Bytes Objects

- PyBytesWriter
  - Create, Finish, Discard
  - High-level API
  - Getters
  - Low-level API
  - Example creating the string b'abc'
  - Update PyBytes_FromStringAndSize() code
  - Update _PyBytes_Resize() code

These functions raise `TypeError` when expecting a bytes parameter and called with a non-bytes parameter.

The internal buffer of `PyBytesObject` always includes an extra trailing null byte for compatibility with null terminated C strings. This extra byte is not counted in `PyBytes_Size` nor in the various *length* and *size* arguments of the functions below.

pair: object; bytes

> This subtype of `PyObject` represents a Python bytes object.

> This instance of `PyTypeObject` represents the Python bytes type; it is the same object as `bytes` in the Python layer.

> Return true if the object *o* is a bytes object or an instance of a subtype of the bytes type. This function always succeeds.

> Return true if the object *o* is a bytes object, but not an instance of a subtype of the bytes type. This function always succeeds.

> Return a new bytes object with a copy of the string *v* as value on success, and `NULL` on failure. The parameter *v* must not be `NULL`; it will not be checked.

> Return a new bytes object with a copy of the string *v* as value and length *len* on success, and `NULL` on failure. If *v* is `NULL`, the contents of the bytes object are uninitialized.
>
> 3.15 Use the `PyBytesWriter` API instead of `PyBytes_FromStringAndSize(NULL, len)`.

> Take a C `printf`-style *format* string and a variable number of arguments, calculate the size of the resulting Python bytes object and return a bytes object with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the *format* string. The following format characters are allowed:
>
> |l||l||L|
>
> | Format Characters | Type | Comment |
> |---|---|---|
> | `%%` | *n/a* | The literal % character. |
> | `%c` | int | A single byte, represented as a C int. |
> | `%d` | int | Equivalent to `printf("%d")` .[^1] |
> | `%u` | unsigned int | Equivalent to `printf("%u")` .[^2] |
> | `%ld` | long | Equivalent to `printf("%ld")` .[^3] |
> | `%lu` | unsigned long | Equivalent to `printf("%lu")` .[^4] |
> | `%zd` | :c:type:`Py_ssize_t` | Equivalent to `printf("%zd")` .[^5] |
> | `%zu` | size_t | Equivalent to `printf("%zu")` .[^6] |

Format Characters	Type	Comment
`%i`	int	Equivalent to `printf("%i")` .[^7]
`%x`	int	Equivalent to `printf("%x")` .[^8]
`%s`	const char*	A null-terminated C character array.
`%p`	const void*	The hex representation of a C pointer. Mostly equivalent to `printf("%p")` except that it is guaranteed to start with the literal `0x` regardless of what the platform's `printf` yields.

An unrecognized format character causes all the rest of the format string to be copied as-is to the result object, and any extra arguments discarded.

> Identical to `PyBytes_FromFormat` except that it takes exactly two arguments.

> Return the bytes representation of object *o* that implements the buffer protocol.
>
> > [!NOTE] If the object implements the buffer protocol, then the buffer must not be mutated while the bytes object is being created.

> Return the length of the bytes in bytes object *o*.

> Similar to `PyBytes_Size`, but without error checking.

> Return a pointer to the contents of *o*. The pointer refers to the internal buffer of *o*, which consists of `len(o) + 1` bytes. The last byte in the buffer is always null,

> regardless of whether there are any other null bytes. The data must not be modified in any way, unless the object was just created using `PyBytes_FromStringAndSize(NULL, size)`. It must not be deallocated. If *o* is not a bytes object at all, `PyBytes_AsString` returns `NULL` and raises `TypeError`.

> Similar to `PyBytes_AsString`, but without error checking.

> Return the null-terminated contents of the object *obj* through the output variables *buffer* and *length*. Returns `0` on success.
>
> If *length* is `NULL`, the bytes object may not contain embedded null bytes; if it does, the function returns `-1` and a `ValueError` is raised.
>
> The buffer refers to an internal buffer of *obj*, which includes an additional null byte at the end (not counted in *length*). The data must not be modified in any way, unless the object was just created using `PyBytes_FromStringAndSize(NULL, size)`. It must not be deallocated. If *obj* is not a bytes object at all, `PyBytes_AsStringAndSize` returns `-1` and raises `TypeError`.
>
> 3.5 Previously, `TypeError` was raised when embedded null bytes were encountered in the bytes object.

> Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*; the caller will own the new reference. The reference to the old value of *bytes* will be "`stolen <steal>`". If the new object cannot be created, the old reference to *bytes* will still be "stolen", the value of **bytes* will be set to `NULL`, and the appropriate exception will be set.
>
> > [!NOTE] If *newpart* implements the buffer protocol, then the buffer must not be mutated while the new bytes object is being created.

> Create a new bytes object in **bytes* containing the contents of *newpart* appended to *bytes*. This version releases the `strong reference` to *newpart* (i.e.

decrements its reference count).

> [!NOTE] If *newpart* implements the buffer protocol, then the buffer must not be mutated while the new bytes object is being created.

> Similar to `sep.join(iterable)` in Python.
>
> *sep* must be Python `bytes` object. (Note that `PyUnicode_Join` accepts `NULL` separator and treats it as a space, whereas `PyBytes_Join` doesn't accept `NULL` separator.)
>
> *iterable* must be an iterable object yielding objects that implement the `buffer protocol <bufferobjects>`.
>
> On success, return a new `bytes` object. On error, set an exception and return `NULL`.
>
> 3.14
>
> > [!NOTE] If *iterable* objects implement the buffer protocol, then the buffers must not be mutated while the new bytes object is being created.

> Resize a bytes object. *newsize* will be the new length of the bytes object. You can think of it as creating a new bytes object and destroying the old one, only more efficiently.
>
> Pass the address of an existing bytes object as an lvalue (it may be written into), and the new size desired. On success, **bytes* holds the resized bytes object and `0` is returned; the address in **bytes* may differ from its input value. If the reallocation fails, the original bytes object at **bytes* is deallocated, **bytes* is set to `NULL`, `MemoryError` is set, and `-1` is returned.
>
> While bytes objects are usually immutable in Python, this special C API allows mutating a bytes object in-place. The returned bytes object can still be mutated

> using `PyBytes_AsString`; except if *newsize* is zero in which case it returns the immutable empty bytes string.
>
> 3.15 Use the `PyBytesWriter` API instead.

> “ Get the string representation of *bytes*. This function is currently used to implement `!bytes.__repr__` in Python.
>
> This function does not do type checking; it is undefined behavior to pass *bytes* as a non-bytes object or `NULL`.
>
> If *smartquotes* is true, the representation will use a double-quoted string instead of single-quoted string when single-quotes are present in *bytes*. For example, the byte string `'Python'` would be represented as `b"'Python'"` when *smartquotes* is true, or `b'\'Python\''` when it is false.
>
> On success, this function returns a `strong reference` to a `str` object containing the representation. On failure, this returns `NULL` with an exception set.

> “ Unescape a backslash-escaped string *s*. *s* must not be `NULL`. *len* must be the size of *s*.
>
> *errors* must be one of `"strict"`, `"replace"`, or `"ignore"`. If *errors* is `NULL`, then `"strict"` is used by default.
>
> On success, this function returns a `strong reference` to a Python `bytes` object containing the unescaped string. On failure, this function returns `NULL` with an exception set.
>
> 3.9 *unicode* and *recode_encoding* are now unused.

PyBytesWriter

The `PyBytesWriter` API can be used to create a Python `bytes` object.

3.15

>

> A bytes writer object.
>
> The API is **not thread safe**. To share a writer with multiple threads, a critical section or a lock is needed.
>
> The instance must be destroyed by `PyBytesWriter_Finish` on success, or `PyBytesWriter_Discard` on error.

Create, Finish, Discard

> Create a `PyBytesWriter` to write *size* bytes.
>
> If *size* is greater than zero, allocate *size* bytes, and set the writer size to *size*. The caller is responsible to write *size* bytes using `PyBytesWriter_GetData`. This function does not overallocate.
>
> On error, set an exception and return `NULL`.
>
> *size* must be positive or zero.

> Finish a `PyBytesWriter` created by `PyBytesWriter_Create`.
>
> On success, return a Python `bytes` object. On error, set an exception and return `NULL`.
>
> The writer instance is invalid after the call in any case. No API can be called on the writer after `PyBytesWriter_Finish`.

> Similar to `PyBytesWriter_Finish`, but resize the writer to *size* bytes before creating the `bytes` object.

> Similar to `PyBytesWriter_Finish`, but resize the writer using *buf* pointer before creating the `bytes` object.

> Set an exception and return `NULL` if *buf* pointer is outside the internal buffer bounds.
>
> Function pseudo-code:
>
> ```
> Py_ssize_t size = (char*)buf - (char*)PyBytesWriter_GetData(writer);
> return PyBytesWriter_FinishWithSize(writer, size);
> ```

> Discard a `PyBytesWriter` created by `PyBytesWriter_Create`.
>
> Do nothing if *writer* is `NULL`.
>
> The writer instance is invalid after the call. No API can be called on the writer after `PyBytesWriter_Discard`.

High-level API

> Grow the *writer* internal buffer by *size* bytes, write *size* bytes of *bytes* at the *writer* end, and add *size* to the *writer* size.
>
> If *size* is equal to `-1`, call `strlen(bytes)` to get the string length.
>
> On success, return `0`. On error, set an exception and return `-1`.

> Similar to `PyBytes_FromFormat`, but write the output directly at the writer end. Grow the writer internal buffer on demand. Then add the written size to the writer size.
>
> On success, return `0`. On error, set an exception and return `-1`.

Getters

>

Get the writer size.

The function does not invalidate pointers returned by `PyBytesWriter_GetData`.

The function cannot fail.

Get the writer data: start of the internal buffer.

The pointer remains valid until a `PyBytesWriter` function other than `PyBytesWriter_GetData` or `PyBytesWriter_GetSize` is called on *writer*.

The function cannot fail.

Low-level API

Resize the writer to *size* bytes. It can be used to enlarge or to shrink the writer. This function typically overallocates to achieve amortized performance when resizing multiple times.

Newly allocated bytes are left uninitialized.

On success, return `0`. On error, set an exception and return `-1`.

size must be positive or zero.

Resize the writer by adding *grow* bytes to the current writer size. This function typically overallocates to achieve amortized performance when resizing multiple times.

Newly allocated bytes are left uninitialized.

On success, return `0`. On error, set an exception and return `-1`.

grow can be negative to shrink the writer.

> Similar to `PyBytesWriter_Grow`, but update also the *buf* pointer.
>
> The *buf* pointer is moved if the internal buffer is moved in memory. The *buf* relative position within the internal buffer is left unchanged.
>
> On error, set an exception and return `NULL`.
>
> *buf* must not be `NULL`.
>
> Function pseudo-code:
>
> ```
> Py_ssize_t pos = (char*)buf - (char*)PyBytesWriter_GetData(writer);
> if (PyBytesWriter_Grow(writer, size) < 0) {
>     return NULL;
> }
> return (char*)PyBytesWriter_GetData(writer) + pos;
> ```

Example creating the string `b'abc'`

Example creating the bytes string `b"abc"`, with a fixed size of 3 bytes:

```
PyObject* create_abc(void)
{
    PyBytesWriter *writer = PyBytesWriter_Create(3);
    if (writer == NULL) {
        return NULL;
    }

    char *str = PyBytesWriter_GetData(writer);
    memcpy(str, "abc", 3);
    return PyBytesWriter_Finish(writer);
}
```

Variant using a pointer to compute the final size:

```
PyObject* create_abc_ptr(void)
{
    PyBytesWriter *writer = PyBytesWriter_Create(3);
```

```
    if (writer == NULL) {
        return NULL;
    }

    char *str = PyBytesWriter_GetData(writer);
    memcpy(str, "abc", 3);
    str += 3;
    return PyBytesWriter_FinishWithPointer(writer, str);
}
```

Update `PyBytes_FromStringAndSize()` code

Example of code using the soft deprecated `PyBytes_FromStringAndSize(NULL, size)` API:

```
int copy_bytes(char *dest, const char *src, Py_ssize_t num_bytes)
{
    ...
}

PyObject* create_string(const char *start, Py_ssize_t num_bytes)
{
    PyObject *result = PyBytes_FromStringAndSize(NULL, num_bytes);
    if (result == NULL) {
        return NULL;
    }
    if (copy_bytes(PyBytes_AS_STRING(result), start, num_bytes) < 0) {
        Py_CLEAR(result);
    }
    return result;
}
```

It can now be updated to:

```
PyObject* create_string(const char *start, Py_ssize_t num_bytes)
{
```

```
    PyBytesWriter *writer = PyBytesWriter_Create(num_bytes);
    if (writer == NULL) {
        return NULL;
    }
    if (copy_bytes(PyBytesWriter_GetData(writer), start, num_bytes) < 0) {
        PyBytesWriter_Discard(writer);
        return NULL;
    }
    return PyBytesWriter_Finish(writer);
}
```

Update `_PyBytes_Resize()` code

Example of code using the soft deprecated `_PyBytes_Resize()` API:

```
PyObject* create_string(Py_ssize_t size)
{
    PyObject *bytes = PyBytes_FromStringAndSize(NULL, size);
    if (bytes == NULL) {
        return NULL;
    }
    char *str = PyBytes_AS_STRING(bytes);

    // ... write data into str ...

    if (_PyBytes_Resize(&bytes, (str - PyBytes_AS_STRING(bytes)))) {
        return NULL;
    }
    return bytes;
}
```

It can now be updated to:

```
PyObject* create_string(Py_ssize_t size)
{
    PyBytesWriter *writer = PyBytesWriter_Create(size);
    if (writer == NULL) {
        return NULL;
```

```
    }
    char *str = PyBytesWriter_GetData(writer);

    // ... write data into str ...

    return PyBytesWriter_FinishWithPointer(writer, str);
}
```

[^1]: For integer specifiers (d, u, ld, lu, zd, zu, i, x): the 0-conversion flag has effect even when a precision is given.

[^2]: For integer specifiers (d, u, ld, lu, zd, zu, i, x): the 0-conversion flag has effect even when a precision is given.

[^3]: For integer specifiers (d, u, ld, lu, zd, zu, i, x): the 0-conversion flag has effect even when a precision is given.

[^4]: For integer specifiers (d, u, ld, lu, zd, zu, i, x): the 0-conversion flag has effect even when a precision is given.

[^5]: For integer specifiers (d, u, ld, lu, zd, zu, i, x): the 0-conversion flag has effect even when a precision is given.

[^6]: For integer specifiers (d, u, ld, lu, zd, zu, i, x): the 0-conversion flag has effect even when a precision is given.

[^7]: For integer specifiers (d, u, ld, lu, zd, zu, i, x): the 0-conversion flag has effect even when a precision is given.

[^8]: For integer specifiers (d, u, ld, lu, zd, zu, i, x): the 0-conversion flag has effect even when a precision is given.

□□□ #2
□ □□□ □□□ 2026-10-31 16:49:47 UTC
□ □□□ □□□ 2026-10-31 16:49:50 UTC