Parsing arguments and building values

These functions are useful when creating your own extensions functions and methods. Additional information and examples are available in Extending and Embedding the Python Interpreter.

The first three of these functions described, `PyArg_ParseTuple()`, `PyArg_ParseTupleAndKeywords()`, and `PyArg_Parse()`, all use *format strings* which are used to tell the function about the expected arguments. The format strings use the same syntax for each of these functions.

Parsing arguments

A format string consists of zero or more “format units.” A format unit describes one Python object; it is usually a single character or a parenthesized sequence of format units. With a few exceptions, a format unit that is not a parenthesized sequence normally corresponds to a single address argument to these functions. In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that matches the format unit; and the entry in [square] brackets is the type of the C variable(s) whose address should be passed.

Strings and buffers

These formats allow accessing an object as a contiguous chunk of memory. You don’t have to provide raw storage for the returned unicode or bytes area.

In general, when a format sets a pointer to a buffer, the buffer is managed by the corresponding Python object, and the buffer shares the lifetime of this object. You won’t have to release any memory yourself. The only exceptions are `es`, `es#`, `et` and `et#`.

However, when a `Py_buffer` structure gets filled, the underlying buffer is locked so that the caller can subsequently use the buffer even inside a `Py_BEGIN_ALLOW_THREADS` block without the risk of mutable data being resized or destroyed. As a result, **you have to call** `PyBuffer_Release()` after you have finished processing the data (or in any early abort case).

Unless otherwise stated, buffers are not NUL-terminated.

Some formats require a read-only bytes-like object, and set a pointer instead of a buffer structure. They work by checking that the object’s

`PyBufferProcs.bf_releasebuffer` field is *NULL*, which disallows mutable objects such as `bytearray`.

> **Note:** For all `#` variants of formats (`s#`, `y#`, etc.), the type of the length argument (int or `Py_ssize_t`) is controlled by defining the macro `PY_SSIZE_T_CLEAN` before including `Python.h`. If the macro was defined, length is a `Py_ssize_t` rather than an `int`. This behavior will change in a future Python version to only support `Py_ssize_t` and drop `int` support. It is best to always define `PY_SSIZE_T_CLEAN`.

`s` (`str`) [const char *]
: Convert a Unicode object to a C pointer to a character string. A pointer to an existing string is stored in the character pointer variable whose address you pass. The C string is NUL-terminated. The Python string must not contain embedded null code points; if it does, a `ValueError` exception is raised. Unicode objects are converted to C strings using `'utf-8'` encoding. If this conversion fails, a `UnicodeError` is raised.

  > **Note:** This format does not accept bytes-like objects. If you want to accept filesystem paths and convert them to C character strings, it is preferable to use the `O&` format with `PyUnicode_FSConverter()` as *converter*.

  Changed in version 3.5: Previously, `TypeError` was raised when embedded null code points were encountered in the Python string.

`s*` (`str` or bytes-like object) [Py_buffer]
: This format accepts Unicode objects as well as bytes-like objects. It fills a `Py_buffer` structure provided by the caller. In this case the resulting C string may contain embedded NUL bytes. Unicode objects are converted to C strings using `'utf-8'` encoding.

`s#` (`str`, read-only bytes-like object) [const char *, int or `Py_ssize_t`]
: Like `s*`, except that it doesn't accept mutable objects. The result is stored into two C variables, the first one a pointer to a C string, the second one its length. The string may contain embedded null bytes. Unicode objects are converted to C strings using `'utf-8'` encoding.

`z` (`str` or `None`) [const char *]
: Like `s`, but the Python object may also be `None`, in which case the C pointer is set to *NULL*.

`z*` (`str`, bytes-like object or `None`) [Py_buffer]
: Like `s*`, but the Python object may also be `None`, in which case the `buf` member of the `Py_buffer` structure is set to *NULL*.

`z#` (`str`, read-only bytes-like object or `None`) [const char *, int]

Like `s#`, but the Python object may also be `None`, in which case the C pointer is set to *NULL*.

`y` (read-only bytes-like object) [const char *]
: This format converts a bytes-like object to a C pointer to a character string; it does not accept Unicode objects. The bytes buffer must not contain embedded null bytes; if it does, a `ValueError` exception is raised.

  Changed in version 3.5: Previously, `TypeError` was raised when embedded null bytes were encountered in the bytes buffer.

`y*` (bytes-like object) [Py_buffer]
: This variant on `s*` doesn't accept Unicode objects, only bytes-like objects. **This is the recommended way to accept binary data.**

`y#` (read-only bytes-like object) [const char *, int]
: This variant on `s#` doesn't accept Unicode objects, only bytes-like objects.

`S` (`bytes`) [PyBytesObject *]
: Requires that the Python object is a `bytes` object, without attempting any conversion. Raises `TypeError` if the object is not a bytes object. The C variable may also be declared as `PyObject*`.

`Y` (`bytearray`) [PyByteArrayObject *]
: Requires that the Python object is a `bytearray` object, without attempting any conversion. Raises `TypeError` if the object is not a `bytearray` object. The C variable may also be declared as `PyObject*`.

`u` (`str`) [Py_UNICODE *]
: Convert a Python Unicode object to a C pointer to a NUL-terminated buffer of Unicode characters. You must pass the address of a `Py_UNICODE` pointer variable, which will be filled with the pointer to an existing Unicode buffer. Please note that the width of a `Py_UNICODE` character depends on compilation options (it is either 16 or 32 bits). The Python string must not contain embedded null code points; if it does, a `ValueError` exception is raised.

  Changed in version 3.5: Previously, `TypeError` was raised when embedded null code points were encountered in the Python string.

  Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsWideCharString()`.

`u#` (`str`) [Py_UNICODE *, int]
: This variant on `u` stores into two C variables, the first one a pointer to a Unicode data buffer, the second one its length. This variant allows null code points.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsWideCharString()`.

`Z` (`str` or `None`) [Py_UNICODE *]
Like `u`, but the Python object may also be `None`, in which case the `Py_UNICODE` pointer is set to *NULL*.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsWideCharString()`.

`Z#` (`str` or `None`) [Py_UNICODE *, int]
Like `u#`, but the Python object may also be `None`, in which case the `Py_UNICODE` pointer is set to *NULL*.

Deprecated since version 3.3, will be removed in version 4.0: Part of the old-style `Py_UNICODE` API; please migrate to using `PyUnicode_AsWideCharString()`.

`U` (`str`) [PyObject *]
Requires that the Python object is a Unicode object, without attempting any conversion. Raises `TypeError` if the object is not a Unicode object. The C variable may also be declared as `PyObject*`.

`w*` (read-write bytes-like object) [Py_buffer]
This format accepts any object which implements the read-write buffer interface. It fills a `Py_buffer` structure provided by the caller. The buffer may contain embedded null bytes. The caller have to call `PyBuffer_Release()` when it is done with the buffer.

`es` (`str`) [const char *encoding, char **buffer]
This variant on `s` is used for encoding Unicode into a character buffer. It only works for encoded data without embedded NUL bytes.

This format requires two arguments. The first is only used as input, and must be a `const char*` which points to the name of an encoding as a NUL-terminated string, or *NULL*, in which case `'utf-8'` encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a `char**`; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument.

`PyArg_ParseTuple()` will allocate a buffer of the needed size, copy the encoded data into this buffer and adjust **buffer* to reference the newly allocated stor-

age. The caller is responsible for calling `PyMem_Free()` to free the allocated buffer after use.

`et` (`str`, `bytes` or `bytearray`) [const char *encoding, char **buffer]
Same as `es` except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

`es#` (`str`) [const char *encoding, char **buffer, int *buffer_length]
This variant on `s#` is used for encoding Unicode into a character buffer. Unlike the `es` format, this variant allows input data which contains NUL characters.

It requires three arguments. The first is only used as input, and must be a `const char*` which points to the name of an encoding as a NUL-terminated string, or *NULL*, in which case `'utf-8'` encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a `char**`; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument. The third argument must be a pointer to an integer; the referenced integer will be set to the number of bytes in the output buffer.

There are two modes of operation:

If **buffer* points a *NULL* pointer, the function will allocate a buffer of the needed size, copy the encoded data into this buffer and set **buffer* to reference the newly allocated storage. The caller is responsible for calling `PyMem_Free()` to free the allocated buffer after usage.

If **buffer* points to a non-*NULL* pointer (an already allocated buffer), `PyArg_ParseTuple()` will use this location as the buffer and interpret the initial value of **buffer_length* as the buffer size. It will then copy the encoded data into the buffer and NUL-terminate it. If the buffer is not large enough, a `ValueError` will be set.

In both cases, **buffer_length* is set to the length of the encoded data without the trailing NUL byte.

`et#` (`str`, `bytes` or `bytearray`) [const char *encoding, char **buffer, int *buffer_length]
Same as `es#` except that byte string objects are passed through without recoding them. Instead, the implementation assumes that the byte string object uses the encoding passed in as parameter.

Numbers

`b` (`int`) [unsigned char]

Convert a nonnegative Python integer to an unsigned tiny int, stored in a C `unsigned char`.

`B` (`int`) [unsigned char]
: Convert a Python integer to a tiny int without overflow checking, stored in a C `unsigned char`.

`h` (`int`) [short int]
: Convert a Python integer to a C `short int`.

`H` (`int`) [unsigned short int]
: Convert a Python integer to a C `unsigned short int`, without overflow checking.

`i` (`int`) [int]
: Convert a Python integer to a plain C `int`.

`I` (`int`) [unsigned int]
: Convert a Python integer to a C `unsigned int`, without overflow checking.

`l` (`int`) [long int]
: Convert a Python integer to a C `long int`.

`k` (`int`) [unsigned long]
: Convert a Python integer to a C `unsigned long` without overflow checking.

`L` (`int`) [long long]
: Convert a Python integer to a C `long long`.

`K` (`int`) [unsigned long long]
: Convert a Python integer to a C `unsigned long long` without overflow checking.

`n` (`int`) [Py_ssize_t]
: Convert a Python integer to a C `Py_ssize_t`.

`c` (`bytes` or `bytearray` of length 1) [char]
: Convert a Python byte, represented as a `bytes` or `bytearray` object of length 1, to a C `char`.

  Changed in version 3.3: Allow `bytearray` objects.

`C` (`str` of length 1) [int]
: Convert a Python character, represented as a `str` object of length 1, to a C `int`.

`f` (`float`) [float]
: Convert a Python floating point number to a C `float`.

`d` (`float`) [double]
: Convert a Python floating point number to a C `double`.

`D` (`complex`) [Py_complex]

Convert a Python complex number to a C `Py_complex` structure.

Other objects

`O` (object) [PyObject *]
: Store a Python object (without any conversion) in a C object pointer. The C program thus receives the actual object that was passed. The object's reference count is not increased. The pointer stored is not *NULL*.

`O!` (object) [*typeobject*, PyObject *]
: Store a Python object in a C object pointer. This is similar to `O`, but takes two C arguments: the first is the address of a Python type object, the second is the address of the C variable (of type `PyObject*`) into which the object pointer is stored. If the Python object does not have the required type, `TypeError` is raised.

`O&` (object) [*converter*, *anything*]
: Convert a Python object to a C variable through a *converter* function. This takes two arguments: the first is a function, the second is the address of a C variable (of arbitrary type), converted to `void *`. The *converter* function in turn is called as follows:

```
status = converter(object, address);
```

: where *object* is the Python object to be converted and *address* is the `void*` argument that was passed to the `PyArg_Parse*()` function. The returned *status* should be `1` for a successful conversion and `0` if the conversion has failed. When the conversion fails, the *converter* function should raise an exception and leave the content of *address* unmodified.

: If the *converter* returns `Py_CLEANUP_SUPPORTED`, it may get called a second time if the argument parsing eventually fails, giving the converter a chance to release any memory that it had already allocated. In this second call, the *object* parameter will be NULL; *address* will have the same value as in the original call.

: *Changed in version 3.1:* `Py_CLEANUP_SUPPORTED` was added.

`p` (`bool`) [int]
: Tests the value passed in for truth (a boolean **p**redicate) and converts the result to its equivalent C true/false integer value. Sets the int to `1` if the expression was true and `0` if it was false. This accepts any valid Python value. See Truth Value Testing for more information about how Python tests values for truth.

: *New in version 3.3.*

`(items)` (`tuple`) [*matching-items*]

The object must be a Python sequence whose length is the number of format units in *items*. The C arguments must correspond to the individual format units in *items*. Format units for sequences may be nested.

It is possible to pass "long" integers (integers whose value exceeds the platform's `LONG_MAX`) however no proper range checking is done — the most significant bits are silently truncated when the receiving field is too small to receive the value (actually, the semantics are inherited from downcasts in C — your mileage may vary).

A few other characters have a meaning in a format string. These may not occur inside nested parentheses. They are:

`|`

Indicates that the remaining arguments in the Python argument list are optional. The C variables corresponding to optional arguments should be initialized to their default value — when an optional argument is not specified, `PyArg_ParseTuple()` does not touch the contents of the corresponding C variable(s).

`$`

`PyArg_ParseTupleAndKeywords()` only: Indicates that the remaining arguments in the Python argument list are keyword-only. Currently, all keyword-only arguments must also be optional arguments, so `|` must always be specified before `$` in the format string.

New in version 3.3.

`:`

The list of format units ends here; the string after the colon is used as the function name in error messages (the "associated value" of the exception that `PyArg_ParseTuple()` raises).

`;`

The list of format units ends here; the string after the semicolon is used as the error message *instead* of the default error message. `:` and `;` mutually exclude each other.

Note that any Python object references which are provided to the caller are *borrowed* references; do not decrement their reference count!

Additional arguments passed to these functions must be addresses of variables whose type is determined by the format string; these are used to store values from the input tuple. There are a few cases, as described in the list of format units above, where these parameters are used as input values; they should match what is specified for the corresponding format unit in that case.

For the conversion to succeed, the *arg* object must match the format and the format must be exhausted. On success, the `PyArg_Parse*()` functions return true, otherwise they return false and raise an appropriate exception. When the `PyArg_Parse*()` functions fail due to conversion failure in one of the format units, the variables at the addresses corresponding to that and the following format units are left untouched.

API Functions

int **PyArg_ParseTuple**(PyObject **args*, const char **format*, ...)

Parse the parameters of a function that takes only positional parameters into local variables. Returns true on success; on failure, it returns false and raises the appropriate exception.

int **PyArg_VaParse**(PyObject **args*, const char **format*, va_list *vargs*)

Identical to `PyArg_ParseTuple()`, except that it accepts a va_list rather than a variable number of arguments.

int **PyArg_ParseTupleAndKeywords**(PyObject **args*, PyObject **kw*, const char **format*, char **keywords[]*, ...)

Parse the parameters of a function that takes both positional and keyword parameters into local variables. The *keywords* argument is a *NULL*-terminated array of keyword parameter names. Empty names denote positional-only parameters. Returns true on success; on failure, it returns false and raises the appropriate exception.

Changed in version 3.6: Added support for positional-only parameters.

int **PyArg_VaParseTupleAndKeywords**(PyObject **args*, PyObject **kw*, const char **format*, char **keywords[]*, va_list *vargs*)

Identical to `PyArg_ParseTupleAndKeywords()`, except that it accepts a va_list rather than a variable number of arguments.

int **PyArg_ValidateKeywordArguments**(PyObject *)

Ensure that the keys in the keywords argument dictionary are strings. This is only needed if `PyArg_ParseTupleAndKeywords()` is not used, since the latter already does this check.

New in version 3.2.

int **PyArg_Parse**(PyObject **args*, const char **format*, ...)

Function used to deconstruct the argument lists of “old-style” functions — these are functions which use the `METH_OLDARGS` parameter parsing method, which has been removed in Python 3. This is not recommended for use in parameter

parsing in new code, and most code in the standard interpreter has been modified to no longer use this for that purpose. It does remain a convenient way to decompose other tuples, however, and may continue to be used for that purpose.

int **PyArg_UnpackTuple**(PyObject **args*, const char **name*, Py_ssize_t *min*, Py_ssize_t *max*, ...)

A simpler form of parameter retrieval which does not use a format string to specify the types of the arguments. Functions which use this method to retrieve their parameters should be declared as `METH_VARARGS` in function or method tables. The tuple containing the actual parameters should be passed as *args*; it must actually be a tuple. The length of the tuple must be at least *min* and no more than *max*; *min* and *max* may be equal. Additional arguments must be passed to the function, each of which should be a pointer to a `PyObject*` variable; these will be filled in with the values from *args*; they will contain borrowed references. The variables which correspond to optional parameters not given by *args* will not be filled in; these should be initialized by the caller. This function returns true on success and false if *args* is not a tuple or contains the wrong number of elements; an exception will be set if there was a failure.

This is an example of the use of this function, taken from the sources for the `_weakref` helper module for weak references:

```
static PyObject *
weakref_ref(PyObject *self, PyObject *args)
{
    PyObject *object;
    PyObject *callback = NULL;
    PyObject *result = NULL;

    if (PyArg_UnpackTuple(args, "ref", 1, 2, &object, &callback))
        result = PyWeakref_NewRef(object, callback);
    }
    return result;
}
```

The call to `PyArg_UnpackTuple()` in this example is entirely equivalent to this call to `PyArg_ParseTuple()`:

```
PyArg_ParseTuple(args, "O|O:ref", &object, &callback)
```

Building values

PyObject* **Py_BuildValue**(const char **format*, ...)

Return value: New reference.
Create a new value based on a format string similar to those accepted by the `PyArg_Parse*()` family of functions and a sequence of values. Returns the value or *NULL* in the case of an error; an exception will be raised if *NULL* is returned.

`Py_BuildValue()` does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

When memory buffers are passed as parameters to supply data to build objects, as for the `s` and `s#` formats, the required data is copied. Buffers provided by the caller are never referenced by the objects created by `Py_BuildValue()`. In other words, if your code invokes `malloc()` and passes the allocated memory to `Py_BuildValue()`, your code is responsible for calling `free()` for that memory once `Py_BuildValue()` returns.

In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that the format unit will return; and the entry in [square] brackets is the type of the C value(s) to be passed.

The characters space, tab, colon and comma are ignored in format strings (but not within format units such as `s#`). This can be used to make long format strings a tad more readable.

`s` (`str` or `None`) [char *]
: Convert a null-terminated C string to a Python `str` object using `'utf-8'` encoding. If the C string pointer is *NULL*, `None` is used.

`s#` (`str` or `None`) [char *, int]
: Convert a C string and its length to a Python `str` object using `'utf-8'` encoding. If the C string pointer is *NULL*, the length is ignored and `None` is returned.

`y` (`bytes`) [char *]
: This converts a C string to a Python `bytes` object. If the C string pointer is *NULL*, `None` is returned.

`y#` (`bytes`) [char *, int]
: This converts a C string and its lengths to a Python object. If the C string pointer is *NULL*, `None` is returned.

`z` (`str` or `None`) [char *]
: Same as `s`.

`z#` (`str` or `None`) [char *, int]

Same as `s#`.

`u` (`str`) [wchar_t *]

Convert a null-terminated `wchar_t` buffer of Unicode (UTF-16 or UCS-4) data to a Python Unicode object. If the Unicode buffer pointer is *NULL*, `None` is returned.

`u#` (`str`) [wchar_t *, int]

Convert a Unicode (UTF-16 or UCS-4) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is *NULL*, the length is ignored and `None` is returned.

`U` (`str` or `None`) [char *]

Same as `s`.

`U#` (`str` or `None`) [char *, int]

Same as `s#`.

`i` (`int`) [int]

Convert a plain C `int` to a Python integer object.

`b` (`int`) [char]

Convert a plain C `char` to a Python integer object.

`h` (`int`) [short int]

Convert a plain C `short int` to a Python integer object.

`l` (`int`) [long int]

Convert a C `long int` to a Python integer object.

`B` (`int`) [unsigned char]

Convert a C `unsigned char` to a Python integer object.

`H` (`int`) [unsigned short int]

Convert a C `unsigned short int` to a Python integer object.

`I` (`int`) [unsigned int]

Convert a C `unsigned int` to a Python integer object.

`k` (`int`) [unsigned long]

Convert a C `unsigned long` to a Python integer object.

`L` (`int`) [long long]

Convert a C `long long` to a Python integer object.

`K` (`int`) [unsigned long long]

Convert a C `unsigned long long` to a Python integer object.

`n` (`int`) [Py_ssize_t]

Convert a C `Py_ssize_t` to a Python integer.

`c` (`bytes` of length 1) [char]
: Convert a C `int` representing a byte to a Python `bytes` object of length 1.

`C` (`str` of length 1) [int]
: Convert a C `int` representing a character to Python `str` object of length 1.

`d` (`float`) [double]
: Convert a C `double` to a Python floating point number.

`f` (`float`) [float]
: Convert a C `float` to a Python floating point number.

`D` (`complex`) [Py_complex *]
: Convert a C `Py_complex` structure to a Python complex number.

`O` (object) [PyObject *]
: Pass a Python object untouched (except for its reference count, which is incremented by one). If the object passed in is a *NULL* pointer, it is assumed that this was caused because the call producing the argument found an error and set an exception. Therefore, `Py_BuildValue()` will return *NULL* but won't raise an exception. If no exception has been raised yet, `SystemError` is set.

`S` (object) [PyObject *]
: Same as `O`.

`N` (object) [PyObject *]
: Same as `O`, except it doesn't increment the reference count on the object. Useful when the object is created by a call to an object constructor in the argument list.

`O&` (object) [*converter*, *anything*]
: Convert *anything* to a Python object through a *converter* function. The function is called with *anything* (which should be compatible with `void *`) as its argument and should return a "new" Python object, or *NULL* if an error occurred.

`(items)` (`tuple`) [*matching-items*]
: Convert a sequence of C values to a Python tuple with the same number of items.

`[items]` (`list`) [*matching-items*]
: Convert a sequence of C values to a Python list with the same number of items.

`{items}` (`dict`) [*matching-items*]

Convert a sequence of C values to a Python dictionary. Each pair of consecutive C values adds one item to the dictionary, serving as key and value, respectively.

If there is an error in the format string, the `SystemError` exception is set and *NULL* returned.

PyObject* **Py_VaBuildValue**(const char **format*, va_list *vargs*)

Identical to `Py_BuildValue()`, except that it accepts a va_list rather than a variable number of arguments.