Subclassing ndarray

Introduction

Subclassing ndarray is relatively simple, but it has some complications compared to other Python objects. On this page we explain the machinery that allows you to subclass ndarray, and the implications for implementing a subclass.

ndarrays and object creation

Subclassing ndarray is complicated by the fact that new instances of ndarray classes can come about in three different ways. These are:

1. Explicit constructor call - as in `MySubClass(params)`. This is the usual route to Python instance creation.
2. View casting - casting an existing ndarray as a given subclass
3. New from template - creating a new instance from a template instance. Examples include returning slices from a subclassed array, creating return types from ufuncs, and copying arrays. See Creating new from template for more details

The last two are characteristics of ndarrays - in order to support things like array slicing. The complications of subclassing ndarray are due to the mechanisms numpy has to support these latter two routes of instance creation.

When to use subclassing

Besides the additional complexities of subclassing a NumPy array, subclasses can run into unexpected behaviour because some functions may convert the subclass to a baseclass and "forget" any additional information associated with the subclass. This can result in surprising behavior if you use NumPy methods or functions you have not explicitly tested.

On the other hand, compared to other interoperability approaches, subclassing can be a useful because many thing will "just work".

This means that subclassing can be a convenient approach and for a long time it was also often the only available approach. However, NumPy now provides additional interoperability protocols described in "Interoperability with NumPy". For many use-cases these interoperability protocols may now be a better fit or supplement the use of subclassing.

Subclassing can be a good fit if:

- you are less worried about maintainability or users other than yourself: Subclass will be faster to implement and additional interoperability can be added "as-needed". And with few users, possible surprises are not an issue.
- you do not think it is problematic if the subclass information is ignored or lost silently. An example is `np.memmap` where "forgetting" about data being memory mapped cannot lead to a wrong result. An example of a subclass that sometimes confuses users are NumPy's masked arrays. When they were introduced, subclassing was the only approach for implementation. However, today we would possibly try to avoid subclassing and rely only on interoperability protocols.

Note that also subclass authors may wish to study Interoperability with NumPy to support more complex use-cases or work around the surprising behavior.

Search the docs ...

What is NumPy?
Installation
NumPy quickstart
NumPy: the absolute basics for beginners
NumPy fundamentals
- Array creation
- Indexing on `ndarrays`
- I/O with NumPy
- Data types
- Broadcasting
- Byte-swapping
- Structured arrays
- Writing custom array containers
- Subclassing ndarray
- Universal functions (`ufunc`) basics
- Copies and views
- Interoperability with NumPy

Miscellaneous
NumPy for MATLAB users
Building from source
Using NumPy C-API
NumPy Tutorials
NumPy How Tos
For downstream package authors

F2PY user guide and reference manual

Glossary

Under-the-hood Documentation for developers

Reporting bugs

Release notes

NumPy license

astropy.units.Quantity and xarray are examples for array-like objects that interoperate well with NumPy. Astropy's Quantity is an example which uses a dual approach of both subclassing and interoperability protocols.

View casting

View casting is the standard ndarray mechanism by which you take an ndarray of any subclass, and return a view of the array as another (specified) subclass:

```
>>> import numpy as np
>>> # create a completely useless ndarray subclass
>>> class C(np.ndarray): pass
>>> # create a standard ndarray
>>> arr = np.zeros((3,))
>>> # take a view of it, as our useless subclass
>>> c_arr = arr.view(C)
>>> type(c_arr)
<class '__main__.C'>
```

Creating new from template

New instances of an ndarray subclass can also come about by a very similar mechanism to View casting, when numpy finds it needs to create a new instance from a template instance. The most obvious place this has to happen is when you are taking slices of subclassed arrays. For example:

```
>>> v = c_arr[1:]
>>> type(v) # the view is of type 'C'
<class '__main__.C'>
>>> v is c_arr # but it's a new instance
False
```

The slice is a *view* onto the original c_arr data. So, when we take a view from the ndarray, we return a new ndarray, of the same class, that points to the data in the original.

There are other points in the use of ndarrays where we need such views, such as copying arrays (c_arr.copy()), creating ufunc output arrays (see also __array_wrap__ for ufuncs and other functions), and reducing methods (like c_arr.mean()).

Relationship of view casting and new-from-template

These paths both use the same machinery. We make the distinction here, because they result in different input to your methods. Specifically, View casting means you have created a new instance of your array type from any potential subclass of ndarray. Creating new from template means you have created a new instance of your class from a pre-existing instance, allowing you - for example - to copy across attributes that are particular to your subclass.

Implications for subclassing

If we subclass ndarray, we need to deal not only with explicit construction of our array type, but also View casting or Creating new from template. NumPy has the machinery to do this, and it is this machinery that makes subclassing slightly non-standard.

There are two aspects to the machinery that ndarray uses to support views and new-from-template in subclasses.

The first is the use of the ndarray.__new__ method for the main work of object initialization, rather then the more usual __init__ method. The second is the use of the __array_finalize__ method to allow subclasses to clean up after the creation of views and new instances from templates.

A brief Python primer on `__new__` and `__init__`

`__new__` is a standard Python method, and, if present, is called before `__init__` when we create a class instance. See the python __new__ documentation for more detail.

For example, consider the following Python code:

```
>>> class C:
>>>     def __new__(cls, *args):
>>>         print('Cls in __new__:', cls)
>>>         print('Args in __new__:', args)
>>>         # The `object` type __new__ method takes a single argument.
>>>         return object.__new__(cls)
>>>     def __init__(self, *args):
>>>         print('type(self) in __init__:', type(self))
>>>         print('Args in __init__:', args)
```

meaning that we get:

```
>>> c = C('hello')
Cls in __new__: <class 'C'>
Args in __new__: ('hello',)
type(self) in __init__: <class 'C'>
Args in __init__: ('hello',)
```

When we call `C('hello')`, the `__new__` method gets its own class as first argument, and the passed argument, which is the string `'hello'`. After python calls `__new__`, it usually (see below) calls our `__init__` method, with the output of `__new__` as the first argument (now a class instance), and the passed arguments following.

As you can see, the object can be initialized in the `__new__` method or the `__init__` method, or both, and in fact ndarray does not have an `__init__` method, because all the initialization is done in the `__new__` method.

Why use `__new__` rather than just the usual `__init__`? Because in some cases, as for ndarray, we want to be able to return an object of some other class. Consider the following:

```
class D(C):
    def __new__(cls, *args):
        print('D cls is:', cls)
        print('D args in __new__:', args)
        return C.__new__(C, *args)

    def __init__(self, *args):
        # we never get here
        print('In D __init__')
```

meaning that:

```
>>> obj = D('hello')
D cls is: <class 'D'>
D args in __new__: ('hello',)
Cls in __new__: <class 'C'>
Args in __new__: ('hello',)
>>> type(obj)
<class 'C'>
```

The definition of `C` is the same as before, but for `D`, the `__new__` method returns an instance of class `C` rather than `D`. Note that the `__init__` method of `D` does not get called. In general, when the `__new__` method returns an object of class other than the class in which it is defined, the `__init__` method of that class is not called.

This is how subclasses of the ndarray class are able to return views that preserve the class type. When taking a view, the standard ndarray machinery creates the new ndarray object with something like:

```
obj = ndarray.__new__(subtype, shape, ...
```

where `subdtype` is the subclass. Thus the returned view is of the same class as the subclass, rather than being of class `ndarray`.

That solves the problem of returning views of the same type, but now we have a new problem. The machinery of ndarray can set the class this way, in its standard methods for taking views, but the ndarray `__new__` method knows nothing of what we have done in our own `__new__` method in order to set attributes, and so on. (Aside - why not call `obj = subdtype.__new__(...` then? Because we may not have a `__new__` method with the same call signature).

The role of `__array_finalize__`

`__array_finalize__` is the mechanism that numpy provides to allow subclasses to handle the various ways that new instances get created.

Remember that subclass instances can come about in these three ways:

1. explicit constructor call (`obj = MySubClass(params)`). This will call the usual sequence of `MySubClass.__new__` then (if it exists) `MySubClass.__init__`.
2. View casting
3. Creating new from template

Our `MySubClass.__new__` method only gets called in the case of the explicit constructor call, so we can't rely on `MySubClass.__new__` or `MySubClass.__init__` to deal with the view casting and new-from-template. It turns out that `MySubClass.__array_finalize__` *does* get called for all three methods of object creation, so this is where our object creation housekeeping usually goes.

- For the explicit constructor call, our subclass will need to create a new ndarray instance of its own class. In practice this means that we, the authors of the code, will need to make a call to `ndarray.__new__(MySubClass,...)`, a class-hierarchy prepared call to `super().__new__(cls, ...)`, or do view casting of an existing array (see below)
- For view casting and new-from-template, the equivalent of `ndarray.__new__(MySubClass,...` is called, at the C level.

The arguments that `__array_finalize__` receives differ for the three methods of instance creation above.

The following code allows us to look at the call sequences and arguments:

```
import numpy as np

class C(np.ndarray):
    def __new__(cls, *args, **kwargs):
        print('In __new__ with class %s' % cls)
        return super().__new__(cls, *args, **kwargs)

    def __init__(self, *args, **kwargs):
        # in practice you probably will not need or want an __init__
        # method for your subclass
        print('In __init__ with class %s' % self.__class__)

    def __array_finalize__(self, obj):
        print('In array_finalize:')
        print('   self type is %s' % type(self))
        print('   obj type is %s' % type(obj))
```

Now:

```
>>> # Explicit constructor
>>> c = C((10,))
In __new__ with class <class 'C'>
In array_finalize:
   self type is <class 'C'>
   obj type is <type 'NoneType'>
In __init__ with class <class 'C'>
>>> # View casting
>>> a = np.arange(10)
>>> cast_a = a.view(C)
In array_finalize:
   self type is <class 'C'>
   obj type is <type 'numpy.ndarray'>
>>> # Slicing (example of new-from-template)
>>> cv = c[:1]
In array_finalize:
   self type is <class 'C'>
   obj type is <class 'C'>
```

The signature of `__array_finalize__` is:

```
def __array_finalize__(self, obj):
```

One sees that the `super` call, which goes to `ndarray.__new__`, passes `__array_finalize__` the new object, of our own class (`self`) as well as the object from which the view has been taken (`obj`). As you can see from the output above, the `self` is always a newly created instance of our subclass, and the type of `obj` differs for the three instance creation methods:

- When called from the explicit constructor, `obj` is `None`
- When called from view casting, `obj` can be an instance of any subclass of ndarray, including our own.
- When called in new-from-template, `obj` is another instance of our own subclass, that we might use to update the new `self` instance.

Because `__array_finalize__` is the only method that always sees new instances being created, it is the sensible place to fill in instance defaults for new object attributes, among other tasks.

This may be clearer with an example.

Simple example - adding an extra attribute to ndarray

```python
import numpy as np

class InfoArray(np.ndarray):

    def __new__(subtype, shape, dtype=float, buffer=None, offset=0,
                strides=None, order=None, info=None):
        # Create the ndarray instance of our type, given the usual
        # ndarray input arguments.  This will call the standard
        # ndarray constructor, but return an object of our type.
        # It also triggers a call to InfoArray.__array_finalize__
        obj = super().__new__(subtype, shape, dtype,
                              buffer, offset, strides, order)
        # set the new 'info' attribute to the value passed
        obj.info = info
        # Finally, we must return the newly created object:
        return obj

    def __array_finalize__(self, obj):
        # ``self`` is a new object resulting from
        # ndarray.__new__(InfoArray, ...), therefore it only has
        # attributes that the ndarray.__new__ constructor gave it -
        # i.e. those of a standard ndarray.
        #
        # We could have got to the ndarray.__new__ call in 3 ways:
        # From an explicit constructor - e.g. InfoArray():
        #    obj is None
        #    (we're in the middle of the InfoArray.__new__
        #    constructor, and self.info will be set when we return to
        #    InfoArray.__new__)
        if obj is None: return
        # From view casting - e.g arr.view(InfoArray):
        #    obj is arr
        #    (type(obj) can be InfoArray)
        # From new-from-template - e.g infoarr[:3]
        #    type(obj) is InfoArray
        #
        # Note that it is here, rather than in the __new__ method,
        # that we set the default value for 'info', because this
        # method sees all creation of default objects - with the
        # InfoArray.__new__ constructor, but also with
        # arr.view(InfoArray).
        self.info = getattr(obj, 'info', None)
        # We do not need to return anything
```

Using the object looks like this:

```
>>> obj = InfoArray(shape=(3,)) # explicit constructor
>>> type(obj)
<class 'InfoArray'>
>>> obj.info is None
True
>>> obj = InfoArray(shape=(3,), info='information')
>>> obj.info
'information'
>>> v = obj[1:] # new-from-template - here - slicing
>>> type(v)
<class 'InfoArray'>
>>> v.info
'information'
>>> arr = np.arange(10)
>>> cast_arr = arr.view(InfoArray) # view casting
>>> type(cast_arr)
<class 'InfoArray'>
>>> cast_arr.info is None
True
```

This class isn't very useful, because it has the same constructor as the bare ndarray object, including passing in buffers and shapes and so on. We would probably prefer the constructor to be able to take an already formed ndarray from the usual numpy calls to `np.array` and return an object.

Slightly more realistic example - attribute added to existing array

Here is a class that takes a standard ndarray that already exists, casts as our type, and adds an extra attribute.

```
import numpy as np

class RealisticInfoArray(np.ndarray):

    def __new__(cls, input_array, info=None):
        # Input array is an already formed ndarray instance
        # We first cast to be our class type
        obj = np.asarray(input_array).view(cls)
        # add the new attribute to the created instance
        obj.info = info
        # Finally, we must return the newly created object:
        return obj

    def __array_finalize__(self, obj):
        # see InfoArray.__array_finalize__ for comments
        if obj is None: return
        self.info = getattr(obj, 'info', None)
```

So:

```
>>> arr = np.arange(5)
>>> obj = RealisticInfoArray(arr, info='information')
>>> type(obj)
<class 'RealisticInfoArray'>
>>> obj.info
'information'
>>> v = obj[1:]
>>> type(v)
<class 'RealisticInfoArray'>
>>> v.info
'information'
```

`__array_ufunc__` for ufuncs

New in version 1.13.

A subclass can override what happens when executing numpy ufuncs on it by overriding the default `ndarray.__array_ufunc__` method. This method is executed *instead* of the ufunc and should return either the result of the operation, or **`NotImplemented`** if the operation requested is not implemented.

The signature of `__array_ufunc__` is:

```
def __array_ufunc__(ufunc, method, *inputs, **kwargs):

- *ufunc* is the ufunc object that was called.
- *method* is a string indicating how the Ufunc was called, either
  ``"__call__"`` to indicate it was called directly, or one of its
  :ref:`methods<ufuncs.methods>`: ``"reduce"``, ``"accumulate"``,
  ``"reduceat"``, ``"outer"``, or ``"at"``.
- *inputs* is a tuple of the input arguments to the ``ufunc``
- *kwargs* contains any optional or keyword arguments passed to the
  function. This includes any ``out`` arguments, which are always
  contained in a tuple.
```

A typical implementation would convert any inputs or outputs that are instances of one's own class, pass everything on to a superclass using `super()`, and finally return the results after possible back-conversion. An example, taken from the test case `test_ufunc_override_with_super` in `core/tests/test_umath.py`, is the following.

```
input numpy as np

class A(np.ndarray):
    def __array_ufunc__(self, ufunc, method, *inputs, out=None, **kwargs):
        args = []
        in_no = []
        for i, input_ in enumerate(inputs):
            if isinstance(input_, A):
                in_no.append(i)
                args.append(input_.view(np.ndarray))
            else:
                args.append(input_)

        outputs = out
        out_no = []
        if outputs:
            out_args = []
            for j, output in enumerate(outputs):
                if isinstance(output, A):
                    out_no.append(j)
                    out_args.append(output.view(np.ndarray))
                else:
                    out_args.append(output)
            kwargs['out'] = tuple(out_args)
        else:
            outputs = (None,) * ufunc.nout

        info = {}
        if in_no:
            info['inputs'] = in_no
        if out_no:
            info['outputs'] = out_no

        results = super().__array_ufunc__(ufunc, method, *args, **kwargs)
        if results is NotImplemented:
            return NotImplemented

        if method == 'at':
            if isinstance(inputs[0], A):
                inputs[0].info = info
            return

        if ufunc.nout == 1:
            results = (results,)

        results = tuple((np.asarray(result).view(A)
                         if output is None else output)
                        for result, output in zip(results, outputs))
        if results and isinstance(results[0], A):
            results[0].info = info

        return results[0] if len(results) == 1 else results
```

So, this class does not actually do anything interesting: it just converts any instances of its own to regular ndarray (otherwise, we'd get infinite recursion!), and adds an `info` dictionary that tells which inputs and outputs it converted. Hence, e.g.,

```
>>> a = np.arange(5.).view(A)
>>> b = np.sin(a)
>>> b.info
{'inputs': [0]}
>>> b = np.sin(np.arange(5.), out=(a,))
>>> b.info
{'outputs': [0]}
>>> a = np.arange(5.).view(A)
>>> b = np.ones(1).view(A)
>>> c = a + b
>>> c.info
{'inputs': [0, 1]}
>>> a += b
>>> a.info
{'inputs': [0, 1], 'outputs': [0]}
```

Note that another approach would be to use `getattr(ufunc, methods)(*inputs, **kwargs)` instead of the `super` call. For this example, the result would be identical, but there is a difference if another operand also defines `__array_ufunc__`. E.g., lets assume that we evaluate `np.add(a, b)`, where `b` is an instance of another class `B` that has an override. If you use `super` as in the example, `ndarray.__array_ufunc__` will notice that `b` has an override, which means it cannot evaluate the result itself. Thus, it will return *NotImplemented* and so will our class `A`. Then, control will be passed over to `b`, which either knows how to deal with us and produces a result, or does not and returns *NotImplemented*, raising a `TypeError`.

If instead, we replace our `super` call with `getattr(ufunc, method)`, we effectively do `np.add(a.view(np.ndarray), b)`. Again, `B.__array_ufunc__` will be called, but now it sees an `ndarray` as the other argument. Likely, it will know how to handle this, and return a new instance of the `B` class to us. Our example class is not set up to handle this, but it might well be the best approach if, e.g., one were to re-implement `MaskedArray` using `__array_ufunc__`.

As a final note: if the `super` route is suited to a given class, an advantage of using it is that it helps in constructing class hierarchies. E.g., suppose that our other class `B` also used the `super` in its `__array_ufunc__` implementation, and we created a class `C` that depended on both, i.e., `class C(A, B)` (with, for simplicity, not another `__array_ufunc__` override). Then any ufunc on an instance of `C` would pass on to `A.__array_ufunc__`, the `super` call in `A` would go to `B.__array_ufunc__`, and the `super` call in `B` would go to `ndarray.__array_ufunc__`, thus allowing `A` and `B` to collaborate.

`__array_wrap__` for ufuncs and other functions

Prior to numpy 1.13, the behaviour of ufuncs could only be tuned using `__array_wrap__` and `__array_prepare__`. These two allowed one to change the output type of a ufunc, but, in contrast to `__array_ufunc__`, did not allow one to make any changes to the inputs. It is hoped to eventually deprecate these, but `__array_wrap__` is also used by other numpy functions and methods, such as `squeeze`, so at the present time is still needed for full functionality.

Conceptually, `__array_wrap__` "wraps up the action" in the sense of allowing a subclass to set the type of the return value and update attributes and metadata. Let's show how this works with an example. First we return to the simpler example subclass, but with a different name and some print statements:

```
import numpy as np

class MySubClass(np.ndarray):

    def __new__(cls, input_array, info=None):
        obj = np.asarray(input_array).view(cls)
        obj.info = info
        return obj

    def __array_finalize__(self, obj):
        print('In __array_finalize__:')
        print('   self is %s' % repr(self))
        print('   obj is %s' % repr(obj))
        if obj is None: return
        self.info = getattr(obj, 'info', None)

    def __array_wrap__(self, out_arr, context=None):
        print('In __array_wrap__:')
        print('   self is %s' % repr(self))
        print('   arr is %s' % repr(out_arr))
        # then just call the parent
        return super().__array_wrap__(self, out_arr, context)
```

We run a ufunc on an instance of our new array:

```
>>> obj = MySubClass(np.arange(5), info='spam')
In __array_finalize__:
   self is MySubClass([0, 1, 2, 3, 4])
   obj is array([0, 1, 2, 3, 4])
>>> arr2 = np.arange(5)+1
>>> ret = np.add(arr2, obj)
In __array_wrap__:
   self is MySubClass([0, 1, 2, 3, 4])
   arr is array([1, 3, 5, 7, 9])
In __array_finalize__:
   self is MySubClass([1, 3, 5, 7, 9])
   obj is MySubClass([0, 1, 2, 3, 4])
>>> ret
MySubClass([1, 3, 5, 7, 9])
>>> ret.info
'spam'
```

Note that the ufunc (`np.add`) has called the `__array_wrap__` method with arguments `self` as `obj`, and `out_arr` as the (ndarray) result of the addition. In turn, the default `__array_wrap__` (`ndarray.__array_wrap__`) has cast the result to class `MySubClass`, and called `__array_finalize__` - hence the copying of the `info` attribute. This has all happened at the C level.

But, we could do anything we wanted:

```
class SillySubClass(np.ndarray):

    def __array_wrap__(self, arr, context=None):
        return 'I lost your data'
```

```
>>> arr1 = np.arange(5)
>>> obj = arr1.view(SillySubClass)
>>> arr2 = np.arange(5)
>>> ret = np.multiply(obj, arr2)
>>> ret
'I lost your data'
```

So, by defining a specific `__array_wrap__` method for our subclass, we can tweak the output from ufuncs. The `__array_wrap__` method requires `self`, then an argument - which is the result of the ufunc - and an optional parameter *context*. This parameter is returned by ufuncs as a 3-element tuple: (name of the ufunc, arguments of the ufunc, domain of the ufunc), but is not set by other numpy functions. Though, as seen above, it is possible to do otherwise, `__array_wrap__` should return an instance of its containing class. See the masked array subclass for an implementation.

In addition to `__array_wrap__`, which is called on the way out of the ufunc, there is also an `__array_prepare__` method which is called on the way into the ufunc, after the output arrays are created but before any computation has been performed. The default implementation does nothing but pass through the array. `__array_prepare__` should not attempt to access the array data or resize the array, it is intended for setting the output array type, updating attributes and metadata, and performing any checks based on the input that may be desired before computation begins. Like `__array_wrap__`, `__array_prepare__` must return an ndarray or subclass thereof or raise an error.

Extra gotchas - custom `__del__` methods and ndarray.base

One of the problems that ndarray solves is keeping track of memory ownership of ndarrays and their views. Consider the case where we have created an ndarray, `arr` and have taken a slice with `v = arr[1:]`. The two objects are looking at the same memory. NumPy keeps track of where the data came from for a particular array or view, with the `base` attribute:

```
>>> # A normal ndarray, that owns its own data
>>> arr = np.zeros((4,))
>>> # In this case, base is None
>>> arr.base is None
True
>>> # We take a view
>>> v1 = arr[1:]
>>> # base now points to the array that it derived from
>>> v1.base is arr
True
>>> # Take a view of a view
>>> v2 = v1[1:]
>>> # base points to the original array that it was derived from
>>> v2.base is arr
True
```

In general, if the array owns its own memory, as for `arr` in this case, then `arr.base` will be None - there are some exceptions to this - see the numpy book for more details.

The `base` attribute is useful in being able to tell whether we have a view or the original array. This in turn can be useful if we need to know whether or not to do some specific cleanup when the subclassed array is deleted. For example, we may only want to do the cleanup if the original array is deleted, but not the views. For an example of how this can work, have a look at the `memmap` class in `numpy.core`.

Subclassing and Downstream Compatibility

When sub-classing `ndarray` or creating duck-types that mimic the `ndarray` interface, it is your responsibility to decide how aligned your APIs will be with those of numpy. For convenience, many numpy functions that have a corresponding `ndarray` method (e.g., `sum`, `mean`, `take`, `reshape`) work by checking if the first argument to a function has a method of the same name. If it exists, the method is called instead of coercing the arguments to a numpy array.

For example, if you want your sub-class or duck-type to be compatible with numpy's `sum` function, the method signature for this object's `sum` method should be the following:

```
def sum(self, axis=None, dtype=None, out=None, keepdims=False):
...
```

This is the exact same method signature for `np.sum`, so now if a user calls `np.sum` on this object, numpy will call the object's own `sum` method and pass in these arguments enumerated above in the signature, and no errors will be raised because the signatures are completely compatible with each other.

If, however, you decide to deviate from this signature and do something like this:

```
def sum(self, axis=None, dtype=None):
...
```

This object is no longer compatible with `np.sum` because if you call `np.sum`, it will pass in unexpected arguments `out` and `keepdims`, causing a TypeError to be raised.

If you wish to maintain compatibility with numpy and its subsequent versions (which might add new keyword arguments) but do not want to surface all of numpy's arguments, your function's signature should accept `**kwargs`. For example:

```
def sum(self, axis=None, dtype=None, **unused_kwargs):
...
```

This object is now compatible with `np.sum` again because any extraneous arguments (i.e. keywords that are not `axis` or `dtype`) will be hidden away in the `**unused_kwargs` parameter.

Previous
Writing custom array containers

Next
Universal functions (`ufunc`) basics