Buffer Protocol

Certain objects available in Python wrap access to an underlying memory array or *buffer*. Such objects include the built-in `bytes` and `bytearray`, and some extension types like `array.array`. Third-party libraries may define their own types for special purposes, such as image processing or numeric analysis.

While each of these types have their own semantics, they share the common characteristic of being backed by a possibly large memory buffer. It is then desirable, in some situations, to access that buffer directly and without intermediate copying.

Python provides such a facility at the C level in the form of the buffer protocol. This protocol has two sides:

- on the producer side, a type can export a "buffer interface" which allows objects of that type to expose information about their underlying buffer. This interface is described in the section Buffer Object Structures;
- on the consumer side, several means are available to obtain a pointer to the raw underlying data of an object (for example a method parameter).

Simple objects such as `bytes` and `bytearray` expose their underlying buffer in byte-oriented form. Other forms are possible; for example, the elements exposed by an `array.array` can be multi-byte values.

An example consumer of the buffer interface is the `write()` method of file objects: any object that can export a series of bytes through the buffer interface can be written to a file. While `write()` only needs read-only access to the internal contents of the object passed to it, other methods such as `readinto()` need write access to the contents of their argument. The buffer interface allows objects to selectively allow or reject exporting of read-write and read-only buffers.

There are two ways for a consumer of the buffer interface to acquire a buffer over a target object:

- call `PyObject_GetBuffer()` with the right parameters;
- call `PyArg_ParseTuple()` (or one of its siblings) with one of the `y*`, `w*` or `s*` format codes.

In both cases, `PyBuffer_Release()` must be called when the buffer isn't needed anymore. Failure to do so could lead to various issues such as resource leaks.

Buffer structure

Buffer structures (or simply "buffers") are useful as a way to expose the binary data from another object to the Python programmer. They can also be used as a zero-copy slicing mechanism. Using their ability to reference a block of memory, it is possible to expose any data to the Python programmer quite easily. The memory could be a large, constant array

in a C extension, it could be a raw block of memory for manipulation before passing to an operating system library, or it could be used to pass around structured data in its native, in-memory format.

Contrary to most data types exposed by the Python interpreter, buffers are not `PyObject` pointers but rather simple C structures. This allows them to be created and copied very simply. When a generic wrapper around a buffer is needed, a memoryview object can be created.

For short instructions how to write an exporting object, see Buffer Object Structures. For obtaining a buffer, see `PyObject_GetBuffer()`.

`Py_buffer`

void *`buf`

A pointer to the start of the logical structure described by the buffer fields. This can be any location within the underlying physical memory block of the exporter. For example, with negative `strides` the value may point to the end of the memory block.

For contiguous arrays, the value points to the beginning of the memory block.

void *`obj`

A new reference to the exporting object. The reference is owned by the consumer and automatically decremented and set to *NULL* by `PyBuffer_Release()`. The field is the equivalent of the return value of any standard C-API function.

As a special case, for *temporary* buffers that are wrapped by `PyMemoryView_FromBuffer()` or `PyBuffer_FillInfo()` this field is *NULL*. In general, exporting objects MUST NOT use this scheme.

Py_ssize_t `len`

`product(shape) * itemsize`. For contiguous arrays, this is the length of the underlying memory block. For non-contiguous arrays, it is the length that the logical structure would have if it were copied to a contiguous representation.

Accessing `((char *)buf)[0] up to ((char *)buf)[len-1]` is only valid if the buffer has been obtained by a request that guarantees contiguity. In most cases such a request will be `PyBUF_SIMPLE` or `PyBUF_WRITABLE`.

int `readonly`

An indicator of whether the buffer is read-only. This field is controlled by the `PyBUF_WRITABLE` flag.

Py_ssize_t `itemsize`

Item size in bytes of a single element. Same as the value of `struct.calcsize()` called on non-NULL `format` values.

Important exception: If a consumer requests a buffer without the `PyBUF_FORMAT` flag, `format` will be set to *NULL*, but `itemsize` still has the value for the original format.

If `shape` is present, the equality `product(shape) * itemsize == len` still holds and the consumer can use `itemsize` to navigate the buffer.

If `shape` is *NULL* as a result of a `PyBUF_SIMPLE` or a `PyBUF_WRITABLE` request, the consumer must disregard `itemsize` and assume `itemsize == 1`.

const char ***format**

A *NUL* terminated string in `struct` module style syntax describing the contents of a single item. If this is *NULL*, `"B"` (unsigned bytes) is assumed.

This field is controlled by the `PyBUF_FORMAT` flag.

int **ndim**

The number of dimensions the memory represents as an n-dimensional array. If it is `0`, `buf` points to a single item representing a scalar. In this case, `shape`, `strides` and `suboffsets` MUST be *NULL*.

The macro `PyBUF_MAX_NDIM` limits the maximum number of dimensions to 64. Exporters MUST respect this limit, consumers of multi-dimensional buffers SHOULD be able to handle up to `PyBUF_MAX_NDIM` dimensions.

Py_ssize_t ***shape**

An array of `Py_ssize_t` of length `ndim` indicating the shape of the memory as an n-dimensional array. Note that `shape[0] * ... * shape[ndim-1] * itemsize` MUST be equal to `len`.

Shape values are restricted to `shape[n] >= 0`. The case `shape[n] == 0` requires special attention. See complex arrays for further information.

The shape array is read-only for the consumer.

Py_ssize_t ***strides**

An array of `Py_ssize_t` of length `ndim` giving the number of bytes to skip to get to a new element in each dimension.

Stride values can be any integer. For regular arrays, strides are usually positive, but a consumer MUST be able to handle the case `strides[n] <= 0`. See complex arrays for further information.

The strides array is read-only for the consumer.

Py_ssize_t ***suboffsets**

An array of `Py_ssize_t` of length `ndim`. If `suboffsets[n] >= 0`, the values stored along the nth dimension are pointers and the suboffset value dictates how

many bytes to add to each pointer after de-referencing. A suboffset value that is negative indicates that no de-referencing should occur (striding in a contiguous memory block).

If all suboffsets are negative (i.e. no de-referencing is needed, then this field must be NULL (the default value).

This type of array representation is used by the Python Imaging Library (PIL). See complex arrays for further information how to access elements of such an array.

The suboffsets array is read-only for the consumer.

void ***internal**

This is for use internally by the exporting object. For example, this might be re-cast as an integer by the exporter and used to store flags about whether or not the shape, strides, and suboffsets arrays must be freed when the buffer is released. The consumer MUST NOT alter this value.

Buffer request types

Buffers are usually obtained by sending a buffer request to an exporting object via `PyObject_GetBuffer()`. Since the complexity of the logical structure of the memory can vary drastically, the consumer uses the *flags* argument to specify the exact buffer type it can handle.

All `Py_buffer` fields are unambiguously defined by the request type.

request-independent fields

The following fields are not influenced by *flags* and must always be filled in with the correct values: `obj`, `buf`, `len`, `itemsize`, `ndim`.

readonly, format

PyBUF_WRITABLE

Controls the `readonly` field. If set, the exporter MUST provide a writable buffer or else report failure. Otherwise, the exporter MAY provide either a read-only or writable buffer, but the choice MUST be consistent for all consumers.

PyBUF_FORMAT

Controls the `format` field. If set, this field MUST be filled in correctly. Otherwise, this field MUST be *NULL*.

`PyBUF_WRITABLE` can be |'d to any of the flags in the next section. Since `PyBUF_SIMPLE` is defined as 0, `PyBUF_WRITABLE` can be used as a stand-alone flag to request a simple writable buffer.

`PyBUF_FORMAT` can be |'d to any of the flags except `PyBUF_SIMPLE`. The latter already implies format `B` (unsigned bytes).

shape, strides, suboffsets

The flags that control the logical structure of the memory are listed in decreasing order of complexity. Note that each flag contains all bits of the flags below it.

Request	shape	strides	suboffsets
PyBUF_INDIRECT	yes	yes	if needed
PyBUF_STRIDES	yes	yes	NULL
PyBUF_ND	yes	NULL	NULL
PyBUF_SIMPLE	NULL	NULL	NULL

contiguity requests

C or Fortran contiguity can be explicitly requested, with and without stride information. Without stride information, the buffer must be C-contiguous.

Request	shape	strides	suboffsets	contig
PyBUF_C_CONTIGUOUS	yes	yes	NULL	C
PyBUF_F_CONTIGUOUS	yes	yes	NULL	F
PyBUF_ANY_CONTIGUOUS	yes	yes	NULL	C or F
PyBUF_ND	yes	NULL	NULL	C

compound requests

All possible requests are fully defined by some combination of the flags in the previous section. For convenience, the buffer protocol provides frequently used combinations as single flags.

In the following table *U* stands for undefined contiguity. The consumer would have to call `PyBuffer_IsContiguous()` to determine contiguity.

Request	shape	strides	suboffsets	contig	readonly	format
PyBUF_FULL	yes	yes	if needed	U	0	yes
PyBUF_FULL_RO	yes	yes	if needed	U	1 or 0	yes
PyBUF_RECORDS	yes	yes	NULL	U	0	yes
PyBUF_RECORDS_RO	yes	yes	NULL	U	1 or 0	yes
PyBUF_STRIDED	yes	yes	NULL	U	0	NULL
PyBUF_STRIDED_RO	yes	yes	NULL	U	1 or 0	NULL
PyBUF_CONTIG	yes	NULL	NULL	C	0	NULL
PyBUF_CONTIG_RO	yes	NULL	NULL	C	1 or 0	NULL

Complex arrays

NumPy-style: shape and strides

The logical structure of NumPy-style arrays is defined by `itemsize`, `ndim`, `shape` and `strides`.

If `ndim == 0`, the memory location pointed to by `buf` is interpreted as a scalar of size `itemsize`. In that case, both `shape` and `strides` are *NULL*.

If `strides` is *NULL*, the array is interpreted as a standard n-dimensional C-array. Otherwise, the consumer must access an n-dimensional array as follows:

```
ptr = (char *)buf + indices[0] * strides[0] + ... + indices[n-1]
* strides[n-1] item = *((typeof(item) *)ptr);
```

As noted above, `buf` can point to any location within the actual memory block. An exporter can check the validity of a buffer with this function:

```
def verify_structure(memlen, itemsize, ndim, shape, strides, offset):
    """Verify that the parameters represent a valid array within
       the bounds of the allocated memory:
           char *mem: start of the physical memory block
           memlen: length of the physical memory block
           offset: (char *)buf - mem
    """
    if offset % itemsize:
        return False
    if offset < 0 or offset+itemsize > memlen:
```

```
        return False
    if any(v % itemsize for v in strides):
        return False

    if ndim <= 0:
        return ndim == 0 and not shape and not strides
    if 0 in shape:
        return True

    imin = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] <= 0)
    imax = sum(strides[j]*(shape[j]-1) for j in range(ndim)
               if strides[j] > 0)

    return 0 <= offset+imin and offset+imax+itemsize <= memlen
```

PIL-style: shape, strides and suboffsets

In addition to the regular items, PIL-style arrays can contain pointers that must be followed in order to get to the next element in a dimension. For example, the regular three-dimensional C-array `char v[2][2][3]` can also be viewed as an array of 2 pointers to 2 two-dimensional arrays: `char (*v[2])[2][3]`. In suboffsets representation, those two pointers can be embedded at the start of `buf`, pointing to two `char x[2][3]` arrays that can be located anywhere in memory.

Here is a function that returns a pointer to the element in an N-D array pointed to by an N-dimensional index when there are both non-NULL strides and suboffsets:

```
void *get_item_pointer(int ndim, void *buf, Py_ssize_t *strides,
                       Py_ssize_t *suboffsets, Py_ssize_t *indices) {
    char *pointer = (char*)buf;
    int i;
    for (i = 0; i < ndim; i++) {
        pointer += strides[i] * indices[i];
        if (suboffsets[i] >=0 ) {
            pointer = *((char**)pointer) + suboffsets[i];
        }
    }
    return (void*)pointer;
}
```

Buffer-related functions

int **PyObject_CheckBuffer**(PyObject **obj*)

Return `1` if *obj* supports the buffer interface otherwise `0`. When `1` is returned, it doesn't guarantee that `PyObject_GetBuffer()` will succeed.

int **PyObject_GetBuffer**(PyObject **exporter*, Py_buffer **view*, int *flags*)

Send a request to *exporter* to fill in *view* as specified by *flags*. If the exporter cannot provide a buffer of the exact type, it MUST raise `PyExc_BufferError`, set `view->obj` to *NULL* and return `-1`.

On success, fill in *view*, set `view->obj` to a new reference to *exporter* and return 0. In the case of chained buffer providers that redirect requests to a single object, `view->obj` MAY refer to this object instead of *exporter* (See Buffer Object Structures).

Successful calls to `PyObject_GetBuffer()` must be paired with calls to `PyBuffer_Release()`, similar to `malloc()` and `free()`. Thus, after the consumer is done with the buffer, `PyBuffer_Release()` must be called exactly once.

void **PyBuffer_Release**(Py_buffer **view*)

Release the buffer *view* and decrement the reference count for `view->obj`. This function MUST be called when the buffer is no longer being used, otherwise reference leaks may occur.

It is an error to call this function on a buffer that was not obtained via `PyObject_GetBuffer()`.

Py_ssize_t **PyBuffer_SizeFromFormat**(const char *)

Return the implied `itemsize` from `format`. This function is not yet implemented.

int **PyBuffer_IsContiguous**(Py_buffer **view*, char *order*)

Return `1` if the memory defined by the *view* is C-style (*order* is `'C'`) or Fortran-style (*order* is `'F'`) contiguous or either one (*order* is `'A'`). Return `0` otherwise.

int **PyBuffer_ToContiguous**(void **buf*, Py_buffer **src*, Py_ssize_t *len*, char *order*)

Copy *len* bytes from *src* to its contiguous representation in *buf*. *order* can be `'C'` or `'F'` (for C-style or Fortran-style ordering). `0` is returned on success, `-1` on error.

This function fails if *len* != *src->len*.

void **PyBuffer_FillContiguousStrides**(int *ndims*, Py_ssize_t **shape*, Py_ssize_t **strides*, int *itemsize*, char *order*)

Fill the *strides* array with byte-strides of a contiguous (C-style if *order* is `'C'` or Fortran-style if *order* is `'F'`) array of the given shape with the given number of bytes per element.

int **PyBuffer_FillInfo**(Py_buffer **view*, PyObject **exporter*, void **buf*, Py_ssize_t *len*, int *readonly*, int *flags*)

Handle buffer requests for an exporter that wants to expose *buf* of size *len* with writability set according to *readonly*. *buf* is interpreted as a sequence of unsigned bytes.

The *flags* argument indicates the request type. This function always fills in *view* as specified by flags, unless *buf* has been designated as read-only and `PyBUF_WRITABLE` is set in *flags*.

On success, set `view->obj` to a new reference to *exporter* and return 0. Otherwise, raise `PyExc_BufferError`, set `view->obj` to *NULL* and return `-1`;

If this function is used as part of a getbufferproc, *exporter* MUST be set to the exporting object and *flags* must be passed unmodified. Otherwise, *exporter* MUST be NULL.