Data type objects (dtype)

A data type object (an instance of **numpy.dtype** class) describes how the bytes in the fixed-size block of memory corresponding to an array item should be interpreted. It describes the following aspects of the data:

1. Type of the data (integer, float, Python object, etc.)
2. Size of the data (how many bytes is in *e.g.* the integer)
3. Byte order of the data (little-endian or big-endian)
4. If the data type is structured, an aggregate of other data types, (*e.g.*, describing an array item consisting of an integer and a float),
   1. what are the names of the "fields" of the structure, by which they can be accessed,
   2. what is the data-type of each field, and
   3. which part of the memory block each field takes.
5. If the data type is a sub-array, what is its shape and data type.

To describe the type of scalar data, there are several built-in scalar types in NumPy for various precision of integers, floating-point numbers, *etc*. An item extracted from an array, *e.g.*, by indexing, will be a Python object whose type is the scalar type associated with the data type of the array.

Note that the scalar types are not **dtype** objects, even though they can be used in place of one whenever a data type specification is needed in NumPy.

Structured data types are formed by creating a data type whose fields contain other data types. Each field has a name by which it can be accessed. The parent data type should be of sufficient size to contain all its fields; the parent is nearly always based on the **void** type which allows an arbitrary item size. Structured data types may also contain nested structured sub-array data types in their fields.

Finally, a data type can describe items that are themselves arrays of items of another data type. These sub-arrays must, however, be of a fixed size.

If an array is created using a data-type describing a sub-array, the dimensions of the sub-array are appended to the shape of the array when the array is created. Sub-arrays in a field of a structured type behave differently, see Field Access.

Sub-arrays always have a C-contiguous memory layout.

Example:
A simple data type containing a 32-bit big-endian integer: (see Specifying and constructing data types for details on construction)

```
>>> dt = np.dtype('>i4')
>>> dt.byteorder
'>'
>>> dt.itemsize
4
>>> dt.name
'int32'
>>> dt.type is np.int32
True
```

The corresponding array scalar type is **int32**.

Example:
A structured data type containing a 16-character string (in field 'name') and a sub-array of two 64-bit floating-point number (in field 'grades'):

```
>>> dt = np.dtype([('name', np.unicode_, 16), ('grades', np.float64, (2,))])
>>> dt['name']
dtype('|U16')
>>> dt['grades']
dtype(('float64',(2,)))
```

Items of an array of this data type are wrapped in an array scalar type that also has two fields:

```
>>> x = np.array([('Sarah', (8.0, 7.0)), ('John', (6.0, 7.0))], dtype=dt)
>>> x[1]
('John', [6.0, 7.0])
>>> x[1]['grades']
array([ 6.,  7.])
>>> type(x[1])
<type 'numpy.void'>
>>> type(x[1]['grades'])
<type 'numpy.ndarray'>
```

Specifying and constructing data types

Whenever a data-type is required in a NumPy function or method, either a **dtype** object or something that can be converted to one can be supplied. Such conversions are done by the **dtype** constructor:

dtype(obj[, align, copy])	Create a data type object.

What can be converted to a data-type object is described below:

dtype object

> Used as-is.

None

> The default data type: **float_**.

Array-scalar types

> The 24 built-in array scalar type objects all convert to an associated data-type object. This is true for their sub-classes as well.
> Note that not all data-type information can be supplied with a type-object: for example, flexible data-types have a default *itemsize* of 0, and require an explicitly given size to be useful.
>
> **Example:**
> ```
> >>> dt = np.dtype(np.int32)      # 32-bit integer
> >>> dt = np.dtype(np.complex128) # 128-bit complex floating-point number
> ```

Generic types

> The generic hierarchical type objects convert to corresponding type objects according to the associations:

number , inexact , floating	float
complexfloating	cfloat
integer , signedinteger	int_
unsignedinteger	uint
character	string
generic, flexible	void

Built-in Python types

> Several python types are equivalent to a corresponding array scalar when used to generate a **dtype** object:

int	int_
bool	bool_
float	float_
complex	cfloat
bytes	bytes_
str	bytes_ (Python2) or unicode_ (Python3)
unicode	unicode_
buffer	void
(all others)	object_

> Note that `str` refers to either null terminated bytes or unicode strings depending on the Python version. In code targeting both Python 2 and 3 `np.unicode_` should be used as a dtype for strings. See Note on string types.
>
> **Example:**
> ```
> >>> dt = np.dtype(float)   # Python-compatible floating-point number
> >>> dt = np.dtype(int)     # Python-compatible integer
> >>> dt = np.dtype(object)  # Python object
> ```

Types with `.dtype`

> Any type object with a `dtype` attribute: The attribute will be accessed and used directly. The attribute must return something that is convertible into a dtype object.

Several kinds of strings can be converted. Recognized strings can be prepended with `'>'` (big-endian), `'<'` (little-endian), or `'='` (hardware-native, the default), to specify the byte order.

One-character strings

> Each built-in data-type has a character code (the updated Numeric typecodes), that uniquely identifies it.
>
> **Example:**
> ```
> >>> dt = np.dtype('b')  # byte, native byte order
> >>> dt = np.dtype('>H') # big-endian unsigned short
> >>> dt = np.dtype('<f') # little-endian single-precision float
> >>> dt = np.dtype('d')  # double-precision floating-point number
> ```

Array-protocol type strings (see The Array Interface)

> The first character specifies the kind of data and the remaining characters specify the number of bytes per item, except for Unicode, where it is interpreted as the number of characters. The item size must correspond to an existing type, or an error will be raised. The supported kinds are

`'?'`	boolean
`'b'`	(signed) byte
`'B'`	unsigned byte
`'i'`	(signed) integer
`'u'`	unsigned integer
`'f'`	floating-point
`'c'`	complex-floating point
`'m'`	timedelta
`'M'`	datetime
`'O'`	(Python) objects
`'S'`, `'a'`	zero-terminated bytes (not recommended)
`'U'`	Unicode string
`'V'`	raw data (**void**)

> **Example:**
> ```
> >>> dt = np.dtype('i4')   # 32-bit signed integer
> >>> dt = np.dtype('f8')   # 64-bit floating-point number
> >>> dt = np.dtype('c16')  # 128-bit complex floating-point number
> >>> dt = np.dtype('a25')  # 25-length zero-terminated bytes
> >>> dt = np.dtype('U25')  # 25-character string
> ```
>
> **Note on string types:**
> For backward compatibility with Python 2 the `S` and `a` typestrings remain zero-terminated bytes and `np.string_` continues to map to `np.bytes_`. To use actual strings in Python 3 use `U` or `np.unicode_`. For signed bytes that do not need zero-termination `b` or `i1` can be used.

String with comma-separated fields

> A short-hand notation for specifying the format of a structured data type is a comma-separated string of basic formats.
> A basic format in this context is an optional shape specifier followed by an array-protocol type string. Parenthesis are required on the shape if it has more than one dimension. NumPy allows a modification on the format in that any string that can uniquely identify the type can be used to specify the data-type in a field. The generated data-type fields are named `'f0'`, `'f1'`, …, 'f<N-1>' where N (>1) is the number of comma-separated basic formats in the string. If the optional shape specifier is provided, then the data-type for the corresponding field describes a sub-array.
>
> **Example:**
> - field named `f0` containing a 32-bit integer
> - field named `f1` containing a 2 x 3 sub-array of 64-bit floating-point numbers
> - field named `f2` containing a 32-bit floating-point number
>
> ```
> >>> dt = np.dtype("i4, (2,3)f8, f4")
> ```
>
> - field named `f0` containing a 3-character string
> - field named `f1` containing a sub-array of shape (3,) containing 64-bit unsigned integers
> - field named `f2` containing a 3 x 4 sub-array containing 10-character strings
>
> ```
> >>> dt = np.dtype("a3, 3u8, (3,4)a10")
> ```

Type strings

> Any string in **numpy.sctypeDict**.keys():
>
> **Example:**
> ```
> >>> dt = np.dtype('uint32')   # 32-bit unsigned integer
> >>> dt = np.dtype('Float64')  # 64-bit floating-point number
> ```

`(flexible_dtype, itemsize)`

> The first argument must be an object that is converted to a zero-sized flexible data-type object, the second argument is an integer providing the desired itemsize.
>
> **Example:**
> ```
> >>> dt = np.dtype((np.void, 10))  # 10-byte wide data block
> >>> dt = np.dtype(('U', 10))   # 10-character unicode string
> ```

`(fixed_dtype, shape)`

> The first argument is any object that can be converted into a fixed-size data-type object. The second argument is the desired shape of this type. If the shape parameter is 1, then the data-type object is equivalent to fixed dtype. If *shape* is a tuple, then the new dtype defines a sub-array of the given shape.
>
> **Example:**
> ```
> >>> dt = np.dtype((np.int32, (2,2)))          # 2 x 2 integer sub-array
> >>> dt = np.dtype(('U10', 1))                 # 10-character string
> >>> dt = np.dtype(('i4, (2,3)f8, f4', (2,3))) # 2 x 3 structured sub-array
> ```

`[(field_name, field_dtype, field_shape), ...]`

> *obj* should be a list of fields where each field is described by a tuple of length 2 or 3. (Equivalent to the `descr` item in the **__array_interface__** attribute.)
> The first element, *field_name*, is the field name (if this is `''` then a standard field name, `'f#'`, is assigned). The field name may also be a 2-tuple of strings where the first string is either a "title" (which may be any string or unicode string) or meta-data for the field which can be any object, and the second string is the "name" which must be a valid Python identifier.
> The second element, *field_dtype*, can be anything that can be interpreted as a data-type.
> The optional third element *field_shape* contains the shape if this field represents an array of the data-type in the second element. Note that a 3-tuple with a third argument equal to 1 is equivalent to a 2-tuple.
> This style does not accept *align* in the **dtype** constructor as it is assumed that all of the memory is accounted for by the array interface description.
>
> **Example:**
> Data-type with fields `big` (big-endian 32-bit integer) and `little` (little-endian 32-bit integer):
> ```
> >>> dt = np.dtype([('big', '>i4'), ('little', '<i4')])
> ```
> Data-type with fields `R`, `G`, `B`, `A`, each being an unsigned 8-bit integer:
> ```
> >>> dt = np.dtype([('R','u1'), ('G','u1'), ('B','u1'), ('A','u1')])
> ```

`{'names': ..., 'formats': ..., 'offsets': ..., 'titles': ..., 'itemsize': ...}`

> This style has two required and three optional keys. The *names* and *formats* keys are required. Their respective values are equal-length lists with the field names and the field formats. The field names must be strings and the field formats can be any object accepted by **dtype** constructor.
> When the optional keys *offsets* and *titles* are provided, their values must each be lists of the same length as the *names* and *formats* lists. The *offsets* value is a list of byte offsets (limited to **ctypes.c_int**) for each field, while the *titles* value is a list of titles for each field (**None** can be used if no title is desired for that field). The *titles* can be any **string** or **unicode** object and will add another entry to the fields dictionary keyed by the title and referencing the same field tuple which will contain the title as an additional tuple member.
> The *itemsize* key allows the total size of the dtype to be set, and must be an integer large enough so all the fields are within the dtype. If the dtype being constructed is aligned, the *itemsize* must also be divisible by the struct alignment. Total dtype *itemsize* is limited to **ctypes.c_int**.
>
> **Example:**
> Data type with fields `r`, `g`, `b`, `a`, each being an 8-bit unsigned integer:
> ```
> >>> dt = np.dtype({'names': ['r','g','b','a'],
> ...                'formats': [uint8, uint8, uint8, uint8]})
> ```
> Data type with fields `r` and `b` (with the given titles), both being 8-bit unsigned integers, the first at byte position 0 from the start of the field and the second at position 2:
> ```
> >>> dt = np.dtype({'names': ['r','b'], 'formats': ['u1', 'u1'],
> ...                'offsets': [0, 2],
> ...                'titles': ['Red pixel', 'Blue pixel']})
> ```

`{'field1': ..., 'field2': ..., ...}`

> This usage is discouraged, because it is ambiguous with the other dict-based construction method. If you have a field called 'names' and a field called 'formats' there will be a conflict.
> This style allows passing in the **fields** attribute of a data-type object.
> *obj* should contain string or unicode keys that refer to `(data-type, offset)` or `(data-type, offset, title)` tuples.
>
> **Example:**
> Data type containing field `col1` (10-character string at byte position 0), `col2` (32-bit float at byte position 10), and `col3` (integers at byte position 14):
> ```
> >>> dt = np.dtype({'col1': ('U10', 0), 'col2': (float32, 10),
>     'col3': (int, 14)})
> ```

`(base_dtype, new_dtype)`

> In NumPy 1.7 and later, this form allows *base_dtype* to be interpreted as a structured dtype. Arrays created with this dtype will have underlying dtype *base_dtype* but will have fields and flags taken from *new_dtype*. This is useful for creating custom structured dtypes, as done in record arrays.
> This form also makes it possible to specify struct dtypes with overlapping fields, functioning like the 'union' type in C. This usage is discouraged, however, and the union mechanism is preferred.
> Both arguments must be convertible to data-type objects with the same total size.
>
> **Example:**
> 32-bit integer, whose first two bytes are interpreted as an integer via field `real`, and the following two bytes via field `imag`.
> ```
> >>> dt = np.dtype((np.int32,{'real':(np.int16, 0),'imag':(np.int16, 2)}))
> ```
> 32-bit integer, which is interpreted as consisting of a sub-array of shape `(4,)` containing 8-bit integers:
> ```
> >>> dt = np.dtype((np.int32, (np.int8, 4)))
> ```
> 32-bit integer, containing fields `r`, `g`, `b`, `a` that interpret the 4 bytes in the integer as four unsigned integers:
> ```
> >>> dt = np.dtype(('i4', [('r','u1'),('g','u1'),('b','u1'),('a','u1')]))
> ```

dtype

NumPy data type descriptions are instances of the **dtype** class.

Attributes

The type of the data is described by the following **dtype** attributes:

dtype.type	The type object used to instantiate a scalar of this data-type.
dtype.kind	A character code (one of 'biufcmMOSUV') identifying the general kind of data.
dtype.char	A unique character code for each of the 21 different built-in types.
dtype.num	A unique number for each of the 21 different built-in types.
dtype.str	The array-protocol typestring of this data-type object.

Size of the data is in turn described by:

dtype.name	A bit-width name for this data-type.
dtype.itemsize	The element size of this data-type object.

Endianness of this data:

dtype.byteorder	A character indicating the byte-order of this data-type object.

Information about sub-data-types in a structured data type:

dtype.fields	Dictionary of named fields defined for this data type, or `None`.
dtype.names	Ordered list of field names, or `None` if there are no fields.

For data types that describe sub-arrays:

dtype.subdtype	Tuple `(item_dtype, shape)` if this **dtype** describes a sub-array, and None otherwise.
dtype.shape	Shape tuple of the sub-array if this data type describes a sub-array, and `()` otherwise.

Attributes providing additional information:

dtype.hasobject	Boolean indicating whether this dtype contains any reference-counted objects in any fields or sub-dtypes.
dtype.flags	Bit-flags describing how this data type is to be interpreted.
dtype.isbuiltin	Integer indicating how this dtype relates to the built-in dtypes.
dtype.isnative	Boolean indicating whether the byte order of this dtype is native to the platform.
dtype.descr	*__array_interface__* description of the data-type.
dtype.alignment	The required alignment (bytes) of this data-type according to the compiler.

Methods

Data types have the following method for changing the byte order:

dtype.newbyteorder([new_order])	Return a new dtype with a different byte order.

The following methods implement the pickle protocol:

dtype.__reduce__	helper for pickle
dtype.__setstate__	

Quick search

search