Structured arrays

Introduction

Structured arrays are ndarrays whose datatype is a composition of simpler datatypes organized as a sequence of named fields. For example,

```
>>> x = np.array([('Rex', 9, 81.0), ('Fido', 3, 27.0)],
...              dtype=[('name', 'U10'), ('age', 'i4'), ('weight', 'f4')])
>>> x
array([('Rex', 9, 81.), ('Fido', 3, 27.)],
      dtype=[('name', '<U10'), ('age', '<i4'), ('weight', '<f4')])
```

Here x is a one-dimensional array of length two whose datatype is a structure with three fields: 1. A string of length 10 or less named ‘name’, 2. a 32-bit integer named ‘age’, and 3. a 32-bit float named ‘weight’.

If you index x at position 1 you get a structure:

```
>>> x[1]
('Fido', 3, 27.)
```

You can access and modify individual fields of a structured array by indexing with the field name:

```
>>> x['age']
array([9, 3], dtype=int32)
>>> x['age'] = 5
>>> x
array([('Rex', 5, 81.), ('Fido', 5, 27.)],
      dtype=[('name', '<U10'), ('age', '<i4'), ('weight', '<f4')])
```

Structured datatypes are designed to be able to mimic ‘structs’ in the C language, and share a similar memory layout. They are meant for interfacing with C code and for low-level manipulation of structured buffers, for example for interpreting binary blobs. For these purposes they support specialized features such as subarrays, nested datatypes, and unions, and allow control over the memory layout of the structure.

Users looking to manipulate tabular data, such as stored in csv files, may find other pydata projects more suitable, such as xarray, pandas, or DataArray. These provide a high-level interface for tabular data analysis and are better optimized for that use. For instance, the C-struct-like memory layout of structured arrays in numpy can lead to poor cache behavior in comparison.

Structured Datatypes

A structured datatype can be thought of as a sequence of bytes of a certain length (the structure’s itemsize) which is interpreted as a collection of fields. Each field has a name, a datatype, and a byte offset within the structure. The datatype of a field may be any numpy datatype including other structured datatypes, and it may also be a subarray data type which behaves like an ndarray of a specified shape. The offsets of the fields are arbitrary, and fields may even overlap. These offsets are usually determined automatically by numpy, but can also be specified.

Structured Datatype Creation

Structured datatypes may be created using the function **numpy.dtype**. There are 4 alternative forms of specification which vary in flexibility and conciseness. These are further documented in the Data Type Objects reference page, and in summary they are:

1. A list of tuples, one tuple per field

   Each tuple has the form `(fieldname, datatype, shape)` where shape is optional. `fieldname` is a string (or tuple if titles are used, see Field Titles below), `datatype` may be any object convertible to a datatype, and `shape` is a tuple of integers specifying subarray shape.

   ```
   >>> np.dtype([('x', 'f4'), ('y', np.float32), ('z', 'f4', (2, 2))])
   dtype([('x', '<f4'), ('y', '<f4'), ('z', '<f4', (2, 2))])
   ```

   If `fieldname` is the empty string `''`, the field will be given a default name of the form `f#`, where `#` is the integer index of the field, counting from 0 from the left:

   ```
   >>> np.dtype([('x', 'f4'), ('', 'i4'), ('z', 'i8')])
   dtype([('x', '<f4'), ('f1', '<i4'), ('z', '<i8')])
   ```

   The byte offsets of the fields within the structure and the total structure itemsize are determined automatically.

2. A string of comma-separated dtype specifications

   In this shorthand notation any of the string dtype specifications may be used in a string and separated by commas. The itemsize and byte offsets of the fields are determined automatically, and the field names are given the default names `f0`, `f1`, etc.

   ```
   >>> np.dtype('i8, f4, S3')
   dtype([('f0', '<i8'), ('f1', '<f4'), ('f2', 'S3')])
   >>> np.dtype('3int8, float32, (2, 3)float64')
   dtype([('f0', 'i1', (3,)), ('f1', '<f4'), ('f2', '<f8', (2, 3))])
   ```

3. A dictionary of field parameter arrays

   This is the most flexible form of specification since it allows control over the byte-offsets of the fields and the itemsize of the structure.

   The dictionary has two required keys, ‘names’ and ‘formats’, and four optional keys, ‘offsets’, ‘itemsize’, ‘aligned’ and ‘titles’. The values for ‘names’ and ‘formats’ should respectively be a list of field names and a list of dtype specifications, of the same length. The optional ‘offsets’ value should be a list of integer byte-offsets, one for each field within the structure. If ‘offsets’ is not given the offsets are determined automatically. The optional ‘itemsize’ value should be an integer describing the total size in bytes of the dtype, which must be large enough to contain all the fields.

   ```
   >>> np.dtype({'names': ['col1', 'col2'], 'formats': ['i4', 'f4']})
   dtype([('col1', '<i4'), ('col2', '<f4')])
   >>> np.dtype({'names': ['col1', 'col2'],
   ...           'formats': ['i4', 'f4'],
   ...           'offsets': [0, 4],
   ...           'itemsize': 12})
   dtype({'names': ['col1', 'col2'], 'formats': ['<i4', '<f4'], 'offsets': [0, 4],
   'itemsize': 12})
   ```

   Offsets may be chosen such that the fields overlap, though this will mean that assigning to one field may clobber any overlapping field’s data. As an exception, fields of **numpy.object_** type cannot overlap with other fields, because of the risk of clobbering the internal object pointer and then dereferencing it.

   The optional ‘aligned’ value can be set to `True` to make the automatic offset computation use aligned offsets (see Automatic Byte Offsets and Alignment), as if the ‘align’ keyword argument of **numpy.dtype** had been set to True.

The optional ‘titles’ value should be a list of titles of the same length as ‘names’, see Field Titles below.

4. A dictionary of field names

   The keys of the dictionary are the field names and the values are tuples specifying type and offset:

```
>>> np.dtype({'col1': ('i1', 0), 'col2': ('f4', 1)})
dtype([('col1', 'i1'), ('col2', '<f4')])
```

   This form was discouraged because Python dictionaries did not preserve order in Python versions before Python 3.6. Field Titles may be specified by using a 3-tuple, see below.

Manipulating and Displaying Structured Datatypes

The list of field names of a structured datatype can be found in the `names` attribute of the dtype object:

```
>>> d = np.dtype([('x', 'i8'), ('y', 'f4')])
>>> d.names
('x', 'y')
```

The field names may be modified by assigning to the `names` attribute using a sequence of strings of the same length.

The dtype object also has a dictionary-like attribute, `fields`, whose keys are the field names (and Field Titles, see below) and whose values are tuples containing the dtype and byte offset of each field.

```
>>> d.fields
mappingproxy({'x': (dtype('int64'), 0), 'y': (dtype('float32'), 8)})
```

Both the `names` and `fields` attributes will equal `None` for unstructured arrays. The recommended way to test if a dtype is structured is with *if dt.names is not None* rather than *if dt.names*, to account for dtypes with 0 fields.

The string representation of a structured datatype is shown in the “list of tuples” form if possible, otherwise numpy falls back to using the more general dictionary form.

Automatic Byte Offsets and Alignment

Numpy uses one of two methods to automatically determine the field byte offsets and the overall itemsize of a structured datatype, depending on whether `align=True` was specified as a keyword argument to **`numpy.dtype`**.

By default (`align=False`), numpy will pack the fields together such that each field starts at the byte offset the previous field ended, and the fields are contiguous in memory.

```
>>> def print_offsets(d):
...     print("offsets:", [d.fields[name][1] for name in d.names])
...     print("itemsize:", d.itemsize)
>>> print_offsets(np.dtype('u1, u1, i4, u1, i8, u2'))
offsets: [0, 1, 2, 6, 7, 15]
itemsize: 17
```

If `align=True` is set, numpy will pad the structure in the same way many C compilers would pad a C-struct. Aligned structures can give a performance improvement in some cases, at the cost of increased datatype size. Padding bytes are inserted between fields such that each field’s byte offset will be a multiple of that field’s alignment, which is usually equal to the field’s size in bytes for simple datatypes, see **`PyArray_Descr.alignment`**. The structure will also have trailing padding added so that its itemsize is a multiple of the largest field’s alignment.

```
>>> print_offsets(np.dtype('u1, u1, i4, u1, i8, u2', align=True))
offsets: [0, 1, 4, 8, 16, 24]
itemsize: 32
```

Note that although almost all modern C compilers pad in this way by default, padding in C structs is C-implementation-dependent so this memory layout is not guaranteed to exactly match that of a corresponding struct in a C program. Some work may be needed, either on the numpy side or the C side, to obtain exact correspondence.

If offsets were specified using the optional `offsets` key in the dictionary-based dtype specification, setting `align=True` will check that each field's offset is a multiple of its size and that the itemsize is a multiple of the largest field size, and raise an exception if not.

If the offsets of the fields and itemsize of a structured array satisfy the alignment conditions, the array will have the `ALIGNED` **`flag`** set.

A convenience function **`numpy.lib.recfunctions.repack_fields`** converts an aligned dtype or array to a packed one and vice versa. It takes either a dtype or structured ndarray as an argument, and returns a copy with fields re-packed, with or without padding bytes.

Field Titles

In addition to field names, fields may also have an associated title, an alternate name, which is sometimes used as an additional description or alias for the field. The title may be used to index an array, just like a field name.

To add titles when using the list-of-tuples form of dtype specification, the field name may be specified as a tuple of two strings instead of a single string, which will be the field's title and field name respectively. For example:

```
>>> np.dtype([(('my title', 'name'), 'f4')])
dtype([(('my title', 'name'), '<f4')])
```

When using the first form of dictionary-based specification, the titles may be supplied as an extra `'titles'` key as described above. When using the second (discouraged) dictionary-based specification, the title can be supplied by providing a 3-element tuple `(datatype, offset, title)` instead of the usual 2-element tuple:

```
>>> np.dtype({'name': ('i4', 0, 'my title')})
dtype([(('my title', 'name'), '<i4')])
```

The `dtype.fields` dictionary will contain titles as keys, if any titles are used. This means effectively that a field with a title will be represented twice in the fields dictionary. The tuple values for these fields will also have a third element, the field title. Because of this, and because the `names` attribute preserves the field order while the `fields` attribute may not, it is recommended to iterate through the fields of a dtype using the `names` attribute of the dtype, which will not list titles, as in:

```
>>> for name in d.names:
...     print(d.fields[name][:2])
(dtype('int64'), 0)
(dtype('float32'), 8)
```

Union types

Structured datatypes are implemented in numpy to have base type **`numpy.void`** by default, but it is possible to interpret other numpy types as structured types using the `(base_dtype, dtype)` form of dtype specification described in Data Type Objects. Here, `base_dtype` is the desired underlying dtype, and fields and flags will be copied from `dtype`. This dtype is similar to a 'union' in C.

Indexing and Assignment to Structured arrays

Assigning data to a Structured Array

There are a number of ways to assign values to a structured array: Using python tuples, using scalar values, or using other structured arrays.

Assignment from Python Native Types (Tuples)

The simplest way to assign values to a structured array is using python tuples. Each assigned value should be a tuple of length equal to the number of fields in the array, and not a list or array as these will trigger numpy's broadcasting rules. The tuple's elements are assigned to the successive fields of the array, from left to right:

```
>>> x = np.array([(1, 2, 3), (4, 5, 6)], dtype='i8, f4, f8')
>>> x[1] = (7, 8, 9)
>>> x
array([(1, 2., 3.), (7, 8., 9.)],
     dtype=[('f0', '<i8'), ('f1', '<f4'), ('f2', '<f8')])
```

Assignment from Scalars

A scalar assigned to a structured element will be assigned to all fields. This happens when a scalar is assigned to a structured array, or when an unstructured array is assigned to a structured array:

```
>>> x = np.zeros(2, dtype='i8, f4, ?, S1')
>>> x[:] = 3
>>> x
array([(3, 3., True, b'3'), (3, 3., True, b'3')],
      dtype=[('f0', '<i8'), ('f1', '<f4'), ('f2', '?'), ('f3', 'S1')])
>>> x[:] = np.arange(2)
>>> x
array([(0, 0., False, b'0'), (1, 1., True, b'1')],
      dtype=[('f0', '<i8'), ('f1', '<f4'), ('f2', '?'), ('f3', 'S1')])
```

Structured arrays can also be assigned to unstructured arrays, but only if the structured datatype has just a single field:

```
>>> twofield = np.zeros(2, dtype=[('A', 'i4'), ('B', 'i4')])
>>> onefield = np.zeros(2, dtype=[('A', 'i4')])
>>> nostruct = np.zeros(2, dtype='i4')
>>> nostruct[:] = twofield
Traceback (most recent call last):
...
TypeError: Cannot cast array data from dtype([('A', '<i4'), ('B', '<i4')]) to
dtype('int32') according to the rule 'unsafe'
```

Assignment from other Structured Arrays

Assignment between two structured arrays occurs as if the source elements had been converted to tuples and then assigned to the destination elements. That is, the first field of the source array is assigned to the first field of the destination array, and the second field likewise, and so on, regardless of field names. Structured arrays with a different number of fields cannot be assigned to each other. Bytes of the destination structure which are not included in any of the fields are unaffected.

```
>>> a = np.zeros(3, dtype=[('a', 'i8'), ('b', 'f4'), ('c', 'S3')])
>>> b = np.ones(3, dtype=[('x', 'f4'), ('y', 'S3'), ('z', 'O')])
>>> b[:] = a
>>> b
array([(0., b'0.0', b''), (0., b'0.0', b''), (0., b'0.0', b'')],
      dtype=[('x', '<f4'), ('y', 'S3'), ('z', 'O')])
```

Assignment involving subarrays

When assigning to fields which are subarrays, the assigned value will first be broadcast to the shape of the subarray.

Indexing Structured Arrays

Accessing Individual Fields

Individual fields of a structured array may be accessed and modified by indexing the array with the field name.

```
>>> x = np.array([(1, 2), (3, 4)], dtype=[('foo', 'i8'), ('bar', 'f4')])
>>> x['foo']
array([1, 3])
>>> x['foo'] = 10
>>> x
array([(10, 2.), (10, 4.)],
      dtype=[('foo', '<i8'), ('bar', '<f4')])
```

The resulting array is a view into the original array. It shares the same memory locations and writing to the view will modify the original array.

```
>>> y = x['bar']
>>> y[:] = 11
>>> x
array([(10, 11.), (10, 11.)],
      dtype=[('foo', '<i8'), ('bar', '<f4')])
```

This view has the same dtype and itemsize as the indexed field, so it is typically a non-structured array, except in the case of nested structures.

```
>>> y.dtype, y.shape, y.strides
(dtype('float32'), (2,), (12,))
```

If the accessed field is a subarray, the dimensions of the subarray are appended to the shape of the result:

```
>>> x = np.zeros((2, 2), dtype=[('a', np.int32), ('b', np.float64, (3, 3))])
>>> x['a'].shape
(2, 2)
>>> x['b'].shape
(2, 2, 3, 3)
```

Accessing Multiple Fields

One can index and assign to a structured array with a multi-field index, where the index is a list of field names.

> **Warning**
>
> The behavior of multi-field indexes changed from Numpy 1.15 to Numpy 1.16.

The result of indexing with a multi-field index is a view into the original array, as follows:

```
>>> a = np.zeros(3, dtype=[('a', 'i4'), ('b', 'i4'), ('c', 'f4')])
>>> a[['a', 'c']]
array([(0, 0.), (0, 0.), (0, 0.)],
     dtype={'names': ['a', 'c'], 'formats': ['<i4', '<f4'], 'offsets': [0, 8],
'itemsize': 12})
```

Assignment to the view modifies the original array. The view's fields will be in the order they were indexed. Note that unlike for single-field indexing, the dtype of the view has the same itemsize as the original array, and has fields at the same offsets as in the original array, and unindexed fields are merely missing.

Warning

In Numpy 1.15, indexing an array with a multi-field index returned a copy of the result above, but with fields packed together in memory as if passed through [numpy.lib.recfunctions.repack_fields](#).

The new behavior as of Numpy 1.16 leads to extra “padding” bytes at the location of unindexed fields compared to 1.15. You will need to update any code which depends on the data having a “packed” layout. For instance code such as:

```
>>> a[['a', 'c']].view('i8')  # Fails in Numpy 1.16
Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
ValueError: When changing to a smaller dtype, its size must be a divisor of
the size of original dtype
```

will need to be changed. This code has raised a `FutureWarning` since Numpy 1.12, and similar code has raised `FutureWarning` since 1.7.

In 1.16 a number of functions have been introduced in the [numpy.lib.recfunctions](#) module to help users account for this change. These are [numpy.lib.recfunctions.repack_fields](#). [numpy.lib.recfunctions.structured_to_unstructured](#), [numpy.lib.recfunctions.unstructured_to_structured](#), [numpy.lib.recfunctions.apply_along_fields](#), [numpy.lib.recfunctions.assign_fields_by_name](#), and [numpy.lib.recfunctions.require_fields](#).

The function [numpy.lib.recfunctions.repack_fields](#) can always be used to reproduce the old behavior, as it will return a packed copy of the structured array. The code above, for example, can be replaced with:

```
>>> from numpy.lib.recfunctions import repack_fields
>>> repack_fields(a[['a', 'c']]).view('i8')  # supported in 1.16
array([0, 0, 0])
```

Furthermore, numpy now provides a new function [numpy.lib.recfunctions.structured_to_unstructured](#) which is a safer and more efficient alternative for users who wish to convert structured arrays to unstructured arrays, as the view above is often intended to do. This function allows safe conversion to an unstructured type taking into account padding, often avoids a copy, and also casts the datatypes as needed, unlike the view. Code such as:

```
>>> b = np.zeros(3, dtype=[('x', 'f4'), ('y', 'f4'), ('z', 'f4')])
>>> b[['x', 'z']].view('f4')
array([0., 0., 0., 0., 0., 0., 0., 0., 0.], dtype=float32)
```

can be made safer by replacing with:

```
>>> from numpy.lib.recfunctions import structured_to_unstructured
>>> structured_to_unstructured(b[['x', 'z']])
array([[0., 0.],
       [0., 0.],
       [0., 0.]], dtype=float32)
```

Assignment to an array with a multi-field index modifies the original array:

```
>>> a[['a', 'c']] = (2, 3)
>>> a
array([(2, 0, 3.), (2, 0, 3.), (2, 0, 3.)],
      dtype=[('a', '<i4'), ('b', '<i4'), ('c', '<f4')])
```

This obeys the structured array assignment rules described above. For example, this means that one can swap the values of two fields using appropriate multi-field indexes:

```
>>> a[['a', 'c']] = a[['c', 'a']]
```

Indexing with an Integer to get a Structured Scalar

Indexing a single element of a structured array (with an integer index) returns a structured scalar:

```
>>> x = np.array([(1, 2., 3.)], dtype='i, f, f')
>>> scalar = x[0]
>>> scalar
(1, 2., 3.)
>>> type(scalar)
<class 'numpy.void'>
```

Unlike other numpy scalars, structured scalars are mutable and act like views into the original array, such that modifying the scalar will modify the original array. Structured scalars also support access and assignment by field name:

```
>>> x = np.array([(1, 2), (3, 4)], dtype=[('foo', 'i8'), ('bar', 'f4')])
>>> s = x[0]
>>> s['bar'] = 100
>>> x
array([(1, 100.), (3, 4.)],
      dtype=[('foo', '<i8'), ('bar', '<f4')])
```

Similarly to tuples, structured scalars can also be indexed with an integer:

```
>>> scalar = np.array([(1, 2., 3.)], dtype='i, f, f')[0]
>>> scalar[0]
1
>>> scalar[1] = 4
```

Thus, tuples might be thought of as the native Python equivalent to numpy's structured types, much like native python integers are the equivalent to numpy's integer types. Structured scalars may be converted to a tuple by calling **numpy.ndarray.item**:

```
>>> scalar.item(), type(scalar.item())
((1, 4.0, 3.0), <class 'tuple'>)
```

Viewing Structured Arrays Containing Objects

In order to prevent clobbering object pointers in fields of **object** type, numpy currently does not allow views of structured arrays containing objects.

Structure Comparison and Promotion

If the dtypes of two void structured arrays are equal, testing the equality of the arrays will result in a boolean array with the dimensions of the original arrays, with elements set to `True` where all fields of the corresponding structures are equal:

```
>>> a = np.array([(1, 1), (2, 2)], dtype=[('a', 'i4'), ('b', 'i4')])
>>> b = np.array([(1, 1), (2, 3)], dtype=[('a', 'i4'), ('b', 'i4')])
>>> a == b
array([True, False])
```

NumPy will promote individual field datatypes to perform the comparison. So the following is also valid (note the `'f4'` dtype for the `'a'` field):

```
>>> b = np.array([(1.0, 1), (2.5, 2)], dtype=[("a", "f4"), ("b", "i4")])
>>> a == b
array([True, False])
```

To compare two structured arrays, it must be possible to promote them to a common dtype as returned by **numpy.result_type** and *np.promote_types*. This enforces that the number of fields, the field names, and the field titles must match precisely. When promotion is not possible, for example due to mismatching field names, NumPy will raise an error. Promotion between two structured dtypes results in a canonical dtype that ensures native byte-order for all fields:

```
>>> np.result_type(np.dtype("i,>i"))
dtype([('f0', '<i4'), ('f1', '<i4')])
>>> np.result_type(np.dtype("i,>i"), np.dtype("i,i"))
dtype([('f0', '<i4'), ('f1', '<i4')])
```

The resulting dtype from promotion is also guaranteed to be packed, meaning that all fields are ordered contiguously and any unnecessary padding is removed:

```
>>> dt = np.dtype("i1,V3,i4,V1")[["f0", "f2"]]
>>> dt
dtype({'names':['f0','f2'], 'formats':['i1','<i4'], 'offsets':[0,4], 'itemsize':9})
>>> np.result_type(dt)
dtype([('f0', 'i1'), ('f2', '<i4')])
```

Note that the result prints without `offsets` or `itemsize` indicating no additional padding. If a structured dtype is created with `align=True` ensuring that `dtype.isalignedstruct` is true, this property is preserved:

```
>>> dt = np.dtype("i1,V3,i4,V1", align=True)[["f0", "f2"]]
>>> dt
dtype({'names':['f0','f2'], 'formats':['i1','<i4'], 'offsets':[0,4], 'itemsize':12},
align=True)
>>> np.result_type(dt)
dtype([('f0', 'i1'), ('f2', '<i4')], align=True)
>>> np.result_type(dt).isalignedstruct
True
```

When promoting multiple dtypes, the result is aligned if any of the inputs is:

```
>>> np.result_type(np.dtype("i,i"), np.dtype("i,i", align=True))
dtype([('f0', '<i4'), ('f1', '<i4')], align=True)
```

The `<` and `>` operators always return `False` when comparing void structured arrays, and arithmetic and bitwise operations are not supported.

> ***Changed in version 1.23:*** Before NumPy 1.23, a warning was given and `False` returned when promotion to a common dtype failed. Further, promotion was much more restrictive: It would reject the mixed float/integer comparison example above.

Record Arrays

As an optional convenience numpy provides an ndarray subclass, **numpy.recarray** that allows access to fields of structured arrays by attribute instead of only by index. Record arrays use a special datatype, **numpy.record**, that allows field access by attribute on the structured scalars obtained from the array. The `numpy.rec` module provides functions for creating recarrays from various objects. Additional helper functions for creating and manipulating structured arrays can be found in **numpy.lib.recfunctions**.

The simplest way to create a record array is with **numpy.rec.array**:

```
>>> recordarr = np.rec.array([(1, 2., 'Hello'), (2, 3., "World")],
...                          dtype=[('foo', 'i4'),('bar', 'f4'), ('baz', 'S10')])
>>> recordarr.bar
array([2., 3.], dtype=float32)
>>> recordarr[1:2]
rec.array([(2, 3., b'World')],
      dtype=[('foo', '<i4'), ('bar', '<f4'), ('baz', 'S10')])
>>> recordarr[1:2].foo
array([2], dtype=int32)
>>> recordarr.foo[1:2]
array([2], dtype=int32)
>>> recordarr[1].baz
b'World'
```

numpy.rec.array can convert a wide variety of arguments into record arrays, including structured arrays:

```
>>> arr = np.array([(1, 2., 'Hello'), (2, 3., "World")],
...                dtype=[('foo', 'i4'), ('bar', 'f4'), ('baz', 'S10')])
>>> recordarr = np.rec.array(arr)
```

The `numpy.rec` module provides a number of other convenience functions for creating record arrays, see record array creation routines.

A record array representation of a structured array can be obtained using the appropriate view:

```
>>> arr = np.array([(1, 2., 'Hello'), (2, 3., "World")],
...                dtype=[('foo', 'i4'),('bar', 'f4'), ('baz', 'a10')])
>>> recordarr = arr.view(dtype=np.dtype((np.record, arr.dtype)),
...                      type=np.recarray)
```

For convenience, viewing an ndarray as type **numpy.recarray** will automatically convert to **numpy.record** datatype, so the dtype can be left out of the view:

```
>>> recordarr = arr.view(np.recarray)
>>> recordarr.dtype
dtype((numpy.record, [('foo', '<i4'), ('bar', '<f4'), ('baz', 'S10')]))
```

To get back to a plain ndarray both the dtype and type must be reset. The following view does so, taking into account the unusual case that the recordarr was not a structured type:

```
>>> arr2 = recordarr.view(recordarr.dtype.fields or recordarr.dtype, np.ndarray)
```

Record array fields accessed by index or by attribute are returned as a record array if the field has a structured type but as a plain ndarray otherwise.

```
>>> recordarr = np.rec.array([('Hello', (1, 2)), ("World", (3, 4))],
...                          dtype=[('foo', 'S6'),('bar', [('A', int), ('B', int)])])
>>> type(recordarr.foo)
<class 'numpy.ndarray'>
>>> type(recordarr.bar)
<class 'numpy.recarray'>
```

Note that if a field has the same name as an ndarray attribute, the ndarray attribute takes precedence. Such fields will be inaccessible by attribute but will still be accessible by index.

Recarray Helper Functions

Collection of utilities to manipulate structured arrays.

Most of these functions were initially implemented by John Hunter for matplotlib. They have been rewritten and extended for convenience.

numpy.lib.recfunctions.append_fields(*base, names, data, dtypes=None, fill_value=- 1, usemask=True, asrecarray=False*) [source]

Add new fields to an existing array.

The names of the fields are given with the *names* arguments, the corresponding values with the *data* arguments. If a single field is appended, *names*, *data* and *dtypes* do not have to be lists but just values.

Parameters:

base : *array*
Input array to extend.

names : *string, sequence*
String or sequence of strings corresponding to the names of the new fields.

data : *array or sequence of arrays*
Array or sequence of arrays storing the fields to add to the base.

dtypes : *sequence of datatypes, optional*
Datatype or sequence of datatypes. If None, the datatypes are estimated from the *data*.

fill_value : *{float}, optional*
Filling value used to pad missing data on the shorter arrays.

usemask : *{False, True}, optional*
Whether to return a masked array or not.

asrecarray : *{False, True}, optional*
Whether to return a recarray (MaskedRecords) or not.

numpy.lib.recfunctions.apply_along_fields(*func, arr*) [source]

Apply function ‘func’ as a reduction across fields of a structured array.

This is similar to *apply_along_axis*, but treats the fields of a structured array as an extra axis. The fields are all first cast to a common type following the type-promotion rules from **numpy.result_type** applied to the field’s dtypes.

Parameters:

func : *function*
Function to apply on the “field” dimension. This function must support an *axis* argument, like np.mean, np.sum, etc.

arr : *ndarray*
Structured array for which to apply func.

Returns:

out : *ndarray*
Result of the recution operation

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> b = np.array([(1, 2, 5), (4, 5, 7), (7, 8 ,11), (10, 11, 12)],
...              dtype=[('x', 'i4'), ('y', 'f4'), ('z', 'f8')])
>>> rfn.apply_along_fields(np.mean, b)
array([ 2.66666667,  5.33333333,  8.66666667, 11.        ])
>>> rfn.apply_along_fields(np.mean, b[['x', 'z']])
array([ 3. ,  5.5,  9. , 11. ])
```

numpy.lib.recfunctions.assign_fields_by_name(*dst, src, zero_unassigned=True*) [source]

Assigns values from one structured array to another by field name.

Normally in numpy >= 1.14, assignment of one structured array to another copies fields "by position", meaning that the first field from the src is copied to the first field of the dst, and so on, regardless of field name.

This function instead copies "by field name", such that fields in the dst are assigned from the identically named field in the src. This applies recursively for nested structures. This is how structure assignment worked in numpy >= 1.6 to <= 1.13.

Parameters:

dst : *ndarray*

src : *ndarray*

The source and destination arrays during assignment.

zero_unassigned : *bool, optional*

If True, fields in the dst for which there was no matching field in the src are filled with the value 0 (zero). This was the behavior of numpy <= 1.13. If False, those fields are not modified.

numpy.lib.recfunctions.drop_fields(*base, drop_names, usemask=True, asrecarray=False*) [source]

Return a new array with fields in *drop_names* dropped.

Nested fields are supported.

Changed in version 1.18.0: drop_fields returns an array with 0 fields if all fields are dropped, rather than returning None as it did previously.

Parameters:

base : *array*

Input array

drop_names : *string or sequence*

String or sequence of strings corresponding to the names of the fields to drop.

usemask : *{False, True}, optional*

Whether to return a masked array or not.

asrecarray : *string or sequence, optional*

Whether to return a recarray or a mrecarray (*asrecarray=True*) or a plain ndarray or masked array with flexible dtype. The default is False.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> a = np.array([(1, (2, 3.0)), (4, (5, 6.0))],
...   dtype=[('a', np.int64), ('b', [('ba', np.double), ('bb', np.int64)])])
>>> rfn.drop_fields(a, 'a')
array([((2., 3),), ((5., 6),)],
      dtype=[('b', [('ba', '<f8'), ('bb', '<i8')])])
>>> rfn.drop_fields(a, 'ba')
array([(1, (3,)), (4, (6,))], dtype=[('a', '<i8'), ('b', [('bb', '<i8')])])
>>> rfn.drop_fields(a, ['ba', 'bb'])
array([(1,), (4,)], dtype=[('a', '<i8')])
```

numpy.lib.recfunctions.find_duplicates(*a, key=None, ignoremask=True, return_index=False*) [source]

Find the duplicates in a structured array along a given key

Parameters:

a : *array-like*

Input array

key : *{string, None}, optional*

Name of the fields along which to check the duplicates. If None, the search is performed by records

ignoremask : *{True, False}, optional*

Whether masked data should be discarded or considered as duplicates.

return_index : *{False, True}, optional*

Whether to return the indices of the duplicated values.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> ndtype = [('a', int)]
>>> a = np.ma.array([1, 1, 1, 2, 2, 3, 3],
...         mask=[0, 0, 1, 0, 0, 0, 1]).view(ndtype)
>>> rfn.find_duplicates(a, ignoremask=True, return_index=True)
(masked_array(data=[(1,), (1,), (2,), (2,)],
             mask=[(False,), (False,), (False,), (False,)],
       fill_value=(999999,),
            dtype=[('a', '<i8')]), array([0, 1, 3, 4]))
```

numpy.lib.recfunctions.flatten_descr(*ndtype*) [source]

Flatten a structured data-type description.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> ndtype = np.dtype([('a', '<i4'), ('b', [('ba', '<f8'), ('bb', '<i4')])])
>>> rfn.flatten_descr(ndtype)
(('a', dtype('int32')), ('ba', dtype('float64')), ('bb', dtype('int32')))
```

numpy.lib.recfunctions.get_fieldstructure(*adtype*, *lastname=None*, *parents=None*) [source]

Returns a dictionary with fields indexing lists of their parent fields.

This function is used to simplify access to fields nested in other fields.

Parameters: adtype : *np.dtype*

Input datatype

lastname : *optional*

Last processed field name (used internally during recursion).

parents : *dictionary*

Dictionary of parent fields (used interbally during recursion).

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> ndtype =  np.dtype([('A', int),
...                     ('B', [('BA', int),
...                            ('BB', [('BBA', int), ('BBB', int)])])])
>>> rfn.get_fieldstructure(ndtype)
... # XXX: possible regression, order of BBA and BBB is swapped
{'A': [], 'B': [], 'BA': ['B'], 'BB': ['B'], 'BBA': ['B', 'BB'], 'BBB': ['B',
'BB']}
```

numpy.lib.recfunctions.get_names(*adtype*) [source]

Returns the field names of the input datatype as a tuple. Input datatype must have fields otherwise error is raised.

Parameters: adtype : *dtype*

Input datatype

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> rfn.get_names(np.empty((1,), dtype=[('A', int)]).dtype)
('A',)
>>> rfn.get_names(np.empty((1,), dtype=[('A',int), ('B', float)]).dtype)
('A', 'B')
>>> adtype = np.dtype([('a', int), ('b', [('ba', int), ('bb', int)])])
>>> rfn.get_names(adtype)
('a', ('b', ('ba', 'bb')))
```

numpy.lib.recfunctions.**get_names_flat**(*adtype*) [source]

Returns the field names of the input datatype as a tuple. Input datatype must have fields otherwise error is raised. Nested structure are flattened beforehand.

Parameters: adtype : *dtype*
Input datatype

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> rfn.get_names_flat(np.empty((1,), dtype=[('A', int)]).dtype) is None
False
>>> rfn.get_names_flat(np.empty((1,), dtype=[('A',int), ('B', str)]).dtype)
('A', 'B')
>>> adtype = np.dtype([('a', int), ('b', [('ba', int), ('bb', int)])])
>>> rfn.get_names_flat(adtype)
('a', 'b', 'ba', 'bb')
```

numpy.lib.recfunctions.**join_by**(*key, r1, r2, jointype='inner', r1postfix='1', r2postfix='2', defaults=None, usemask=True, asrecarray=False*) [source]

Join arrays *r1* and *r2* on key *key*.

The key should be either a string or a sequence of string corresponding to the fields used to join the array. An exception is raised if the *key* field cannot be found in the two input arrays. Neither *r1* nor *r2* should have any duplicates along *key*: the presence of duplicates will make the output quite unreliable. Note that duplicates are not looked for by the algorithm.

Parameters: key : *{string, sequence}*
A string or a sequence of strings corresponding to the fields used for comparison.

r1, r2 : *arrays*
Structured arrays.

jointype : *{‘inner’, ‘outer’, ‘leftouter’}, optional*
If ‘inner’, returns the elements common to both r1 and r2. If ‘outer’, returns the common elements as well as the elements of r1 not in r2 and the elements of not in r2. If ‘leftouter’, returns the common elements and the elements of r1 not in r2.

r1postfix : *string, optional*
String appended to the names of the fields of r1 that are present in r2 but absent of the key.

r2postfix : *string, optional*
String appended to the names of the fields of r2 that are present in r1 but absent of the key.

defaults : *{dictionary}, optional*
Dictionary mapping field names to the corresponding default values.

usemask : *{True, False}, optional*
Whether to return a MaskedArray (or MaskedRecords is *asrecarray*==*True*) or a ndarray.

asrecarray : *{False, True}, optional*

Whether to return a recarray (or MaskedRecords if *usemask*==*True*) or just a flexible-type ndarray.

Notes

- The output is sorted along the key.
- A temporary array is formed by dropping the fields not in the key for the two arrays and concatenating the result. This array is then sorted, and the common entries selected. The output is constructed by filling the fields with the selected entries. Matching is not preserved if there are some duplicates...

numpy.lib.recfunctions.merge_arrays(*seqarrays, fill_value=- 1, flatten=False, usemask=False, asrecarray=False*) [source]

Merge arrays field by field.

Parameters: seqarrays : *sequence of ndarrays*

Sequence of arrays

fill_value : *{float}, optional*

Filling value used to pad missing data on the shorter arrays.

flatten : *{False, True}, optional*

Whether to collapse nested fields.

usemask : *{False, True}, optional*

Whether to return a masked array or not.

asrecarray : *{False, True}, optional*

Whether to return a recarray (MaskedRecords) or not.

Notes

- Without a mask, the missing value will be filled with something, depending on what its corresponding type:
  - `-1` for integers
  - `-1.0` for floating point numbers
  - `'-'` for characters
  - `'-1'` for strings
  - `True` for boolean values
- XXX: I just obtained these values empirically

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> rfn.merge_arrays((np.array([1, 2]), np.array([10., 20., 30.])))
array([( 1, 10.), ( 2, 20.), (-1, 30.)],
      dtype=[('f0', '<i8'), ('f1', '<f8')])
```

```
>>> rfn.merge_arrays((np.array([1, 2], dtype=np.int64),
...         np.array([10., 20., 30.])), usemask=False)
 array([(1, 10.0), (2, 20.0), (-1, 30.0)],
        dtype=[('f0', '<i8'), ('f1', '<f8')])
>>> rfn.merge_arrays((np.array([1, 2]).view([('a', np.int64)]),
...               np.array([10., 20., 30.])),
...              usemask=False, asrecarray=True)
rec.array([( 1, 10.), ( 2, 20.), (-1, 30.)],
          dtype=[('a', '<i8'), ('f1', '<f8')])
```

numpy.lib.recfunctions.rec_append_fields(*base, names, data, dtypes=None*) [source]

Add new fields to an existing array.

The names of the fields are given with the *names* arguments, the corresponding values with the *data* arguments. If a single field is appended, *names*, *data* and *dtypes* do not have to be lists but just values.

Parameters:

base : *array*

Input array to extend.

names : *string, sequence*

String or sequence of strings corresponding to the names of the new fields.

data : *array or sequence of arrays*

Array or sequence of arrays storing the fields to add to the base.

dtypes : *sequence of datatypes, optional*

Datatype or sequence of datatypes. If None, the datatypes are estimated from the *data*.

Returns:

appended_array : *np.recarray*

See also

`append_fields`

`numpy.lib.recfunctions.rec_drop_fields(base, drop_names)` [source]

Returns a new numpy.recarray with fields in *drop_names* dropped.

`numpy.lib.recfunctions.rec_join(key, r1, r2, jointype='inner', r1postfix='1', r2postfix='2', defaults=None)` [source]

Join arrays *r1* and *r2* on keys. Alternative to join_by, that always returns a np.recarray.

See also

`join_by`
equivalent function

`numpy.lib.recfunctions.recursive_fill_fields(input, output)` [source]

Fills fields from output with fields from input, with support for nested structures.

Parameters:

input : *ndarray*

Input array.

output : *ndarray*

Output array.

Notes

- *output* should be at least the same size as *input*

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> a = np.array([(1, 10.), (2, 20.)], dtype=[('A', np.int64), ('B',
np.float64)])
>>> b = np.zeros((3,), dtype=a.dtype)
>>> rfn.recursive_fill_fields(a, b)
array([(1, 10.), (2, 20.), (0,  0.)], dtype=[('A', '<i8'), ('B', '<f8')])
```

`numpy.lib.recfunctions.rename_fields(base, namemapper)` [source]

Rename the fields from a flexible-datatype ndarray or recarray.

Nested fields are supported.

Parameters:

base : *ndarray*

Input array whose fields must be modified.

namemapper : *dictionary*

Dictionary mapping old field names to their new version.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> a = np.array([(1, (2, [3.0, 30.])), (4, (5, [6.0, 60.]))],
...   dtype=[('a', int),('b', [('ba', float), ('bb', (float, 2))])])
>>> rfn.rename_fields(a, {'a':'A', 'bb':'BB'})
array([(1, (2., [ 3., 30.])), (4, (5., [ 6., 60.]))],
      dtype=[('A', '<i8'), ('b', [('ba', '<f8'), ('BB', '<f8', (2,))])])
```

numpy.lib.recfunctions.repack_fields(*a*, *align=False*, *recurse=False*) [source]

Re-pack the fields of a structured array or dtype in memory.

The memory layout of structured datatypes allows fields at arbitrary byte offsets. This means the fields can be separated by padding bytes, their offsets can be non-monotonically increasing, and they can overlap.

This method removes any overlaps and reorders the fields in memory so they have increasing byte offsets, and adds or removes padding bytes depending on the *align* option, which behaves like the *align* option to **numpy.dtype**.

If *align=False*, this method produces a “packed” memory layout in which each field starts at the byte the previous field ended, and any padding bytes are removed.

If *align=True*, this methods produces an “aligned” memory layout in which each field’s offset is a multiple of its alignment, and the total itemsize is a multiple of the largest alignment, by adding padding bytes as needed.

Parameters:

a : *ndarray or dtype*

array or dtype for which to repack the fields.

align : *boolean*

If true, use an “aligned” memory layout, otherwise use a “packed” layout.

recurse : *boolean*

If True, also repack nested structures.

Returns:

repacked : *ndarray or dtype*

Copy of *a* with fields repacked, or *a* itself if no repacking was needed.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> def print_offsets(d):
...     print("offsets:", [d.fields[name][1] for name in d.names])
...     print("itemsize:", d.itemsize)
...
>>> dt = np.dtype('u1, <i8, <f8', align=True)
>>> dt
dtype({'names': ['f0', 'f1', 'f2'], 'formats': ['u1', '<i8', '<f8'], 'offsets':
[0, 8, 16], 'itemsize': 24}, align=True)
>>> print_offsets(dt)
offsets: [0, 8, 16]
itemsize: 24
>>> packed_dt = rfn.repack_fields(dt)
>>> packed_dt
dtype([('f0', 'u1'), ('f1', '<i8'), ('f2', '<f8')])
>>> print_offsets(packed_dt)
offsets: [0, 1, 9]
itemsize: 17
```

numpy.lib.recfunctions.require_fields(*array*, *required_dtype*) [source]

Casts a structured array to a new dtype using assignment by field-name.

This function assigns from the old to the new array by name, so the value of a field in the output array is the value of the field with the same name in the source array. This has the effect of creating a new ndarray containing only the fields “required” by the required_dtype.

If a field name in the required_dtype does not exist in the input array, that field is created and set to 0 in the output array.

Parameters: **a** : *ndarray*
array to cast

required_dtype : *dtype*
datatype for output array

Returns: **out** : *ndarray*
array with the new dtype, with field values copied from the fields in the input array with the same name

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> a = np.ones(4, dtype=[('a', 'i4'), ('b', 'f8'), ('c', 'u1')])
>>> rfn.require_fields(a, [('b', 'f4'), ('c', 'u1')])
array([(1., 1), (1., 1), (1., 1), (1., 1)],
  dtype=[('b', '<f4'), ('c', 'u1')])
>>> rfn.require_fields(a, [('b', 'f4'), ('newf', 'u1')])
array([(1., 0), (1., 0), (1., 0), (1., 0)],
  dtype=[('b', '<f4'), ('newf', 'u1')])
```

numpy.lib.recfunctions.stack_arrays(*arrays*, *defaults=None*, *usemask=True*, *asrecarray=False*, *autoconvert=False*) [source]

Superposes arrays fields by fields

Parameters: **arrays** : *array or sequence*
Sequence of input arrays.

defaults : *dictionary, optional*
Dictionary mapping field names to the corresponding default values.

usemask : *{True, False}, optional*
Whether to return a MaskedArray (or MaskedRecords is *asrecarray==True*) or a ndarray.

asrecarray : *{False, True}, optional*

Whether to return a recarray (or MaskedRecords if *usemask*==*True*) or just a flexible-type ndarray.

autoconvert : *{False, True}, optional*

Whether automatically cast the type of the field to the maximum.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> x = np.array([1, 2,])
>>> rfn.stack_arrays(x) is x
True
>>> z = np.array([('A', 1), ('B', 2)], dtype=[('A', '|S3'), ('B', float)])
>>> zz = np.array([('a', 10., 100.), ('b', 20., 200.), ('c', 30., 300.)],
...   dtype=[('A', '|S3'), ('B', np.double), ('C', np.double)])
>>> test = rfn.stack_arrays((z,zz))
>>> test
masked_array(data=[(b'A', 1.0, --), (b'B', 2.0, --), (b'a', 10.0, 100.0),
                   (b'b', 20.0, 200.0), (b'c', 30.0, 300.0)],
             mask=[(False, False,  True), (False, False,  True),
                   (False, False, False), (False, False, False),
                   (False, False, False)],
       fill_value=(b'N/A', 1.e+20, 1.e+20),
            dtype=[('A', 'S3'), ('B', '<f8'), ('C', '<f8')])
```

numpy.lib.recfunctions.structured_to_unstructured(*arr*, *dtype=None*, *copy=False*, *casting='unsafe'*) [source]

Converts an n-D structured array into an (n+1)-D unstructured array.

The new array will have a new last dimension equal in size to the number of field-elements of the input array. If not supplied, the output datatype is determined from the numpy type promotion rules applied to all the field datatypes.

Nested fields, as well as each element of any subarray fields, all count as a single field-elements.

Parameters: arr : *ndarray*

Structured array or dtype to convert. Cannot contain object datatype.

dtype : *dtype, optional*

The dtype of the output unstructured array.

copy : *bool, optional*

See copy argument to **numpy.ndarray.astype**. If true, always return a copy. If false, and *dtype* requirements are satisfied, a view is returned.

casting : *{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional*

See casting argument of **numpy.ndarray.astype**. Controls what kind of data casting may occur.

Returns: unstructured : *ndarray*

Unstructured array with one more dimension.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> a = np.zeros(4, dtype=[('a', 'i4'), ('b', 'f4,u2'), ('c', 'f4', 2)])
>>> a
array([(0, (0., 0), [0., 0.]), (0, (0., 0), [0., 0.]),
       (0, (0., 0), [0., 0.]), (0, (0., 0), [0., 0.])],
      dtype=[('a', '<i4'), ('b', [('f0', '<f4'), ('f1', '<u2')]), ('c', '<f4',
(2,))])
>>> rfn.structured_to_unstructured(a)
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

```
>>> b = np.array([(1, 2, 5), (4, 5, 7), (7, 8 ,11), (10, 11, 12)],
...              dtype=[('x', 'i4'), ('y', 'f4'), ('z', 'f8')])
>>> np.mean(rfn.structured_to_unstructured(b[['x', 'z']]), axis=-1)
array([ 3. ,  5.5,  9. , 11. ])
```

numpy.lib.recfunctions.**unstructured_to_structured**(*arr*, *dtype=None*, *names=None*, *align=False*, *copy=False*, *casting='unsafe'*) [source]

Converts an n-D unstructured array into an (n-1)-D structured array.

The last dimension of the input array is converted into a structure, with number of field-elements equal to the size of the last dimension of the input array. By default all output fields have the input array's dtype, but an output structured dtype with an equal number of fields-elements can be supplied instead.

Nested fields, as well as each element of any subarray fields, all count towards the number of field-elements.

Parameters:

arr : *ndarray*
Unstructured array or dtype to convert.

dtype : *dtype, optional*
The structured dtype of the output array

names : *list of strings, optional*
If dtype is not supplied, this specifies the field names for the output dtype, in order. The field dtypes will be the same as the input array.

align : *boolean, optional*
Whether to create an aligned memory layout.

copy : *bool, optional*
See copy argument to **numpy.ndarray.astype**. If true, always return a copy. If false, and *dtype* requirements are satisfied, a view is returned.

casting : *{'no', 'equiv', 'safe', 'same_kind', 'unsafe'}, optional*
See casting argument of **numpy.ndarray.astype**. Controls what kind of data casting may occur.

Returns:

structured : *ndarray*
Structured array with fewer dimensions.

Examples

```
>>> from numpy.lib import recfunctions as rfn
>>> dt = np.dtype([('a', 'i4'), ('b', 'f4,u2'), ('c', 'f4', 2)])
>>> a = np.arange(20).reshape((4,5))
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> rfn.unstructured_to_structured(a, dt)
array([( 0, ( 1.,  2), [ 3.,  4.]), ( 5, ( 6.,  7), [ 8.,  9.]),
       (10, (11., 12), [13., 14.]), (15, (16., 17), [18., 19.])],
      dtype=[('a', '<i4'), ('b', [('f0', '<f4'), ('f1', '<u2')]), ('c', '<f4',
(2,))])
```

Previous
Byte-swapping

Next
Writing custom array containers