register them into `Quantity` 2.0. Having a separate library is also a good way to invite maintainers of the Array libraries to help manage, if they are interested, the `Quantity` interface to their library.

6) Develop a `jaxtyping`-like layer for units and dimension checks

This will enable static (through a mypy extension or through JIT like in `jax`) and runtime checking of unit propagation. The author recommends that the `Quantity` API working group also lead this effort, since it is desirable that the unit annotations, like the value annotations, be the same across libraries.

Conclusion

In this document we outlined and considered the design of a new `Quantity` class for `Astropy`. In contrast to the current limitation to only `NumPy` arrays, this new class will be able to be used with any array library, including `NumPy`, `JAX`, `CuPy`, and others. The new `Quantity` class architecture is also designed for the future: compatible with emerging Python data standards, built for easy interoperability and extensibility, and designed for static typing and associated compiled speedups.

We provide three reference libraries that demonstrate how to build the `Quantity` 2.0 class and support `Dask` arrays, to build an Array-API function library that works with `JAX` arrays in a `Quantity`, and to build a `JAX`-supporting `Quantity`.

Reference Libraries:

1. <https://github.com/nstarman/units>
2. <https://github.com/GalacticDynamics/array-api-jax-compat>
3. <https://github.com/GalacticDynamics/jax-quantity>

In []:

In []: