Search the docs ...

Writing custom array containers

Numpy's dispatch mechanism, introduced in numpy version v1.16 is the recommended approach for writing custom N-dimensional array containers that are compatible with the numpy API and provide custom implementations of numpy functionality. Applications include dask arrays, an N-dimensional array distributed across multiple nodes, and cupy arrays, an N-dimensional array on a GPU.

To get a feel for writing custom array containers, we'll begin with a simple example that has rather narrow utility but illustrates the concepts involved.

```
>>> import numpy as np
>>> class DiagonalArray:
...     def __init__(self, N, value):
...         self._N = N
...         self._i = value
...     def __repr__(self):
...         return f"{self.__class__.__name__}(N={self._N}, value={self._i})"
...     def __array__(self, dtype=None):
...         return self._i * np.eye(self._N, dtype=dtype)
```

Our custom array can be instantiated like:

```
>>> arr = DiagonalArray(5, 1)
>>> arr
DiagonalArray(N=5, value=1)
```

We can convert to a numpy array using **`numpy.array`** or **`numpy.asarray`**, which will call its `__array__` method to obtain a standard `numpy.ndarray`.

```
>>> np.asarray(arr)
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

If we operate on `arr` with a numpy function, numpy will again use the `__array__` interface to convert it to an array and then apply the function in the usual way.

```
>>> np.multiply(arr, 2)
array([[2., 0., 0., 0., 0.],
       [0., 2., 0., 0., 0.],
       [0., 0., 2., 0., 0.],
       [0., 0., 0., 2., 0.],
       [0., 0., 0., 0., 2.]])
```

Notice that the return type is a standard `numpy.ndarray`.

```
>>> type(np.multiply(arr, 2))
<class 'numpy.ndarray'>
```

How can we pass our custom array type through this function? Numpy allows a class to indicate that it would like to handle computations in a custom-defined way through the interfaces `__array_ufunc__` and `__array_function__`. Let's take one at a time, starting with `_array_ufunc__`. This method covers Universal functions (ufunc), a class of functions that includes, for example, **`numpy.multiply`** and **`numpy.sin`**.

The `__array_ufunc__` receives:

- `ufunc`, a function like `numpy.multiply`
- `method`, a string, differentiating between `numpy.multiply(...)` and variants like `numpy.multiply.outer`, `numpy.multiply.accumulate`, and so on. For the common case, `numpy.multiply(...)`, `method == '__call__'`.
- `inputs`, which could be a mixture of different types
- `kwargs`, keyword arguments passed to the function

For this example we will only handle the method `__call__`

```
>>> from numbers import Number
>>> class DiagonalArray:
...     def __init__(self, N, value):
...         self._N = N
...         self._i = value
...     def __repr__(self):
...         return f"{self.__class__.__name__}(N={self._N}, value={self._i})"
...     def __array__(self, dtype=None):
...         return self._i * np.eye(self._N, dtype=dtype)
...     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
...         if method == '__call__':
...             N = None
...             scalars = []
...             for input in inputs:
...                 if isinstance(input, Number):
...                     scalars.append(input)
...                 elif isinstance(input, self.__class__):
...                     scalars.append(input._i)
...                     if N is not None:
...                         if N != self._N:
...                             raise TypeError("inconsistent sizes")
...                     else:
...                         N = self._N
...                 else:
...                     return NotImplemented
...             return self.__class__(N, ufunc(*scalars, **kwargs))
...         else:
...             return NotImplemented
```

Now our custom array type passes through numpy functions.

```
>>> arr = DiagonalArray(5, 1)
>>> np.multiply(arr, 3)
DiagonalArray(N=5, value=3)
>>> np.add(arr, 3)
DiagonalArray(N=5, value=4)
>>> np.sin(arr)
DiagonalArray(N=5, value=0.8414709848078965)
```

At this point `arr + 3` does not work.

```
>>> arr + 3
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'DiagonalArray' and 'int'
```

To support it, we need to define the Python interfaces `__add__`, `__lt__`, and so on to dispatch to the corresponding ufunc. We can achieve this conveniently by inheriting from the mixin `NDArrayOperatorsMixin`.

```
>>> import numpy.lib.mixins
>>> class DiagonalArray(numpy.lib.mixins.NDArrayOperatorsMixin):
...     def __init__(self, N, value):
...         self._N = N
...         self._i = value
...     def __repr__(self):
...         return f"{self.__class__.__name__}(N={self._N}, value={self._i})"
...     def __array__(self, dtype=None):
...         return self._i * np.eye(self._N, dtype=dtype)
...     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
...         if method == '__call__':
...             N = None
...             scalars = []
...             for input in inputs:
...                 if isinstance(input, Number):
...                     scalars.append(input)
...                 elif isinstance(input, self.__class__):
...                     scalars.append(input._i)
...                     if N is not None:
...                         if N != self._N:
...                             raise TypeError("inconsistent sizes")
...                     else:
...                         N = self._N
...                 else:
...                     return NotImplemented
...             return self.__class__(N, ufunc(*scalars, **kwargs))
...         else:
...             return NotImplemented
```

```
>>> arr = DiagonalArray(5, 1)
>>> arr + 3
DiagonalArray(N=5, value=4)
>>> arr > 0
DiagonalArray(N=5, value=True)
```

Now let’s tackle `__array_function__`. We’ll create dict that maps numpy functions to our custom variants.

```
>>> HANDLED_FUNCTIONS = {}
>>> class DiagonalArray(numpy.lib.mixins.NDArrayOperatorsMixin):
...     def __init__(self, N, value):
...         self._N = N
...         self._i = value
...     def __repr__(self):
...         return f"{self.__class__.__name__}(N={self._N}, value={self._i})"
...     def __array__(self, dtype=None):
...         return self._i * np.eye(self._N, dtype=dtype)
...     def __array_ufunc__(self, ufunc, method, *inputs, **kwargs):
...         if method == '__call__':
...             N = None
...             scalars = []
...             for input in inputs:
...                 # In this case we accept only scalar numbers or DiagonalArrays.
...                 if isinstance(input, Number):
...                     scalars.append(input)
...                 elif isinstance(input, self.__class__):
...                     scalars.append(input._i)
...                     if N is not None:
...                         if N != self._N:
...                             raise TypeError("inconsistent sizes")
...                     else:
...                         N = self._N
...                 else:
...                     return NotImplemented
...             return self.__class__(N, ufunc(*scalars, **kwargs))
...         else:
...             return NotImplemented
...     def __array_function__(self, func, types, args, kwargs):
...         if func not in HANDLED_FUNCTIONS:
...             return NotImplemented
...         # Note: this allows subclasses that don't override
...         # __array_function__ to handle DiagonalArray objects.
...         if not all(issubclass(t, self.__class__) for t in types):
...             return NotImplemented
...         return HANDLED_FUNCTIONS[func](*args, **kwargs)
...
```

A convenient pattern is to define a decorator `implements` that can be used to add functions to `HANDLED_FUNCTIONS`.

```
>>> def implements(np_function):
...    "Register an __array_function__ implementation for DiagonalArray objects."
...    def decorator(func):
...        HANDLED_FUNCTIONS[np_function] = func
...        return func
...    return decorator
...
```

Now we write implementations of numpy functions for `DiagonalArray`. For completeness, to support the usage `arr.sum()` add a method `sum` that calls `numpy.sum(self)`, and the same for `mean`.

```
>>> @implements(np.sum)
... def sum(arr):
...     "Implementation of np.sum for DiagonalArray objects"
...     return arr._i * arr._N
...
>>> @implements(np.mean)
... def mean(arr):
...     "Implementation of np.mean for DiagonalArray objects"
...     return arr._i / arr._N
...
>>> arr = DiagonalArray(5, 1)
>>> np.sum(arr)
5
>>> np.mean(arr)
0.2
```

If the user tries to use any numpy functions not included in `HANDLED_FUNCTIONS`, a `TypeError` will be raised by numpy, indicating that this operation is not supported. For example, concatenating two `DiagonalArrays` does not produce another diagonal array, so it is not supported.

```
>>> np.concatenate([arr, arr])
Traceback (most recent call last):
...
TypeError: no implementation found for 'numpy.concatenate' on types that implement
__array_function__: [<class '__main__.DiagonalArray'>]
```

Additionally, our implementations of `sum` and `mean` do not accept the optional arguments that numpy's implementation does.

```
>>> np.sum(arr, axis=0)
Traceback (most recent call last):
...
TypeError: sum() got an unexpected keyword argument 'axis'
```

The user always has the option of converting to a normal `numpy.ndarray` with **`numpy.asarray`** and using standard numpy from there.

```
>>> np.concatenate([np.asarray(arr), np.asarray(arr)])
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

Refer to the dask source code and cupy source code for more fully-worked examples of custom array containers.

See also NEP 18.

Previous
Structured arrays

Next
Subclassing ndarray