Python/C API Reference Manual

Guido van Rossum
Corporation for National Research Initiatives (CNRI)
1895 Preston White Drive, Reston, Va 20191, USA
E-mail: guido@CNRI.Reston.Va.US, guido@python.org

December 12, 1997
Release 1.5b2

Abstract

This manual documents the API used by C (or 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.

Contents

Chapter 1

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. Python 1.5 introduces a number of new API functions as well as some changes to the build process that make embedding much simpler. This manual describes the 1.5 state of affair.

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 header files: stdio.h, string.h, errno.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 only. 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.

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 (obviously!) 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. 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.

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 Language 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 iff 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 severly 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(a)` to increment an object's reference count by one, and `Py_DECREF(a)` to decrement it by one. The decref macro is considerably more complex than the incref one, since it must check whether the reference count becomes zero and then cause the object's deallocator, which 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, and so on, if this is a compound object type such as a list. 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 oject'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 holding 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 expelained 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 to ownership is transferred, the caller is said to *borrow* the reference. Nothing needs to be done for a borrowed reference.

Conversely, when calling a function passes it 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 it 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 anyway):

```
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; PyObject_SetItem() refuses 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:

```
PyObject *l, *x;
l = PyList_New(3);
x = PyInt_FromLong(1L);
PyObject_SetItem(l, 0, x); Py_DECREF(x);
x = PyInt_FromLong(2L);
PyObject_SetItem(l, 1, x); Py_DECREF(x);
x = PyString_FromString("three");
PyObject_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:

```
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 referece 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) *don't enter into it!* Thus, if you extract an item from a list using `PyList_GetItem()`, yo 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()`, 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 = PyObject_Size(list);
    if (n < 0)
        return -1; /* Has no length */
    for (i = 0; i < n; i++) {
        item = PySequence_GetItem(list, i);
        if (item == NULL)
            return -1; /* Not a sequence, or other failure */
        if (PyInt_Check(item))
            total += PyInt_AsLong(item);
        Py_DECREF(item); /* Discared 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. 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, till 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 on 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 object `sys.exc_type`, `sys.exc_value`, `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 interpreter, 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 discards any object references that it owns, and returns an error indicator, but it should *not* set another exception – that would overwrite the exception that was just raised, and lose important reason 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:

```
def incr_item(dict, key):
    try:
        item = dict[key]
    except KeyError:
        item = 0
    return 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 (faulure) */

    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 succesful.

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/python1.5` (replacing `1.5` with the current interpreter version) 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/python1.5`. (In fact, this particular path is also the "fallback" location, used when no executable file named `python` is found along `$PATH`.) The user can override this behavior by setting the environment variable `$PYTHONHOME`, or insert additional directories in front of the standard path by setting `$PYTHONPATH`.

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

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

Chapter 2

Basic Utilities

XXX These utilities should be moved to some other section...

`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.

`void Py_Exit(`*int status*`)`

Exit the current process. This calls `Py_Finalize()` and then calls the standard C library function `exit(0)`.

`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.

Chapter 3

Reference Counting

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

`void Py_INCREF(`*PyObject *o*`)`
Increment the reference count for object `o`. The object must not be `NULL`; if you aren't sure that it isn't `NULL`, use `Py_XINCREF()`.

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

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

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

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

The following functions or macros are only for internal use: `_Py_Dealloc`, `_Py_ForgetReference`, `_Py_NewReference`, as well as the global variable `_Py_RefTotal`.

XXX Should mention Py_Malloc(), Py_Realloc(), Py_Free(), PyMem_Malloc(), PyMem_Realloc(), PyMem_Free(), PyMem_NEW(), PyMem_RESIZE(), PyMem_DEL(), PyMem_XDEL().

Chapter 4

Exception Handling

The functions 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 of 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.

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

`PyObject * PyErr_Occurred()`
: Test whether the error indicator is set. If set, return the exception `type` (the first argument to the last call to one of the `PyErr_Set*()` functions or to `PyErr_Restore()`). If not set, return `NULL`. You do not own a reference to the return value, so you do not need to `Py_DECREF()` it. Note: do not compare the return value to a specific exception; use `PyErr_ExceptionMatches` instead, shown below.

`int PyErr_ExceptionMatches(`*PyObject *exc*`)`
: **(NEW in 1.5a4!)**
Equivalent to `PyErr_GivenExceptionMatches(PyErr_Occurred(),` *exc*`)`. This should only be called when an exception is actually set.

`int PyErr_GivenExceptionMatches(`*PyObject *given, PyObject *exc*`)`
: **(NEW in 1.5a4!)** Return true if the *given* exception matches the exception in *exc*. If *exc* is a class object, this also returns true when *given* is a subclass. If *exc* is a tuple, all exceptions in the tuple (and recursively in subtuples) are searched for a match. This should only be called when an exception is actually set.

`void PyErr_NormalizeException(`*PyObject**exc, PyObject**val, PyObject**tb*`)`
: **(NEW in 1.5a4!)** 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.

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

`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.

`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.

`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.

`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.

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

`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.

`PyObject * PyErr_NoMemory()`
: 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.

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

can write `return PyErr_NoMemory();` when the system call returns an error.

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

`int PyErr_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 `KeyboadInterrupt` exception. If an exception is raised the error indicator is set and the function returns 1; otherwise the function returns 0. The error indicator may or may not be cleared if it was previously set.

`void PyErr_SetInterrupt()`
: This function is obsolete (XXX or platform dependent?). It simulates the effect of a `SIGINT` signal arriving – the next time `PyErr_CheckSignals()` is called, `KeyboadInterrupt` will be raised.

`PyObject * PyErr_NewException(`*char *name, PyObject *base, PyObject *dict*`)`
: **(NEW in 1.5a4!)** 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`. Normally, this creates a class object derived from the root for all exceptions, the built-in name `Exception` (accessible in C as `PyExc_Exception`). In this case 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). When the user has specified the `-X` command line option to use string exceptions, for backward compatibility, or when the *base* argument is not a class object (and not `NULL`), a string object created from the entire *name* argument is returned. 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.

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 string objects. For completeness, here are all the variables (the first four are new in Python 1.5a4): `PyExc_Exception`, `PyExc_StandardError`, `PyExc_ArithmeticError`, `PyExc_LookupError`, `PyExc_AssertionError`, `PyExc_AttributeError`, `PyExc_EOFError`, `PyExc_FloatingPointError`, `PyExc_IOError`, `PyExc_ImportError`, `PyExc_IndexError`, `PyExc_KeyError`, `PyExc_KeyboardInterrupt`, `PyExc_MemoryError`, `PyExc_NameError`, `PyExc_OverflowError`, `PyExc_RuntimeError`, `PyExc_SyntaxError`, `PyExc_SystemError`, `PyExc_SystemExit`, `PyExc_TypeError`, `PyExc_ValueError`, `PyExc_ZeroDivisionError`.

Chapter 5

Utilities

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 `name` 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()`.

5.2 Importing modules

`PyObject * PyImport_ImportModule` (*char *name*)

This is a simplified interface to `PyImport_ImportModuleEx` below, leaving the *globals* and *locals* arguments set to `NULL`. When the *name* argument contains a dot (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).

`PyObject * PyImport_ImportModuleEx` (*char *name, PyObject *globals, PyObject *locals, PyObject *fromlist*)

(NEW in 1.5a4!) 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.

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

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

`PyObject * PyImport_AddModule(`*char *name*`)`
: Return 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. Because the former action is most common, this does not return a new reference, and you do not own the returned reference. 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 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.

`extern PyObject * _PyImport_FindExtension`
: char *, char * For internal use only.

`extern PyObject * _PyImport_FixupExtension`
: char *, char * For internal use only.

`int PyImport_ImportFrozenModule(`*char **`)`
: Load a frozen module. 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 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.

Chapter 6

Debugging

XXX Explain Py_DEBUG, Py_TRACE_REFS, Py_REF_DEBUG.

Chapter 7

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.

`int PyRun_AnyFile(`*FILE *, char **`)`

`int PyRun_SimpleString(`*char **`)`

`int PyRun_SimpleFile(`*FILE *, char **`)`

`int PyRun_InteractiveOne(`*FILE *, char **`)`

`int PyRun_InteractiveLoop(`*FILE *, char **`)`

`struct _node * PyParser_SimpleParseString(`*char *, int*`)`

`struct _node * PyParser_SimpleParseFile(`*FILE *, char *, int*`)`

`PyObject *PyRun_String(`*char *, int, PyObject *, PyObject **`)`

`PyObject *PyRun_File(`*FILE *, char *, int, PyObject *, PyObject **`)`

`PyObject *Py_CompileString(`*char *, char *, int*`)`

Chapter 8

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 flag a Python exception.

8.1 Object Protocol

`int PyObject_Print` (*PyObject *o, FILE *fp, int flags*)
: Print an object `o`, on file `fp`. Returns -1 on error The flags argument is used to enable certain printing options. The only option currently supported is `Py_Print_RAW`.

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

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

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

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

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

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

`int PyObject_DelAttrString` (*PyObject *o, char *attr_name*)
: Delete attribute named `attr_name`, for object `o`. Returns -1 on failure. This is the equivalent of the

Python statement: `del o.attr_name`.

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

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

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

`PyObject* PyObject_Repr(`*PyObject *o*`)`
: Compute the 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 the 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.

`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(o, 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 mkvalue-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(o,args)`.

`PyObject* PyObject_CallMethod(`*PyObject *o, char *m, char *format, ...*`)`
: Call the method named `m` of object `o` with a variable number of C arguments. The C arguments are described by a mkvalue 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.

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

`int PyObject_IsTrue(`*PyObject *o*`)`
: Returns 1 if the object `o` is considered to be true, and 0 otherwise. This is equivalent to the Python expression: `not not o`. This function always succeeds.

`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)`.

`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)`.

`PyObject* PyObject_GetItem(`*PyObject *o, PyObject *key*`)`
: Return element of `o` corresponding to the object `key` or `NULL` on failure. This is the equivalent of the Python expression: `o[key]`.

`int PyObject_SetItem(`*PyObject *o, PyObject *key, PyObject *v*`)`
: Map the object `key` to the value `v`. Returns -1 on failure. This is the equivalent of the Python statement: `o[key]=v`.

`int PyObject_DelItem(`*PyObject *o, PyObject *key, PyObject *v*`)`
: Delete the mapping for `key` from `*o`. Returns -1 on failure. This is the equivalent of the Python statement: `del o[key]`.

8.2 Number Protocol

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

`PyObject* PyNumber_Add(`*PyObject *o1, PyObject *o2*`)`
: Returns the result of adding `o1` and `o2`, or null on failure. This is the equivalent of the Python expression: `o1+o2`.

`PyObject* PyNumber_Subtract(`*PyObject *o1, PyObject *o2*`)`
: Returns the result of subtracting `o2` from `o1`, or null on failure. This is the equivalent of the Python expression: `o1-o2`.

`PyObject* PyNumber_Multiply(`*PyObject *o1, PyObject *o2*`)`
: Returns the result of multiplying `o1` and `o2`, or null on failure. This is the equivalent of the Python expression: `o1*o2`.

`PyObject* PyNumber_Divide(`*PyObject *o1, PyObject *o2*`)`
: Returns the result of dividing `o1` by `o2`, or null on failure. This is the equivalent of the Python expression: `o1/o2`.

`PyObject* PyNumber_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.

`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 result of ”anding” `o2` and `o2` on success and `NULL` on failure. This is the equivalent of the Python expression: `o1 and 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 result of `o1` and `o2` on success, or `NULL` on failure. This is the equivalent of the Python expression: `o1 or o2`.

`PyObject* PyNumber_Coerce(`*PyObject *o1, PyObject *o2*`)`
: This function takes the addresses of two variables of type `PyObject*`.

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

`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)`.

8.3 Sequence protocol

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

`PyObject* PySequence_Concat(`*PyObject *o1, PyObject *o2*`)`
: Return the concatination 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_GetItem(`*PyObject *o, int i*`)`
: Return the ith 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 `v`. 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` on `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_In(*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 for which o[i]==value. On error, return -1. This is equivalent to the Python expression: o.index(value).

8.4 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 sequence 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().

int PyMapping_Clear(*PyObject *o*)
Make object o empty. Returns 1 on success and 0 on failure. This is equivalent to the Python statement: for key in o.keys(): del o[key]

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]`.

`PyObject* 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`.

8.5 Constructors

`PyObject* PyFile_FromString(`*char *file_name, char *mode*`)`
On success, returns a new file object that is opened on the file given by `file_name`, with a file mode given by `mode`, where `mode` has the same semantics as the standard C routine, fopen. On failure, return -1.

`PyObject* PyFile_FromFile(`*FILE *fp, char *file_name, char *mode, int close_on_del*`)`
Return a new file object for an already opened standard C file pointer, `fp`. A file name, `file_name`, and open mode, `mode`, must be provided as well as a flag, `close_on_del`, that indicates whether the file is to be closed when the file object is destroyed. On failure, return -1.

`PyObject* PyFloat_FromDouble(`*double v*`)`
Returns a new float object with the value `v` on success, and `NULL` on failure.

`PyObject* PyInt_FromLong(`*long v*`)`
Returns a new int object with the value `v` on success, and `NULL` on failure.

`PyObject* PyList_New(`*int l*`)`
Returns a new list of length `l` on success, and `NULL` on failure.

`PyObject* PyLong_FromLong(`*long v*`)`
Returns a new long object with the value `v` on success, and `NULL` on failure.

`PyObject* PyLong_FromDouble(`*double v*`)`
Returns a new long object with the value `v` on success, and `NULL` on failure.

`PyObject* PyDict_New()`
Returns a new empty dictionary on success, and `NULL` on failure.

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

`PyObject* PyString_FromStringAndSize(`*char *v, int l*`)`
Returns a new string object with the value `v` and length `l` on success, and `NULL` on failure.

`PyObject* PyTuple_New(`*int l*`)`
Returns a new tuple of length `l` on success, and `NULL` on failure.

Chapter 9

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; e.g. to check that an object is a dictionary, use `PyDict_Check()`.

Chapter 10

Defining New Object Types

`PyObject * _PyObject_New(`*PyTypeObject *type*`)`

`PyObject * _PyObject_NewVar(`*PyTypeObject *type, int size*`)`

`TYPE _PyObject_NEW(`*TYPE, PyTypeObject **`)`

`TYPE _PyObject_NEW_VAR(`*TYPE, PyTypeObject *, int size*`)`

Chapter 11

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()`

(NEW in 1.5a4!) 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()`

(NEW in 1.5a3!) 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 applcation calls `Py_Initialize()` and

Py_Finalize() more than once.

PyThreadState * Py_NewInterpreter()

(NEW in 1.5a3!) 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 as 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 function *is* called again.

Bugs and caveats: Because sub-interpreters (and the main interpreter) are part of the same process, the insulation between them isn't perfect – for example, using low-level file operations like os.close() they can (accidentally or maliciously) affect each other's open files. Because of the way extensions are shared between (sub-)interpreters, some extensions may not work properly; this is especially likely when the extension makes use of (static) global variables, or when the extension manipulates its module's dictionary after its initialization. It is possible to insert objects created in one sub-interpreter into a namespace of another sub-interpreter; this should be done with great care to avoid sharing user-defined functions, methods, instances or classes between sub-interpreters, since import operations executed by such objects may affect the wrong (sub-)interpreter's dictionary of loaded modules. (XXX This is a hard-to-fix bug that will be addressed in a future release.)

void Py_EndInterpreter(*PyThreadState *tstate*)

(NEW in 1.5a3!) 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.

void Py_SetProgramName(*char *name*)

(NEW in 1.5a3!) 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.

`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.

`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.

`char * Py_GetExecPrefix()`

Return the "exec-prefix" for installed platform-*de*pendent 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.

`char * Py_GetProgramFullPath()`

(NEW in 1.5a3!) 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`.

`char * Py_GetPath()`

Return the default module search path; this is computed from the program name (set by `Py_SetProgramName()` above) and some environment variables. The returned string consists of a series of directory names separated by a platform dependent delimiter character. The delimiter character is `':'` on Unix, `';'` 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 * Py_GetVersion()`

Return the version of this Python interpreter. This is a string that looks something like

```
"1.5a3 (#67, Aug 1 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 * Py_GetPlatform()`

Return the platform identifier for the current platform. On Unix, this is formed from the "official" name of the operating system, converted to lower case, followed by the major revision number; e.g., for Solaris 2.x, which is also known as SunOS 5.x, the value is `"sunos5"`. On 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 * Py_GetCopyright()`

Return the official copyright string for the current Python version, for example

```
"Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as the list `sys.copyright`.

`const char * Py_GetCompiler()`

Return an indication of the compiler used to build the current Python version, in square brackets, for example

```
"[GCC 2.7.2.2]"
```

The returned string points into static storage; the caller should not modify its value. The value is available to Python code as part of the variable `sys.version`.

`const char * Py_GetBuildInfo()`

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`.

`int PySys_SetArgv(`*int argc, char **argv*`)`

11.1 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 proram: 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 release 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:

```
Py_BEGIN_ALLOW_THREADS
...Do some blocking I/O operation...
Py_END_ALLOW_THREADS
```

The BEGIN macro opens a new block and declares a hidden local variable; the END macro closes the block. Another advantage of using these two macros is that when Python is compiled without thread support, they are defined empty, thus saving the thread state and lock manipulations.

When thread support is enabled, the block above expands to the following code:

```
{
    PyThreadState *_save;
    _save = PyEval_SaveThread();
    ...Do some blocking I/O operation...
    PyEval_RestoreThread(_save);
}
```

Using even lower level primitives, we can get roughly the same effect as follows:

```
{
    PyThreadState *_save;
    _save = PyThreadState_Swap(NULL);
    PyEval_ReleaseLock();
    ...Do some blocking I/O operation...
    PyEval_AcquireLock();
    PyThreadState_Swap(_save);
}
```

There are some subtle differences; in particular, `PyEval_RestoreThread()` saves and restores the value of the global variable `errno`, since the lock manipulation does not guarantee that `errno` is left alone. Also, when thread support is disabled, `PyEval_SaveThread()` and `PyEval_RestoreThread()` don't manipulate the lock; in this case, `PyEval_ReleaseLock()` and `PyEval_AcquireLock()` are not available. (This is done so that dynamically loaded extensions compiled with thread support enabled can be loaded by an interpreter that was compiled with disabled thread support.)

The global interpreter lock is used to protect the pointer to the current thread state. When releasing the lock and saving the thread state, the current thread state pointer must be retrieved before the lock is released (since another thread could immediately acquire the lock and store its own thread state in the global variable). Reversely, 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 hold 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).

XXX More?

`PyInterpreterState`

(NEW in 1.5a3!) This data structure represents the state shared by a number of cooperating threads. Threads belonging to the same interpreter share their module administration and a few other internal items. There are no public members in this structure.

Threads belonging to different interpreters initially share nothing, except process state like available memory, open file descriptors and such. The global interpreter lock is also shared by all threads, regardless of to which interpreter they belong.

`PyThreadState`

(NEW in 1.5a3!) 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()`
(NEW in 1.5a3!) 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()`
(NEW in 1.5a3!) 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*`)`
(NEW in 1.5a3!) 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*`)`
(NEW in 1.5a3!) 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()`
(Different return type in 1.5a3!) 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*`)`
(Different argument type in 1.5a3!) 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.)

`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. They are all new in 1.5a3.

`PyInterpreterState * PyInterpreterState_New()`
: Create a new interpreter state object. The interpreter lock must be held.

`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 must be held.

`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.

11.2 Defining New Object Types

XXX To be done:

PyObject, PyVarObject

PyObject_HEAD, PyObject_HEAD_INIT, PyObject_VAR_HEAD

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

PyNumberMethods

PySequenceMethods

PyMappingMethods

PyBufferProcs

PyTypeObject

DL_IMPORT

PyType_Type

Py*_Check

Py_None, _Py_NoneStruct

_PyObject_New, _PyObject_NewVar

PyObject_NEW, PyObject_NEW_VAR

Chapter 12

Specific Data Types

This chapter describes the functions that deal with specific types of Python objects. It is structured like the "family tree" of Python object types.

12.1 Fundamental Objects

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

12.1.1 Type Objects

`PyTypeObject`

`PyObject * PyType_Type`

12.1.2 The None Object

`PyObject * Py_None`
XXX macro

12.2 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.

12.2.1 String Objects

`PyStringObject`
This subtype of `PyObject` represents a Python string object.

`PyTypeObject PyString_Type`
This instance of `PyTypeObject` represents the Python string type.

`int PyString_Check` (*PyObject *o*)

`PyObject * PyString_FromStringAndSize` (*const char *, int*)

`PyObject * PyString_FromString` (*const char **)

`int PyString_Size` (*PyObject **)

`char * PyString_AsString` (*PyObject **)

`void PyString_Concat` (*PyObject **, PyObject **)

`void PyString_ConcatAndDel` (*PyObject **, PyObject **)

`int _PyString_Resize` (*PyObject **, int*)

`PyObject * PyString_Format` (*PyObject *, PyObject **)

`void PyString_InternInPlace` (*PyObject ***)

`PyObject * PyString_InternFromString` (*const char **)

`char * PyString_AS_STRING` (*PyStringObject **)

`int PyString_GET_SIZE` (*PyStringObject **)

12.2.2 Tuple Objects

`PyTupleObject`
This subtype of `PyObject` represents a Python tuple object.

`PyTypeObject PyTuple_Type`
This instance of `PyTypeObject` represents the Python tuple type.

`int PyTuple_Check` (*PyObject *p*)
Return true if the argument is a tuple object.

`PyTupleObject * PyTuple_New` (*int s*)
Return a new tuple object of size `s`

`int PyTuple_Size` (*PyTupleObject *p*)
akes a pointer to a tuple object, and returns the size of that tuple.

`PyObject * PyTuple_GetItem` (*PyTupleObject *p, int pos*)
returns the object at position `pos` in the tuple pointed to by `p`.

`PyObject * PyTuple_GET_ITEM` (*PyTupleObject *p, int pos*)
does the same, but does no checking of it's arguments.

`PyTupleObject * PyTuple_GetSlice` (*PyTupleObject *p, int low, int high*)
takes a slice of the tuple pointed to by `p` from `low` to `high` and returns it as a new tuple.

`int PyTuple_SetItem` (*PyTupleObject *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.

`void PyTuple_SET_ITEM` (*PyTupleObject *p, int pos, PyObject *o*)
does the same, but does no error checking, and should *only* be used to fill in brand new tuples.

`PyTupleObject * _PyTuple_Resize` (*PyTupleObject *p, int new, int last_is_sticky*) can be used to resize a tuple. Because tuples are *supposed* to be immutable, this should only be used if there is only one module referencing the object. Do *not* use this if the tuple may already be known to some other part of the code. `last_is_sticky` is a flag - if set, the tuple will grow or shrink at the front, otherwise it will grow or shrink at the end. Think of this as destroying the old tuple and creating a new one, only more efficiently.

12.2.3 List Objects

`PyListObject`
This subtype of `PyObject` represents a Python list object.

`PyTypeObject PyList_Type`
This instance of `PyTypeObject` represents the Python list type.

`int PyList_Check(`*PyObject *p*`)`
returns true if it's argument is a `PyListObject`

`PyObject * PyList_New(`*int size*`)`

`int PyList_Size(`*PyObject **`)`

`PyObject * PyList_GetItem(`*PyObject *, int*`)`

`int PyList_SetItem(`*PyObject *, int, PyObject **`)`

`int PyList_Insert(`*PyObject *, int, PyObject **`)`

`int PyList_Append(`*PyObject *, PyObject **`)`

`PyObject * PyList_GetSlice(`*PyObject *, int, int*`)`

`int PyList_SetSlice(`*PyObject *, int, int, PyObject **`)`

`int PyList_Sort(`*PyObject **`)`

`int PyList_Reverse(`*PyObject **`)`

`PyObject * PyList_AsTuple(`*PyObject **`)`

`PyObject * PyList_GET_ITEM(`*PyObject *list, int i*`)`

`int PyList_GET_SIZE(`*PyObject *list*`)`

12.3 Mapping Objects

12.3.1 Dictionary Objects

`PyDictObject`
This subtype of `PyObject` represents a Python dictionary object.

`PyTypeObject PyDict_Type`
This instance of `PyTypeObject` represents the Python dictionary type.

`int PyDict_Check(`*PyObject *p*`)`
returns true if it's argument is a PyDictObject

`PyDictObject * PyDict_New()`
: returns a new empty dictionary.

`void PyDict_Clear(`*PyDictObject *p*`)`
: empties an existing dictionary and deletes it.

`int PyDict_SetItem(`*PyDictObject *p, PyObject *key, PyObject *val*`)`
: inserts `value` into the dictionary with a key of `key`. Both `key` and `value` should be PyObjects, and `key` should be hashable.

`int PyDict_SetItemString(`*PyDictObject *p, char *key, PyObject *val*`)`
: inserts `value` into the dictionary using `key` as a key. `key` should be a char *

`int PyDict_DelItem(`*PyDictObject *p, PyObject *key*`)`
: removes the entry in dictionary `p` with key `key`. `key` is a PyObject.

`int PyDict_DelItemString(`*PyDictObject *p, char *key*`)`
: removes the entry in dictionary `p` which has a key specified by the `char *key`.

`PyObject * PyDict_GetItem(`*PyDictObject *p, PyObject *key*`)`
: returns the object from dictionary `p` which has a key `key`.

`PyObject * PyDict_GetItemString(`*PyDictObject *p, char *key*`)`
: does the same, but `key` is specified as a `char *`, rather than a `PyObject *`.

`PyListObject * PyDict_Items(`*PyDictObject *p*`)`
: returns a PyListObject containing all the items from the dictionary, as in the mapping method `items()` (see the Reference Guide)

`PyListObject * PyDict_Keys(`*PyDictObject *p*`)`
: returns a PyListObject containing all the keys from the dictionary, as in the mapping method `keys()` (see the Reference Guide)

`PyListObject * PyDict_Values(`*PyDictObject *p*`)`
: returns a PyListObject containing all the values from the dictionary, as in the mapping method `values()` (see the Reference Guide)

`int PyDict_Size(`*PyDictObject *p*`)`
: returns the number of items in the dictionary.

`int PyDict_Next(`*PyDictObject *p, int ppos, PyObject **pkey, PyObject **pvalue*`)`

12.4 Numeric Objects

12.4.1 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.

`int PyInt_Check(`*PyObject **`)`

`PyIntObject * PyInt_FromLong(`*long ival*`)` 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. :-)

`long PyInt_AS_LONG(`*PyIntObject *io*`)`
returns the value of the object `io`.

`long PyInt_AsLong(`*PyObject *io*`)`
will first attempt to cast the object to a PyIntObject, if it is not already one, and the return it's value.

`long PyInt_GetMax()`
returns the systems idea of the largest int it can handle (LONG_MAX, as defined in the system header files)

12.4.2 Long Integer Objects

`PyLongObject`
This subtype of `PyObject` represents a Python long integer object.

`PyTypeObject PyLong_Type`
This instance of `PyTypeObject` represents the Python long integer type.

`int PyLong_Check(`*PyObject *p*`)`
returns true if it's argument is a `PyLongObject`

`PyObject * PyLong_FromLong(`*long*`)`

`PyObject * PyLong_FromUnsignedLong(`*unsigned long*`)`

`PyObject * PyLong_FromDouble(`*double*`)`

`long PyLong_AsLong(`*PyObject **`)`

`unsigned long PyLong_AsUnsignedLong(`*PyObject* `)`

`double PyLong_AsDouble(`*PyObject **`)`

`PyObject * *PyLong_FromString(`*char *, char **, int*`)`

12.4.3 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.

`int PyFloat_Check(`*PyObject *p*`)`
returns true if it's argument is a `PyFloatObject`

`PyObject * PyFloat_FromDouble(`*double*`)`

`double PyFloat_AsDouble(`*PyObject **`)` `double PyFloat_AS_DOUBLE(`*PyFloatObject **`)`

12.4.4 Complex Number Objects

`Py_complex`
: typedef struct double real; double imag;

`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 it's argument is a `PyComplexObject`

`Py_complex _Py_c_sum(`*Py_complex, Py_complex*`)`

`Py_complex _Py_c_diff(`*Py_complex, Py_complex*`)`

`Py_complex _Py_c_neg(`*Py_complex*`)`

`Py_complex _Py_c_prod(`*Py_complex, Py_complex*`)`

`Py_complex _Py_c_quot(`*Py_complex, Py_complex*`)`

`Py_complex _Py_c_pow(`*Py_complex, Py_complex*`)`

`PyObject * PyComplex_FromCComplex(`*Py_complex*`)`

`PyObject * PyComplex_FromDoubles(`*double real, double imag*`)`

`double PyComplex_RealAsDouble(`*PyObject *op*`)`

`double PyComplex_ImagAsDouble(`*PyObject *op*`)`

`Py_complex PyComplex_AsCComplex(`*PyObject *op*`)`

12.5 Other Objects

12.5.1 File Objects

`PyFileObject`
: This subtype of `PyObject` represents a Python file object.

`PyTypeObject PyFile_Type`
: This instance of `PyTypeObject` represents the Python file type.

`int PyFile_Check(`*PyObject *p*`)`
: returns true if it's argument is a `PyFileObject`

`PyObject * PyFile_FromString(`*char *name, char *mode*`)`
: creates a new PyFileObject pointing to the file specified in `name` with the mode specified in `mode`

`PyObject * PyFile_FromFile(`*FILE *fp, char *name, char *mode, int (*close*`)`
:) creates a new PyFileObject from the already-open `fp`. The function `close` will be called when the

file should be closed.

`FILE * PyFile_AsFile(`*PyFileObject *p*`)`
: returns the file object associated with `p` as a `FILE *`

`PyStringObject * PyFile_GetLine(`*PyObject *p, int n*`)`
: undocumented as yet

`PyStringObject * PyFile_Name(`*PyObject *p*`)`
: returns the name of the file specified by `p` as a PyStringObject

`void PyFile_SetBufSize(`*PyFileObject *p, int n*`)`
: on systems with `setvbuf` only

`int PyFile_SoftSpace(`*PyFileObject *p, int newflag*`)`
: same as the file object method `softspace`

`int PyFile_WriteObject(`*PyObject *obj, PyFileObject *p*`)`
: writes object `obj` to file object `p`

`int PyFile_WriteString(`*char *s, PyFileObject *p*`)`
: writes string `s` to file object `p`

Index