BEOPEN.COM TERMS AND CONDITIONS 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 OPEN SOURCE GPL-COMPATIBLE LICENSE AGREEMENT

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.

CWI PERMISSIONS STATEMENT AND DISCLAIMER

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 MATHEMATISCH 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.
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 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 ................................................. 11

4 Exception Handling ............................................... 13  
4.1 Standard Exceptions ........................................... 16  
4.2 Deprecation of String Exceptions ............................. 16

5 Utilities ............................................................. 17  
5.1 OS Utilities ....................................................... 17  
5.2 Process Control .................................................. 17  
5.3 Importing Modules .............................................. 18

6 Abstract Objects Layer ............................................ 21  
6.1 Object Protocol .................................................. 21  
6.2 Number Protocol ............................................... 23  
6.3 Sequence Protocol ............................................. 26  
6.4 Mapping Protocol ............................................... 28

7 Concrete Objects Layer ............................................ 29  
7.1 Fundamental Objects .......................................... 29  
7.2 Sequence Objects .............................................. 30  
7.3 Mapping Objects ................................................ 42  
7.4 Numeric Objects ............................................... 43  
7.5 Other Objects ................................................... 46

8 Initialization, Finalization, and Threads ....................... 51  
8.1 Thread State and the Global Interpreter Lock ............... 54

9 Memory Management .............................................. 59  
9.1 Overview ......................................................... 59  
9.2 Memory Interface ............................................... 60  
9.3 Examples ......................................................... 60
CHAPTER
ONE

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 that 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:

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

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

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 <python2.1/Python.h>’; this will break on
multi-platform builds since the platform independent headers under prefix include the platform specific headers from exec_prefix.

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_INCREF()` to increment an object’s reference count by one, and `Py_DECREF()` to decrement it by one. The `Py_DECREF()` macro is considerably more complex than the `inref` 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 `Py_DECREF()`, 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. Note that we talk of owning references, never of owning objects; objects are always shared! When a function owns a reference, it has to dispose of it properly — either by passing ownership on (usually to its caller) or by calling `Py_DECREF()` or `Py_XDECREF()`.

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 a reference to an object, there are two possibilities: the function steals a reference to the object, or it does not. 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"));
```

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);
    x = PyInt_FromLong(1L);
    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):
PyObject *t, *l;

    t = Py_BuildValue("(iis)", 1, 2, "three");
    l = Py_BuildValue("[iis]", 1, 2, "three");

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 (i.e., 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 (i.e., the type of 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 `__builtin__`, `__main__` and `sys`. 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/python2.1’ 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/python2.1’. (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 by setting PYTHONPATH.

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 PyRun:AnyFile(FILE *fp, char *filename)

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(char *command)

Executes the Python source code from command in the __main__ module. 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.
```

```c
int PyRun:SimpleFile(FILE *fp, char *filename)

Similar to PyRun:SimpleString(), but the Python source code is read from fp instead of an in-memory string. filename should be the name of the file.
```

```c
int PyRun:InteractiveOne(FILE *fp, char *filename)

Read and execute a single statement from a file associated with an interactive device. 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 in case of a parse error. (Note that ‘errcode.h’ is not included by ‘Python.h’, so
must be included specifically if needed.)
```

```c
int PyRun:InteractiveLoop(FILE *fp, char *filename)

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.
```

```c
struct _node* PyParser:SimpleParseString(char *str, int start)

Parse Python source code from str using the start token start. 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.
```

```c
struct _node* PyParser:SimpleParseFile(FILE *fp, char *filename, int start)

Similar to PyParser:SimpleParseString(), but the Python source code is read from fp instead of an
in-memory string. filename should be the name of the file.
```
PyObject* **PyRun_String**(char *str, int start, PyObject *globals, PyObject *locals)

*Return value: New reference.*

Execute Python source code from *str* in the context specified by the dictionaries *globals* and *locals*. 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, char *filename, int start, PyObject *globals, PyObject *locals)

*Return value: New reference.*

Similar to **PyRun_String()**, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file.

PyObject* **Py_CompileString**(char *str, char *filename, int start)

*Return value: New reference.*

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.

int **Py_eval_input**

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

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 **Py_CompileString()**. This is the symbol to use when compiling arbitrarily long Python source code.

int **Py_single_input**

The start symbol from the Python grammar for a single statement; for use with **Py_CompileString()**. This is the symbol used for the interactive interpreter loop.
Reference Counting

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

```c
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 () .
```

```c
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.
```

```c
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.
```

```c
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.
```

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_Parse*() 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.

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.

```c
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!)
}
```

```c
PyObject* PyErr_Occurred()
{
    Return value: Borrowed reference.
    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 the subclass of the expected exception.)
}
```

```c
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.
}
```

```c
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.
}
```

```c
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.
}
```

```c
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-NULL value or traceback. The exception type should be a string or class; if it is a class, the value should be an instance of that 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, i.e. you must own 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.

```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. You need not increment its reference count.

```c
PyObject* PyErr_Format(PyObject *exception, const char *format, ...)
```

**Return value:** Always NULL.

This function sets the error indicator. `exception` should be a Python exception (string or 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>‘x’</td>
<td>A string, as a char * 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.

A new reference is returned, which is owned by the caller.

```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.
A wrapper function around a system call can write `return PyErr_SetFromErrno();` when the system call returns an error.

```c
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.

```c
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 (e.g. `Py_DecREF` 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`, and `PyExc_RuntimeWarning`. `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.

```c
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.

```c
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.

```c
void PyErr_SetInterrupt()
```

This function is obsolete. It 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.

```c
PyObject* PyErr_NewException (char *name, PyObject *base, PyObject *dict)
```

*Return value: New reference.*

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.

```c
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
The function is called with a single argument `obj` that identifies where 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_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></td>
</tr>
<tr>
<td>PyExc_ZeroDivisionError</td>
<td>ZeroDivisionError</td>
<td>(2)</td>
</tr>
</tbody>
</table>

Notes:

1. This is a base class for other standard exceptions.
2. 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, such as parsing function arguments and constructing Python values from C values.

### 5.1 OS Utilities

**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 `'???'`.

**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()`.

**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.

**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 and on the Macintosh). `USE_CHECKSTACK` will be defined automatically; you should never change the definition in your own code.

**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)`.

**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

**void Py_FatalError(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.

```c
void Py_Exit(int status)
    Exit the current process. This calls Py_Finalize() and then calls the standard C library function `exit(status)`.
```

```c
int Py_AtExit(void (*func)())
    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

```c
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 (i.e., 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 (the module may still be created in this case — examine sys.modules to find out).
```

```c
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 (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.
```

```c
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.
```

```c
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).
```

```c
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 in in the modules dictionary. Warning: 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. Return NULL with an exception set on failure.
```
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 compile(), load the module. Return a new reference to the module object, or NULL with an exception set if an error occurred (the module may still be created in this case). (This function would reload the module if it was already imported.)

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. 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 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 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 freeze utility (see ‘Tools/freeze’ in the Python source distribution). Its definition, found in ‘Include/import.h’, is:

```
struct _frozen {
    char *name;
    unsigned char *code;
    int size;
};
```

struct _frozen* PyImport_FrozenModules

This pointer is initialized to point to an array of 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 PyImport_ExtendInittab(), returning -1 if the table could not be extended. The new module can be imported by the name name, and uses the function initfunc as the initialization function called on the first attempted import. This should be called before Py_Initialize().

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().
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

```c
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().
```

```c
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.
```

```c
PyObject* PyObject_GetAttrString (PyObject *o, char *attr_name)
    Return value: New reference. 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'.
```

```c
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.
```

```c
PyObject* PyObject_GetAttr (PyObject *o, PyObject *attr_name)
    Return value: New reference. 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'.
```

```c
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'.
```

```c
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'.
```

```c
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'.
```

```c
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`.

```c
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)`.
```

```c
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 success, NULL on failure. This is the equivalent of the Python expression `unistr(o)`.
Called by the `unistr()` built-in function.

```c
int PyObject_IsInstance (PyObject *inst, PyObject *cls)
    Return 1 if inst is an instance of the class cls or a subclass of cls. If cls is a type object rather than a class object, PyObject_IsInstance() returns 1 if inst is of type cls. If inst is not a class instance and cls is neither a type object or class object, 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.
```

Subclass determination is done in a fairly straightforward way, but includes a wrinkle that implementors of extensions 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.

```c
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 either derived or cls is not an actual class object, this function uses the generic algorithm described above. New in version 2.1.
```

```c
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_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)`.

PyObject* PyObject_CallFunction (PyObject *callable_object, char *format, ...)
Call a callable Python object callable_object, 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_object, args)`.

```c
PyObject* PyObject_CallMethod(PyObject *o, char *method, char *format, ...)  
Call the method named `m` of object `o` with a variable number of C arguments. The C arguments are described by a `Py_BuildValue()` 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 `o.method (args)`. Note that special method names, such as `__add__`, `__getitem__`, and so on are not supported. The specific abstract-object routines for these must be used.
```

```c
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)`.
```

```c
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`. This function always succeeds.
```

```c
PyObject* PyObject_Type(PyObject *o)  
On success, returns a type object corresponding to the object type of object `o`. On failure, returns `NULL`. This is equivalent to the Python expression `type(o)`.
```

```c
int PyObject_Length(PyObject *o)  
Return the length of object `o`. If the object `o` provides both sequence and mapping protocols, the sequence length is returned. On error, `-1` is returned. This is the equivalent to the Python expression `len(o)`.
```

```c
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]`.
```

```c
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`.
```

```c
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]`.
```

```c
int PyObject_AsFileDescriptor(PyObject *o)  
Derives a file-descriptor from a Python object. If the object is an integer or long integer, its value is returned. If not, the object’s `fileno()` method is called if it exists; the method must return an integer or long integer, which is returned as the file descriptor value. Returns `-1` on failure.
```

### 6.2 Number Protocol

```c
int PyNumber_Check(PyObject *o)  
Returns 1 if the object `o` provides numeric protocols, and false otherwise. This function always succeeds.
```

```c
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_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 o2 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)
Returns the “bitwise exclusive or” of o1 by o2 on success, or NULL on failure. This is the equivalent of the Python expression ‘o1 ^ 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 expression ‘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 expression ‘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 expression ‘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 expression ‘o1 /= o2’.

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 expression ‘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 expression ‘o1 **= o2’ when o3 is Py_None, or an in-place variant of ‘pow(o1, o2, o3)’ otherwise. If o3 is to be ignored, pass Py_None 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 expression ‘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 expression ‘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 expression ‘o1 &= o2’.

6.2. Number Protocol 25
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 expression ‘o1 ^= o2’.

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 expression ‘o1 |= o2’.

PyObject* PyNumber_InPlaceCoerce (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_InPlaceCoerce (&o1, &o2) is equivalent to the Python statement ‘o1, o2 = coerce(o1, o2)’.

PyObject* PyNumber_Int (PyObject *o)
    Returns the o converted to an integer object on success, or NULL on failure. This is the equivalent of the Python expression ‘int(o)’.

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)’.

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

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

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)’.

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

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’.

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

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
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-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-th element of o. Returns -1 on failure. This is the equivalent of the Python statement ‘o[i] = v’.

int PySequence_DelItem (PyObject *o, int i)
Delete the i-th element of object o. Returns -1 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]’.

PyObject* PySequence_Tuple (PyObject *o)
Returns the o as a tuple on success, and NULL on failure. This is equivalent to the Python expression ‘tuple(o)’.

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. If o is a tuple, a new reference will be returned, otherwise a tuple will be constructed with the appropriate contents.

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 PySe-
sequence_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'th element of o, assuming that o was returned by PySequence_Fast(), and that i is within bounds. The caller is expected to get the length of the sequence by calling PyObject_Size() on o, since lists and tuples are guaranteed to always return their true length.

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 `len(o)`.

int PyMapping_DelItemString (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 `del o[key]`.

int PyMapping_DelItem (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 `del o[key]`.

int PyMapping_HasKeyString (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 `o.has_key(key)`. This function always succeeds.

int PyMapping_HasKey (PyObject *o, PyObject *key)
Return 1 if the mapping object has the key key and 0 otherwise. This is equivalent to the Python expression `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 `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 `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 `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 `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 `o[key] = v`. 

28 Chapter 6. Abstract Objects Layer
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)
Returns true if the object o is a type object.

int PyType_HasFeature(PyObject *o, int feature)
Returns true if the type object o sets the feature feature. Type features are denoted by single bit flags.

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.
7.2 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.

7.2.1 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.

```c
int PyString_Check(PyObject *o)
Return true if the object o is a string object.
```

```c
PyObject* PyString_FromString(const char *v)
Returns a new string object with the value v on success, and NULL on failure.
```

```c
PyObject* PyString_FromStringAndSize(const char *v, int len)
Returns 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.
```

```c
int PyString_Size(PyObject *string)
Returns the length of the string in string object string.
```

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

```c
char* PyString_AsString(PyObject *string)
Returns a null-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.
```

```c
char* PyString_AS_STRING(PyObject *string)
Macro form of PyString_AsString() but without error checking.
```

```c
int PyStringAsStringAndSize(PyObject *obj, char **buffer, int *length)
Returns a null-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 set to 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.
```

```c
void PyString_Concat(PyObject **string, PyObject *newpart)
Creates 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.
```
void PyTuple_PyObject() (PyObject **string, PyObject *newpart)

Creates a new string object in *string containing the contents of newpart appended to string. This version decrements the reference count of newpart.

int PyString_Resizer(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.

PyObject* PyString_Format(PyObject *format, PyObject *args)


Returns a new string object from format and args. Analogous to format % args. The args argument must be a tuple.

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.)

PyObject* PyString_InternFromString(const char *v)


A combination of PyString_FromString() 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.

PyObject* PyString_Decode(const char *s, int size, const char *encoding, const char *errors)


Create a string object by decoding size bytes of the encoded buffer 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. Returns NULL in case an exception was raised by the codec.

PyObject* PyString_Encode(const Py_UNICODE *s, int size, const char *encoding, const char *errors)


Encodes the Py_UNICODE buffer of the given size and returns a Python string 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. Returns NULL in case an exception was raised by the codec.

PyObject* PyString_AsEncodedString(PyObject *unicode, const char *encoding, const char *errors)


Encodes a string object and returns the result as Python string 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. Returns NULL in case an exception was raised by the codec.

7.2.2 Unicode Objects

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

Py_UNICODE

This type represents a 16-bit unsigned storage type which is used by Python internally as basis for holding Unicode ordinals. On platforms where wchar_t is available and also has 16-bits, 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 unsigned short.

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:

```c
int PyUnicode_Check(PyObject *o)
    Returns true if the object o is a Unicode object.
```

```c
int PyUnicode_GET_SIZE(PyObject *o)
    Returns the size of the object. o has to be a PyUnicodeObject (not checked).
```

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

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

```c
const char* PyUnicode_AS_DATA(PyObject *o)
    Returns a (const char *) 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.

```c
int Py_UNICODE_ISSPACE(Py_UNICODE ch)
    Returns 1/0 depending on whether ch is a whitespace character.
```

```c
int Py_UNICODE_ISLOWER(Py_UNICODE ch)
    Returns 1/0 depending on whether ch is a lowercase character.
```

```c
int Py_UNICODE_ISUPPER(Py_UNICODE ch)
    Returns 1/0 depending on whether ch is an uppercase character.
```

```c
int Py_UNICODE_ISTITLE(Py_UNICODE ch)
    Returns 1/0 depending on whether ch is a titlecase character.
```

```c
int Py_UNICODE_ISLINEBREAK(Py_UNICODE ch)
    Returns 1/0 depending on whether ch is a linebreak character.
```

```c
int Py_UNICODE_ISDECIMAL(Py_UNICODE ch)
    Returns 1/0 depending on whether ch is a decimal character.
```

```c
int Py_UNICODE_ISDIGIT(Py_UNICODE ch)
    Returns 1/0 depending on whether ch is a digit character.
```

```c
int Py_UNICODE_ISNUMERIC(Py_UNICODE ch)
    Returns 1/0 depending on whether ch is a numeric character.
```

```c
int Py_UNICODE_ISALPHA(Py_UNICODE ch)
    Returns 1/0 depending on whether ch is an alphabetic character.
```

```c
int Py_UNICODE_ISALNUM(Py_UNICODE ch)
    Returns 1/0 depending on whether ch is an alphanumerical character.
```

These APIs can be used for fast direct character conversions:

```c
Py_UNICODE Py_UNICODE_TOLOWER(Py_UNICODE ch)
    Returns the character ch converted to lower case.
```

```c
Py_UNICODE Py_UNICODE_TOUPPER(Py_UNICODE ch)
    Returns the character ch converted to upper case.
```

```c
Py_UNICODE Py_UNICODE_TOITLE(Py_UNICODE ch)
    Returns the character ch converted to title case.
int **Py_UNICODE** **TODECIMAL** (**Py_UNICODE** *ch*)
Returns the character *ch* converted to a decimal positive integer. Returns -1 in case this is not possible. Does not raise exceptions.

int **Py_UNICODE** **TODIGIT** (**Py_UNICODE** *ch*)
Returns the character *ch* converted to a single digit integer. Returns -1 in case this is not possible. Does not raise exceptions.

double **Py_UNICODE** **TOUNUMERIC** (**Py_UNICODE** *ch*)
Returns the character *ch* converted to a (positive) double. Returns -1.0 in case this is not possible. 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*)
*Return value: New reference.*
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.

**Py_UNICODE** * PyUnicode** AsUnicode** (**PyObject** *unicode*)
*Return a read-only pointer to the Unicode object’s internal Py_UNICODE buffer.*

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

**PyObject** * PyUnicode** FromEncodedObject** (**PyObject** *obj, const char **encoding**, const char **errors**)
*Return value: New reference.*
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 (see the next section for details).
3. All other objects cause an exception.

The API returns NULL in case of an error. The caller is responsible for decrementing the returned objects.

**PyObject** * PyUnicode** FromObject** (**PyObject** *obj*)
*Return value: New reference.*
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.

**PyObject** * PyUnicode** FromWideChar** (**const wchar_t** *w, int size*)
*Return value: New reference.*
Create a Unicode Object from the wchar_t buffer *w* of the given size. Returns NULL on failure.

int **PyUnicode** **AsWideChar** (**PyUnicodeObject** *unicode, wchar_t **w, int size*)
Copies the Unicode Object contents into the wchar_t buffer *w*. At most *size* wchar_t characters are copied. Returns the number of wchar_t characters copied or -1 in case of an error.
Builtin Codecs

Python provides a set of builtin 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 UTF-8.

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 builtin codecs is “strict” (ValueErrors are 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:

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. Returns NULL in case an exception was raised by the codec.

PyObject* PyUnicode_Encode (const Py_UNICODE *s, int size, const char *encoding, const char *errors)

Encodes the Py_UNICODE buffer of the given size and returns 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. Returns NULL in case an exception was raised by the codec.

PyObject* PyUnicode_AsEncodedString (PyObject *unicode, const char *encoding, const char *errors)

Encodes a Unicode objects and returns the result as 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. Returns NULL in case an exception was raised by the codec.

These are the UTF-8 codec APIs:

PyObject* PyUnicode_DecodeUTF8 (const char *s, int size, const char *errors)

Creates a Unicode object by decoding size bytes of the UTF-8 encoded string s. Returns NULL in case an exception was raised by the codec.

PyObject* PyUnicode_EncodeUTF8 (const Py_UNICODE *s, int size, const char *errors)

Encodes the Py_UNICODE buffer of the given size using UTF-8 and returns a Python string object. Returns NULL in case an exception was raised by the codec.

PyObject* PyUnicode_AsUTF8String (PyObject *unicode)

Encodes a Unicode objects using UTF-8 and returns the result as Python string object. Error handling is “strict”. Returns NULL in case an exception was raised by the codec.

These are the UTF-16 codec APIs:

PyObject* PyUnicode_DecodeUTF16 (const char *s, int size, const char *errors)


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. BOM marks 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.

Returns NULL in case an exception was raised by the codec.

PyObject* \texttt{PyUnicode\_EncodeUTF16} (const \texttt{Py\_UNICODE} *s, int size, const char *errors, int byteorder)


Returns a Python string object holding the UTF-16 encoded value of the Unicode data in \texttt{s}.

If byteorder is not 0, output is written according to the following byte order:

\begin{itemize}
  \item byteorder == -1: little endian
  \item byteorder == 0: native byte order (writes a BOM mark)
  \item byteorder == 1: big endian
\end{itemize}

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.

Note that \texttt{Py\_UNICODE} data is being interpreted as UTF-16 reduced to UCS-2. This trick makes it possible to add full UTF-16 capabilities at a later point without compromising the APIs.

Returns NULL in case an exception was raised by the codec.

PyObject* \texttt{PyUnicode\_AsUTF16String} (PyObject *unicode)


Returns a Python string using the UTF-16 encoding in native byte order. The string always starts with a BOM mark. Error handling is “strict”. Returns NULL in case an exception was raised by the codec.

These are the “Unicode Escape” codec APIs:

PyObject* \texttt{PyUnicode\_DecodeUnicodeEscape} (const char *s, int size, const char *errors)


Creates a Unicode object by decoding size bytes of the Unicode-Escape encoded string s. Returns NULL in case an exception was raised by the codec.

PyObject* \texttt{PyUnicode\_EncodeUnicodeEscape} (const \texttt{Py\_UNICODE} *s, int size, const char *errors)


Encodes the \texttt{Py\_UNICODE} buffer of the given size using Unicode-Escape and returns a Python string object. Returns NULL in case an exception was raised by the codec.

PyObject* \texttt{PyUnicode\_AsUnicodeEscapeString} (PyObject *unicode)


Encodes a Unicode objects using Unicode-Escape and returns the result as Python string object. Error handling is “strict”. Returns NULL in case an exception was raised by the codec.

These are the “Raw Unicode Escape” codec APIs:

PyObject* \texttt{PyUnicode\_DecodeRawUnicodeEscape} (const char *s, int size, const char *errors)


Creates a Unicode object by decoding size bytes of the Raw-Unicode-Escape encoded string s. Returns NULL in case an exception was raised by the codec.

PyObject* \texttt{PyUnicode\_EncodeRawUnicodeEscape} (const \texttt{Py\_UNICODE} *s, int size, const char *errors)


Encodes the \texttt{Py\_UNICODE} buffer of the given size using Raw-Unicode-Escape and returns a Python string object. Returns NULL in case an exception was raised by the codec.

7.2. Sequence Objects 35
PyObject* PyUnicode_AsRawUnicodeEscapeString(PyObject *unicode)
{
    return PyUnicode_AsRawUnicodeEscapeString(unicode);
}


Encodes a Unicode objects using Raw-Unicode-Escape and returns the result as Python string object. Error
handling is “strict”. Returns NULL in case 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.

PyObject* PyUnicode_DecodeLatin1(const char *s, int size, const char *errors)
{
    return PyUnicode_DecodeLatin1(s, size, errors);
}


Creates a Unicode object by decoding size bytes of the Latin-1 encoded string s. Returns NULL in case an
exception was raised by the codec.

PyObject* PyUnicode_EncodeLatin1(const Py_UNICODE *s, int size, const char *errors)
{
    return PyUnicode_EncodeLatin1(s, size, errors);
}


Encodes the Py_UNICODE buffer of the given size using Latin-1 and returns a Python string object. Returns
NULL in case an exception was raised by the codec.

PyObject* PyUnicode_AsLatin1String(PyObject *unicode)
{
    return PyUnicode_AsLatin1String(unicode);
}


Encodes a Unicode objects using Latin-1 and returns the result as Python string object. Error handling is “strict”.
Returns NULL in case 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.

PyObject* PyUnicode_DecodeASCII(const char *s, int size, const char *errors)
{
    return PyUnicode_DecodeASCII(s, size, errors);
}


Creates a Unicode object by decoding size bytes of the ASCII encoded string s. Returns NULL in case an
exception was raised by the codec.

PyObject* PyUnicode_EncodeASCII(const Py_UNICODE *s, int size, const char *errors)
{
    return PyUnicode_EncodeASCII(s, size, errors);
}


Encodes the Py_UNICODE buffer of the given size using ASCII and returns a Python string object. Returns
NULL in case an exception was raised by the codec.

PyObject* PyUnicode_AsASCIIString(PyObject *unicode)
{
    return PyUnicode_AsASCIIString(unicode);
}


Encodes a Unicode objects using ASCII and returns the result as Python string object. Error handling is “strict”.
Returns NULL in case 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 inter-
preted 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 inter-
preted 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(const char *s, int size, PyObject *mapping, const char *errors)
{
    return PyUnicode_DecodeCharmap(s, size, mapping, errors);
}


Creates a Unicode object by decoding size bytes of the encoded string s using the given mapping object. Returns
PyObject* PyUnicode_EncodeCharmap(const Py_UNICODE *s, int size, PyObject *mapping, const char *errors)

Encodes the Py_UNICODE buffer of the given size using the given mapping object and returns a Python string object. Returns NULL in case an exception was raised by the codec.

PyObject* PyUnicode_AsCharmapString(PyObject *unicode, PyObject *mapping)

Encodes a Unicode objects using the given mapping object and returns the result as Python string object. Error handling is “strict”. Returns NULL in case 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)

Translates a Py_UNICODE buffer of the given length by applying a character mapping table to it and returns the resulting Unicode object. Returns 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 must only provide the __getitem__ interface, e.g. dictionaries or sequences. 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 (const char *s, int size, const char *errors)

Creates a Unicode object by decoding size bytes of the MBCS encoded string s. Returns NULL in case an exception was raised by the codec.

PyObject* PyUnicode_EncodeMBCS (const Py_UNICODE *s, int size, const char *errors)

Encodes the Py_UNICODE buffer of the given size using MBCS and returns a Python string object. Returns NULL in case an exception was raised by the codec.

PyObject* PyUnicode_AsMBCSString (PyObject *unicode)

Encodes a Unicode objects using MBCS and returns the result as Python string object. Error handling is “strict”. Returns NULL in case 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 apporpriate.

They all return NULL or -1 in case an exception occurs.

PyObject* PyUnicodeConcat (PyObject *left, PyObject *right)

Concat two strings giving a new Unicode string.

PyObject* PyUnicodeSplit (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 maxsplit)

Split a Unicode string at line breaks, returning a list of Unicode strings. CRLF is considered to be one line break. 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 must only provide the __getitem__ interface, e.g. dictionaries or sequences. 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.

PyObject* 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.

PyObject* 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), 0 otherwise.

PyObject* PyUnicode_Count(PyObject *str, PyObject *substr, int start, int end)

Count the number of occurrences of substr in str[start:end]

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, greater than resp.

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

Returns 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)

Checks whether element is contained in container and returns true or false accordingly.

 element has to coerce to a one element Unicode string. -1 is returned in case of an error.

7.2.3 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 `Pyargs_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 10.5), 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 value: New reference.*
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 value: New reference.*
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 value: New reference.*
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)**
*Return value: New reference.*
Similar to PyBuffer_FromMemory(), but the returned buffer is writable.

**PyObject** PyBuffer_New(int size)

*Return value: New reference.*

Returns a new writable buffer object that maintains its own memory buffer of size bytes. ValueError is returned if size is not zero or positive.

### 7.2.4 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 the argument is a tuple object.

**PyObject** PyTuple_New(int len)

*Return value: New reference.*

Return a new tuple object of size len, or NULL on failure.

int PyTuple_Size(PyObject *p)

Takes a pointer to a tuple object, and returns the size of that tuple.

**PyObject** PyTuple_GetItem(PyObject *p, int pos)

*Return value: Borrowed reference.*

Returns the object at position pos in the tuple pointed to by p. If pos is out of bounds, returns NULL and sets an IndexError exception.

**PyObject** PyTuple_GET_ITEM(PyObject *p, int pos)

*Return value: Borrowed reference.*

Does the same, but does no checking of its arguments.

**PyObject** PyTuple_GetSlice(PyObject *p, int low, int high)

*Return value: New reference.*

Takes a slice of the tuple pointed to by p from low to high and returns it as a new tuple.

int PyTuple_SetItem(PyObject *p, int pos, PyObject *o)

Inserts a reference to object o at position pos of the tuple pointed to by p. It returns 0 on success. **Note:** This function “steals” a reference to o.

void PyTuple_SET_ITEM(PyObject *p, int pos, PyObject *o)

Does the same, 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, int last_is_sticky)

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. The last_is_sticky flag is not used and should always be false. Think of this as destroying the old tuple and creating a new one, only more efficiently. Returns 0 on success and -1 on failure (in which case a MemoryError or SystemError will be raised).
7.2.5 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.

```c
int PyList_Check(PyObject *p)
    Returns true if its argument is a PyListObject.
```

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

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

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

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

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

```c
int PyList_SetItem(PyObject *list, int index, PyObject *item)
    Sets the item at index index in list to item. Note: This function “steals” a reference to item.
```

```c
PyObject* PyList_SET_ITEM(PyObject *list, int i, PyObject *o)
    Return value: Borrowed reference.
    Macro form of PyList_SetItem() without error checking. Note: This function “steals” a reference to item.
```

```c
int PyList_Insert(PyObject *list, int index, PyObject *item)
    Inserts the item item into list list in front of index index. Returns 0 if successful; returns -1 and raises an exception if unsuccessful. Analogous to list.insert(index, item).
```

```c
int PyList_Append(PyObject *list, PyObject *item)
    Appends the object item at the end of list list. Returns 0 if successful; returns -1 and sets an exception if unsuccessful. Analogous to list.append(item).
```

```c
PyObject* PyList_GetSlice(PyObject *list, int low, int high)
    Returns a list of the objects in list containing the objects between low and high. Returns NULL and sets an exception if unsuccessful. Analogous to list[low:high].
```

```c
int PyList_SetSlice(PyObject *list, int low, int high, PyObject *itemlist)
    Sets the slice of list between low and high to the contents of itemlist. Analogous to list[low:high] = itemlist. Returns 0 on success, -1 on failure.
```

```c
int PyList_Sort(PyObject *list)
    Sorts the items of list in place. Returns 0 on success, -1 on failure. This is equivalent to `list.sort()`.
```

```c
int PyList_Reverse(PyObject *list)
    Reverses the items of list in place. Returns 0 on success, -1 on failure. This is the equivalent of `list.reverse()`.
```
PyObject* PyList_AsTuple (PyObject *list)
Returns a new tuple object containing the contents of list; equivalent to ‘tuple (list)’.

7.3 Mapping Objects

7.3.1 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)
Returns true if its argument is a PyDictObject.

PyObject* PyDict_New ()
Returns a new empty dictionary, or NULL on failure.

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

PyObject* PyDict_Copy (PyObject *p)
Returns a new dictionary that contains the same key-value pairs as p. Empties an existing dictionary of all
key-value pairs.

int PyDict_SetItem (PyObject *p, PyObject *key, PyObject *val)
Inserts value into the dictionary with a key of key. key must be hashable; if it isn’t, TypeError will be raised.

int PyDict_SetItemString (PyObject *p, char *key, PyObject *val)
Inserts value into the dictionary using key as a key. key should be a char*. The key object is created using
PyString_FromString (key).

int PyDict_DelItem (PyObject *p, PyObject *key)
Removes the entry in dictionary p with key key. key must be hashable; if it isn’t, TypeError is raised.

int PyDict_DelItemString (PyObject *p, char *key)
Removes the entry in dictionary p which has a key specified by the string key.

PyObject* PyDict_GetItem (PyObject *p, PyObject *key)
Return value: Borrowed reference.
Returns the object from dictionary p which has a key key. Returns NULL if the key key is not present, but without
setting an exception.

PyObject* PyDict_GetItemString (PyObject *p, char *key)
Return value: Borrowed reference.
This is the same as PyDict_GetItem (), but key is specified as a char*, rather than a PyObject*.

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

PyObject* PyDict.Keys (PyObject *p)
Returns a PyListObject containing all the keys from the dictionary, as in the dictionary method keys() (see the Python Library Reference).

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

int PyDict_Size(PyObject *p)
Returns the number of items in the dictionary. This is equivalent to 'len (p)' on a dictionary.

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.

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, 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);
    }

7.4 Numeric Objects

7.4.1 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)
Returns true if \( o \) is of type PyInt_Type.

\textbf{PyObject* \%PyInt\_FromLong\( (\text{long} \ ival)\)}

\textit{Return value: New reference.}

Creates 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. :-)

\textbf{long \%PyInt\_AsLong\( (\text{PyObject *} \text{io})\)}

Will first attempt to cast the object to a PyIntObject, if it is not already one, and then return its value.

\textbf{long \%PyInt\_AS\_LONG\( (\text{PyObject *} \text{io})\)}

Returns the value of the object \( \text{io} \). No error checking is performed.

\textbf{long \%PyInt\_GetMax\( ()\)}

Returns the system’s idea of the largest integer it can handle (\%LONG\_MAX, as defined in the system header files).

### 7.4.2 Long Integer Objects

\textbf{PyLongObject}

This subtype of PyObject represents a Python long integer object.

\textbf{PyTypeObject \%PyLong\_Type}

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

\textbf{int \%PyLong\_Check\( (\text{PyObject *} \text{p})\)}

Returns true if its argument is a PyLongObject.

\textbf{PyObject* \%PyLong\_FromLong\( (\text{long} \ v)\)}

\textit{Return value: New reference.}

Returns a new PyLongObject object from \( v \), or NULL on failure.

\textbf{PyObject* \%PyLong\_FromUnsignedLong\( (\text{unsigned long} \ v)\)}

\textit{Return value: New reference.}

Returns a new PyLongObject object from a C unsigned long, or NULL on failure.

\textbf{PyObject* \%PyLong\_FromDouble\( (\text{double} \ v)\)}

\textit{Return value: New reference.}

Returns a new PyLongObject object from the integer part of \( v \), or NULL on failure.

\textbf{long \%PyLong\_AsLong\( (\text{PyObject *} \text{pylong})\)}

Returns a C long representation of the contents of \%pylong. If \%pylong is greater than \%LONG\_MAX, an OverflowError is raised.OverflowError

\textbf{unsigned long \%PyLong\_AsUnsignedLong\( (\text{PyObject *} \text{pylong})\)}

Returns a C unsigned long representation of the contents of \%pylong. If \%pylong is greater than \%ULONG\_MAX, an OverflowError is raised.OverflowError

\textbf{double \%PyLong\_AsDouble\( (\text{PyObject *} \text{pylong})\)}

Returns a C double representation of the contents of \%pylong.

\textbf{PyObject* \%PyLong\_FromString\( (\text{char *} \text{str} , \text{char *} \text{pend} , \text{int} \ base)\)}

\textit{Return value: New reference.}

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 base 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.

7.4.3 Floating Point Objects

**PyFloatObject**
This subtype of PyObject represents a Python floating point object.

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

```c
int PyFloat_Check (PyObject *p)
Returns true if its argument is a PyFloatObject.
```

```c
PyObject* PyFloat_FromDouble (double v)
Creates a PyFloatObject object from v, or NULL on failure.
```

```c
double PyFloat_AsDouble (PyObject *pyfloat)
Returns a C double representation of the contents of pyfloat.
```

```c
double PyFloat_AS_DOUBLE (PyObject *pyfloat)
Returns a C double representation of the contents of pyfloat, but without error checking.
```

7.4.4 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.

**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.
```

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

7.4. Numeric Objects
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)
Returns true if its argument is a PyComplexObject.

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)
Returns a new PyComplexObject object from real and imag.

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

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

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

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)
Returns true if its argument is a PyFileObject.

PyObject* PyFile_FromString(char *filename, char *mode)
On success, returns 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, returns 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. RETURNS NULL ON FAILURE.

FILE* PyFile_AsFile (PyFileObject *p)
Returns 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.
Returns 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_SoftSpace (PyObject *obj, PyFileObject *p, int flags)
This function exists for internal use by the interpreter. Sets the softspace attribute of *p* to newflag and returns 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)
Writes 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(). Returns 0 on success or -1 on failure; the appropriate exception will be set.

int PyFile_WriteString (char *s, PyFileObject *p, int flags)
Writes string *s* to file object *p*. Returns 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)
Returns 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. Other Objects
7.5.3 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)  
Returns true if its argument is a module object.

PyObject* **PyModule_New** (char *name)  
*Return value: New reference.*  
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.

char* **PyModuleGetName** (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* **PyModuleGetFilename** (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 **PyModuleAddObject** (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. Returns -1 on error, 0 on success. New in version 2.0.

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

int **PyModuleAddStringConstant** (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. Returns -1 on error, 0 on success. New in version 2.0.

7.5.4 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.

int **PyCObject_Check** (PyObject *p)  
Returns true if its argument is a PyCObject.

PyObject* **PyCObject_FromVoidPtr** (void* cobj, void (*destr)(void *))  
*Return value: New reference.*  
Creates a PyCObject from the void *cobj. The destr function will be called when the object is reclaimed,
unless it is NULL.

```c
PyObject* PyCObject_FromVoidPtrAndDesc(void* cobj, void* desc, void (*destr)(void *, void *))
Creates 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.
```

```c
void* PyCObject_AsVoidPtr(PyObject* self)
Returns the object void * that the PyCObject self was created with.
```

```c
void* PyCObject_GetDesc(PyObject* self)
Returns the description void * that the PyCObject self was created with.
```
CHAPTER EIGHT

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.

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 extension 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 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 insu-
lation 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 dic-
tionary 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 interpreted 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.

cchar* PyGetProgramFullPath()

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.

cchar* PyGetPath()

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, ‘;’ on DOS/Windows, and ‘\n’ (the ASCII newline character) on Macintosh. 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.

const char* PyGetVersion()

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 the list sys.version.

const char* PyGetPlatform()

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 Macintosh, it is ‘mac’. 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.

const char* PyGetCopyright()

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 the list sys.copyright.

const char* PyGetCompiler()

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.

const char* PyGetBuildInfo()
Return information about the sequence number and build date and time of the current Python interpreter instance, for example

"#67, Aug 1 1997, 22:34:28"

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 ten 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`. This is new in Python 1.5; in earlier versions, such state was stored in global variables, and switching threads could cause problems. In particular, exception handling is now thread safe, when the application uses `sys.exc_info()` to access the exception last raised in the current thread.

There’s one global variable left, 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:

```c
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:

```c
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.

When creating a thread data structure, you need to provide an interpreter state data structure. The interpreter state data structure holds global data that is shared by all threads in an interpreter, for example the module administration (**sys.modules**). Depending on your needs, you can either create a new interpreter state data structure, or share the interpreter state data structure used by the Python main thread (to access the latter, you must obtain the thread state and access its **interp** member; this must be done by a thread that is created by Python or by the main thread after Python is initialized).

### 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.

8.1. Thread State and the Global Interpreter Lock
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.

void PyEval_InitThreads()

Initialize and acquire the global interpreter lock. It should be called in the main thread before creating 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 it 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.

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.

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.

void PyEval_AcquireThread(PyThreadState *tstate)

Acquire the global interpreter lock and then 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.

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.

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.)

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_BEGIN_BLOCK_THREADS
This macro expands to ‘PyEval_RestoreThread(_save); ’ i.e. 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_BEGIN_UNBLOCK_THREADS
This macro expands to ‘_save = PyEval_SaveThread(); ’ i.e. 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.
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 on 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, 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 non-NULL pointer.

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); 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().

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 behaviour 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*.

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*.

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 depreqated in extension modules.

```c
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:
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:

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.

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.
Defining New Object Types

PyObject* _PyObject_New(PyTypeObject *type)

PyVarObject* _PyObject_NewVar (PyTypeObject *type, int size)

void _PyObject_Del (PyObject *op)

PyObject* PyObject_Init (PyObject *op, PyTypeObject *type)
    Return value: Borrowed reference.

PyVarObject* PyObject_InitVar (PyVarObject *op, PyTypeObject *type, int size)
    Return value: Borrowed reference.

TYPE* PyObject_New (TYPE, PyTypeObject *type)

TYPE* PyObject_NewVar (TYPE, PyTypeObject *type, int size)

void PyObject_Del (PyObject *op)

PyObject* PyObject_New (TYPE, PyTypeObject *type)

void PyObject_Del (PyObject *op)

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.

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.

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.

PyArg_ParseTupleAndKeywords, PyArg_ParseTuple, PyArg_Parse

Py_BuildValue

DL_IMPORT

_Py_NoneStruct

10.1 Common Object Structures

PyObject, PyVarObject

PyObject_HEAD, PyObject_HEAD_INIT, PyObject_VAR_HEAD

Typedefs: unaryfunc, binaryfunc, ternaryfunc, inquiry, coercion, intargfunc, intintargfunc, intobjargproc, intintobjargproc, objobjargproc, destructor, printfunc, getattrfunc, getattrofunc, setattrfunc, setattrofunc, cmpfunc, reprfunc, hashfunc

---

typeof the functions used to implement most Python callables in C.

---

PyCFunction

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>

PyObject* Py_FindMethod(PyMethodDef[] table, PyObject *ob, char *name)

*Return value: New reference.*

Return a bound method object for an extension type implemented in C. This function also handles the special attribute __methods__, returning a list of all the method names defined in `table`.

10.2 Mapping Object Structures

PyMappingMethods

Structure used to hold pointers to the functions used to implement the mapping protocol for an extension type.

10.3 Number Object Structures

PyNumberMethods

Structure used to hold pointers to the functions an extension type uses to implement the number protocol.

10.4 Sequence Object Structures

PySequenceMethods

Structure used to hold pointers to the functions which an object uses to implement the sequence protocol.
10.5 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.
10.6 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.

To create a container type, the \texttt{tp\_flags} field of the type object must include the \texttt{Py\_TPFLAGS\_GC} and provide an implementation of the \texttt{tp\_traverse} handler. The computed value of the \texttt{tp\_basicsize} field must include \texttt{PyGC\_HEAD\_SIZE} as well. If instances of the type are mutable, a \texttt{tp\_clear} implementation must also be provided.

\texttt{Py\_TPFLAGS\_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.

\texttt{PyGC\_HEAD\_SIZE}

Extra memory needed for the garbage collector. Container objects must include this in the calculation of their \texttt{tp\_basicsize}. If the collector is disabled at compile time then this is 0.

Constructors for container types must conform to two rules:

1. The memory for the object must be allocated using \texttt{PyObject\_New()} or \texttt{PyObject\_VarNew()}.
2. Once all the fields which may contain references to other containers are initialized, it must call \texttt{PyObject\_GC\_Init()}. 

\texttt{void PyObject\_GC\_Init (PyObject \*op)}

Adds the object \texttt{op} to the set of container objects tracked by the collector. The collector can run at unexpected times so objects must be valid while being tracked. This should be called once all the fields followed by the \texttt{tp\_traverse} handler become valid, usually near the end of the constructor.

Similarly, the deallocator for the object must conform to a similar pair of rules:

1. Before fields which refer to other containers are invalidated, \texttt{PyObject\_GC\_Fini()} must be called.
2. The object’s memory must be deallocated using \texttt{PyObject\_Del()}.

\texttt{void PyObject\_GC\_Fini (PyObject \*op)}

Remove the object \texttt{op} from the set of container objects tracked by the collector. Note that \texttt{PyObject\_GC\_Init()} can be called again on this object to add it back to the set of tracked objects. The deallocator (\texttt{tp\_dealloc} handler) should call this for the object before any of the fields used by the \texttt{tp\_traverse} handler become invalid.

\textbf{Note:} Any container which may be referenced from another object reachable by the collector must itself be tracked by the collector, so it is generally not safe to call this function anywhere but in the object’s deallocator.

The \texttt{tp\_traverse} handler accepts a function parameter of this type:

\texttt{int (*visitproc) (PyObject *object, void *arg)}

Type of the visitor function passed to the \texttt{tp\_traverse} handler. The function should be called with an object to traverse as \texttt{object} and the third parameter to the \texttt{tp\_traverse} handler as \texttt{arg}.

The \texttt{tp\_traverse} handler must have the following type:

\texttt{int (*traverseproc) (PyObject *self, visitproc visit, void *arg)}

Traversal function for a container object. Implementations must call the \texttt{visit} function for each object directly
contained by \textit{self}, with the parameters to \textit{visit} being the contained object and the \textit{arg} value passed to the handler. If \textit{visit} returns a non-zero value then an error has occurred and that value should be returned immediately.

The \texttt{tp\_clear} handler must be of the \texttt{inquiry} type, or \texttt{NULL} if the object is immutable.

\begin{verbatim}
int (*inquiry)(PyObject *self)
\end{verbatim}

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 (i.e., don’t just call \texttt{Py\_DECREF() on a reference). The collector will call this method if it detects that this object is involved in a reference cycle.

### 10.6.1 Example Cycle Collector Support

This example shows only enough of the implementation of an extension type to show how the garbage collector support needs to be added. It shows the definition of the object structure, the \texttt{tp\_traverse}, \texttt{tp\_clear} and \texttt{tp\_dealloc} implementations, the type structure, and a constructor — the module initialization needed to export the constructor to Python is not shown as there are no special considerations there for the collector. To make this interesting, assume that the module exposes ways for the \texttt{container} field of the object to be modified. Note that since no checks are made on the type of the object used to initialize \texttt{container}, we have to assume that it may be a container.

\begin{verbatim}
#include "Python.h"

typedef struct {
  PyObject\_HEAD
  PyObject *container;
} MyObject;

static int
my\_traverse(MyObject *self, visitproc visit, void *arg)
{
  if (self->container != NULL)
    return visit(self->container, arg);
  else
    return 0;
}

static int
my\_clear(MyObject *self)
{
  Py\_XDECREF(self->container);
  self->container = NULL;

  return 0;
}

static void
my\_dealloc(MyObject *self)
{
  PyObject\_GC\_Finii((PyObject *) self);
  Py\_XDECREF(self->container);
  Py\_DECREF(self);
  Py\_OBJ\_Del(self);
}
\end{verbatim}
static PyTypeObject
MyObject_Type = {
    PyObject_HEAD_INIT(NULL)
    0,
    "MyObject",
    sizeof(MyObject) + PyGC_HEAD_SIZE,
    0,
    (destructor)my_dealloc,    /* tp_dealloc */
    0,    /* tp_print */
    0,    /* tp_getattr */
    0,    /* tp_setattr */
    0,    /* tp_compare */
    0,    /* tp_repr */
    0,    /* tp_as_number */
    0,    /* tp_as_sequence */
    0,    /* tp_as_mapping */
    0,    /* tp_hash */
    0,    /* tp_call */
    0,    /* tp_str */
    0,    /* tp_getattro */
    0,    /* tp_setattro */
    0,    /* tp_as_buffer */
Py_TPFLAGS_DEFAULT | Py_TPFLAGS_GC,
    0,    /* tp_doc */
    (traverseproc)my_traverse,    /* tp_traverse */
    (inquiry)my_clear,    /* tp_clear */
    0,    /* tp_richcompare */
    0,    /* tp_weaklistoffset */
};

/* This constructor should be made accessible from Python. */
static PyObject *
new_object(PyObject *unused, PyObject *args)
{
    PyObject *container = NULL;
    MyObject *result = NULL;

    if (PyArg_ParseTuple(args, "|O:new_object", &container)) {
        result = PyObject_New(MyObject, &MyObject_Type);
        if (result != NULL) {
            result->container = container;
            PyObject_GC_Init();
        }
    }
    return (PyObject *) result;
}
APPENDIX

A

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.

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.

Before submitting a report, please log into SourceForge if you are a member; this will make it possible for the developers to contact you for additional information if needed. If you are not a SourceForge member but would not mind the developers contacting you, you may include your email address in your bug description. In this case, please realize that the information is publically available and cannot be protected.

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 near the bottom 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. If you have a SourceForge account and logged in to report 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.
## Symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>_PyImport_FindExtension()</td>
<td>19</td>
</tr>
<tr>
<td>_PyImport_Fini()</td>
<td>19</td>
</tr>
<tr>
<td>_PyImport_FixupExtension()</td>
<td>19</td>
</tr>
<tr>
<td>_PyImport_Init()</td>
<td>19</td>
</tr>
<tr>
<td>_PyObject_Del()</td>
<td>63</td>
</tr>
<tr>
<td>_PyObject_New()</td>
<td>63</td>
</tr>
<tr>
<td>_PyObject_NewVar()</td>
<td>63</td>
</tr>
<tr>
<td>_PyString_Resize()</td>
<td>31</td>
</tr>
<tr>
<td>_PyTuple_Resize()</td>
<td>40</td>
</tr>
<tr>
<td>_Py_c_diff()</td>
<td>45</td>
</tr>
<tr>
<td>_Py_c_neg()</td>
<td>45</td>
</tr>
<tr>
<td>_Py_c_pow()</td>
<td>46</td>
</tr>
<tr>
<td>_Py_c_prod()</td>
<td>45</td>
</tr>
<tr>
<td>_Py_c_quot()</td>
<td>46</td>
</tr>
<tr>
<td>_Py_c_sum()</td>
<td>45</td>
</tr>
<tr>
<td><em>all</em>_</td>
<td></td>
</tr>
<tr>
<td><em>builtins</em>_</td>
<td>8, 51</td>
</tr>
<tr>
<td><em>dict</em>_</td>
<td>48</td>
</tr>
<tr>
<td><em>doc</em>_</td>
<td>48</td>
</tr>
<tr>
<td><em>file</em>_</td>
<td>48</td>
</tr>
<tr>
<td><em>import</em>_</td>
<td>18</td>
</tr>
<tr>
<td><em>main</em>_</td>
<td>8, 51</td>
</tr>
<tr>
<td><em>name</em>_</td>
<td>48</td>
</tr>
</tbody>
</table>

### A

- abort(), 18
- abs() (built-in function), 24
- apply() (built-in function), 22, 23
- argv (in module sys), 54

### B

- buffer
  - object, 38
  - buffer interface, 38
  - BufferType (in module types), 39

### C

- calloc(), 59
- cleanup functions, 18
- close() (in module os), 52
- cmp() (built-in function), 22
- CObject
  - object, 48
- coerce() (built-in function), 26
- compile() (built-in function), 19
  - complex number
    - object, 45
- copyright (in module sys), 53

### D

- dictionary
  - object, 42
- DictionaryType (in module types), 42
- DictType (in module types), 42
- divmod() (built-in function), 24

### E

- environment variables
  - PATH, 8
  - PYTHONHOME, 8
  - PYTHONPATH, 8
  - exec_prefix, 1, 2
  - prefix, 1, 2
- EOFError (built-in exception), 47
- errno, 55
- exc_info() (in module sys), 6, 54
- exc_traceback (in module sys), 6, 13
- exc_type (in module sys), 6, 13
- exc_value (in module sys), 6, 13
- Exception (built-in exception), 16
- exec_prefix, 1, 2
- executable (in module sys), 53
- exit(), 18

### F

- file
  - object, 46
- FileType (in module types), 46
- float() (built-in function), 26
  - floating point
    - object, 45
FloatType (in modules types), 45
fopen(), 46
free(), 59
freeze utility, 19

G
global interpreter lock, 54

H
hash() (built-in function), 23

I
ihooks (standard module), 18
incr_item(), 6, 7
instance
  object, 47
int() (built-in function), 26
getcharbufferproc (C type), 66
getreadbufferproc (C type), 65
getsegcountproc (C type), 65
getwritebufferproc (C type), 65
inquiry (C type), 67
traverseproc (C type), 66
visitproc (C type), 66
integer
  object, 43
interpreter lock, 54
IntType (in modules types), 43

K
KeyboardInterrupt (built-in exception), 15

L
len() (built-in function), 23, 26, 28, 41, 43
list
  object, 41
ListType (in module types), 41
lock, interpreter, 54
long() (built-in function), 26
long integer
  object, 44
LONG_MAX, 44
LongType (in modules types), 44

M
main(), 52, 54
malloc(), 59
mapping
  object, 42
module
  object, 48
  search path, 8, 51, 53
modules (in module sys), 18, 51

ModuleType (in module types), 48

N
None
  object, 29
numeric
    object, 43

O
object
  buffer, 38
  CObject, 48
  complex number, 45
dictionary, 42
  file, 46
  floating point, 45
  instance, 47
  integer, 43
  list, 41
  long integer, 44
  mapping, 42
  module, 48
  None, 29
  numeric, 43
  sequence, 30
  string, 30
tuple, 40
type, 2, 29

P
package variable
  __all__, 18
PATH, 8
path
  module search, 8, 51, 53
  path (in module sys), 8, 51, 53
  platform (in module sys), 53
  pow() (built-in function), 24, 25
  prefix, 1, 2
Py_AtExit(), 18
Py_BEGIN_ALLOW_THREADS, 55
Py_BEGIN_ALLOW_THREADS (macro), 56
Py_BEGIN_BLOCK_THREADS (macro), 57
Py_BEGIN_UNBLOCK_THREADS (macro), 57
Py_CompileString(), 10
Py_CompileString(), 10
Py_complex (C type), 45
Py_DECREF(), 11
Py_DECREF(), 2
Py_END_ALLOW_THREADS, 55
Py_END_ALLOW_THREADS (macro), 57
Py_END_OF_BUFFER, 39
Py_EndInterpreter(), 52
Py_eval_input, 10
Py_Exit(), 18
Py_FatalError(), 17
Py_FatalError(), 54
Py_FdIsInteractive(), 17
Py_file_input, 10
Py_Finalize(), 51
Py_Finalize(), 18, 51, 52
Py_FindMethod(), 64
Py_GetBuildInfo(), 53
Py_GetCompiler(), 53
Py_GetCopyright(), 53
Py_GetExecPrefix(), 52
Py_GetExecPrefix(), 8
Py_GetPath(), 53
Py_GetPath(), 8, 52
Py_GetPlatform(), 53
Py_GetPrefix(), 52
Py_GetPrefix(), 8
Py_GetProgramFullPath(), 53
Py_GetProgramFullPath(), 8
Py_GetProgramName(), 52
Py_GetVersion(), 53
Py_INCREF(), 11
Py_INCREF(), 2
Py_Initialize(), 51
Py_Initialize(), 8, 52, 56
Py_InitModule(), 63
Py_InitModule3(), 63
Py_InitModule4(), 63
Py_IsInitialized(), 51
Py_IsInitialized(), 8
Py_NewInterpreter(), 51
Py_None, 29
Py_PRINT_RAW, 47
Py_SetProgramName(), 52
Py_SetProgramName(), 8, 51–53
Py_single_input, 10
Py_TPFLAGS_GC (data in ), 66
Py_TPFLAGS_HAVE_GETCHARBUFFER (data in ), 65
Py_UNICODE (C type), 31
Py_UNICODE_ISALNUM(), 32
Py_UNICODE_ISALPHA(), 32
Py_UNICODE_ISDECIMAL(), 32
Py_UNICODE_ISDIGIT(), 32
Py_UNICODE_ISLINEBREAK(), 32
Py_UNICODE_ISLOWER(), 32
Py_UNICODE_ISNUMERIC(), 32
Py_UNICODE_ISSPACE(), 32
Py_UNICODE_ISSINGLE(), 32
Py_UNICODE_ISUPPER(), 32
Py_UNICODE_TODECIMAL(), 33
Py_UNICODE_TODIGIT(), 33
Py_UNICODE_TOLOWER(), 32
Py_UNICODE_TOORDER(), 33
Py_UNICODE_TOUPPER(), 32
Py_XDECREF(), 11
Py_XDECREF(), 7
Py_XINCREF(), 11
PyBuffer_Check(), 39
PyBuffer_FromMemory(), 39
PyBuffer_FromObject(), 39
PyBuffer_FromReadWriteMemory(), 39
PyBuffer_FromReadWriteObject(), 39
PyBuffer_New(), 40
PyBuffer_Type, 39
PyBufferObject (C type), 39
PyBufferProcs, 39
PyBufferProcs (C type), 65
PyCallable_Check(), 22
PyCFunction (C type), 64
PyCObject (C type), 48
PyCObject_AsVoidPtr(), 49
PyCObject_Check(), 48
PyCObject_FromVoidPtr(), 48
PyCObject_FromVoidPtrAndDesc(), 49
PyCObject_GetDesc(), 49
PyComplex_AsCComplex(), 46
PyComplex_Check(), 46
PyComplex_FromCComplex(), 46
PyComplex_FromDoubles(), 46
PyComplex_ImagAsDouble(), 46
PyComplex_RealAsDouble(), 46
PyComplex_Type, 46
PyComplexObject (C type), 46
PyDict_Check(), 42
PyDict_Clear(), 42
PyDict_Copy(), 42
PyDict_DelItem(), 42
PyDict_DelItemString(), 42
PyDict_GetItem(), 42
PyDict_GetItemString(), 42
PyDict_Items(), 42
PyDict_Keys(), 42
PyDict_New(), 42
PyDict_Next(), 43
PyDict_SetItem(), 42
PyDict_SetItemString(), 42
PyDict_Size(), 43
PyDict_Type, 42
PyDict_Values(), 43
PyDictObject (C type), 42
PyErr_BadArgument(), 14
PyErr_BadInternalCall(), 15
PyErr_CheckSignals(), 15
PyErr_Clear(), 13
<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PyLong</td>
<td>Methods</td>
</tr>
<tr>
<td>FromUnsignedLong()</td>
<td>44</td>
</tr>
<tr>
<td>PyLongType</td>
<td>44</td>
</tr>
<tr>
<td>PyLongObject</td>
<td>(C type), 44</td>
</tr>
<tr>
<td>PyMapping</td>
<td>Methods</td>
</tr>
<tr>
<td>Check()</td>
<td>28</td>
</tr>
<tr>
<td>DelItem()</td>
<td>28</td>
</tr>
<tr>
<td>DelItemString()</td>
<td>28</td>
</tr>
<tr>
<td>GetItemString()</td>
<td>28</td>
</tr>
<tr>
<td>HasKey()</td>
<td>28</td>
</tr>
<tr>
<td>HasKeyString()</td>
<td>28</td>
</tr>
<tr>
<td>Items()</td>
<td>28</td>
</tr>
<tr>
<td>Keys()</td>
<td>28</td>
</tr>
<tr>
<td>Length()</td>
<td>28</td>
</tr>
<tr>
<td>SetItemString()</td>
<td>28</td>
</tr>
<tr>
<td>Values()</td>
<td>28</td>
</tr>
<tr>
<td>PyMem</td>
<td>Methods</td>
</tr>
<tr>
<td>Del()</td>
<td>60</td>
</tr>
<tr>
<td>Free()</td>
<td>60</td>
</tr>
<tr>
<td>Malloc()</td>
<td>60</td>
</tr>
<tr>
<td>New()</td>
<td>60</td>
</tr>
<tr>
<td>Realloc()</td>
<td>60</td>
</tr>
<tr>
<td>Resize()</td>
<td>60</td>
</tr>
<tr>
<td>PyMethodDef</td>
<td>Methods</td>
</tr>
<tr>
<td>(C type), 64</td>
<td></td>
</tr>
<tr>
<td>PyModule</td>
<td>Methods</td>
</tr>
<tr>
<td>AddIntConstant()</td>
<td>48</td>
</tr>
<tr>
<td>AddObject()</td>
<td>48</td>
</tr>
<tr>
<td>AddStringConstant()</td>
<td>48</td>
</tr>
<tr>
<td>Check()</td>
<td>48</td>
</tr>
<tr>
<td>GetDict()</td>
<td>48</td>
</tr>
<tr>
<td>GetFilename()</td>
<td>48</td>
</tr>
<tr>
<td>GetName()</td>
<td>48</td>
</tr>
<tr>
<td>New()</td>
<td>48</td>
</tr>
<tr>
<td>Type</td>
<td>48</td>
</tr>
<tr>
<td>PyNumber</td>
<td>Methods</td>
</tr>
<tr>
<td>Absolute()</td>
<td>24</td>
</tr>
<tr>
<td>Add()</td>
<td>23</td>
</tr>
<tr>
<td>And()</td>
<td>24</td>
</tr>
<tr>
<td>Check()</td>
<td>23</td>
</tr>
<tr>
<td>Coerce()</td>
<td>26</td>
</tr>
<tr>
<td>Divide()</td>
<td>24</td>
</tr>
<tr>
<td>Divmod()</td>
<td>24</td>
</tr>
<tr>
<td>Float()</td>
<td>26</td>
</tr>
<tr>
<td>InPlaceAdd()</td>
<td>25</td>
</tr>
<tr>
<td>InPlaceAnd()</td>
<td>25</td>
</tr>
<tr>
<td>InPlaceDivide()</td>
<td>25</td>
</tr>
<tr>
<td>InPlaceLshift()</td>
<td>25</td>
</tr>
<tr>
<td>InPlaceMultiply()</td>
<td>25</td>
</tr>
<tr>
<td>InPlaceOr()</td>
<td>26</td>
</tr>
<tr>
<td>InPlacePower()</td>
<td>25</td>
</tr>
<tr>
<td>InPlaceRemainder()</td>
<td>25</td>
</tr>
<tr>
<td>InPlaceRshift()</td>
<td>25</td>
</tr>
<tr>
<td>InPlaceSubtract()</td>
<td>25</td>
</tr>
<tr>
<td>InPlaceXor()</td>
<td>26</td>
</tr>
<tr>
<td>Int()</td>
<td>26</td>
</tr>
<tr>
<td>Invert()</td>
<td>24</td>
</tr>
<tr>
<td>Long()</td>
<td>26</td>
</tr>
<tr>
<td>Lshift()</td>
<td>24</td>
</tr>
<tr>
<td>Multiply()</td>
<td>24</td>
</tr>
<tr>
<td>Negative()</td>
<td>24</td>
</tr>
<tr>
<td>Or()</td>
<td>25</td>
</tr>
<tr>
<td>Positive()</td>
<td>24</td>
</tr>
<tr>
<td>Power()</td>
<td>24</td>
</tr>
<tr>
<td>Remainder()</td>
<td>24</td>
</tr>
<tr>
<td>Rshift()</td>
<td>24</td>
</tr>
<tr>
<td>Subtract()</td>
<td>24</td>
</tr>
<tr>
<td>Xor()</td>
<td>25</td>
</tr>
<tr>
<td>PyObject</td>
<td>Methods</td>
</tr>
<tr>
<td>AsFileDescriptor()</td>
<td>23</td>
</tr>
<tr>
<td>CallFunction()</td>
<td>22</td>
</tr>
<tr>
<td>CallMethod()</td>
<td>23</td>
</tr>
<tr>
<td>CallObject()</td>
<td>22</td>
</tr>
<tr>
<td>Cmp()</td>
<td>22</td>
</tr>
<tr>
<td>Compare()</td>
<td>22</td>
</tr>
<tr>
<td>DEL()</td>
<td>63</td>
</tr>
<tr>
<td>Del()</td>
<td>63</td>
</tr>
<tr>
<td>DelAttr()</td>
<td>21</td>
</tr>
<tr>
<td>DelAttrString()</td>
<td>21</td>
</tr>
<tr>
<td>DelItem()</td>
<td>23</td>
</tr>
<tr>
<td>GC Fini()</td>
<td>66</td>
</tr>
<tr>
<td>GC Init()</td>
<td>66</td>
</tr>
<tr>
<td>GetAttr()</td>
<td>21</td>
</tr>
<tr>
<td>GetAttrString()</td>
<td>21</td>
</tr>
<tr>
<td>GetItem()</td>
<td>23</td>
</tr>
<tr>
<td>HasAttr()</td>
<td>21</td>
</tr>
<tr>
<td>HasAttrString()</td>
<td>21</td>
</tr>
<tr>
<td>Hash()</td>
<td>23</td>
</tr>
<tr>
<td>Init()</td>
<td>63</td>
</tr>
<tr>
<td>InitVar()</td>
<td>63</td>
</tr>
<tr>
<td>IsInstance()</td>
<td>22</td>
</tr>
<tr>
<td>IsSubclass()</td>
<td>22</td>
</tr>
<tr>
<td>IsTrue()</td>
<td>23</td>
</tr>
<tr>
<td>Length()</td>
<td>23</td>
</tr>
<tr>
<td>NEW()</td>
<td>63</td>
</tr>
<tr>
<td>NEW VAR()</td>
<td>63</td>
</tr>
<tr>
<td>New()</td>
<td>63</td>
</tr>
<tr>
<td>NewVar()</td>
<td>63</td>
</tr>
<tr>
<td>Print()</td>
<td>21</td>
</tr>
<tr>
<td>Repr()</td>
<td>22</td>
</tr>
<tr>
<td>SetAttr()</td>
<td>21</td>
</tr>
<tr>
<td>SetAttrString()</td>
<td>21</td>
</tr>
<tr>
<td>SetItem()</td>
<td>23</td>
</tr>
<tr>
<td>Str()</td>
<td>22</td>
</tr>
<tr>
<td>Type()</td>
<td>23</td>
</tr>
<tr>
<td>Unicode()</td>
<td>22</td>
</tr>
<tr>
<td>PyOS</td>
<td>Methods</td>
</tr>
<tr>
<td>AfterFork()</td>
<td>17</td>
</tr>
<tr>
<td>CheckStack()</td>
<td>17</td>
</tr>
<tr>
<td>GetLastModificationTime()</td>
<td>17</td>
</tr>
<tr>
<td>getsig()</td>
<td>17</td>
</tr>
<tr>
<td>setsig()</td>
<td>17</td>
</tr>
<tr>
<td>PyParser</td>
<td>Methods</td>
</tr>
<tr>
<td>SimpleParseFile()</td>
<td>9</td>
</tr>
<tr>
<td>SimpleParseString()</td>
<td>9</td>
</tr>
</tbody>
</table>
PyUnicode_EncodeMBCS(), 37
PyUnicode_EncodeRawUnicodeEscape(), 35
PyUnicode_EncodeUnicodeEscape(), 35
PyUnicode_EncodeUTF16(), 35
PyUnicode_EncodeUTF8(), 34
PyUnicode_Find(), 38
PyUnicode_Format(), 38
PyUnicode_FromEncodedObject(), 33
PyUnicode_FromObject(), 33
PyUnicode_FromUnicode(), 33
PyUnicode_FromWideChar(), 33
PyUnicode_GET_DATA_SIZE(), 32
PyUnicode_GET_SIZE(), 32
PyUnicode_GetSize(), 33
PyUnicode_Join(), 38
PyUnicode_Replace(), 38
PyUnicode_Split(), 37
PyUnicode_Splitlines(), 38
PyUnicode_Tailmatch(), 38
PyUnicode_Translate(), 38
PyUnicode_TranslateCharmap(), 37
PyUnicode_Type, 32
PyUnicodeObject (C type), 31

R
realloc(), 59
reload() (built-in function), 18
repr() (built-in function), 22
reexec (standard module), 18

S
search
    path, module, 8, 51, 53
sequence
    object, 30
set_all(), 4
setcheckinterval () (in module sys), 54
setvbuf (), 47
SIGINT, 15
signal (built-in module), 15
softspace (file attribute), 47
stderr (in module sys), 51
stdin (in module sys), 51
stdout (in module sys), 51
str () (built-in function), 22
strerror (), 14
string
    object, 30
StringType (in module types), 30
_frozen (C type), 19
_inittab (C type), 19
sum_list (), 5
sum_sequence (), 5, 6