8.7. `array` — Efficient arrays of numeric values

This module defines an object type which can compactly represent an array of basic values: characters, integers, floating point numbers. Arrays are sequence types and behave very much like lists, except that the type of objects stored in them is constrained. The type is specified at object creation time by using a *type code*, which is a single character. The following type codes are defined:

Type code	C Type	Python Type	Minimum size in bytes	Notes
`'b'`	signed char	int	1	
`'B'`	unsigned char	int	1	
`'u'`	Py_UNICODE	Unicode character	2	(1)
`'h'`	signed short	int	2	
`'H'`	unsigned short	int	2	
`'i'`	signed int	int	2	
`'I'`	unsigned int	int	2	
`'l'`	signed long	int	4	
`'L'`	unsigned long	int	4	
`'q'`	signed long long	int	8	(2)
`'Q'`	unsigned long long	int	8	(2)
`'f'`	float	float	4	
`'d'`	double	float	8	

Notes:

1. The `'u'` type code corresponds to Python's obsolete unicode character (`Py_UNICODE` which is `wchar_t`). Depending on the platform, it can be 16 bits or 32 bits.

   `'u'` will be removed together with the rest of the `Py_UNICODE` API.

   Deprecated since version 3.3, will be removed in version 4.0.

2. The `'q'` and `'Q'` type codes are available only if the platform C compiler used to build Python supports C `long long`, or, on Windows, `__int64`.

New in version 3.3.

The actual representation of values is determined by the machine architecture (strictly speaking, by the C implementation). The actual size can be accessed through the `itemsize` attribute.

The module defines the following type:

class `array.`**`array`**(*typecode*[, *initializer*])

A new array whose items are restricted by *typecode*, and initialized from the optional *initializer* value, which must be a list, a bytes-like object, or iterable over elements of the appropriate type.

If given a list or string, the initializer is passed to the new array's `fromlist()`, `frombytes()`, or `fromunicode()` method (see below) to add initial items to the array. Otherwise, the iterable initializer is passed to the `extend()` method.

`array.`**`typecodes`**

A string with all available type codes.

Array objects support the ordinary sequence operations of indexing, slicing, concatenation, and multiplication. When using slice assignment, the assigned value must be an array object with the same type code; in all other cases, `TypeError` is raised. Array objects also implement the buffer interface, and may be used wherever bytes-like objects are supported.

The following data items and methods are also supported:

`array.`**`typecode`**

The typecode character used to create the array.

`array.`**`itemsize`**

The length in bytes of one array item in the internal representation.

`array.`**`append`**(*x*)

Append a new item with value *x* to the end of the array.

`array.`**`buffer_info`**()

Return a tuple `(address, length)` giving the current memory address and the length in elements of the buffer used to hold array's contents. The size of the memory buffer in bytes can be computed as `array.buffer_info()[1] * array.itemsize`. This is occasionally useful when working with low-level (and inherently unsafe) I/O interfaces that require memory addresses, such as certain `ioctl()` operations. The returned numbers are valid as long as the array exists and no length-changing operations are applied to it.

Note: When using array objects from code written in C or C++ (the only way to effectively make use of this information), it makes more sense to use the buffer interface supported by array objects. This method is maintained for backward compatibility and should be avoided in new code. The buffer interface is documented in Buffer Protocol.

`array.`**`byteswap`**()

"Byteswap" all items of the array. This is only supported for values which are 1, 2, 4, or 8 bytes in size; for other types of values, `RuntimeError` is raised. It is useful when reading data from a file written on a machine with a different byte order.

`array.`**`count`**(*x*)

Return the number of occurrences of *x* in the array.

`array.`**`extend`**(*iterable*)

Append items from *iterable* to the end of the array. If *iterable* is another array, it must have *exactly* the same type code; if not, `TypeError` will be raised. If *iterable* is not an array, it must be iterable and its elements must be the right type to be appended to the array.

`array.`**`frombytes`**(*s*)

Appends items from the string, interpreting the string as an array of machine values (as if it had been read from a file using the `fromfile()` method).

New in version 3.2: `fromstring()` is renamed to `frombytes()` for clarity.

`array.`**`fromfile`**(*f*, *n*)

Read *n* items (as machine values) from the file object *f* and append them to the end of the array. If less than *n* items are available, `EOFError` is raised, but the items that were available are still inserted into the array. *f* must be a real built-in file object; something else with a `read()` method won't do.

`array.`**`fromlist`**(*list*)

Append items from the list. This is equivalent to `for x in list: a.append(x)` except that if there is a type error, the array is unchanged.

`array.`**`fromstring`**()

Deprecated alias for `frombytes()`.

`array.`**`fromunicode`**(*s*)

Extends this array with data from the given unicode string. The array must be a type `'u'` array; otherwise a `ValueError` is raised. Use `array.frombytes`

`(unicodestring.encode(enc))` to append Unicode data to an array of some other type.

`array.`**`index`**(*x*)
Return the smallest *i* such that *i* is the index of the first occurrence of *x* in the array.

`array.`**`insert`**(*i*, *x*)
Insert a new item with value *x* in the array before position *i*. Negative values are treated as being relative to the end of the array.

`array.`**`pop`**([*i*])
Removes the item with the index *i* from the array and returns it. The optional argument defaults to `-1`, so that by default the last item is removed and returned.

`array.`**`remove`**(*x*)
Remove the first occurrence of *x* from the array.

`array.`**`reverse`**()
Reverse the order of the items in the array.

`array.`**`tobytes`**()
Convert the array to an array of machine values and return the bytes representation (the same sequence of bytes that would be written to a file by the `tofile()` method.)

New in version 3.2: `tostring()` is renamed to `tobytes()` for clarity.

`array.`**`tofile`**(*f*)
Write all items (as machine values) to the file object *f*.

`array.`**`tolist`**()
Convert the array to an ordinary list with the same items.

`array.`**`tostring`**()
Deprecated alias for `tobytes()`.

`array.`**`tounicode`**()
Convert the array to a unicode string. The array must be a type `'u'` array; otherwise a `ValueError` is raised. Use `array.tobytes().decode(enc)` to obtain a unicode string from an array of some other type.

When an array object is printed or converted to a string, it is represented as `array(typecode, initializer)`. The *initializer* is omitted if the array is empty, otherwise it is a string if the *typecode* is `'u'`, otherwise it is a list of numbers. The string is

guaranteed to be able to be converted back to an array with the same type and value using `eval()`, so long as the `array` class has been imported using `from array import array`. Examples:

```
array('l')
array('u', 'hello \u2641')
array('l', [1, 2, 3, 4, 5])
array('d', [1.0, 2.0, 3.14])
```

See also:

Module `struct`
: Packing and unpacking of heterogeneous binary data.

Module `xdrlib`
: Packing and unpacking of External Data Representation (XDR) data as used in some remote procedure call systems.

The Numerical Python Documentation
: The Numeric Python extension (NumPy) defines another array type; see http://www.numpy.org/ for further information about Numerical Python.