Universal functions (ufunc) basics

See also

Universal functions (ufunc)

A universal function (or ufunc for short) is a function that operates on `ndarrays` in an element-by-element fashion, supporting array broadcasting, type casting, and several other standard features. That is, a ufunc is a "vectorized" wrapper for a function that takes a fixed number of specific inputs and produces a fixed number of specific outputs.

In NumPy, universal functions are instances of the `numpy.ufunc` class. Many of the built-in functions are implemented in compiled C code. The basic ufuncs operate on scalars, but there is also a generalized kind for which the basic elements are sub-arrays (vectors, matrices, etc.), and broadcasting is done over other dimensions. The simplest example is the addition operator:

```
>>> np.array([0,2,3,4]) + np.array([1,1,-1,2])
array([1, 3, 2, 6])
```

One can also produce custom `numpy.ufunc` instances using the `numpy.frompyfunc` factory function.

Ufunc methods

All ufuncs have four methods. They can be found at Methods. However, these methods only make sense on scalar ufuncs that take two input arguments and return one output argument. Attempting to call these methods on other ufuncs will cause a `ValueError`.

The reduce-like methods all take an *axis* keyword, a *dtype* keyword, and an *out* keyword, and the arrays must all have dimension >= 1. The *axis* keyword specifies the axis of the array over which the reduction will take place (with negative values counting backwards). Generally, it is an integer, though for `numpy.ufunc.reduce`, it can also be a tuple of `int` to reduce over several axes at once, or `None`, to reduce over all axes. For example:

```
>>> x = np.arange(9).reshape(3,3)
>>> x
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.add.reduce(x, 1)
array([ 3, 12, 21])
>>> np.add.reduce(x, (0, 1))
36
```

The *dtype* keyword allows you to manage a very common problem that arises when naively using `ufunc.reduce`. Sometimes you may have an array of a certain data type and wish to add up all of its elements, but the result does not fit into the data type of the array. This commonly happens if you have an array of single-byte integers. The *dtype* keyword allows you to alter the data type over which the reduction takes place (and therefore the type of the output). Thus, you can ensure that the output is a data type with precision large enough to handle your output. The responsibility of altering the reduce type is mostly up to you. There is one exception: if no *dtype* is given for a reduction on the "add" or "multiply" operations, then if the input type is an integer (or Boolean) data-type and smaller than the size of the `numpy.int_` data type, it will be internally upcast to the `int_` (or `numpy.uint`) data-type. In the previous example:

```
>>> x.dtype
dtype('int64')
>>> np.multiply.reduce(x, dtype=float)
array([ 0., 28., 80.])
```

Finally, the *out* keyword allows you to provide an output array (for single-output ufuncs, which are currently the only ones supported; for future extension, however, a tuple with a single argument can be passed in). If *out* is given, the *dtype* argument is ignored. Considering x from the previous example:

```
>>> y = np.zeros(3, dtype=int)
>>> y
array([0, 0, 0])
>>> np.multiply.reduce(x, dtype=float, out=y)
array([ 0, 28, 80])
```

Ufuncs also have a fifth method, **numpy.ufunc.at**, that allows in place operations to be performed using advanced indexing. No buffering is used on the dimensions where advanced indexing is used, so the advanced index can list an item more than once and the operation will be performed on the result of the previous operation for that item.

Output type determination

The output of the ufunc (and its methods) is not necessarily an **ndarray**, if all input arguments are not **ndarrays**. Indeed, if any input defines an **__array_ufunc__** method, control will be passed completely to that function, i.e., the ufunc is overridden.

If none of the inputs overrides the ufunc, then all output arrays will be passed to the **__array_prepare__** and **__array_wrap__** methods of the input (besides **ndarrays**, and scalars) that defines it **and** has the highest **__array_priority__** of any other input to the universal function. The default **__array_priority__** of the ndarray is 0.0, and the default **__array_priority__** of a subtype is 0.0. Matrices have **__array_priority__** equal to 10.0.

All ufuncs can also take output arguments. If necessary, output will be cast to the data-type(s) of the provided output array(s). If a class with an **__array__** method is used for the output, results will be written to the object returned by **__array__**. Then, if the class also has an **__array_prepare__** method, it is called so metadata may be determined based on the context of the ufunc (the context consisting of the ufunc itself, the arguments passed to the ufunc, and the ufunc domain.) The array object returned by **__array_prepare__** is passed to the ufunc for computation. Finally, if the class also has an **__array_wrap__** method, the returned **ndarray** result will be passed to that method just before passing control back to the caller.

Broadcasting

See also

Broadcasting basics

Each universal function takes array inputs and produces array outputs by performing the core function element-wise on the inputs (where an element is generally a scalar, but can be a vector or higher-order sub-array for generalized ufuncs). Standard broadcasting rules are applied so that inputs not sharing exactly the same shapes can still be usefully operated on.

By these rules, if an input has a dimension size of 1 in its shape, the first data entry in that dimension will be used for all calculations along that dimension. In other words, the stepping machinery of the ufunc will simply not step along that dimension (the stride will be 0 for that dimension).

Type casting rules

Note

In NumPy 1.6.0, a type promotion API was created to encapsulate the mechanism for determining output types. See the functions `numpy.result_type`, `numpy.promote_types`, and `numpy.min_scalar_type` for more details.

At the core of every ufunc is a one-dimensional strided loop that implements the actual function for a specific type combination. When a ufunc is created, it is given a static list of inner loops and a corresponding list of type signatures over which the ufunc operates. The ufunc machinery uses this list to determine which inner loop to use for a particular case. You can inspect the `.types` attribute for a particular ufunc to see which type combinations have a defined inner loop and which output type they produce (character codes are used in said output for brevity).

Casting must be done on one or more of the inputs whenever the ufunc does not have a core loop implementation for the input types provided. If an implementation for the input types cannot be found, then the algorithm searches for an implementation with a type signature to which all of the inputs can be cast “safely.” The first one it finds in its internal list of loops is selected and performed, after all necessary type casting. Recall that internal copies during ufuncs (even for casting) are limited to the size of an internal buffer (which is user settable).

Note

Universal functions in NumPy are flexible enough to have mixed type signatures. Thus, for example, a universal function could be defined that works with floating-point and integer values. See `numpy.ldexp` for an example.

By the above description, the casting rules are essentially implemented by the question of when a data type can be cast “safely” to another data type. The answer to this question can be determined in Python with a function call: `can_cast(fromtype, totype)`. The example below shows the results of this call for the 24 internally supported types on the author’s 64-bit system. You can generate this table for your system with the code given in the example.

Example

Code segment showing the “can cast safely” table for a 64-bit system. Generally the output depends on the system; your system might result in a different table.

```
>>> mark = {False: ' -', True: ' Y'}
>>> def print_table(ntypes):
...     print('X ' + ' '.join(ntypes))
...     for row in ntypes:
...         print(row, end='')
...         for col in ntypes:
...             print(mark[np.can_cast(row, col)], end='')
...         print()
...
>>> print_table(np.typecodes['All'])
X ? b h i l q p B H I L Q P e f d g F D G S U V O M m
? Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y - Y
b - Y Y Y Y Y Y - - - - - - Y Y Y Y Y Y Y Y Y Y Y - Y
h - - Y Y Y Y Y - - - - - - - Y Y Y Y Y Y Y Y Y Y - Y
i - - - Y Y Y Y - - - - - - - - Y Y - Y Y Y Y Y Y - Y
l - - - - Y Y Y - - - - - - - - Y Y - Y Y Y Y Y Y - Y
q - - - - Y Y Y - - - - - - - - Y Y - Y Y Y Y Y Y - Y
p - - - - Y Y Y - - - - - - - - Y Y - Y Y Y Y Y Y - Y
B - - Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y Y - Y
H - - - Y Y Y Y - Y Y Y Y Y - Y Y Y Y Y Y Y Y Y Y - Y
I - - - - Y Y Y - - Y Y Y Y - - Y Y - Y Y Y Y Y Y - Y
L - - - - - - - - - - Y Y Y - - Y Y - Y Y Y Y Y Y - -
Q - - - - - - - - - - Y Y Y - - Y Y - Y Y Y Y Y Y - -
P - - - - - - - - - - Y Y Y - - Y Y - Y Y Y Y Y Y - -
e - - - - - - - - - - - - - Y Y Y Y Y Y Y Y Y Y Y - -
f - - - - - - - - - - - - - - Y Y Y Y Y Y Y Y Y Y - -
d - - - - - - - - - - - - - - - Y Y - Y Y Y Y Y Y - -
g - - - - - - - - - - - - - - - - Y - - Y Y Y Y Y - -
F - - - - - - - - - - - - - - - - - Y Y Y Y Y Y Y - -
D - - - - - - - - - - - - - - - - - - Y Y Y Y Y Y - -
G - - - - - - - - - - - - - - - - - - - Y Y Y Y Y - -
S - - - - - - - - - - - - - - - - - - - - Y Y Y Y - -
U - - - - - - - - - - - - - - - - - - - - - Y Y Y - -
V - - - - - - - - - - - - - - - - - - - - - - Y Y - -
O - - - - - - - - - - - - - - - - - - - - - - - Y - -
M - - - - - - - - - - - - - - - - - - - - - - Y Y Y -
m - - - - - - - - - - - - - - - - - - - - - - Y Y - Y
```

You should note that, while included in the table for completeness, the 'S', 'U', and 'V' types cannot be operated on by ufuncs. Also, note that on a 32-bit system the integer types may have different sizes, resulting in a slightly altered table.

Mixed scalar-array operations use a different set of casting rules that ensure that a scalar cannot "upcast" an array unless the scalar is of a fundamentally different kind of data (i.e., under a different hierarchy in the data-type hierarchy) than the array. This rule enables you to use scalar constants in your code (which, as Python types, are interpreted accordingly in ufuncs) without worrying about whether the precision of the scalar constant will cause upcasting on your large (small precision) array.

Use of internal buffers

Internally, buffers are used for misaligned data, swapped data, and data that has to be converted from one data type to another. The size of internal buffers is settable on a per-thread basis. There can be up to $2(n_{\mathrm{inputs}} + n_{\mathrm{outputs}})$ buffers of the specified size created to handle the data from all the inputs and outputs of a ufunc. The default size of a buffer is 10,000 elements. Whenever buffer-based calculation would be needed, but all input arrays are smaller than the buffer size, those misbehaved or incorrectly-typed arrays will be copied before the calculation proceeds. Adjusting the size of the buffer may therefore alter the speed at which ufunc calculations of various sorts are completed. A simple interface for setting this variable is accessible using the function **`numpy.setbufsize`**.

Error handling

Universal functions can trip special floating-point status registers in your hardware (such as divide-by-zero). If available on your platform, these registers will be regularly checked during calculation. Error handling is controlled on a per-thread basis, and can be configured using the functions **`numpy.seterr`** and **`numpy.seterrcall`**.

Overriding ufunc behavior

Classes (including ndarray subclasses) can override how ufuncs act on them by defining certain special methods. For details, see Standard array subclasses.

Previous
Subclassing ndarray

Next
Copies and views