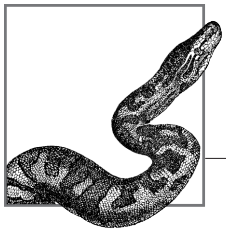

25

Extending and Embedding Classic Python

CPython runs on a portable, C-coded virtual machine. Python's built-in objects—such as numbers, sequences, dictionaries, sets, and files—are coded in C, as are several modules in Python's standard library. Modern platforms support dynamically loaded libraries, with file extensions such as *.dll* on Windows, *.so* on Linux, and *.dylib* on Mac: building Python produces such binary files. You can code your own extension modules for Python in C (or any language that can produce C-callable libraries), using the Python C API covered in this chapter. With this API, you can produce and deploy dynamic libraries that Python scripts and interactive sessions can later use with the **`import`** statement, covered in "The import Statement" in Chapter 7.

Extending Python means building modules that Python code can **`import`** to access the features the modules supply. *Embedding* Python means executing Python code from an application coded in another language. For such execution to be useful, Python code must in turn be able to access some of your application's functionality. In practice, therefore, embedding implies some extending, as well as a few embedding-specific operations. The three main reasons for wishing to extend Python can be summarized as follows:

- Reimplementing some functionality (originally coded in Python) in a lower-level language, intending to improve performance
- Letting Python code access some existing functionality supplied by libraries coded in (or, at any rate, callable from) lower-level languages
- Letting Python code access some existing functionality of an application that is in the process of embedding Python as the application's scripting language

Embedding and extending are covered in Python's online documentation; there, you can find an in-depth tutorial and an extensive reference manual. Many details are best studied in Python's extensively documented C sources. Download Python's source distribution and study the sources of Python's core, C-coded extension modules, and the example extensions supplied for this purpose.

This chapter covers the basics of extending and embedding Python with C. It also mentions, but does not cover in depth, other ways to extend Python. Do notice that, as the online docs put it, several excellent third-party modules (such as Cython, covered in "Cython" on page 38, CFFI, mentioned in "Extending Python Without Python's C API" on page 36, and Numba, CLIF, `pybind11`, and SWIG, not covered in this book) "offer both simpler and more sophisticated approaches to creating C and C++ extensions for Python."

This Chapter Assumes Some Knowledge of C

Although we include some non-C extension options at the end of this chapter, to extend or embed Python using the C API (or most of the above-mentioned third-party modules), you must know the C and/or C++ programming languages. We do not cover C and C++ in this book, but there are many print and online resources that you can consult to learn them (make sure not to confuse C with its subtly different close relative C++: they are different, although similar, languages). In most of the rest of this chapter, we assume that you have at least some knowledge of C.

Extending Python with Python's C API

A Python extension module named *x* resides in a dynamic library with the same filename (*x.pyd* on Windows; *x.so* on most Unix-like platforms) in an appropriate directory (often the *site-packages* subdirectory of the Python library directory). You generally build the *x* extension module from a C source file *x.c* (or, more conventionally, *xmodule.c*) whose overall structure is:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>/* omitted: the body of the x module */

PyMODINIT_FUNC
PyInit_x(void)
{
    /* omitted: code that creates and initializes the x module */
}
```

When you have built the extension module and installed it somewhere in Python's path, the Python statement **`import`** *x* loads the dynamic library, then locates and calls the module initialization function, which must do all that is needed to initialize the module object named *x*.

Building and Installing C-Coded Python Extensions

To build and install a C-coded Python extension module, it's simplest and most productive to use the third-party module `setuptools` (a variant of the deprecated standard library module `distutils`; just **`pip install setuptools`** to install `setuptools`). In the same directory as *x.c*, place a file named *setup.py* that contains the following statements:

```
from setuptools import setup, Extension
setup(name='x', ext_modules=[Extension('x',sources=['x.c'])])
```

From a shell prompt in this directory, you can now run:

```
C:\the_dir> python setup.py install
```

to build and install the module, making it usable in your Python installation. (You'll normally want to do this within a virtual environment, with **`venv`**, as covered in "Python Environments" in Chapter 7, to avoid affecting the global state of your Python installation; however, for simplicity, we omit that important step in this chapter.)

As discussed in Chapter 24, `setuptools` performs compilation and linking, with the right compiler and linker commands and flags, and copies the resulting dynamic library into the right directory, based on your Python installation (depending on that installation's details, you may need to have administrator or superuser privileges for this; for example, on macOS or Linux, you may run **`sudo python setup.py install`**, although using **`venv`** instead is usually best). Your Python code (running in the appropriate virtual environment, if needed) can then access the resulting module with the statement **`import`** *x*.

What C compiler do you need?

To compile C-coded extensions to Python, you normally need the same C compiler used to build the Python version you want to extend. For most Linux platforms, this means the free `gcc` compiler that normally comes with your platform or can be freely downloaded for it. (You might also consider `clang`, widely reputed to offer better error messages.) For Mac users, `gcc` (actually a frontend to `clang`) comes with Apple's free XCode (aka "Developer Tools") IDE, which you can download and install from Apple's App Store (you only need the "command line" subset of XCode).

For Windows, you ideally need the Microsoft product known as Visual Studio 14.*x*, though other versions of Visual Studio might also work (see the Python wiki for details).

Overview of C-Coded Python Extension Modules

Your C function `PyInit_`*x* generally has the following overall structure:

```
PyMODINIT_FUNC
PyInit_x(void)
{
```

```
    PyObject* m = PyModule_Create(&x_module);
    // x_module is the instance of struct PyModuleDef describing the
    // module and in particular connecting to its methods (functions)

    // Then: calls to PyModule_AddObject(m, "somename", someobj)
    // to add exceptions or other classes, and module constants.
    // And at last, when all is done:
    return m;
}
```

More details are provided in "The Initialization Module" on page 6. `x_module` is a `struct` like:[1]

```
static struct PyModuleDef x_module = {
   PyModuleDef_HEAD_INIT,
   "x",      /* the name of the module */
   x_doc,    /* the module's docstring, may be NULL */
   -1,       /* size of per-subinterpreter state of the module, or -1
                if the module keeps state in global variables, and thus
                the module does not support subinterpreters */
   x_methods /* array of the module's method definitions */
};
```

and within it, *x_methods* is an array of `PyMethodDef` `structs`. Each `PyMethodDef` `struct` in the *x_methods* array describes a C function that your module *x* makes available to Python code that imports *x*. Each such C function has the following overall structure:

```
static PyObject*func_with_named_args(PyObject* self,
                                     PyObject* args, PyObject* kwds)
{
    /* omitted: body of function, accessing arguments via the Python C
       API function PyArg_ParseTupleAndKeywords, returning a PyObject*
       result, NULL for errors */
}
```

or a slightly simpler variant:

```
static PyObject*func_with_positional_args_only(PyObject* self,
                                               PyObject* args)
{
    /* omitted: body of function, accessing arguments via the Python C
       API function PyArg_ParseTuple, returning a PyObject* result,
       NULL for errors */
}
```

How C-coded functions access arguments passed by Python code is covered in "Accessing Arguments" on page 9. How such functions build Python objects is

1 Other fields can be specified if the module uses the advanced multiphase initialization approach, which we don't cover in this book; with the single-phase initialization that we cover here, use no other fields.

covered in "Creating Python Values" on page 14, and how they raise or propagate exceptions back to the Python code that called them is covered in Chapter 6. When your module defines new Python types, aka classes, your C code defines one or more instances of the `struct PyTypeObject`. This subject is covered in "Defining New Types" on page 28.

A simple example using all these concepts is shown in "A Simple Extension Example" on page 25. A toy "Hello World" example module could be as simple as:

```
#include <Python.h>
static PyObject*
hello(PyObject* self)
{
    return Py_BuildValue("s", "Hello, Python extensions world!");
}
static char hello_docs[] =
    "hello(): return a popular greeting phrase\n";
static PyMethodDef hello_funcs[] = {
    {"helloworld", (PyCFunction)hello, METH_NOARGS, hello_docs},
    {NULL}
};
static struct PyModuleDef hello_module = {
   PyModuleDef_HEAD_INIT,
   "hello",
   hello_docs,
   -1,
   hello_funcs
};

PyMODINIT_FUNC
PyInit_hello(void)
{
   return PyModule_Create(&hello_module);
}
```

The C string passed to `Py_BuildValue` is encoded in UTF-8, and the result is a Python `str` instance. Save this as *hello.c* and build it through a *setup.py* script with `setuptools`, as follows:

```
from setuptools import setup, Extension
setup(name='hello',
      ext_modules=[Extension('hello',sources=['hello.c'])])
```

After you have run **`python setup.py install`**, you can use the newly installed module—for example, from a Python interactive session—as follows:

```
>>> import hello
>>> print(hello.helloworld())
Hello, Python extensions world!
>>>
```

Return Values of Python's C API Functions

All functions in the Python C API return either an `int` or a `PyObject*`. Most functions returning an `int` return `0` in case of success and `-1` to indicate errors. Some functions return results that are true or false: these functions return `0` to indicate false and an integer not equal to `0` to indicate true, and never indicate errors. Functions returning `PyObject*` return `NULL` in case of errors. See Chapter 6 for more details on handling and raising errors.

The Initialization Module

The `PyInit_x` function must contain, at a minimum, a call to the function `PyModule_Create` with a single parameter, the address of the `struct PyModuleDef` that defines the module's details. In addition, `PyInit_x` may contain one or more calls to the functions listed in Table 25-1, all returning `-1` on error and `0` on success.

Table 25-1. Functions that can be called by `PyInit_x`

`PyModule_AddIntConstant`	`int PyModule_AddIntConstant(PyObject* module, char* name, long value)` Adds to the module *module* an attribute named *name* with integer value *value*.
`PyModule_AddObject`	`int PyModule_AddObject(PyObject* module, char* name, PyObject* value)` Adds to the module *module* an attribute named *name* with the value *value* and steals a reference to *value*, as covered in "Reference Counting" on page 7.
`PyModule_AddStringConstant`	`int PyModule_AddStringConstant(PyObject* module, char* name, char* value)` Adds to the module *module* an attribute named *name* with the string value *value* (encoded in UTF-8, and `0`-terminated as usual in C code).

Sometimes, as part of the job of initializing your new module, you need to access something within another module. If you were coding in Python, you would just **`import`** *`othermodule`*, then access attributes of *`othermodule`*. When coding a Python extension in C, it can be almost as simple: call `PyImport_Import` for the other module, then `PyModule_GetDict` to get the other module's `__dict__`. These functions are described in Table 25-2.

Table 25-2. Functions for accessing items in other modules

`PyImport_Import`	`PyObject* PyImport_Import(PyObject* name)` Imports the module named in Python string object *name* and returns a new reference to the module object, like Python's `__import__(name)`. `PyImport_Import` is the highest-level, simplest, and most often used way to import a module.

`PyModule_GetDict` `PyObject* PyModule_GetDict(PyObject* module)`
Returns a borrowed reference (see "Reference Counting" on page 7) to the dictionary of the module *module*.

The PyMethodDef struct

To add functions to a module (or nonspecial methods to new types, as covered in "Defining New Types" on page 28), you must describe the functions or methods in an array of `PyMethodDef structs`, and terminate the array with a *sentinel* (i.e., a `struct` whose fields are all `0` or `NULL`). `PyMethodDef` is defined as follows:

```
typedef struct {
    char* ml_name;          /* Python name of function or method */
    PyCFunction ml_meth;    /* pointer to C function implementing it */
    int ml_flags;           /* flag describing how to pass arguments */
    char* ml_doc;           /* docstring for the function or method */
} PyMethodDef
```

You must cast the second field to `(PyCFunction)` unless the C function's signature is exactly `PyObject* function(PyObject* self, PyObject* args)`, which is the `typedef` for `PyCFunction`. This signature is correct when `ml_flags` is `METH_O`, which indicates a function that accepts a single argument, or `METH_VARARGS`, which indicates a function that accepts positional arguments. For `METH_O`, *args* is the only argument. For `METH_VARARGS`, *args* is a tuple of all arguments, to be parsed with the C API function `PyArg_ParseTuple`. However, `ml_flags` can also be `METH_NOARGS`, which indicates a function that accepts no arguments, or `METH_KEYWORDS`, which indicates a function that accepts both positional and named arguments. For `METH_NOARGS`, the signature is `PyObject* function(PyObject* self)`, without further arguments. For `METH_KEYWORDS`, the signature is:

```
PyObject* function(PyObject* self, PyObject* args, PyObject* kwds)
```

args is the tuple of positional arguments, and *kwds* is the dictionary of named arguments; both are parsed with the C API function `PyArg_ParseTupleAndKeywords`. In these cases, you do need to explicitly cast the second field to `(PyCFunction)`.

When a C-coded function implements a module's function, the *self* parameter of the C function is `NULL`, for any value of the `ml_flags` field. When a C-coded function implements a nonspecial method of an extension type, the *self* parameter points to the instance on which the method is being called.

Reference Counting

Python objects live on the heap, and C code sees them as pointers to instances of type `PyObject*`. Each `PyObject` counts how many references to itself are outstanding and destroys itself when the number of references goes down to zero. To make this possible, your code must use Python-supplied macros: `Py_INCREF` to add a reference to a Python object, and `Py_DECREF` to abandon a reference to a Python object. The `Py_XINCREF` and `Py_XDECREF` macros are like `Py_INCREF` and `Py_DECREF`,

but you may also use them innocuously on a null pointer. The test for a nonnull pointer is implicitly performed inside the `Py_XINCREF` and `Py_XDECREF` macros, saving you the little bother of writing out that test explicitly when you don't know for sure whether the pointer might be null.

A `PyObject*` *p*, which your code receives by calling or being called by other functions, is known as a *new reference* when the code that supplies *p* has already called `Py_INCREF` on your behalf. Otherwise, it is known as a *borrowed reference*. Your code is said to *own* new references it holds, but not borrowed ones. You can call `Py_INCREF` on a borrowed reference to make it into a reference that you own; you must do this when you will use the reference across calls to code that might cause the count of the reference you borrowed to be decremented. You must *always* call `Py_DECREF` before abandoning or overwriting references that you own, but *never* on references you don't own. Therefore, understanding which interactions transfer reference ownership and which ones rely on reference borrowing is absolutely crucial. For most functions in the C API, and for *all* functions that you write and Python calls, two general rules apply:

- `PyObject*` arguments are borrowed references.
- A `PyObject*` returned as the function's result transfers ownership.

For each of these rules, there are a few exceptions for some functions in the C API. `PyList_SetItem` and `PyTuple_SetItem` *steal* a reference to the item they are setting (but not to the list or tuple object into which they're setting it), meaning that they take ownership even though by the general rules that item would be a borrowed reference. `PyList_SET_ITEM` and `PyTuple_SET_ITEM`, C preprocessor macros which implement faster versions of the item-setting functions, are also reference thieves, as is `PyModule_AddObject`, covered in Table 25-1. There are no other exceptions to the first rule. The rationale for these exceptions, which may help you remember them, is that the object you just created will be owned by the list, tuple, or module, so the reference stealing semantics saves you an unnecessary use of `Py_DECREF` immediately afterward.

The second rule has more exceptions than the first one. There are several cases in which the returned `PyObject*` is a borrowed reference rather than a new reference. The abstract functions—whose names begin with `PyObject_`, `PySequence_`, `PyMapping_` , and `PyNumber_`—return new references. This is because you can call them on objects of many types, and there might not be any other reference to the object that they return (i.e., the returned object might have to be created on the fly). The concrete functions—whose names begin with `PyList_`, `PyTuple_`, `PyDict_`, and so on—return a borrowed reference when the semantics of the object they return ensure that there must be some other reference to the returned object somewhere.

In this chapter, we mention all cases of exceptions to these rules (i.e., reasonably frequent cases of return of borrowed references, and rare cases of reference stealing from arguments) on all functions we cover. When we do not explicitly say otherwise, the function follows the rules: its `PyObject*` arguments, if any, are borrowed references, and the `PyObject*` result that it returns, if any, is a new reference.

Accessing Arguments

A function that has `ml_flags` in its `PyMethodDef` set to `METH_NOARGS` is called from Python with no arguments. The corresponding C function has a signature with only one argument, *`self`*. When `ml_flags` is `METH_O`, Python code calls the function with exactly one argument. The C function's second argument is a borrowed reference to the object that the Python caller passes as the argument's value.

When `ml_flags` is `METH_VARARGS`, Python code calls the function with any number of positional arguments, which the Python interpreter implicitly collects into a tuple. The C function's second argument is a borrowed reference to the tuple. Your C code then calls the `PyArg_ParseTuple` function, described here:

`PyArg_ParseTuple` `int PyArg_ParseTuple(PyObject* tuple, char* format, ...)`
Returns `0` for errors, and a value not equal to `0` for success. *`tuple`* is the `PyObject*` that was the C function's second argument. *`format`* is a C string that describes mandatory and optional arguments. The remaining arguments of `PyArg_ParseTuple` are addresses of C variables in which to put the values extracted from the tuple. Any `PyObject*` variables among the C variables are borrowed references.

Table 25-3 lists the commonly used code strings, of which zero or more are joined to form the string *`format`*.

Table 25-3. Format codes for `PyArg_ParseTuple`

Code	C type(s)	Meaning
`c`	`int`	A Python `bytes` or `bytearray` of length `1` becomes a C `int`.
`C`	`int`	A Python `str` of length `1` becomes a C `int`.
`d`	`double`	A Python `float` becomes a C `double`.
`D`	`Py_Complex`	A Python `complex` becomes a C `Py_Complex`.
`es`	`const char* + char**`	A Python `str` without embedded NULs, encoded with the encoding named by the `const char*` (`'utf-8'` when that pointer is `NULL`) becomes a C `char*` pointing to a newly allocated NUL-terminated buffer (C code must eventually free the buffer via `PyMem_Free`).

Code	C type(s)	Meaning
es#	const char* + char** + Py_ssize_t*	Like es, but the str can have embedded NULs, and the buffer is not NUL-terminated and is newly allocated only when the char** initially points to a NULL char* (otherwise, it must point to a pointer to an already-allocated buffer, and the location where the Py_ssize_t* points is set to the new buffer length—which must be <= the previous one, or else the call raises a ValueError). When the buffer gets newly allocated, the length of the new buffer goes where the Py_ssize_t* points.
et	const char* + char**	Like es, but the Python object can also be a bytes or bytearray (in which case Python ignores the encoding: it just copies the bytes to the newly allocated buffer).
et#	const char* + char** + Py_ssize_t*	Like es#, but the Python object can also be a bytes or bytearray (in which case Python ignores the encoding: it just copies the bytes to the buffer).
f	float	A Python float becomes a C float.
i, I	int	A Python int becomes a C int/unsigned int.
k, K	unsigned long	A Python int becomes a C unsigned long/ unsigned long long.
l, L	long int	A Python int becomes a C long/long long.
n	Py_ssize_t	A Python int becomes a C Py_ssize_t.
O	PyObject*	Gets a non-NULL borrowed reference to the Python argument.
O!	type + PyObject*	Like O, plus runtime type checking (see "Passing general objects" on page 11).
O&	convert + void*	Arbitrary conversion (see "Passing general objects" on page 11).
p	int	Evaluates any Python object as either true (it then sets the C int to 1) or false (it then sets the C int to 0).
s	const char*	A Python str without embedded NULs becomes a C const char* (encoded in UTF-8). No allocation is necessary.
s*	Py_buffer	Copies a Python str (encoded in UTF-8), bytes, or bytearray into the Py_buffer the caller provides.
s#	const char* + Py_ssize_t	Any Python str (encoded in UTF-8) or bytes becomes a C address (that points to the characters of the str or bytes argument) and a length. No allocation is necessary.
S	PyBytesObject*	A Python bytes becomes a borrowed reference to a Python object.

Code	C type(s)	Meaning
w*	Py_buffer	Copies a Python bytearray (or any other Python object that implements the read/write buffer interface) into the Py_buffer the caller provides.
y	const char*	A Python bytes without embedded NULLs becomes a C const char*. No allocation is necessary.
y*	Py_buffer	Like s*, but the Python object must be a bytes or bytearray, not a str. Best way to receive binary data.
y#	const char* + Py_ssize_t	A Python bytes becomes a C address and length. No allocation is necessary.
Y	PyByteArrayObject*	A Python bytearray becomes a borrowed reference to a Python object.
z	const char*	Like s, but the Python object can also be **None** (in which case the C pointer is set to NULL).
z*	Py_buffer	Like z*, but the Python object can also be **None** (in which case the field buf in the Py_buffer is set to NULL).
z#	const char* + Py_ssize_t	Like s#, but the Python object can also be **None** (in which case the C pointer is set to NULL).
(...)	As per ...	A Python sequence is treated as one argument per item.
|		Indicates that the following arguments are optional.
$		Indicates that the following arguments are keyword only (must come after | and only works with PyArg_ParseTupleAndKeywords).
:		Format end, followed by function name for error messages.
;		Format end, followed by entire error message text.

Code formats d to n (and other rarely used codes for tiny and short ints on the C side) accept numeric arguments from Python. Python "coerces" the corresponding values. For example, a code of i can correspond to a Python float; Python truncates the fractional part as if using the built-in function int. Py_Complex is a C struct with two fields named real and imag, both of type double. PyArg_ParseTuple will cause Python to raise an OverflowError exception when a given Python value exceeds the supported range for the target C type.

Passing general objects

O is the most general format code and accepts any argument, which you can later check and/or convert as needed. The variant O! corresponds to two arguments in the variable arguments: first the address of a Python type object, then the address of a PyObject*. O! checks that the corresponding value belongs to the given type (or any subtype of that type) before setting the PyObject* to point to the value; otherwise, it raises TypeError (the whole call fails, and the error is set to an appropriate

TypeError instance, as covered in Chapter 6). The variant O& also corresponds to two arguments in the variable arguments: first the address of a converter function you coded, then a void* (i.e., any address). The converter function must have the signature int *convert*(PyObject*, void*). Python calls your conversion function with the value passed from Python as the first argument, and the void* from the variable arguments as the second argument. The conversion function must either return 0 and raise an exception to indicate an error, or return 1 and store whatever is appropriate via the void* it gets.

Passing "strings"

There are many ways to pass Python "strings" in the general sense (str, bytes, and bytearray objects, sometimes demanding that the "string" contain no embedded null characters) as C arguments (i.e., as NUL-terminated arrays of chars, when feasible; or else, in a more general *address+length* arrangement). For example:

- The code format s accepts a string from Python and the address of a char* (i.e., a char**) among the variable arguments. It changes the char* to point at the string's buffer, which your C code must treat as a read-only, nul-terminated array of chars (i.e., a typical C string; however, your code must *not* modify it). The Python string must contain no embedded null characters; the resulting encoding is UTF-8. s# is similar, but corresponds to two arguments among the variable arguments: first the address of a char*, then the address of an int, which gets set to the string's length. The Python string can contain embedded nulls, and therefore so can the buffer to which the char* is set to point.
- z and z# are similar to s and s#, but the corresponding Python argument can also be None, in which case the C-side char* is set to NULL.
- y and y# are similar to s and s#, but the corresponding Python argument is a bytes, not a str. Strings are a typical example of read-only buffers. es and es# are also similar to s and s#, but they also accept an optional encoding name (by default, 'utf-8') with which to encode the str into bytes, and use a newly allocated buffer (which the C code must eventually free with PyMem_Free). et and et# are similar to es and es#, but the Python object can also be a bytes or bytearray, in which case Python ignores the encoding and just copies the bytes into the newly allocated buffer.

All of these string-accepting formats also have variants ending with * (s*, z*, y*, es*, et*) where the C argument is a struct Py_buffer which the caller supplies; in these cases, the function fills up the struct according to the incoming Python string. There's also a w* (without a non-* form) where the Python object must be a bytearray (or other read/write buffer).

Structuring the format

When one of the arguments is a Python sequence of known fixed length, you can use format codes for each of its items, and corresponding C addresses among the variable arguments, by grouping the format codes in parentheses. For example, the code `(ii)` corresponds to a Python sequence of two numbers and, among the remaining arguments, corresponds to two addresses of `int`s.

The format string may include a vertical bar (`|`) to indicate that all following arguments are optional. In this case, you must initialize the C variables, whose addresses you pass among the variable arguments for later arguments, to suitable default values before you call `PyArg_ParseTuple`. `PyArg_ParseTuple` does not change the C variables corresponding to optional arguments that were not passed in a given call from Python to your C-coded function.

An example

For example, here's the start of a function to be called with one mandatory integer argument, optionally followed by another integer argument defaulting to `23` if absent (rather like **`def`** `f(x, y=23, /):` in Python, except that the arguments must be numbers):

```
PyObject* f(PyObject* self, PyObject* args) {
  int x, y=23;
  if(!PyArg_ParseTuple(args, "i|i", &x, &y)
    return NULL;
  /* rest of function snipped */
}
```

The format string may optionally end with `:` *`name`* to indicate that *`name`* must be used as the function name if any error messages result. Alternatively, the format string may end with `;` *`text`* to indicate that *`text`* must be used as the error message if `PyArg_ParseTuple` detects errors (this form is rarely used).

Named (aka "keyword") arguments

A function that has `ml_flags` in its `PyMethodDef` set to `METH_KEYWORDS` accepts positional and named arguments. Python code calls the function with any number of positional arguments, which the Python interpreter collects into a tuple, and named arguments, which the Python interpreter collects into a dictionary. The C function's second argument is a borrowed reference to the tuple, and the third one is a borrowed reference to the dictionary. Your C code then calls the `PyArg_ParseTupleAndKeywords` function, described here:

`PyArg_ParseTupleAndKeywords`	`int PyArg_ParseTupleAndKeywords(PyObject* tuple, PyObject* dict, char* format, char** kwlist, ...)` Returns `0` for errors, and a value not equal to `0` for success. *`tuple`* is the `PyObject*` that was the C function's second argument. *`dict`* is the `PyObject*` that was the C function's third argument. *`format`* is the same as for `PyArg_ParseTuple`, except that it cannot include the `(...)` format code to parse nested sequences, but can optionally include a `$` (after a `|`) to indicate that all following arguments are optional and named only. *`kwlist`* is an array of `char*` terminated by a `NULL` sentinel, with the names of the parameters, one after the other.

For example, the following C code:

```
static PyObject*
func_c(PyObject* self, PyObject* args, PyObject* kwds)
{
    static char* argnames[] = {"x", "y", "z", NULL};
    double x, y=0.0, z=0.0;
    if(!PyArg_ParseTupleAndKeywords(args,kwds,"d|dd",argnames,&x,&y,&z))
        return NULL;
    /* rest of function snipped */
```

is roughly equivalent to this Python code:

```
def func_py(x, y=0., z=0.):
    x, y, z = map(float, (x,y,z))
    # rest of function snipped
```

Creating Python Values

C functions that communicate with Python must often build Python values, both to return as their `PyObject*` result and for other purposes, such as setting items and attributes. The simplest and handiest way to build a Python value is most often with the `Py_BuildValue` function:

`Py_BuildValue`	`PyObject* Py_BuildValue(char* format, ...)` *`format`* is a C string (similar to the one you pass to `PyArg_ParseTuple`) describing the Python object to build. The following arguments of `Py_BuildValue` are C values from which the result is built. The `PyObject*` result is a new reference.

Table 25-4 lists the commonly used code strings, of which zero or more are joined into string *`format`*. `Py_BuildValue` builds and returns a tuple if *`format`* contains two or more format codes, or if *`format`* begins with `(` and ends with `)`. Otherwise, the result is not a tuple. When you pass buffers—as, for example, in the case of format code `s#`—`Py_BuildValue` copies the data. You can therefore modify, abandon, or `free()` your original copy of the data after `Py_BuildValue` returns. `Py_BuildValue`

always returns a new reference (except for format code N). Called with an empty *format*, Py_BuildValue('') returns a new reference to None.

Table 25-4. Main format codes for Py_BuildValue

Code	C type	Meaning
B	unsigned char	A C unsigned char becomes a Python int.
b	char	A C char becomes a Python int.
C	char	A C char becomes a Python str of length 1.
c	char	A C char becomes a Python bytes of length 1.
D	double	A C double becomes a Python float.
d	Py_Complex*	A C Py_Complex becomes a Python complex.
f	float	A C float becomes a Python float.
H	unsigned short	A C unsigned short becomes a Python int.
h	short	A C short becomes a Python int.
I	unsigned int	A C unsigned int becomes a Python int.
i	int	A C int becomes a Python int.
K	unsigned long long	A C unsigned long long becomes a Python int (if the platform supports it).
k	unsigned long	A C unsigned long becomes a Python int.
L	long long	A C long long becomes a Python int (if the platform supports it).
l	long	A C long becomes a Python int.
n	Py_ssize_t	A C Py_ssize_t becomes a Python int.
N	PyObject*	Passes a Python object and *steals* a reference, or passes NULL if the pointer is NULL.
O	PyObject*	Passes a Python object and INCREFs it, or passes NULL if the pointer is NULL.
O&	convert + void*	Arbitrary conversion (see immediately after this table).
s	char*	A C 0-terminated char* becomes a Python str (decoding with 'utf-8'), or None if the pointer is NULL.
s#	char* + int	A C char* and length become a Python str (decoding with 'utf-8'), or None if the pointer is NULL.
u	const wchar_t*	A C wide character (wchar) (UTF-16 or UCS-4) NUL-terminated string becomes a Python str, or None if the pointer is NULL.
u#	const wchar_t* + int	A C wide character (wchar) (UCS-2 or UCS-4) string and length become a Python str, or None if the pointer is NULL.
y	char* + int	A C char NUL-terminated string becomes a Python bytes, or None if the pointer is NULL.

Code	C type	Meaning
`y#`	`char* + int`	A C `char` string and length become a Python `bytes`, or `None` if the pointer is `NULL`.
`z`	`char*`	A C `0`-terminated `char*` becomes a Python `str` (decoding with `'utf-8'`), or `None` if the pointer is `NULL`.
`z#`	`char* + int`	A C `char*` and length become a Python `str` (decoding with `'utf-8'`), or `None` if the pointer is `NULL`.
`(...)`	As per `...`	Builds a Python tuple from C values.
`[...]`	As per `...`	Builds a Python list from C values.
`{...}`	As per `...`	Builds a Python dictionary from C values, alternating keys and values (must be an even number of C values).

The code `O&` corresponds to two arguments among the variable arguments: first, the address of a converter function you code, then a `void*` (i.e., any address). The converter function must have the signature `PyObject* convert(void*)`. Python calls the conversion function with the `void*` from the variable arguments as the only argument. The conversion function must either return `NULL` and raise an exception (as covered in the next section) to indicate an error, or return a new reference `PyObject*` built from data obtained through the `void*`.

The code `{...}` builds dictionaries from an even number of C values, alternately keys and values. For example, `Py_BuildValue("{issi}",23,"zig","zag",42)` returns a new `PyObject*` for `{23: 'zig', 'zag': 42}`.

Note the crucial difference between codes `N` and `O`: `N` *steals* a reference from the corresponding `PyObject*` value among the variable arguments, so it's convenient to build an object with a reference you own that you would otherwise have to `Py_DECREF`. `O` does *not* steal a reference, so it is convenient to build an object with a reference you don't own, or a reference you must also keep elsewhere.

Exceptions

To propagate exceptions raised from other functions you call, just return `NULL` as the `PyObject*` result from your C function. To raise your own exceptions, first set the current exception indicator, then return `NULL`. Python's built-in exception classes (covered in "Standard Exception Classes" in Chapter 6) are globally available with names starting with `PyExc_`, such as `PyExc_AttributeError`, `PyExc_KeyError`, and so on. Your extension module can also supply and use its own exception classes. The most commonly used C API functions related to raising exceptions are listed in Table 25-5.

Table 25-5. C API functions for raising exceptions

`PyErr_Format`	`PyObject* PyErr_Format(PyObject* type, char* format, ...)`
	Raises an exception of class *type*, which must be either a built-in such as `PyExc_IndexError` or an exception class created with `PyErr_NewException`. Builds the associated value from the format string *format*, which has syntax similar to C's `printf`'s, and the following C values indicated as variable arguments (`...`) above. Returns `NULL`, so your C code can just use, for example:
	`return PyErr_Format(PyExc_KeyError, "Unknown key name (%s)", thekeystring);`

`PyErr_NewException`	`PyObject* PyErr_NewException(char* name, PyObject* base, PyObject* dict)`
	Extends the exception class *base*, with extra class attributes and methods from dictionary *dict* (usually `NULL`, indicating "no extra class attributes or methods"; that is, the same as an empty `dict`). Creates a new exception class named *name* (the string *name* must be of the form "*modulename.classname*") and returns a new reference to the new class object. When *base* is `NULL`, uses `PyExc_Exception` as the base class. You normally call this function during initialization of a module object. For example:
	`PyModule_AddObject(module, "error", PyErr_NewException( "mymodule.error", NULL, NULL));`

`PyErr_NoMemory`	`PyObject* PyErr_NoMemory()`
	Raises an out-of-memory error and returns `NULL`, so your code can just use:
	`return PyErr_NoMemory();`

`PyErr_SetFromErrno`	`PyObject* PyErr_SetFromErrno(PyObject* type)`
	Raises an exception of class *type*, which must be a built-in such as `PyExc_OSError` or an exception class created with `PyErr_NewException`. Takes all details from `errno`, which C standard library functions and system calls set for many error cases, and the standard C library function `strerror`, which translates such error codes into appropriate strings. Returns `NULL`, so your code can just use, for example:
	`return PyErr_SetFromErrno(PyExc_IOError);`

`PyErr_SetFromErrnoWithFilename`	`PyObject* PyErr_SetFromErrnoWithFilename(PyObject* type, char* filename)` Like `PyErr_SetFromErrno`, but also provides the string *filename* as part of the exception's value. When *filename* is `NULL`, works just like `PyErr_SetFromErrno`.
`PyErr_SetObject`	`void PyErr_SetObject(PyObject* type, PyObject* value)` Raises an exception of class *type*, which must be a built-in such as `PyExc_KeyError` or an exception class created with `PyErr_NewException`, with *value* as the associated value (a borrowed reference). `PyErr_SetObject` is a `void` function (i.e., returns no value).

`PyErr_SetString`	`void PyErr_SetString(PyObject* type, char* message)` Raises an exception of class *type*, which must be a built-in such as `PyExc_KeyError` or an exception class created with `PyErr_NewException`, with *message* (NUL-terminated, encoded in UTF-8) as the error message.

Your C code may want to deal with an exception and continue, as a **`try/except`** statement would let you do in Python code. Table 25-6 lists the most commonly used C API functions related to catching exceptions.

Table 25-6. C API functions for catching exceptions

`PyErr_Clear`	`void PyErr_Clear()` Clears the error indicator. Innocuous if no error is pending.
`PyErr_ExceptionMatches`	`int PyErr_ExceptionMatches(PyObject* type)` Call only when an error is pending, as indicated by a non-`NULL` return value from `PyErr_Occurred`; otherwise the whole program might crash. Returns a value `!=0` when the pending exception is an instance of the given *type* or any subclass of *type*, or `0` when the pending exception is not such an instance.
`PyErr_Occurred`	`PyObject* PyErr_Occurred()` Returns `NULL` if no error is pending; otherwise, a borrowed reference to the type of the pending exception. (*Don't* use the specific returned value; instead, call `PyErr_ExceptionMatches` to catch exceptions of subclasses as well, as is normal and expected.)
`PyErr_Print`	`void PyErr_Print()` Call only when an error is pending, as indicated by a non-`NULL` return value from `PyErr_Occurred`; otherwise the whole program might crash. Outputs a standard traceback to `sys.stderr`, then clears the error indicator.

If you need to process errors in very advanced ways, study other error-related functions of the C API, such as `PyErr_Fetch`, `PyErr_Normalize`, `PyErr_GivenExceptionMatches`, and `PyErr_Restore`, in the online docs. This book does not cover those advanced, rarely needed possibilities.

Abstract Layer Functions

The code for a C extension typically needs to use some Python functionality. For example, your code may need to examine or set attributes and items of Python objects, call Python-coded and Python built-in functions and methods, and so on. In most cases, the best approach is for your code to call functions from the *abstract layer* of Python's C API. These are functions that you can call on any Python object (functions whose names start with `PyObject_`), or on any object within a wide category, such as mappings, numbers, or sequences (with names starting with `PyMapping_`, `PyNumber_`, and `PySequence_`).

Many of the functions callable specifically on a typed object in these categories duplicate functionality also available from `PyObject_` functions. In these cases, for generality, you should almost invariably use the "abstract" `PyObject_` function instead. We don't cover such almost-redundant functions in this book.

Functions in the abstract layer raise Python exceptions if you call them on objects to which they are not applicable. All of these functions accept borrowed references for `PyObject*` arguments and return a new reference (`NULL` for an exception) if they return a `PyObject*` result.

The most frequently used abstract layer functions are listed in Table 25-7.

Table 25-7. Commonly used abstract layer functions

`PyCallable_Check`	`int PyCallable_Check(PyObject* x)` Returns `1` (true) when *x* is callable, like `callable(x)`; otherwise, returns `0` (false).
`PyIter_Check`	`int PyIter_Check(PyObject* x)` Returns `1` (true) when *x* is an iterator; otherwise, returns `0` (false).
`PyIter_Next`	`PyObject* PyIter_Next(PyObject* x)` Returns the next item from iterator *x*. Returns `NULL` *without* raising any exception when *x*'s iteration is finished (i.e., when `next(x)` raises `StopIteration`).
`PyNumber_Check`	`int PyNumber_Check(PyObject* x)` Returns `1` (true) when *x* is a number; otherwise, returns `0` (false).
`PyObject_Call`	`PyObject* PyObject_Call(PyObject* f, PyObject* args, PyObject* kwds)` Calls the callable Python object *f* with positional arguments in `tuple` *args* (may be empty, but never `NULL`) and named arguments in `dict` *kwds*. Returns the call's result. Like *f*`(*args, **kwds)`.
`PyObject_CallFunction`	`PyObject* PyObject_CallFunction(PyObject* f, char* format, ...)` Calls the callable Python object *f* with positional arguments described by string *format*, with the same format codes as `Py_BuildValue`. When *format* is `NULL`, calls *x* with no arguments. Returns the call's result.
`PyObject_CallFunctionObjArgs`	`PyObject* PyObject_CallFunctionObjArgs(PyObject* f, ..., NULL)` Calls the callable Python object *f* with positional arguments passed as zero or more `PyObject*` arguments. Returns the call's result.
`PyObject_CallMethod`	`PyObject* PyObject_CallMethod(PyObject* x, char* method, char* format, ...)` Calls the method named *method* of Python object *x* with positional arguments described by string *format*, with the same format codes as `Py_BuildValue` (see Table 25-4). When *format* is `NULL`, calls the method with no arguments. Returns the call's result.

`PyObject_CallMethodObjArgs`	`PyObject* PyObject_CallMethodObjArgs(PyObject* x, char* method, ..., NULL)` Calls the method named *method* of Python object *x* with positional arguments passed as zero or more `PyObject*` arguments. Returns the call's result.
`PyObject_CallObject`	`PyObject* PyObject_CallObject(PyObject* f, PyObject* args)` Calls the callable Python object *f* with positional arguments in `tuple` *args* (may be `NULL` or an empty `tuple` to pass no arguments). Returns the call's result. Like *f*(**args*).
`PyObject_DelAttrString`	`int PyObject_DelAttrString(PyObject* x, char* name)` Deletes *x*'s attribute named *name*, like **`del`** *x*.*name*.
`PyObject_DelItem`	`int PyObject_DelItem(PyObject* x, PyObject* key)` Deletes *x*'s item with key (or index) *key*, like **`del`** *x*[*key*].
`PyObject_DelItemString`	`int PyObject_DelItemString(PyObject* x, char* key)` Deletes *x*'s item with key *key*, like **`del`** *x*[*key*].
`PyObject_GetAttrString`	`PyObject* PyObject_GetAttrString(PyObject* x, char* name)` Returns *x*'s attribute *name*, like *x*.*name*.
`PyObject_GetItem`	`PyObject* PyObject_GetItem(PyObject* x, PyObject* key)` Returns *x*'s item with key (or index) *key*, like *x*[*key*].
`PyObject_GetItemString`	`int PyObject_GetItemString(PyObject* x, char* key)` Returns *x*'s item with key *key*, like *x*[*key*].
`PyObject_GetIter`	`PyObject* PyObject_GetIter(PyObject* x)` Returns an iterator on *x*, like `iter(`*x*`)`.
`PyObject_HasAttrString`	`int PyObject_HasAttrString(PyObject* x, char* name)` Returns `1` (true) if *x* has an attribute *name*, like `hasattr(`*x*`, `*name*`)`; otherwise, returns `0` (false).
`PyObject_IsTrue`	`int PyObject_IsTrue(PyObject* x)` Returns `1` (true) if *x* is true for Python, like `bool(`*x*`)`; otherwise, returns `0` (false).
`PyObject_Length`	`int PyObject_Length(PyObject* x)` Returns *x*'s length, like `len(`*x*`)`.
`PyObject_Repr`	`PyObject* PyObject_Repr(PyObject* x)` Returns *x*'s detailed string representation, like `repr(`*x*`)`.

`PyObject_RichCompare`	`PyObject* PyObject_RichCompare(PyObject* x, PyObject* y, int op)` Performs the comparison indicated by *op* between *x* and *y*, and returns the result as a new reference to a Python bool object (**`True`** or **`False`**). *op* can be `Py_EQ`, `Py_NE`, `Py_LT`, `Py_LE`, `Py_GT`, or `Py_GE`, corresponding to Python comparisons `x == y`, `x != y`, `x < y`, `x <= y`, `x > y`, or `x >= y`. Returns `NULL` when it needs to indicate that an exception has been raised.
`PyObject_RichCompareBool`	`int PyObject_RichCompareBool(PyObject* x, PyObject* y, int op)` Like `PyObject_RichCompare`, but returns `0` for false, `1` for true, or `-1` to indicate that an exception has been raised.
`PyObject_SetAttrString`	`int PyObject_SetAttrString(PyObject* x, char* name, PyObject* v)` Sets *x*'s attribute named *name* to *v*, like `x.name = v`.
`PyObject_SetItem`	`int PyObject_SetItem(PyObject* x, PyObject* k, PyObject *v)` Sets *x*'s item with key (or index) *key* to *v*, like `x[key] = v`.
`PyObject_SetItemString`	`int PyObject_SetItemString(PyObject* x, char* key, PyObject *v)` Sets *x*'s item with key *key* to *v*, like `x[key] = v`.
`PyObject_Str`	`PyObject* PyObject_Str(PyObject* x)` Returns *x*'s readable string form, like `str(x)`. To get the result as bytes, use `PyObject_Bytes`.
`PyObject_Type`	`PyObject* PyObject_Type(PyObject* x)` Returns *x*'s type object, like `type(x)`.
`PySequence_Contains`	`int PySequence_Contains(PyObject* x, PyObject* v)` Returns `1` (true) if *v* is an item in *x*, like *v* **`in`** *x*; otherwise, returns `0` (false).
`PySequence_DelSlice`	`int PySequence_DelSlice(PyObject* x, int start, int stop)` Deletes *x*'s slice from *start* to *stop*, like **`del`** `x[start:stop]`.
`PySequence_Fast`	`PyObject* PySequence_Fast(PyObject* x)` Returns a new reference to a tuple with the same items as sequence *x*, unless *x* is a list, in which case `PySequence_Fast` returns a new reference to *x*. When you need to get many items of an arbitrary sequence *x*, it's fastest to call `t = PySequence_Fast(x)`, then call `PySequence_Fast_GET_ITEM(t, i)` as many times as needed, and finally call `Py_DECREF(t)`.
`PySequence_Fast_GET_ITEM`	`PyObject* PySequence_Fast_GET_ITEM(PyObject* x, int i)` Returns item *i* of *x*, where *x* must be the result of `PySequence_Fast` and `!=NULL`, and `0 <= i < PySequence_Fast_GET_SIZE(t)`. Violating these conditions can cause program crashes. This approach is optimized for speed, not for safety.

`PySequence_Fast_GET_SIZE`	`int PySequence_Fast_GET_SIZE(PyObject* x)` Returns the length of *x*. *x* must be the result of `PySequence_Fast` and `!=NULL`.
`PySequence_GetSlice`	`PyObject* PySequence_GetSlice(PyObject* x, int start, int stop)` Returns *x*'s slice from `start` to `stop`, like *x*`[start:stop]`.
`PySequence_List`	`PyObject* PySequence_List(PyObject* x)` Returns a new list object with the same items as *x*, like `list(x)`.
`PySequence_SetSlice`	`int PySequence_SetSlice(PyObject* x, int start, int stop, PyObject* v)` Sets *x*'s slice from `start` to `stop` to *v*, like *x*`[start:stop] = v`. Just as in the equivalent Python statement, *v* must be an iterable.
`PySequence_Tuple`	`PyObject* PySequence_Tuple(PyObject* x)` Returns a new reference to a tuple with the same items as *x*, like `tuple(x)`.

Other functions, whose names start with `PyNumber_`, let you perform numeric operations. Unary `PyNumber` functions, which take one argument `PyObject*` *x* and return a `PyObject*`, are listed in Table 25-8 with their Python equivalents.

Table 25-8. Unary `PyNumber` functions

Function	Python equivalent
`PyNumber_Absolute`	`abs(x)`
`PyNumber_Float`	`float(x)`
`PyNumber_Invert`	`~x`
`PyNumber_Long`	`int(x)`
`PyNumber_Negative`	`-x`
`PyNumber_Positive`	`+x`

Binary `PyNumber` functions, which take two `PyObject*` arguments *x* and *y* and return a `PyObject*`, are similarly listed in Table 25-9.

Table 25-9. Binary `PyNumber` functions

Function	Python equivalent
`PyNumber_Add`	`x + y`
`PyNumber_And`	`x & y`
`PyNumber_Divide`	`x // y`
`PyNumber_Divmod`	`divmod(x, y)`
`PyNumber_FloorDivide`	`x // y`
`PyNumber_Lshift`	`x << y`

Function	Python equivalent
PyNumber_MatrixMultiply	x @ y
PyNumber_Multiply	x * y
PyNumber_Or	x | y
PyNumber_Remainder	x % y
PyNumber_Rshift	x >> y
PyNumber_Subtract	x - y
PyNumber_TrueDivide	x / y
PyNumber_Xor	x ^ y

All the binary PyNumber functions have in-place equivalents whose names start with PyNumber_InPlace, such as PyNumber_InPlaceAdd and so on. The in-place versions try to modify the first argument in place, if possible, and in any case return a new reference to the result, be it the first argument (modified) or a new object. Python's built-in numbers are immutable; therefore, when the first argument is a number of a built-in type, the in-place versions work just the same as the ordinary versions. The function PyNumber_Divmod returns a tuple with two items (the quotient and the remainder) and has no in-place equivalent.

There is one ternary PyNumber function, PyNumber_Power:

PyNumber_Power	PyObject* PyNumber_Power(PyObject* *x*, PyObject* *y*, PyObject* *z*) When *z* is Py_None, returns *x* raised to the *y* power, like *x* ** *y* or, equivalently, pow(*x*, *y*). Otherwise, returns (*x* ** *y*) % *z*, like pow(*x*, *y*, *z*). The in-place version is named PyNumber_InPlacePower.

Concrete Layer Functions

Each specific type of Python built-in object supplies concrete functions to operate on instances of that type, with names starting with Py*<type>*_ (e.g., PyInt_ for functions related to Python ints). Most such functions duplicate the functionality of abstract layer functions or auxiliary functions covered earlier in this chapter, such as Py_BuildValue, which can generate objects of many types. In this section, we cover just a few frequently used functions from the concrete layer that provide unique functionality, or very substantial convenience or extra speed. For most types, you can check whether an object belongs to the type by calling Py*<type>*_Check, which also accepts instances of subtypes, or Py*<type>*_CheckExact, which accepts only instances of *type*, not of subtypes. Signatures are the same as for the function PyIter_Check, covered in Table 25-10.

Table 25-10. Frequently useful concrete functions

`PyDict_GetItem`	`PyObject* PyDict_GetItem(PyObject* x, PyObject* key)` Returns a borrowed reference to the value corresponding to key *key* of dictionary *x*, or `NULL` if *key* is not in *x*.
`PyDict_Merge`	`int PyDict_Merge(PyObject* x, PyObject* y, int override)` Updates dictionary *x* by merging the items of dictionary *y* into *x*. *override* determines what happens when a key *k* is present in both *x* and *y*: when *override* is `0`, *x*`[`*k*`]` remains the same; otherwise, *x*`[`*k*`]` is replaced ("overridden") by the value *y*`[`*k*`]`.
`PyDict_MergeFromSeq2`	`int PyDict_MergeFromSeq2(PyObject* x, PyObject* y, int override)` Like `PyDict_Merge`, except that *y* is not a dictionary but a sequence of sequences, where each subsequence has length `2` and is used as a `(`*key*`, `*value*`)` pair.
`PyDict_Next`	`int PyDict_Next(PyObject* x, int* pos, PyObject** k, PyObject** v)` Iterates over items in dictionary *x*. You must initialize `*`*pos* to `0` at the start of the iteration: `PyDict_Next` uses and updates `*`*pos* to keep track of its place. For each successful iteration step, `PyDict_Next` returns `1`; when there are no more items, it returns `0`. The function updates `*`*k* and `*`*v* to point to the next key and value, respectively (borrowed references), at each step that returns `1`. You can pass either *k* or *v* as `NULL` when you are not interested in the key or value. During an iteration, you must not change the set of *x*'s keys in any way, but you can change *x*'s values as long as the set of keys remains identical.
`PyFloat_AS_DOUBLE`	`double PyFloat_AS_DOUBLE(PyObject* x)` Returns the C `double` value of Python `float` *x*, very quickly, without any error checking.
`PyList_GET_ITEM`	`PyObject* PyList_GET_ITEM(PyObject* x, int pos)` Returns the *pos*-th item of list *x*, without any error checking.
`PyList_New`	`PyObject* PyList_New(int length)` Returns a new, uninitialized list of the given *length*. You must then initialize the list, typically by calling `PyList_SET_ITEM` *length* times.
`PyList_SET_ITEM`	`int PyList_SET_ITEM(PyObject* x, int pos, PyObject* v)` Sets the *pos*-th item of list *x* to *v*, without any error checking. Steals a reference to *v*. Use only *right after* creating a new list *x* with `PyList_New`.
`PyUnicode_AS_UNICODE`	`char* PyUnicode_AS_UNICODE(PyObject* x)` Returns a pointer to the internal bytes buffer of string *x*, fast, without error checking. *Don't* modify the buffer, unless you just allocated it by calling `PyUnicode_FromStringAndSize(NULL, `*size*`)`.

PyUnicode_ AsStringAndSize	int PyUnicode_AsStringAndSize(PyObject* *x*, char** *buffer*, int* *length*) Puts a pointer to the internal buffer of string *x* in **buffer*, and *x*'s length in **length*. *Don't* modify the buffer, unless you just allocated it by calling PyUnicode_FromStringAndSize(NULL, *size*).
PyUnicode_FromFormat	PyObject* PyUnicode_FromFormat(char* *format*, ...) Returns a new Python string built from format string *format*, which has syntax similar to printf's, and the following C values indicated as variable arguments (...) above.
PyUnicode_ FromStringAndSize	PyObject* PyUnicode_FromStringAndSize(char* *data*, int *size*) Returns a Python string of length *size*, copying *size* bytes from *data*. When *data* is NULL, the Python string is uninitialized, and you must initialize it. You can get the pointer to the string's internal buffer by calling PyUnicode_AS_UNICODE.
PyTuple_GET_ITEM	PyObject* PyTuple_GET_ITEM(PyObject* *x*, int *pos*) Returns the *pos*-th item of tuple *x*, without error checking.
PyTuple_New	PyObject* PyTuple_New(int *length*) Returns a new, uninitialized tuple of the given *length*. You must initialize the tuple, usually via PyTuple_SET_ITEM *length* times.
PyTuple_SET_ITEM	int PyTuple_SET_ITEM(PyObject* *x*, int *pos*, PyObject* *v*) Sets the *pos*-th item of tuple *x* to *v*, without error checking. Steals a reference to *v*. Use only immediately after creating a new tuple *x* with PyTuple_New.

A Simple Extension Example

Example 25-1 exposes the functionality of Python C API functions PyDict_Merge and PyDict_MergeFromSeq2 for Python use. The update method of dicts works like PyDict_Merge with *override*=1, but this example is (very slightly!) more general.

Example 25-1. A simple Python extension module merge.c

```
#include <Python.h>
static PyObject*
merge(PyObject* self, PyObject* args, PyObject* kwds)
{
    static char* argnames[] = {"x","y","override",NULL};
    PyObject *x, *y;
    int override = 0;
    if(!PyArg_ParseTupleAndKeywords(args, kwds, "O!O|i", argnames,
        &PyDict_Type, &x, &y, &override))
            return NULL;
    if(-1 == PyDict_Merge(x, y, override)) {
        if(!PyErr_ExceptionMatches(PyExc_AttributeError))
```

```
            return NULL;
        PyErr_Clear();
        if(-1 == PyDict_MergeFromSeq2(x, y, override))
            return NULL;
    }
    return Py_BuildValue("");
}
static char merge_docs[] = "\
merge(x, y, override=False): merge into dict x the items of dict y (or\n\
  the pairs that are the items of y, if y is a sequence), with\n\
  optional override. Alters dict x directly, returns None.\n\
";
static PyObject*
mergenew(PyObject* self, PyObject* args, PyObject* kwds)
{
    static char* argnames[] = {"x","y","override",NULL};
    PyObject *x, *y, *result;
    int override = 0;
    if(!PyArg_ParseTupleAndKeywords(args, kwds, "O!O|i", argnames,
        &PyDict_Type, &x, &y, &override))
            return NULL;
    result = PyObject_CallMethod(x, "copy", "");
    if(!result)
        return NULL;
    if(-1 == PyDict_Merge(result, y, override)) {
        if(!PyErr_ExceptionMatches(PyExc_AttributeError))
            return NULL;
        PyErr_Clear();
        if(-1 == PyDict_MergeFromSeq2(result, y, override))
            return NULL;
    }
    return result;
}
static char mergenew_docs[] = "\
mergenew(x, y, override=False): merge into dict x the items of dict y\n\
  (or the pairs that are the items of y, if y is a sequence), with\n\
  optional override. Does NOT alter x, but rather returns the\n\
  modified copy as the function's result.\n\
";
static PyMethodDef merge_funcs[] = {
    {"merge", (PyCFunction)merge, METH_VARARGS | METH_KEYWORDS,
     merge_docs},
    {"mergenew", (PyCFunction)mergenew, METH_VARARGS | METH_KEYWORDS,
     mergenew_docs},
    {NULL}
};
static char merge_module_docs[] = "Example extension module";
static struct PyModuleDef merge_module = {
    PyModuleDef_HEAD_INIT,
    "merge",
    merge_module_docs,
    -1,
```

```
    merge_funcs
};

PyMODINIT_FUNC
PyInit_merge(void)
{
    return PyModule_Create(&merge_module);
}
```

This example declares as `static` every function and global variable in the C source file except `PyInit_merge`, which must be "public" so Python can call it. Since the functions and variables are exposed to Python via `PyMethodDef` structures, Python does not need to see their names directly. Declaring them `static` is best: it ensures their names don't accidentally end up in the whole program's global namespace, as might otherwise happen on some platforms, possibly causing conflicts and errors.

As described in Table 25-3, the format string `"O!O|i"` passed to `PyArg_ParseTupleAndKeywords` indicates that the function `merge` accepts three arguments from Python: an object with a type constraint, a generic object, and an optional integer. At the same time, the format string indicates that the variable part of `PyArg_ParseTupleAndKeywords`'s arguments must contain four addresses in the following order: the address of a Python type object, two addresses of `PyObject*` variables, and the address of an `int` variable. The `int` variable must be previously initialized to its intended default value, since the corresponding Python argument is optional.

In keeping with these requirements, after the *`argnames`* argument the code passes `&PyDict_Type` (i.e., the address of the dictionary type object), then the addresses of the two `PyObject*` variables. Finally, it passes the address of the variable *`override`*, an `int` that was previously initialized to `0`, since the default when the *`override`* argument is not explicitly passed from Python is `False` ("no overriding"). When the return value of `PyArg_ParseTupleAndKeywords` is `0`, the code returns `NULL` to propagate the exception; this diagnoses most cases where Python code passes incorrect arguments to our new function `merge`.

When the arguments appear to be OK, it tries `PyDict_Merge`, which succeeds if *`y`* is a dictionary. When `PyDict_Merge` raises an `AttributeError`, indicating that *`y`* does not have a `keys` attribute, the code clears the error and tries again, this time with `PyDict_MergeFromSeq2`, which succeeds when *`y`* is a sequence of pairs. If that also fails, it returns `NULL` to propagate the exception. Otherwise, it returns `None` in the simplest way (i.e., with `return Py_BuildValue("")`) to indicate success.

The `mergenew` function basically duplicates `merge`'s functionality; however, `mergenew` does not alter its arguments, but rather builds and returns a new dictionary as the function's result. The C API function `PyObject_CallMethod` lets `mergenew` call the `copy` method of its first Python-passed argument, a dictionary object, and obtain a new dictionary object, which it then alters (with exactly the same logic as the `merge` function). It then returns the altered dictionary as the function result (thus, there's no need to call `Py_BuildValue` in this case).

The code of Example 25-1 must be in a source file named *merge.c*. In the same directory, create the following script named *setup.py*:

```
from setuptools import setup, Extension
setup(name='merge',
      ext_modules=[Extension('merge',sources=['merge.c'])])
```

Run **`python setup.py install`** at a shell prompt in this directory (ideally, in a virtual environment; or, if you insist, with a user ID having appropriate privileges to write into your Python installation, or using **`sudo`** on Unix-like systems if necessary). This builds the dynamically loaded library for the `merge` module, and copies it to the appropriate directory for the virtual environment (or your Python installation). Now, Python code (in the appropriate virtual environment, if any) can use the module. For example:

```
import merge
x = {'a':1,'b':2 }
merge.merge(x,[['b',3],['c',4]])
print(x)                            # {'a':1, 'b':2, 'c':4 }
print(merge.mergenew(x,{'a':5,'d':6},
                     override=1))  # {'a':5, 'b':2, 'c':4, 'd':6 }
print(x)                            # {'a':1, 'b':2, 'c':4 }
```

This example shows the difference between `merge` (which alters its first argument) and `mergenew` (which returns a new object and does not alter its argument). It also shows that the second argument can be either a dictionary or a sequence of two-item subsequences, and shows the default operation (where keys that are already in the `dict` are left alone) versus the `override` option (where keys coming from the second argument take precedence, as in Python dictionaries' `update` method).

Defining New Types

In your extension modules, you will often want to define new types and make them available to Python. A type's definition is held in a `struct` named `PyTypeObject`. Most of the fields of `PyTypeObject` are pointers to functions. Some fields point to other `structs`, which in turn are blocks of pointers to functions. `PyTypeObject` also includes a few fields that give the type's name, size, and behavior details (option flags). You can leave almost all fields of `PyTypeObject` set to `NULL` if you do not supply the related functionality. You can point some fields to functions in the Python C API to supply fundamental object functionality in standard ways.

The best way to implement a type is to copy from the Python sources one of three files in the directory *Modules*, which Python supplies exactly for such didactical purposes, and edit it. The files are named *xxlimited.c*, *xxmodule.c*, and *xxsubtype.c* (the latter focused on subclassing built-in types, with two example types, one each subclassing from `list` and `dict`, respectively).

See the online docs for detailed information on `PyTypeObject` and other related `structs`. The file *Include/object.h* in the Python sources contains the declarations of these types, as well as several important comments that you should study.

Per-instance data

To represent each instance of your type, declare a C `struct` that starts, right after the opening brace, with the macro `PyObject_HEAD`. The macro expands into the data fields that your `struct` must begin with to be a Python object. These fields include the reference count and a pointer to the instance's type. Any pointer to your structure can be correctly cast to a `PyObject*`. You can choose to look at this practice as a kind of C-level implementation of a (single) inheritance mechanism.

The `PyTypeObject struct` defining your type must contain the size of your per-instance `struct`, as well as pointers to the C functions you write to operate on your structure. Thus, you normally place `PyTypeObject` toward the end of your C-coded module's source code, after the definitions of the per-instance `struct` and of all the functions operating on instances of that `struct`. Each *x* pointing to a `struct` starting with `PyObject_HEAD`, and in particular each `PyObject*` *x*, has a field *x*`->ob_type` that is the address of the `PyTypeObject` structure that is *x*'s Python type object.

The PyTypeObject definition

Given a per-instance `struct` such as:

```
typedef struct {
    PyObject_HEAD
    /* other data needed by instances of this type, omitted */
} mytype;
```

the related `PyTypeObject struct` almost invariably begins in a way similar to:

```
static PyTypeObject t_mytype = {
/* tp_head */      PyObject_HEAD_INIT(NULL) /* NULL for portability */
/* tp_name */      "mymodule.mytype", /* type name, including module */
/* tp_basicsize */ sizeof(mytype),
/* tp_itemsize */  0,                 /* 0 except variable-size type */
/* tp_dealloc */   (destructor)mytype_dealloc,
/* tp_print */     0,                 /* usually 0, use str instead */
/* tp_getattr */   0,                 /* usually 0 (see getattro) */
/* tp_setattr */   0,                 /* usually 0 (see setattro) */
/* tp_compare*/    0,                 /* see also richcompare */
/* tp_repr */      (reprfunc)mytype_str,  /* like Python's __repr__*/
    /* rest of struct omitted */
```

For maximum portability, the `PyObject_HEAD_INIT` macro at the start of the `PyTypeObject struct` must have an argument of `NULL`. During module initialization, call `PyType_Ready(&t_mytype)`, which, among other tasks, inserts in `t_mytype` the address of its type (the type of a type is also known as a *metatype*), normally `&PyType_Type`. Another slot in `PyTypeObject` pointing to another type object is `tp_base`, which comes later in the structure. In the structure definition itself, you must have a `tp_base` of `NULL`, again for maximum compatibility. Before you invoke `PyType_Ready(&t_mytype)`, you can optionally set `t_mytype.tp_base` to the

address of another type object. When you do so, your type inherits from the other type, just as a class coded in Python can optionally inherit from another type. For a Python type coded in C, "inheriting" means that, for most fields of `PyTypeObject`, if you set the field to `NULL`, `PyType_Ready` copies the corresponding field from the base type. A type must explicitly assert in its field `tp_flags` that it is usable as a base type; otherwise, no type can inherit from it.

The `tp_itemsize` field is of interest only for types that, like tuples, have instances of different sizes, and can definitely determine each instance's size at creation time. Most types just set `tp_itemsize` to `0`. You'll usually set the fields `tp_getattr` and `tp_setattr` to `NULL`, since they exist only for backward compatibility; modern types use the fields `tp_getattro` and `tp_setattro` instead. The `tp_repr` field is typical of most of the following fields, which we omit here: the field holds the address of a function, which corresponds directly to a Python special method (here, `__repr__`). You can set the field to `NULL`, indicating that your type does not supply the special method; otherwise, set the field to point to an appropriate function. When you set the field to `NULL` but point to a base type from the `tp_base` slot, you inherit the special method, if any, from your base type. You often need to cast your functions to the specific `typedef` type that a field needs (here, the `reprfunc` type for the `tp_repr` field) because the `typedef` has a first argument `PyObject* self`, while your functions—being specific to your type—normally use more specific pointers. For example:

```
static PyObject* mytype_str(mytype* self) { ... /* rest omitted */
```

Alternatively, you can declare `mytype_str` with a `PyObject* self`, then use a cast `(mytype*)self` in the function's body. Either alternative is acceptable style, but it is more common to locate the casts in the `PyTypeObject` declaration.

Instance initialization and finalization

The task of finalizing your instances is split among two functions. The `tp_dealloc` slot must never be `NULL`, except for immortal types (i.e., types whose instances are never deallocated). Python calls *x*`->ob_type->tp_dealloc(`*x*`)` on each instance *x* whose reference count decreases to `0`, and the function thus called must release any resources held by object *x*, including *x*'s memory. When an instance of `mytype` holds no other resources that must be released (in particular, no owned references to other Python objects that you would have to `DECREF`), `mytype`'s destructor can be extremely simple:

```
static void mytype_dealloc(PyObject *x)
{
    x->ob_type->tp_free((PyObject*)x);
}
```

The function in the `tp_free` slot has the specific task of freeing *x*'s memory. Often, you can just put in slot `tp_free` the address of the C API function `PyObject_Del`.

The task of initializing your instances is split among three functions. To allocate memory for new instances of your type, put in slot `tp_alloc` the C API function `PyType_GenericAlloc`, which does absolutely minimal initialization, clearing the newly allocated memory bytes to `0` except for the type pointer and reference count. Similarly, you can often set the field `tp_new` to the C API function `PyType_GenericNew`. In this case, you can perform all per-instance initialization in the function you put in slot `tp_init`, which has the signature:

```
int init_fn_name(PyObject *self, PyObject *orgs, PyObject *kwds)
```

The positional and named arguments to the function in slot `tp_init` are those passed when calling the type to create the new instance, just as, in Python, the positional and named arguments to `__init__` are those passed when calling the class. Again, just like for types (classes) defined in Python, the general rule is to do as little initialization as feasible in `tp_new` and do as much as possible in `tp_init`. Using `PyType_GenericNew` for `tp_new` accomplishes this. However, you can choose to define your own `tp_new` for special types, such as ones that have immutable instances, where initialization must happen earlier. The signature is:

```
PyObject* new_fn_name(PyObject *subtype, PyObject *args, PyObject *kwds)
```

The function in `tp_new` returns the newly created instance, normally an instance of *`subtype`* (which may be a subtype of yours). The function in `tp_init`, on the other hand, must return `0` for success, or `-1` to indicate an exception.

If your type is subclassable, it is important that any instance invariants be established before the function in `tp_new` returns. For example, if it must be guaranteed that a certain field of the instance is never `NULL`, that field must be set to a non-`NULL` value by the function in `tp_new`. Subtypes of your type might fail to call your `tp_init` function; therefore, such indispensable initializations, needed to establish type invariants, should always be in `tp_new` for subclassable types.

Attribute access

Access to attributes of your instances, including methods (as covered in "Attribute Reference Basics" in Chapter 4), goes through the functions in slots `tp_getattro` and `tp_setattro` of your `PyTypeObject struct`. Normally, you use the standard C API functions `PyObject_GenericGetAttr` and `PyObject_GenericSetAttr`, which implement standard semantics. Specifically, these API functions access your type's methods via the slot `tp_methods`, pointing to a sentinel-terminated array of `PyMethodDef structs`, and your instances' members via the slot `tp_members`, a sentinel-terminated array of `PyMemberDef structs`:

```
typedef struct {
    char* name;     /* Python-visible name of the member */
    int type;       /* code defining the data type of the member */
    int offset;     /* member's offset in the per-instance struct */
    int flags;      /* READONLY for a read-only member */
```

```
    char* doc;      /* docstring for the member */
} PyMemberDef;
```

As an exception to the general rule that including *Python.h* gets you all the declarations you need, you have to include *structmember.h* explicitly to have your C source see the declaration of `PyMemberDef`.

`type` is generally `T_OBJECT` for members that are `PyObject*`, but many other type codes are defined in *Include/structmember.h* for members that your instances hold as C-native data (e.g., `T_DOUBLE` for `double`, or `T_STRING` for `char*` encoded in UTF-8). For example, say that your per-instance `struct` is something like this:

```
typedef struct {
    PyObject_HEAD
    double datum;
    char* name;
} mytype;
```

To expose to Python the per-instance attributes *`datum`* (read/write) and *`name`* (read-only), define the following array and point your `PyTypeObject`'s `tp_members` to it:

```
static PyMemberDef[] mytype_members = {
    {"datum", T_DOUBLE, offsetof(mytype, datum), 0, "Current datum"},
    {"name", T_STRING, offsetof(mytype, name), READONLY, "Datum name"},
    {NULL},
};
```

Using `PyObject_GenericGetAttr` and `PyObject_GenericSetAttr` for `tp_getattro` and `tp_setattro` also provides further possibilities; see the online docs for more details.

`tp_getset` points to a sentinel-terminated array of `PyGetSetDef` `struct`s, the equivalent of having `property` instances in a Python-coded class.

If your `PyTypeObject`'s `tp_dictoffset` field is not equal to `0`, the field's value must be the offset, within the per-instance `struct`, of a `PyObject*` pointing to a Python dictionary. In this case, the generic attribute access API functions use that dictionary to let Python code set arbitrary attributes on your type's instances, just like for instances of Python-coded classes.

Use the `tp_dict` field of your `PyTypeObject` `struct` to specify a per-type (*not* per-instance) dictionary. You can set slot `tp_dict` to `NULL`, and `PyType_Ready` will initialize the dictionary appropriately. Alternatively, you can set `tp_dict` to a dictionary of type attributes; `PyType_Ready` will then add other entries to that same dictionary, in addition to the type attributes you set. It is generally easier to start with `tp_dict` set to `NULL`, call `PyType_Ready` to create and initialize the per-type dictionary, and then, if need be, add any further entries to the dictionary via explicit C code.

The `tp_flags` field is a `long` whose bits determine your type `struct`'s exact layout, mostly for backward compatibility; set this field to `Py_TPFLAGS_DEFAULT` to indicate

that you are defining a normal, modern type. If your type supports cyclic garbage collection, set `tp_flags` to `Py_TPFLAGS_DEFAULT|Py_TPFLAGS_HAVE_GC`. Your type should support cyclic garbage collection if instances of the type contain `PyObject*` fields that might point to arbitrary objects and form part of a reference loop. To enable this support, it's not enough to add `Py_TPFLAGS_HAVE_GC` to the `tp_flags` field; you also have to supply appropriate functions, indicated by the fields `tp_traverse` and `tp_clear`, and register and unregister your instances appropriately with the cyclic garbage collector. Supporting cyclic garbage collection is an advanced subject, and we don't cover it further in this book; see the online docs for details. Similarly, we don't cover the advanced subject of supporting weak references, also well covered online.

The field `tp_doc`, a `char*`, is a nul-terminated character string that is your type's docstring.

Other fields point to `structs` (whose fields point to functions); you can set each such field to `NULL` to indicate that you support none of those functions. The fields pointing to such blocks of functions are: `tp_as_number`, for special methods typically supplied by numbers; `tp_as_sequence`, for special methods typically supplied by sequences; `tp_as_mapping`, for special methods typically supplied by mappings; and `tp_as_buffer`, for the special methods of the buffer protocol.

For example, objects that are not sequences can still support one or some of the methods listed in the block to which `tp_as_sequence` points, and in this case the `PyTypeObject struct` must have a non-`NULL` `tp_as_sequence` field, even if the block of function pointers it points to is in turn mostly full of `NULL`s. An example of this would be Python's `dict` class, which supplies a `__contains__` special method so that you can check if *x* is in *d* when *d* is a dictionary. At the C code level, the method is a function pointed to by the field `sq_contains`, which is part of the `PySequenceMethods struct` to which the field `tp_as_sequence` points. Therefore, the `PyTypeObject struct` for the `dict` type, named `PyDict_Type`, has a non-`NULL` value for `tp_as_sequence`, even though a dictionary supplies no other field in `PySequenceMethods` than `sq_contains`, and therefore all other fields in `*(PyDict_Type.tp_as_sequence)` are `NULL`.

Type definition example

Example 25-2 is a complete Python extension module that defines the very simple type `intpair`, each instance of which holds two integers named `first` and `second`.

Example 25-2. Defining a new `intpair` type

```
#include "Python.h"
#include "structmember.h"
/* per-instance data structure */
typedef struct {
    PyObject_HEAD
```

```
    int first, second;
} intpair;
static int
intpair_init(PyObject *self, PyObject *args, PyObject *kwds)
{
    static char* nams[] = {"first","second",NULL};
    float first_arg, second_arg;
    if(!PyArg_ParseTupleAndKeywords(args, kwds, "ff", nams,
                                    &first_arg, &second_arg))
            return -1;
    ((intpair*)self)->first = (int)first_arg;
    ((intpair*)self)->second = (int)second_arg;
    return 0;
}
static void
intpair_dealloc(PyObject *self)
{
    self->ob_type->tp_free(self);
}
static PyObject*
intpair_str(PyObject* self)
{
    return PyUnicode_FromFormat("intpair(%d,%d)",
        ((intpair*)self)->first, ((intpair*)self)->second);
}
static PyMemberDef intpair_members[] = {
    {"first", T_INT, offsetof(intpair, first), 0, "first item" },
    {"second", T_INT, offsetof(intpair, second), 0, "second item" },
    {NULL}
};
static PyTypeObject t_intpair = {
    PyObject_HEAD_INIT(0)               /* tp_head */
    "intpair.intpair",                  /* tp_name */
    sizeof(intpair),                    /* tp_basicsize */
    0,                                  /* tp_itemsize */
    intpair_dealloc,                    /* tp_dealloc */
    0,                                  /* tp_print */
    0,                                  /* tp_getattr */
    0,                                  /* tp_setattr */
    0,                                  /* tp_compare */
    intpair_str,                        /* tp_repr */
    0,                                  /* tp_as_number */
    0,                                  /* tp_as_sequence */
    0,                                  /* tp_as_mapping */
    0,                                  /* tp_hash */
    0,                                  /* tp_call */
    0,                                  /* tp_str */
    PyObject_GenericGetAttr,            /* tp_getattro */
    PyObject_GenericSetAttr,            /* tp_setattro */
    0,                                  /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT,
    "two ints (first,second)",
```

```
    0,                          /* tp_traverse */
    0,                          /* tp_clear */
    0,                          /* tp_richcompare */
    0,                          /* tp_weaklistoffset */
    0,                          /* tp_iter */
    0,                          /* tp_iternext */
    0,                          /* tp_methods */
    intpair_members,            /* tp_members */
    0,                          /* tp_getset */
    0,                          /* tp_base */
    0,                          /* tp_dict */
    0,                          /* tp_descr_get */
    0,                          /* tp_descr_set */
    0,                          /* tp_dictoffset */
    intpair_init,               /* tp_init */
    PyType_GenericAlloc,        /* tp_alloc */
    PyType_GenericNew,          /* tp_new */
    PyObject_Del,               /* tp_free */
};
static PyMethodDef no_methods[] = { {NULL} };
static char intpair_docs[] =
    "intpair: data type with int members .first, .second\n";
static struct PyModuleDef intpair_module = {
    PyModuleDef_HEAD_INIT,
    "intpair",
    intpair_docs,
    -1,
    no_methods
};

PyMODINIT_FUNC
PyInit_intpair(void)
{
    PyObject* this_module = PyModule_Create(&intpair_module);
    PyType_Ready(&t_intpair);
    PyObject_SetAttrString(this_module, "intpair", (PyObject*)&t_intpair);
    return this_module;
}
```

The `intpair` type defined in Example 25-2 gives minimal benefits when compared to an equivalent definition in Python, such as:

```
    class intpair:
        __slots__ = ('first', 'second')
        def __init__(self, first, second):
            self.first = first
            self.second = second
        def __repr__(self):
            return f'intpair({self.first},{self.second})'
```

The C-coded version does, however, ensure that the two attributes are integers, truncating floating-point or complex number arguments as needed (in Python, you could approximate that functionality by passing the arguments through `int`, but it wouldn't be quite the same thing, as Python would then also accept argument values such as the string `'23'`, while the C version wouldn't).[2] For example:

```
import intpair
x=intpair.intpair(1.2,3.4)          # x is: intpair(1,3)
print(x.first, x.second)            # prints '1 3'
```

The C-coded version of `intpair` occupies a little less memory than the Python version. However, the purpose of Example 25-2 is purely tutorial: to present a C-coded Python extension that defines a simple new type.

Extending Python Without Python's C API

You can code Python extensions in other classic compiled languages besides C. For Fortran, we recommend Pearu Peterson's F2PY. This is now part of NumPy, since Fortran is often used for numeric processing. If, as recommended for numerical computations, you have installed `numpy`, then you don't need to also separately install `f2py`.

For C++, you could of course use the same approaches as for C (adding `extern "C"` where needed). Many C++-specific alternatives exist; out of them all, SIP, CLIF, and `cppyy` (with full support for PyPy as well as CPython) appear to be the ones most actively maintained and supported. If your C++ compiler offers a fully compliant implementation of C++11 (which, more than a decade since that standard's release, is likely to be the case), `pybind11` is also well worth considering; its documentation explains its advantages clearly and in detail.

A popular alternative to the C API is Cython, a "Python dialect" focused on generating C code from Python-like syntax, with a few additions centered on precise specification of the C-level side of things. We highly recommend it, and cover it in "Cython" on page 38.

A new project with much promise for the future is HPy, a direct alternative to Python's C API with zero overhead and many advantages (easy, fast portability to PyPy and other Python implementations, a "debug mode" to help your development, an arguably more elegant design, and so forth). HPy is currently in an early alpha release stage of development, so we cannot (yet!) recommend it for production use; however, if you know how to program in C, we do recommend you read HPy's documentation, **`pip install hpy`** (in a virtual environment, of course), and explore it a bit.

2 Calling `operator.index` instead of `int` would be closer to the C-coded funtionality.

CFFI

Another excellent alternative to the Python C API is CFFI, the C Foreign Function Interface for Python. It originated in the PyPy project, and is perfect for it, but also fully supports CPython.

You can use CFFI in various modes—*in-line* or *out-of-line*, and, separately, *ABI* or *API*:

in-line mode
: Everything gets set up anew each time you import your Python module using CFFI.

out-of-line mode
: A separate step prepares an extension module, and then you import that extension module from your Python code.

ABI mode
: CFFI accesses existing C libraries (dynamic ones, typically with the extension *.dll* in Windows, *.so* in Linux, and *.dylib* in macOS; or static ones, typically with the extension *.lib* in Windows and *.a* in Unix-like systems) at the binary level.

API mode
: With better speed and robustness, you can use CFFI in *API mode*, in which CFFI accesses C libraries by generating, then compiling, C code. This mode also lets you use C libraries for which you have source code (that is, *.h* and *.c* files) rather than binary forms already compiled (i.e., dynamic or static libraries).

API mode needs access to a C compiler for your machine, but nowadays that isn't really a problem, since good, free C compilers are available for just about every environment (for a few suggestions, see "What C compiler do you need?" on page 3, earlier in this chapter).

When used in in-line or ABI mode, CFFI roughly matches the functionality of the standard Python library module `ctypes`, discussed briefly in the next section, but with greater speed and reliability (e.g., fewer mysterious crashes due to minor, hard-to-diagnose errors). When you use the out-of-line and API modes, it's usually even preferable (much faster, and *much* more reliable). CFFI's excellent documentation explains all of this clearly, and includes many instructive examples.

ctypes

The `ctypes` module of the Python standard library lets you load existing C-coded dynamic libraries and call functions defined in those libraries from your Python code. The module is popular because it allows handy "hacks" without requiring access to a C compiler or even a third-party Python extension. However, the resulting programs can often be fragile, platform-dependent, and hard to port to

other versions of the same libraries, since they rely on details of the libraries' binary interfaces.

We recommend avoiding `ctypes` in production code, and using instead one of the excellent alternatives covered or mentioned in this chapter.

Cython

The Cython language offers a convenient way to write extensions for Python.[3] The Cython website provides excellent documentation of all details of Cython programming; in this section; we cover only a few essentials to help you get started.

Cython is a nearly complete subset of Python (with a declared intention of eventually becoming a complete superset of it); there are just four marginal, minor differences that the Cython authors do not intend to fix. To this vast subset of Python, it then adds a few statements that allow C-like declarations, and optional C-like types for variables. You can automatically compile Cython programs (source files with the extension *.pyx*) into efficient machine code (via an intermediate C code generation step), producing Python-importable extensions.

Here is a simple example; code it in the source file *hello.pyx* in a new empty directory:

```
def hello(char *name):
    return 'Hello, ' + name + '!'
```

This is almost exactly like Python, except that the parameter `name` is preceded by `char*`, declaring that its type must always be a C `0`-terminated string. As you can see from the body, in your Cython code, you can use its value as a normal Python string.

When you install Cython (**`pip install cython`**), you also gain a way to build Cython source files into Python extensions via `setuptools`. Code the following in the file *setup.py* in your new directory:

```
from setuptools import setup
from Cython.Build import cythonize

setup(name='hello', ext_modules=cythonize('hello.pyx'))
```

Now run **`python setup.py install`** in the new directory (ignore warnings; they're expected and benign), and import and use the new module—for example, from an interactive Python session:

```
>>> import hello
>>> hello.hello('Alex')
'Hello, Alex!'
```

3 Cython supports CPython "natively"; however, it's also usable with PyPy, with several caveats.

The cdef and cpdef statements and function parameters

You can use the Cython keyword `cdef` mostly as you would **`def`**, but `cdef` defines functions that are internal to the extension module, not visible on the outside, while **`def`** functions can also be called by Python code that imports the module. `cpdef` functions can be called both internally (with speed very close to `cdef` ones) and by external Python (just like **`def`** ones), but are otherwise identical to `cdef` functions.

For any kind of function, parameters and return values with unspecified types—or, even better, ones explicitly typed as `object`—become `PyObject*` pointers in the generated C code (with implicit and standard handling of reference incrementing and decrementing). `cdef` functions may also have parameters and return values of any other C type; **`def`** functions, in addition to untyped (or, equivalently, `object`) arguments, can only accept `int`, `float`, and `char*` types. For example, here's a `cdef` function to specifically sum two integers:

```
cdef int sum2i(int a, int b):
    return a + b
```

You can also use `cdef` to declare C variables—scalars, arrays, and pointers—pretty much like in C:

```
cdef int x, y[23], *z
```

and declare `structs`, `unions`, and `enums` Pythonically (end the head clause with a colon, then indent):

```
cdef struct Ural:
    int x, y
    float z
```

Afterward, refer to the new type by name only—here, `Ural`. Never use the keywords `struct`, `union`, or `enum`, except in the `cdef` declaring the type.

External declarations To interface with external C code, you can declare variables as `cdef extern`, with the same effect that `extern` has in the C language. Usually, the C declarations of a library you want to use are in a *.h* C header file; to ensure that the Cython-generated C code includes that header file, use the following `cdef`:

```
cdef extern from "someheader.h":
```

and follow it with a block of indented `cdef`-style declarations (*without* repeating the `cdef` in the block). Only declare functions and variables that you want to use in your Cython code. Cython does not read the C header file—it trusts your Cython declarations in the block, not generating any C code for them. Cython implicitly uses the Python C API, covered at the start of this chapter, but you can explicitly access any of its functions. For example, if your Cython file contains:

```
cdef extern from "Python.h":
    object PyUnicode_FromStringAndSize(char *, int)
```

the following Cython code can use `PyUnicode_FromStringAndSize`. This may come in handy, since, by default, C "strings" are deemed to be terminated by a zero character, but with this function you may instead explicitly specify a C string's length and also get any zero character(s) it may contain.

Conveniently, Cython lets you group such declarations in *.pxd* files (roughly analogous to C's *.h* files, while *.pyx* Cython files are roughly analogous to C's *.c* files). *.pxd* files can also include `cdef inline` declarations, to be inlined at compile time. A *.pyx* file can import the declarations in a *.pxd* file by using the keyword `cimport`, analogous to Python's **`import`**.

Cython comes with several useful *.pxd* files in its *Cython/includes* directory. In particular, the *.pxd* file *cpython* already has all the useful `cdef extern from "Python.h"` declarations: just `cimport cpython` to access them.

cdef classes A `cdef` **`class`** statement lets you define a new Python type in Cython. It may include `cdef` declarations of attributes (which apply to every instance, not to the type as a whole), which are normally invisible from Python code; however, you can specifically declare attributes as `cdef public` to make them normal attributes from Python's viewpoint, or `cdef readonly` to make them visible but read-only from Python (Python-visible attributes must be numbers, strings, or `objects`).

A `cdef` class supports special methods (with some caveats), properties (with a special syntax), and (single-only) inheritance. To declare a property, use the following in the body of the `cdef` **`class`** statement:

```
property name:
```

followed by indented **`def`** statements for methods `__get__(self)` and, optionally, `__set__(self, value)` and `__del__(self)`.

A `cdef` class's `__new__` is different from that of a normal Python class: the first argument is `self`, the new instance, already allocated and with its memory filled with `0`s. Cython always calls the special method `__cinit__(self)` right after the instance allocation, to allow further initialization; `__init__`, if defined, is called next. At object destruction time, Cython calls the special method `__dealloc__(self)` to let you undo whatever allocations `__new__` and/or `__cinit__` have done (`cdef` classes have no `__del__` special method).

There are no righthand-side versions of arithmetic special methods, such as `__radd__` to go with `__add__`, like in Python; rather, if (say) `a + b` can't find or use `type(a).__add__`, it next calls `type(b).__add__(a, b)`. Note the order of arguments: there's *no* swapping! You may need to attempt some type checking to ensure that you perform the correct operation in all cases.

To make the instances of a `cdef` class into iterators, define a special method `__next__(self)`.

Here is a Cython equivalent of Example 25-2:

```
cdef class intpair:
    cdef public int first, second
    def __init__(self, first, second):
        self.first = first
        self.second = second
    def __repr__(self):
        return f'intpair({self.first}, {self.second})'
```

Like the C-coded extension in Example 25-2, this Cython-coded extension offers no substantial advantage with respect to a Python-coded equivalent. The simplicity and conciseness of the Cython code is much closer to that of Python than to the verbosity and boilerplate needed in C; yet, the machine code generated from this Cython file is very close to what gets generated from the C code in Example 25-2.

The ctypedef statement

You can use the keyword `ctypedef` to declare a name (synonym) for a type, e.g.:

```
ctypedef char* string
```

The for…from statement

In addition to the usual Python **`for`** statements, Cython has another form of **`for`**:

```
for variable from lower_expression <= variable < upper_expression:
```

This is the most common form, but you could use either `<` or `<=` on either side of the *`variable`* after the **`from`** keyword; alternatively, you could use `>` and/or `>=` to have a backward loop (you cannot mix a `<` or `<=` on one side and `>` or `>=` on the other).

The **`for...from`** statement is faster than the usual Python **`for`** *`variable`* **`in`** `range(...):`, when the variable and loop boundaries are C-kind `int`s. However, in modern Cython, **`for`** *`variable`* **`in`** `range(...):` is optimized to near-equivalence to **`for...from`**, so the classic Pythonic **`for`** *`variable`* **`in`** `range(...):` can usually be chosen, for simplicity and readability.

Cython expressions

In addition to Python expression syntax, Cython can use some, but not all, of C's additions to it. To take the address of variable *`var`*, use `&`*`var`*, like in C. To dereference a pointer *`p`*, however, use *`p`*`[0]`; the equivalent C syntax `*`*`p`* is not valid Cython. Where in C you would use *`p`*`->`*`q`*, use *`p`*`.`*`q`* in Cython. The null pointer uses the Cython keyword `NULL`. For `char` constants, use the syntax `c'x'`. For casts, use angle brackets, such as `<int>`*`somefloat`* where in C you would code `(int)`*`somefloat`*; also, use casts only on C values and onto C types, *never* with Python values and types (let Cython perform type conversion for you automatically when Python values or types occur).

A Cython example: Greatest common divisor

Euclid's algorithm for finding the greatest common divisor (GCD) of two numbers is quite simple to implement in pure Python:

```
def gcd(dividend, divisor):
    remainder = dividend % divisor
    while remainder:
        dividend = divisor
        divisor = remainder
        remainder = dividend % divisor
    return divisor
```

The Cython version is almost identical:

```
def gcd(int dividend, int divisor):
    cdef int remainder
    remainder = dividend % divisor
    while remainder:
        dividend = divisor
        divisor = remainder
        remainder = dividend % divisor
    return divisor
```

On an old MacBook Air laptop, `gcd(454803, 278255)` takes about 1 microsecond in the Python version, and 0.22 microseconds in CPython. A 350% speedup for so little effort can be well worth the bother (assuming that this function takes up a substantial fraction of your program's execution time), even though the pure Python version has some practical advantages (it runs in PyPy, not just in CPython; it's effortlessly cross-platform; and so on).

Embedding Python

If you have an application in C or C++ (or another classic compiled language), you may want to *embed* Python as your application's scripting language. To embed Python in a language other than C, that language must be able to call C functions (how you do that varies not just by language, but by specific implementation of the language: what compiler, what linker, and so on). In this section, we cover the C view of things; because the details of how to call C functions from other languages vary widely, we don't go into those here.

Installing Resident Extension Modules

For Python scripts to communicate with your application, it must supply extension modules with Python-accessible functions and classes that expose the application's functionality. When, as is normal, these modules are linked with your application (rather than residing in dynamic libraries that Python can load when necessary), you'll need to register your modules with Python as additional built-in modules by calling the `PyImport_AppendInittab` C API function, which has the following signature:

`PyImport_AppendInittab`	`int PyImport_AppendInittab(char* name, void (*initfunc)(void))` *name* is the module name, which Python scripts use to **`import`** the module. *initfunc* is the module initialization function, taking no argument and returning no result, as covered in "The Initialization Module" on page 6 (i.e., *initfunc* is the module's function that would be named `init`*name* for a normal extension module in a dynamic library). Call `PyImport_AppendInittab` *before* `Py_Initialize` (described in Table 25-12).

Setting Arguments

You may want to set the program name and arguments, which Python scripts can access as `sys.argv`, by calling either or both of the C API functions listed in Table 25-11.

Table 25-11. C API functions for setting arguments

`Py_SetProgramName`	`void Py_SetProgramName(char* name)` Sets the program name, which Python scripts can access as `sys.argv[0]`. `Py_SetProgramName` must be called *before* `Py_Initialize`.
`PySys_SetArgv`	`void PySys_SetArgv(int argc, char** argv)` Sets the program arguments, which Python scripts can access as `sys.argv[1:]`, to the *argc* 0-terminated strings in array *argv*. `PySys_SetArgv` must be called *after* `Py_Initialize`.

Python Initialization and Finalization

After installing extra built-in modules and optionally setting the program name, your application initializes Python. At the end, when Python is no longer needed, your application finalizes Python. The relevant C API functions are listed in Table 25-12.

Table 25-12. C API functions for initializing and finalizing Python

`Py_Initialize`	`void Py_Initialize(void)` Initializes the Python environment. Do not make any other Python C API call before this one, except `PyImport_AppendInittab` and `Py_SetProgramName`; those functions must be called before `Py_Initialize`.
`Py_Finalize`	`void Py_Finalize(void)` Frees all memory and other resources that Python is able to free. Do not make any other Python C API call after calling this function.

Running Python Code

Your application can run Python source code from a character string or from a file. To run or compile Python source code, choose one of the constants defined in *Python.h*, listed in Table 25-13.

Table 25-13. Python.h constants

`Py_eval_input`	The code is an expression to evaluate (like passing `'eval'` to the Python built-in function `compile`).
`Py_file_input`	The code is a block of one or more statements to execute (like `'exec'` for `compile`; just like in that case, a trailing `'\n'` must close compound statements).
`Py_single_input`	The code is a single statement for interactive execution (like `'single'` for `compile`; implicitly outputs the results of expression statements).

Running Python source code is similar to passing a source code string to Python's `exec` or `eval` (and entails the same security risks if that source code comes from somewhere you do not totally, completely trust, as discussed in Chapter 14). Table 25-14 describes two general functions you can use for this task.

Table 25-14. C API functions for running Python source code

`PyRun_File`	`PyObject* PyRun_File(FILE*` *`fp`*`, char*` *`filename`*`, int` *`start`*`, PyObject*` *`globals`*`, PyObject*` *`locals`*`)` *`fp`* is a file of Python source code open for reading. *`filename`* is the name of the file, to use in error messages. *`start`* is one of the constants `Py_..._input` that define the execution mode. *`globals`* and *`locals`* are `dicts` to use as global and local namespaces for the execution (you may use the same `dict` twice). `PyRun_File` returns the result of the expression when *`start`* is `Py_eval_input`, a new reference to `Py_None` otherwise, or `NULL` to indicate that an exception has been raised.
`PyRun_String`	`PyObject* PyRun_String(char*` *`astring`*`, int` *`start`*`, PyObject*` *`globals`*`, PyObject*` *`locals`*`)` Like `PyRun_File`, but the source is in the NUL-terminated string *`astring`*.

The dictionaries *`locals`* and *`globals`* are often new, empty `dicts` (conveniently built by `Py_BuildValue("{}")`), or the dictionary of a module. `PyImport_Import` is a convenient way to get an existing module object; `PyModule_GetDict` gets a module's dictionary.

When you want to create a new module object on the fly, often in order to populate it with `PyRun_` calls, use the `PyModule_New` C API function:

`PyModule_New`	`PyObject* PyModule_New(char& name)`
	Returns a new, empty module object for a module named *name*. Before using the new object, add to the object a string attribute named `__file__`. For example:
	`PyObject* newmod = PyModule_New("mymodule");`
	`PyModule_AddStringConstant(newmod, "__file__", "<synth>");`
	After this code runs, the module object *newmod* is ready; you can get the module's `dict` with `PyModule_GetDict(newmod)` and pass the `dict` to such functions as `PyRun_String` as the *globals* and possibly the *locals* argument.

To run Python code repeatedly, and to separate the diagnosis of syntax errors from that of runtime exceptions raised by the code when it runs, you can compile the Python source to a code object, then keep the code object and run it repeatedly. This is just as true when using the C API as when dynamically executing in Python, as covered in "Dynamic Execution and exec" in Chapter 14. Table 25-15 lists two C API functions you can use for this task.

Table 25-15. C API functions for compiling Python source code

`Py_CompileString`	`PyObject* Py_CompileString(char* code, char* filename, int start)` *code* is a NUL-terminated string of source code. *filename* is the name of the file to use in error messages. *start* is one of the constants that define execution mode. `Py_CompileString` returns the Python code object that contains the bytecode, or `NULL` for syntax errors.
`PyEval_EvalCode`	`PyObject* PyEval_EvalCode(PyObject* co, PyObject* globals, PyObject* locals)` *co* is a Python code object, as returned by `Py_CompileString`, for example. *globals* and *locals* are `dicts` (possibly the same `dict`) to use as the global and local namespaces for the execution. `PyEval_EvalCode` returns the result of the expression when *co* was compiled with `Py_eval_input`, a new reference to `Py_None` otherwise, or `NULL` to indicate the execution has raised an exception.