Interoperability with NumPy

NumPy's ndarray objects provide both a high-level API for operations on array-structured data and a concrete implementation of the API based on strided in-RAM storage. While this API is powerful and fairly general, its concrete implementation has limitations. As datasets grow and NumPy becomes used in a variety of new environments and architectures, there are cases where the strided in-RAM storage strategy is inappropriate, which has caused different libraries to reimplement this API for their own uses. This includes GPU arrays (CuPy), Sparse arrays (**`scipy.sparse`**, PyData/Sparse) and parallel arrays (Dask arrays) as well as various NumPy-like implementations in deep learning frameworks, like TensorFlow and PyTorch. Similarly, there are many projects that build on top of the NumPy API for labeled and indexed arrays (XArray), automatic differentiation (JAX), masked arrays (**`numpy.ma`**), physical units (astropy.units, pint, unyt), among others that add additional functionality on top of the NumPy API.

Yet, users still want to work with these arrays using the familiar NumPy API and re-use existing code with minimal (ideally zero) porting overhead. With this goal in mind, various protocols are defined for implementations of multi-dimensional arrays with high-level APIs matching NumPy.

Broadly speaking, there are three groups of features used for interoperability with NumPy:

1. Methods of turning a foreign object into an ndarray;
2. Methods of deferring execution from a NumPy function to another array library;
3. Methods that use NumPy functions and return an instance of a foreign object.

We describe these features below.

1. Using arbitrary objects in NumPy

The first set of interoperability features from the NumPy API allows foreign objects to be treated as NumPy arrays whenever possible. When NumPy functions encounter a foreign object, they will try (in order):

1. The buffer protocol, described in the Python C-API documentation.
2. The `__array_interface__` protocol, described in this page. A precursor to Python's buffer protocol, it defines a way to access the contents of a NumPy array from other C extensions.
3. The `__array__()` method, which asks an arbitrary object to convert itself into an array.

For both the buffer and the `__array_interface__` protocols, the object describes its memory layout and NumPy does everything else (zero-copy if possible). If that's not possible, the object itself is responsible for returning a `ndarray` from `__array__()`.

DLPack is yet another protocol to convert foreign objects to NumPy arrays in a language and device agnostic manner. NumPy doesn't implicitly convert objects to ndarrays using DLPack. It provides the function **`numpy.from_dlpack`** that accepts any object implementing the `__dlpack__` method and outputs a NumPy ndarray (which is generally a view of the input object's data buffer). The Python Specification for DLPack page explains the `__dlpack__` protocol in detail.

The array interface protocol

The array interface protocol defines a way for array-like objects to re-use each other's data buffers. Its implementation relies on the existence of the following attributes or methods:

- `__array_interface__`: a Python dictionary containing the shape, the element type, and optionally, the data buffer address and the strides of an array-like object;
- `__array__()`: a method returning the NumPy ndarray view of an array-like object;

The `__array_interface__` attribute can be inspected directly:

```
>>> import numpy as np
>>> x = np.array([1, 2, 5.0, 8])
>>> x.__array_interface__
{'data': (94708397920832, False), 'strides': None, 'descr': [('', '<f8')],
'typestr': '<f8', 'shape': (4,), 'version': 3}
```

The `__array_interface__` attribute can also be used to manipulate the object data in place:

```
>>> class wrapper():
...     pass
...
>>> arr = np.array([1, 2, 3, 4])
>>> buf = arr.__array_interface__
>>> buf
{'data': (140497590272032, False), 'strides': None, 'descr': [('', '<i8')],
'typestr': '<i8', 'shape': (4,), 'version': 3}
>>> buf['shape'] = (2, 2)
>>> w = wrapper()
>>> w.__array_interface__ = buf
>>> new_arr = np.array(w, copy=False)
>>> new_arr
array([[1, 2],
       [3, 4]])
```

We can check that `arr` and `new_arr` share the same data buffer:

```
>>> new_arr[0, 0] = 1000
>>> new_arr
array([[1000,    2],
       [   3,    4]])
>>> arr
array([1000, 2, 3, 4])
```

The `__array__()` method

The `__array__()` method ensures that any NumPy-like object (an array, any object exposing the array interface, an object whose `__array__()` method returns an array or any nested sequence) that implements it can be used as a NumPy array. If possible, this will mean using `__array__()` to create a NumPy ndarray view of the array-like object. Otherwise, this copies the data into a new ndarray object. This is not optimal, as coercing arrays into ndarrays may cause performance problems or create the need for copies and loss of metadata, as the original object and any attributes/behavior it may have had, is lost.

To see an example of a custom array implementation including the use of `__array__()`, see Writing custom array containers.

The DLPack Protocol

The DLPack protocol defines a memory-layout of strided n-dimensional array objects. It offers the following syntax for data exchange:

1. A **`numpy.from_dlpack`** function, which accepts (array) objects with a `__dlpack__` method and uses that method to construct a new array containing the data from `x`.
2. `__dlpack__(self, stream=None)` and `__dlpack_device__` methods on the array object, which will be called from within `from_dlpack`, to query what device the array is on (may be needed to pass in the correct stream, e.g. in the case of multiple GPUs) and to access the data.

Unlike the buffer protocol, DLPack allows exchanging arrays containing data on devices other than the CPU (e.g. Vulkan or GPU). Since NumPy only supports CPU, it can only convert objects whose data exists on the CPU. But other libraries, like PyTorch and CuPy, may exchange data on GPU using this protocol.

2. Operating on foreign objects without converting

A second set of methods defined by the NumPy API allows us to defer the execution from a NumPy function to another array library.

Consider the following function.

```
>>> import numpy as np
>>> def f(x):
...     return np.mean(np.exp(x))
```

Note that **`np.exp`** is a ufunc, which means that it operates on ndarrays in an element-by-element fashion. On the other hand, **`np.mean`** operates along one of the array's axes.

We can apply `f` to a NumPy ndarray object directly:

```
>>> x = np.array([1, 2, 3, 4])
>>> f(x)
21.1977562209304
```

We would like this function to work equally well with any NumPy-like array object.

NumPy allows a class to indicate that it would like to handle computations in a custom-defined way through the following interfaces:

- `__array_ufunc__`: allows third-party objects to support and override ufuncs.
- `__array_function__`: a catch-all for NumPy functionality that is not covered by the `__array_ufunc__` protocol for universal functions.

As long as foreign objects implement the `__array_ufunc__` or `__array_function__` protocols, it is possible to operate on them without the need for explicit conversion.

The `__array_ufunc__` protocol

A universal function (or ufunc for short) is a "vectorized" wrapper for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs. The output of the ufunc (and its methods) is not necessarily a ndarray, if not all input arguments are ndarrays. Indeed, if any input defines an `__array_ufunc__` method, control will be passed completely to that function, i.e., the ufunc is overridden. The `__array_ufunc__` method defined on that (non-ndarray) object has access to the NumPy ufunc. Because ufuncs have a well-defined structure, the foreign `__array_ufunc__` method may rely on ufunc attributes like `.at()`, `.reduce()`, and others.

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 `__array_function__` protocol

To achieve enough coverage of the NumPy API to support downstream projects, there is a need to go beyond `__array_ufunc__` and implement a protocol that allows arguments of a NumPy function to take control and divert execution to another function (for example, a GPU or parallel implementation) in a way that is safe and consistent across projects.

The semantics of `__array_function__` are very similar to `__array_ufunc__`, except the operation is specified by an arbitrary callable object rather than a ufunc instance and method. For more details, see NEP 18 — A dispatch mechanism for NumPy's high level array functions.

3. Returning foreign objects

A third type of feature set is meant to use the NumPy function implementation and then convert the return value back into an instance of the foreign object. The `__array_finalize__` and `__array_wrap__` methods act behind the scenes to ensure that the return type of a NumPy function can be specified as needed.

The `__array_finalize__` method is the mechanism that NumPy provides to allow subclasses to handle the various ways that new instances get created. This method is called whenever the system internally allocates a new array from an object which is a subclass (subtype) of the ndarray. It can be used to change attributes after construction, or to update meta-information from the "parent."

The `__array_wrap__` method "wraps up the action" in the sense of allowing any object (such as user-defined functions) to set the type of its return value and update attributes and metadata. This can be seen as the opposite of the `__array__` method. At the end of every object that implements `__array_wrap__`, this method is called on the input object with the highest *array priority*, or the output object if one was specified. The `__array_priority__` attribute is used to determine what type of object to return in situations where there is more than one possibility for the Python type of the returned object. For example, subclasses may opt to use this method to transform the output array into an instance of the subclass and update metadata before returning the array to the user.

For more information on these methods, see Subclassing ndarray and Specific features of ndarray sub-typing.

Interoperability examples

Example: Pandas `Series` objects

Consider the following:

```
>>> import pandas as pd
>>> ser = pd.Series([1, 2, 3, 4])
>>> type(ser)
pandas.core.series.Series
```

Now, `ser` is **not** a ndarray, but because it implements the __array_ufunc__ protocol, we can apply ufuncs to it as if it were a ndarray:

```
>>> np.exp(ser)
   0     2.718282
   1     7.389056
   2    20.085537
   3    54.598150
   dtype: float64
>>> np.sin(ser)
   0    0.841471
   1    0.909297
   2    0.141120
   3   -0.756802
   dtype: float64
```

We can even do operations with other ndarrays:

```
>>> np.add(ser, np.array([5, 6, 7, 8]))
   0     6
   1     8
   2    10
   3    12
   dtype: int64
>>> f(ser)
21.1977562209304
>>> result = ser.__array__()
>>> type(result)
numpy.ndarray
```

Example: PyTorch tensors

PyTorch is an optimized tensor library for deep learning using GPUs and CPUs. PyTorch arrays are commonly called *tensors*. Tensors are similar to NumPy's ndarrays, except that tensors can run on GPUs or other hardware accelerators. In fact, tensors and NumPy arrays can often share the same underlying memory, eliminating the need to copy data.

```
>>> import torch
>>> data = [[1, 2],[3, 4]]
>>> x_np = np.array(data)
>>> x_tensor = torch.tensor(data)
```

Note that `x_np` and `x_tensor` are different kinds of objects:

```
>>> x_np
array([[1, 2],
       [3, 4]])
>>> x_tensor
tensor([[1, 2],
        [3, 4]])
```

However, we can treat PyTorch tensors as NumPy arrays without the need for explicit conversion:

```
>>> np.exp(x_tensor)
tensor([[ 2.7183,  7.3891],
        [20.0855, 54.5982]], dtype=torch.float64)
```

Also, note that the return type of this function is compatible with the initial data type.

Warning

While this mixing of ndarrays and tensors may be convenient, it is not recommended. It will not work for non-CPU tensors, and will have unexpected behavior in corner cases. Users should prefer explicitly converting the ndarray to a tensor.

Note

PyTorch does not implement `__array_function__` or `__array_ufunc__`. Under the hood, the `Tensor.__array__()` method returns a NumPy ndarray as a view of the tensor data buffer. See this issue and the `__torch_function__` implementation for details.

Note also that we can see `__array_wrap__` in action here, even though `torch.Tensor` is not a subclass of ndarray:

```
>>> import torch
>>> t = torch.arange(4)
>>> np.abs(t)
tensor([0, 1, 2, 3])
```

PyTorch implements `__array_wrap__` to be able to get tensors back from NumPy functions, and we can modify it directly to control which type of objects are returned from these functions.

Example: CuPy arrays

CuPy is a NumPy/SciPy-compatible array library for GPU-accelerated computing with Python. CuPy implements a subset of the NumPy interface by implementing `cupy.ndarray`, a counterpart to NumPy ndarrays.

```
>>> import cupy as cp
>>> x_gpu = cp.array([1, 2, 3, 4])
```

The `cupy.ndarray` object implements the `__array_ufunc__` interface. This enables NumPy ufuncs to be applied to CuPy arrays (this will defer operation to the matching CuPy CUDA/ROCm implementation of the ufunc):

```
>>> np.mean(np.exp(x_gpu))
array(21.19775622)
```

Note that the return type of these operations is still consistent with the initial type:

```
>>> arr = cp.random.randn(1, 2, 3, 4).astype(cp.float32)
>>> result = np.sum(arr)
>>> print(type(result))
<class 'cupy._core.core.ndarray'>
```

See this page in the CuPy documentation for details.

`cupy.ndarray` also implements the `__array_function__` interface, meaning it is possible to do operations such as

```
>>> a = np.random.randn(100, 100)
>>> a_gpu = cp.asarray(a)
>>> qr_gpu = np.linalg.qr(a_gpu)
```

CuPy implements many NumPy functions on `cupy.ndarray` objects, but not all. See the CuPy documentation for details.

Example: Dask arrays

Dask is a flexible library for parallel computing in Python. Dask Array implements a subset of the NumPy ndarray interface using blocked algorithms, cutting up the large array into many small arrays. This allows computations on larger-than-memory arrays using multiple cores.

Dask supports `__array__()` and `__array_ufunc__`.

```
>>> import dask.array as da
>>> x = da.random.normal(1, 0.1, size=(20, 20), chunks=(10, 10))
>>> np.mean(np.exp(x))
dask.array<mean_agg-aggregate, shape=(), dtype=float64, chunksize=(),
chunktype=numpy.ndarray>
>>> np.mean(np.exp(x)).compute()
5.090097550553843
```

Note

Dask is lazily evaluated, and the result from a computation isn't computed until you ask for it by invoking `compute()`.

See the Dask array documentation and the scope of Dask arrays interoperability with NumPy arrays for details.

Example: DLPack

Several Python data science libraries implement the `__dlpack__` protocol. Among them are PyTorch and CuPy. A full list of libraries that implement this protocol can be found on this page of DLPack documentation.

Convert a PyTorch CPU tensor to NumPy array:

```
>>> import torch
>>> x_torch = torch.arange(5)
>>> x_torch
tensor([0, 1, 2, 3, 4])
>>> x_np = np.from_dlpack(x_torch)
>>> x_np
array([0, 1, 2, 3, 4])
>>> # note that x_np is a view of x_torch
>>> x_torch[1] = 100
>>> x_torch
tensor([  0, 100,   2,   3,   4])
>>> x_np
array([  0, 100,   2,   3,   4])
```

The imported arrays are read-only so writing or operating in-place will fail:

```
>>> x.flags.writeable
False
>>> x_np[1] = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: assignment destination is read-only
```

A copy must be created in order to operate on the imported arrays in-place, but will mean duplicating the memory. Do not do this for very large arrays:

```
>>> x_np_copy = x_np.copy()
>>> x_np_copy.sort()  # works
```

Note

Note that GPU tensors can't be converted to NumPy arrays since NumPy doesn't support GPU devices:

```
>>> x_torch = torch.arange(5, device='cuda')
>>> np.from_dlpack(x_torch)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: Unsupported device in DLTensor.
```

But, if both libraries support the device the data buffer is on, it is possible to use the `__dlpack__` protocol (e.g. PyTorch and CuPy):

```
>>> x_torch = torch.arange(5, device='cuda')
>>> x_cupy = cupy.from_dlpack(x_torch)
```

Similarly, a NumPy array can be converted to a PyTorch tensor:

```
>>> x_np = np.arange(5)
>>> x_torch = torch.from_dlpack(x_np)
```

Read-only arrays cannot be exported:

```
>>> x_np = np.arange(5)
>>> x_np.flags.writeable = False
>>> torch.from_dlpack(x_np)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File ".../site-packages/torch/utils/dlpack.py", line 63, in from_dlpack
    dlpack = ext_tensor.__dlpack__()
TypeError: NumPy currently only supports dlpack for writeable arrays
```

Further reading

- The array interface protocol
- Writing custom array containers
- Special attributes and methods (details on the `__array_ufunc__` and `__array_function__` protocols)
- Subclassing ndarray (details on the `__array_wrap__` and `__array_finalize__` methods)
- Specific features of ndarray sub-typing (more details on the implementation of `__array_finalize__`, `__array_wrap__` and `__array_priority__`)
- NumPy roadmap: interoperability
- PyTorch documentation on the Bridge with NumPy

Previous
Copies and views

Next
Miscellaneous