

Structured Data: NumPy's Structured Arrays

While **often** our data can be well **represented by a homogeneous array** of values, **sometimes this is not the case.**

This chapter demonstrates the use of NumPy's **structured arrays** and **record arrays**, which provide **efficient storage for compound, heterogeneous data.**

While the **patterns shown here are useful for simple operations**, scenarios like this often lend themselves to the use of **Pandas DataFrame**s, which we'll explore in **Part 3.**

```
In [1]: import numpy as np
```

Imagine that we have several **categories of data** on a number of people (say, name, age, and weight), and we'd like to **store these values** for use in a Python program.

It would be possible to **store** these in **three separate arrays**:

```
In [7]: name = ['Alice', 'Bob', 'Cathy', 'Doug']  
age = [25, 45, 37, 19]  
weight = [55.0, 85.5, 68.0, 61.5]
```

But this is a bit **clumsy**.

There's **nothing here that tells** us that the **three arrays are related**.

NumPy's structured arrays allow us to do this more naturally by using a **single structure** to store all of this data.

Recall that previously we created a **simple array** using an expression like this:

```
In [4]: x = np.zeros(4, dtype=int)
```

```
In [5]: x
```

```
Out[5]: array([0, 0, 0, 0])
```

We can similarly create a **structured array** using a **compound data type specification**:

```
In [6]: # Use a compound data type for structured arrays  
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),  
                           'formats':('U10', 'i4', 'f8')})
```

```
In [3]: data
```

```
Out[3]: array([(' ', 0, 0.), (' ', 0, 0.), (' ', 0, 0.), (' ', 0, 0.)],
              dtype=[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

Here:

- `'U10'` translates to **"Unicode string of maximum length 10,"**
- `'i4'` translates to **"4-byte (i.e., 32-bit) integer,"**
- and `'f8'` translates to **"8-byte (i.e., 64-bit) float."**

Now that we've created an **empty container array**, we can **fill the array with our lists of values**:

```
In [8]: data['name'] = name
        data['age'] = age
        data['weight'] = weight
        print(data)
```

```
[('Alice', 25, 55. ) ('Bob', 45, 85.5) ('Cathy', 37, 68. )  
 ('Doug', 19, 61.5)]
```

As we had hoped, the data is now **conveniently arranged** in **one structured array**.

The handy thing with structured arrays is that we can now **refer to values either by index or by name**:

```
In [ ]: # Get all names  
data['name']
```

```
Out[ ]: array(['Alice', 'Bob', 'Cathy', 'Doug'], dtype='<U10')
```

```
In [9]: # Get first row of data  
data[-1]
```

```
Out[9]: ('Doug', 19, 61.5)
```

```
In [10]: # Get the name from the last row
data[-1]['name']
```

```
Out[10]: 'Doug'
```

Using **Boolean masking**, we can even do some more **sophisticated operations**, such as filtering on age:

```
In [11]: # Recall:
data['age'] < 30
```

```
Out[11]: array([ True, False, False,  True])
```

```
In [12]: data[data['age'] < 30]
```

```
Out[12]: array([('Alice', 25, 55. ), ('Doug', 19, 61.5)],
              dtype=[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

```
In [ ]: # Get names where age is under 30
data[data['age'] < 30]['name']
```

```
Out[ ]: array(['Alice', 'Doug'], dtype='<U10')
```

If you'd like to do **any operations that are any more complicated than these**, you should probably **consider the Pandas package**, covered in **Part 4**.

As you'll see, **Pandas** provides a **DataFrame** object.

Which is **a structure built on NumPy arrays** that offers a **variety of useful data manipulation functionality** similar to what you've seen here, as well as much, much more.

Exploring Structured Array Creation

Structured array data types can be **specified in a number of ways**.

Earlier, **we saw the dictionary method**:

```
In [ ]: np.dtype({'names':('name', 'age', 'weight'),  
                  'formats':('U10', 'i4', 'f8')})
```

```
Out[ ]: dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f8')])
```

For clarity, **numerical types** can be specified using **Python types or NumPy dtypes** instead:

```
In [ ]: np.dtype({'names':('name', 'age', 'weight'),  
                  'formats':((np.str_, 10), int, np.float32)})
```



```
Out[ ]: dtype([('name', '<U10'), ('age', '<i8'), ('weight', '<f4')])
```

A **compound type** can also be **specified as a list of tuples**:

```
In [ ]: np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
```

```
Out[ ]: dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

If the **names of the types do not matter** to you, you can **specify the types alone** in a comma-separated string:

```
In [ ]: np.dtype('S10,i4,f8')
```

```
Out[ ]: dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

The **shortened string format codes** may **not be immediately intuitive**, but they are **built on simple principles**.

- **The first (optional) character** `<` or `>`, means "**little endian**" or "**big endian**," respectively, and specifies the ordering convention for significant bits.
- **The next character** specifies **the type of data**: characters, bytes, ints, floating points, and so on (see the table below).
- **The last character** or characters represent the **size of the object in bytes**.

Character	Description	Example
'b'	Byte	<code>np.dtype('b')</code>
'i'	Signed integer	<code>np.dtype('i4') == np.int32</code>
'u'	Unsigned integer	<code>np.dtype('u1') == np.uint8</code>

Character	Description	Example
'f'	Floating point	<code>np.dtype('f8') == np.int64</code>
'c'	Complex floating point	<code>np.dtype('c16') == np.complex128</code>
'S', 'a'	String	<code>np.dtype('S5')</code>
'U'	Unicode string	<code>np.dtype('U') == np.str_</code>
'V'	Raw data (void)	<code>np.dtype('V') == np.void</code>

More Advanced Compound Types

It is possible to **define even more advanced compound types**.

For example, you can create a type where **each element contains an array or matrix of values**.

Here, we'll create a data type with a **mat** component consisting of a **3×3 floating-point matrix**:

```
In [ ]: tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
X = np.zeros(1, dtype=tp)
print(X[0])
print(X['mat'][0])
```

```
(0, [[0., 0., 0.], [0., 0., 0.], [0., 0., 0.]])
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

Now **each element** in the **X** array **consists of an id** and a **3×3 matrix**.

Why would you use **this rather than** a simple **multidimensional array**, or perhaps a **Python dictionary**?

One reason is that this **NumPy dtype** **directly maps onto a C structure definition.**

So the **buffer containing the array** content can be **accessed directly** within an appropriately written C program.

If you find yourself writing a Python interface to a legacy C or Fortran library that manipulates structured data, **structured arrays can provide a powerful interface.**

Record Arrays: Structured Arrays with a Twist

NumPy also provides **record arrays** (instances of the `np.recarray` class).

Which are **almost identical to the structured arrays** just described.

But with **one additional feature**:

Fields can be accessed as attributes rather than as dictionary keys.

Recall that **we previously accessed the ages in our sample dataset by writing:**

```
In [ ]: data['age']
```

```
Out[ ]: array([25, 45, 37, 19], dtype=int32)
```

If we **view our data as a record array instead**, we can **access this with slightly fewer keystrokes**:

```
In [ ]: data_rec = data.view(np.recarray)
        data_rec.age
```

```
Out[ ]: array([25, 45, 37, 19], dtype=int32)
```

The **downside** is that **for record arrays**, there is some **extra overhead involved in accessing the fields**, even when using the same syntax:

```
In [ ]: %timeit data['age']  
        %timeit data_rec['age']  
        %timeit data_rec.age
```

121 ns \pm 1.4 ns per loop (mean \pm std. dev. of 7 runs, 1000000 loops each)

2.41 μ s \pm 15.7 ns per loop (mean \pm std. dev. of 7 runs, 100000 loops each)

3.98 μ s \pm 20.5 ns per loop (mean \pm std. dev. of 7 runs, 100000 loops each)

Whether the **more convenient notation is worth the (slight) overhead will depend on your own application.**

On to Pandas

This chapter on structured and record arrays is purposely located at the end of this part of the book:

It **leads so well into the next package** we will cover: **Pandas**.

Structured arrays can come in **handy in certain situations**:

Like **when you're using NumPy arrays to map onto binary data formats in C, Fortran, or another language**.

But for **day-to-day use of structured data**, the **Pandas** package is a **much better choice**.

We'll explore it in depth in the chapters that follow.