Abstract

This manual documents the API used by C and C++ programmers who want to write extension modules or embed Python. It is a companion to *Extending and Embedding the Python Interpreter*, which describes the general principles of extension writing but does not document the API functions in detail.

**Warning:** The current version of this document is incomplete. I hope that it is nevertheless useful. I will continue to work on it, and release new versions from time to time, independent from Python source code releases.
# CONTENTS

1 Introduction .......................... 1
   1.1 Include Files .......................... 1
   1.2 Objects, Types and Reference Counts .......................... 2
   1.3 Exceptions .......................... 5
   1.4 Embedding Python .......................... 7

2 The Very High Level Layer ................. 9

3 Reference Counting .......................... 13

4 Exception Handling .......................... 15
   4.1 Standard Exceptions .......................... 18
   4.2 Deprecation of String Exceptions .......................... 19

5 Utilities .......................... 21
   5.1 Operating System Utilities .......................... 21
   5.2 Process Control .......................... 21
   5.3 Importing Modules .......................... 22
   5.4 Data marshalling support .......................... 24
   5.5 Parsing arguments and building values .......................... 25

6 Abstract Objects Layer .......................... 31
   6.1 Object Protocol .......................... 31
   6.2 Number Protocol .......................... 34
   6.3 Sequence Protocol .......................... 37
   6.4 Mapping Protocol .......................... 39
   6.5 Iterator Protocol .......................... 40
   6.6 Buffer Protocol .......................... 40

7 Concrete Objects Layer .......................... 43
   7.1 Fundamental Objects .......................... 43
   7.2 Numeric Objects .......................... 44
   7.3 Sequence Objects .......................... 48
   7.4 Mapping Objects .......................... 61
   7.5 Other Objects .......................... 63

8 Initialization, Finalization, and Threads .......................... 73
   8.1 Thread State and the Global Interpreter Lock .......................... 76
   8.2 Profiling and Tracing .......................... 80
   8.3 Advanced Debugger Support .......................... 81

9 Memory Management .......................... 83
   9.1 Overview .......................... 83
   9.2 Memory Interface .......................... 84
**Introduction**

The Application Programmer’s Interface to Python gives C and C++ programmers access to the Python interpreter at a variety of levels. The API is equally usable from C++, but for brevity it is generally referred to as the Python/C API. There are two fundamentally different reasons for using the Python/C API. The first reason is to write extension modules for specific purposes; these are C modules that extend the Python interpreter. This is probably the most common use. The second reason is to use Python as a component in a larger application; this technique is generally referred to as embedding Python in an application.

Writing an extension module is a relatively well-understood process, where a “cookbook” approach works well. There are several tools that automate the process to some extent. While people have embedded Python in other applications since its early existence, the process of embedding Python is less straightforward than writing an extension.

Many API functions are useful independent of whether you’re embedding or extending Python; moreover, most applications that embed Python will need to provide a custom extension as well, so it’s probably a good idea to become familiar with writing an extension before attempting to embed Python in a real application.

### 1.1 Include Files

All function, type and macro definitions needed to use the Python/C API are included in your code by the following line:

```c
#include "Python.h"
```

This implies inclusion of the following standard headers: `<stdio.h>`, `<string.h>`, `<errno.h>`, `<limits.h>`, and `<stdlib.h>` (if available).

**Warning:** Since Python may define some pre-processor definitions which affect the standard headers on some systems, you must include `Python.h` before any standard headers are included.

All user visible names defined by Python.h (except those defined by the included standard headers) have one of the prefixes ‘Py’ or ‘_Py’. Names beginning with ‘_Py’ are for internal use by the Python implementation and should not be used by extension writers. Structure member names do not have a reserved prefix.

**Important:** user code should never define names that begin with ‘Py’ or ‘_Py’. This confuses the reader, and jeopardizes the portability of the user code to future Python versions, which may define additional names beginning with one of these prefixes.

The header files are typically installed with Python. On UNIX, these are located in the directories `prefix/include/pythonversion/` and `exec_prefix/include/pythonversion/`, where prefix and exec_prefix are defined by the corresponding parameters to Python’s `configure` script and `version is sys.version[:3]`. On Windows, the headers are installed in `prefix/include`, where prefix is the installation directory specified to the installer.

To include the headers, place both directories (if different) on your compiler’s search path for includes. Do not place the parent directories on the search path and then use `#include <python2.4/Python.h>`; this will
break on multi-platform builds since the platform independent headers under prefix include the platform specific headers from exec_prefix.

C++ users should note that though the API is defined entirely using C, the header files do properly declare the entry points to be extern "C", so there is no need to do anything special to use the API from C++.

1.2 Objects, Types and Reference Counts

Most Python/C API functions have one or more arguments as well as a return value of type PyObject*. This type is a pointer to an opaque data type representing an arbitrary Python object. Since all Python object types are treated the same way by the Python language in most situations (e.g., assignments, scope rules, and argument passing), it is only fitting that they should be represented by a single C type. Almost all Python objects live on the heap: you never declare an automatic or static variable of type PyObject, only pointer variables of type PyObject* can be declared. The sole exception are the type objects; since these must never be deallocated, they are typically static PyTypeObject objects.

All Python objects (even Python integers) have a type and a reference count. An object’s type determines what kind of object it is (e.g., an integer, a list, or a user-defined function; there are many more as explained in the Python Reference Manual). For each of the well-known types there is a macro to check whether an object is of that type; for instance, ‘PyList_Check(a)’ is true if (and only if) the object pointed to by a is a Python list.

1.2.1 Reference Counts

The reference count is important because today’s computers have a finite (and often severely limited) memory size; it counts how many different places there are that have a reference to an object. Such a place could be another object, or a global (or static) C variable, or a local variable in some C function. When an object’s reference count becomes zero, the object is deallocated. If it contains references to other objects, their reference count is decremented. Those other objects may be deallocated in turn, if this decrement makes their reference count become zero, and so on. (There’s an obvious problem with objects that reference each other here; for now, the solution is “don’t do that.”)

Reference counts are always manipulated explicitly. The normal way is to use the macro Py_INCR() to increment an object’s reference count by one, and PyDECREF() to decrement it by one. The PyDECREF() macro is considerably more complex than the incref one, since it must check whether the reference count becomes zero and then cause the object’s deallocator to be called. The deallocator is a function pointer contained in the object’s type structure. The type-specific deallocator takes care of decrementing the reference counts for other objects contained in the object if this is a compound object type, such as a list, as well as performing any additional finalization that’s needed. There’s no chance that the reference count can overflow; at least as many bits are used to hold the reference count as there are distinct memory locations in virtual memory (assuming sizeof(long) >= sizeof(char*)). Thus, the reference count increment is a simple operation.

It is not necessary to increment an object’s reference count for every local variable that contains a pointer to an object. In theory, the object’s reference count goes up by one when the variable is made to point to it and it goes down by one when the variable goes out of scope. However, these two cancel each other out, so at the end the reference count hasn’t changed. The only real reason to use the reference count is to prevent the object from being deallocated as long as our variable is pointing to it. If we know that there is at least one other reference to the object that lives at least as long as our variable, there is no need to increment the reference count temporarily. An important situation where this arises is in objects that are passed as arguments to C functions in an extension module that are called from Python; the call mechanism guarantees to hold a reference to every argument for the duration of the call.

However, a common pitfall is to extract an object from a list and hold on to it for a while without incrementing its reference count. Some other operation might conceivably remove the object from the list, decrementing its reference count and possible deallocating it. The real danger is that innocent-looking operations may invoke arbitrary Python code which could do this; there is a code path which allows control to flow back to the user from a PyDECREF(), so almost any operation is potentially dangerous.

A safe approach is to always use the generic operations (functions whose name begins with 'PyObject_', 'PyNumber_', 'PySequence_' or 'PyMapping_'). These operations always increment the reference count
of the object they return. This leaves the caller with the responsibility to call Py_DECREF() when they are done with the result; this soon becomes second nature.

Reference Count Details

The reference count behavior of functions in the Python/C API is best explained in terms of ownership of references. Ownership pertains to references, never to objects (objects are not owned: they are always shared). "Owning a reference" means being responsible for calling Py_DECREF on it when the reference is no longer needed. Ownership can also be transferred, meaning that the code that receives ownership of the reference then becomes responsible for eventually decref'ing it by calling Py_DECREF() or Py_XDECREF() when it’s no longer needed—or passing on this responsibility (usually to its caller). When a function passes ownership of a reference on to its caller, the caller is said to receive a new reference. When no ownership is transferred, the caller is said to borrow the reference. Nothing needs to be done for a borrowed reference.

Conversely, when a calling function passes it a reference to an object, there are two possibilities: the function steals a reference to the object, or it does not. Stealing a reference means that when you pass a reference to a function, that function assumes that it now owns that reference, and you are not responsible for it any longer.

Few functions steal references; the two notable exceptions are PyList_SetItem() and PyTuple_SetItem(), which steal a reference to the item (but not to the tuple or list into which the item is put!). These functions were designed to steal a reference because of a common idiom for populating a tuple or list with newly created objects; for example, the code to create the tuple (1, 2, "three") could look like this (forgetting about error handling for the moment; a better way to code this is shown below):

```c
PyObject *t;

    t = PyTuple_New(3);
    PyTuple_SetItem(t, 0, PyInt_FromLong(1L));
    PyTuple_SetItem(t, 1, PyInt_FromLong(2L));
    PyTuple_SetItem(t, 2, PyString_FromString("three"));
```

Here, PyInt_FromLong() returns a new reference which is immediately stolen by PyTuple_SetItem(). When you want to keep using an object although the reference to it will be stolen, use Py_INCREF() to grab another reference before calling the reference-stealing function.

Incidentally, PyTuple_SetItem() is the only way to set tuple items; PySequence_SetItem() and PyObject_SetItem() refuse to do this since tuples are an immutable data type. You should only use PyTuple_SetItem() for tuples that you are creating yourself.

Equivalent code for populating a list can be written using PyList_New() and PyList_SetItem(). Such code can also use PySequence_SetItem(); this illustrates the difference between the two (the extra Py_DECREF() calls):

```c
PyObject *l, *x;

    l = PyList_New(3);
    PySequence_SetItem(l, 0, x); Py_DECREF(x);
    x = PyInt_FromLong(2L);
    PySequence_SetItem(l, 1, x); Py_DECREF(x);
    x = PyString_FromString("three");
    PySequence_SetItem(l, 2, x); Py_DECREF(x);
```

You might find it strange that the "recommended" approach takes more code. However, in practice, you will rarely use these ways of creating and populating a tuple or list. There's a generic function, Py_BuildValue(), that can create most common objects from C values, directed by a format string. For example, the above two blocks of code could be replaced by the following (which also takes care of the error checking):

1.2. Objects, Types and Reference Counts
It is much more common to use `PyObject_SetItem()` and friends with items whose references you are only borrowing, like arguments that were passed in to the function you are writing. In that case, their behaviour regarding reference counts is much saner, since you don’t have to increment a reference count so you can give a reference away ("have it be stolen"). For example, this function sets all items of a list (actually, any mutable sequence) to a given item:

```c
int
set_all(PyObject *target, PyObject *item)
{
    int i, n;

    n = PyObject_Length(target);
    if (n < 0)
        return -1;
    for (i = 0; i < n; i++) {
        if (PyObject_SetItem(target, i, item) < 0)
            return -1;
    }
    return 0;
}
```

The situation is slightly different for function return values. While passing a reference to most functions does not change your ownership responsibilities for that reference, many functions that return a reference to an object give you ownership of the reference. The reason is simple: in many cases, the returned object is created on the fly, and the reference you get is the only reference to the object. Therefore, the generic functions that return object references, like `PyObject_GetItem()` and `PySequence_GetItem()`, always return a new reference (the caller becomes the owner of the reference).

It is important to realize that whether you own a reference returned by a function depends on which function you call only — the plumage (the type of the object passed as an argument to the function) doesn’t enter into it! Thus, if you extract an item from a list using `PyList_GetItem()`, you don’t own the reference — but if you obtain the same item from the same list using `PySequence_GetItem()` (which happens to take exactly the same arguments), you do own a reference to the returned object.

Here is an example of how you could write a function that computes the sum of the items in a list of integers; once using `PyList_GetItem()`, and once using `PySequence_GetItem()`.
long
sum_list(PyObject *list)
{
  int i, n;
  long total = 0;
  PyObject *item;

  n = PyList_Size(list);
  if (n < 0)
    return -1; /* Not a list */
  for (i = 0; i < n; i++) {
    item = PyList_GetItem(list, i); /* Can’t fail */
    if (!PyInt_Check(item)) continue; /* Skip non-integers */
    total += PyInt_AsLong(item);
  }
  return total;
}

long
sum_sequence(PyObject *sequence)
{
  int i, n;
  long total = 0;
  PyObject *item;
  n = PySequence_Length(sequence);
  if (n < 0)
    return -1; /* Has no length */
  for (i = 0; i < n; i++) {
    item = PySequence_GetItem(sequence, i);
    if (item == NULL)
      return -1; /* Not a sequence, or other failure */
    if (PyInt_Check(item))
      total += PyInt_AsLong(item);
    Py_DECREF(item); /* Discard reference ownership */
  }
  return total;
}

1.2.2 Types

There are few other data types that play a significant role in the Python/C API; most are simple C types such as int, long, double and char*. A few structure types are used to describe static tables used to list the functions exported by a module or the data attributes of a new object type, and another is used to describe the value of a complex number. These will be discussed together with the functions that use them.

1.3 Exceptions

The Python programmer only needs to deal with exceptions if specific error handling is required; unhandled exceptions are automatically propagated to the caller, then to the caller’s caller, and so on, until they reach the top-level interpreter, where they are reported to the user accompanied by a stack traceback.

For C programmers, however, error checking always has to be explicit. All functions in the Python/C API can raise exceptions, unless an explicit claim is made otherwise in a function’s documentation. In general, when a function encounters an error, it sets an exception, discards any object references that it owns, and returns an error.
indicator — usually NULL or −1. A few functions return a Boolean true/false result, with false indicating an error. Very few functions return no explicit error indicator or have an ambiguous return value, and require explicit testing for errors with PyErr_Occurred().

Exception state is maintained in per-thread storage (this is equivalent to using global storage in an unthreaded application). A thread can be in one of two states: an exception has occurred, or not. The function PyErr_Occurred() can be used to check for this: it returns a borrowed reference to the exception type object when an exception has occurred, and NULL otherwise. There are a number of functions to set the exception state: PyErr_SetString() is the most common (though not the most general) function to set the exception state, and PyErr_Clear() clears the exception state.

The full exception state consists of three objects (all of which can be NULL): the exception type, the corresponding exception value, and the traceback. These have the same meanings as the Python objects sys.exc_type, sys.exc_value, and sys.exc_traceback; however, they are not the same: the Python objects represent the last exception being handled by a Python try ... except statement, while the C level exception state only exists while an exception is being passed on between C functions until it reaches the Python bytecode interpreter’s main loop, which takes care of transferring it to sys.exc_type and friends.

Note that starting with Python 1.5, the preferred, thread-safe way to access the exception state from Python code is to call the function sys.exc_info(), which returns the per-thread exception state for Python code. Also, the semantics of both ways to access the exception state have changed so that a function which catches an exception will save and restore its thread’s exception state so as to preserve the exception state of its caller. This prevents common bugs in exception handling code caused by an innocent-looking function overwriting the exception being handled; it also reduces the often unwanted lifetime extension for objects that are referenced by the stack frames in the traceback.

As a general principle, a function that calls another function to perform some task should check whether the called function raised an exception, and if so, pass the exception state on to its caller. It should discard any object references that it owns, and return an error indicator, but it should not set another exception — that would overwrite the exception that was just raised, and lose important information about the exact cause of the error.

A simple example of detecting exceptions and passing them on is shown in the sum_sequence() example above. It so happens that that example doesn’t need to clean up any owned references when it detects an error. The following example function shows some error cleanup. First, to remind you why you like Python, we show the equivalent Python code:

```python
def incr_item(dict, key):
    try:
        item = dict[key]
    except KeyError:
        item = 0
    dict[key] = item + 1
```

Here is the corresponding C code, in all its glory:
int incr_item(PyObject *dict, PyObject *key) {
    /* Objects all initialized to NULL for Py_XDECREF */
    PyObject *item = NULL, *const_one = NULL, *incremented_item = NULL;
    int rv = -1; /* Return value initialized to -1 (failure) */

    item = PyObject_GetItem(dict, key);
    if (item == NULL) {
        /* Handle KeyError only: */
        if (!PyErr_ExceptionMatches(PyExc_KeyError))
            goto error;
        /* Clear the error and use zero: */
        PyErr_Clear();
        item = PyInt_FromLong(0L);
        if (item == NULL)
            goto error;
    }
    const_one = PyInt_FromLong(1L);
    if (const_one == NULL)
        goto error;

    incremented_item = PyNumber_Add(item, const_one);
    if (incremented_item == NULL)
        goto error;

    if (PyObject_SetItem(dict, key, incremented_item) < 0)
        goto error;
    rv = 0; /* Success */
    /* Continue with cleanup code */

error:
    /* Cleanup code, shared by success and failure path */

    /* Use Py_XDECREF() to ignore NULL references */
    Py_XDECREF(item);
    Py_XDECREF(const_one);
    Py_XDECREF(incremented_item);

    return rv; /* -1 for error, 0 for success */
}

This example represents an endorsed use of the goto statement in C! It illustrates the use of
PyErr_ExceptionMatches() and PyErr_Clear() to handle specific exceptions, and the use of
Py_XDECREF() to dispose of owned references that may be NULL (note the ‘X’ in the name; Py_DECREF()
would crash when confronted with a NULL reference). It is important that the variables used to hold owned references
are initialized to NULL for this to work; likewise, the proposed return value is initialized to -1 (failure) and
only set to success after the final call made is successful.

1.4 Embedding Python

The one important task that only embedders (as opposed to extension writers) of the Python interpreter have to
worry about is the initialization, and possibly the finalization, of the Python interpreter. Most functionality of the
interpreter can only be used after the interpreter has been initialized.

The basic initialization function is Py_Initialize(). This initializes the table of loaded modules, and creates
the fundamental modules __builtins__, __main__, sys, and exceptions. It also initializes the module
search path (sys.path).

`Py_Initialize()` does not set the “script argument list” (sys.argv). If this variable is needed by Python code that will be executed later, it must be set explicitly with a call to `PySys_SetArgv(argc, argv)` subsequent to the call to `Py_Initialize()`.

On most systems (in particular, on UNIX and Windows, although the details are slightly different), `Py_Initialize()` calculates the module search path based upon its best guess for the location of the standard Python interpreter executable, assuming that the Python library is found in a fixed location relative to the Python interpreter executable. In particular, it looks for a directory named ‘lib/python2.4’ relative to the parent directory where the executable named ‘python’ is found on the shell command search path (the environment variable PATH).

For instance, if the Python executable is found in ‘/usr/local/bin/python’, it will assume that the libraries are in ‘/usr/local/lib/python2.4’. (In fact, this particular path is also the “fallback” location, used when no executable file named ‘python’ is found along PATH.) The user can override this behavior by setting the environment variable PYTHONHOME, or insert additional directories in front of the standard path by setting PYTHONPATH.

The embedding application can steer the search by calling `Py_SetProgramName(file)` before calling `Py_Initialize()`. Note that PYTHONHOME still overrides this and PYTHONPATH is still inserted in front of the standard path. An application that requires total control has to provide its own implementation of `Py_GetPath()`, `Py_GetPrefix()`, `Py_GetExecPrefix()`, and `Py_GetProgramFullPath()` (all defined in ‘Modules/getpath.c’).

Sometimes, it is desirable to “uninitialize” Python. For instance, the application may want to start over (make another call to `Py_Initialize()` or the application is simply done with its use of Python and wants to free all memory allocated by Python. This can be accomplished by calling `Py_Finalize()`. The function `Py_IsInitialized()` returns true if Python is currently in the initialized state. More information about these functions is given in a later chapter.
The Very High Level Layer

The functions in this chapter will let you execute Python source code given in a file or a buffer, but they will not let you interact in a more detailed way with the interpreter.

Several of these functions accept a start symbol from the grammar as a parameter. The available start symbols are `Py_eval_input`, `Py_file_input`, and `Py_single_input`. These are described following the functions which accept them as parameters.

Note also that several of these functions take `FILE*` parameters. On particular issue which needs to be handled carefully is that the `FILE` structure for different C libraries can be different and incompatible. Under Windows (at least), it is possible for dynamically linked extensions to actually use different libraries, so care should be taken that `FILE*` parameters are only passed to these functions if it is certain that they were created by the same library that the Python runtime is using.

```c
int Py_Main(int argc, char **argv)
    The main program for the standard interpreter. This is made available for programs which embed Python. The `argc` and `argv` parameters should be prepared exactly as those which are passed to a C program's `main()` function. It is important to note that the argument list may be modified (but the contents of the strings pointed to by the argument list are not). The return value will be the integer passed to the `sys.exit()` function, 1 if the interpreter exits due to an exception, or 2 if the parameter list does not represent a valid Python command line.
```

```c
int PyRun<AnyFile>(FILE *fp, const char *filename)
    This is a simplified interface to `PyRun<AnyFileExFlags>()` below, leaving `closeit` set to 0 and `flags` set to NULL.
```

```c
int PyRun<AnyFileFlags>(FILE *fp, const char *filename, PyCompilerFlags *flags)
    This is a simplified interface to `PyRun<AnyFileExFlags>()` below, leaving the `closeit` argument set to 0.
```

```c
int PyRun<AnyFileEx>(FILE *fp, const char *filename, int closeit)
    This is a simplified interface to `PyRun<AnyFileExFlags>()` below, leaving the `flags` argument set to NULL.
```

```c
int PyRun<AnyFileExFlags>(FILE *fp, const char *filename, int closeit, PyCompilerFlags *flags)
    If `fp` refers to a file associated with an interactive device (console or terminal input or UNIX pseudo-terminal), return the value of `PyRun_InteractiveLoop()`, otherwise return the result of `PyRun_SimpleFile()`. If `filename` is NULL, this function uses "???" as the filename.
```

```c
int PyRun_SimpleString(const char *command)
    This is a simplified interface to `PyRun_SimpleStringFlags()` below, leaving the `PyCompilerFlags` argument set to NULL.
```

```c
int PyRun_SimpleStringFlags(const char *command, PyCompilerFlags *flags)
    Executes the Python source code from `command` in the `__main__` module according to the `flags` argument. If `__main__` does not already exist, it is created. Returns 0 on success or -1 if an exception was raised. If there was an error, there is no way to get the exception information. For the meaning of `flags`, see below.
```

```c
int PyRun_SimpleFile(FILE *fp, const char *filename)
    This is a simplified interface to `PyRun_SimpleFileExFlags()` below, leaving `closeit` set to 0 and
int PyRun__SimpleFileFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)
This is a simplified interface to PyRun__SimpleFileExFlags() below, leaving closeit set to 0.

int PyRun__SimpleFileEx(FILE *fp, const char *filename, int closeit)
This is a simplified interface to PyRun__SimpleFileExFlags() below, leaving flags set to NULL.

int PyRun__SimpleFileExFlags(FILE *fp, const char *filename, int closeit, PyCompilerFlags *flags)
Similar to PyRun__SimpleStringFlags(), but the Python source code is read from fp instead of an in-memory string. filename should be the name of the file. If closeit is true, the file is closed before PyRun__SimpleFileExFlags returns.

int PyRun__InteractiveOne(FILE *fp, const char *filename)
This is a simplified interface to PyRun__InteractiveOneFlags() below, leaving flags set to NULL.

int PyRun__InteractiveOneFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)
Read and execute a single statement from a file associated with an interactive device according to the flags argument. If filename is NULL, "???" is used instead. The user will be prompted using sys.ps1 and sys.ps2. Returns 0 when the input was executed successfully, -1 if there was an exception, or an error code from the ‘errcode.h’ include file distributed as part of Python if there was a parse error. (Note that ‘errcode.h’ is not included by ‘Python.h’, so must be included specifically if needed.)

int PyRun__InteractiveLoop(FILE *fp, const char *filename)
This is a simplified interface to PyRun__InteractiveLoopFlags() below, leaving flags set to NULL.

int PyRun__InteractiveLoopFlags(FILE *fp, const char *filename, PyCompilerFlags *flags)
Read and execute statements from a file associated with an interactive device until EOF is reached. If filename is NULL, "???" is used instead. The user will be prompted using sys.ps1 and sys.ps2. Returns 0 at EOF.

struct _node* PyParser__SimpleParseString(const char *str, int start)
This is a simplified interface to PyParser__SimpleParseStringFlagsFilename() below, leaving filename set to NULL and flags set to 0.

struct _node* PyParser__SimpleParseStringFlags(const char *str, int start, int flags)
This is a simplified interface to PyParser__SimpleParseStringFlagsFilename() below, leaving filename set to NULL.

struct _node* PyParser__SimpleParseStringFlagsFilename(const char *str, const char *filename, int start, int flags)
Parse Python source code from str using the start token start according to the flags argument. The result can be used to create a code object which can be evaluated efficiently. This is useful if a code fragment must be evaluated many times.

struct _node* PyParser__SimpleParseFile(FILE *fp, const char *filename, int start)
This is a simplified interface to PyParser__SimpleParseFileFlags() below, leaving flags set to 0.

struct _node* PyParser__SimpleParseFileFlags(FILE *fp, const char *filename, int start, int flags)
Similar to PyParser__SimpleParseStringFlagsFilename(), but the Python source code is read from fp instead of an in-memory string.

PyObject* PyRun__String(const char *str, int start, PyObject *globals, PyObject *locals)
This is a simplified interface to PyRun__StringFlags() below, leaving flags set to NULL.

PyObject* PyRun__StringFlags(const char *str, int start, PyObject *globals, PyObject *locals, PyCompilerFlags *flags)
Execute Python source code from str in the context specified by the dictionaries globals and locals with the compiler flags specified by flags. The parameter start specifies the start token that should be used to parse the source code.
Returns the result of executing the code as a Python object, or NULL if an exception was raised.

PyObject* PyRun__File(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals)
This is a simplified interface to `PyRun_FileExFlags()` below, leaving `closeit` set to 0 and `flags` set to NULL.

```c
PyObject* PyRun_FileEx(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, int closeit)
```

This is a simplified interface to `PyRun_FileExFlags()` below, leaving `flags` set to NULL.

```c
PyObject* PyRun_FileFlags(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, PyCompilerFlags *flags)
```

This is a simplified interface to `PyRun_FileExFlags()` below, leaving `closeit` set to 0.

```c
PyObject* PyRun_FileExFlags(FILE *fp, const char *filename, int start, PyObject *globals, PyObject *locals, int closeit, PyCompilerFlags *flags)
```

Similar to `PyRun_StringFlags()`, but the Python source code is read from `fp` instead of an in-memory string. `filename` should be the name of the file. If `closeit` is true, the file is closed before `PyRun_FileExFlags()` returns.

```c
PyObject* PyCompileString(const char *str, const char *filename, int start)
```


This is a simplified interface to `PyCompileStringFlags()` below, leaving `flags` set to NULL.

```c
PyObject* PyCompileStringFlags(const char *str, const char *filename, int start, PyCompilerFlags *flags)
```

Parse and compile the Python source code in `str`, returning the resulting code object. The start token is given by `start`; this can be used to constrain the code which can be compiled and should be `Py_eval_input`, `Py_file_input`, or `Py_single_input`. The filename specified by `filename` is used to construct the code object and may appear in tracebacks or `SyntaxError` exception messages. This returns NULL if the code cannot be parsed or compiled.

```c
int Py_eval_input
```

The start symbol from the Python grammar for isolated expressions; for use with `PyCompileString()`.

```c
int Py_file_input
```

The start symbol from the Python grammar for sequences of statements as read from a file or other source; for use with `PyCompileString()`. This is the symbol to use when compiling arbitrarily long Python source code.

```c
int Py_single_input
```

The start symbol from the Python grammar for a single statement; for use with `PyCompileString()`. This is the symbol used for the interactive interpreter loop.

```c
struct PyCompilerFlags
```

This is the structure used to hold compiler flags. In cases where code is only being compiled, it is passed as `int flags`, and in cases where code is being executed, it is passed as `PyCompilerFlags *flags`. In this case, from `__future__ import` can modify `flags`.

Whenever `PyCompilerFlags *flags` is NULL, `cf_flags` is treated as equal to 0, and any modification due to from `__future__ import` is discarded.

```c
struct PyCompilerFlags {
    int cf_flags;
}
```

```c
int CO_FUTURE_DIVISION
```

This bit can be set in `flags` to cause division operator `/` to be interpreted as “true division” according to PEP 238.
Reference Counting

The macros in this section are used for managing reference counts of Python objects.

void **Py INCREF** (**PyObject** *o*)
Increment the reference count for object *o*. The object must not be NULL; if you aren’t sure that it isn’t NULL, use **Py XINCREF** ().

void **Py XINCREF** (**PyObject** *o*)
Increment the reference count for object *o*. The object may be NULL, in which case the macro has no effect.

void **Py DECREF** (**PyObject** *o*)
Decrement the reference count for object *o*. The object must not be NULL; if you aren’t sure that it isn’t NULL, use **Py XDECREF** (). If the reference count reaches zero, the object’s type’s deallocation function (which must not be NULL) is invoked.

**Warning:** The deallocation function can cause arbitrary Python code to be invoked (e.g. when a class instance with a **del** method is deallocated). While exceptions in such code are not propagated, the executed code has free access to all Python global variables. This means that any object that is reachable from a global variable should be in a consistent state before **Py DECREF** () is invoked. For example, code to delete an object from a list should copy a reference to the deleted object in a temporary variable, update the list data structure, and then call **Py DECREF** () for the temporary variable.

void **Py XDECREF** (**PyObject** *o*)
Decrement the reference count for object *o*. The object may be NULL, in which case the macro has no effect; otherwise the effect is the same as for **Py DECREF** (), and the same warning applies.

void **Py CLEAR** (**PyObject** *o*)
Decrement the reference count for object *o*. The object may be NULL, in which case the macro has no effect; otherwise the effect is the same as for **Py DECREF** (), except that the argument is also set to NULL. The warning for **Py DECREF** () does not apply with respect to the object passed because the macro carefully uses a temporary variable and sets the argument to NULL before decrementing its reference count.

It is a good idea to use this macro whenever decrementing the value of a variable that might be traversed during garbage collection.

New in version 2.4.

The following functions are for runtime dynamic embedding of Python: **Py IncRef**(PyObject *o*), **Py DecRef**(PyObject *o*). They are simply exported function versions of **Py XINCREF** () and **Py XDECREF** (), respectively.

The following functions or macros are only for use within the interpreter core: **_Py Dealloc** (), **_Py ForgetReference** (), **_Py NewReference** (), as well as the global variable **Py RefTotal**.
The functions described in this chapter will let you handle and raise Python exceptions. It is important to understand some of the basics of Python exception handling. It works somewhat like the UNIX errno variable: there is a global indicator (per thread) of the last error that occurred. Most functions don’t clear this on success, but will set it to indicate the cause of the error on failure. Most functions also return an error indicator, usually NULL if they are supposed to return a pointer, or -1 if they return an integer (exception: the PyArg_*( ) functions return 1 for success and 0 for failure).

When a function must fail because some function it called failed, it generally doesn’t set the error indicator; the function it called already set it. It is responsible for either handling the error and clearing the exception or returning after cleaning up any resources it holds (such as object references or memory allocations); it should not continue normally if it is not prepared to handle the error. If returning due to an error, it is important to indicate to the caller that an error has been set. If the error is not handled or carefully propagated, additional calls into the Python/C API may not behave as intended and may fail in mysterious ways.

The error indicator consists of three Python objects corresponding to the Python variables sys.exc_type, sys.exc_value and sys.exc_traceback. API functions exist to interact with the error indicator in various ways. There is a separate error indicator for each thread.

- **void PyErr_Print()**
  - Print a standard traceback to sys.stderr and clear the error indicator. Call this function only when the error indicator is set. (Otherwise it will cause a fatal error!)

- **PyObject* PyErr_Occurred()**
  - Test whether the error indicator is set. If set, return the exception type (the first argument to the last call to one of the PyErr_Set*( ) functions or to PyErr_Restore( )). If not set, return NULL. You do not own a reference to the return value, so you do not need to Py_DECREF ( ) it. **Note:** Do not compare the return value to a specific exception; use PyErr_ExceptionMatches ( ) instead, shown below. (The comparison could easily fail since the exception may be an instance instead of a class, in the case of a class exception, or it may be a subclass of the expected exception.)

- **int PyErr_ExceptionMatches (PyObject *exc)**
  - Equivalent to ‘PyErr_GivenExceptionMatches (PyErr_Occurred (), exc)’. This should only be called when an exception is actually set; a memory access violation will occur if no exception has been raised.

- **int PyErr_GivenExceptionMatches (PyObject *given, PyObject *exc)**
  - Return true if the given exception matches the exception in exc. If exc is a class object, this also returns true when given is an instance of a subclass. If exc is a tuple, all exceptions in the tuple (and recursively in subtuples) are searched for a match. If given is NULL, a memory access violation will occur.

- **void PyErr_NormalizeException (PyObject** **exc, PyObject** **val, PyObject** **tb)**
  - Under certain circumstances, the values returned by PyErr_Fetch ( ) below can be “unnormalized”, meaning that *exc is a class object but *val is not an instance of the same class. This function can be used to instantiate the class in that case. If the values are already normalized, nothing happens. The delayed normalization is implemented to improve performance.

- **void PyErr_Clear()**
Clear the error indicator. If the error indicator is not set, there is no effect.

```c
void PyErr_Fetch(PyObject **ptype, PyObject **pvalue, PyObject **ptraceback)
Retrieve the error indicator into three variables whose addresses are passed. If the error indicator is not set, set all three variables to NULL. If it is set, it will be cleared and you own a reference to each object retrieved. The value and traceback object may be NULL even when the type object is not. Note: This function is normally only used by code that needs to handle exceptions or by code that needs to save and restore the error indicator temporarily.
```

```c
void PyErr_Restore(PyObject *type, PyObject *value, PyObject *traceback)
Set the error indicator from the three objects. If the error indicator is already set, it is cleared first. If the objects are NULL, the error indicator is cleared. Do not pass a NULL type and non-NUL value or traceback. The exception type should be a class. Do not pass an invalid exception type or value. (Violating these rules will cause subtle problems later.) This call takes away a reference to each object before the call and after the call you no longer own these references. (If you don’t understand this, don’t use this function. I warned you.) Note: This function is normally only used by code that needs to save and restore the error indicator temporarily; use PyErr_Fetch() to save the current exception state.
```

```c
void PyErr_SetString(PyObject *type, char *message)
This is the most common way to set the error indicator. The first argument specifies the exception type; it is normally one of the standard exceptions, e.g. PyExc_RuntimeError. You need not increment its reference count. The second argument is an error message; it is converted to a string object.
```

```c
void PyErr_SetObject(PyObject *type, PyObject *value)
This function is similar to PyErr_SetString() but lets you specify an arbitrary Python object for the "value" of the exception.
```

```c
PyObject* PyErr_Format(PyObject *exception, const char *format, ...)
Return value: Always NULL.
This function sets the error indicator and returns NULL exception should be a Python exception (class, not an instance). format should be a string, containing format codes, similar to printf(). The width.precision before a format code is parsed, but the width part is ignored.
```

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>'c'</td>
<td>Character, as an int parameter</td>
</tr>
<tr>
<td>'d'</td>
<td>Number in decimal, as an int parameter</td>
</tr>
<tr>
<td>'x'</td>
<td>Number in hexadecimal, as an int parameter</td>
</tr>
<tr>
<td>'s'</td>
<td>A string, as a char * parameter</td>
</tr>
<tr>
<td>'p'</td>
<td>A hex pointer, as a void * parameter</td>
</tr>
</tbody>
</table>

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

```c
void PyErr_SetNone(PyObject *type)
This is a shorthand for PyErr_SetObject(type, Py_None).
```

```c
int PyErr_BadArgument()
This is a shorthand for PyErr_SetString(PyExc_TypeError, message), where message indicates that a built-in operation was invoked with an illegal argument. It is mostly for internal use.
```

```c
PyObject* PyErr_NoMemory()
Return value: Always NULL.
This is a shorthand for PyErr_SetNone(PyExc_MemoryError); it returns NULL so an object allocation function can write 'return PyErr_NoMemory();' when it runs out of memory.
```

```c
PyObject* PyErr_SetFromErrno(PyObject *type)
Return value: Always NULL.
This is a convenience function to raise an exception when a C library function has returned an error and set the C variable errno. It constructs a tuple object whose first item is the integer errno value and whose second item is the corresponding error message (gotten from strerror()), and then calls PyErr_SetObject(type, object). On UNIX, when the errno value is EINTR, indicating an interrupted system call, this calls PyErr_CheckSignals(), and if that set the error indicator, leaves it set to that. The function always returns NULL, so a wrapper function around a system call can write 'return PyErr_SetFromErrno(type);' when the system call returns an error.
```

Chapter 4. Exception Handling
PyObject* PyErr_SetFromErrnoWithFilename (PyObject *type, char *filename)

Return value: Always NULL.

Similar to PyErr_SetFromErrno(), with the additional behavior that if filename is not NULL, it is passed to the constructor of type as a third parameter. In the case of exceptions such as IOError and OSError, this is used to define the filename attribute of the exception instance.

PyObject* PyErr_SetFromWindowsErr (int ierr)

Return value: Always NULL.

This is a convenience function to raise WindowsError. If called with ierr of 0, the error code returned by a call to GetLastError() is used instead. It calls the Win32 function FormatMessage() to retrieve the Windows description of error code given by ierr or GetLastError(), then it constructs a tuple object whose first item is the ierr value and whose second item is the corresponding error message (got from FormatMessage()), and then calls 'PyErr_SetObject (PyExc_WindowsError, object).' This function always returns NULL. Availability: Windows.

PyObject* PyErr_SetExcFromWindowsErr (PyObject *type, int ierr)

Similar to PyErr_SetFromWindowsErr(), with an additional parameter specifying the exception type to be raised. Availability: Windows. New in version 2.3.

PyObject* PyErr_SetFromWindowsErrWithFilename (int ierr, char *filename)

Return value: Always NULL.

Similar to PyErr_SetFromWindowsErr(), with the additional behavior that if filename is not NULL, it is passed to the constructor of WindowsError as a third parameter. Availability: Windows.

PyObject* PyErr_SetExcFromWindowsErrWithFilename (PyObject *type, int ierr, char *filename)

Similar to PyErr_SetFromWindowsErrWithFilename(), with an additional parameter specifying the exception type to be raised. Availability: Windows. New in version 2.3.

void PyErr_BadInternalCall ()

This is a shorthand for 'PyErr_SetString(PyExc_TypeError, message)', where message indicates that an internal operation (e.g. a Python/C API function) was invoked with an illegal argument. It is mostly for internal use.

int PyErr_Warn (PyObject *category, char *message)

Issue a warning message. The category argument is a warning category (see below) or NULL; the message argument is a message string.

This function normally prints a warning message to sys.stderr; however, it is also possible that the user has specified that warnings are to be turned into errors, and in that case this will raise an exception. It is also possible that the function raises an exception because of a problem with the warning machinery (the implementation imports the warnings module to do the heavy lifting). The return value is 0 if no exception is raised, or -1 if an exception is raised. (It is not possible to determine whether a warning message is actually printed, nor what the reason is for the exception; this is intentional.) If an exception is raised, the caller should do its normal exception handling (for example, PyErr_DECREASE() owned references and return an error value).

Warning categories must be subclasses of Warning; the default warning category is RuntimeWarning.

The standard Python warning categories are available as global variables whose names are 'PyExc_-' followed by the Python exception name. These have the type PyObject*; they are all class objects. Their names are PyExc_Warning, PyExc_UserWarning, PyExc_DeprecationWarning, PyExc_SyntaxWarning, PyExc_RuntimeWarning, and PyExc_FutureWarning. PyExc_Warning is a subclass of PyExc_Exception; the other warning categories are subclasses of PyExc_Warning.

For information about warning control, see the documentation for the warnings module and the -W option in the command line documentation. There is no C API for warning control.

int PyErr_WarnExplicit (PyObject *category, char *message, char *filename, int lineno, char *module, PyObject *registry)

Issue a warning message with explicit control over all warning attributes. This is a straightforward wrapper around the Python function warnings.warn_explicit(), see there for more information. The module and registry arguments may be set to NULL to get the default effect described there.

int PyErr_CheckSignals ()
This function interacts with Python’s signal handling. It checks whether a signal has been sent to the processes and if so, invokes the corresponding signal handler. If the signal module is supported, this can invoke a signal handler written in Python. In all cases, the default effect for SIGINT is to raise the KeyboardInterrupt exception. If an exception is raised the error indicator is set and the function returns 1; otherwise the function returns 0. The error indicator may or may not be cleared if it was previously set.

void PyErr_SetInterrupt()
This function simulates the effect of a SIGINT signal arriving — the next time PyErr_CheckSignals() is called, KeyboardInterrupt will be raised. It may be called without holding the interpreter lock.

PyObject* PyErr_NewException(char *name, PyObject *base, PyObject *dict)
This utility function creates and returns a new exception object. The name argument must be the name of the new exception, a C string of the form module.class. The base and dict arguments are normally NULL. This creates a class object derived from the root for all exceptions, the built-in name Exception (accessible in C as PyExc_Exception). The __module__ attribute of the new class is set to the first part (up to the last dot) of the name argument, and the class name is set to the last part (after the last dot). The base argument can be used to specify an alternate base class. The dict argument can be used to specify a dictionary of class variables and methods.

void PyErr_WriteUnraisable(PyObject *obj)
This utility function prints a warning message to sys.stderr when an exception has been set but it is impossible for the interpreter to actually raise the exception. It is used, for example, when an exception occurs in an __del__() method.

The function is called with a single argument obj that identifies the context in which the unraisable exception occurred. The repr of obj will be printed in the warning message.

4.1 Standard Exceptions

All standard Python exceptions are available as global variables whose names are ‘PyExc_’ followed by the Python exception name. These have the type PyObject*; they are all class objects. For completeness, here are all the variables:
<table>
<thead>
<tr>
<th>C Name</th>
<th>Python Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyExc_Exception</td>
<td>Exception</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_StandardError</td>
<td>StandardError</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_ArithmeticError</td>
<td>ArithmeticError</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_LookupError</td>
<td>LookupError</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_AssertionError</td>
<td>AssertionError</td>
<td></td>
</tr>
<tr>
<td>PyExc_AttributeError</td>
<td>AttributeError</td>
<td></td>
</tr>
<tr>
<td>PyExc_EOFError</td>
<td>EOFError</td>
<td></td>
</tr>
<tr>
<td>PyExc_EnvironmentError</td>
<td>EnvironmentError</td>
<td>(1)</td>
</tr>
<tr>
<td>PyExc_FloatingPointError</td>
<td>FloatingPointError</td>
<td></td>
</tr>
<tr>
<td>PyExc_IOError</td>
<td>IOError</td>
<td></td>
</tr>
<tr>
<td>PyExc_ImportError</td>
<td>ImportError</td>
<td></td>
</tr>
<tr>
<td>PyExc_IndexError</td>
<td>IndexError</td>
<td></td>
</tr>
<tr>
<td>PyExc_KeyError</td>
<td>KeyError</td>
<td></td>
</tr>
<tr>
<td>PyExc_KeyboardInterrupt</td>
<td>KeyboardInterrupt</td>
<td></td>
</tr>
<tr>
<td>PyExc_MemoryError</td>
<td>MemoryError</td>
<td></td>
</tr>
<tr>
<td>PyExc_NameError</td>
<td>NameError</td>
<td></td>
</tr>
<tr>
<td>PyExc_NotImplementedError</td>
<td>NotImplementedError</td>
<td></td>
</tr>
<tr>
<td>PyExc_OSError</td>
<td>OSError</td>
<td></td>
</tr>
<tr>
<td>PyExc_OverflowError</td>
<td>OverflowError</td>
<td></td>
</tr>
<tr>
<td>PyExc_ReferenceError</td>
<td>ReferenceError</td>
<td>(2)</td>
</tr>
<tr>
<td>PyExc_RuntimeError</td>
<td>RuntimeError</td>
<td></td>
</tr>
<tr>
<td>PyExc_SyntaxError</td>
<td>SyntaxError</td>
<td></td>
</tr>
<tr>
<td>PyExc_SystemError</td>
<td>SystemError</td>
<td></td>
</tr>
<tr>
<td>PyExc_SystemExit</td>
<td>SystemExit</td>
<td></td>
</tr>
<tr>
<td>PyExc_TypeError</td>
<td>TypeError</td>
<td></td>
</tr>
<tr>
<td>PyExc_ValueError</td>
<td>ValueError</td>
<td></td>
</tr>
<tr>
<td>PyExc_WindowsError</td>
<td>WindowsError</td>
<td>(3)</td>
</tr>
<tr>
<td>PyExc_ZeroDivisionError</td>
<td>ZeroDivisionError</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

(1) This is a base class for other standard exceptions.

(2) This is the same as `weakref.ReferenceError`.

(3) Only defined on Windows; protect code that uses this by testing that the preprocessor macro `MS_WINDOWS` is defined.

4.2 Deprecation of String Exceptions

All exceptions built into Python or provided in the standard library are derived from `Exception`.

String exceptions are still supported in the interpreter to allow existing code to run unmodified, but this will also change in a future release.
The functions in this chapter perform various utility tasks, ranging from helping C code be more portable across platforms, using Python modules from C, and parsing function arguments and constructing Python values from C values.

5.1 Operating System Utilities

```c
int Py_FdIsInteractive(FILE *fp, char *filename)
Return true (nonzero) if the standard I/O file fp with name filename is deemed interactive. This is the case
for files for which ‘isatty(fileno(fp))’ is true. If the global flag Py_InteractiveFlag is true, this function also returns true if the filename pointer is NULL or if the name is equal to one of the strings ‘<stdin>’ or ‘???’.
```

```c
long PyOS_GetLastModificationTime(char *filename)
Return the time of last modification of the file filename. The result is encoded in the same way as the
timestamp returned by the standard C library function time().
```

```c
void PyOS_AfterFork()
Function to update some internal state after a process fork; this should be called in the new process if the
Python interpreter will continue to be used. If a new executable is loaded into the new process, this function
does not need to be called.
```

```c
int PyOS_CheckStack()
Return true when the interpreter runs out of stack space. This is a reliable check, but is only available
when USE_STACKCHECK is defined (currently on Windows using the Microsoft Visual C++ compiler).
USE_STACKCHECK will be defined automatically; you should never change the definition in your own
code.
```

```c
PyOS_sighandler_t PyOS_getsig(int i)
Return the current signal handler for signal i. This is a thin wrapper around either sigaction() or
signal(). Do not call those functions directly! PyOS_sighandler_t is a typedef alias for void (*)(int).
```

```c
PyOS_sighandler_t PyOS_setsig(int i, PyOS_sighandler_t h)
Set the signal handler for signal i to be h; return the old signal handler. This is a thin wrapper around either
sigaction() or signal(). Do not call those functions directly! PyOS_sighandler_t is a typedef
alias for void (*)(int).
```

5.2 Process Control

```c
void Py_FatalError(const char *message)
Print a fatal error message and kill the process. No cleanup is performed. This function should only be
invoked when a condition is detected that would make it dangerous to continue using the Python interpreter;
e.g., when the object administration appears to be corrupted. On UNIX, the standard C library function
abort() is called which will attempt to produce a ‘core’ file.
void Py_Exit(int status)
    Exit the current process. This calls Py_Finalize() and then calls the standard C library function exit(status).

int Py_AtExit(void (*func)(void))
    Register a cleanup function to be called by Py_Finalize(). The cleanup function will be called with no arguments and should return no value. At most 32 cleanup functions can be registered. When the registration is successful, Py_AtExit() returns 0; on failure, it returns -1. The cleanup function registered last is called first. Each cleanup function will be called at most once. Since Python’s internal finalization will have completed before the cleanup function, no Python APIs should be called by func.

5.3 Importing Modules

PyObject* PyImport_ImportModule(char *name)
    This is a simplified interface to PyImport_ImportModuleEx() below, leaving the globals and locals arguments set to NULL. When the name argument contains a dot (when it specifies a submodule of a package), the fromlist argument is set to the list ['**'] so that the return value is the named module rather than the top-level package containing it as would otherwise be the case. (Unfortunately, this has an additional side effect when name in fact specifies a subpackage instead of a submodule: the submodules specified in the package’s __all__ variable are loaded.) Return a new reference to the imported module, or NULL with an exception set on failure (before Python 2.4, the module may still be created in this case). Like for import(), the return value when a submodule of a package was requested is normally the top-level package, unless a non-empty fromlist was given. Changed in version 2.4: failing imports remove incomplete module objects.

PyObject* PyImport_ImportModuleEx(char *name, PyObject *globals, PyObject *locals, PyObject *fromlist)
    Import a module. This is best described by referring to the built-in Python function __import__() as the standard __import__() function calls this function directly.

    The return value is a new reference to the imported module or top-level package, or NULL with an exception set on failure (before Python 2.4, the module may still be created in this case). Like for __import__(), the return value when a submodule of a package was requested is normally the top-level package, unless a non-empty fromlist was given. Changed in version 2.4: failing imports remove incomplete module objects.

PyObject* PyImport_Import(PyObject *name)
    This is a higher-level interface that calls the current “import hook function”. It invokes the __import__() function from the __builtins__ of the current globals. This means that the import is done using whatever import hooks are installed in the current environment, e.g. by rexec or ihooks.

PyObject* PyImport_ReloadModule(PyObject *m)
    Reload a module. This is best described by referring to the built-in Python function reload(), as the standard reload() function calls this function directly. Return a new reference to the reloaded module, or NULL with an exception set on failure (the module still exists in this case).

PyObject* PyImport_AddModule(char *name)
    Return value: Borrowed reference.
    Return the module object corresponding to a module name. The name argument may be of the form package.module. First check the modules dictionary if there’s one there, and if not, create a new one and insert it in the modules dictionary. Return NULL with an exception set on failure. Note: This function does not load or import the module; if the module wasn’t already loaded, you will get an empty module object. Use PyImport_ImportModule() or one of its variants to import a module. Package structures implied by a dotted name for name are not created if not already present.

PyObject* PyImport_ExecCodeModule(char *name, PyObject *co)
    Given a module name (possibly of the form package.module) and a code object read from a Python
bytecode file or obtained from the built-in function \texttt{compile()}, load the module. Return a new reference to the module object, or NULL with an exception set if an error occurred. Before Python 2.4, the module could still be created in error cases. Starting with Python 2.4, \texttt{name} is removed from \texttt{sys.modules} in error cases, and even if \texttt{name} was already in \texttt{sys.modules} on entry to \texttt{PyImport_ExecCodeModule()}. Leaving incompletely initialized modules in \texttt{sys.modules} is dangerous, as imports of such modules have no way to know that the module object is an unknown (and probably damaged with respect to the module author’s intents) state.

This function will reload the module if it was already imported. See \texttt{PyImport_ReloadModule()} for the intended way to reload a module.

If \texttt{name} points to a dotted name of the form \texttt{package.module}, any package structures not already created will still not be created.

Changed in version 2.4: \texttt{name} is removed from \texttt{sys.modules} in error cases.

\begin{verbatim}
long PyImport_GetMagicNumber()
Return the magic number for Python bytecode files (a.k.a. '.pyc' and '.pyo' files). The magic number should be present in the first four bytes of the bytecode file, in little-endian byte order.

PyObject* PyImport_GetModuleDict()
Return value: Borrowed reference.
Return the dictionary used for the module administration (a.k.a. \texttt{sys.modules}). Note that this is a per-interpreter variable.

void _PyImport_Init()
Initialize the import mechanism. For internal use only.

void PyImport_Cleanup()
Empty the module table. For internal use only.

void _PyImport_Fini()
Finalize the import mechanism. For internal use only.

PyObject* _PyImport_FindExtension(char *, char *)
For internal use only.

PyObject* _PyImport_FixupExtension(char *, char *)
For internal use only.

int PyImport_ImportFrozenModule(char *name)
Load a frozen module named \texttt{name}. Return 1 for success, 0 if the module is not found, and -1 with an exception set if the initialization failed. To access the imported module on a successful load, use \texttt{PyImport_ImportModule()}. (Note the misnomer — this function would reload the module if it was already imported.)

struct _frozen
This is the structure type definition for frozen module descriptors, as generated by the \texttt{freeze} utility (see ‘Tools/freeze’ in the Python source distribution). Its definition, found in ‘\texttt{include/import.h}’, is:

\begin{verbatim}
struct _frozen {
    char *name;
    unsigned char *code;
    int size;
};
\end{verbatim}

struct _frozen* PyImport_FrozenModules
This pointer is initialized to point to an array of \texttt{struct _frozen} records, terminated by one whose members are all NULL or zero. When a frozen module is imported, it is searched in this table. Third-party code could play tricks with this to provide a dynamically created collection of frozen modules.

int PyImport_AppendInittab(char *name, void (*initfunc)(void))
Add a single module to the existing table of built-in modules. This is a convenience wrapper around \texttt{PyImport_ExtendInittab()}, returning -1 if the table could not be extended. The new module can be imported by the name \texttt{name}, and uses the function \texttt{initfunc} as the initialization function called on the first attempted import. This should be called before \texttt{Py_Initialize()}.
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}

5.3. Importing Modules 23
struct _inittab
Structure describing a single entry in the list of built-in modules. Each of these structures gives the name and initialization function for a module built into the interpreter. Programs which embed Python may use an array of these structures in conjunction with PyImport_ExtendInittab() to provide additional built-in modules. The structure is defined in 'Include/import.h' as:

```c
struct _inittab {
    char *name;
    void (*initfunc)(void);
};
```

int PyImport_ExtendInittab(struct _inittab *newtab)
Add a collection of modules to the table of built-in modules. The newtab array must end with a sentinel entry which contains NULL for the name field; failure to provide the sentinel value can result in a memory fault. Returns 0 on success or -1 if insufficient memory could be allocated to extend the internal table. In the event of failure, no modules are added to the internal table. This should be called before Py_Initialize().

5.4 Data marshalling support

These routines allow C code to work with serialized objects using the same data format as the marshal module. There are functions to write data into the serialization format, and additional functions that can be used to read the data back. Files used to store marshalled data must be opened in binary mode.

Numeric values are stored with the least significant byte first.

The module supports two versions of the data format: version 0 is the historical version, version 1 (new in Python 2.4) shares interned strings in the file, and upon unmarshalling, Py_MARSHAL_VERSION indicates the current file format (currently 1).

void PyMarshal_WriteLongToFile(long value, FILE *file, int version)
Marshal a long integer, value, to file. This will only write the least-significant 32 bits of value; regardless of the size of the native long type.

Changed in version 2.4: version indicates the file format.

void PyMarshal_WriteObjectToFile(PyObject *value, FILE *file, int version)
Marshals a Python object, value, to file.

Changed in version 2.4: version indicates the file format.

PyObject* PyMarshal_WriteObjectToString(PyObject *value, int version)
Return a string object containing the marshalled representation of value.

Changed in version 2.4: version indicates the file format.

The following functions allow marshalled values to be read back in.

XXX What about error detection? It appears that reading past the end of the file will always result in a negative numeric value (where that’s relevant), but it’s not clear that negative values won’t be handled properly when there’s no error. What’s the right way to tell? Should only non-negative values be written using these routines?

long PyMarshal_ReadLongFromFile(FILE *file)
Return a C long from the data stream in a FILE* opened for reading. Only a 32-bit value can be read in using this function, regardless of the native size of long.

int PyMarshal_ReadShortFromFile(FILE *file)
Return a C short from the data stream in a FILE* opened for reading. Only a 16-bit value can be read in using this function, regardless of the native size of short.

PyObject* PyMarshal_ReadObjectFromFile(FILE *file)
Return a Python object from the data stream in a FILE* opened for reading. On error, sets the appropriate exception (EOFError or TypeError) and returns NULL.
PyObject* PyMarshal_ReadLastObjectFromFile(FILE *file)


Return a Python object from the data stream in a FILE* opened for reading. Unlike PyMarshal_ReadObjectFromFile(), this function assumes that no further objects will be read from the file, allowing it to aggressively load file data into memory so that the de-serialization can operate from data in memory rather than reading a byte at a time from the file. Only use these variant if you are certain that you won’t be reading anything else from the file. On error, sets the appropriate exception (EOFError or TypeError) and returns NULL.

PyObject* PyMarshal_ReadObjectFromString(char *string, int len)


Return a Python object from the data stream in a character buffer containing len bytes pointed to by string. On error, sets the appropriate exception (EOFError or TypeError) and returns NULL.

5.5 Parsing arguments and building values

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

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

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

‘s’ (string or Unicode object) [const char *] Convert a Python string or Unicode object to a C pointer to a character string. You must not provide storage for the string itself; a pointer to an existing string is stored into the character pointer variable whose address you pass. The C string is NUL-terminated. The Python string must not contain embedded NUL bytes; if it does, a TypeError exception is raised. Unicode objects are converted to C strings using the default encoding. If this conversion fails, a UnicodeError is raised.

‘s#’ (string, Unicode or any read buffer compatible object) [const char *, int] This variant on ‘s’ stores into two C variables, the first one a pointer to a character string, the second one its length. In this case the Python string may contain embedded null bytes. Unicode objects pass back a pointer to the default encoded string version of the object if such a conversion is possible. All other read-buffer compatible objects pass back a reference to the raw internal data representation.

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

‘z#’ (string or None or any read buffer compatible object) [const char *, int] This is to ‘s#’ as ‘z’ is to ‘s’.

‘u’ (Unicode object) [Py_UNICODE *] Convert a Python Unicode object to a C pointer to a NUL-terminated buffer of 16-bit Unicode (UTF-16) data. As with ‘s’, there is no need to provide storage for the Unicode data buffer; a pointer to the existing Unicode data is stored into the Py_UNICODE pointer variable whose address you pass.

‘u#’ (Unicode object) [Py_UNICODE *, int] This variant on ‘u’ stores into two C variables, the first one a pointer to a Unicode data buffer, the second one its length. Non-Unicode objects are handled by interpreting their read-buffer pointer as pointer to a Py_UNICODE array.

‘es’ (string, Unicode object or character buffer compatible object) [const char *encoding, char **buffer] This variant on ‘s’ is used for encoding Unicode and objects convertible to Unicode into a character buffer. It only works for encoded data without embedded NUL bytes.
This format requires two arguments. The first is only used as input, and must be a `const char*` which points to the name of an encoding as a NUL-terminated string, or NULL, in which case the default encoding is used. An exception is raised if the named encoding is not known to Python. The second argument must be a `char**`; the value of the pointer it references will be set to a buffer with the contents of the argument text. The text will be encoded in the encoding specified by the first argument.

PyArg_ParseTuple() will allocate a buffer of the needed size, copy the encoded data into this buffer and adjust `*buffer` to reference the newly allocated storage. The caller is responsible for calling PyMem_Free() to free the allocated buffer after use.

`et` (string, Unicode object or character buffer compatible object) [const char *encoding, char **buffer]
Same as `es` except that 8-bit string objects are passed through without recoding them. Instead, the implementation assumes that the string object uses the encoding passed in as parameter.

`es#` (string, Unicode object or character buffer compatible object) [const char *encoding, char **buffer, int *buffer_length]
This variant on `es#` is used for encoding Unicode and objects convertible to Unicode into a character buffer. Unlike the `es` format, this variant allows input data which contains NUL characters.

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

There are two modes of operation:

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

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

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

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

`b` (integer) [char] Convert a Python integer to a tiny int, stored in a C char.

`B` (integer) [unsigned char] Convert a Python integer to a tiny int without overflow checking, stored in a C unsigned char. New in version 2.3.

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

`H` (integer) [unsigned short int] Convert a Python integer to a C unsigned short int, without overflow checking. New in version 2.3.

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

`I` (integer) [unsigned int] Convert a Python integer to a C unsigned int, without overflow checking. New in version 2.3.

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

`k` (integer) [unsigned long] Convert a Python integer or long integer to a C unsigned long without overflow checking. New in version 2.3.

`L` (integer) [PY_LONG_LONG] Convert a Python integer to a C long long. This format is only available on platforms that support long long (or int64 on Windows).

`K` (integer) [unsigned PY_LONG_LONG] Convert a Python integer or long integer to a C unsigned long long without overflow checking. This format is only available on platforms that support unsigned long long (or unsigned _int64 on Windows). New in version 2.3.
‘c’ (string of length 1) [char] Convert a Python character, represented as a string of length 1, to a C char.

‘f’ (float) [float] Convert a Python floating point number to a C float.

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

‘D’ (complex) [Py_complex] Convert a Python complex number to a C Py_complex structure.

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

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

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

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

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

‘S’ (string) [PyStringObject *] Like ‘O’ but requires that the Python object is a string object. Raises TypeError if the object is not a string object. The C variable may also be declared as PyObject*.

‘U’ (Unicode string) [PyUnicodeObject *] Like ‘O’ but requires that the Python object is a Unicode object. Raises TypeError if the object is not a Unicode object. The C variable may also be declared as PyObject*.

‘t#’ (read-only character buffer) [char *, int] Like ‘s#’, but accepts any object which implements the read-only buffer interface. The char* variable is set to point to the first byte of the buffer, and the int is set to the length of the buffer. Only single-segment buffer objects are accepted; TypeError is raised for all others.

‘w’ (read-write character buffer) [char *] Similar to ‘s’, but accepts any object which implements the read-write buffer interface. The caller must determine the length of the buffer by other means, or use ‘w#’ instead. Only single-segment buffer objects are accepted; TypeError is raised for all others.

‘w#’ (read-write character buffer) [char *, int] Like ‘s#’, but accepts any object which implements the read-write buffer interface. The char* variable is set to point to the first byte of the buffer, and the int is set to the length of the buffer. Only single-segment buffer objects are accepted; TypeError is raised for all others.

‘ (items) ’ (tuple) [matching-items] The object must be a Python sequence whose length is the number of format units in items. The C arguments must correspond to the individual format units in items. Format units for sequences may be nested.

Note: Prior to Python version 1.5.2, this format specifier only accepted a tuple containing the individual parameters, not an arbitrary sequence. Code which previously caused TypeError to be raised here may now proceed without an exception. This is not expected to be a problem for existing code.

It is possible to pass Python long integers where integers are requested; however no proper range checking is done — the most significant bits are silently truncated when the receiving field is too small to receive the value (actually, the semantics are inherited from downcasts in C — your mileage may vary).

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

‘|’ Indicates that the remaining arguments in the Python argument list are optional. The C variables corresponding to optional arguments should be initialized to their default value — when an optional argument is not specified, PyArg_ParseTuple() does not touch the contents of the corresponding C variable(s).
The list of format units ends here; the string after the colon is used as the function name in error messages (the “associated value” of the exception that PyArg_ParseTuple() raises).

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

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

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

For the conversion to succeed, the arg object must match the format and the format must be exhausted. On success, the PyArg_Parse*() functions return true, otherwise they return false and raise an appropriate exception.

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

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

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

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

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

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

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

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

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

Function used to deconstruct the argument lists of “old-style” functions — these are functions which use the METH_OLDARGS parameter parsing method. This is not recommended for use in parameter parsing in new code, and most code in the standard interpreter has been modified to no longer use this for that purpose. It does remain a convenient way to decompose other tuples, however, and may continue to be used for that purpose.

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

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

This is an example of the use of this function, taken from the sources for the _weakref helper module for weak references:
static PyObject *
weakref_ref(PyObject *self, PyObject *args)
{
    PyObject *object;
    PyObject *callback = NULL;
    PyObject *result = NULL;
    if (PyArg_UnpackTuple(args, "ref", 1, 2, &object, &callback)) {
        result = PyWeakref_NewRef(object, callback);
    }
    return result;
}

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

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

New in version 2.2.

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


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

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

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

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

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

‘s’ (string) [char *]Convert a null-terminated C string to a Python object. If the C string pointer is NULL, None is used.

‘s#' (string) [char *, int]Convert a C string and its length to a Python object. If the C string pointer is NULL, the length is ignored and None is returned.

‘z’ (string or None) [char *]Same as ‘s’.

‘z#’ (string or None) [char *, int]Same as ‘s#’.

‘u’ (Unicode string) [Py_UNICODE *]Convert a null-terminated buffer of Unicode (UCS-2 or UCS-4) data to a Python Unicode object. If the Unicode buffer pointer is NULL, None is returned.

‘uf’ (Unicode string) [Py_UNICODE *, int]Convert a Unicode (UCS-2 or UCS-4) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is NULL, the length is ignored and None is returned.

‘i’ (integer) [int]Convert a plain C int to a Python integer object.

‘b’ (integer) [char]Convert a plain C char to a Python integer object.

‘h’ (integer) [short int]Convert a plain C short int to a Python integer object.

‘l’ (integer) [long int]Convert a C long int to a Python integer object.

5.5. Parsing arguments and building values 29
‘B’ (integer) [unsigned char] Convert a C unsigned char to a Python integer object.
‘H’ (integer) [unsigned short int] Convert a C unsigned short int to a Python integer object.
‘I’ (integer/long) [unsigned int] Convert a C unsigned int to a Python integer object or a Python long integer object, if it is larger than sys.maxint.
‘K’ (integer/long) [unsigned long] Convert a C unsigned long to a Python integer object or a Python long integer object, if it is larger than sys.maxint.
‘L’ (long) [PY_LONG_LONG] Convert a C long long to a Python long integer object. Only available on platforms that support long long.
‘K’ (long) [unsigned PY_LONG_LONG] Convert a C unsigned long long to a Python long integer object. Only available on platforms that support unsigned long long.
‘c’ (string of length 1) [char] Convert a C int representing a character to a Python string of length 1.
‘d’ (float) [double] Convert a C double to a Python floating point number.
‘f’ (float) [float] Same as ‘d’.
‘D’ (complex) [Py_complex *] Convert a C Py_complex structure to a Python complex number.
‘O’ (object) [PyObject *] Pass a Python object untouched (except for its reference count, which is incremented by one). If the object passed in is a NULL pointer, it is assumed that this was caused because the call producing the argument found an error and set an exception. Therefore, Py_BuildValue() will return NULL but won’t raise an exception. If no exception has been raised yet, SystemError is set.
‘S’ (object) [PyObject *] Same as ‘O’.
‘W’ (object) [PyObject *] Same as ‘O’, except it doesn’t increment the reference count on the object. Useful when the object is created by a call to an object constructor in the argument list.
‘Os’ (object) [converter, anything] Convert anything to a Python object through a converter function. The function is called with anything (which should be compatible with void *) as its argument and should return a “new” Python object, or NULL if an error occurred.
‘(items)’ (tuple) [matching-items] Convert a sequence of C values to a Python tuple with the same number of items.
‘(items)’ (list) [matching-items] Convert a sequence of C values to a Python list with the same number of items.
‘(items)’ (dictionary) [matching-items] Convert a sequence of C values to a Python dictionary. Each pair of consecutive C values adds one item to the dictionary, serving as key and value, respectively.

If there is an error in the format string, the SystemError exception is set and NULL returned.
Abstract Objects Layer

The functions in this chapter interact with Python objects regardless of their type, or with wide classes of object types (e.g. all numerical types, or all sequence types). When used on object types for which they do not apply, they will raise a Python exception.

6.1 Object Protocol

int PyObject_Print (PyObject *o, FILE *fp, int flags)
Print an object o, on file fp. Returns −1 on error. The flags argument is used to enable certain printing options. The only option currently supported is Py_PRINT_RAW; if given, the str() of the object is written instead of the repr().

int PyObject_HasAttrString (PyObject *o, char *attr_name)
Returns 1 if o has the attribute attr_name, and 0 otherwise. This is equivalent to the Python expression ‘hasattr(o, attr_name)’. This function always succeeds.

PyObject* PyObject_GetAttrString (PyObject *o, char *attr_name)
Retrieve an attribute named attr_name from object o. Returns the attribute value on success, or NULL on failure. This is the equivalent of the Python expression ‘o.attr_name’.

int PyObject_HasAttr (PyObject *o, PyObject *attr_name)
Returns 1 if o has the attribute attr_name, and 0 otherwise. This is equivalent to the Python expression ‘hasattr(o, attr_name)’. This function always succeeds.

PyObject* PyObject_GetAttr (PyObject *o, PyObject *attr_name)
Retrieve an attribute named attr_name from object o. Returns the attribute value on success, or NULL on failure. This is the equivalent of the Python expression ‘o.attr_name’.

int PyObject_SetAttrString (PyObject *o, char *attr_name, PyObject *v)
Set the value of the attribute named attr_name, for object o, to the value v. Returns −1 on failure. This is the equivalent of the Python statement ‘o.attr_name = v’.

int PyObject_SetAttr (PyObject *o, PyObject *attr_name, PyObject *v)
Set the value of the attribute named attr_name, for object o, to the value v. Returns −1 on failure. This is the equivalent of the Python statement ‘o.attr_name = v’.

int PyObject_DelAttrString (PyObject *o, char *attr_name)
Delete attribute named attr_name, for object o. Returns −1 on failure. This is the equivalent of the Python statement ‘del o.attr_name’.

int PyObject_DelAttr (PyObject *o, PyObject *attr_name)
Delete attribute named attr_name, for object o. Returns −1 on failure. This is the equivalent of the Python statement ‘del o.attr_name’.

PyObject* PyObject_RichCompare (PyObject *o1, PyObject *o2, int opid)
Compare the values of o1 and o2 using the operation specified by opid, which must be one of Py_LT,
Py_LE, Py_EQ, Py_NE, Py_GT, or Py_GE, corresponding to <, <=, !=, >, or >= respectively. This
is the equivalent of the Python expression ‘o1 op o2’, where op is the operator corresponding to opid.
Returns the value of the comparison on success, or NULL on failure.

int PyObject_RichCompareBool (PyObject *o1, PyObject *o2, int opid)
Compare the values of o1 and o2 using the operation specified by opid, which must be one of Py_LT,
Py_LE, Py_EQ, Py_NE, Py_GT, or Py_GE, corresponding to <, <=, !=, >, or >= respectively.
Returns -1 on error, 0 if the result is false, 1 otherwise. This is the equivalent of the Python expression ‘o1
op o2’, where op is the operator corresponding to opid.

int PyObject_Cmp (PyObject *o1, PyObject *o2, int *result)
Compare the values of o1 and o2 using a routine provided by o1, if one exists, otherwise with a routine
provided by o2. The result of the comparison is returned in result. Returns -1 on failure. This is the
equivalent of the Python statement ‘result = cmp (o1, o2)’.

int PyObject_Compare (PyObject *o1, PyObject *o2)
Compare the values of o1 and o2 using a routine provided by o1, if one exists, otherwise with a routine
provided by o2. Returns the result of the comparison on success. On error, the value returned is undefined;
use PyErr_Occurred () to detect an error. This is equivalent to the Python expression ‘cmp (o1, o2)’.

PyObject* PyObject_Repr (PyObject *o)
Compute a string representation of object o. Returns the string representation on success, NULL on failure.
This is the equivalent of the Python expression ‘repr (o)’. Called by the repr () built-in function and by
reverse quotes.

PyObject* PyObject_Str (PyObject *o)
Compute a string representation of object o. Returns the string representation on success, NULL on failure.
This is the equivalent of the Python expression ‘str (o)’. Called by the str () built-in function and by
the print statement.

PyObject* PyObject_Unicode (PyObject *o)
Compute a Unicode string representation of object o. Returns the Unicode string representation on suc-
cess, NULL on failure. This is the equivalent of the Python expression ‘unicode (o)’. Called by the
unicode () built-in function.

int PyObject_IsInstance (PyObject *inst, PyObject *cls)
Returns 1 if inst is an instance of the class cls or a subclass of cls, or 0 if not. On error, returns -1 and
sets an exception. If cls is a type object rather than a class object, PyObject_IsInstance () returns
1 if inst is of type cls. If cls is a tuple, the check will be done against every entry in cls. The result will
be 1 when at least one of the checks returns 1, otherwise it will be 0. If inst is not a class instance and cls
is neither a type object, nor a class object, nor a tuple, inst must have a __class__ attribute — the class
relationship of the value of that attribute with cls will be used to determine the result of this function. New
in version 2.1. Changed in version 2.2: Support for a tuple as the second argument added.

Subclass determination is done in a fairly straightforward way, but includes a wrinkle that implementors of exten-
sions to the class system may want to be aware of. If A and B are class objects, B is a subclass of A if it inherits from
A either directly or indirectly. If either is not a class object, a more general mechanism is used to determine the
class relationship of the two objects. When testing if B is a subclass of A, if A is B, PyObject_IsSubclass ()
returns true. If A and B are different objects, B’s __bases__ attribute is searched in a depth-first fashion for A
— the presence of the __bases__ attribute is considered sufficient for this determination.

int PyObject_IsSubclass (PyObject *derived, PyObject *cls)
Returns 1 if the class derived is identical to or derived from the class cls, otherwise returns 0. In case of
an error, returns -1. If cls is a tuple, the check will be done against every entry in cls. The result will be 1
when at least one of the checks returns 1, otherwise it will be 0. If either derived or cls is not an actual class
object (or tuple), this function uses the generic algorithm described above. New in version 2.1. Changed
in version 2.3: Older versions of Python did not support a tuple as the second argument.

int PyCallable_Check (PyObject *o)
Determine if the object o is callable. Return 1 if the object is callable and 0 otherwise. This function always
succeeds.

PyObject* **PyObject_Call** (PyObject *callable_object, PyObject *args, PyObject *kw)
Call a callable Python object callable_object, with arguments given by the tuple args, and named arguments given by the dictionary kw. If no named arguments are needed, kw may be NULL. args must not be NULL, use an empty tuple if no arguments are needed. Returns the result of the call on success, or NULL on failure. This is the equivalent of the Python expression ‘apply (callable_object, args, kw)’ or ‘callable_object(*args, **kw)’. New in version 2.2.

PyObject* **PyObject_CallObject** (PyObject *callable_object, PyObject *args)
Call a callable Python object callable_object, with arguments given by the tuple args. If no arguments are needed, then args may be NULL. Returns the result of the call on success, or NULL on failure. This is the equivalent of the Python expression ‘apply (callable_object, args)’ or ‘callable_object(*args)’.

PyObject* **PyObject_CallFunction** (PyObject *callable, char *format, ...)
Call a callable Python object callable, with a variable number of C arguments. The C arguments are described using a Py_BuildValue() style format string. The format may be NULL, indicating that no arguments are provided. Returns the result of the call on success, or NULL on failure. This is the equivalent of the Python expression ‘apply (callable, args)’ or ‘callable(*args)’.

PyObject* **PyObject_CallMethod** (PyObject *o, char *method, char *format, ...)
Call the method named method of object o with a variable number of C arguments. The C arguments are described by a Py_BuildValue() format string that should produce a tuple. The format may be NULL, indicating that no arguments are provided. Returns the result of the call on success, or NULL on failure. This is the equivalent of the Python expression ‘o . method (args)’.

PyObject* **PyObject_CallFunctionObjArgs** (PyObject *callable, ..., NULL)
Call a callable Python object callable, with a variable number of PyObject* arguments. The arguments are provided as a variable number of parameters followed by NULL. Returns the result of the call on success, or NULL on failure. New in version 2.2.

PyObject* **PyObject_CallMethodObjArgs** (PyObject *o, PyObject *name, ..., NULL)
Calls a method of the object o, where the name of the method is given as a Python string object in name. It is called with a variable number of PyObject* arguments. The arguments are provided as a variable number of parameters followed by NULL. Returns the result of the call on success, or NULL on failure. New in version 2.2.

int **PyObject_Hash** (PyObject *o)
Compute and return the hash value of an object o. On failure, return -1. This is the equivalent of the Python expression ‘hash (o)’.

int **PyObject_IsTrue** (PyObject *o)
Returns 1 if the object o is considered to be true, and 0 otherwise. This is equivalent to the Python expression ‘not not o’. On failure, return -1.

int **PyObject_Not** (PyObject *o)
Returns 0 if the object o is considered to be true, and 1 otherwise. This is equivalent to the Python expression ‘not o’. On failure, return -1.

PyObject* **PyObject_Type** (PyObject *o)
When o is non-NULL, returns a type object corresponding to the object type of object o. On failure, raises SystemError and returns NULL. This is equivalent to the Python expression type (o). This function increments the reference count of the return value. There’s really no reason to use this function instead of the common expression o->ob_type, which returns a pointer of type PyTypeObject*, except when the incremented reference count is needed.

int **PyObject_TypeCheck** (PyObject *o, PyTypeObject *type)
Return true if the object o is of type type or a subtype of type. Both parameters must be non-NULL. New in
version 2.2.

int PyObject_Length(PyObject *o)
int PyObject_Size(PyObject *o)

Return the length of object o. If the object o provides either the sequence and mapping protocols, the sequence length is returned. On error, -1 is returned. This is the equivalent to the Python expression ‘len(o)’.

PyObject* PyObject_GetItem(PyObject *o, PyObject *key)

Return element of o corresponding to the object key or NULL on failure. This is the equivalent of the Python expression ‘o[key]’.

int PyObject_SetItem(PyObject *o, PyObject *key, PyObject *v)

Map the object key to the value v. Returns -1 on failure. This is the equivalent of the Python statement ‘o[key] = v’.

int PyObject_DelItem(PyObject *o, PyObject *key)

Delete the mapping for key from o. Returns -1 on failure. This is the equivalent of the Python statement ‘del o[key]’.

int PyNumber_Check(PyObject *o)

Returns 1 if the object o provides numeric protocols, and false otherwise. This function always succeeds.

PyObject* PyNumber_Add(PyObject *o1, PyObject *o2)

Returns the result of adding o1 and o2, or NULL on failure. This is the equivalent of the Python expression ‘o1 + o2’.

PyObject* PyNumber_Subtract(PyObject *o1, PyObject *o2)

Returns the result of subtracting o2 from o1, or NULL on failure. This is the equivalent of the Python expression ‘o1 - o2’.

PyObject* PyNumber_Multiply(PyObject *o1, PyObject *o2)

Returns the result of multiplying o1 and o2, or NULL on failure. This is the equivalent of the Python expression ‘o1 * o2’.

PyObject* PyNumber_Divide(PyObject *o1, PyObject *o2)

Returns the result of dividing o1 by o2, or NULL on failure. This is the equivalent of the Python expression ‘o1 / o2’.

6.2 Number Protocol

int PyNumber_Check(PyObject *o)

Returns 1 if the object o provides numeric protocols, and false otherwise. This function always succeeds.

PyObject* PyNumber_Add(PyObject *o1, PyObject *o2)

Returns the result of adding o1 and o2, or NULL on failure. This is the equivalent of the Python expression ‘o1 + o2’.

PyObject* PyNumber_Subtract(PyObject *o1, PyObject *o2)

Returns the result of subtracting o2 from o1, or NULL on failure. This is the equivalent of the Python expression ‘o1 - o2’.

PyObject* PyNumber_Multiply(PyObject *o1, PyObject *o2)

Returns the result of multiplying o1 and o2, or NULL on failure. This is the equivalent of the Python expression ‘o1 * o2’.

PyObject* PyNumber_Divide(PyObject *o1, PyObject *o2)

Returns the result of dividing o1 by o2, or NULL on failure. This is the equivalent of the Python expression ‘o1 / o2’.
PyObject* PyNumber_FloorDivide(PyObject *o1, PyObject *o2)  
Return the floor of o1 divided by o2, or NULL on failure. This is equivalent to the “classic” division of integers. New in version 2.2.

PyObject* PyNumber_TrueDivide(PyObject *o1, PyObject *o2)  
Return a reasonable approximation for the mathematical value of o1 divided by o2, or NULL on failure. The return value is “approximate” because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers. New in version 2.2.

PyObject* PyNumber_Remainder(PyObject *o1, PyObject *o2)  
Returns the remainder of dividing o1 by o2, or NULL on failure. This is the equivalent of the Python expression 'o1 % o2'.

PyObject* PyNumber_Divmod(PyObject *o1, PyObject *o2)  
See the built-in function divmod(). Returns NULL on failure. This is the equivalent of the Python expression 'divmod(o1, o2)'.

PyObject* PyNumber_Power(PyObject *o1, PyObject *o2, PyObject *o3)  
See the built-in function pow(). Returns NULL on failure. This is the equivalent of the Python expression 'pow(o1, o2, o3)'. where o3 is optional. If o3 is to be ignored, pass Py_None in its place (passing NULL for o3 would cause an illegal memory access).

PyObject* PyNumber_Negative(PyObject *o)  
Returns the negation of o on success, or NULL on failure. This is the equivalent of the Python expression '-o'.

PyObject* PyNumber_Positive(PyObject *o)  
Returns o on success, or NULL on failure. This is the equivalent of the Python expression '+o'.

PyObject* PyNumber.Absolute(PyObject *o)  
Returns the absolute value of o, or NULL on failure. This is the equivalent of the Python expression 'abs(o)'.

PyObject* PyNumber_Invert(PyObject *o)  
Returns the bitwise negation of o on success, or NULL on failure. This is the equivalent of the Python expression '˜o'.

PyObject* PyNumber_Lshift(PyObject *o1, PyObject *o2)  
Returns the result of left shifting o1 by o2 on success, or NULL on failure. This is the equivalent of the Python expression 'o1 << o2'.

PyObject* PyNumber_Rshift(PyObject *o1, PyObject *o2)  
Returns the result of right shifting o1 by o2 on success, or NULL on failure. This is the equivalent of the Python expression 'o1 >> o2'.

PyObject* PyNumber_And(PyObject *o1, PyObject *o2)  
Returns the “bitwise and” of o1 and o2 on success and NULL on failure. This is the equivalent of the Python expression 'o1 & o2'.

PyObject* PyNumber_Xor(PyObject *o1, PyObject *o2)  

6.2. Number Protocol
Returns the “bitwise exclusive or” of \( o1 \) by \( o2 \) on success, or NULL on failure. This is the equivalent of the Python expression \( o1 \ ^{\dagger} o2 \).

PyObject* PyNumber_Or (PyObject *o1, PyObject *o2)
Returns the “bitwise or” of \( o1 \) and \( o2 \) on success, or NULL on failure. This is the equivalent of the Python expression \( o1 \ | \ o2 \).

PyObject* PyNumber_InPlaceAdd (PyObject *o1, PyObject *o2)
Returns the result of adding \( o1 \) and \( o2 \), or NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python statement \( o1 \ += o2 \).

PyObject* PyNumber_InPlaceSubtract (PyObject *o1, PyObject *o2)
Returns the result of subtracting \( o2 \) from \( o1 \), or NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python statement \( o1 \ -= o2 \).

PyObject* PyNumber_InPlaceMultiply (PyObject *o1, PyObject *o2)
Returns the result of multiplying \( o1 \) and \( o2 \), or NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python statement \( o1 \ *= o2 \).

PyObject* PyNumber_InPlaceDivide (PyObject *o1, PyObject *o2)
Returns the result of dividing \( o1 \) by \( o2 \), or NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python statement \( o1 \ /= o2 \).

PyObject* PyNumber_InPlaceFloorDivide (PyObject *o1, PyObject *o2)
Returns the mathematical floor of dividing \( o1 \) by \( o2 \), or NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python statement \( o1 \ //= o2 \). New in version 2.2.

PyObject* PyNumber_InPlaceTrueDivide (PyObject *o1, PyObject *o2)
Return a reasonable approximation for the mathematical value of \( o1 \) divided by \( o2 \), or NULL on failure. The return value is “approximate” because binary floating point numbers are approximate; it is not possible to represent all real numbers in base two. This function can return a floating point value when passed two integers. The operation is done in-place when \( o1 \) supports it. New in version 2.2.

PyObject* PyNumber_InPlaceRemainder (PyObject *o1, PyObject *o2)
Returns the remainder of dividing \( o1 \) by \( o2 \), or NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python statement \( o1 \ %= o2 \).

PyObject* PyNumber_InPlacePower (PyObject *o1, PyObject *o2, PyObject *o3)
See the built-in function pow(). Returns NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python statement \( o1 \ **= o2 \) when \( o3 \) is Py_NULL, or an in-place variant of ‘pow(o1, o2, o3)’ otherwise. If \( o3 \) is to be ignored, pass Py_NULL in its place (passing NULL for \( o3 \) would cause an illegal memory access).

PyObject* PyNumber_InPlaceLshift (PyObject *o1, PyObject *o2)
Returns the result of left shifting \( o1 \) by \( o2 \) on success, or NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python statement \( o1 \ <<= o2 \).

PyObject* PyNumber_InPlaceRshift (PyObject *o1, PyObject *o2)
Returns the result of right shifting \( o1 \) by \( o2 \) on success, or NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python statement \( o1 \ >>= o2 \).

PyObject* PyNumber_InPlaceAnd (PyObject *o1, PyObject *o2)
Returns the “bitwise and” of \( o1 \) and \( o2 \) on success and NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python statement ‘\( o1 \&= o2 \)’.

```c
PyObject* PyNumber_InPlaceAnd (PyObject *o1, PyObject *o2)
Returns the “bitwise and” of \( o1 \) and \( o2 \) on success, or NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python statement ‘\( o1 \&= o2 \)’.
```

```c
PyObject* PyNumber_InPlaceXor (PyObject *o1, PyObject *o2)
Returns the “bitwise exclusive or” of \( o1 \) by \( o2 \) on success, or NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python statement ‘\( o1 ^= o2 \)’.
```

```c
PyObject* PyNumber_InPlaceOr (PyObject *o1, PyObject *o2)
Returns the “bitwise or” of \( o1 \) and \( o2 \) on success, or NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python statement ‘\( o1 |= o2 \)’.
```

```c
int PyNumber_Coerce (PyObject **p1, PyObject **p2)
This function takes the addresses of two variables of type PyObject*. If the objects pointed to by \( *p1 \) and \( *p2 \) have the same type, increment their reference count and return 0 (success). If the objects can be converted to a common numeric type, replace \( *p1 \) and \( *p2 \) by their converted value (with ‘new’ reference counts), and return 0. If no conversion is possible, or if some other error occurs, return -1 (failure) and don’t increment the reference counts. The call PyNumber_Coerce(&\( o1 \), &\( o2 \)) is equivalent to the Python statement ‘\( o1, o2 = coerce(o1, o2) \)’.
```

```c
PyObject* PyNumber_Int (PyObject *o)
Returns the \( o \) converted to an integer object on success, or NULL on failure. If the argument is outside the integer range a long object will be returned instead. This is the equivalent of the Python expression ‘int(\( o \))’.
```

```c
PyObject* PyNumber_Long (PyObject *o)
Returns the \( o \) converted to a long integer object on success, or NULL on failure. This is the equivalent of the Python expression ‘long(\( o \))’.
```

```c
PyObject* PyNumber_Float (PyObject *o)
Returns the \( o \) converted to a float object on success, or NULL on failure. This is the equivalent of the Python expression ‘float(\( o \))’.
```

### 6.3. Sequence Protocol

```c
int PySequence_Check (PyObject *o)
Return 1 if the object provides sequence protocol, and 0 otherwise. This function always succeeds.
```

```c
int PySequence_Size (PyObject *o)
Returns the number of objects in sequence \( o \) on success, and -1 on failure. For objects that do not provide sequence protocol, this is equivalent to the Python expression ‘\( len(o) \)’.
```

```c
int PySequence_Length (PyObject *o)
Alternate name for PySequence_Size().
```

```c
PyObject* PySequence_Concat (PyObject *o1, PyObject *o2)
Return the concatenation of \( o1 \) and \( o2 \) on success, and NULL on failure. This is the equivalent of the Python expression ‘\( o1 + o2 \)’.
```

```c
PyObject* PySequence_Repeat (PyObject *o, int count)
Return the result of repeating sequence object \( o \) \( \times \) count times, or NULL on failure. This is the equivalent of the Python expression ‘\( o \times count \)’.
```

```c
PyObject* PySequence_InPlaceConcat (PyObject *o1, PyObject *o2)
Return the concatenation of \( o1 \) and \( o2 \) on success, and NULL on failure. The operation is done in-place when \( o1 \) supports it. This is the equivalent of the Python expression ‘\( o1 += o2 \)’.
```

PyObject* PySequence_InPlaceRepeat (PyObject *o, int count)
Return the result of repeating sequence object o count times, or NULL on failure. The operation is done
in-place when o supports it. This is the equivalent of the Python expression ‘o *= count’.

PyObject* PySequence_GetItem (PyObject *o, int i)
Return the i\textsuperscript{th} element of o, or NULL on failure. This is the equivalent of the Python expression ‘o[i]’.

PyObject* PySequence_GetSlice (PyObject *o, int i1, int i2)
Return the slice of sequence object o between i1 and i2, or NULL on failure. This is the equivalent of the Python expression ‘o[i1:i2]’.

int PySequence_SetItem (PyObject *o, int i, PyObject *v)
Assign object v to the i\textsuperscript{th} element of o. Returns −1 on failure. This is the equivalent of the Python statement
‘o[i] = v’. This function does not steal a reference to v.

int PySequence_DelItem (PyObject *o, int i)
delete o[i] on failure. This is the equivalent of the Python statement
‘del o[i]’.

int PySequence_SetSlice (PyObject *o, int i1, int i2, PyObject *v)
Assign the sequence object v to the slice in sequence object o from i1 to i2. This is the equivalent of the Python statement
‘o[i1:i2] = v’.

int PySequence_DelSlice (PyObject *o, int i1, int i2)
delete the slice in sequence object o from i1 to i2. Returns −1 on failure. This is the equivalent of the Python statement
‘del o[i1:i2]’.

int PySequence_Count (PyObject *o, PyObject *value)
Return the number of occurrences of value in o, that is, return the number of keys for which o[key] == value. On failure, return −1. This is equivalent to the Python expression ‘o.count(value)’.

int PySequence_Contains (PyObject *o, PyObject *value)
Determine if o contains value. If an item in o is equal to value, return 1, otherwise return 0. On error, return −1. This is equivalent to the Python expression ‘value in o’.

int PySequence_Index (PyObject *o,PyObject *value)
Return the first index i for which o[i] == value. On error, return −1. This is equivalent to the Python expression ‘o.index(value)’.

PyObject* PySequence_List (PyObject *o)
Return a list object with the same contents as the arbitrary sequence o. The returned list is guaranteed to be new.

PyObject* PySequence_Tuple (PyObject *o)
Return a tuple object with the same contents as the arbitrary sequence o or NULL on failure. If o is a tuple, a new reference will be returned, otherwise a tuple will be constructed with the appropriate contents. This is equivalent to the Python expression ‘tuple(o)’.

PyObject* PySequence_Fast (PyObject *o, const char *m)
Returns the sequence o as a tuple, unless it is already a tuple or list, in which case o is returned. Use PySequence_Fast_GET_ITEM() to access the members of the result. Returns NULL on failure. If the object is not a sequence, raises TypeError with m as the message text.

PyObject* PySequence_Fast_GET_ITEM (PyObject *o, int i)
Return value: Borrowed reference.
Return the i\textsuperscript{th} element of o, assuming that o was returned by PySequence_Fast(), o is not NULL, and that i is within bounds.

PyObject** PySequence_Fast_ITEMS (PyObject *o)
Return the underlying array of PyObject pointers. Assumes that o was returned by PySequence_Fast() and o is not NULL. New in version 2.4.

PyObject* PySequence_ITEM(PyObject *o, int i)
Return the i\textsuperscript{th} element of o or NULL on failure. Macro form of PySequence_GetItem() but without checking that PySequence_Check(o) is true and without adjustment for negative indices. New in version 2.3.

int PySequence_Fast_GET_SIZE(PyObject *o)
Returns the length of o, assuming that o was returned by PySequence_Fast() and that o is not NULL. The size can also be gotten by calling PySequence_Size() on o, but PySequence_Fast_GET_SIZE() is faster because it can assume o is a list or tuple.

6.4 Mapping Protocol

int PyMapping_Check(PyObject *o)
Return 1 if the object provides mapping protocol, and 0 otherwise. This function always succeeds.

int PyMapping_Length(PyObject *o)
Returns the number of keys in object o on success, and -1 on failure. For objects that do not provide mapping protocol, this is equivalent to the Python expression ‘\texttt{len(o)}’.

int PyMapping_DeleteString(PyObject *o, char *key)
Remove the mapping for object key from the object o. Return -1 on failure. This is equivalent to the Python statement ‘\texttt{del o[key]}’.

int PyMapping_Delete(PyObject *o, PyObject *key)
Remove the mapping for object key from the object o. Return -1 on failure. This is equivalent to the Python statement ‘\texttt{del o[key]}’.

int PyMappingHasKeyString(PyObject *o, char *key)
On success, return 1 if the mapping object has the key key and 0 otherwise. This is equivalent to the Python expression ‘\texttt{o.has_key(key)}’. This function always succeeds.

int PyMappingHasKey(PyObject *o, PyObject *key)
Return 1 if the mapping object has the key key and 0 otherwise. This is equivalent to the Python expression ‘\texttt{o.has_key(key)}’. This function always succeeds.

PyObject* PyMapping_Keys(PyObject *o)
On success, return a list of the keys in object o. On failure, return NULL. This is equivalent to the Python expression ‘\texttt{o.keys()’.

PyObject* PyMapping_Values(PyObject *o)
On success, return a list of the values in object o. On failure, return NULL. This is equivalent to the Python expression ‘\texttt{o.values()}’.

PyObject* PyMapping_Items(PyObject *o)
On success, return a list of the items in object o, where each item is a tuple containing a key-value pair. On failure, return NULL. This is equivalent to the Python expression ‘\texttt{o.items()}’.

PyObject* PyMapping_GetItemString(PyObject *o, char *key)
Return element of o corresponding to the object key or NULL on failure. This is the equivalent of the Python expression ‘\texttt{o[key]}’.

int PyMapping_SetItemString(PyObject *o, char *key, PyObject *v)
Map the object key to the value v in object o. Returns -1 on failure. This is the equivalent of the Python statement ‘\texttt{o[key] = v}’.

6.4. Mapping Protocol 39
6.5 Iterator Protocol

New in version 2.2.

There are only a couple of functions specifically for working with iterators.

```c
int PyIter_Check(PyObject *o)
Return true if the object o supports the iterator protocol.
```

```c
PyObject* PyIter_Next(PyObject *o)
Return the next value from the iteration o. If the object is an iterator, this retrieves the next value from the iteration, and returns NULL with no exception set if there are no remaining items. If the object is not an iterator, TypeError is raised, or if there is an error in retrieving the item, returns NULL and passes along the exception.
```

To write a loop which iterates over an iterator, the C code should look something like this:

```c
PyObject *iterator = PyObject_GetIter(obj);
PyObject *item;
if (iterator == NULL) {
    /* propagate error */
}
while (item = PyIter_Next(iterator)) {
    /* do something with item */
    ...
    /* release reference when done */
    Py_DECREF(item);
}
Py_DECREF(iterator);
if (PyErr_Occurred()) {
    /* propagate error */
} else {
    /* continue doing useful work */
}
```

6.6 Buffer Protocol

```c
int PyObject_AsCharBuffer(PyObject *obj, const char **buffer, int *buffer_len)
Returns a pointer to a read-only memory location useable as character-based input. The obj argument must support the single-segment character buffer interface. On success, returns 0, sets buffer to the memory location and buffer_len to the buffer length. Returns -1 and sets a TypeError on error. New in version 1.6.
```

```c
int PyObject_AsReadBuffer(PyObject *obj, const void **buffer, int *buffer_len)
Returns a pointer to a read-only memory location containing arbitrary data. The obj argument must support the single-segment readable buffer interface. On success, returns 0, sets buffer to the memory location and buffer_len to the buffer length. Returns -1 and sets a TypeError on error. New in version 1.6.
```

```c
int PyObject_CheckReadBuffer(PyObject *o)
Returns 1 if o supports the single-segment readable buffer interface. Otherwise returns 0. New in version 2.2.
```

```c
int PyObject_AsWriteBuffer(PyObject *obj, void **buffer, int *buffer_len)
Returns a pointer to a writeable memory location. The obj argument must support the single-segment,
character buffer interface. On success, returns 0, sets buffer to the memory location and buffer_len to the buffer length. Returns -1 and sets a TypeError on error. New in version 1.6.
CHAPTER
SEVEN

Concrete Objects Layer

The functions in this chapter are specific to certain Python object types. Passing them an object of the wrong type is not a good idea; if you receive an object from a Python program and you are not sure that it has the right type, you must perform a type check first; for example, to check that an object is a dictionary, use `PyDict_Check()`. The chapter is structured like the “family tree” of Python object types.

**Warning:** While the functions described in this chapter carefully check the type of the objects which are passed in, many of them do not check for NULL being passed instead of a valid object. Allowing NULL to be passed in can cause memory access violations and immediate termination of the interpreter.

7.1 Fundamental Objects

This section describes Python type objects and the singleton object `None`.

7.1.1 Type Objects

`PyTypeObject`

The C structure of the objects used to describe built-in types.

`PyObject* PyType_Type`

This is the type object for type objects; it is the same object as `types.TypeType` in the Python layer.

`int PyType_Check (PyObject *o)`

Return true if the object `o` is a type object, including instances of types derived from the standard type object. Return false in all other cases.

`int PyType_CheckExact (PyObject *o)`

Return true if the object `o` is a type object, but not a subtype of the standard type object. Return false in all other cases. New in version 2.2.

`int PyType_HasFeature (PyObject *o, int feature)`

Return true if the type object `o` sets the feature `feature`. Type features are denoted by single bit flags.

`int PyType_IS_GC (PyObject *o)`

Return true if the type object includes support for the cycle detector; this tests the type flag `Py_TPFLAGS_HAVE_GC`. New in version 2.0.

`int PyType_IsSubtype (PyTypeObject *a, PyTypeObject *b)`

Return true if `a` is a subtype of `b`. New in version 2.2.

`PyObject* PyType_GenericAlloc (PyTypeObject *type, int nitems)`

*Return value: New reference.*

New in version 2.2.

`PyObject* PyType_GenericNew (PyTypeObject *type, PyObject *args, PyObject *kwds)`

*Return value: New reference.*

New in version 2.2.
int PyType_Ready (PyTypeObject *type)
    Finalize a type object. This should be called on all type objects to finish their initialization. This function is responsible for adding inherited slots from a type’s base class. Return 0 on success, or return -1 and sets an exception on error. New in version 2.2.

7.1.2 The None Object

Note that the PyTypeObject for None is not directly exposed in the Python/C API. Since None is a singleton, testing for object identity (using ‘==’ in C) is sufficient. There is no PyNone_Check() function for the same reason.

PyObject* Py_None
    The Python None object, denoting lack of value. This object has no methods. It needs to be treated just like any other object with respect to reference counts.

Py_RETURN_NONE
    Properly handle returning Py_None from within a C function.

7.2 Numeric Objects

7.2.1 Plain Integer Objects

PyIntObject
    This subtype of PyObject represents a Python integer object.

PyTypeObject PyInt_Type
    This instance of PyTypeObject represents the Python plain integer type. This is the same object as types.IntType.

int PyInt_Check (PyObject *o)
    Return true if o is of type PyInt_Type or a subtype of PyInt_Type. Changed in version 2.2: Allowed subtypes to be accepted.

int PyInt_CheckExact (PyObject *o)
    Return true if o is of type PyInt_Type, but not a subtype of PyInt_Type. New in version 2.2.

PyObject* PyInt_FromString (char *str, char **pend, int base)
    Return a new PyIntObject or PyLongObject based on the string value in str, which is interpreted according to the radix in base. If pend is non-NULL, *pend will point to the first character in str which follows the representation of the number. If base is 0, the radix will be determined based on the leading characters of str: if str starts with ‘0x’ or ‘0X’, radix 16 will be used; if str starts with ‘0’, radix 8 will be used; otherwise radix 10 will be used. If base is not 0, it must be between 2 and 36, inclusive. Leading spaces are ignored. If there are no digits, ValueError will be raised. If the string represents a number too large to be contained within the machine’s long int type and overflow warnings are being suppressed, a PyLongObject will be returned. If overflow warnings are not being suppressed, NULL will be returned in this case.

PyObject* PyInt_FromLong (long ival)
    Create a new integer object with a value of ival.
    The current implementation keeps an array of integer objects for all integers between −1 and 100, when you create an int in that range you actually just get back a reference to the existing object. So it should be possible to change the value of 1. I suspect the behaviour of Python in this case is undefined. :-)

long PyInt_AsLong (PyObject *io)
    Will first attempt to cast the object to a PyIntObject, if it is not already one, and then return its value.

long PyInt_AS_LONG (PyObject *io)
    Return the value of the object io. No error checking is performed.
unsigned long PyInt_AsUnsignedLongMask (PyObject *io)
Will first attempt to cast the object to a PyIntObject or PyLongObject, if it is not already one, and then return its value as unsigned long. This function does not check for overflow. New in version 2.3.

unsigned long long PyInt_AsUnsignedLongLongMask (PyObject *io)
Will first attempt to cast the object to a PyIntObject or PyLongObject, if it is not already one, and then return its value as unsigned long long, without checking for overflow. New in version 2.3.

long PyInt_GetMax ()
Return the system’s idea of the largest integer it can handle (LONG_MAX, as defined in the system header files).

7.2.2 Boolean Objects

Booleans in Python are implemented as a subclass of integers. There are only two booleans, Py_False and Py_True. As such, the normal creation and deletion functions don’t apply to booleans. The following macros are available, however.

int PyBool_Check (PyObject *o)
Return true if o is of type PyBool_Type. New in version 2.3.

PyObject* Py_FALSE
The Python False object. This object has no methods. It needs to be treated just like any other object with respect to reference counts.

PyObject* Py_TRUE
The Python True object. This object has no methods. It needs to be treated just like any other object with respect to reference counts.

Py_RETURN_FALSE
Return Py_FALSE from a function, properly incrementing its reference count. New in version 2.4.

Py_RETURN_TRUE
Return Py_TRUE from a function, properly incrementing its reference count. New in version 2.4.

PyObject* PyBool_FromLong (long v)
Return a new reference to Py_TRUE or Py_FALSE depending on the truth value of v. New in version 2.3.

7.2.3 Long Integer Objects

PyLongObject
This subtype of PyObject represents a Python integer object.

PyTypeObject PyLong_Type
This instance of PyTypeObject represents the Python long integer type. This is the same object as types.LongType.

int PyLong_Check (PyObject *p)
Return true if its argument is a PyLongObject or a subtype of PyLongObject. Changed in version 2.2: Allowed subtypes to be accepted.

int PyLong_CheckExact (PyObject *p)
Return true if its argument is a PyLongObject, but not a subtype of PyLongObject. New in version 2.2.

PyObject* PyLong_FromLong (long v)
Return a new PyLongObject object from v, or NULL on failure.

PyObject* PyLong_FromUnsignedLong (unsigned long v)
Return a new PyLongObject object from a C unsigned long, or NULL on failure.
PyObject* PyLong_FromLongLong (long long v)
Return a new PyLongObject object from a C long long, or NULL on failure.

PyObject* PyLong_FromUnsignedLongLong (unsigned long long v)
Return a new PyLongObject object from a C unsigned long long, or NULL on failure.

PyObject* PyLong_FromDouble (double v)
Return a new PyLongObject object from the integer part of v, or NULL on failure.

PyObject* PyLong_FromString (char *str, char **pend, int base)
Return a new PyLongObject based on the string value in str, which is interpreted according to the radix in base. If pend is non-NULL, *pend will point to the first character in str which follows the representation of the number. If base is 0, the radix will be determined based on the leading characters of str: if str starts with ‘0x’ or ‘0X’, radix 16 will be used; if str starts with ‘0’, radix 8 will be used; otherwise radix 10 will be used. If base is not 0, it must be between 2 and 36, inclusive. Leading spaces are ignored. If there are no digits, ValueError will be raised.

PyObject* PyLong_FromUnicode (Py_UNICODE *u, int length, int base)
Convert a sequence of Unicode digits to a Python long integer value. The first parameter, u, points to the first character of the Unicode string, length gives the number of characters, and base is the radix for the conversion. The radix must be in the range [2, 36]; if it is out of range, ValueError will be raised. New in version 1.6.

PyObject* PyLong_FromVoidPtr (void *p)
Create a Python integer or long integer from the pointer p. The pointer value can be retrieved from the resulting value using PyLong_AsVoidPtr(). New in version 1.5.2.

long PyLong_AsLong (PyObject *pylong)
Return a C long representation of the contents of pylong. If pylong is greater than LONG_MAX, an OverflowError is raised.

unsigned long PyLong_AsUnsignedLong (PyObject *pylong)
Return a C unsigned long representation of the contents of pylong. If pylong is greater than ULONG_MAX, an OverflowError is raised.

long long PyLong_AsLongLong (PyObject *pylong)
Return a C long long from a Python long integer. If pylong cannot be represented as a long long, an OverflowError will be raised. New in version 2.2.

unsigned long long PyLong_AsUnsignedLongLong (PyObject *pylong)
Return a C unsigned long long from a Python long integer. If pylong cannot be represented as an unsigned long long, an OverflowError will be raised if the value is positive, or a TypeError will be raised if the value is negative. New in version 2.2.

unsigned long PyLong_AsUnsignedLongMask (PyObject *io)
Return a C unsigned long from a Python long integer, without checking for overflow. New in version 2.3.

unsigned long PyLong_AsUnsignedLongLongMask (PyObject *io)
Return a C unsigned long long from a Python long integer, without checking for overflow. New in version 2.3.

double PyLong_AsDouble (PyObject *pylong)
Return a C double representation of the contents of pylong. If pylong cannot be approximately represented as a double, an OverflowError exception is raised and -1.0 will be returned.

void* PyLong_AsVoidPtr (PyObject *pylong)
Convert a Python integer or long integer pylong to a C void pointer. If pylong cannot be converted, an OverflowError will be raised. This is only assured to produce a usable void pointer for values created
7.2.4 Floating Point Objects

**PyFloatObject**

This subtype of PyObject represents a Python floating point object.

**PyFloatObject**

This instance of PyTypeObject represents the Python floating point type. This is the same object as types.FloatType.

```c
int PyFloat_Check(PyObject *p)
```

Return true if its argument is a PyFloatObject or a subtype of PyFloatObject. Changed in version 2.2: Allowed subtypes to be accepted.

```c
int PyFloat_CheckExact(PyObject *p)
```

Return true if its argument is a PyFloatObject, but not a subtype of PyFloatObject. New in version 2.2.

```c
PyObject* PyFloat_FromString(PyObject *str, char **pend)
```

Create a PyFloatObject object based on the string value in str, or NULL on failure. The pend argument is ignored. It remains only for backward compatibility.

```c
PyObject* PyFloat_FromDouble(double v)
```

Return a C double representation of the contents of pyfloat.

```c
double PyFloat_AsDouble(PyObject *pyfloat)
```

Return a C double representation of the contents of pyfloat, but without error checking.

7.2.5 Complex Number Objects

Python’s complex number objects are implemented as two distinct types when viewed from the C API: one is the Python object exposed to Python programs, and the other is a C structure which represents the actual complex number value. The API provides functions for working with both.

**Complex Numbers as C Structures**

Note that the functions which accept these structures as parameters and return them as results do so by value rather than dereferencing them through pointers. This is consistent throughout the API.

```c
Py_complex
```

The C structure which corresponds to the value portion of a Python complex number object. Most of the functions for dealing with complex number objects use structures of this type as input or output values, as appropriate. It is defined as:

```c
typedef struct {
    double real;
    double imag;
} Py_complex;
```

```c
Py_complex _Py_c_sum(Py_complex left, Py_complex right)
```

Return the sum of two complex numbers, using the C Py_complex representation.

```c
Py_complex _Py_c_diff(Py_complex left, Py_complex right)
```

Return the difference between two complex numbers, using the C Py_complex representation.

```c
Py_complex _Py_c_neg(Py_complex complex)
```

Return the negation of the complex number complex, using the C Py_complex representation.
Py_complex _Py_c_prod(Py_complex left, Py_complex right)
    Return the product of two complex numbers, using the C Py_complex representation.

Py_complex _Py_c_quot(Py_complex dividend, Py_complex divisor)
    Return the quotient of two complex numbers, using the C Py_complex representation.

Py_complex _Py_c_pow(Py_complex num, Py_complex exp)
    Return the exponentiation of num by exp, using the C Py_complex representation.

Complex Numbers as Python Objects

PyComplexObject
    This subtype of PyObject represents a Python complex number object.

PyTypeObject PyComplex_Type
    This instance of PyTypeObject represents the Python complex number type.

int PyComplex_Check(PyObject *p)
    Return true if its argument is a PyComplexObject or a subtype of PyComplexObject. Changed in version 2.2: Allowed subtypes to be accepted.

int PyComplex_CheckExact(PyObject *p)
    Return true if its argument is a PyComplexObject, but not a subtype of PyComplexObject. New in version 2.2.

PyObject* PyComplex_FromCComplex(Py_complex v)
    Create a new Python complex number object from a C Py_complex value.

PyObject* PyComplex_FromDoubles(double real, double imag)
    Return a new PyComplexObject object from real and imag.

double PyComplex_RealAsDouble(PyObject *op)
    Return the real part of op as a C double.

double PyComplex_ImagAsDouble(PyObject *op)
    Return the imaginary part of op as a C double.

Py_complex PyComplex_AsCComplex(PyObject *op)
    Return the Py_complex value of the complex number op.

7.3 Sequence Objects

Generic operations on sequence objects were discussed in the previous chapter; this section deals with the specific kinds of sequence objects that are intrinsic to the Python language.

7.3.1 String Objects

These functions raise TypeError when expecting a string parameter and are called with a non-string parameter.

PyStringObject
    This subtype of PyObject represents a Python string object.

PyTypeObject PyString_Type
    This instance of PyTypeObject represents the Python string type; it is the same object as types.TypeType in the Python layer.

int PyString_Check(PyObject *o)
    Return true if the object o is a string object or an instance of a subtype of the string type. Changed in version 2.2: Allowed subtypes to be accepted.
int **PyString_CheckExact**(PyObject *o)
Return true if the object o is a string object, but not an instance of a subtype of the string type. New in version 2.2.

PyObject** **PyString_FromString**(const char *v)
*Return value: New reference.*
Return a new string object with the value v on success, and NULL on failure. The parameter v must not be NULL; it will not be checked.

PyObject** **PyString_FromStringAndSize**(const char *v, int len)
*Return value: New reference.*
Return a new string object with the value v and length len on success, and NULL on failure. If v is NULL, the contents of the string are uninitialized.

PyObject** **PyString_FromFormat**(const char *format, ...)
*Return value: New reference.*
Take a C printf()-style format string and a variable number of arguments, calculate the size of the resulting Python string and return a string with the values formatted into it. The variable arguments must be C types and must correspond exactly to the format characters in the format string. The following format characters are allowed:

<table>
<thead>
<tr>
<th>Format Characters</th>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>%n/a</td>
<td>n/a</td>
<td>The literal % character.</td>
</tr>
<tr>
<td>%c</td>
<td>int</td>
<td>A single character, represented as an C int.</td>
</tr>
<tr>
<td>%d</td>
<td>int</td>
<td>Exactly equivalent to printf(&quot;%d&quot;).</td>
</tr>
<tr>
<td>%ld</td>
<td>long</td>
<td>Exactly equivalent to printf(&quot;%ld&quot;).</td>
</tr>
<tr>
<td>%i</td>
<td>int</td>
<td>Exactly equivalent to printf(&quot;%i&quot;).</td>
</tr>
<tr>
<td>%x</td>
<td>int</td>
<td>Exactly equivalent to printf(&quot;%x&quot;).</td>
</tr>
<tr>
<td>%s</td>
<td>char*</td>
<td>A null-terminated C character array.</td>
</tr>
<tr>
<td>%p</td>
<td>void*</td>
<td>The hex representation of a C pointer. Mostly equivalent to printf(&quot;%p&quot;) except that it</td>
</tr>
</tbody>
</table>

PyObject** **PyString_FromFormatV**(const char *format, va_list vargs)
*Return value: New reference.*
Identical to PyString_FromFormat() except that it takes exactly two arguments.

int **PyString_Size**(PyObject *string)
Return the length of the string in string object string.

int **PyString_GET_SIZE**(PyObject *string)
Macro form of PyString_Size() but without error checking.

char** **PyString_AsString**(PyObject *string)
Return a NUL-terminated representation of the contents of string. The pointer refers to the internal buffer of string, not a copy. The data must not be modified in any way, unless the string was just created using PyString_FromStringAndSize(NULL, size). It must not be deallocated. If string is a Unicode object, this function computes the default encoding of string and operates on that. If string is not a string object at all, PyString_AsString() returns NULL and raises TypeError.

char** **PyString_AS_STRING**(PyObject *string)
Macro form of PyString_AsString() but without error checking. Only string objects are supported; no Unicode objects should be passed.

int **PyString_AsStringAndSize**(PyObject *obj, char **buffer, int *length)
Return a NUL-terminated representation of the contents of the object obj through the output variables buffer and length.

The function accepts both string and Unicode objects as input. For Unicode objects it returns the default encoded version of the object. If length is NULL, the resulting buffer may not contain NULL characters; if it does, the function returns -1 and a TypeError is raised.

The buffer refers to an internal string buffer of obj, not a copy. The data must not be modified in any way, unless the string was just created using PyString_FromStringAndSize(NULL, size). It must not be deallocated. If string is a Unicode object, this function computes the default encoding of string and operates on that. If string is not a string object at all, PyString_AsStringAndSize() returns -1 and
raises TypeError.

```c
void PyString_Concat(PyObject **string, PyObject *newpart)
Create a new string object in *string containing the contents of newpart appended to string; the caller will
own the new reference. The reference to the old value of string will be stolen. If the new string cannot be
created, the old reference to string will still be discarded and the value of *string will be set to NULL; the
appropriate exception will be set.
```  

```c
void PyString_ConcatAndDel(PyObject **string, PyObject *newpart)
Create a new string object in *string containing the contents of newpart appended to string. This version
decrements the reference count of newpart.
```  

```c
int _PyString_Resize(PyObject **string, int newsize)
A way to resize a string object even though it is “immutable”. Only use this to build up a brand new string
object; don’t use this if the string may already be known in other parts of the code. It is an error to call this
function if the refcount on the input string object is not one. Pass the address of an existing string object
as an lvalue (it may be written into), and the new size desired. On success, *string holds the resized string
object and 0 is returned; the address in *string may differ from its input value. If the reallocation fails, the
original string object at *string is deallocated, *string is set to NULL, a memory exception is set, and −1 is
returned.
```  

```c
PyObject* PyString_Format(PyObject *format, PyObject *args)
Return a new string object from format and args. Analogous to format % args. The args argument must
be a tuple.
```  

```c
void PyString_InternInPlace(PyObject **string)
Intern the argument *string in place. The argument must be the address of a pointer variable pointing to
a Python string object. If there is an existing interned string that is the same as *string, it sets *string to
it (decrementing the reference count of the old string object and incrementing the reference count of
the interned string object), otherwise it leaves *string alone and interns it (incrementing its reference count).
(Clarification: even though there is a lot of talk about reference counts, think of this function as reference-
count-neutral; you own the object after the call if and only if you owned it before the call.)
```  

```c
PyObject* PyString_InternFromString(const char *v)
A combination of PyString_FROM_STRING() and PyString_InternInPlace(), returning either
a new string object that has been interned, or a new (“owned”) reference to an earlier interned string object
with the same value.
```  

```c
PyObject* PyString.Decode(const char *s, int size, const char *encoding, const char *errors)
Create an object by decoding size bytes of the encoded buffer s using the codec registered for encoding.
encoding and errors have the same meaning as the parameters of the same name in the unicode() built-
in function. The codec to be used is looked up using the Python codec registry. Return NULL if an exception
was raised by the codec.
```  

```c
PyObject* PyString.AsDecodedObject(PyObject *str, const char *encoding, const char *errors)
Decode a string object by passing it to the codec registered for encoding and return the result as Python
object. encoding and errors have the same meaning as the parameters of the same name in the string
encode() method. The codec to be used is looked up using the Python codec registry. Return NULL if an
exception was raised by the codec.
```  

```c
PyObject* PyString.Encode(const char *s, int size, const char *encoding, const char *errors)
Encode the char buffer of the given size by passing it to the codec registered for encoding and return a
Python object. encoding and errors have the same meaning as the parameters of the same name in the string
encode() method. The codec to be used is looked up using the Python codec registry. Return NULL if an
exception was raised by the codec.
```  

```c
PyObject* PyString.AsEncodedObject(PyObject *str, const char *encoding, const char *errors)
```
Encode a string object using the codec registered for encoding and return the result as Python object. encoding and errors have the same meaning as the parameters of the same name in the string encode() method. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.

7.3.2 Unicode Objects

These are the basic Unicode object types used for the Unicode implementation in Python:

**Py_UNICODE**

This type represents the storage type which is used by Python internally as basis for holding Unicode ordinals. Python’s default builds use a 16-bit type for Py_UNICODE and store Unicode values internally as UCS2. It is also possible to build a UCS4 version of Python (most recent Linux distributions come with UCS4 builds of Python). These builds then use a 32-bit type for Py_UNICODE and store Unicode data internally as UCS4. On platforms where wchar_t is available and compatible with the chosen Python Unicode build variant, Py_UNICODE is a typedef alias for wchar_t to enhance native platform compatibility. On all other platforms, Py_UNICODE is a typedef alias for either unsigned short (UCS2) or unsigned long (UCS4).

Note that UCS2 and UCS4 Python builds are not binary compatible. Please keep this in mind when writing extensions or interfaces.

**PyUnicodeObject**

This subtype of PyObject represents a Python Unicode object.

**PyTypeObject PyUnicode_Type**

This instance of PyTypeObject represents the Python Unicode type.

The following APIs are really C macros and can be used to do fast checks and to access internal read-only data of Unicode objects:

- int PyUnicode_Check(PyObject *o)
  Return true if the object o is a Unicode object or an instance of a Unicode subtype. Changed in version 2.2: Allowed subtypes to be accepted.

- int PyUnicode_CheckExact(PyObject *o)
  Return true if the object o is a Unicode object, but not an instance of a subtype. New in version 2.2.

- int PyUnicode_GET_SIZE(PyObject *o)
  Return the size of the object. o has to be a PyUnicodeObject (not checked).

- int PyUnicode_GET_DATA_SIZE(PyObject *o)
  Return the size of the object’s internal buffer in bytes. o has to be a PyUnicodeObject (not checked).

- Py_UNICODE* PyUnicode_AS_UNICODE(PyObject *o)
  Return a pointer to the internal Py_UNICODE buffer of the object. o has to be a PyUnicodeObject (not checked).

- const char* PyUnicode_AS_DATA(PyObject *o)
  Return a pointer to the internal buffer of the object. o has to be a PyUnicodeObject (not checked).

Unicode provides many different character properties. The most often needed ones are available through these macros which are mapped to C functions depending on the Python configuration:

- int Py_UNICODE_ISSPACE(Py_UNICODE ch)
  Return 1 or 0 depending on whether ch is a whitespace character.

- int Py_UNICODE_ISLOWER(Py_UNICODE ch)
  Return 1 or 0 depending on whether ch is a lowercase character.

- int Py_UNICODE_ISUPPER(Py_UNICODE ch)
  Return 1 or 0 depending on whether ch is an uppercase character.

- int Py_UNICODEIRSTITLE(Py_UNICODE ch)
  Return 1 or 0 depending on whether ch is a titlecase character.
int Py_UNICODE_ISLINEBREAK(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a linebreak character.

int Py_UNICODE_ISDECIMAL(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a decimal character.

int Py_UNICODE_ISDIGIT(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a digit character.

int Py_UNICODE_ISNUMERIC(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is a numeric character.

int Py_UNICODE_ISALPHA(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is an alphabetic character.

int Py_UNICODE_ISALNUM(Py_UNICODE ch)
    Return 1 or 0 depending on whether ch is an alphanumeric character.

These APIs can be used for fast direct character conversions:

Py_UNICODE Py_UNICODE_TOLOWER(Py_UNICODE ch)
    Return the character ch converted to lower case.

Py_UNICODE Py_UNICODE_TOUNPER(Py_UNICODE ch)
    Return the character ch converted to upper case.

Py_UNICODE Py_UNICODE_TOTITLE(Py_UNICODE ch)
    Return the character ch converted to title case.

int Py_UNICODE_TODECIMAL(Py_UNICODE ch)
    Return the character ch converted to a decimal positive integer. Return -1 if this is not possible. This macro does not raise exceptions.

int Py_UNICODE_TODIGIT(Py_UNICODE ch)
    Return the character ch converted to a single digit integer. Return -1 if this is not possible. This macro does not raise exceptions.

double Py_UNICODE_TONUMERIC(Py_UNICODE ch)
    Return the character ch converted to a (positive) double. Return -1.0 if this is not possible. This macro does not raise exceptions.

To create Unicode objects and access their basic sequence properties, use these APIs:

PyObject* PyUnicode_FromUnicode(const Py_UNICODE *u, int size)
    Create a Unicode Object from the Py_UNICODE buffer u of the given size. u may be NULL which causes the contents to be undefined. It is the user’s responsibility to fill in the needed data. The buffer is copied into the new object. If the buffer is not NULL, the return value might be a shared object. Therefore, modification of the resulting Unicode object is only allowed when u is NULL.

Py_UNICODE* PyUnicode_AsUnicode(PyObject *unicode)
    Return a read-only pointer to the Unicode object’s internal Py_UNICODE buffer, NULL if unicode is not a Unicode object.

int PyUnicode_GetSize(PyObject *unicode)
    Return the length of the Unicode object.

PyObject* PyUnicode_FromEncodedObject(PyObject *obj, const char *encoding, const char *errors)
    Coerce an encoded object obj to an Unicode object and return a reference with incremented refcount.

    Coercion is done in the following way:

1. Unicode objects are passed back as-is with incremented refcount. Note: These cannot be decoded; passing a non-NULL value for encoding will result in a TypeError.
2. String and other char buffer compatible objects are decoded according to the given encoding and using the error handling defined by errors. Both can be NULL to have the interface use the default values
3. All other objects cause an exception.

The API returns NULL if there was an error. The caller is responsible for decref’ing the returned objects.

```c
PyObject* PyUnicode_FromObject(PyObject *obj)
```

Shortcut for `PyUnicode_FromEncodedObject(obj, NULL, "strict")` which is used throughout the interpreter whenever coercion to Unicode is needed.

If the platform supports wchar_t and provides a header file wchar.h, Python can interface directly to this type using the following functions. Support is optimized if Python’s own `Py_UNICODE` type is identical to the system’s wchar_t.

```c
PyObject* PyUnicode_FromWideChar(const wchar_t *w, int size)
Create a Unicode object from the wchar_t buffer w of the given size. Return NULL on failure.
```

```c
int PyUnicode_AsWideChar(PyUnicodeObject *unicode, wchar_t *w, int size)
Copy the Unicode object contents into the wchar_t buffer w. At most size wchar_t characters are copied (excluding a possibly trailing 0-termination character). Return the number of wchar_t characters copied or -1 in case of an error. Note that the resulting wchar_t string may or may not be 0-terminated. It is the responsibility of the caller to make sure that the wchar_t string is 0-terminated in case this is required by the application.
```

## Built-in Codecs

Python provides a set of built-in codecs which are written in C for speed. All of these codecs are directly usable via the following functions.

Many of the following APIs take two arguments encoding and errors. These parameters encoding and errors have the same semantics as the ones of the builtin unicode() Unicode object constructor.

Setting encoding to NULL causes the default encoding to be used which is ASCII. The file system calls should use `Py_FileSystemDefaultEncoding` as the encoding for file names. This variable should be treated as read-only: On some systems, it will be a pointer to a static string, on others, it will change at run-time (such as when the application invokes setlocale).

Error handling is set by errors which may also be set to NULL meaning to use the default handling defined for the codec. Default error handling for all built-in codecs is "strict" (ValueError is raised).

The codecs all use a similar interface. Only deviation from the following generic ones are documented for simplicity.

These are the generic codec APIs:

```c
PyObject* PyUnicode_Decode(const char *s, int size, const char *encoding, const char *errors)
Create a Unicode object by decoding size bytes of the encoded string s. encoding and errors have the same meaning as the parameters of the same name in the unicode() builtin function. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.
```

```c
PyObject* PyUnicode_Encode(const Py_UNICODE *s, int size, const char *encoding, const char *errors)
Encode the Py_UNICODE buffer of the given size and return a Python string object. encoding and errors have the same meaning as the parameters of the same name in the Unicode encode() method. The codec to be used is looked up using the Python codec registry. Return NULL if an exception was raised by the codec.
```

```c
PyObject* PyUnicode_AsEncodedString(PyObject *unicode, const char *encoding, const char *errors)
Encode a Unicode object and return the result as Python string object. encoding and errors have the same
These are the UTF-8 codec APIs:

```c
PyObject* PyUnicode_DecodeUTF8 (const char *s, int size, const char *errors)
Create a Unicode object by decoding size bytes of the UTF-8 encoded string s. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_DecodeUTF8Stateful (const char *s, int size, const char *errors, int *consumed)
If consumed is NULL, behave like PyUnicode_DecodeUTF8(). If consumed is not NULL, trailing incomplete UTF-8 byte sequences will not be treated as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in consumed. New in version 2.4.

PyObject* PyUnicode_EncodeUTF8 (const Py_UNICODE *s, int size, const char *errors)
Encode the Py_UNICODE buffer of the given size using UTF-8 and return a Python string object. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_AsUTF8String (PyObject *unicode)
Encode a Unicode objects using UTF-8 and return the result as Python string object. Error handling is “strict”. Return NULL if an exception was raised by the codec.
```

These are the UTF-16 codec APIs:

```c
PyObject* PyUnicode_DecodeUTF16 (const char *s, int size, const char *errors, int *byteorder)
Decode length bytes from a UTF-16 encoded buffer string and return the corresponding Unicode object. errors (if non-NULL) defines the error handling. It defaults to “strict”. If byteorder is non-NULL, the decoder starts decoding using the given byte order:

*byteorder == -1: little endian
*byteorder == 0: native order
*byteorder == 1: big endian

and then switches according to all byte order marks (BOM) it finds in the input data. BOMs are not copied into the resulting Unicode string. After completion, *byteorder is set to the current byte order at the end of input data.

If byteorder is NULL, the codec starts in native order mode.
Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_DecodeUTF16Stateful (const char *s, int size, const char *errors, int *byteorder, int *consumed)
If consumed is NULL, behave like PyUnicode_DecodeUTF16(). If consumed is not NULL, PyUnicode_DecodeUTF16Stateful() will not treat trailing incomplete UTF-16 byte sequences (i.e. an odd number of bytes or a split surrogate pair) as an error. Those bytes will not be decoded and the number of bytes that have been decoded will be stored in consumed. New in version 2.4.

PyObject* PyUnicode_EncodeUTF16 (const Py_UNICODE *s, int size, const char *errors, int byteorder)
Return a Python string object holding the UTF-16 encoded value of the Unicode data in s. If byteorder is not 0, output is written according to the following byte order:

byteorder == -1: little endian
byteorder == 0: native byte order (writes a BOM mark)
byteorder == 1: big endian

If byteorder is 0, the output string will always start with the Unicode BOM mark (U+FEFF). In the other two modes, no BOM mark is prepended.
If `Py_UNICODE_WIDE` is defined, a single `Py_UNICODE` value may get represented as a surrogate pair. If it is not defined, each `Py_UNICODE` values is interpreted as an UCS-2 character.

Return NULL if an exception was raised by the codec.

```c
PyObject* PyUnicode_AsUTF16String(PyObject *unicode)
Return a Python string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is “strict”. Return NULL if an exception was raised by the codec.
```

These are the “Unicode Escape” codec APIs:

```c
PyObject* PyUnicode_DecodeUnicodeEscape(const char *s, int size, const char *errors)
Create a Unicode object by decoding size bytes of the Unicode-Escape encoded string s. Return NULL if an exception was raised by the codec.
```

```c
PyObject* PyUnicode_EncodeUnicodeEscape(const Py_UNICODE *s, int size, const char *errors)
Encode the Py_UNICODE buffer of the given size using Unicode-Escape and return a Python string object. Return NULL if an exception was raised by the codec.
```

```c
PyObject* PyUnicode_AsUnicodeEscapeString(PyObject *unicode)
Encode a Unicode objects using Unicode-Escape and return the result as Python string object. Error handling is “strict”. Return NULL if an exception was raised by the codec.
```

These are the “Raw Unicode Escape” codec APIs:

```c
PyObject* PyUnicode_DecodeRawUnicodeEscape(const char *s, int size, const char *errors)
Create a Unicode object by decoding size bytes of the Raw-Unicode-Escape encoded string s. Return NULL if an exception was raised by the codec.
```

```c
PyObject* PyUnicode_EncodeRawUnicodeEscape(const Py_UNICODE *s, int size, const char *errors)
Encode the Py_UNICODE buffer of the given size using Raw-Unicode-Escape and return a Python string object. Return NULL if an exception was raised by the codec.
```

```c
PyObject* PyUnicode_AsRawUnicodeEscapeString(PyObject *unicode)
Encode a Unicode objects using Raw-Unicode-Escape and return the result as Python string object. Error handling is “strict”. Return NULL if an exception was raised by the codec.
```

These are the Latin-1 codec APIs: Latin-1 corresponds to the first 256 Unicode ordinals and only these are accepted by the codecs during encoding.

```c
PyObject* PyUnicode_DecodeLatin1(const char *s, int size, const char *errors)
Create a Unicode object by decoding size bytes of the Latin-1 encoded string s. Return NULL if an exception was raised by the codec.
```

```c
PyObject* PyUnicode_EncodeLatin1(const Py_UNICODE *s, int size, const char *errors)
Encode the Py_UNICODE buffer of the given size using Latin-1 and return a Python string object. Return NULL if an exception was raised by the codec.
```

```c
PyObject* PyUnicode_AsLatin1String(PyObject *unicode)
Encode a Unicode objects using Latin-1 and return the result as Python string object. Error handling is “strict”. Return NULL if an exception was raised by the codec.
```

These are the ASCII codec APIs. Only 7-bit ASCII data is accepted. All other codes generate errors.

```c
PyObject* PyUnicode_DecodeASCII(const char *s, int size, const char *errors)
```
Create a Unicode object by decoding size bytes of the ASCII encoded string s. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_EncodeASCII(const Py_UNICODE *s, int size, const char *errors)
Encode the Py_UNICODE buffer of the given size using ASCII and return a Python string object. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_AsASCIIString(PyObject *unicode)
Encode a Unicode objects using ASCII and return the result as Python string object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

These are the mapping codec APIs:

This codec is special in that it can be used to implement many different codecs (and this is in fact what was done to obtain most of the standard codecs included in the encodings package). The codec uses mapping to encode and decode characters.

Decoding mappings must map single string characters to single Unicode characters, integers (which are then interpreted as Unicode ordinals) or None (meaning “undefined mapping” and causing an error).

Encoding mappings must map single Unicode characters to single string characters, integers (which are then interpreted as Latin-1 ordinals) or None (meaning “undefined mapping” and causing an error).

The mapping objects provided must only support the __getitem__ mapping interface.

If a character lookup fails with a LookupError, the character is copied as-is meaning that its ordinal value will be interpreted as Unicode or Latin-1 ordinal resp. Because of this, mappings only need to contain those mappings which map characters to different code points.

PyObject* PyUnicode_DecodeCharmap(char *s, int size, PyObject *mapping, const char *errors)
Create a Unicode object by decoding size bytes of the encoded string s using the given mapping object. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_EncodeCharmap(const Py_UNICODE *s, int size, PyObject *mapping, const char *errors)
Encode the Py_UNICODE buffer of the given size using the given mapping object and return a Python string object. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_AsCharmapString(PyObject *unicode, PyObject *mapping)
Encode a Unicode objects using the given mapping object and return the result as Python string object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

The following codec API is special in that maps Unicode to Unicode.

PyObject* PyUnicode_TranslateCharmap(const Py_UNICODE *s, int size, PyObject *table, const char *errors)
Translate a Py_UNICODE buffer of the given length by applying a character mapping table to it and return the resulting Unicode object. Return NULL when an exception was raised by the codec.

The mapping table must map Unicode ordinal integers to Unicode ordinal integers or None (causing deletion of the character).

Mapping tables need only provide the method __getitem__() interface; dictionaries and sequences work well. Unmapped character ordinals (ones which cause a LookupError) are left untouched and are copied as-is.

These are the MBCS codec APIs. They are currently only available on Windows and use the Win32 MBCS converters to implement the conversions. Note that MBCS (or DBCS) is a class of encodings, not just one. The target encoding is defined by the user settings on the machine running the codec.

PyObject* PyUnicode_DecodeMBCS(char *s, int size, const char *errors)
Create a Unicode object by decoding size bytes of the MBCS encoded string s. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_EncodeMBCS (const Py_UNICODE *s, int size, const char *errors)
Encode the Py_UNICODE buffer of the given size using MBCS and return a Python string object. Return NULL if an exception was raised by the codec.

PyObject* PyUnicode_AsMBCSString (PyObject *unicode)
Encode a Unicode objects using MBCS and return the result as Python string object. Error handling is “strict”. Return NULL if an exception was raised by the codec.

Methods and Slot Functions

The following APIs are capable of handling Unicode objects and strings on input (we refer to them as strings in the descriptions) and return Unicode objects or integers as appropriate.
They all return NULL or −1 if an exception occurs.

PyObject* PyUnicode_Concat (PyObject *left, PyObject *right)
Concat two strings giving a new Unicode string.

PyObject* PyUnicode_Split (PyObject *s, PyObject *sep, int maxsplit)
Split a string giving a list of Unicode strings. If sep is NULL, splitting will be done at all whitespace substrings. Otherwise, splits occur at the given separator. At most maxsplit splits will be done. If negative, no limit is set. Separators are not included in the resulting list.

PyObject* PyUnicode_Splitlines (PyObject *s, int keepend)
Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. If keepend is 0, the Line break characters are not included in the resulting strings.

PyObject* PyUnicode_Translate (PyObject *str, PyObject *table, const char *errors)
Translate a string by applying a character mapping table to it and return the resulting Unicode object.
The mapping table must map Unicode ordinal integers to Unicode ordinal integers or None (causing deletion of the character).
Mapping tables need only provide the __getitem__() interface; dictionaries and sequences work well.
Unmapped character ordinals (ones which cause a LookupError) are left untouched and are copied as-is.
errors has the usual meaning for codecs. It may be NULL which indicates to use the default error handling.

PyObject* PyUnicode_Join (PyObject *separator, PyObject *seq)
Join a sequence of strings using the given separator and return the resulting Unicode string.

int PyUnicode_Tailmatch (PyObject *str, PyObject *substr, int start, int end, int direction)
Return 1 if substr matches str[start:end] at the given tail end (direction == -1 means to do a prefix match, direction == 1 a suffix match), 0 otherwise. Return −1 if an error occurred.

int PyUnicode_Find (PyObject *str, PyObject *substr, int start, int end, int direction)
Return the first position of substr in str[start:end] using the given direction (direction == 1 means to do a forward search, direction == -1 a backward search). The return value is the index of the first match; a value of −1 indicates that no match was found, and −2 indicates that an error occurred and an exception has been set.

int PyUnicode_Count (PyObject *str, PyObject *substr, int start, int end)
Return the number of non-overlapping occurrences of substr in str[start:end]. Return −1 if an error occurred.

7.3. Sequence Objects 57
PyObject* PyUnicode_Replace(PyObject *str, PyObject *substr, PyObject *replstr, int maxcount)

Replace at most maxcount occurrences of substr in str with replstr and return the resulting Unicode object. maxcount == -1 means replace all occurrences.

int PyUnicode_Compare(PyObject *left, PyObject *right)

Compare two strings and return -1, 0, 1 for less than, equal, and greater than, respectively.

PyObject* PyUnicode_Format(PyObject *format, PyObject *args)

Return a new string object from format and args; this is analogous to format % args. The args argument must be a tuple.

int PyUnicode_Contains(PyObject *container, PyObject *element)

Check whether element is contained in container and return true or false accordingly.
element has to coerce to a one element Unicode string. -1 is returned if there was an error.

7.3.3 Buffer Objects

Python objects implemented in C can export a group of functions called the “buffer interface.” These functions can be used by an object to expose its data in a raw, byte-oriented format. Clients of the object can use the buffer interface to access the object data directly, without needing to copy it first.

Two examples of objects that support the buffer interface are strings and arrays. The string object exposes the character contents in the buffer interface’s byte-oriented form. An array can also expose its contents, but it should be noted that array elements may be multi-byte values.

An example user of the buffer interface is the file object’s write() method. Any object that can export a series of bytes through the buffer interface can be written to a file. There are a number of format codes to PyArg_ParseTuple() that operate against an object’s buffer interface, returning data from the target object.

More information on the buffer interface is provided in the section “Buffer Object Structures” (section 10.7), under the description for PyBufferProcs.

A “buffer object” is defined in the ‘bufferobject.h’ header (included by ‘Python.h’). These objects look very similar to string objects at the Python programming level: they support slicing, indexing, concatenation, and some other standard string operations. However, their data can come from one of two sources: from a block of memory, or from another object which exports the buffer interface.

Buffer objects are useful as a way to expose the data from another object’s buffer interface to the Python programmer. They can also be used as a zero-copy slicing mechanism. Using their ability to reference a block of memory, it is possible to expose any data to the Python programmer quite easily. The memory could be a large, constant array in a C extension, it could be a raw block of memory for manipulation before passing to an operating system library, or it could be used to pass around structured data in its native, in-memory format.

PyBufferObject
This subtype of PyObject represents a buffer object.

PyTypeObject PyBuffer_Type
The instance of PyTypeObject which represents the Python buffer type; it is the same object as types.BufferType in the Python layer.

int Py_END_OF_BUFFER
This constant may be passed as the size parameter to PyBuffer_FromObject() or PyBuffer_FromReadWriteObject(). It indicates that the new PyBufferObject should refer to base object from the specified offset to the end of its exported buffer. Using this enables the caller to avoid querying the base object for its length.

int PyBuffer_Check(PyObject *p)
Return true if the argument has type PyBuffer_Type.

PyObject* PyBuffer_FromObject(PyObject *base, int offset, int size)
Return a new read-only buffer object. This raises TypeError if base doesn’t support the read-only buffer
protocol or doesn’t provide exactly one buffer segment, or it raises ValueError if offset is less than zero.
The buffer will hold a reference to the base object, and the buffer’s contents will refer to the base object’s buffer interface, starting as position offset and extending for size bytes. If size is Py_END_OF_BUFFER, then the new buffer’s contents extend to the length of the base object’s exported buffer data.

PyObject* PyBuffer_FromReadWriteObject (PyObject *base, int offset, int size)
Return a new writable buffer object. Parameters and exceptions are similar to those for PyBuffer_FromObject(). If the base object does not export the writeable buffer protocol, then TypeError is raised.

PyObject* PyBuffer_FromMemory (void *ptr, int size)
Return a new read-only buffer object that reads from a specified location in memory, with a specified size. The caller is responsible for ensuring that the memory buffer, passed in as ptr, is not deallocated while the returned buffer object exists. Raises ValueError if size is less than zero. Note that Py_END_OF_BUFFER may not be passed for the size parameter; ValueError will be raised in that case.

PyObject* PyBuffer_FromReadWriteMemory (void *ptr, int size)
Similar to PyBuffer_FromMemory(), but the returned buffer is writable.

PyObject* PyBuffer_New (int size)
Return a new writable buffer object that maintains its own memory buffer of size bytes. ValueError is returned if size is not zero or positive. Note that the memory buffer (as returned by PyObject_AsWriteBuffer()) is not specifically aligned.

7.3.4 Tuple Objects

PyTupleObject
This subtype of PyObject represents a Python tuple object.

PyTypeObject PyTuple_Type
This instance of PyTypeObject represents the Python tuple type; it is the same object as types.TupleType in the Python layer.

int PyTuple_Check (PyObject *p)
Return true if p is a tuple object or an instance of a subtype of the tuple type. Changed in version 2.2: Allowed subtypes to be accepted.

int PyTuple_CheckExact (PyObject *p)
Return true if p is a tuple object, but not an instance of a subtype of the tuple type. New in version 2.2.

PyObject* PyTuple_New (int len)
Return a new tuple object of size len, or NULL on failure.

PyObject* PyTuple_Pack (int n, ...)
Return a new tuple object of size n, or NULL on failure. The tuple values are initialized to the subsequent n C arguments pointing to Python objects. ‘PyTuple_Pack(2, a, b)’ is equivalent to ‘Py_BuildValue("(OO)", a, b)’. New in version 2.4.

int PyTuple_Size (PyObject *p)
Take a pointer to a tuple object, and return the size of that tuple.

int PyTuple_GET_SIZE (PyObject *p)
Return the size of the tuple p, which must be non-NULL and point to a tuple; no error checking is performed.

PyObject* PyTuple_GetItem (PyObject *p, int pos)
Return value: Borrowed reference.
Return the object at position pos in the tuple pointed to by p. If pos is out of bounds, return NULL and sets an IndexError exception.
PyObject* PyTuple_GET_ITEM(PyObject *p, int pos)
Retum value: Borrowed reference.
Like PyTuple_GetItem(), but does no checking of its arguments.

PyObject* PyTuple_GetSlice (PyObject *p, int low, int high)
Take a slice of the tuple pointed to by p from low to high and return it as a new tuple.

int PyTuple_SetItem (PyObject *p, int pos, PyObject *o)
Insert a reference to object o at position pos of the tuple pointed to by p. Return 0 on success. Note: This function "steals" a reference to o.

void PyTuple_SET_ITEM (PyObject *p, int pos, PyObject *o)
Like PyTuple_SetItem(), but does no error checking, and should only be used to fill in brand new tuples. Note: This function "steals" a reference to o.

int _PyTuple_Resize (PyObject **p, int newsize)
Can be used to resize a tuple. newsize will be the new length of the tuple. Because tuples are supposed to be immutable, this should only be used if there is only one reference to the object. Do not use this if the tuple may already be known to some other part of the code. The tuple will always grow or shrink at the end. Think of this as destroying the old tuple and creating a new one, only more efficiently. Returns 0 on success. Client code should never assume that the resultant value of *p will be the same as before calling this function. If the object referenced by *p is replaced, the original *p is destroyed. On failure, returns -1 and sets *p to NULL, and raises MemoryError or SystemError. Changed in version 2.2: Removed unused third parameter, last_is_sticky.

7.3.5 List Objects

PyListObject
This subtype of PyObject represents a Python list object.

PyTypeObject PyList_Type
This instance of PyTypeObject represents the Python list type. This is the same object as types.ListType.

int PyList_Check (PyObject *p)
Return true if p is a list object or an instance of a subtype of the list type. Changed in version 2.2: Allowed subtypes to be accepted.

int PyList_CheckExact (PyObject *p)
Return true if p is a list object, but not an instance of a subtype of the list type. New in version 2.2.

PyObject* PyList_New (int len)
Return a new list of length len on success, or NULL on failure.

int PyList_Size (PyObject *list)
Return the length of the list object in list; this is equivalent to 'len (list)' on a list object.

int PyList_GET_SIZE (PyObject *list)
Macro form of PyList_Size() without error checking.

PyObject* PyList_GetItem (PyObject *list, int index)
Return value: Borrowed reference.
Return the object at position pos in the list pointed to by p. If pos is out of bounds, return NULL and set an IndexError exception.

PyObject* PyList_GET_ITEM (PyObject *list, int i)
Return value: Borrowed reference.
Macro form of PyList_GetItem() without error checking.

int PyList_SetItem (PyObject *list, int index, PyObject *item)
Set the item at index index in list to item. Return 0 on success or -1 on failure. Note: This function "steals" a reference to item and discards a reference to an item already in the list at the affected position.
void PyList_SET_ITEM(PyObject *list, int i, PyObject *o)

Macro form of PyList_SetItem() without error checking. This is normally only used to fill in new lists where there is no previous content. **Note:** This function "steals" a reference to item, and, unlike PyList_SetItem(), does not discard a reference to any item that it being replaced; any reference in list at position i will be leaked.

int PyList_Insert(PyObject *list, int index, PyObject *item)

Insert the item item into list in front of index index. Return 0 if successful; return -1 and set an exception if unsuccessful. Analogous to list.insert(index, item).

int PyList_Append(PyObject *list, PyObject *item)

Append the object item at the end of list list. Return 0 if successful; return -1 and set an exception if unsuccessful. Analogous to list.append(item).

PyObject* PyList_GetSlice(PyObject *list, int low, int high)


Return a list of the objects in list containing the objects between low and high. Return NULL and set an exception if unsuccessful. Analogous to list[low:high].

int PyList_SetSlice(PyObject *list, int low, int high, PyObject *itemlist)

Set the slice of list between low and high to the contents of itemlist. Analogous to list[low:high] = itemlist. The itemlist may be NULL, indicating the assignment of an empty list (slice deletion). Return 0 on success, -1 on failure.

int PyList_Sort(PyObject *list)

Sort the items of list in place. Return 0 on success, -1 on failure. This is equivalent to ‘list.sort()’.

int PyList_ReversPyList_Revers迚e(PyObject *list)

Reverse the items of list in place. Return 0 on success, -1 on failure. This is the equivalent of ‘list.reverse()’.

PyObject* PyList_AsTuple(PyObject *list)


Return a new tuple object containing the contents of list; equivalent to ‘tuple(list)’.

7.4 Mapping Objects

7.4.1 Dictionary Objects

PyDictObject

This subtype of PyObject represents a Python dictionary object.

PyTypeObject PyDict_Type

This instance of PyTypeObject represents the Python dictionary type. This is exposed to Python programs as types.DictType and types.DictionaryType.

int PyDict_Check(PyObject *p)

Return true if p is a dict object or an instance of a subtype of the dict type. Changed in version 2.2: Allowed subtypes to be accepted.

int PyDict_CheckExact(PyObject *p)

Return true if p is a dict object, but not an instance of a subtype of the dict type. New in version 2.4.

PyObject* PyDict_New()


Return a new empty dictionary, or NULL on failure.

PyObject* PyDictProxy_New(PyObject *dict)


Return a proxy object for a mapping which enforces read-only behavior. This is normally used to create a proxy to prevent modification of the dictionary for non-dynamic class types. New in version 2.2.

void PyDict_Clear(PyObject *p)
Empty an existing dictionary of all key-value pairs.

```c
int PyDict_Contains (PyObject *p, PyObject *key)
```
Determine if dictionary `p` contains `key`. If an item in `p` is matches `key`, return 1, otherwise return 0. On error, return -1. This is equivalent to the Python expression ‘key in p’. New in version 2.4.

```c
PyObject* PyDict_Copy (PyObject *p)
```
Return a new dictionary that contains the same key-value pairs as `p`. New in version 1.6.

```c
int PyDict_SetItem (PyObject *p, PyObject *key, PyObject *val)
```
Insert `value` into the dictionary `p` with a key of `key`. `key` must be hashable; if it isn’t, `TypeError` will be raised. Return 0 on success or -1 on failure.

```c
int PyDict_SetItemString (PyObject *p, char *key, PyObject *val)
```
Insert `value` into the dictionary `p` using `key` as a key. `key` should be a `char*`. The key object is created using `PyString_FromString(key)`. Return 0 on success or -1 on failure.

```c
int PyDict_DelItem (PyObject *p, PyObject *key)
```
Remove the entry in dictionary `p` with key `key`. `key` must be hashable; if it isn’t, `TypeError` is raised. Return 0 on success or -1 on failure.

```c
int PyDict_DelItemString (PyObject *p, char *key)
```
Remove the entry in dictionary `p` which has a key specified by the string `key`. Return 0 on success or -1 on failure.

```c
PyObject* PyDict_GetItem (PyObject *p, PyObject *key)
```
Return the object from dictionary `p` which has a key `key`. Return NULL if the key `key` is not present, but without setting an exception.

```c
PyObject* PyDict_GetItemString (PyObject *p, char *key)
```
This is the same as `PyDict_GetItem()`, but `key` is specified as a `char*`, rather than a `PyObject*`.

```c
PyObject* PyDict_Items (PyObject *p)
```
Return a `PyListObject` containing all the items from the dictionary, as in the dictionary method `items()` (see the Python Library Reference).

```c
PyObject* PyDict_Keys (PyObject *p)
```
Return a `PyListObject` containing all the keys from the dictionary, as in the dictionary method `keys()` (see the Python Library Reference).

```c
PyObject* PyDict_Values (PyObject *p)
```
Return a `PyListObject` containing all the values from the dictionary `p`, as in the dictionary method `values()` (see the Python Library Reference).

```c
int PyDict_Size (PyObject *p)
```
Return the number of items in the dictionary. This is equivalent to ‘len (p) ’ on a dictionary.

```c
int PyDict_Next (PyObject *p, int *ppos, PyObject **pkey, PyObject **pvalue)
```
Iterate over all key-value pairs in the dictionary `p`. The `int` referred to by `ppos` must be initialized to 0 prior to the first call to this function to start the iteration; the function returns true for each pair in the dictionary, and false once all pairs have been reported. The parameters `pkey` and `pvalue` should either point to `PyObject*` variables that will be filled in with each key and value, respectively, or may be NULL. Any references returned through them are borrowed. `ppos` should not be altered during iteration. Its value represents offsets within the internal dictionary structure, and since the structure is sparse, the offsets are not consecutive.

For example:
PyObject *key, *value;
int pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    /* do something interesting with the values... */
    ...
}

The dictionary $p$ should not be mutated during iteration. It is safe (since Python 2.1) to modify the values of
the keys as you iterate over the dictionary, but only so long as the set of keys does not change. For example:

PyObject *key, *value;
int pos = 0;

while (PyDict_Next(self->dict, &pos, &key, &value)) {
    int i = PyInt_AS_LONG(value) + 1;
    PyObject *o = PyInt_FromLong(i);
    if (o == NULL)
        return -1;
    if (PyDict_SetItem(self->dict, key, o) < 0) {
        Py_DECREF(o);
        return -1;
    }
    Py_DECREF(o);
}

int PyDict_Merge(PyObject *a, PyObject *b, int override)
Iterate over mapping object $b$ adding key-value pairs to dictionary $a$. $b$ may be a dictionary, or any object
supporting PyMapping_Keys() and PyObject_GetItem(). If override is true, existing pairs in $a$
will be replaced if a matching key is found in $b$, otherwise pairs will only be added if there is not a matching
key in $a$. Return 0 on success or -1 if an exception was raised. New in version 2.2.

int PyDict_Update(PyObject *a, PyObject *b)
This is the same as PyDict_Merge($a$, $b$, 1) in C, or $a$.update($b$) in Python. Return 0 on success
or -1 if an exception was raised. New in version 2.2.

int PyDict_MergeFromSeq2(PyObject *a, PyObject *seq2, int override)
Update or merge into dictionary $a$, from the key-value pairs in seq2. seq2 must be an iterable object produc-
ing iterable objects of length 2, viewed as key-value pairs. In case of duplicate keys, the last wins if override
is true, else the first wins. Return 0 on success or -1 if an exception was raised. Equivalent Python (except
for the return value):

```
def PyDict_MergeFromSeq2(a, seq2, override):
    for key, value in seq2:
        if override or key not in a:
            a[key] = value
```

New in version 2.2.

7.5 Other Objects

7.5.1 File Objects

Python’s built-in file objects are implemented entirely on the FILE* support from the C standard library. This is
an implementation detail and may change in future releases of Python.

PyFileObject
This subtype of PyObject represents a Python file object.
PyTypeObject PyFile_Type
This instance of PyTypeObject represents the Python file type. This is exposed to Python programs as types.FileType.

int PyFile_Check (PyObject *p)
Return true if its argument is a PyFileObject or a subtype of PyFileObject. Changed in version 2.2: Allowed subtypes to be accepted.

int PyFile_CheckExact (PyObject *p)
Return true if its argument is a PyFileObject, but not a subtype of PyFileObject. New in version 2.2.

PyObject* PyFile_FromString (char *filename, char *mode)
On success, return a new file object that is opened on the file given by filename, with a file mode given by mode, where mode has the same semantics as the standard C routine fopen(). On failure, return NULL.

PyObject* PyFile_FromFile (FILE *fp, char *name, char *mode, int (*close)(FILE*))
Create a new PyFileObject from the already-open standard C file pointer, fp. The function close will be called when the file should be closed. Return NULL on failure.

FILE* PyFile_AsFile (PyFileObject *p)
Return the file object associated with p as a FILE*.

PyObject* PyFile_GetLine (PyObject *p, int n)
Equivalent to p.readline([n]), this function reads one line from the object p. p may be a file object or any object with a readline() method. If n is 0, exactly one line is read, regardless of the length of the line. If n is greater than 0, no more than n bytes will be read from the file; a partial line can be returned. In both cases, an empty string is returned if the end of the file is reached immediately. If n is less than 0, however, one line is read regardless of length, but EOFError is raised if the end of the file is reached immediately.

PyObject* PyFile_Name (PyObject *p)
Return value: Borrowed reference.
Return the name of the file specified by p as a string object.

void PyFile_SetBufSize (PyFileObject *p, int n)
Available on systems with setvbuf() only. This should only be called immediately after file object creation.

int PyFile_Encoding (PyFileObject *p, char *enc)
Set the file’s encoding for Unicode output to enc. Return 1 on success and 0 on failure. New in version 2.3.

int PyFile_SoftSpace (PyObject *p, int newflag)
This function exists for internal use by the interpreter. Set the softspace attribute of p to newflag and return the previous value. p does not have to be a file object for this function to work properly; any object is supported (though its only interesting if the softspace attribute can be set). This function clears any errors, and will return 0 as the previous value if the attribute either does not exist or if there were errors in retrieving it. There is no way to detect errors from this function, but doing so should not be needed.

int PyFile_WriteObject (PyObject *obj, PyFileObject *p, int flags)
Write object obj to file object p. The only supported flag for flags is Py_PRINT_RAW; if given, the str() of the object is written instead of the repr(). Return 0 on success or -1 on failure; the appropriate exception will be set.

int PyFile_WriteString (const char *s, PyFileObject *p)
Write string s to file object p. Return 0 on success or -1 on failure; the appropriate exception will be set.

7.5.2 Instance Objects
There are very few functions specific to instance objects.
PyTypeObject  PyInstance_Type  
Type object for class instances.

int  PyInstance_Check (PyObject *obj)  
Return true if obj is an instance.

PyObject*  PyInstance_New (PyObject *class, PyObject *arg, PyObject *kw)  
Create a new instance of a specific class. The parameters arg and kw are used as the positional and keyword parameters to the object’s constructor.

PyObject*  PyInstance_NewRaw (PyObject *class, PyObject *dict)  
Create a new instance of a specific class without calling it’s constructor. class is the class of new object. The dict parameter will be used as the object’s __dict__; if NULL, a new dictionary will be created for the instance.

7.5.3 Method Objects

There are some useful functions that are useful for working with method objects.

PyTypeObject  PyMethod_Type  
This instance of PyTypeObject represents the Python method type. This is exposed to Python programs as types.MethodType.

int  PyMethod_Check (PyObject *o)  
Return true if o is a method object (has type PyMethod_Type). The parameter must not be NULL.

PyObject*  PyMethod_New (PyObject *func, PyObject *self, PyObject *class)  
Return a new method object, with func being any callable object; this is the function that will be called when the method is called. If this method should be bound to an instance, self should be the instance and class should be the class of self, otherwise self should be NULL and class should be the class which provides the unbound method.

PyObject*  PyMethod_Class (PyObject *meth)  
Return value:  Borrowed reference.  
Return the class object from which the method meth was created; if this was created from an instance, it will be the class of the instance.

PyObject*  PyMethod_GET_CLASS (PyObject *meth)  
Return value:  Borrowed reference.  
Macro version of PyMethod_Class() which avoids error checking.

PyObject*  PyMethod_Function (PyObject *meth)  
Return value:  Borrowed reference.  
Return the function object associated with the method meth.

PyObject*  PyMethod_GET_FUNCTION (PyObject *meth)  
Return value:  Borrowed reference.  
Macro version of PyMethod_Function() which avoids error checking.

PyObject*  PyMethod_Self (PyObject *meth)  
Return value:  Borrowed reference.  
Return the instance associated with the method meth if it is bound, otherwise return NULL.

PyObject*  PyMethod_GET_SELF (PyObject *meth)  
Return value:  Borrowed reference.  
Macro version of PyMethod_Self() which avoids error checking.

7.5.4 Module Objects

There are only a few functions special to module objects.
PyTypeObject PyModule_Type
This instance of PyTypeObject represents the Python module type. This is exposed to Python programs as types.ModuleType.

int PyModule_Check (PyObject *p)
Return true if p is a module object, or a subtype of a module object. Changed in version 2.2: Allowed subtypes to be accepted.

int PyModule_CheckExact (PyObject *p)
Return true if p is a module object, but not a subtype of PyModule_Type. New in version 2.2.

PyObject* PyModule_New (char *name)
Return a new module object with the __name__ attribute set to name. Only the module’s __doc__ and __name__ attributes are filled in; the caller is responsible for providing a __file__ attribute.

PyObject* PyModule_GetDict (PyObject *module)
Return value: Borrowed reference.
Return the dictionary object that implements module’s namespace; this object is the same as the __dict__ attribute of the module object. This function never fails. It is recommended extensions use other PyModule_*() and PyObject_*() functions rather than directly manipulate a module’s __dict__.

char* PyModule_GetName (PyObject *module)
Return module’s __name__ value. If the module does not provide one, or if it is not a string, SystemError is raised and NULL is returned.

char* PyModule_GetFilename (PyObject *module)
Return the name of the file from which module was loaded using module’s __file__ attribute. If this is not defined, or if it is not a string, raise SystemError and return NULL.

int PyModule_AddObject (PyObject *module, char *name, PyObject *value)
Add an object to module as name. This is a convenience function which can be used from the module’s initialization function. This steals a reference to value. Return -1 on error, 0 on success. New in version 2.0.

int PyModule_AddIntConstant (PyObject *module, char *name, long value)
Add an integer constant to module as name. This convenience function can be used from the module’s initialization function. Return -1 on error, 0 on success. New in version 2.0.

int PyModule_AddStringConstant (PyObject *module, char *name, char *value)
Add a string constant to module as name. This convenience function can be used from the module’s initialization function. The string value must be null-terminated. Return -1 on error, 0 on success. New in version 2.0.

7.5.5 Iterator Objects

Python provides two general-purpose iterator objects. The first, a sequence iterator, works with an arbitrary sequence supporting the __getitem__() method. The second works with a callable object and a sentinel value, calling the callable for each item in the sequence, and ending the iteration when the sentinel value is returned.

PyTypeObject PySeqIter_Type
Type object for iterator objects returned by PySeqIter_New() and the one-argument form of the iter() built-in function for built-in sequence types. New in version 2.2.

int PySeqIter_Check (op)
Return true if the type of op is PySeqIter_Type. New in version 2.2.

PyObject* PySeqIter_New (PyObject *seq)
Return an iterator that works with a general sequence object, seq. The iteration ends when the sequence raises IndexError for the subscripting operation. New in version 2.2.

PyTypeObject PyCallIter_Type
Type object for iterator objects returned by `PyCallIter_New()` and the two-argument form of the `iter()` built-in function. New in version 2.2.

```c
int PyCallIter_Check (op)
Return true if the type of op is PyCallIter_Type. New in version 2.2.
```

```c
PyObject* PyCallIter_New (PyObject *callable, PyObject *sentinel)
Return a new iterator. The first parameter, callable, can be any Python callable object that can be called with no parameters; each call to it should return the next item in the iteration. When callable returns a value equal to sentinel, the iteration will be terminated. New in version 2.2.
```

### 7.5.6 Descriptor Objects

“Descriptors” are objects that describe some attribute of an object. They are found in the dictionary of type objects.

```c
PyTypeObject PyProperty_Type
The type object for the built-in descriptor types. New in version 2.2.
```

```c
PyObject* PyDescr_NewGetSet (PyTypeObject *type, PyGetSetDef *getset)
New in version 2.2.
```

```c
PyObject* PyDescr_NewMember (PyTypeObject *type, PyMemberDef *meth)
New in version 2.2.
```

```c
PyObject* PyDescr_NewMethod (PyTypeObject *type, PyMethodDef *meth)
New in version 2.2.
```

```c
PyObject* PyDescr_NewWrapper (PyTypeObject *type, struct wrapperbase *wrapper, void *wrapped)
New in version 2.2.
```

```c
PyObject* PyDescr_NewClassMethod (PyTypeObject *type, PyMethodDef *method)
New in version 2.3.
```

```c
int PyDescr_IsData (PyObject *descr)
Return true if the descriptor objects descr describes a data attribute, or false if it describes a method. descr must be a descriptor object; there is no error checking. New in version 2.2.
```

```c
PyObject* PyWrapper_New (PyObject *, PyObject *)
New in version 2.2.
```

### 7.5.7 Slice Objects

```c
PyTypeObject PySlice_Type
The type object for slice objects. This is the same as types.SliceType.
```

```c
int PySlice_Check (PyObject *ob)
Return true if ob is a slice object; ob must not be NULL.
```

```c
PyObject* PySlice_New (PyObject *start, PyObject *stop, PyObject *step)
Return a new slice object with the given values. The start, stop, and step parameters are used as the values of the slice object attributes of the same names. Any of the values may be NULL, in which case the None will be used for the corresponding attribute. Return NULL if the new object could not be allocated.
```

```c
int PySlice_GetIndices (PySliceObject *slice, int length, int *start, int *stop, int *step)
Retrieve the start, stop and step indices from the slice object slice, assuming a sequence of length length. Treats indices greater than length as errors.
```
Returns 0 on success and -1 on error with no exception set (unless one of the indices was not `None` and failed to be converted to an integer, in which case -1 is returned with an exception set).

You probably do not want to use this function. If you want to use slice objects in versions of Python prior to 2.3, you would probably do well to incorporate the source of `PySlice_GetIndicesEx`, suitably renamed, in the source of your extension.

```c
int PySlice_GetIndicesEx(PySliceObject *slice, int length, int *start, int *stop, int *step, int *slice-length)
```

Usable replacement for `PySlice_GetIndices`. Retrieve the start, stop, and step indices from the slice object `slice` assuming a sequence of length `length`, and store the length of the slice in `slicelength`. Out of bounds indices are clipped in a manner consistent with the handling of normal slices.

Returns 0 on success and -1 on error with exception set.

New in version 2.3.

### 7.5.8 Weak Reference Objects

Python supports weak references as first-class objects. There are two specific object types which directly implement weak references. The first is a simple reference object, and the second acts as a proxy for the original object as much as it can.

```c
int PyWeakref_Check(PyObject *ob)
```

Return true if `ob` is either a reference or proxy object. New in version 2.2.

```c
int PyWeakref_CheckRef(PyObject *ob)
```

Return true if `ob` is a reference object. New in version 2.2.

```c
int PyWeakref_CheckProxy(PyObject *ob)
```

Return true if `ob` is a proxy object. New in version 2.2.

```c
PyObject* PyWeakref_NewRef(PyObject *ob, PyObject *callback)
```


Return a weak reference object for the object `ob`. This will always return a new reference, but is not guaranteed to create a new object; an existing reference object may be returned. The second parameter, `callback`, can be a callable object that receives notification when `ob` is garbage collected; it should accept a single parameter, which will be the weak reference object itself. `callback` may also be `None` or `NULL`. If `ob` is not a weakly-referencable object, or if `callback` is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`. New in version 2.2.

```c
PyObject* PyWeakref_NewProxy(PyObject *ob, PyObject *callback)
```


Return a weak reference proxy object for the object `ob`. This will always return a new reference, but is not guaranteed to create a new object; an existing proxy object may be returned. The second parameter, `callback`, can be a callable object that receives notification when `ob` is garbage collected; it should accept a single parameter, which will be the weak reference object itself. `callback` may also be `None` or `NULL`. If `ob` is not a weakly-referencable object, or if `callback` is not callable, `None`, or `NULL`, this will return `NULL` and raise `TypeError`. New in version 2.2.

```c
PyObject* PyWeakref_GetObject(PyObject *ref)
```

Return value: Borrowed reference.

Return the referenced object from a weak reference, `ref`. If the referent is no longer live, returns `None`. New in version 2.2.

```c
PyObject* PyWeakref_GET_OBJECT(PyObject *ref)
```

Return value: Borrowed reference.

Similar to `PyWeakref_GetObject()`, but implemented as a macro that does no error checking. New in version 2.2.
7.5.9 CObjects

Refer to *Extending and Embedding the Python Interpreter*, section 1.12, “Providing a C API for an Extension Module,” for more information on using these objects.

**PyCObject**

This subtype of PyObject represents an opaque value, useful for C extension modules who need to pass an opaque value (as a void* pointer) through Python code to other C code. It is often used to make a C function pointer defined in one module available to other modules, so the regular import mechanism can be used to access C APIs defined in dynamically loaded modules.

```c
int PyCObject_Check(PyObject *p)
Return true if its argument is a PyCObject.

PyObject* PyCObject_FromVoidPtr(void* cobj, void (*destr)(void *))
Create a PyCObject from the void *cobj. The destr function will be called when the object is reclaimed, unless it is NULL.

PyObject* PyCObject_FromVoidPtrAndDesc(void* cobj, void* desc, void (*destr)(void *, void *))
Create a PyCObject from the void *cobj. The destr function will be called when the object is reclaimed. The desc argument can be used to pass extra callback data for the destructor function.

void* PyCObject_AsVoidPtr(PyObject* self)
Return the object void * that the PyCObject self was created with.

void* PyCObject_GetDesc(PyObject* self)
Return the description void * that the PyCObject self was created with.

int PyCObject_SetVoidPtr(PyObject* cell, PyObject *value)
Set the void pointer inside self to value. The PyCObject must not have an associated destructor. Return true on success, false on failure.
```

7.5.10 Cell Objects

“Cell” objects are used to implement variables referenced by multiple scopes. For each such variable, a cell object is created to store the value; the local variables of each stack frame that references the value contains a reference to the cells from outer scopes which also use that variable. When the value is accessed, the value contained in the cell is used instead of the cell object itself. This de-referencing of the cell object requires support from the generated byte-code; these are not automatically de-referenced when accessed. Cell objects are not likely to be useful elsewhere.

**PyCellObject**
The C structure used for cell objects.

**PyTypeObject** PyCell_Type
The type object corresponding to cell objects

```c
int PyCell_Check(ob)
Return true if ob is a cell object; ob must not be NULL.

PyObject* PyCell_New(PyObject *ob)
Create and return a new cell object containing the value ob. The parameter may be NULL.

PyObject* PyCell_Get(PyObject *cell)
Return the contents of the cell cell.

PyObject* PyCell_GET(PyObject *cell)
Return value: Borrowed reference.
Return the contents of the cell cell, but without checking that cell is non-NULL and a cell object.

int PyCell_Set(PyObject *cell, PyObject *value)
```
Set the contents of the cell object `cell` to `value`. This releases the reference to any current content of the cell. `value` may be `NULL`. `cell` must be non-NULL; if it is not a cell object, -1 will be returned. On success, 0 will be returned.

```c
void PyCell_SET (PyObject *cell, PyObject *value)
```

Sets the value of the cell object `cell` to `value`. No reference counts are adjusted, and no checks are made for safety; `cell` must be non-NULL and must be a cell object.

### 7.5.11 Generator Objects

Generator objects are what Python uses to implement generator iterators. They are normally created by iterating over a function that yields values, rather than explicitly calling `PyGen_New`.

- **PyGenObject**
  - The C structure used for generator objects.
- **PyTypeObject PyGen_Type**
  - The type object corresponding to generator objects

```c
int PyGen_Check (PyObject *ob)
```

Return true if `ob` is a generator object; `ob` must not be NULL.

```c
int PyGen_CheckExact (PyObject *ob)
```

Return true if `ob`'s type is `PyGen_Type` is a generator object; `ob` must not be NULL.

```c
PyObject* PyGen_New (PyFrameObject *frame)
```

Create and return a new generator object based on the `frame` object. The parameter must not be NULL.

### 7.5.12 DateTime Objects

Various date and time objects are supplied by the `datetime` module. Before using any of these functions, the header file 'datetime.h' must be included in your source (note that this is not include by 'Python.h'), and macro `PyDateTime_IMPORT()` must be invoked. The macro arranges to put a pointer to a C structure in a static variable `PyDateTimeAPI`, which is used by the following macros.

Type-check macros:

```c
int PyDate_Check (PyObject *ob)
```

Return true if `ob` is of type `PyDateTime_DateType` or a subtype of `PyDateTime_DateType`. `ob` must not be NULL. New in version 2.4.

```c
int PyDate_CheckExact (PyObject *ob)
```

Return true if `ob` is of type `PyDateTime_DateType`. `ob` must not be NULL. New in version 2.4.

```c
int PyDateTime_Check (PyObject *ob)
```

Return true if `ob` is of type `PyDateTime_DateTimeType` or a subtype of `PyDateTime_DateTimeType`. `ob` must not be NULL. New in version 2.4.

```c
int PyDateTime_CheckExact (PyObject *ob)
```

Return true if `ob` is of type `PyDateTime_DateTimeType`. `ob` must not be NULL. New in version 2.4.

```c
int PyTime_Check (PyObject *ob)
```

Return true if `ob` is of type `PyDateTime_TimeType` or a subtype of `PyDateTime_TimeType`. `ob` must not be NULL. New in version 2.4.

```c
int PyTime_CheckExact (PyObject *ob)
```

Return true if `ob` is of type `PyDateTime_TimeType`. `ob` must not be NULL. New in version 2.4.

```c
int PyDelta_Check (PyObject *ob)
```

Return true if `ob` is of type `PyDateTime_DeltaType` or a subtype of `PyDateTime_DeltaType`. `ob` must not be NULL. New in version 2.4.

```c
int PyDelta_CheckExact (PyObject *ob)
```

Return true if `ob` is of type `PyDateTime_DeltaType`. `ob` must not be NULL. New in version 2.4.
int PyTZInfo_Check(PyObject *ob)
    Return true if ob is of type PyDateTime_TZInfoType or a subtype of PyDateTime_TZInfoType. ob must not be NULL. New in version 2.4.

int PyTZInfo_CheckExact(PyObject *ob)
    Return true if ob is of type PyDateTime_TZInfoType. ob must not be NULL. New in version 2.4.

Macros to create objects:

PyObject* PyDate_FromDate(int year, int month, int day)
    Return a datetime.date object with the specified year, month and day. New in version 2.4.

PyObject* PyDateTime_FromDateAndTime(int year, int month, int day, int hour, int minute, int second, int usecond)
    Return a datetime.datetime object with the specified year, month, day, hour, minute, second and microsecond. New in version 2.4.

PyObject* PyTime_FromTime(int hour, int minute, int second, int usecond)
    Return a datetime.time object with the specified hour, minute, second and microsecond. New in version 2.4.

PyObject* PyDelta_FromDSU(int days, int seconds, int useconds)
    Return a datetime.timedelta object representing the given number of days, seconds and microseconds. Normalization is performed so that the resulting number of microseconds and seconds lie in the ranges documented for datetime.timedelta objects. New in version 2.4.

Macros to extract fields from date objects. The argument must be an instance of PyDateTime_Date, including subclasses (such as PyDateTime_DateTime). The argument must not be NULL, and the type is not checked:

int PyDateTime_GET_YEAR(PyDateTime_Date *o)
    Return the year, as a positive int. New in version 2.4.

int PyDateTime_GET_MONTH(PyDateTime_Date *o)
    Return the month, as an int from 1 through 12. New in version 2.4.

int PyDateTime_GET_DAY(PyDateTime_Date *o)
    Return the day, as an int from 1 through 31. New in version 2.4.

Macros to extract fields from datetime objects. The argument must be an instance of PyDateTime_DateTime, including subclasses. The argument must not be NULL, and the type is not checked:

int PyDateTime_DATE_GET_HOUR(PyDateTime_DateTime *o)
    Return the hour, as an int from 0 through 23. New in version 2.4.

int PyDateTime_DATE_GET_MINUTE(PyDateTime_DateTime *o)
    Return the minute, as an int from 0 through 59. New in version 2.4.

int PyDateTime_DATE_GET_SECOND(PyDateTime_DateTime *o)
    Return the second, as an int from 0 through 59. New in version 2.4.

int PyDateTime_DATE_GET_MICROSECOND(PyDateTime_DateTime *o)
    Return the microsecond, as an int from 0 through 999999. New in version 2.4.

Macros to extract fields from time objects. The argument must be an instance of PyDateTime_Time, including subclasses. The argument must not be NULL, and the type is not checked:

int PyDateTime_TIME_GET_HOUR(PyDateTime_Time *o)
    Return the hour, as an int from 0 through 23. New in version 2.4.

int PyDateTime_TIME_GET_MINUTE(PyDateTime_Time *o)
    Return the minute, as an int from 0 through 59. New in version 2.4.

int PyDateTime_TIME_GET_SECOND(PyDateTime_Time *o)
    Return the second, as an int from 0 through 59. New in version 2.4.

int PyDateTime_TIME_GET_MICROSECOND(PyDateTime_Time *o)
    Return the microsecond, as an int from 0 through 999999. New in version 2.4.

Macros for the convenience of modules implementing the DB API:

7.5. Other Objects
PyObject* **PyDateTime_FromTimestamp**(PyObject *args)
Create and return a new `datetime.datetime` object given an argument tuple suitable for passing to `datetime.datetime.fromtimestamp()`. New in version 2.4.

PyObject* **PyDate_FromTimestamp**(PyObject *args)
Create and return a new `datetime.date` object given an argument tuple suitable for passing to `datetime.date.fromtimestamp()`. New in version 2.4.
Initialization, Finalization, and Threads

void Py_Initialize()
Initialize the Python interpreter. In an application embedding Python, this should be called before using any other Python/C API functions; with the exception of Py_SetProgramName(), PyEval_InitThreads(), PyEval_ReleaseLock(), and PyEval_AcquireLock(). This initializes the table of loaded modules (sys.modules), and creates the fundamental modules __builtin__, __main__, and sys. It also initializes the module search path (sys.path). It does not set sys.argv; use PySys_SetArgv() for that. This is a no-op when called for a second time (without calling Py_Finalize() first). There is no return value; it is a fatal error if the initialization fails.

void Py_InitializeEx(int initsigs)
This function works like Py_Initialize() if initsigs is 1. If initsigs is 0, it skips initialization registration of signal handlers, which might be useful when Python is embedded. New in version 2.4.

int Py_IsInitialized()
Return true (nonzero) when the Python interpreter has been initialized, false (zero) if not. After Py_Finalize() is called, this returns false until Py_Initialize() is called again.

void Py_Finalize()
Undo all initializations made by Py_Initialize() and subsequent use of Python/C API functions, and destroy all sub-interpreters (see Py_NewInterpreter() below) that were created and not yet destroyed since the last call to Py_Initialize(). Ideally, this frees all memory allocated by the Python interpreter. This is a no-op when called for a second time (without calling Py_Initialize() again first). There is no return value; errors during finalization are ignored.

This function is provided for a number of reasons. An embedding application might want to restart Python without having to restart the application itself. An application that has loaded the Python interpreter from a dynamically loadable library (or DLL) might want to free all memory allocated by Python before unloading the DLL. During a hunt for memory leaks in an application a developer might want to free all memory allocated by Python before exiting from the application.

Bugs and caveats: The destruction of modules and objects in modules is done in random order; this may cause destructors (__del__() methods) to fail when they depend on other objects (even functions) or modules. Dynamically loaded extension modules loaded by Python are not unloaded. Small amounts of memory allocated by the Python interpreter may not be freed (if you find a leak, please report it). Memory tied up in circular references between objects is not freed. Some memory allocated by extension modules may not be freed. Some extensions may not work properly if their initialization routine is called more than once; this can happen if an application calls Py_Initialize() and Py_Finalize() more than once.

PyThreadState* Py_NewInterpreter()
Create a new sub-interpreter. This is an (almost) totally separate environment for the execution of Python code. In particular, the new interpreter has separate, independent versions of all imported modules, including the fundamental modules __builtin__, __main__ and sys. The table of loaded modules (sys.modules) and the module search path (sys.path) are also separate. The new environment has no sys.argv variable. It has new standard I/O stream file objects sys.stdin, sys.stdout and sys.stderr (however these refer to the same underlying FILE structures in the C library).

The return value points to the first thread state created in the new sub-interpreter. This thread state is made in the current thread state. Note that no actual thread is created; see the discussion of thread states below.
If creation of the new interpreter is unsuccessful, NULL is returned; no exception is set since the exception state is stored in the current thread state and there may not be a current thread state. (Like all other Python/C API functions, the global interpreter lock must be held before calling this function and is still held when it returns; however, unlike most other Python/C API functions, there needn’t be a current thread state on entry.)

Extension modules are shared between (sub-)interpreters as follows: the first time a particular extension is imported, it is initialized normally, and a (shallow) copy of its module’s dictionary is squirreled away. When the same extension is imported by another (sub-)interpreter, a new module is initialized and filled with the contents of this copy; the extension’s init function is not called. Note that this is different from what happens when an extension is imported after the interpreter has been completely re-initialized by calling Py_Finalize() and Py_Initialize(); in that case, the extension’s init module function is called again.

**Bugs and caveats:** Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn’t perfect — for example, using low-level file operations like os.close() they can (accidentally or maliciously) affect each other’s open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly; this is especially likely when the extension makes use of (static) global variables, or when the extension manipulates its module’s dictionary after its initialization. It is possible to insert objects created in one sub-interpreter into a namespace of another sub-interpreter; this should be done with great care to avoid sharing user-defined functions, methods, instances or classes between sub-interpreters, since import operations executed by such objects may affect the wrong (sub-)interpreter’s dictionary of loaded modules. (XXX This is a hard-to-fix bug that will be addressed in a future release.)

```c
void Py_EndInterpreter (PyThreadState *tstate)

Destroy the (sub-)interpreter represented by the given thread state. The given thread state must be the current thread state. See the discussion of thread states below. When the call returns, the current thread state is NULL. All thread states associated with this interpreter are destroyed. (The global interpreter lock must be held before calling this function and is still held when it returns.) Py_Finalize() will destroy all sub-interpreters that haven’t been explicitly destroyed at that point.
```

```c
void Py_SetProgramName (char *name)

This function should be called before Py_Initialize() is called for the first time, if it is called at all. It tells the interpreter the value of the argv[0] argument to the main() function of the program. This is used by Py_GetPath() and some other functions below to find the Python run-time libraries relative to the interpreter executable. The default value is ’python’. The argument should point to a zero-terminated character string in static storage whose contents will not change for the duration of the program’s execution. No code in the Python interpreter will change the contents of this storage.
```

```c
char* Py_GetProgramName ()

Return the program name set with Py_SetProgramName(), or the default. The returned string points into static storage; the caller should not modify its value.
```

```c
char* Py_GetPrefix ()

Return the prefix for installed platform-independent files. This is derived through a number of complicated rules from the program name set with Py_SetProgramName() and some environment variables; for example, if the program name is ’/usr/local/bin/python’, the prefix is ’/usr/local’. The returned string points into static storage; the caller should not modify its value. This corresponds to the prefix variable in the top-level ’Makefile’ and the --prefix argument to the configure script at build time. The value is available to Python code as sys.prefix. It is only useful on UNIX. See also the next function.
```

```c
char* Py_GetExecPrefix ()

Return the exec-prefix for installed platform-dependent files. This is derived through a number of complicated rules from the program name set with Py_SetProgramName() and some environment variables; for example, if the program name is ’/usr/local/bin/python’, the exec-prefix is ’/usr/local’. The returned string points into static storage; the caller should not modify its value. This corresponds to the exec_prefix variable in the top-level ’Makefile’ and the --exec-prefix argument to the configure script at build time. The value is available to Python code as sys.exec_prefix. It is only useful on UNIX.
```

Background: The exec-prefix differs from the prefix when platform dependent files (such as executables and
shared libraries) are installed in a different directory tree. In a typical installation, platform dependent files may be installed in the `/usr/local/plat` subtree while platform independent may be installed in `/usr/local`.

Generally speaking, a platform is a combination of hardware and software families, e.g. Sparc machines running the Solaris 2.x operating system are considered the same platform, but Intel machines running Solaris 2.x are another platform, and Intel machines running Linux are yet another platform. Different major revisions of the same operating system generally also form different platforms. Non-UNIX operating systems are a different story; the installation strategies on those systems are so different that the prefix and exec-prefix are meaningless, and set to the empty string. Note that compiled Python bytecode files are platform independent (but not independent from the Python version by which they were compiled!).

System administrators will know how to configure the `mount` or `automount` programs to share `/usr/local` between platforms while having `/usr/local/plat` be a different filesystem for each platform.

```c
char* Py_GetProgramFullPath()
```

Return the full program name of the Python executable; this is computed as a side-effect of deriving the default module search path from the program name (set by `Py_SetProgramName()` above). The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.executable`.

```c
char* Py_GetPath()
```

Return the default module search path; this is computed from the program name (set by `Py_SetProgramName()` above) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is `:` on UNIX and Mac OS X, `;` on Windows. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as the list `sys.path`, which may be modified to change the future search path for loaded modules.

```c
const char* Py_GetVersion()
```

Return the version of this Python interpreter. This is a string that looks something like

"1.5 (#67, Dec 31 1997, 22:34:28) [GCC 2.7.2.2]"

The first word (up to the first space character) is the current Python version; the first three characters are the major and minor version separated by a period. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.version`.

```c
const char* Py_GetPlatform()
```

Return the platform identifier for the current platform. On UNIX, this is formed from the “official” name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is ‘sunos5’. On Mac OS X, it is ‘darwin’. On Windows, it is ‘win’. The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.platform`.

```c
const char* Py_GetCopyright()
```

Return the official copyright string for the current Python version, for example

‘Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam’

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as `sys.copyright`.

```c
const char* Py_GetCompiler()
```

Return an indication of the compiler used to build the current Python version, in square brackets, for example:

"[GCC 2.7.2.2]"

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

```c
const char* Py_GetBuildInfo()
```

Return information about the sequence number and build date and time of the current Python interpreter instance, for example
The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable sys.version.

```c
int PySys_SetArgv (int argc, char **argv)
```

Set sys.argv based on argc and argv. These parameters are similar to those passed to the program’s main() function with the difference that the first entry should refer to the script file to be executed rather than the executable hosting the Python interpreter. If there isn’t a script that will be run, the first entry in argv can be an empty string. If this function fails to initialize sys.argv, a fatal condition is signalled using Py_FatalError().

## 8.1 Thread State and the Global Interpreter Lock

The Python interpreter is not fully thread safe. In order to support multi-threaded Python programs, there’s a global lock that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the global interpreter lock may operate on Python objects or call Python/C API functions. In order to support multi-threaded Python programs, the interpreter regularly releases and reacquires the lock — by default, every 100 bytecode instructions (this can be changed with sys.setcheckinterval()). The lock is also released and reacquired around potentially blocking I/O operations like reading or writing a file, so that other threads can run while the thread that requests the I/O is waiting for the I/O operation to complete.

The Python interpreter needs to keep some bookkeeping information separate per thread — for this it uses a data structure called PyThreadState. There’s one global variable, however: the pointer to the current PyThreadState structure. While most thread packages have a way to store “per-thread global data,” Python’s internal platform independent thread abstraction doesn’t support this yet. Therefore, the current thread state must be manipulated explicitly.

This is easy enough in most cases. Most code manipulating the global interpreter lock has the following simple structure:

```
Save the thread state in a local variable.
Release the interpreter lock.
...Do some blocking I/O operation...
Reacquire the interpreter lock.
Restore the thread state from the local variable.
```

This is so common that a pair of macros exists to simplify it:

```c
Py_BEGIN_ALLOW_THREADS
...Do some blocking I/O operation...
Py_END_ALLOW_THREADS
```

The Py_BEGIN_ALLOW_THREADS macro opens a new block and declares a hidden local variable; the Py_END_ALLOW_THREADS macro closes the block. Another advantage of using these two macros is that when Python is compiled without thread support, they are defined empty, thus saving the thread state and lock manipulations.

When thread support is enabled, the block above expands to the following code:
PyThreadState *save;
    save = PyEval_SaveThread();
    ...Do some blocking I/O operation...
    PyEval_RestoreThread(save);

Using even lower level primitives, we can get roughly the same effect as follows:

PyThreadState *save;
    save = PyThreadState_Swap(NULL);
    PyEval_ReleaseLock();
    ...Do some blocking I/O operation...
    PyEval_AcquireLock();
    PyThreadState_Swap(save);

There are some subtle differences; in particular, PyEval_RestoreThread() saves and restores the value of
the global variable errno, since the lock manipulation does not guarantee that errno is left alone. Also, when
thread support is disabled, PyEval_SaveThread() and PyEval_RestoreThread() don’t manipulate
the lock; in this case, PyEval_ReleaseLock() and PyEval_AcquireLock() are not available. This is
done so that dynamically loaded extensions compiled with thread support enabled can be loaded by an interpreter
that was compiled with disabled thread support.

The global interpreter lock is used to protect the pointer to the current thread state. When releasing the lock
and saving the thread state, the current thread state pointer must be retrieved before the lock is released (since another
thread could immediately acquire the lock and store its own thread state in the global variable). Conversely, when
acquiring the lock and restoring the thread state, the lock must be acquired before storing the thread state pointer.

Why am I going on with so much detail about this? Because when threads are created from C, they don’t have
the global interpreter lock, nor is there a thread state data structure for them. Such threads must bootstrap themselves
into existence, by first creating a thread state data structure, then acquiring the lock, and finally storing their thread
state pointer, before they can start using the Python/C API. When they are done, they should reset the thread state
pointer, release the lock, and finally free their thread state data structure.

Beginning with version 2.3, threads can now take advantage of the PyGILState_*() functions to do all of the
above automatically. The typical idiom for calling into Python from a C thread is now:

PyGILState_STATE gstate;
gstate = PyGILState_Ensure(); /* Perform Python actions here. */
result = CallSomeFunction(); /* evaluate result */

/* Release the thread. No Python API allowed beyond this point. */
PyGILState_Release(gstate);

Note that the PyGILState_*() functions assume there is only one global interpreter (created automatically by Py_Initialize()). Python still supports the creation of additional interpreters (using Py_NewInterpreter()), but mixing multiple interpreters and the PyGILState_*() API is unsupported.

PyInterpreterState

This data structure represents the state shared by a number of cooperating threads. Threads belonging to
the same interpreter share their module administration and a few other internal items. There are no public
members in this structure.

Threads belonging to different interpreters initially share nothing, except process state like available memory,
open file descriptors and such. The global interpreter lock is also shared by all threads, regardless of to
which interpreter they belong.

**PyThreadState**

This data structure represents the state of a single thread. The only public data member is
PyInterpreterState *interp, which points to this thread’s interpreter state.

```c
void PyEval_InitThreads()
```

Initialize and acquire the global interpreter lock. It should be called in the main thread before cre-
ating a second thread or engaging in any other thread operations such as PyEval_ReleaseLock() or
PyEval_ReleaseThread(tstate). It is not needed before calling PyEval_SaveThread() or
PyEval_RestoreThread().

This is a no-op when called for a second time. It is safe to call this function before calling
Py_Initialize().

When only the main thread exists, no lock operations are needed. This is a common situation (most Python
programs do not use threads), and the lock operations slow the interpreter down a bit. Therefore, the lock
is not created initially. This situation is equivalent to having acquired the lock: when there is only a single
thread, all object accesses are safe. Therefore, when this function initializes the lock, it also acquires it.
Before the Python thread module creates a new thread, knowing that either it has the lock or the lock
hasn’t been created yet, it calls PyEval_InitThreads(). When this call returns, it is guaranteed that
the lock has been created and that the calling thread has acquired it.

It is not safe to call this function when it is unknown which thread (if any) currently has the global interpreter
lock.

This function is not available when thread support is disabled at compile time.

```c
int PyEval_ThreadsInitialized()
```

Returns a non-zero value if PyEval_InitThreads() has been called. This function can be called
without holding the lock, and therefore can be used to avoid calls to the locking API when running single-
threaded. This function is not available when thread support is disabled at compile time. New in version
2.4.

```c
void PyEval_AcquireLock()
```

Acquire the global interpreter lock. The lock must have been created earlier. If this thread already has the
lock, a deadlock ensues. This function is not available when thread support is disabled at compile time.

```c
void PyEval_ReleaseLock()
```

Release the global interpreter lock. The lock must have been created earlier. This function is not available
when thread support is disabled at compile time.

```c
void PyEval_AcquireThread(PyThreadState *tstate)
```

Acquire the global interpreter lock and set the current thread state to tstate, which should not be NULL. The
lock must have been created earlier. If this thread already has the lock, deadlock ensues. This function is
not available when thread support is disabled at compile time.

```c
void PyEval_ReleaseThread(PyThreadState *tstate)
```

Reset the current thread state to NULL and release the global interpreter lock. The lock must have been
created earlier and must be held by the current thread. The tstate argument, which must not be NULL, is
only used to check that it represents the current thread state — if it isn’t, a fatal error is reported. This
function is not available when thread support is disabled at compile time.

```c
PyThreadState* PyEval_SaveThread()
```

Release the interpreter lock (if it has been created and thread support is enabled) and reset the thread state
to NULL, returning the previous thread state (which is not NULL). If the lock has been created, the current
thread must have acquired it. (This function is available even when thread support is disabled at compile
time.)

```c
void PyEval_RestoreThread(PyThreadState *tstate)
```

Acquire the interpreter lock (if it has been created and thread support is enabled) and set the thread state
to tstate, which must not be NULL. If the lock has been created, the current thread must not have acquired
it, otherwise deadlock ensues. (This function is available even when thread support is disabled at compile
time.)

The following macros are normally used without a trailing semicolon; look for example usage in the Python source.
distribution.

**Py_BEGIN_ALLOW_THREADS**
This macro expands to '{PyThreadState *save; _save = PyEval_SaveThread();}'. Note that it contains an opening brace; it must be matched with a following **Py_END_ALLOW_THREADS** macro. See above for further discussion of this macro. It is a no-op when thread support is disabled at compile time.

**Py_END_ALLOW_THREADS**
This macro expands to 'PyEval_RestoreThread(_save); }'. Note that it contains a closing brace; it must be matched with an earlier **Py_BEGIN_ALLOW_THREADS** macro. See above for further discussion of this macro. It is a no-op when thread support is disabled at compile time.

**Py_BLOCK_THREADS**
This macro expands to 'PyEval_RestoreThread(_save);': it is equivalent to **Py_END_ALLOW_THREADS** without the closing brace. It is a no-op when thread support is disabled at compile time.

**Py_UNBLOCK_THREADS**
This macro expands to '_save = PyEval_SaveThread();': it is equivalent to **Py_BEGIN_ALLOW_THREADS** without the opening brace and variable declaration. It is a no-op when thread support is disabled at compile time.

All of the following functions are only available when thread support is enabled at compile time, and must be called only when the interpreter lock has been created.

- **PyInterpreterState** PyInterpreterState_New()
  Create a new interpreter state object. The interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

- **void** PyInterpreterState_Clear (PyInterpreterState *interp)
  Reset all information in an interpreter state object. The interpreter lock must be held.

- **void** PyInterpreterState_Delete (PyInterpreterState *interp)
  Destroy an interpreter state object. The interpreter lock need not be held. The interpreter state must have been reset with a previous call to **PyInterpreterState_Clear**().

- **PyThreadState** PyThreadState_New (PyInterpreterState *interp)
  Create a new thread state object belonging to the given interpreter object. The interpreter lock need not be held, but may be held if it is necessary to serialize calls to this function.

- **void** PyThreadState_Clear (PyThreadState *tstate)
  Reset all information in a thread state object. The interpreter lock must be held.

- **void** PyThreadState_Delete (PyThreadState *tstate)
  Destroy a thread state object. The interpreter lock need not be held. The thread state must have been reset with a previous call to **PyThreadState_Clear**().

- **PyThreadState** PyThreadState_Get ()
  Return the current thread state. The interpreter lock must be held. When the current thread state is NULL, this issues a fatal error (so that the caller needn’t check for NULL).

- **PyThreadState** PyThreadState_Swap (PyThreadState *tstate)
  Swap the current thread state with the thread state given by the argument *tstate*, which may be NULL. The interpreter lock must be held.

- **PyObject** PyThreadState_GetDict ()
  **Return value:** Borrowed reference.
  Return a dictionary in which extensions can store thread-specific state information. Each extension should use a unique key to use to store state in the dictionary. It is okay to call this function when no current thread state is available. If this function returns NULL, no exception has been raised and the caller should assume no current thread state is available. Changed in version 2.3: Previously this could only be called when a current thread is active, and NULL meant that an exception was raised.

- **int** PyThreadState_SetAsyncExc (long id, PyObject *exc)
  Asynchronously raise an exception in a thread. The id argument is the thread id of the target thread; exc
is the exception object to be raised. This function does not steal any references to exc. To prevent naive misuse, you must write your own C extension to call this. Must be called with the GIL held. Returns the number of thread states modified; if it returns a number greater than one, you’re in trouble, and you should call it again with exc set to NULL to revert the effect. This raises no exceptions. New in version 2.3.

PyGILState_STATE PyGILState_Ensure()
Ensure that the current thread is ready to call the Python C API regardless of the current state of Python, or of its thread lock. This may be called as many times as desired by a thread as long as each call is matched with a call to PyGILState_Release(). In general, other thread-related APIs may be used between PyGILState_Ensure() and PyGILState_Release() calls as long as the thread state is restored to its previous state before the Release(). For example, normal usage of the Py_BEGIN_ALLOW_THREADS and Py_END_ALLOW_THREADS macros is acceptable.

The return value is an opaque "handle" to the thread state when PyGILState_Acquire() was called, and must be passed to PyGILState_Release() to ensure Python is left in the same state. Even though recursive calls are allowed, these handles cannot be shared - each unique call to PyGILState_Ensure must save the handle for its call to PyGILState_Release.

When the function returns, the current thread will hold the GIL. Failure is a fatal error. New in version 2.3.

void PyGILState_Release(PyGILState_STATE)
Release any resources previously acquired. After this call, Python’s state will be the same as it was prior to the corresponding PyGILState_Ensure call (but generally this state will be unknown to the caller, hence the use of the GILState API.)

Every call to PyGILState_Ensure() must be matched by a call to PyGILState_Release() on the same thread. New in version 2.3.

8.2 Profiling and Tracing

The Python interpreter provides some low-level support for attaching profiling and execution tracing facilities. These are used for profiling, debugging, and coverage analysis tools.

Starting with Python 2.2, the implementation of this facility was substantially revised, and an interface from C was added. This C interface allows the profiling or tracing code to avoid the overhead of calling through Python-level callable objects, making a direct C function call instead. The essential attributes of the facility have not changed; the interface allows trace functions to be installed per-thread, and the basic events reported to the trace function are the same as had been reported to the Python-level trace functions in previous versions.

int (*Py_tracefunc)(PyObject *obj, PyFrameObject *frame, int what, PyObject *arg)
The type of the trace function registered using PyEval_SetProfile() and PyEval_SetTrace(). The first parameter is the object passed to the registration function as obj, frame is the frame object to which the event pertains, what is one of the constants PyTrace_CALL, PyTrace_EXCEPTION, PyTrace_LINE, PyTrace_RETURN, PyTrace_C_CALL, PyTrace_C_EXCEPTION, or PyTrace_C_RETURN, and arg depends on the value of what:

<table>
<thead>
<tr>
<th>Value of what</th>
<th>Meaning of arg</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyTrace_CALL</td>
<td>Always NULL.</td>
</tr>
<tr>
<td>PyTrace_EXCEPTION</td>
<td>Exception information as returned by sys.exc_info().</td>
</tr>
<tr>
<td>PyTrace_LINE</td>
<td>Always NULL.</td>
</tr>
<tr>
<td>PyTrace_RETURN</td>
<td>Value being returned to the caller.</td>
</tr>
<tr>
<td>PyTrace_C_CALL</td>
<td>Name of function being called.</td>
</tr>
<tr>
<td>PyTrace_C_EXCEPTION</td>
<td>Always NULL.</td>
</tr>
<tr>
<td>PyTrace_C_RETURN</td>
<td>Always NULL.</td>
</tr>
</tbody>
</table>

int PyTrace_CALL
The value of the what parameter to a Py_tracefunc function when a new call to a function or method is being reported, or a new entry into a generator. Note that the creation of the iterator for a generator function is not reported as there is no control transfer to the Python bytecode in the corresponding frame.

int PyTrace_EXCEPTION
The value of the what parameter to a Py_tracefunc function when an exception has been raised. The callback function is called with this value for what when after any bytecode is processed after which the
exception becomes set within the frame being executed. The effect of this is that as exception propagation causes the Python stack to unwind, the callback is called upon return to each frame as the exception propagates. Only trace functions receive these events; they are not needed by the profiler.

`int PyTrace_LINE`

The value passed as the `what` parameter to a trace function (but not a profiling function) when a line-number event is being reported.

`int PyTrace_RETURN`

The value for the `what` parameter to `Py_tracefunc` functions when a call is returning without propagating an exception.

`int PyTrace_C_CALL`

The value for the `what` parameter to `Py_tracefunc` functions when a C function is about to be called.

`int PyTrace_C_EXCEPTION`

The value for the `what` parameter to `Py_tracefunc` functions when a C function has thrown an exception.

`int PyTrace_C_RETURN`

The value for the `what` parameter to `Py_tracefunc` functions when a C function has returned.

`void PyEval_SetProfile(Py_tracefunc func, PyObject *obj)`

Set the profiler function to `func`. The `obj` parameter is passed to the function as its first parameter, and may be any Python object, or NULL. If the profile function needs to maintain state, using a different value for `obj` for each thread provides a convenient and thread-safe place to store it. The profile function is called for all monitored events except the line-number events.

`void PyEval_SetTrace(Py_tracefunc func, PyObject *obj)`

Set the tracing function to `func`. This is similar to `PyEval_SetProfile()`, except the tracing function does receive line-number events.

### 8.3 Advanced Debugger Support

These functions are only intended to be used by advanced debugging tools.

`PyInterpreterState* PyInterpreterState_Head()`

Return the interpreter state object at the head of the list of all such objects. New in version 2.2.

`PyInterpreterState* PyInterpreterState_Next(PyInterpreterState *interp)`

Return the next interpreter state object after `interp` from the list of all such objects. New in version 2.2.

`PyThreadState * PyInterpreterState_ThreadHead(PyInterpreterState *interp)`

Return the a pointer to the first `PyThreadState` object in the list of threads associated with the interpreter `interp`. New in version 2.2.

`PyThreadState* PyThreadState_Next(PyThreadState *tstate)`

Return the next thread state object after `tstate` from the list of all such objects belonging to the same `PyInterpreterState` object. New in version 2.2.
CHAPTER
NINE

Memory Management

9.1 Overview

Memory management in Python involves a private heap containing all Python objects and data structures. The management of this private heap is ensured internally by the Python memory manager. The Python memory manager has different components which deal with various dynamic storage management aspects, like sharing, segmentation, preallocation or caching.

At the lowest level, a raw memory allocator ensures that there is enough room in the private heap for storing all Python-related data by interacting with the memory manager of the operating system. On top of the raw memory allocator, several object-specific allocators operate on the same heap and implement distinct memory management policies adapted to the peculiarities of every object type. For example, integer objects are managed differently within the heap than strings, tuples or dictionaries because integers imply different storage requirements and speed/space tradeoffs. The Python memory manager thus delegates some of the work to the object-specific allocators, but ensures that the latter operate within the bounds of the private heap.

It is important to understand that the management of the Python heap is performed by the interpreter itself and that the user has no control over it, even if she regularly manipulates object pointers to memory blocks inside that heap. The allocation of heap space for Python objects and other internal buffers is performed on demand by the Python memory manager through the Python/C API functions listed in this document.

To avoid memory corruption, extension writers should never try to operate on Python objects with the functions exported by the C library: malloc(), calloc(), realloc() and free(). This will result in mixed calls between the C allocator and the Python memory manager with fatal consequences, because they implement different algorithms and operate on different heaps. However, one may safely allocate and release memory blocks with the C library allocator for individual purposes, as shown in the following example:

```c
PyObject *res;
char *buf = (char *) malloc(BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
...Do some I/O operation involving buf...
res = PyString_FromString(buf);
free(buf); /* malloc'ed */
return res;
```

In this example, the memory request for the I/O buffer is handled by the C library allocator. The Python memory manager is involved only in the allocation of the string object returned as a result.

In most situations, however, it is recommended to allocate memory from the Python heap specifically because the latter is under control of the Python memory manager. For example, this is required when the interpreter is extended with new object types written in C. Another reason for using the Python heap is the desire to inform the Python memory manager about the memory needs of the extension module. Even when the requested memory is used exclusively for internal, highly-specific purposes, delegating all memory requests to the Python memory manager causes the interpreter to have a more accurate image of its memory footprint as a whole. Consequently,
under certain circumstances, the Python memory manager may or may not trigger appropriate actions, like garbage collection, memory compaction or other preventive procedures. Note that by using the C library allocator as shown in the previous example, the allocated memory for the I/O buffer escapes completely the Python memory manager.

9.2 Memory Interface

The following function sets, modeled after the ANSI C standard, but specifying behavior when requesting zero bytes, are available for allocating and releasing memory from the Python heap:

```c
void* PyMem_Malloc(size_t n)
Allocates n bytes and returns a pointer of type void* to the allocated memory, or NULL if the request fails. Requesting zero bytes returns a distinct non-NULL pointer if possible, as if PyMem_Malloc(1) had been called instead. The memory will not have been initialized in any way.
```

```c
void* PyMem_Realloc(void *p, size_t n)
Resizes the memory block pointed to by p to n bytes. The contents will be unchanged to the minimum of the old and the new sizes. If p is NULL, the call is equivalent to PyMem_Malloc(n); else if n is equal to zero, the memory block is resized but is not freed, and the returned pointer is non-NULL. Unless p is NULL, it must have been returned by a previous call to PyMem_Malloc() or PyMem_Realloc().
```

```c
void PyMem_Free(void *p)
Frees the memory block pointed to by p, which must have been returned by a previous call to PyMem_Malloc() or PyMem_Realloc(). Otherwise, or if PyMem_Free(p) has been called before, undefined behavior occurs. If p is NULL, no operation is performed.
```

The following type-oriented macros are provided for convenience. Note that TYPE refers to any C type.

```c
TYPE* PyMem_New(TYPE, size_t n)
Same as PyMem_Malloc(), but allocates (n * sizeof(TYPE)) bytes of memory. Returns a pointer cast to TYPE*. The memory will not have been initialized in any way.
```

```c
TYPE* PyMem_Resize(void *p, TYPE, size_t n)
Same as PyMem_Realloc(), but the memory block is resized to (n * sizeof(TYPE)) bytes. Returns a pointer cast to TYPE*.
```

```c
void PyMem_Del(void *p)
Same as PyMem_Free().
```

In addition, the following macro sets are provided for calling the Python memory allocator directly, without involving the C API functions listed above. However, note that their use does not preserve binary compatibility across Python versions and is therefore deprecated in extension modules.

PyMem_MALLOC(), PyMem_REALLOC(), PyMem_FREE().
PyMem_NEW(), PyMem_RESIZE(), PyMem_DEL().

9.3 Examples

Here is the example from section 9.1, rewritten so that the I/O buffer is allocated from the Python heap by using the first function set:

```python
PyObject *res;
char *buf = (char *) PyMem_Malloc(BUFSIZ); /* for I/O */
if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyString_FromString(buf);
PyMem_Free(buf); /* allocated with PyMem_Malloc */
return res;
```
The same code using the type-oriented function set:

```c
PyObject *res;
char *buf = PyMem_New(char, BUFSIZ); /* for I/O */

if (buf == NULL)
    return PyErr_NoMemory();
/* ...Do some I/O operation involving buf... */
res = PyString_FromString(buf);
PyMem_Del(buf); /* allocated with PyMem_New */
return res;
```

Note that in the two examples above, the buffer is always manipulated via functions belonging to the same set. Indeed, it is required to use the same memory API family for a given memory block, so that the risk of mixing different allocators is reduced to a minimum. The following code sequence contains two errors, one of which is labeled as fatal because it mixes two different allocators operating on different heaps.

```c
char *buf1 = PyMem_New(char, BUFSIZ);
char *buf2 = (char *) malloc(BUFSIZ);
char *buf3 = (char *) PyMem_Malloc(BUFSIZ);
...
PyMem_Del(buf3); /* Wrong -- should be PyMem_Free() */
free(buf2); /* Right -- allocated via malloc() */
free(buf1); /* Fatal -- should be PyMem_Del() */
```

In addition to the functions aimed at handling raw memory blocks from the Python heap, objects in Python are allocated and released with PyObject_New(), PyObject_NewVar() and PyObject_Del(), or with their corresponding macros PyObject_NEW(), PyObject_NEW_VAR() and PyObject_DEL(). These will be explained in the next chapter on defining and implementing new object types in C.
CHAPTER
TEN

Object Implementation Support

This chapter describes the functions, types, and macros used when defining new object types.

10.1 Allocating Objects on the Heap

PyObject* _PyObject_New (PyTypeObject *type)

PyObject* _PyObject_NewVar (PyTypeObject *type, int size)

void _PyObject_Del (PyObject *op)

PyObject* PyObject_Init (PyObject *op, PyTypeObject *type)
Return value: Borrowed reference.
Initialize a newly-allocated object op with its type and initial reference. Returns the initialized object. If type indicates that the object participates in the cyclic garbage detector, it is added to the detector’s set of observed objects. Other fields of the object are not affected.

PyVarObject* PyObject_InitVar (PyVarObject *op, PyTypeObject *type, int size)
Return value: Borrowed reference.
This does everything PyObject_Init() does, and also initializes the length information for a variable-size object.

TYPE* PyObject_New (TYPE, PyTypeObject *type)
Allocate a new Python object using the C structure type TYPE and the Python type object type. Fields not defined by the Python object header are not initialized; the object’s reference count will be one. The size of the memory allocation is determined from the tp_basicsize field of the type object.

TYPE* PyObject_NewVar (TYPE, PyTypeObject *type, int size)
Allocate a new Python object using the C structure type TYPE and the Python type object type. Fields not defined by the Python object header are not initialized. The allocated memory allows for the TYPE structure plus size fields of the size given by the tp_itemsize field of type. This is useful for implementing objects like tuples, which are able to determine their size at construction time. Embedding the array of fields into the same allocation decreases the number of allocations, improving the memory management efficiency.

void PyObject_Del (PyObject *op)
Releases memory allocated to an object using PyObject_New() or PyObject_NewVar(). This is normally called from the tp_dealloc handler specified in the object’s type. The fields of the object should not be accessed after this call as the memory is no longer a valid Python object.

TYPE* PyObject_NEW (TYPE, PyTypeObject *type)
Macro version of PyObject_New(), to gain performance at the expense of safety. This does not check type for a NULL value.

TYPE* PyObject_NEW_VAR (TYPE, PyTypeObject *type, int size)
Macro version of PyObject_NewVar(), to gain performance at the expense of safety. This does not
check type for a NULL value.

void PyObject_DEL(PyObject *op)
Macro version of PyObject_Del().

PyObject* Py_InitModule(char *name, PyMethodDef *methods)
Return value: Borrowed reference.
Create a new module object based on a name and table of functions, returning the new module object.
Changed in version 2.3: Older versions of Python did not support NULL as the value for the methods argument.

PyObject* Py_InitModule3(char *name, PyMethodDef *methods, char *doc)
Return value: Borrowed reference.
Create a new module object based on a name and table of functions, returning the new module object. If doc is non-NULL, it will be used to define the docstring for the module.
Changed in version 2.3: Older versions of Python did not support NULL as the value for the methods argument.

PyObject* Py_InitModule4(char *name, PyMethodDef *methods, char *doc, PyObject *self, int apiver)
Return value: Borrowed reference.
Create a new module object based on a name and table of functions, returning the new module object. If doc is non-NULL, it will be used to define the docstring for the module. If self is non-NULL, it will passed to the functions of the module as their (otherwise NULL) first parameter. (This was added as an experimental feature, and there are no known uses in the current version of Python.) For apiver, the only value which should be passed is defined by the constant PYTHON_API_VERSION.

Note: Most uses of this function should probably be using the Py_InitModule3() instead; only use this if you are sure you need it.
Changed in version 2.3: Older versions of Python did not support NULL as the value for the methods argument.

DL_IMPORT

PyObject _Py_NoneStruct
Object which is visible in Python as None. This should only be accessed using the Py_None macro, which evaluates to a pointer to this object.

10.2 Common Object Structures

There are a large number of structures which are used in the definition of object types for Python. This section describes these structures and how they are used.

All Python objects ultimately share a small number of fields at the beginning of the object’s representation in memory. These are represented by the PyObject and PyVarObject types, which are defined, in turn, by the expansions of some macros also used, whether directly or indirectly, in the definition of all other Python objects.

PyObject
All object types are extensions of this type. This is a type which contains the information Python needs to treat a pointer to an object as an object. In a normal “release” build, it contains only the objects reference count and a pointer to the corresponding type object. It corresponds to the fields defined by the expansion of the PyObject_HEAD macro.

PyVarObject
This is an extension of PyObject that adds the ob_size field. This is only used for objects that have some notion of length. This type does not often appear in the Python/C API. It corresponds to the fields defined by the expansion of the PyObject_VAR_HEAD macro.

These macros are used in the definition of PyObject and PyVarObject:

PyObject_HEAD
This is a macro which expands to the declarations of the fields of the PyObject type; it is used when declaring new types which represent objects without a varying length. The specific fields it expands to de-
pend on the definition of Py_TRACE_REFS. By default, that macro is not defined, and PyObject _HEAD expands to:

```c
int ob_refcnt;
PyTypeObject *ob_type;
```

When Py_TRACE_REFS is defined, it expands to:

```c
PyObject *_ob_next, *_ob_prev;
int ob_refcnt;
PyTypeObject *ob_type;
```

**PyObject_VAR_HEAD**
This is a macro which expands to the declarations of the fields of the PyVarObject type; it is used when declaring new types which represent objects with a length that varies from instance to instance. This macro always expands to:

```c
PyObject_HEAD
int ob_size;
```

Note that PyObject _HEAD is part of the expansion, and that it’s own expansion varies depending on the definition of Py_TRACE_REFS.

**PyObject_HEAD_INIT**

**PyCFunction**
Type of the functions used to implement most Python callables in C. Functions of this type take two PyObject* parameters and return one such value. If the return value is NULL, an exception shall have been set. If not NULL, the return value is interpreted as the return value of the function as exposed in Python. The function must return a new reference.

**PyMethodDef**
Structure used to describe a method of an extension type. This structure has four fields:

<table>
<thead>
<tr>
<th>Field</th>
<th>C Type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ml_name</td>
<td>char *</td>
<td>name of the method</td>
</tr>
<tr>
<td>ml_meth</td>
<td>PyCFunction</td>
<td>pointer to the C implementation</td>
</tr>
<tr>
<td>ml_flags</td>
<td>int</td>
<td>flag bits indicating how the call should be constructed</td>
</tr>
<tr>
<td>ml_doc</td>
<td>char *</td>
<td>points to the contents of the docstring</td>
</tr>
</tbody>
</table>

The ml_meth is a C function pointer. The functions may be of different types, but they always return PyObject*. If the function is not of the PyCFunction, the compiler will require a cast in the method table. Even though PyCFunction defines the first parameter as PyObject*, it is common that the method implementation uses a the specific C type of the self object.

The ml_flags field is a bitfield which can include the following flags. The individual flags indicate either a calling convention or a binding convention. Of the calling convention flags, only METH_VARARGS and METH_KEYWORDS can be combined (but note that METH_KEYWORDS alone is equivalent to METH_VARARGS | METH_KEYWORDS). Any of the calling convention flags can be combined with a binding flag.

**METH_VARARGS**
This is the typical calling convention, where the methods have the type PyCFunction. The function expects two PyObject* values. The first one is the self object for methods; for module functions, it has the value given to Py_InitModule4 () (or NULL if Py_InitModule () was used). The second parameter (often called args) is a tuple object representing all arguments. This parameter is typically processed using PyArg_ParseTuple () or PyArg_UnpackTuple.

**METH_KEYWORDS**
Methods with these flags must be of type PyCFunctionWithKeywords. The function expects three parameters: self, args, and a dictionary of all the keyword arguments. The flag

---

10.2. Common Object Structures
is typically combined with METH_VARARGS, and the parameters are typically processed using
PyArg_ParseTupleAndKeywords().

METH_NOARGS
Methods without parameters don’t need to check whether arguments are given if they are listed with the
METH_NOARGS flag. They need to be of type PyCFunction. When used with object methods, the first
parameter is typically named self and will hold a reference to the object instance. In all cases the second
parameter will be NULL.

METH_O
Methods with a single object argument can be listed with the METH_O flag, instead of invoking
PyArg_ParseTuple() with a "O" argument. They have the type PyCFunction, with the self pa-
rameter, and a PyObject* parameter representing the single argument.

METH_OLDARGS
This calling convention is deprecated. The method must be of type PyCFunction. The second argument
is NULL if no arguments are given, a single object if exactly one argument is given, and a tuple of objects
if more than one argument is given. There is no way for a function using this convention to distinguish
between a call with multiple arguments and a call with a tuple as the only argument.

These two constants are not used to indicate the calling convention but the binding when use with methods of
classes. These may not be used for functions defined for modules. At most one of these flags may be set for any
given method.

METH_CLASS
The method will be passed the type object as the first parameter rather than an instance of the type. This is
used to create class methods, similar to what is created when using the classmethod() built-in function.
New in version 2.3.

METH_STATIC
The method will be passed NULL as the first parameter rather than an instance of the type. This is used
to create static methods, similar to what is created when using the staticmethod() built-in function.
New in version 2.3.

One other constant controls whether a method is loaded in place of another definition with the same method name.

METH_COEXIST
The method will be loaded in place of existing definitions. Without METH_COEXIST, the default is to skip
repeated definitions. Since slot wrappers are loaded before the method table, the existence of a sq__contains
slot, for example, would generate a wrapped method named __contains__() and preclude the loading
of a corresponding PyCFunction with the same name. With the flag defined, the PyCFunction will be loaded
in place of the wrapper object and will co-exist with the slot. This is helpful because calls to PyCFunctions
are optimized more than wrapper object calls. New in version 2.4.

PyObject* Py_FindMethod(PyMethodDef table[], PyObject *ob, char *name)
Return a bound method object for an extension type implemented in C. This can be use-
ful in the implementation of a tp_getattro or tp_getattr handler that does not use the
PyObject_GenericGetAttr() function.

10.3 Type Objects

Perhaps one of the most important structures of the Python object system is the structure that defines a new type:
the PyTypeObject structure. Type objects can be handled using any of the PyObject_*() or PyType_*() func-
tions, but do not offer much that’s interesting to most Python applications. These objects are fundamental to
how objects behave, so they are very important to the interpreter itself and to any extension module that implements
new types.

Type objects are fairly large compared to most of the standard types. The reason for the size is that each type
object stores a large number of values, mostly C function pointers, each of which implements a small part of the
type’s functionality. The fields of the type object are examined in detail in this section. The fields will be described
in the order in which they occur in the structure.
typedef struct _typeobject {
    PyObject_VAR_HEAD
    char *tp_name; /* For printing, in format "<module>.<name>" */
    int tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */
    destructor tp_dealloc;
    printfunc tp_print;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    cmpfunc tp_compare;
    reprfunc tp_repr;

    /* Method suites for standard classes */
    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */
    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;

    /* Flags to define presence of optional/expanded features */
    long tp_flags;
    char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    long tp_weaklistoffset;

    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
} PyTypeObject

10.3. Type Objects
struct PyMemberDef *tp_members;
struct PyGetSetDef *tp_getset;
struct _typeobject *tp_base;
PyObject *tp_dict;
descrgetfunc tp_descr_get;
desrcsetfunc tp_descr_set;
long tp_dictoffset;
initproc tp_init;
allocfunc tp_alloc;
newfunc tp_new;
freefunc tp_free; /* Low-level free-memory routine */
inquiry tp_is_gc; /* For PyObject_IS_GC */
PyObject *tp_bases;
PyObject *tp_mro; /* method resolution order */
PyObject *tp_cache;
PyObject *tp_subclasses;
PyObject *tp_weaklist;
} PyTypeObject;

The type object structure extends the PyVarObject structure. The ob_size field is used for dynamic types (created by type_new(), usually called from a class statement). Note that PyType_Type (the metatype) initializes tp_itemsize, which means that its instances (i.e. type objects) must have the ob_size field.

PyObject* __ob_next
PyObject* __ob_prev

These fields are only present when the macro Py_TRACE_REFS is defined. Their initialization to NULL is taken care of by the PyObject_HEAD_INIT macro. For statically allocated objects, these fields always remain NULL. For dynamically allocated objects, these two fields are used to link the object into a doubly-linked list of all live objects on the heap. This could be used for various debugging purposes; currently the only use is to print the objects that are still alive at the end of a run when the environment variable PYTHONDUMPREFS is set.

These fields are not inherited by subtypes.

int ob_refcnt
This is the type object’s reference count, initialized to 1 by the PyObject_HEAD_INIT macro. Note that for statically allocated type objects, the type’s instances (objects whose ob_type points back to the type) do not count as references. But for dynamically allocated type objects, the instances do count as references. This field is not inherited by subtypes.

PyTypeObject* ob_type
This is the type’s type, in other words its metatype. It is initialized by the argument to the PyObject_HEAD_INIT macro, and its value should normally be &PyType_Type. However, for dynamically loadable extension modules that must be usable on Windows (at least), the compiler complains that this is not a valid initializer. Therefore, the convention is to pass NULL to the PyObject_HEAD_INIT macro and to initialize this field explicitly at the start of the module’s initialization function, before doing anything else. This is typically done like this:

    Foo_Type.ob_type = &PyType_Type;

This should be done before any instances of the type are created. PyType_Ready() checks if ob_type is NULL, and if so, initializes it: in Python 2.2, it is set to &PyType_Type; in Python 2.2.1 and later it is initialized to the ob_type field of the base class. PyType_Ready() will not change this field if it is non-zero.

In Python 2.2, this field is not inherited by subtypes. In 2.2.1, and in 2.3 and beyond, it is inherited by subtypes.

int ob_size
For statically allocated type objects, this should be initialized to zero. For dynamically allocated type objects, this field has a special internal meaning.

This field is not inherited by subtypes.
char* tp_name
Pointer to a NUL-terminated string containing the name of the type. For types that are accessible as module globals, the string should be the full module name, followed by a dot, followed by the type name; for built-in types, it should be just the type name. If the module is a submodule of a package, the full package name is part of the full module name. For example, a type named T defined in module M in subpackage Q in package P should have the tp_name initializer "P.Q.M.T".

For dynamically allocated type objects, this should just be the type name, and the module name explicitly stored in the type dict as the value for key ‘__module__’.

For statically allocated type objects, the tp_name field should contain a dot. Everything before the last dot is made accessible as the __module__ attribute, and everything after the last dot is made accessible as the __name__ attribute.

If no dot is present, the entire tp_name field is made accessible as the __name__ attribute, and the __module__ attribute is undefined (unless explicitly set in the dictionary, as explained above). This means your type will be impossible to pickle.

This field is not inherited by subtypes.

int tp_basicsize
int tp_itemsize
These fields allow calculating the size in bytes of instances of the type.

There are two kinds of types: types with fixed-length instances have a zero tp_itemsize field, types with variable-length instances have a non-zero tp_itemsize field. For a type with fixed-length instances, all instances have the same size, given in tp_basicsize.

For a type with variable-length instances, the instances must have an ob_size field, and the instance size is tp_basicsize plus N times tp_itemsize, where N is the “length” of the object. The value of N is typically stored in the instance’s ob_size field. There are exceptions: for example, long ints use a negative ob_size to indicate a negative number, and N is abs(ob_size) there. Also, the presence of an ob_size field in the instance layout doesn’t mean that the instance structure is variable-length (for example, the structure for the list type has fixed-length instances, yet those instances have a meaningful ob_size field).

The basic size includes the fields in the instance declared by the macro PyObject_HEAD or PyObject_VAR_HEAD (whichever is used to declare the instance struct) and this in turn includes the _ob_prev and _ob_next fields if they are present. This means that the only correct way to get an initializer for the tp_basicsize is to use the sizeof operator on the struct used to declare the instance layout. The basic size does not include the GC header size (this is new in Python 2.2; in 2.1 and 2.0, the GC header size was included in tp_basicsize).

These fields are inherited separately by subtypes. If the base type has a non-zero tp_itemsize, it is generally not safe to set tp_itemsize to a different non-zero value in a subtype (though this depends on the implementation of the base type).

A note about alignment: if the variable items require a particular alignment, this should be taken care of by the value of tp_basicsize. Example: suppose a type implements an array of double. tp_itemsize is sizeof(double). It is the programmer’s responsibility that tp_basicsize is a multiple of sizeof(double) (assuming this is the alignment requirement for double).

destructor tp_dealloc
A pointer to the instance destructor function. This function must be defined unless the type guarantees that its instances will never be deallocated (as is the case for the singletons None and Ellipsis).

The destructor function is called by the Py_DECREF() and Py_XDECREF() macros when the new reference count is zero. At this point, the instance is still in existence, but there are no references to it. The destructor function should free all references which the instance owns, free all memory buffers owned by the instance (using the freeing function corresponding to the allocation function used to allocate the buffer), and finally (as its last action) call the type’s tp_free function. If the type is not subtypable (doesn’t have the Py_TPFLAGS_BASETYPE flag bit set), it is permissible to call the object deallocator directly instead of via tp_free. The object deallocator should be the one used to allocate the instance; this is normally PyObject_DEL() if the instance was allocated using PyObject_New() or PyObject_VarNew(), or PyObject_GC_Del() if the instance was allocated using PyObject_GC_New() or PyObject_GC_VarNew().

10.3. Type Objects
This field is inherited by subtypes.

printfunc tp_print
An optional pointer to the instance print function.

The print function is only called when the instance is printed to a real file; when it is printed to a pseudo-file (like a StringIO instance), the instance's tp_repr or tp_str function is called to convert it to a string. These are also called when the type's tp_print field is NULL. A type should never implement tp_print in a way that produces different output than tp_repr or tp_str would.

The print function is called with the same signature as PyObject_Print(): int tp_print(PyObject *self, FILE *file, int flags). The self argument is the instance to be printed. The file argument is the stdio file to which it is to be printed. The flags argument is composed of flag bits. The only flag bit currently defined is Py_PRINT_RAW. When the Py_PRINT_RAW flag bit is set, the instance should be printed the same way as tp_str would format it; when the Py_PRINT_RAW flag bit is clear, the instance should be printed the same way as tp_repr would format it. It should return -1 and set an exception condition when an error occurred during the comparison.

It is possible that the tp_print field will be deprecated. In any case, it is recommended not to define tp_print, but instead to rely on tp_repr and tp_str for printing.

This field is inherited by subtypes.

getattrfunc tp_getattr
An optional pointer to the get-attribute-string function.

This field is deprecated. When it is defined, it should point to a function that acts the same as the tp_getattro function, but taking a C string instead of a Python string object to give the attribute name.

The signature is the same as for PyObject_GetAttrString().

This field is inherited by subtypes together with tp_getattro: a subtype inherits both tp_getattr and tp_getattro from its base type when the subtype's tp_getattr and tp_getattro are both NULL.

setattrfunc tp_setattr
An optional pointer to the set-attribute-string function.

This field is deprecated. When it is defined, it should point to a function that acts the same as the tp_setattro function, but taking a C string instead of a Python string object to give the attribute name.

The signature is the same as for PyObject_SetAttrString().

This field is inherited by subtypes together with tp_setattro: a subtype inherits both tp_setattr and tp_setattro from its base type when the subtype's tp_setattr and tp_setattro are both NULL.

cmpfunc tp_compare
An optional pointer to the three-way comparison function.

The signature is the same as for PyObject_Compare(). The function should return 1 if self greater than other, 0 if self is equal to other, and -1 if self less than other. It should return -1 and set an exception condition when an error occurred during the comparison.

This field is inherited by subtypes together with tp_richcompare and tp_hash: a subtype inherits all three of tp_compare, tp_richcompare, and tp_hash when the subtype's tp_compare, tp_richcompare, and tp_hash are all NULL.

reprfunc tp_repr
An optional pointer to a function that implements the built-in function repr().

The signature is the same as for PyObject_Repr(); it must return a string or a Unicode object. Ideally, this function should return a string that, when passed to eval(), given a suitable environment, returns an object with the same value. If this is not feasible, it should return a string starting with ‘<’ and ending with ‘>’ from which both the type and the value of the object can be deduced.

When this field is not set, a string of the form ‘<%s object at %p>’ is returned, where %s is replaced by the type name, and %p by the object’s memory address.

This field is inherited by subtypes.

PyNumberMethods *tp_as_number;
hashfunc `tp_hash`
An optional pointer to a function that implements the built-in function `hash()`.
The signature is the same as for `PyObject_Hash()`: it must return a C long. The value -1 should not be returned as a normal return value; when an error occurs during the computation of the hash value, the function should set an exception and return -1.
When this field is not set, two possibilities exist: if the `tp_compare` and `tp_richcompare` fields are both NULL, a default hash value based on the object’s address is returned; otherwise, a TypeError is raised.
This field is inherited by subtypes together with `tp_richcompare` and `tp_compare`: a subtype inherits all three of `tp_compare`, `tp_richcompare`, and `tp_hash`, when the subtype’s `tp_compare`, `tp_richcompare` and `tp_hash` are all NULL.

ternaryfunc `tp_call`
An optional pointer to a function that implements calling the object. This should be NULL if the object is not callable. The signature is the same as for `PyObject_Call()`.
This field is inherited by subtypes.

reprfunc `tp_str`
An optional pointer to a function that implements the built-in operation `str()`. (Note that `str` is a type now, and `str()` calls the constructor for that type. This constructor calls `PyObject_Str()` to do the actual work, and `PyObject_Str()` will call this handler.)
The signature is the same as for `PyObject_Str()`: it must return a string or a Unicode object. This function should return a “friendly” string representation of the object, as this is the representation that will be used by the print statement.
When this field is not set, `PyObject_Repr()` is called to return a string representation.
This field is inherited by subtypes.

getattrofunc `tp_getattro`
An optional pointer to the get-attribute function.
The signature is the same as for `PyObject_GetAttr()`. It is usually convenient to set this field to `PyObject_GenericGetAttr()`, which implements the normal way of looking for object attributes.
This field is inherited by subtypes together with `tp_getattr`: a subtype inherits both `tp_getattr` and `tp_getattro` from its base type when the subtype’s `tp_getattr` and `tp_getattro` are both NULL.

setattrofunc `tp_setattro`
An optional pointer to the set-attribute function.
The signature is the same as for `PyObject_SetAttr()`. It is usually convenient to set this field to `PyObject_GenericSetAttr()`, which implements the normal way of setting object attributes.
This field is inherited by subtypes together with `tp_setattr`: a subtype inherits both `tp_setattr` and `tp_setattro` from its base type when the subtype’s `tp_setattr` and `tp_setattro` are both NULL.

PyBufferProcs* `tp_as_buffer`
Pointer to an additional structure that contains fields relevant only to objects which implement the buffer interface. These fields are documented in “Buffer Object Structures” (section 10.7).
The `tp_as_buffer` field is not inherited, but the contained fields are inherited individually.

long `tp_flags`
This field is a bit mask of various flags. Some flags indicate variant semantics for certain situations; others are used to indicate that certain fields in the type object (or in the extension structures referenced via `tp_as_number`, `tp_as_sequence`, `tp_as_mapping`, and `tp_as_buffer`) that were historically
not always present are valid; if such a flag bit is clear, the type fields it guards must not be accessed and must be considered to have a zero or NULL value instead.

Inheritance of this field is complicated. Most flag bits are inherited individually, i.e. if the base type has a flag bit set, the subtype inherits this flag bit. The flag bits that pertain to extension structures are strictly inherited if the extension structure is inherited, i.e. the base type’s value of the flag bit is copied into the subtype together with a pointer to the extension structure. The Py_TPFLAGS_HAVE_GC flag bit is inherited together with the tp_traverse and tp_clear fields, i.e. if the Py_TPFLAGS_HAVE_GC flag bit is clear in the subtype and the tp_traverse and tp_clear fields in the subtype exist (as indicated by the Py_TPFLAGS_HAVE_RICHCOMPARE flag bit) and have NULL values.

The following bit masks are currently defined; these can be or-ed together using the | operator to form the value of the tp_flags field. The macro PyType_HasFeature() takes a type and a flags value, tp and f, and checks whether tp->tp_flags & f is non-zero.

**Py_TPFLAGS_HAVE_GETCHARBUFFER**

If this bit is set, the PyBufferProcs struct referenced by tp_as_buffer has the bf_getcharbuffer field.

**Py_TPFLAGS_HAVE_SEQUENCE_IN**

If this bit is set, the PySequenceMethods struct referenced by tp_as_sequence has the sq_contains field.

**Py_TPFLAGS_GC**

This bit is obsolete. The bit it used to name is no longer in use. The symbol is now defined as zero.

**Py_TPFLAGS_HAVE_INPLACEOPS**

If this bit is set, the PySequenceMethods struct referenced by tp_as_sequence and the PyNumberMethods structure referenced by tp_as_number contain the fields for in-place operators. In particular, this means that the PyNumberMethods structure has the fields nb_inplace_add, nb_inplace_subtract, nb_inplace_multiply, nb_inplace_divide, nb_inplace_remainder, nb_inplace_power, nb_inplace_lshift, nb_inplace_rshift, nb_inplace_and, nb_inplace_xor, and nb_inplace_or; and the PySequenceMethods struct has the fields sq_inplace_concat and sq_inplace_repeat.

**Py_TPFLAGS_CHECKTYPES**

If this bit is set, the binary and ternary operations in the PyNumberMethods structure referenced by tp_as_number accept arguments of arbitrary object types, and do their own type conversions if needed. If this bit is clear, those operations require that all arguments have the current type as their type, and the caller is supposed to perform a coercion operation first. This applies to nb_add, nb_subtract, nb_multiply, nb_divide, nb_remainder, nb_divmod, nb_power, nb_lshift, nb_rshift, nb_and, nb_xor, and nb_or.

**Py_TPFLAGS_HAVE_RICHCOMPARE**

If this bit is set, the type object has the tp_richcompare field, as well as the tp_traverse and the tp_clear fields.

**Py_TPFLAGS_HAVE_WEAKREFS**

If this bit is set, the tp_weaklistoffset field is defined. Instances of a type are weakly referenceable if the type’s tp_weaklistoffset field has a value greater than zero.

**Py_TPFLAGS_HAVE_ITER**

If this bit is set, the type object has the tp_iter and tp_iternext fields.

**Py_TPFLAGS_HAVE_CLASS**

If this bit is set, the type object has several new fields defined starting in Python 2.2: tp_methods, tp_members, tp_getset, tp_base, tp_dict, tp_descr_get, tp_descr_set, tp_dictoffset, tp_init, tp_alloc, tp_new, tp_free, tp_is_gc, tp_bases, tp_mro, tp_cache, tp_subclasses, and tp_weaklist.

**Py_TPFLAGS_HEAPTYPE**

This bit is set when the type object itself is allocated on the heap. In this case, the ob_type field of its instances is considered a reference to the type, and the type object is INCREF’ed when a new instance is created, and DECREF’ed when an instance is destroyed (this does not apply to instances of subtypes; only the type referenced by the instance’s ob_type gets INCREF’ed or DECREF’ed).
**Py_TPFLAGS_BASETYPE**
This bit is set when the type can be used as the base type of another type. If this bit is clear, the type cannot be subtyped (similar to a "final" class in Java).

**Py_TPFLAGS_READY**
This bit is set when the type object has been fully initialized by PyType_Ready().

**Py_TPFLAGS_READYING**
This bit is set while PyType_Ready() is in the process of initializing the type object.

**Py_TPFLAGS_HAVE_GC**
This bit is set when the object supports garbage collection. If this bit is set, instances must be created using PyObject_GC_New() and destroyed using PyObject_GC_Del(). More information in section XXX about garbage collection. This bit also implies that the GC-related fields tp_traverse and tp_clear are present in the type object; but those fields also exist when Py_TPFLAGS_HAVE_GC is clear but Py_TPFLAGS_HAVE_RICHCOMPARE is set.

**Py_TPFLAGS_DEFAULT**
This is a bitmask of all the bits that pertain to the existence of certain fields in the type object and its extension structures. Currently, it includes the following bits: Py_TPFLAGS_HAVE_GETCHARBUFFER, Py_TPFLAGS_HAVE_SEQUENCE_IN, Py_TPFLAGS_HAVE_INPLACEOPS, Py_TPFLAGS_HAVE_RICHCOMPARE, Py_TPFLAGS_HAVE_WEAKREFS, Py_TPFLAGS_HAVE_ITER, and Py_TPFLAGS_HAVE_CLASS.

char* tp_doc
An optional pointer to a NUL-terminated C string giving the docstring for this type object. This is exposed as the __doc__ attribute on the type and instances of the type.

This field is not inherited by subtypes.

The following three fields only exist if the Py_TPFLAGS_HAVE_RICHCOMPARE flag bit is set.

traverseproc tp_traverse
An optional pointer to a traversal function for the garbage collector. This is only used if the Py_TPFLAGS_HAVE_GC flag bit is set. More information in section 10.9 about garbage collection.

This field is inherited by subtypes together with tp_clear and the Py_TPFLAGS_HAVE_GC flag bit: the flag bit, tp_traverse, and tp_clear are all inherited from the base type if they are all zero in the subtype and the subtype has the Py_TPFLAGS_HAVE_RICHCOMPARE flag bit set.

inquiry tp_clear
An optional pointer to a clear function for the garbage collector. This is only used if the Py_TPFLAGS_HAVE_GC flag bit is set. More information in section 10.9 about garbage collection.

This field is inherited by subtypes together with tp_clear and the Py_TPFLAGS_HAVE_GC flag bit: the flag bit, tp_traverse, and tp_clear are all inherited from the base type if they are all zero in the subtype and the subtype has the Py_TPFLAGS_HAVE_RICHCOMPARE flag bit set.

richcmpfunc tp_richcompare
An optional pointer to a rich comparison function.

The signature is the same as for PyObject_RichCompare(). The function should return 1 if the requested comparison returns true, 0 if it returns false. It should return -1 and set an exception condition when an error occurred during the comparison.

This field is inherited by subtypes together with tp_compare and tp_hash: a subtype inherits all three of tp_compare, tp_richcompare, and tp_hash, when the subtype’s tp_compare, tp_richcompare, and tp_hash are all NULL.

The following constants are defined to be used as the third argument for tp_richcompare and for PyObject_RichCompare():

---

10.3. Type Objects 97
The next field only exists if the `Py_TPFLAGS_HAVE_WEAKREFS` flag bit is set.

```c
long tp_weaklistoffset
```

If the instances of this type are weakly referenceable, this field is greater than zero and contains the offset in the instance structure of the weak reference list head (ignoring the GC header, if present); this offset is used by `PyObject_ClearWeakRefs()` and the `PyWeakref_*()` functions. The instance structure needs to include a field of type `PyObject*` which is initialized to `NULL`.

Do not confuse this field with `tp_weaklist`; that is the list head for weak references to the type object itself.

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype uses a different weak reference list head than the base type. Since the list head is always found via `tp_weaklistoffset`, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types are weakly referenceable, the type is made weakly referenceable by adding a weak reference list head slot to the instance layout and setting the `tp_weaklistoffset` of that slot’s offset.

When a type’s `__slots__` declaration contains a slot named `__weakref__`, that slot becomes the weak reference list head for instances of the type, and the slot’s offset is stored in the type’s `tp_weaklistoffset`.

When a type’s `__slots__` declaration does not contain a slot named `__weakref__`, the type inherits its `tp_weaklistoffset` from its base type.

The next two fields only exist if the `Py_TPFLAGS_HAVE_CLASS` flag bit is set.

```c
getiterfunc tp_iter
```

An optional pointer to a function that returns an iterator for the object. Its presence normally signals that the instances of this type are iterable (although sequences may be iterable without this function, and classic instances always have this function, even if they don’t define an `__iter__` method).

This function has the same signature as `PyObject_GetIter()`.

This field is inherited by subtypes.

```c
iternextfunc tp_iternext
```

An optional pointer to a function that returns the next item in an iterator, or raises `StopIteration` when the iterator is exhausted. Its presence normally signals that the instances of this type are iterators (although classic instances always have this function, even if they don’t define a `next()` method).

Iterator types should also define the `tp_iter` function, and that function should return the iterator instance itself (not a new iterator instance).

This function has the same signature as `PyIter_Next()`.

This field is inherited by subtypes.

The next fields, up to and including `tp_weaklist`, only exist if the `Py_TPFLAGS_HAVE_CLASS` flag bit is set.

```c
struct PyMethodDef* tp_methods
```

An optional pointer to a static NULL-terminated array of `PyMethodDef` structures, declaring regular methods of this type.

For each entry in the array, an entry is added to the type’s dictionary (see `tp_dict` below) containing a method descriptor.

This field is not inherited by subtypes (methods are inherited through a different mechanism).
struct PyMemberDef* tp_members
An optional pointer to a static NULL-terminated array of PyMemberDef structures, declaring regular data members (fields or slots) of instances of this type.

For each entry in the array, an entry is added to the type’s dictionary (see tp_dict below) containing a member descriptor.

This field is not inherited by subtypes (members are inherited through a different mechanism).

struct PyGetSetDef* tp_getset
An optional pointer to a static NULL-terminated array of PyGetSetDef structures, declaring computed attributes of instances of this type.

For each entry in the array, an entry is added to the type’s dictionary (see tp_dict below) containing a getset descriptor.

This field is not inherited by subtypes (computed attributes are inherited through a different mechanism).

Docs for PyGetSetDef (XXX belong elsewhere):

typedef PyObject *(getter)(PyObject *, void *);
typedef int *(setter)(PyObject *, PyObject *, void *);

typedef struct PyGetSetDef {
    char *name; /* attribute name */
    getter get; /* C function to get the attribute */
    setter set; /* C function to set the attribute */
    char *doc; /* optional doc string */
    void *closure; /* optional additional data for getter and setter */
} PyGetSetDef;

PyTypeObject* tp_base
An optional pointer to a base type from which type properties are inherited. At this level, only single inheritance is supported; multiple inheritance require dynamically creating a type object by calling the metatype.

This field is not inherited by subtypes (obviously), but it defaults to &PyBaseObject_Type (which to Python programmers is known as the type object).

PyObject* tp_dict
The type’s dictionary is stored here by PyType_Ready(). This field should normally be initialized to NULL before PyType_Ready is called; it may also be initialized to a dictionary containing initial attributes for the type. Once PyType_Ready() has initialized the type, extra attributes for the type may be added to this dictionary only if they don’t correspond to overloaded operations (like __add__()).

This field is not inherited by subtypes (though the attributes defined in here are inherited through a different mechanism).

descrgetfunc tp_descr_get
An optional pointer to a ”descriptor get” function.

XXX blah, blah.

This field is inherited by subtypes.

descrsetfunc tp_descr_set
An optional pointer to a ”descriptor set” function.

XXX blah, blah.

This field is inherited by subtypes.

long tp_dictoffset
If the instances of this type have a dictionary containing instance variables, this field is non-zero and contains the offset in the instances of the type of the instance variable dictionary; this offset is used by PyObject_GenericGetAttr().

Do not confuse this field with tp_dict; that is the dictionary for attributes of the type object itself.

10.3. Type Objects 99
If the value of this field is greater than zero, it specifies the offset from the start of the instance structure. If the value is less than zero, it specifies the offset from the end of the instance structure. A negative offset is more expensive to use, and should only be used when the instance structure contains a variable-length part. This is used for example to add an instance variable dictionary to subtypes of `str` or `tuple`. Note that the `tp_basicsize` field should account for the dictionary added to the end in that case, even though the dictionary is not included in the basic object layout. On a system with a pointer size of 4 bytes, `tp_dictoffset` should be set to $-4$ to indicate that the dictionary is at the very end of the structure.

The real dictionary offset in an instance can be computed from a negative `tp_dictoffset` as follows:

\[
\text{dictoffset} = \text{tp_basicsize} + \text{abs}(\text{ob_size}) \times \text{tp_itemsize} + \text{tp_dictoffset}
\]

\[
\text{if \hspace{1em} dictoffset \hspace{1em} is \hspace{1em} not \hspace{1em} aligned \hspace{1em} on \hspace{1em} sizeof(void*):\n\hspace{1em} round \hspace{1em} up \hspace{1em} to \hspace{1em} sizeof(void*)}
\]

where `tp_basicsize`, `tp_itemsize` and `tp_dictoffset` are taken from the type object, and `ob_size` is taken from the instance. The absolute value is taken because long ints use the sign of `ob_size` to store the sign of the number. (There's never a need to do this calculation yourself; it is done for you by `PyObject_GetDictPtr`.)

This field is inherited by subtypes, but see the rules listed below. A subtype may override this offset; this means that the subtype instances store the dictionary at a difference offset than the base type. Since the dictionary is always found via `tp_dictoffset`, this should not be a problem.

When a type defined by a class statement has no `__slots__` declaration, and none of its base types has an instance variable dictionary, a dictionary slot is added to the instance layout and the `tp_dictoffset` is set to that slot's offset.

When a type defined by a class statement has a `__slots__` declaration, the type inherits its `tp_dictoffset` from its base type.

(Adding a slot named `__dict__` to the `__slots__` declaration does not have the expected effect, it just causes confusion. Maybe this should be added as a feature just like `__weakref__` though.)

**initproc tp_init**
An optional pointer to an instance initialization function.

This function corresponds to the `__init__()` method of classes. Like `__init__()`, it is possible to create an instance without calling `__init__()`, and it is possible to reinitialize an instance by calling its `__init__()` method again.

The function signature is

\[
\text{int \hspace{1em} tp\_init\hspace{1em}(PyObject \hspace{1em} *self, \hspace{1em} PyObject \hspace{1em} *args, \hspace{1em} PyObject \hspace{1em} *kwds)}
\]

The self argument is the instance to be initialized; the `args` and `kwds` arguments represent positional and keyword arguments of the call to `__init__()`. The `tp_init` function, if not NULL, is called when an instance is created normally by calling its type, after the type’s `tp_new` function has returned an instance of the type. If the `tp_new` function returns an instance of some other type that is not a subtype of the original type, no `tp_init` function is called; if `tp_new` returns an instance of a subtype of the original type, the subtype’s `tp_init` is called. (VERSION NOTE: described here is what is implemented in Python 2.2.1 and later. In Python 2.2, the `tp_init` of the type of the object returned by `tp_new` was always called, if not NULL.)

This field is inherited by subtypes.

**allocfunc tp_alloc**
An optional pointer to an instance allocation function.

The function signature is

\[
\text{PyObject \hspace{1em} *}\text{tp\_alloc\hspace{1em}(PyTypeObject \hspace{1em} *self, \hspace{1em} int \hspace{1em} nitems)}
\]

The purpose of this function is to separate memory allocation from memory initialization. It should return a pointer to a block of memory of adequate length for the instance, suitably aligned, and initialized to zeros, but with `ob_refcnt` set to 1 and `ob_type` set to the type argument. If the type’s
tp_itemsize is non-zero, the object's ob_size field should be initialized to nitems and the length of the allocated memory block should be tp Basicsize + nitems*tp Itemsize, rounded up to a multiple of sizeof(void*); otherwise, nitems is not used and the length of the block should be tp Basicsize.

Do not use this function to do any other instance initialization, not even to allocate additional memory; that should be done by tp_new.

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is always set to PyType_GenericAlloc(), to force a standard heap allocation strategy. That is also the recommended value for statically defined types.

newfunc tp_new
An optional pointer to an instance creation function.
If this function is NULL for a particular type, that type cannot be called to create new instances; presumably there is some other way to create instances, like a factory function.
The function signature is

    PyObject *tp_new(PyTypeObject *subtype, PyObject *args, PyObject *kwds)

The subtype argument is the type of the object being created; the args and kwds arguments represent positional and keyword arguments of the call to the type. Note that subtype doesn’t have to equal the type whose tp_new function is called; it may be a subtype of that type (but not an unrelated type).
The tp_new function should call subtype->tp_alloc(subtype, nitems) to allocate space for the object, and then do only as much further initialization as is absolutely necessary. Initialization that can safely be ignored or repeated should be placed in the tp_init handler. A good rule of thumb is that for immutable types, all initialization should take place in tp_new, while for mutable types, most initialization should be deferred to tp_init.

This field is inherited by subtypes, except it is not inherited by static types whose tp_base is NULL or &PyBaseObject_Type. The latter exception is a precaution so that old extension types don’t become callable simply by being linked with Python 2.2.

destructor tp_free
An optional pointer to an instance deallocation function.
The signature of this function has changed slightly: in Python 2.2 and 2.2.1, its signature is destructor:

    void tp_free(PyObject *)

In Python 2.3 and beyond, its signature is freefunc:

    void tp_free(void *)

The only initializer that is compatible with both versions is _PyObject_Del, whose definition has suitably adapted in Python 2.3.

This field is inherited by static subtypes, but not by dynamic subtypes (subtypes created by a class statement); in the latter, this field is set to a deallocator suitable to match PyType_GenericAlloc() and the value of the Py_TPFLAGS_HAVE_GC flag bit.

inquiry tp_is_gc
An optional pointer to a function called by the garbage collector.
The garbage collector needs to know whether a particular object is collectible or not. Normally, it is sufficient to look at the object's type's tp_flags field, and check the Py_TPFLAGS_HAVE_GC flag bit. But some types have a mixture of statically and dynamically allocated instances, and the statically allocated instances are not collectible. Such types should define this function; it should return 1 for a collectible instance, and 0 for a non-collectible instance. The signature is

    int tp_is_gc(PyObject *self)
(The only example of this are types themselves. The metatype, PyType_Type, defines this function to distinguish between statically and dynamically allocated types.)

This field is inherited by subtypes. (VERSION NOTE: in Python 2.2, it was not inherited. It is inherited in 2.2.1 and later versions.)

PyObject* tp_bases
TUPLE of base types.
This is set for types created by a class statement. It should be NULL for statically defined types.
This field is not inherited.

PyObject* tp_mro
TUPLE containing the expanded set of base types, starting with the type itself and ending with object, in Method Resolution Order.
This field is not inherited; it is calculated fresh by PyType_Ready().

PyObject* tp_cache
Unused. Not inherited. Internal use only.

PyObject* tp_subclasses
List of weak references to subclasses. Not inherited. Internal use only.

PyObject* tp_weaklist
Weak reference list head, for weak references to this type object. Not inherited. Internal use only.

The remaining fields are only defined if the feature test macro COUNT_ALLOCS is defined, and are for internal use only. They are documented here for completeness. None of these fields are inherited by subtypes.

int tp_allocs
Number of allocations.

int tp_frees
Number of frees.

int tp_maxalloc
Maximum simultaneously allocated objects.

PyTypeObject* tp_next
Pointer to the next type object with a non-zero tp_allocs field.

Also, note that, in a garbage collected Python, tp_dealloc may be called from any Python thread, not just the thread which created the object (if the object becomes part of a refcount cycle, that cycle might be collected by a garbage collection on any thread). This is not a problem for Python API calls, since the thread on which tp_dealloc is called will own the Global Interpreter Lock (GIL). However, if the object being destroyed in turn destroys objects from some other C or C++ library, care should be taken to ensure that destroying those objects on the thread which called tp_dealloc will not violate any assumptions of the library.

10.4 Mapping Object Structures

PyMappingMethods
Structure used to hold pointers to the functions used to implement the mapping protocol for an extension type.

10.5 Number Object Structures

PyNumberMethods
Structure used to hold pointers to the functions an extension type uses to implement the number protocol.
10.6 Sequence Object Structures

**PySequenceMethods**
Structure used to hold pointers to the functions which an object uses to implement the sequence protocol.

10.7 Buffer Object Structures

The buffer interface exports a model where an object can expose its internal data as a set of chunks of data, where each chunk is specified as a pointer/length pair. These chunks are called *segments* and are presumed to be non-contiguous in memory.

If an object does not export the buffer interface, then its *tp_as_buffer* member in the PyTypeObject structure should be NULL. Otherwise, the *tp_as_buffer* will point to a PyBufferProcs structure.

**Note:** It is very important that your PyTypeObject structure uses Py_TPFLAGS_DEFAULT for the value of the *tp_flags* member rather than 0. This tells the Python runtime that your PyBufferProcs structure contains the *bf_getcharbuffer* slot. Older versions of Python did not have this member, so a new Python interpreter using an old extension needs to be able to test for its presence before using it.

**PyBufferProcs**
Structure used to hold the function pointers which define an implementation of the buffer protocol.

- The first slot is *bf_getreadbuffer*, of type *getreadbufferproc*. If this slot is NULL, then the object does not support reading from the internal data. This is non-sensical, so implementors should fill this in, but callers should test that the slot contains a non-NULL value.

- The next slot is *bf_getwritebuffer* having type *getwritebufferproc*. This slot may be NULL if the object does not allow writing into its returned buffers.

- The third slot is *bf_getsegcount*, with type *getsegcountproc*. This slot must not be NULL and is used to inform the caller how many segments the object contains. Simple objects such as PyString_Type and PyBuffer_Type objects contain a single segment.

- The last slot is *bf_getcharbuffer*, of type *getcharbufferproc*. This slot will only be present if the Py_TPFLAGS_HAVE_GETCHARBUFFER flag is present in the *tp_flags* field of the object’s PyTypeObject. Before using this slot, the caller should test whether it is present by using the PyType_HasFeature() function. If present, it may be NULL, indicating that the object’s contents cannot be used as 8-bit characters. The slot function may also raise an error if the object’s contents cannot be interpreted as 8-bit characters. For example, if the object is an array which is configured to hold floating point values, an exception may be raised if a caller attempts to use *bf_getcharbuffer* to fetch a sequence of 8-bit characters. This notion of exporting the internal buffers as “text” is used to distinguish between objects that are binary in nature, and those which have character-based content.

**Note:** The current policy seems to state that these characters may be multi-byte characters. This implies that a buffer size of *N* does not mean there are *N* characters present.

**Py_TPFLAGS_HAVE_GETCHARBUFFER**
Flag bit set in the type structure to indicate that the *bf_getcharbuffer* slot is known. This being set does not indicate that the object supports the buffer interface or that the *bf_getcharbuffer* slot is non-NULL.

```
int (*getreadbufferproc) (PyObject *self, int segment, void **ptrptr)
```

Return a pointer to a readable segment of the buffer. This function is allowed to raise an exception, in which case it must return -1. The *segment* which is passed must be zero or positive, and strictly less than the number of segments returned by the *bf_getsegcount* slot function. On success, it returns the length of the buffer memory, and sets *ptrptr* to a pointer to that memory.

```
int (*getwritebufferproc) (PyObject *self, int segment, void **ptrptr)
```

Return a pointer to a writable memory buffer in *ptrptr*, and the length of that segment as the function return value. The memory buffer must correspond to buffer segment *segment*. Must return -1 and set an exception on error. TypeError should be raised if the object only supports read-only buffers, and SystemError should be raised when *segment* specifies a segment that doesn’t exist.
int (*getsegcountproc) (PyObject *self, int *lenp)
    Return the number of memory segments which comprise the buffer. If lenp is not NULL, the implementation
    must report the sum of the sizes (in bytes) of all segments in *lenp. The function cannot fail.

int (*getcharbufferproc) (PyObject *self, int segment, const char **ptrptr)
    Return the size of the memory buffer in ptrptr for segment segment. *ptrptr is set to the memory buffer.

10.8 Supporting the Iterator Protocol

10.9 Supporting Cyclic Garbage Collection

Python’s support for detecting and collecting garbage which involves circular references requires support from
object types which are “containers” for other objects which may also be containers. Types which do not store
references to other objects, or which only store references to atomic types (such as numbers or strings), do not
need to provide any explicit support for garbage collection.

An example showing the use of these interfaces can be found in “Supporting the Cycle Collector” in Extending
and Embedding the Python Interpreter.

To create a container type, the tp_flags field of the type object must include the Py_TPFLAGS_HAVE_GC
and provide an implementation of the tp_traverse handler. If instances of the type are mutable, a tp_clear
implementation must also be provided.

Py_TPFLAGS_HAVE_GC
    Objects with a type with this flag set must conform with the rules documented here. For convenience these
    objects will be referred to as container objects.

Constructors for container types must conform to two rules:

1. The memory for the object must be allocated using PyObject_GC_New() or
   PyObject_GC_VarNew().
2. Once all the fields which may contain references to other containers are initialized, it must call
   PyObject_GC_Track().

TYPE* PyObject_GC_New(TYPE, PyTypeObject *type)
    Analogous to PyObject_New() but for container objects with the Py_TPFLAGS_HAVE_GC flag set.

TYPE* PyObject_GC_NewVar(TYPE, PyTypeObject *type, int size)
    Analogous to PyObject_NewVar() but for container objects with the Py_TPFLAGS_HAVE_GC flag set.

PyVarObject * PyObject_GC_Resize(PyVarObject *op, int)
    Resize an object allocated by PyObject_NewVar(). Returns the resized object or NULL on failure.

void PyObject_GC_Track(PyObject *op)
    Adds the object op to the set of container objects tracked by the collector. The collector can run at unex-
    pected times so objects must be valid while being tracked. This should be called once all the fields followed
    by the tp_traverse handler become valid, usually near the end of the constructor.

void _PyObject_GC_TRACK(PyObject *op)
    A macro version of PyObject_GC_Track(). It should not be used for extension modules.

Similarly, the deallocator for the object must conform to a similar pair of rules:

1. Before fields which refer to other containers are invalidated, PyObject_GC_UnTrack() must be called.
2. The object’s memory must be deallocated using PyObject_GC_Del().

void PyObject_GC_Del(PyObject *op)
    Releases memory allocated to an object using PyObject_GC_New() or PyObject_GC_NewVar().

104 Chapter 10. Object Implementation Support
void *PyObject_GC_UnTrack (PyObject *op)

Remove the object op from the set of container objects tracked by the collector. Note that *PyObject_GC_Track() can be called again on this object to add it back to the set of tracked objects. The deallocator (tp_dealloc handler) should call this for the object before any of the fields used by the tp_traverse handler become invalid.

void _PyObject_GC_UNTRACK (PyObject *op)

A macro version of PyObject_GC_UnTrack(). It should not be used for extension modules.

The tp_traverse handler must have the following type:

int (*visitproc) (PyObject *object, void *arg)

Type of the visitor function passed to the tp_traverse handler. The function should be called with an object to traverse as object and the third parameter to the tp_traverse handler as arg. The Python core uses several visitor functions to implement cyclic garbage detection; it’s not expected that users will need to write their own visitor functions.

The tp_traverse handler must have the following type:

int (*traverseproc) (PyObject *self, visitproc visit, void *arg)

Traversal function for a container object. Implementations must call the visit function for each object directly contained by self, with the parameters to visit being the contained object and the arg value passed to the handler. The visit function must not be called with a NULL object argument. If visit returns a non-zero value that value should be returned immediately.

To simplify writing tp_traverse handlers, a Py_VISIT() macro is provided. In order to use this macro, the tp_traverse implementation must name its arguments exactly visit and arg:

void Py_VISIT (PyObject *o)

Call the visit callback, with arguments o and arg. If visit returns a non-zero value, then return it. Using this macro, tp_traverse handlers look like:

```c
static int
my_traverse (Noddy *self, visitproc visit, void *arg)
{
    Py_VISIT (self->foo);
    Py_VISIT (self->bar);
    return 0;
}
```

New in version 2.4.

The tp_clear handler must be of the inquiry type, or NULL if the object is immutable.

int (*inquiry) (PyObject *self)

Drop references that may have created reference cycles. Immutable objects do not have to define this method since they can never directly create reference cycles. Note that the object must still be valid after calling this method (don’t just call PyTuple_GETREF() on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.
Reporting Bugs

Python is a mature programming language which has established a reputation for stability. In order to maintain this reputation, the developers would like to know of any deficiencies you find in Python or its documentation.

Before submitting a report, you will be required to log into SourceForge; this will make it possible for the developers to contact you for additional information if needed. It is not possible to submit a bug report anonymously.

All bug reports should be submitted via the Python Bug Tracker on SourceForge (http://sourceforge.net/bugs/?group_id=5470). The bug tracker offers a Web form which allows pertinent information to be entered and submitted to the developers.

The first step in filing a report is to determine whether the problem has already been reported. The advantage in doing so, aside from saving the developers time, is that you learn what has been done to fix it; it may be that the problem has already been fixed for the next release, or additional information is needed (in which case you are welcome to provide it if you can!). To do this, search the bug database using the search box on the left side of the page.

If the problem you’re reporting is not already in the bug tracker, go back to the Python Bug Tracker (http://sourceforge.net/bugs/?group_id=5470). Select the “Submit a Bug” link at the top of the page to open the bug reporting form.

The submission form has a number of fields. The only fields that are required are the “Summary” and “Details” fields. For the summary, enter a very short description of the problem; less than ten words is good. In the Details field, describe the problem in detail, including what you expected to happen and what did happen. Be sure to include the version of Python you used, whether any extension modules were involved, and what hardware and software platform you were using (including version information as appropriate).

The only other field that you may want to set is the “Category” field, which allows you to place the bug report into a broad category (such as “Documentation” or “Library”).

Each bug report will be assigned to a developer who will determine what needs to be done to correct the problem. You will receive an update each time action is taken on the bug.

See Also:

How to Report Bugs Effectively
(http://www-mice.cs.ucl.ac.uk/multimedia/software/documentation/ReportingBugs.html)

Article which goes into some detail about how to create a useful bug report. This describes what kind of information is useful and why it is useful.

Bug Writing Guidelines
(http://www.mozilla.org/quality/bug-writing-guidelines.html)

Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.
## History and License

### B.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see [http://www.cwi.nl/](http://www.cwi.nl/)) in the Netherlands as a successor of a language called ABC. Guido remains Python’s principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see [http://www.cnri.reston.va.us/](http://www.cnri.reston.va.us/)) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen Python-Labs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see [http://www.zope.com/](http://www.zope.com/)). In 2001, the Python Software Foundation (PSF, see [http://www.python.org/psf/](http://www.python.org/psf/)) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see [http://www.opensource.org/](http://www.opensource.org/) for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

<table>
<thead>
<tr>
<th>Release</th>
<th>Derived from</th>
<th>Year</th>
<th>Owner</th>
<th>GPL compatible?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9.0 thru 1.2</td>
<td>n/a</td>
<td>1991-1995</td>
<td>CWI</td>
<td>yes</td>
</tr>
<tr>
<td>1.3 thru 1.5.2</td>
<td>1.2</td>
<td>1995-1999</td>
<td>CNRI</td>
<td>yes</td>
</tr>
<tr>
<td>1.6</td>
<td>1.5.2</td>
<td>2000</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.0</td>
<td>1.6</td>
<td>2000</td>
<td>BeOpen.com</td>
<td>no</td>
</tr>
<tr>
<td>1.6.1</td>
<td>1.6</td>
<td>2001</td>
<td>CNRI</td>
<td>no</td>
</tr>
<tr>
<td>2.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.0.1</td>
<td>2.0+1.6.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.1</td>
<td>2.1+2.0.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2</td>
<td>2.1.1</td>
<td>2001</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.2</td>
<td>2.1.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.1.3</td>
<td>2.1.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.1</td>
<td>2.2</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.2</td>
<td>2.2.1</td>
<td>2002</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.2.3</td>
<td>2.2.2</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3</td>
<td>2.2.2</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.1</td>
<td>2.3</td>
<td>2002-2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.2</td>
<td>2.3.1</td>
<td>2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.3</td>
<td>2.3.2</td>
<td>2003</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.4</td>
<td>2.3.3</td>
<td>2004</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.3.5</td>
<td>2.3.4</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4</td>
<td>2.3</td>
<td>2004</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.1</td>
<td>2.4</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.2</td>
<td>2.4.1</td>
<td>2005</td>
<td>PSF</td>
<td>yes</td>
</tr>
<tr>
<td>2.4.3</td>
<td>2.4.2</td>
<td>2006</td>
<td>PSF</td>
<td>yes</td>
</tr>
</tbody>
</table>
Note: GPL-compatible doesn’t mean that we’re distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don’t.

Thanks to the many outside volunteers who have worked under Guido’s direction to make these releases possible.

B.2 Terms and conditions for accessing or otherwise using Python

**PSF LICENSE AGREEMENT FOR PYTHON 2.4.3**

1. This LICENSE AGREEMENT is between the Python Software Foundation (“PSF”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 2.4.3 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.4.3 alone or in any derivative version, provided, however, that PSF’s License Agreement and PSF’s notice of copyright, i.e., “Copyright © 2001-2006 Python Software Foundation; All Rights Reserved” are retained in Python 2.4.3 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.4.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.4.3.

4. PSF is making Python 2.4.3 available to Licensee on an “AS IS” basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.4.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.4.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.4.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.4.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

**BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0**

**BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1**

1. This LICENSE AGREEMENT is between BeOpen.com (“BeOpen”), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization (“Licensee”) accessing and otherwise using this software in source or binary form and its associated documentation (“the Software”).

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an “AS IS” basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT
LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the “BeOpen Python” logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 (“CNRI”), and the Individual or Organization (“Licensee”) accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI’s License Agreement and CNRI’s notice of copyright, i.e., “Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved” are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI’s License Agreement, Licensee may substitute the following text (omitting the quotes): “Python 1.6.1 is made available subject to the terms and conditions in CNRI’s License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013.”

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an “AS IS” basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia’s conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable
material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the “ACCEPT” button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. All rights reserved.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATHISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

B.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

B.3.1 Mersenne Twister

The _random module includes code based on a download from http://www.math.keio.ac.jp/~matumoto/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.

3. The names of its contributors may not be used to endorse or promote
   products derived from this software without specific prior written
   permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Any feedback is very welcome.
http://www.math.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp

B.3.2  Sockets

The socket module uses the functions, getaddrinfo, and getnameinfo, which are coded in separate
B.3.3 Floating point exception control

The source for the fpectl module includes the following notice:
B.3.4 MD5 message digest algorithm

The source code for the md5 module contains the following notice:
Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All rights reserved.

License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing this software or this function.

License is also granted to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this documentation and/or software.

B.3.5 Asynchronous socket services

The asynchat and asyncore modules contain the following notice:

Copyright 1996 by Sam Rushing

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Sam Rushing not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

B.3.6 Cookie management

The Cookie module contains the following notice:
Copyright 2000 by Timothy O’Malley <timo@alum.mit.edu>

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Timothy O’Malley not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Timothy O’Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL Timothy O’Malley BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

B.3.7 Profiling

The profile and pstats modules contain the following notice:

Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software and its associated documentation for any purpose (subject to the restriction in the following sentence) without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of InfoSeek not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. This permission is explicitly restricted to the copying and modification of the software to remain in Python, compiled Python, or other languages (such as C) wherein the modified or derived code is exclusively imported into a Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

B.3.8 Execution tracing

The trace module contains the following notice:
B.3.9 UUencode and UUdecode functions

The uu module contains the following notice:

Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appears in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with python standard
The `xmlrpclib` module contains the following notice:

The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its associated documentation, you agree that you have read, understood, and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and its associated documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appears in all copies, and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Secret Labs AB or the author not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
Symbols

(PyImport_FindExtension(), 23
(PyImport_Fini(), 23
(PyImport_FixupExtension(), 23
(PyImport_Init(), 23
(PyObject_Del(), 87
(PyObject_GC_TRACK(), 104
(PyObject_GC_UNTRACK(), 105
(PyObject_New(), 87
(PyObject_NewVar(), 87
(PyString_Resize(), 50
(PyTuple_Resize(), 60
(PyNoneStruct, 88
(PyCdiff(), 47
(PyCneg(), 47
(PyCpow(), 48
(PyCprod(), 48
(PyCquot(), 48
(PyCsum(), 47
__all__ (package variable), 22
__builtin__ (built-in module), 7, 73
__dict__ (module attribute), 66
__doc__ (module attribute), 66
__file__ (module attribute), 66
__import__ () (built-in function), 22
__main__ (built-in module), 7, 73
__name__ (module attribute), 66
_ob_next (PyObject member), 92
_ob_prev (PyObject member), 92

A

abort(), 21
abs() (built-in function), 35
apply() (built-in function), 33
argv (in module sys), 76

B

buffer
  object, 58
buffer interface, 58
BufferType (in module types), 58

C

calloc(), 83
classmethod() (built-in function), 90
cleanup functions, 22
close() (in module os), 74
cmp() (built-in function), 32
CO_FUTURE_DIVISION, 11
CObject
  object, 69
code() (built-in function), 37
compile() (built-in function), 23
complex number
  object, 47
copyright (in module sys), 75

D

dictionary
  object, 61
DictionaryType (in module types), 61
DictType (in module types), 61
divmod() (built-in function), 35

e

evironment variables
  PATH, 8
  PYTHONDUMPREFS, 92
  PYTHONHOME, 8
  PYTHONPATH, 8
  exec_prefix, 1, 2
  prefix, 1, 2
EOFError (built-in exception), 64
erno, 77
exc_info() (in module sys), 6
exc_traceback (in module sys), 6, 15
exc_type (in module sys), 6, 15
exc_value (in module sys), 6, 15
Exception (built-in exception), 19
exceptions (built-in module), 7
exec_prefix, 1, 2
executable (in module sys), 75
exit(), 22

F

file
  object, 63
FileType (in module types), 64
float() (built-in function), 37
floating point
  object, 47
FloatType (in modules types), 47
fopen(), 64
free(), 83
freeze utility, 23
G
global interpreter lock, 76
H
hash() (built-in function), 33, 95
I
ihooks (standard module), 22
incr_item(), 6, 7
instance
object, 64
int() (built-in function), 37
getcharbufferproc (C type), 104
getreadbufferproc (C type), 103
getsegcountproc (C type), 104
getwritebufferproc (C type), 103
inquiry (C type), 105
Py_tracefunc (C type), 80
traverseproc (C type), 105
integer
object, 44
interpreter lock, 76
IntType (in modules types), 44
K
KeyboardInterrupt (built-in exception), 18
L
len() (built-in function), 34, 37, 39, 60, 62
list
object, 60
ListType (in module types), 60
lock, interpreter, 76
long() (built-in function), 37
long integer
object, 45
LONG_MAX, 45, 46
LongType (in modules types), 45
M
main(), 74, 76
malloc(), 83
mapping
object, 61
METH_CLASS (data in ), 90
METH_COEXIST (data in ), 90
METH_KEYWORDS (data in ), 89
METH_NOARGS (data in ), 90
METH_O (data in ), 90
METH_OLDARGS (data in ), 90
METH_STATIC (data in ), 90
METH_VARARGS (data in ), 89
method
object, 65
MethodType (in module types), 65
module
object, 65
search path, 8, 73, 75
modules (in module sys), 22, 73
ModuleType (in module types), 66
N
None
object, 44
numeric
object, 44
O
ob_refcnt (PyObject member), 92
ob_size (PyObject member), 92
ob_type (PyObject member), 92
object
buffer, 58
CObject, 69
complex number, 47
dictionary, 61
file, 63
floating point, 47
instance, 64
integer, 44
list, 60
long integer, 45
mapping, 61
method, 65
module, 65
None, 44
numeric, 44
sequence, 48
string, 48
tuple, 59
type, 2, 43
OverflowError (built-in exception), 46
P
package variable
__all__, 22
PATH, 8
path
module search, 8, 73, 75
path (in module sys), 8, 73, 75
platform (in module sys), 75
pow() (built-in function), 35, 36
prefix, 1, 2
Py_AtExit(), 22
Py_BEGIN_ALLOW_THREADS, 76
Py_BEGIN_ALLOW_THREADS (macro), 79
Py_BLOCK_THREADS (macro), 79
Py_BuildValue(), 29
Py_CLEAR(), 13
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyCallIter_Type</td>
<td>66</td>
</tr>
<tr>
<td>PyCell_Check</td>
<td>69</td>
</tr>
<tr>
<td>PyCell_GET</td>
<td>69</td>
</tr>
<tr>
<td>PyCell_Get</td>
<td>69</td>
</tr>
<tr>
<td>PyCell_New</td>
<td>69</td>
</tr>
<tr>
<td>PyCell_SET</td>
<td>70</td>
</tr>
<tr>
<td>PyCell_Set</td>
<td>69</td>
</tr>
<tr>
<td>PyCell_Type</td>
<td>69</td>
</tr>
<tr>
<td>PyCellObject (C type)</td>
<td>69</td>
</tr>
<tr>
<td>PyCFObject (C type)</td>
<td>89</td>
</tr>
<tr>
<td>PyCObject (C type)</td>
<td>69</td>
</tr>
<tr>
<td>PyCObject_AsVoidPtr</td>
<td>69</td>
</tr>
<tr>
<td>PyCObject_Check</td>
<td>69</td>
</tr>
<tr>
<td>PyCObject_FromVoidPtr</td>
<td>69</td>
</tr>
<tr>
<td>PyCObject_FromVoidPtrAndDesc</td>
<td>69</td>
</tr>
<tr>
<td>PyCObject_GetDesc</td>
<td>69</td>
</tr>
<tr>
<td>PyCObject_SetVoidPtr</td>
<td>69</td>
</tr>
<tr>
<td>PyComplex_AsCComplex</td>
<td>48</td>
</tr>
<tr>
<td>PyComplex_Check</td>
<td>48</td>
</tr>
<tr>
<td>PyComplex_CheckExact</td>
<td>48</td>
</tr>
<tr>
<td>PyComplex_FromCComplex</td>
<td>48</td>
</tr>
<tr>
<td>PyComplex_FromDoubles</td>
<td>48</td>
</tr>
<tr>
<td>PyComplex_RealAsDouble</td>
<td>48</td>
</tr>
<tr>
<td>PyComplex_Type</td>
<td>48</td>
</tr>
<tr>
<td>PyComplexObject (C type)</td>
<td>48</td>
</tr>
<tr>
<td>PyDate_Check</td>
<td>70</td>
</tr>
<tr>
<td>PyDate_CheckExact</td>
<td>70</td>
</tr>
<tr>
<td>PyDate_FromDate</td>
<td>71</td>
</tr>
<tr>
<td>PyDate_FromTimestamp</td>
<td>72</td>
</tr>
<tr>
<td>PyDateTime_DateTime_GET_HOUR</td>
<td>71</td>
</tr>
<tr>
<td>PyDateTime_DateTime_GET_MICROSECOND</td>
<td>71</td>
</tr>
<tr>
<td>PyDateTime_DateTime_GET_MINUTE</td>
<td>71</td>
</tr>
<tr>
<td>PyDateTime_DateTime_GET_SECOND</td>
<td>71</td>
</tr>
<tr>
<td>PyDateTime_DateTime_FROM_DATE_AND_TIME</td>
<td>71</td>
</tr>
<tr>
<td>PyDateTime_DateTime_FROM_TIMESTAMP</td>
<td>71</td>
</tr>
<tr>
<td>PyDateTime_GET_DAY</td>
<td>71</td>
</tr>
<tr>
<td>PyDateTime_GET_MONTH</td>
<td>71</td>
</tr>
<tr>
<td>PyDateTime_GET_YEAR</td>
<td>71</td>
</tr>
<tr>
<td>PyDateTime_TIME_TIME_GET_HOUR</td>
<td>71</td>
</tr>
<tr>
<td>PyDateTime_TIME_TIME_GET_MICROSECOND</td>
<td>71</td>
</tr>
<tr>
<td>PyDateTime_TIME_TIME_GET_MINUTE</td>
<td>71</td>
</tr>
<tr>
<td>PyDateTime_TIME_TIME_GET_SECOND</td>
<td>71</td>
</tr>
<tr>
<td>PyDelta_Check</td>
<td>70</td>
</tr>
<tr>
<td>PyDelta_CheckExact</td>
<td>70</td>
</tr>
<tr>
<td>PyDelta_FromDSU</td>
<td>71</td>
</tr>
<tr>
<td>PyDescr_IsData</td>
<td>67</td>
</tr>
<tr>
<td>PyDescr_NewClassMethod</td>
<td>67</td>
</tr>
<tr>
<td>PyDescr_NewGetSet</td>
<td>67</td>
</tr>
<tr>
<td>PyDescr_NewMember</td>
<td>67</td>
</tr>
<tr>
<td>PyDescr_NewMethod</td>
<td>67</td>
</tr>
<tr>
<td>PyDescr_NewWrapper</td>
<td>67</td>
</tr>
<tr>
<td>PyDict_Check</td>
<td>61</td>
</tr>
<tr>
<td>PyDict_CheckExact</td>
<td>61</td>
</tr>
<tr>
<td>PyDict_Clear</td>
<td>61</td>
</tr>
<tr>
<td>PyDictContains</td>
<td>62</td>
</tr>
<tr>
<td>PyDict_Copy</td>
<td>62</td>
</tr>
<tr>
<td>PyDict_DelItem</td>
<td>62</td>
</tr>
<tr>
<td>PyDict_DelItemString</td>
<td>62</td>
</tr>
<tr>
<td>PyDict_GetItem</td>
<td>62</td>
</tr>
<tr>
<td>PyDict_GetItemString</td>
<td>62</td>
</tr>
<tr>
<td>PyDict_Items</td>
<td>62</td>
</tr>
<tr>
<td>PyDict_Size</td>
<td>62</td>
</tr>
<tr>
<td>PyDict_Type</td>
<td>61</td>
</tr>
<tr>
<td>PyDict_Update</td>
<td>63</td>
</tr>
<tr>
<td>PyDictValues</td>
<td>62</td>
</tr>
<tr>
<td>PyDictObject (C type)</td>
<td>61</td>
</tr>
<tr>
<td>PyDictProxy_New</td>
<td>61</td>
</tr>
<tr>
<td>PyErr_BadArgument</td>
<td>16</td>
</tr>
<tr>
<td>PyErr_BadInternalCall</td>
<td>17</td>
</tr>
<tr>
<td>PyErr_CheckSignals</td>
<td>17</td>
</tr>
<tr>
<td>PyErr_Clear</td>
<td>15</td>
</tr>
<tr>
<td>PyErr_Cleared</td>
<td>6, 7</td>
</tr>
<tr>
<td>PyErr_ExceptionMatches</td>
<td>15</td>
</tr>
<tr>
<td>PyErr_ExceptionMatchesString</td>
<td>7</td>
</tr>
<tr>
<td>PyErr_Fetch</td>
<td>16</td>
</tr>
<tr>
<td>PyErr_Format</td>
<td>16</td>
</tr>
<tr>
<td>PyErr_GivenExceptionMatches</td>
<td>15</td>
</tr>
<tr>
<td>PyErr_NewException</td>
<td>18</td>
</tr>
<tr>
<td>PyErr_NoMemory</td>
<td>16</td>
</tr>
<tr>
<td>PyErr_NormalizeException</td>
<td>15</td>
</tr>
<tr>
<td>PyErr_Occurred</td>
<td>15</td>
</tr>
<tr>
<td>PyErr_OccurredString</td>
<td>6</td>
</tr>
<tr>
<td>PyErr_Print</td>
<td>15</td>
</tr>
<tr>
<td>PyErr_Restore</td>
<td>16</td>
</tr>
<tr>
<td>PyErr_SetExcFromWindowsErr</td>
<td>17</td>
</tr>
<tr>
<td>PyErr_SetExcFromWindowsErrWithFilename</td>
<td>17</td>
</tr>
<tr>
<td>PyErr_SetFromErrno</td>
<td>17</td>
</tr>
<tr>
<td>PyErr_SetFromErrnoWithFilename</td>
<td>17</td>
</tr>
<tr>
<td>PyErr_SetFromWindowsErr</td>
<td>17</td>
</tr>
<tr>
<td>PyErr_SetFromWindowsErrWithFilename</td>
<td>17</td>
</tr>
<tr>
<td>PyErr_SetInterrupt</td>
<td>18</td>
</tr>
<tr>
<td>PyErr_SetNone</td>
<td>16</td>
</tr>
<tr>
<td>PyErr_SetObject</td>
<td>16</td>
</tr>
<tr>
<td>PyErr_SetString</td>
<td>16</td>
</tr>
<tr>
<td>PyErr_WriteUnraisable</td>
<td>18</td>
</tr>
<tr>
<td>PyEval_AcquireLock</td>
<td>78</td>
</tr>
<tr>
<td>PyEval_AcquireLock</td>
<td>73, 77</td>
</tr>
<tr>
<td>PyEval_AcquireThread</td>
<td>78</td>
</tr>
<tr>
<td>PyEval_InitThreads</td>
<td>78</td>
</tr>
</tbody>
</table>
PyList_New(), 60
PyList_Reverse(), 61
PyList_SET_ITEM(), 61
PyList_SetItem(), 60
PyList_SetItem(), 3
PyList_SetSlice(), 61
PyList_Size(), 60
PyList_Sort(), 61
PyList_Type(), 60
PyListObject (C type), 60
PyLong_AsDouble(), 46
PyLong_AsLong(), 46
PyLong_AsLongLong(), 46
PyLong_AsUnsignedLong(), 45
PyLong_AsUnsignedLongLong(), 46
PyLong_AsUnsignedLongLongMask(), 46
PyLong_AsUnsignedLongMask(), 46
PyLong_AsVoidPtr(), 46
PyLong_Check(), 45
PyLong_CheckExact(), 45
PyLong_FromDouble(), 46
PyLong_FromLong(), 45
PyLong_FromLongLong(), 46
PyLong_FromString(), 46
PyLong_FromUnicode(), 46
PyLong_FromUnsignedLong(), 45
PyLong_FromUnsignedLongLong(), 46
PyLong_FromVoidPtr(), 46
PyLong_Type(), 45
PyLongObject (C type), 45
PyMapping_Check(), 39
PyMapping_DelItem(), 39
PyMapping_DelItemString(), 39
PyMapping_GetItemString(), 39
PyMapping_HasKey(), 39
PyMapping_HasKeyString(), 39
PyMapping_Items(), 39
PyMapping_Keys(), 39
PyMapping_LENGTH(), 39
PyMapping_SetItemString(), 39
PyMapping_Values(), 39
PyMappingMethods (C type), 102
PyMarshal_ReadLastErrorObjectFromFile(), 25
PyMarshal_ReadLastObjectFromFile(), 25
PyMarshal_ReadLongFromFile(), 24
PyMarshal_ReadObjectFromFile(), 24
PyMarshal_ReadObjectFromFileString(), 25
PyMarshal_ReadShortFromFile(), 24
PyMarshal_WriteLongToFile(), 24
PyMarshal_WriteObjectToFile(), 24
PyMarshal_WriteObjectToString(), 24
PyMem_Del(), 84
PyMem_Free(), 84
PyMem_Malloc(), 84
PyMem_New(), 84
PyMem_Realloc(), 84
PyMem_Resize(), 84
PyMethod_Check(), 65
PyMethod_Class(), 65
PyMethod_Function(), 65
PyMethod_GET_CLASS(), 65
PyMethod_GET_FUNCTION(), 65
PyMethod_GET_SELF(), 65
PyMethod_New(), 65
PyMethod_Type(), 65
PyMethodDef (C type), 89
PyModule_AddIntConstant(), 66
PyModule_AddObject(), 66
PyModule_AddStringConstant(), 66
PyModule_Check(), 66
PyModule_CheckExact(), 66
PyModule_GetDict(), 66
PyModule_GetFilename(), 66
PyModuleGetName(), 66
PyModule_New(), 66
PyModule_Type(), 66
PyNumber.Absolute(), 35
PyNumber_Add(), 34
PyNumber_And(), 35
PyNumber_Check(), 34
PyNumber_Coerce(), 37
PyNumber_Divide(), 34
PyNumber_Divmod(), 35
PyNumber_Divmod(), 35
PyNumber_Float(), 37
PyNumber_FloorDivide(), 35
PyNumber_InPlaceAdd(), 36
PyNumber_InPlaceAnd(), 36
PyNumber_InPlaceDivide(), 36
PyNumber_InPlaceFloorDivide(), 36
PyNumber_InPlaceLShift(), 36
PyNumber_InPlaceMultiply(), 36
PyNumber_InPlaceOr(), 37
PyNumber_InPlacePower(), 36
PyNumber_InPlaceRemainder(), 36
PyNumber_InPlaceRShift(), 36
PyNumber_InPlaceSubtract(), 36
PyNumber_InPlaceTrueDivide(), 36
PyNumber_InPlaceXor(), 37
PyNumber_Int(), 37
PyNumber_Invert(), 35
PyNumber_Long(), 37
PyNumber_LShift(), 35
PyNumber_Multiply(), 34
PyNumber_Negative(), 35
PyNumber_Not(), 36
PyNumber_Positive(), 35
PyNumber_Power(), 35
PyNumber_Remainder(), 35
PyNumber_RShift(), 35
PyNumber_Subtract(), 34
PyNumber_TrueDivide(), 35
PyNumber_Xor(), 35
PyNumberMethods (C type), 102
PyObject (C type), 88
PyObject_AsCharBuffer(), 40
tp_subclasses (PyTypeObject member), 102
tp_traverse (PyTypeObject member), 97
tp_weaklist (PyTypeObject member), 102
tp_weaklistoffset (PyTypeObject member), 98

tuple
    object, 59
tuple() (built-in function), 38, 61
TupleType (in module types), 59
type
    object, 2, 43
type() (built-in function), 33
TypeType (in module types), 43

U
ULONG_MAX, 46
unicode() (built-in function), 32

V
version (in module sys), 75, 76