uarray Documentation

Release 0.5.1+7.gb0a5070

Quansight-Labs

Aug 14, 2019

CONTENTS

Warning: *uarray* is a developer tool, it is not meant to be used directly by end-users.

Warning: This document is meant to elicit discussion from the broader community and to help drive the direction that *uarray* goes towards. Examples provided here may not be immediately stable.

Note: This page describes the overall philosophy behind *uarray*. For usage instructions, see the *uarray* API documentation page. If you are interested in an augmentation for NEP-22, please see the unumpy page.

uarray is a backend system for Python that allows you to separately define an API, along with backends that contain separate implementations of that API.

unumpy builds on top of *uarray*. It is an effort to specify the core NumPy API, and provide backends for the API.

CHAPTER

ONE

WHAT'S NEW IN UARRAY?

uarray is, to our knowledge, the first backend-system for Python that's generic enough to cater to the use-cases of many libraries, while at the same time, being library independent.

unumpy is the first approach to leverage *uarray* in order to build a generic backend system for (what we hope will be) the core NumPy API. It will be possible to create a backend object, and use that to perform operations. In addition, it will be possible to change the used backend via a context manager.

CHAPTER

TWO

BENEFITS FOR END USERS

End-users can easily take their code written for one backend and use it on another backend with a simple switch (using a Python context manager). This can have any number of effects, depending on the functionality of the library. For example:

- For Matplotlib, changing styles of plots or producing different windows or image formats.
- For Tensorly, providing a different computation backend that can be distributed or target the GPU or sparse arrays.
- For `unumpy`, it can do a similar thing: provide users with code they already wrote for *numpy* and easily switch to a different backend.

CHAPTER

THREE

BENEFITS FOR LIBRARY AUTHORS

To library authors, the benefits come in two forms: First, it allows them to build their libraries to be implementation independent. In code that builds itself on top of `unumpy`, it would be very easy to target the GPU, use sparse arrays or do any kind of distributed computing.

The second is to allow a way to separate the interface from implementation, and easily allow a way to switch an implementation.

CHAPTER

FOUR

RELATION TO THE NUMPY DUCK-ARRAY ECOSYSTEM

uarray is a backend/dispatch mechanism with a focus on array computing and the needs of the wider array community, by allowing a clean way to register an implementation for any Python object (functions, classes, class methods, properties, dtypes, ...), it also provides an important building block for NEP-22. It is meant to address the shortcomings of NEP-18 and NEP-13; while still holding nothing in *uarray* itself that's specific to array computing or the NumPy API.

CHAPTER

FIVE

WHERE TO FROM HERE?

Choose the documentation page relevant to you:

- *Documentation for API authors*
- *Documentation for backend providers*
- *End-user quickstart*

5.1 End-user quickstart

Ideally, the only thing an end-user should have to do is set the backend and and its options. Given a backend, you (as the end-user) can decide to do one of two things:

- Set the backend permanently (use the `set_global_backend` function).
- Set the backend temporarily (use the `set_backend` context manager).

Note: API authors may want to wrap these methods and provide their own methods.

Also of note may be the `BackendNotImplementedError`, which is raised when none of the selected backends have an implementation for a multimethod.

5.1.1 Setting the backend temporarily

To set the backend temporarily, use the `set_backend` context manager.

```
import uarray as ua

with ua.set_backend(mybackend):
    # Use multimethods (or code dependent on them) here.
```

5.1.2 Setting the backend permanently

To set the backend permanently, use the `set_global_backend` method. It is a recommendation that the global backend should not depend on any other backend, as it is not guaranteed that another backend will be available.

You can also register backends other than the global backend for permanent use, but the global backend will be tried first outside of a `set_backend` context. This can be done via `register_backend`.

```
import uarray as ua

ua.set_global_backend(mybackend)

# Use relevant multimethods here.
```

5.2 Documentation for backend providers

Backend providers can provide a back-end for a defined API within the *uarray* ecosystem. To find out how to define your own API with *uarray*, see *Documentation for API authors*. To find out how your backend will be provided, use *End-user quickstart*.

Backend providers need to be aware of three protocols: `__ua_domain__`, `__ua_function__` and `__ua_convert__`. The first two are mandatory and the last is optional.

5.2.1 `__ua_domain__`

`__ua_domain__` is a string containing the domain of the backend. This is, by convention, the name of the module (or one of its dependencies or parents) that contain the multimethods. For example, `scipy` and `numpy.fft` could both use the `numpy` domain.

5.2.2 `__ua_function__`

This is the most important protocol, one that defines the implementation of a multimethod. It has the signature `(method, args, kwargs)`. Note that it is called in this form, so if your backend is an object instead of a module, you should add `self`. `method` is the multimethod being called, and it is guaranteed that it is in the same domain as the backend. `args` and `kwargs` are the arguments to the function, possibly after conversion (explained below)

Returning `NotImplemented` signals that the backend does not support this operation.

5.2.3 `__ua_convert__`

All dispatchable arguments are passed through `__ua_convert__` before being passed into `__ua_function__`. This protocol has the signature `(dispatchables, coerce)`, where `dispatchables` is an iterable of *Dispatchable* and `coerce` is whether or not to coerce forcefully. `dispatch_type` is the mark of the object to be converted, and `coerce` specifies whether or not to "force" the conversion. By convention, operations larger than `O(log n)` (where `n` is the size of the object in memory) should only be done if `coerce` is `True`. In addition, there are arguments wrapped as non-coercible via the `coercible` attribute, if these *must* be coerced, then one should return `NotImplemented`.

A convenience wrapper for converting a single object, *wrap_single_convertor* is provided.

Returning `NotImplemented` signals that the backend does not support conversion of the given object.

5.2.4 `skip_backend`

If a backend consumes multimethods from a domain, and provides multimethods for that same domain, it may wish to have the ability to use multimethods while excluding itself from the list of tried backends in order to avoid infinite

recursion. This allows the backend to implement its functions in terms of functions provided by other backends. This is the purpose of the *skip_backend* decorator.

5.2.5 The process that takes place when the backend is tried

First of all, the backend's __ua_convert__ method is tried. If it does not return NotImplemented, then the backend's __ua_function__ protocol is tried. If a value other than NotImplemented is returned, it is assumed to be the final return value. Any exceptions raised are propagated up the call stack.

5.2.6 Examples

Examples for library authors can be found in the source of unumpy.numpy_backend and other *_backend.py files in this directory.

5.3 Documentation for API authors

Multimethods are the most important part of *uarray*. They are created via the *generate_multimethod* function. Multimethods define the API of a project, and backends have to be written against this API. You should see *Documentation for backend providers* for how to define a backend against the multimethods you write, or *End-user quickstart* for how to switch backends for a given API.

A multimethod has the following parts:

- Domain
- Argument extractor
- Argument replacer
- Default implementation

We will go through each of these in detail now.

5.3.1 Domain

See the *glossary for domain*.

5.3.2 Argument extractor

An argument extractor extracts arguments *marked* as a given type from the list of given arguments. Note that the objects extracted don't necessarily have to be in the list of arguments, they can be arbitrarily nested within the arguments. For example, extracting each argument from a list is a possibility. Note that the order is important, as it will come into play later. This function should return an iterable of *Dispatchable*.

This function has the same signature as the multimethod itself, and the documentation, name and so on are copied from the argument extractor via functools.wraps.

5.3.3 Argument replacer

The argument replacer takes in the arguments and dispatchable arguments, and its job is to replace the arguments previously extracted by the argument extractor by by other arguments provided in the list. Therefore, the signature of this function is `(args, kwargs, dispatchable_args)`, and it returns an `args`/`kwargs` pair. We realise this is a hard problem in general, so we have provided a few simplifications, such as that the default-valued keyword arguments will be removed from the list.

We recommend following the pattern in this file for optimal operation: passing the `args`/`kwargs` into a function with a similar signature and then return the modified `args`/`kwargs`.

5.3.4 Default implementation

This is a default implementation for the multimethod, ideally with the same signature as the original multimethod. It can also be used to provide one multimethod in terms of others, even if the default implementation for the. downstream multimethods is not defined.

5.3.5 Examples

Examples of writing multimethods are found in this file. It also teaches some advanced techniques, such as overriding instance methods, including `__call__`. The same philosophy may be used to override properties, static methods and class methods.

5.4 Glossary

5.4.1 Multimethod

A method, possibly with a default/reference implementation, that can have other implementations provided by different backends.

If a multimethod does not have an implementation, a *`BackendNotImplementedError`* is raised.

5.4.2 Backend

A backend is an entity that can provide implementations for different functions. It can also (optionally) receive some options from the user about how to process the implementations. A backend can be set permanently or temporarily.

5.4.3 Domain

A domain is a collection or grouping of multimethods. A domain's string, by convention (although not by force) is the name of the module that provides the multimethods.

5.4.4 Dispatching

Dispatching is the process of forward a function call to an implementation in a backend.

5.4.5 Conversion

A backend might have different object types compared to the reference implementation, or it might require some other conversions of objects. Conversion is the process of converting any given object into a library's native form.

Coercion

Coercions are conversions that may take a long time, usually those involving copying or moving of data. As a rule of thumb, conversions longer than `O(log n)` (where `n` is the size of the object in memory) should be made into coercions.

5.4.6 Marking

Marking is the process of telling the backend what convertor to use for a given argument.

5.5 uarray

`uarray` is built around a back-end protocol, and overridable multimethods. It is necessary to define multimethods for back-ends to be able to override them. See the documentation of *`generate_multimethod`* on how to write multimethods.

Let's start with the simplest:

`__ua_domain__` defines the back-end *domain*. The domain consists of period- separated string consisting of the modules you extend plus the submodule. For example, if a submodule `module2.submodule` extends `module1` (i.e., it exposes dispatchables marked as types available in `module1`), then the domain string should be `"module1.module2.submodule"`.

For the purpose of this demonstration, we'll be creating an object and setting its attributes directly. However, note that you can use a module or your own type as a backend as well.

```
>>> class Backend: pass
>>> be = Backend()
>>> be.__ua_domain__ = "ua_examples"
```

It might be useful at this point to sidetrack to the documentation of *`generate_multimethod`* to find out how to generate a multimethod overridable by *`uarray`*. Needless to say, writing a backend and creating multimethods are mostly orthogonal activities, and knowing one doesn't necessarily require knowledge of the other, although it is certainly helpful. We expect core API designers/specifiers to write the multimethods, and implementors to override them. But, as is often the case, similar people write both.

Without further ado, here's an example multimethTrueod:

```
>>> import uarray as ua
>>> def override_me(a, b):
...   return Dispatchable(a, int),
>>> def override_replacer(args, kwargs, dispatchables):
...     return (dispatchables[0], args[1]), {}
>>> overridden_me = ua.generate_multimethod(
...     override_me, override_replacer, "ua_examples"
... )
```

Next comes the part about overriding the multimethod. This requires the `__ua_function__` protocol, and the `__ua_convert__` protocol. The `__ua_function__` protocol has the signature `(method, args, kwargs)`

where `method` is the passed multimethod, `args`/`kwargs` specify the arguments and `dispatchables` is the list of converted dispatchables passed in.

```
>>> def __ua_function__(method, args, kwargs):
...     return method.__name__, args, kwargs
>>> be.__ua_function__ = __ua_function__
```

The other protocol of interest is the `__ua_convert__` protocol. It has the signature `(dispatchables, coerce)`. When `coerce` is `False`, conversion between the formats should ideally be an `O(1)` operation, but it means that no memory copying should be involved, only views of the existing data.

```
>>> def __ua_convert__(dispatchables, coerce):
...     for d in dispatchables:
...         if d.type is int:
...             if coerce and d.coercible:
...                 yield str(d.value)
...             else:
...                 yield d.value
>>> be.__ua_convert__ = __ua_convert__
```

Now that we have defined the backend, the next thing to do is to call the multimethod.

```
>>> with ua.set_backend(be):
...      overridden_me(1, "2")
('override_me', (1, '2'), {})
```

Note that the marked type has no effect on the actual type of the passed object. We can also coerce the type of the input.

```
>>> with ua.set_backend(be, coerce=True):
...     overridden_me(1, "2")
...     overridden_me(1.0, "2")
('override_me', ('1', '2'), {})
('override_me', ('1.0', '2'), {})
```

Another feature is that if you remove `__ua_convert__`, the arguments are not converted at all and it's up to the backend to handle that.

```
>>> del be.__ua_convert__
>>> with ua.set_backend(be):
...     overridden_me(1, "2")
('override_me', (1, '2'), {})
```

You also have the option to return `NotImplemented`, in which case processing moves on to the next back-end, which in this case, doesn't exist. The same applies to `__ua_convert__`.

```
>>> be.__ua_function__ = lambda *a, **kw: NotImplemented
>>> with ua.set_backend(be):
...     overridden_me(1, "2")
Traceback (most recent call last):
    ...
uarray.backend.BackendNotImplementedError: ...
```

The last possibility is if we don't have `__ua_convert__`, in which case the job is left up to `__ua_function__`, but putting things back into arrays after conversion will not be possible.

Functions

all_of_type(arg_type)	Marks all unmarked arguments as a given type.
create_multimethod(*args, **kwargs)	Creates a decorator for generating multimethods.
generate_multimethod(argument_extractor, ...)	Generates a multimethod.
mark_as(dispatch_type)	Creates a utility function to mark something as a specific type.
set_backend(backend[, coerce, only])	A context manager that sets the preferred backend.
set_global_backend(backend[, coerce, only])	This utility method replaces the default backend for permanent use.
register_backend(backend)	This utility method sets registers backend for permanent use.
clear_backends(domain[, registered, globals])	This utility method clears registered backends.
skip_backend(backend)	A context manager that allows one to skip a given backend from processing entirely.
wrap_single_convertor(convert_single)	Wraps a `__ua_convert__` defined for a single element to all elements.

5.5.1 all_of_type

uarray.**all_of_type**(*arg_type*)

Marks all unmarked arguments as a given type.

Examples

```
>>> @all_of_type(str)
... def f(a, b):
...     return a, Dispatchable(b, int)
>>> f('a', 1)
(<Dispatchable: type=<class 'str'>, value='a'>, <Dispatchable: type=<class 'int'>,
↪ value=1>)
```

5.5.2 create_multimethod

uarray.**create_multimethod**(**args*, ***kwargs*)

Creates a decorator for generating multimethods.

This function creates a decorator that can be used with an argument extractor in order to generate a multimethod. Other than for the argument extractor, all arguments are passed on to *generate_multimethod*.

See also:

generate_multimethod() Generates a multimethod.

5.5.3 generate_multimethod

uarray.**generate_multimethod**(*argument_extractor: Callable[[...], Tuple[Dispatchable, ...]]*, *argument_replacer: Callable[[Tuple, Dict, Tuple], Tuple[Tuple, Dict]]*, *domain: str*, *default: Optional[Callable] = None*)

Generates a multimethod.

Parameters

- **argument_extractor** (*ArgumentExtractorType*) – A callable which extracts the dispatchable arguments. Extracted arguments should be marked by the *Dispatchable* class. It has the same signature as the desired multimethod.
- **argument_replacer** (*ArgumentReplacerType*) – A callable with the signature (args, kwargs, dispatchables), which should also return an (args, kwargs) pair with the dispatchables replaced inside the args/kwargs.
- **domain** (*str*) – A string value indicating the domain of this multimethod.
- **default** (*Optional[Callable], optional*) – The default implementation of this multimethod, where `None` (the default) specifies there is no default implementation.

Examples

In this example, `a` is to be dispatched over, so we return it, while marking it as an `int`. The trailing comma is needed because the args have to be returned as an iterable.

```
>>> def override_me(a, b):
...   return Dispatchable(a, int),
```

Next, we define the argument replacer that replaces the dispatchables inside args/kwargs with the supplied ones.

```
>>> def override_replacer(args, kwargs, dispatchables):
...     return (dispatchables[0], args[1]), {}
```

Next, we define the multimethod.

```
>>> overridden_me = generate_multimethod(
...     override_me, override_replacer, "ua_examples"
... )
```

Notice that there's no default implementation, unless you supply one.

```
>>> overridden_me(1, "a")
Traceback (most recent call last):
    ...
uarray.backend.BackendNotImplementedError: ...
>>> overridden_me2 = generate_multimethod(
...     override_me, override_replacer, "ua_examples", default=lambda x, y: (x, y)
... )
>>> overridden_me2(1, "a")
(1, 'a')
```

See also:

uarray() See the module documentation for how to override the method by creating backends.

5.5.4 mark_as

uarray.**mark_as**(*dispatch_type*)

Creates a utility function to mark something as a specific type.

Examples

```
>>> mark_int = mark_as(int)
>>> mark_int(1)
<Dispatchable: type=<class 'int'>, value=1>
```

5.5.5 set_backend

uarray.**set_backend**(*backend*, *coerce=False*, *only=False*)

A context manager that sets the preferred backend.

Parameters

- **backend** – The backend to set.
- **coerce** – Whether or not to coerce to a specific backend's types. Implies `only`.
- **only** – Whether or not this should be the last backend to try.

See also:

skip_backend() A context manager that allows skipping of backends.

set_global_backend() Set a single, global backend for a domain.

5.5.6 set_global_backend

uarray.**set_global_backend**(*backend*, *coerce=False*, *only=False*)

This utility method replaces the default backend for permanent use. It will be tried in the list of backends automatically, unless the `only` flag is set on a backend. This will be the first tried backend outside the *set_backend* context manager.

Note that this method is not thread-safe.

> **Warning:** We caution library authors against using this function in their code. We do *not* support this use-case. This function is meant to be used only by users themselves, or by a reference implementation, if one exists.

Parameters **backend** – The backend to register.

See also:

set_backend() A context manager that allows setting of backends.

skip_backend() A context manager that allows skipping of backends.

5.5.7 register_backend

uarray.**register_backend**(*backend*)

This utility method sets registers backend for permanent use. It will be tried in the list of backends automatically, unless the `only` flag is set on a backend.

Note that this method is not thread-safe.

Parameters **backend** – The backend to register.

5.5.8 clear_backends

uarray.**clear_backends**(*domain*, *registered=True*, *globals=False*)

This utility method clears registered backends.

Warning: We caution library authors against using this function in their code. We do *not* support this use-case. This function is meant to be used only by users themselves.

Warning: Do NOT use this method inside a multimethod call, or the program is likely to crash.

Parameters

- **domain** (*Optional[str]*) – The domain for which to de-register backends. `None` means de-register for all domains.
- **registered** (*bool*) – Whether or not to clear registered backends. See *register_backend*.
- **globals** (*bool*) – Whether or not to clear global backends. See *set_global_backend*.

See also:

register_backend() Register a backend globally.

set_global_backend() Set a global backend.

5.5.9 skip_backend

uarray.**skip_backend**(*backend*)

A context manager that allows one to skip a given backend from processing entirely. This allows one to use another backend's code in a library that is also a consumer of the same backend.

Parameters **backend** – The backend to skip.

See also:

set_backend() A context manager that allows setting of backends.

set_global_backend() Set a single, global backend for a domain.

5.5.10 wrap_single_convertor

uarray.**wrap_single_convertor**(*convert_single*)

Wraps a `__ua_convert__` defined for a single element to all elements. If any of them return `NotImplemented`, the operation is assumed to be undefined.

Accepts a signature of (value, type, coerce).

Classes

Dispatchable(value, dispatch_type[, coercible])	A utility class which marks an argument with a specific dispatch type.

5.5.11 Dispatchable

class uarray.**Dispatchable**(*value*, *dispatch_type*, *coercible=True*)

A utility class which marks an argument with a specific dispatch type.

value

The value of the Dispatchable.

type

The type of the Dispatchable.

Examples

```
>>> x = Dispatchable(1, str)
>>> x
<Dispatchable: type=<class 'str'>, value=1>
```

See also:

all_of_type Marks all unmarked parameters of a function.

mark_as Allows one to create a utility function to mark as a given type.

Methods

Dispatchable.__init__(value, dispatch_type)	Initialize self.

Dispatchable.__init__

Dispatchable.**__init__**(*value*, *dispatch_type*, *coercible=True*)

Initialize self. See help(type(self)) for accurate signature.

Exceptions

BackendNotImplementedError	An exception that is thrown when no compatible backend is found for a method.

5.5.12 BackendNotImplementedError

exception uarray.**BackendNotImplementedError**

An exception that is thrown when no compatible backend is found for a method.

5.5.13 Design Philosophies

The following section discusses the design philosophies of *uarray*, and the reasoning behind some of these philosophies.

Modularity

uarray (and its sister modules unumpy and others to come) were designed from the ground-up to be modular. This is part of why *uarray* itself holds the core backend and dispatch machinery, and unumpy holds the actual multimethods. Also, unumpy can be developed completely separately to *uarray*, although the ideal place to have it would be NumPy itself.

However, the benefit to having it separate is that it could span multiple NumPy versions, even before NEP-18 (or even NEP-13) was available. Another benefit is that it can have a faster release cycle to help it achieve this.

Separate Imports

Code wishing to use the backend machinery for NumPy (as an example) will use the statement `import unumpy as np` instead of the usual `import numpy as np`. This is deliberate: it makes dispatching opt-in instead of being forced to use it, and the overhead associated with it. However, a package is free to define its main methods as the dispatchable versions, thereby allowing dispatch on the default implementation.

Extensibility *and* Choice

If some effort is put into the dispatch machinery, it's possible to dispatch over arbitrary objects — including arrays, dtypes, and so on. A method defines the type of each dispatchable argument, and backends are *only* passed types they know how to dispatch over, when deciding whether or not to use that backend. For example, if a backend doesn't know how to dispatch over dtypes, it won't be asked to decide based on that front.

Methods can have a default implementation in terms of other methods, but they're still overridable.

This means that only one framework is needed to, for example, dispatch over `ufunc` s, arrays, dtypes and all other primitive objects in NumPy, while keeping the core *uarray* code independent of NumPy and even unumpy.

Backends can span modules, so SciPy could jump in and define its own methods on NumPy objects and make them overridable within the NumPy backend.

User Choice

The users of unumpy or *uarray* can choose which backend they want to prefer with a simple context manager. They also have the ability to force a backend, and to skip a backend. This is useful for array-like objects that provide other array-like objects by composing them. For example, Dask could perform all its blockwise function calls with the following psuedocode (obviously, this is simplified):

```
in_arrays = extract_inner_arrays(input_arrays)
out_arrays = []
for input_arrays_single in in_arrays:
    args, kwargs = blockwise_function.replace_args_kwargs(
        args, kwargs, input_arrays_single)
    with ua.skip_backend(DaskBackend):
        out_arrays_single = blockwise_function(*args, **kwargs)
    out_arrays.append(out_arrays_single)

return combine_arrays(out_arrays)
```

A user would simply do the following:

```
with ua.use_backend(DaskBackend):
    # Write all your code here
    # It will prefer the Dask backend
```

There is no default backend, to `unumpy`, NumPy is just another backend. One can register backends, which will all be tried in indeterminate order when no backend is selected.

Addressing past flaws

The progress on NumPy's side for defining an override mechanism has been slow, with NEP-13 being first introduced in 2013, and with the wealth of dispatchable objects (including arrays, ufuns and dtypes), and with the advent of libraries like Dask, CuPy, Xarray, PyData/Sparse and XND, it has become clear that the need for alternative array-like implementations is growing. There are even other libraries like PyTorch, and TensorFlow that'd be possible to express in NumPy API-like terms. Another example includes the Keras API, for which an overridable `ukeras` could be created, similar to `unumpy`.

`uarray` is intended to have fast development to fill the need posed by these communities, while keeping itself as general as possible, and quickly reach maturity, after which backwards compatibility will be guaranteed.

Performance considerations will come only after such a state has been reached.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

u

INDEX

Symbols

A

B

C

D

G

M

R

S

T

U

V

W