24. Extending and Embedding Classic Python

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:

```
                                                  module
#include <Python.h>/* omitted: the body of the x*/        PyMODINIT_FUNCPyInit_x(void)
{    /* omitted: the code that initializes the module named x */}
```

In v2, the module initialization function’s name is `initx`; use C preprocessor constructs such as `#if PY_MAJOR_VERSION >= 3` if you’re coding extensions meant to compile for both v2 and v3. For example, the module initialization function, for such a portable extension, might start:

```
PyMODINIT_FUNC
#if PY_MAJOR_VERSION >=
3
PyInit_x(void)
#else
initx(void)
#endif
{
```

When you have built and installed the extension module, a 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 distribution utilities, `distutils` (or its improved third-party variant, `setuptools`). In the same directory as *x.c*, place a file named *setup.py* that contains the following statements:

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

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

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

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

`setuptools` performs compilation and linking steps, with the right compiler and linker commands and flags, and copies the resulting dynamic library into the right directory, dependent on your Python installation (depending on that installation's details, you may need to have administrator or superuser privileges for the installation; for example, on a Mac or Linux, you may run `sudo python setup.py install`, although using `venv` instead is more likely to be what serves you 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 consider `clang`, widely reputed to offer better error messages.) On the Macintosh, *gcc* (actually a frontend to `clang`) comes with Apple's free XCode (AKA Developer Tools) integrated development environment (IDE), which you may download and install separately from Apple's App Store.

For Windows, you ideally need the Microsoft product known as VS2015 for v3, VS2010 for v2. However, other versions of Microsoft Visual Studio may also work.

Compatibility of C-coded extensions among Python versions

In general, a C-coded extension compiled to run with one version of Python has not been guaranteed to run with another. For example, a version compiled for Python 3.4 is only certain to run with 3.4, not with 3.3 or 3.5. On Windows, you cannot even try to run an extension with a different version of Python; elsewhere, such as on Linux or macOS, a given extension may happen to work on more than one version of Python, but you may get a warning when the module is imported, and the prudent course is to heed the warning: recompile the extension appropriately. (Since Python 3.5, you should be able to compile extensions forward-compatibly.)

At a C-source level, on the other hand, compatibility is almost always preserved within a major version (though not between v2 and v3).

Overview of C-Coded Python Extension Modules

Your C function `PyInit_x` generally has the following overall structure (from now on, we chiefly cover v3; see the v2 tutorial and reference for slight differences):

```
PyMODINIT_FUNCPyInit_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
done:                                  return m;}
```

More details are covered in “The Initialization Module”. x_module is a struct like:

```
static struct PyModuleDef x_module = {
   PyModuleDef_HEAD_INIT,
             /* the name of the module
   "x",      */
             /* the module's docstring, may be NULL
   x_doc,    */
   -1,
/* size of per-interpreter state of the module, or -
1
                if the module keeps state in global variables.
*/
             /* array of the module's method definitions
   x_methods */
};
```

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”. How such functions build Python objects is covered in “Creating Python Values”, and how they raise or propagate exceptions back to the Python code that called them is covered in Chapter 5. 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”.

A simple example using all these concepts is shown in “A Simple Extension Example”. A toy-level “Hello World” example module could be as simple as:

```
#include
<Python.h>

static PyObject*
hello(PyObject* self)
{
                                "Hello, Python extensions
    return Py_BuildValue("s", 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, which in v2 is also UTF-8 encoded. As previously mentioned, this is for v3. For the slight differences in module initialization in v2, see the online docs; for this trivial extension, all you need is to guard the whole definition of `helloworld_module` in a `#if PY_MAJOR_VERSION >= 3` / `#endif` (since in v2 there is no such type as `PyModuleDef`), and change the module initialization function accordingly, to:

```
PyMODINIT_FUNC
#if PY_MAJOR_VERSION >=
3
PyInit_hello(void)
{
    return PyModule_Create(&hello_module
);
#else
inithello(void)
{
    Py_InitModule3("hello",
        hello_funcs, hello_docs);
#endif
}
```

Save this as *hello.c* and build it through a *setup.py* script with `distutils`, such as:

```
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—such as:

```
                                       Hello, Python extensions
>>> import hello>>> print hello.hello()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 `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 5 for more details on how C-coded functions handle and raise errors.

The Initialization Module

The `PyInit_`*`x`* function must contain, at a minimum, a call to the function `Py_Module_Create`, (or, since 3.5, `PyModuleDef_Init`), with, as the only parameter, the address of the `struct PyModuleDef` that defines the module's details. In addition, it may have one or more calls to the functions listed in Table 24-1, all returning `-1` on error, `0` on success.

Table 24-1.

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".
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).

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`. 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__`.

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") 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"), you must describe the functions or methods in an array of `PyMethodDef structs`, and terminate the array with a *sentinel* (i.e., a structure 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 of the type `PyObject*`. Each `PyObject` counts how many references to itself are outstanding and destroys itself when the number of references goes down to `0`. 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 need to 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, the following general rules apply:

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

For each of the two 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 general rules that item would be a borrowed reference. `PyList_SET_ITEM` and `PyTuple_SET_ITEM`, the C preprocessor macros, which implement faster versions of the item-setting functions, are also reference-stealers. So is `PyModule_AddObject`, covered in Table 24-1. 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 save 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 resulting 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 show all cases of exceptions to these rules (i.e., return of borrowed references and rare cases of reference stealing from arguments) regarding all functions we cover. When we don't explicitly mention a function as being an exception, the function follows the rules: its `PyObject*` arguments, if any, are borrowed references, and its `PyObject*` result, 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:

PyArg_ParseTuple

```
int                                      ,
PyArg_ParseTuple(PyObject*     tuplechar*   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 following 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 24-2 lists the commonly used code strings, of which zero or more are joined to form string `format`.

Table 24-2. Format codes for PyArg_ParseTuple

Code	C type	Meaning
`c`	`int`	A Python `bytes`, `bytearray`, or `str` of length `1` becomes a C `int`(`char`, in v2).
`d`	`double`	A Python `float` becomes a C `double`.
`D`	`Py_Complex`	A Python `complex` becomes a C `Py_Complex`.
`f`	`float`	A Python `float` becomes a C `float`.
`i`	`int`	A Python `int` becomes a C `int`.
`l`	`long`	A Python `int` becomes a C `long`.

Code	C type	Meaning
`L`	`long long`	A Python `int` becomes a C `long long` (`__int64` on Windows).
`O`	`PyObject*`	Gets non-`NULL` borrowed reference to Python argument.
`O!`	`type+PyObject*`	Like code `O`, plus type checking (see below).
`O&`	`convert+void*`	Arbitrary conversion (see below).
`s`	`char*`	Python string without embedded nulls to C `char*` (encoded in UTF-8).
`s#`	`char*+int`	Any Python string to C address and length.
`t#`	`char*+int`	Read-only single-segment buffer to C address and length.
`u`	`Py_UNICODE*`	Python Unicode without embedded nulls to C.
`u#`	`Py_UNICODE*+int`	Any Python Unicode C address and length.
`w#`	`char*+int`	Read/write single-segment buffer to C address and length.
`z`	`char*`	Like `s`, but also accepts `None` (sets C `char*` to `NULL`).
`z#`	`char*+int`	Like `s#`, but also accepts `None` (sets C `char*` to `NULL` and `int` to `0`).
`(...)`	as per `...`	A Python sequence is treated as one argument per item.
`|`		All following arguments are optional.
`:`		Format end, followed by function name for error messages.
`;`		Format end, followed by entire error message text.

Code formats `d` to `n` (and rarely used other codes for unsigned chars and short ints) accept numeric arguments from Python. Python coerces the corresponding values. For example, a code of `i` can correspond to a Python `float`; the fractional part gets truncated, as if the built-in function `int` had been called. `Py_Complex` is a C struct with two fields named `real` and `imag`, both of type `double`.

`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 5). 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 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 (as covered in Chapter 5) to indicate an error, or return `1` and store whatever is appropriate via the `void*` it gets.

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, null-terminated array of `char`s (i.e., a typical C string; however, your code must *not* modify it). The Python string must contain no embedded null characters; in v3, 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. `u` and `u#` are similar, but specifically accept a Unicode string (in both v3 and v3), and the C-side pointers must be `Py_UNICODE*` rather than `char*`. `Py_UNICODE` is a macro defined in *Python.h*, and corresponds to the type of a Python Unicode character in the implementation (this is often, but not always, a C `wchar_t`).

`t#` and `w#` are similar to `s#`, but the corresponding Python argument can be any object of a type respecting the buffer protocol, respectively read-only and read/write. Strings are a typical example of read-only buffers. `mmap` and `array` instances are typical examples of read/write buffers, and like all read/write buffers they are also acceptable where a read-only buffer is required (i.e., for a `t#`).

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, 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.

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 function must be called with positional arguments only and the arguments must be numeric):

```
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 entire error message if `PyArg_ParseTuple` detects errors (this form is rarely used).

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.

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. `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=0.0, z=0.0):
    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 that describes 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 24-3 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 24-3. 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 Unicode string of length 1 (v3 only).
c	char	A C char becomes a Python bytestring of length 1 (bytes in v3, str in v2).
D	double	A C double becomes a Python float.
d	Py_Complex	A C Py_Complex becomes a Python complex.
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 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 platform supports it).
l	long	A C long becomes a Python int.
N	PyObject*	Passes a Python object and *steals* a reference.
O	PyObject*	Passes a Python object and INCREFs it.
O&	convert+void*	Arbitrary conversion (see below).
s	char*	C 0-terminated char* to Python string (bytes in v2, Unicode decoding with utf8 in v3), or NULL to None.
s#	char*+int	C char* and length to Python string (like s), or NULL to None.
u	Py_UNICODE*	C wide char (UCS-2 or UCS-4), null-terminated string to Python Unicode, or NULL to None.
u#	Py_UNICODE*+int	C wide char (UCS-2 or UCS-4) string and length to Python Unicode, or NULL to None.

Code	C type	Meaning
`y`	`char*+int`	C char null-terminated string to `bytes`, or `NULL` to `None` (v3 only).
`y#`	`char*+int`	C char string and length to `bytes`, or `NULL` to `None` (v3 only).
`(...)`	As per `...`	Builds Python tuple from C values.
`[...]`	As per `...`	Builds Python list from C values.
`{...}`	As per `...`	Builds 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 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 Chapter 5) 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 the 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 including a reference you own that you would otherwise have to `Py_DECREF`. `O` does no reference stealing, so it's appropriate to build an object including 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, set the current-exception indicator, then return `NULL`. Python's built-in exception classes (covered in "Standard Exception Classes") 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 the following:

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 code can just use:

```
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` (normally `NULL`, meaning no extra class attributes or methods), creating a new exception class named `name` (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(
    "mymod.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_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_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:

```
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 like `PyErr_SetFromErrno`.

Your C code may want to deal with an exception and continue, as a `try`/`except` statement would let you do in Python code. The most commonly used C API functions related to catching exceptions are the following:

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, or 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`, in order to catch exceptions of subclasses as well, as is normal and expected.)

PyErr_Print

`void PyErr_Print()`

Call only when an error is pending, or the whole program might crash. Outputs a standard traceback to `sys.stderr`, then clears the error indicator.

If you need to process errors in highly sophisticated 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 and 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_`, respectively).

Many of the functions callable on specifically typed objects within these categories duplicate functionality that is also available from `PyObject_` functions. In these cases, you should almost invariably use the more general `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 24-4.

Table 24-4.

PyCallable_Check	`int PyCallable_Check(PyObject* x)` True when `x` is callable, like `callable(x)`.
PyIter_Check	`int PyIter_Check(PyObject* x)` True when `x` is an iterator.
PyIter_Next	`PyObject* PyIter_Next(PyObject* x)` Returns the next item from iterator `x`. Returns `NULL` *without* raising exceptions when `x`'s iteration is finished (i.e., when `next(x)` raises `StopIteration`).
PyNumber_Check	`int PyNumber_Check(PyObject* x)` True when `x` is a number.
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_CallObject	`PyObject* PyObject_CallObject(PyObject* f,PyObject* args)` Calls the callable Python object `f` with positional arguments in `tuple args` (may be `NULL`). Returns the call's result. Like `f(*args)`.

PyObject_CallFunction	`PyObject* PyObject_CallFunction(PyObject* f, char* format,...)` Calls the callable Python object `f` with positional arguments described by the format string `format`, using the same format codes as `Py_BuildValue`, covered in "Creating Python Values". 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 the format string `format`, using the same format codes as `Py_BuildValue`. 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_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)` True if `x` has an attribute `name`, like `hasattr(x,name)`.
PyObject_IsTrue	`int PyObject_IsTrue(PyObject* x)` True if `x` is true for Python, like `bool(x)`.
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 Python object. `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`.
PyObject_RichCompareBool	`int PyObject_RichCompareBool(PyObject* x,PyObject* y,int op)` Like `PyObject_RichCompare`, but returns `0` for false and `1` for true.
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 (Unicode in v3, bytes in v2), like `str(x)`. To get the result as bytes in v3, use `PyObject_Bytes`; to get the result as `unicode` in v2, use `PyObject_Unicode`.
PyObject_Type	`PyObject* PyObject_Type(PyObject* x)` Returns `x`'s type object, like `type(x)`.
PySequence_Contains	`int PySequence_Contains(PyObject* x,PyObject* v)` True if `v` is an item in `x`, like `v in x`.
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 `x`, unless `x` is a list, in which case 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)` once, 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 the `i` item of `x`, where `x` must be the result of `PySequence_Fast`, `x !=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`, `x !=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 also be a sequence.
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 24-5 with their Python equivalents.

Table 24-5. Unary PyNumber functions

Function	Python equivalent
`PyNumber_Absolute`	`abs(x)`
`PyNumber_Float`	`float(x)`
`PyNumber_Int`	`int(x)` (v2 only)
`PyNumber_Invert`	`~x`
`PyNumber_Long`	`long(x)` (`int(x)` in v3)
`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 24-6.

Table 24-6. Binary PyNumber functions

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

Function	Python equivalent
PyNumber_FloorDivide	x // y
PyNumber_Lshift	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 (nontruncating)
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 Pytype_ (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 some frequently used functions from the concrete layer that provide unique functionality, or very substantial convenience or speed. For most types, you can check whether an object belongs to the type by calling Pytype_Check, which also accepts instances of subtypes, or Pytype_CheckExact, which accepts only instances of type, not of subtypes. Signatures are the same as for function PyIter_Check, covered in Table 24-4.

All functions whose name start with PyString are actually named PyBytes in v3 (and work on Python bytes objects, not str ones), but the names starting with PyString remain available (as synonyms implemented by C preprocessor macros) for convenience. Concrete-layer functions dealing with str in v3 and unicode in v2 have names starting with PyUnicode, but you're usually better off using the corresponding abstract-layer functions instead, so we don't cover the concrete-layer equivalents in this book.

PyDict_GetItem	`PyObject* PyDict_GetItem(PyObject* x,PyObject* key)` Returns a borrowed reference to the item with key key of dictionary x.
PyDict_GetItemString	`int PyDict_GetItemString(PyObject* x,char* key)` Returns a borrowed reference to the item with null-terminated string key key of dictionary x.
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, returns 1; when there are no more items, returns 0. 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 in any way the set of x's keys, but you can change x's values as long as the set of keys remains identical.
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 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.
PyFloat_AS_DOUBLE	`double PyFloat_AS_DOUBLE(PyObject* x)` Returns the C double value of Python float x, very fast, 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_GET_ITEM	`PyObject* PyList_GET_ITEM(PyObject* x,int pos)` Returns the `pos` item of list `x`, without any error checking.
PyList_SET_ITEM	`int PyList_SET_ITEM(PyObject* x,int pos,PyObject* v)` Sets the `pos` item of list `x` to `v`, without any error checking. Steals a reference to `v`. Use only immediately after creating a new list `x` with `PyList_New`.
PyString_AS_STRING	`char* PyString_AS_STRING(PyObject* x)` Returns a pointer to the internal buffer of string `x`, very fast, without any error checking. You must not modify the buffer in any way, unless you just allocated it by calling `PyString_FromStringAndSize(NULL,size)`.
PyString_AsStringAndSize	`int PyString_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`. You must not modify the buffer in any way, unless you just allocated it by calling `PyString_FromStringAndSize(NULL,size)`.
PyString_FromFormat	`PyObject* PyString_FromFormat(char* format,...)` Returns a Python string built from format string `format`, which has syntax similar to `printf`'s, and the following C values indicated as variable arguments (`...`) above.
PyString_FromStringAndSize	`PyObject* PyString_FromFormat(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 `PyString_AS_STRING`.
PyTuple_New	`PyObject* PyTuple_New(int length)` Returns a new, uninitialized tuple of the given `length`. You must then initialize the tuple, typically by calling `PyTuple_SET_ITEM` `length` times.
PyTuple_GET_ITEM	`PyObject* PyTuple_GET_ITEM(PyObject* x,int pos)` Returns the `pos` item of tuple `x`, without error checking.

PyTuple_SET_ITEM	`int PyTuple_SET_ITEM(PyObject* x,int pos, PyObject* v)` Sets the `pos` 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 24-1 exposes the functionality of Python C API functions `PyDict_Merge` and `PyDict_MergeFromSeq2` for Python use. The `update` method of `dict`s works like `PyDict_Merge` with `override=1`, but Example 24-1 is slightly more general.

Example 24-1. 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_TypeError))
            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_TypeError))
            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_KEYWORDS, merge_docs},
    {"mergenew", (PyCFunction)mergenew, METH_KEYWORDS, mergenew_docs},
    {NULL}
};

                                  "Example extension
static char merge_module_docs[] = 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` (in v3; it would be named `initmerge` in v2), which must be visible from the outside 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. Therefore, declaring them `static` is best: this ensures that 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.

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.

And indeed, after the `argnames` argument, the code passes `&PyDict_Type` (i.e., the address of the dictionary type object). Then it passes 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 isn't explicitly passed from Python, is "no overriding." If the return value of `PyArg_ParseTupleAndKeywords` is `0`, then the code immediately returns `NULL` to propagate the exception; this automatically diagnoses most cases where Python code passes wrong arguments to our new function `merge`.

When the arguments appear to be okay, it tries `PyDict_Merge`, which succeeds if `y` is a dictionary. When `PyDict_Merge` raises a `TypeError`, indicating that `y` is not a dictionary, 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 that 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 24-1 must reside 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'
])])
```

Now, run `python setup.py install` at a shell prompt in this directory (with a user ID having appropriate privileges to write into your Python installation, or use `sudo` on Unix-like systems if necessary—or, best, use a virtual environment!). This command builds the dynamically loaded library for the `merge` extension module, and copies it to the appropriate directory for your Python installation. Now Python code (in the appropriate virtual environment, if you have, as recommended, used `venv`) can use the module. For example:

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

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. It also demonstrates the default operation (where keys that are already in the first argument 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 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 in order 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* (v3 only), *xxmodule.c*, and *xxsubtype.c* (the latter focused on subclassing built-in types, with two types, one each subclassing from `list` and `dict`, respectively).

See the online docs for detailed documentation 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 would do well to 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 in order 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's characteristics and behavior 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 the `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 the per-instance 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)
/* use NULL for MSVC++
*/
/* tp_internal                     /* must be 0
*/                0,               */
/* 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                      /* usually 0 (see getattro)
*/                0,               */
/* tp_setattr                      /* usually 0 (see setattro)
*/                0,               */
                                   /* see also richcompare
/* tp_compare*/    0,              */
/* tp_repr                            /* like Python's __repr__
*/                (reprfunc)mytype_str,  */
    /* rest of struct omitted
    */
```

For portability to Microsoft Visual C++, the `PyObject_HEAD_INIT` macro at the start of the `PyTypeObject` must have an argument of `NULL`. During module initialization, you must 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 compatibility with Microsoft Visual C++. However, 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 a built-in 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’s 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 determine instance size once and forever at creation time. Most types just set `tp_itemsize` to `0`. The fields `tp_getattr` and `tp_setattr` are generally set to `NULL` because 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 are omitted 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, or else set the field to point to a function with the needed functionality. If you set the field to `NULL` but also 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:

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

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's 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 resource 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 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_name(PyObject *self,PyObject *args,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, as 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_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’s 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”), 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:

```
                                   /* Python-visible name of the member
typedef struct {    char* name;    */                                        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                    /* docstring for the member
*/                                     char* doc;     */
} 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 in order 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*`). For example, say that your per-instance struct is something like this:

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

Expose to Python the per-instance attributes `datum` (read/write) and `name` (read-only) by defining the following array and pointing your `PyTypeObject`’s `tp_members` to it:

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

Using `PyObject_GenericGetAttr` and `PyObject_GenericSetAttr` for `tp_getattro` and `tp_setattro` also provides further possibilities, which we do not cover in detail in this book. `tp_getset` points to a sentinel-terminated array of `PyGetSetDef` structs, the equivalent of having `property` instances in a Python-coded class. If your PyTypeObject's field tp_dictoffset is not equal to 0, the field's value must be the offset, within the per-instance struct, of a PyObject* that points to a Python dictionary. In this case, the generic attribute access API functions use that dictionary to allow Python code to set arbitrary attributes on your type's instances, just like for instances of Python-coded classes.

Another dictionary is per-type, not per-instance: the PyObject* for the per-type dictionary is slot tp_dict of your PyTypeObject struct. You can set slot tp_dict to NULL, and then PyType_Ready initializes the dictionary appropriately. Alternatively, you can set tp_dict to a dictionary of type attributes, and then PyType_Ready adds other entries to that same dictionary, in addition to the type attributes you set. It's 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.

Field tp_flags 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. Set tp_flags to Py_TPFLAGS_DEFAULT|Py_TPFLAGS_HAVE_GC if your type supports cyclic garbage collection. 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 support cyclic garbage collection, it's not enough to add Py_TPFLAGS_HAVE_GC to field tp_flags; you also have to supply appropriate functions, indicated by the slots 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. Similarly, we don't cover the advanced subject of supporting weak references, also well covered online.

The field tp_doc, a char*, is a null-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 a few of the methods listed in the block to which tp_as_sequence points, and in this case the PyTypeObject must have a non-NULL field tp_as_sequence, even if the block of function pointers it points to is in turn mostly full of NULLs. For example, dictionaries supply a __contains__ special method so that you can check if x 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 except sq_contains, and therefore all other fields in *(PyDict_Type.tp_as_sequence) are NULL.

Type definition example

Example 24-2 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 24-2. Defining a new intpair type

```
#include
"Python.h"
#include
"structmember.h"
```

```
 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};
    int first, second;
    if(!PyArg_ParseTupleAndKeywords(args, kwds, "ii", nams, &first, &second
))
        return -1;
    ((intpair*)self)->first = first;
    ((intpair*)self)->second = second;
    return 0;
}

static void
intpair_dealloc(PyObject *self)
{
    self->ob_type->tp_free(self);
}

static PyObject*
intpair_str(PyObject* self)
{
    return PyString_FromFormat("intpair(%d,%d)",
        ((intpair*)self)->first, ((intpair*)self)->second);
}

static PyMemberDef intpair_members[] = {
                                                  "first
    {"first", T_INT, offsetof(intpair, first), 0, item"        },
    {"second", T_INT, offsetof(intpair, second), 0, "second item" },
    {NULL}
};

static PyTypeObject t_intpair = {
                                        /* tp_head
    PyObject_HEAD_INIT(0)               */
                                        /* tp_internal
    0,                                  */
                                        /* tp_name
    "intpair.intpair",                  */
    sizeof(intpair),                    /* tp_basicsize */
                                        /* tp_itemsize
    0,                                  */
                                        /* tp_dealloc
    intpair_dealloc,                    */
                                        /* tp_print
    0,                                  */
                                        /* tp_getattr
```

```
    0,                              */
    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 */
                                    /* tp_free
```

```
    _PyObject_Del,                      */
};

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 24-2 gives just about no substantial benefits when compared to an equivalent definition in Python, such as:

```
class intpair(object):
    __slots__ = 'first', 'second'
    def __init__(self, first, second):
        self.first = first
        self.second = second
    def __repr__(self):
        return 'intpair(%s,%s)' % (self.first, self.second
)
```

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

```
import intpair
x=intpair.intpair(1.2,3.4)
# x is:
intpair(1,3)
```

The C-coded version of `intpair` occupies a little less memory than the Python version. However, the purpose of Example 24-2 is purely didactic: to present a C-coded Python extension that defines a simple new type.