The Very High Level Layer

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

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

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

int **Py_Main**(int *argc*, wchar_t ***argv*)

The main program for the standard interpreter. This is made available for programs which embed Python. The *argc* and *argv* parameters should be prepared exactly as those which are passed to a C program's `main()` function (converted to wchar_t according to the user's locale). It is important to note that the argument list may be modified (but the contents of the strings pointed to by the argument list are not). The return value will be `0` if the interpreter exits normally (i.e., without an exception), `1` if the interpreter exits due to an exception, or `2` if the parameter list does not represent a valid Python command line.

Note that if an otherwise unhandled `SystemExit` is raised, this function will not return `1`, but exit the process, as long as `Py_InspectFlag` is not set.

int **PyRun_AnyFile**(FILE **fp*, const char **filename*)

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving *closeit* set to `0` and *flags* set to *NULL*.

int **PyRun_AnyFileFlags**(FILE **fp*, const char **filename*, PyCompilerFlags **flags*)

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving the *closeit* argument set to `0`.

int **PyRun_AnyFileEx**(FILE **fp*, const char **filename*, int *closeit*)

This is a simplified interface to `PyRun_AnyFileExFlags()` below, leaving the *flags* argument set to *NULL*.

int **PyRun_AnyFileExFlags**(FILE **fp*, const char **filename*, int *closeit*, PyCompilerFlags **flags*)

If *fp* refers to a file associated with an interactive device (console or terminal input or Unix pseudo-terminal), return the value of `PyRun_InteractiveLoop()`, otherwise return the result of `PyRun_SimpleFile()`. *filename* is decoded from the filesystem encoding (`sys.getfilesystemencoding()`). If *filename* is *NULL*, this function uses `"???"` as the filename.

int **PyRun_SimpleString**(const char **command*)

This is a simplified interface to `PyRun_SimpleStringFlags()` below, leaving the *PyCompilerFlags** argument set to NULL.

int **PyRun_SimpleStringFlags**(const char **command*, PyCompilerFlags **flags*)

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

Note that if an otherwise unhandled `SystemExit` is raised, this function will not return `-1`, but exit the process, as long as `Py_InspectFlag` is not set.

int **PyRun_SimpleFile**(FILE **fp*, const char **filename*)

This is a simplified interface to `PyRun_SimpleFileExFlags()` below, leaving *closeit* set to `0` and *flags* set to *NULL*.

int **PyRun_SimpleFileEx**(FILE **fp*, const char **filename*, int *closeit*)

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

int **PyRun_SimpleFileExFlags**(FILE **fp*, const char **filename*, int *closeit*, PyCompilerFlags **flags*)

Similar to `PyRun_SimpleStringFlags()`, but the Python source code is read from *fp* instead of an in-memory string. *filename* should be the name of the file, it is decoded from the filesystem encoding (`sys.getfilesystemencoding()`). If *closeit* is true, the file is closed before PyRun_SimpleFileExFlags returns.

int **PyRun_InteractiveOne**(FILE **fp*, const char **filename*)

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

int **PyRun_InteractiveOneFlags**(FILE **fp*, const char **filename*, PyCompilerFlags **flags*)

Read and execute a single statement from a file associated with an interactive device according to the *flags* argument. The user will be prompted using `sys.ps1` and `sys.ps2`. *filename* is decoded from the filesystem encoding (`sys.getfilesystemencoding()`).

Returns `0` when the input was executed successfully, `-1` if there was an exception, or an error code from the `errcode.h` include file distributed as part of Python if there was a parse error. (Note that `errcode.h` is not included by `Python.h`, so must be included specifically if needed.)

int **`PyRun_InteractiveLoop`**(FILE **fp*, const char **filename*)

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

int **`PyRun_InteractiveLoopFlags`**(FILE **fp*, const char **filename*, PyCompilerFlags **flags*)

Read and execute statements from a file associated with an interactive device until EOF is reached. The user will be prompted using `sys.ps1` and `sys.ps2`. *filename* is decoded from the filesystem encoding (`sys.getfilesystemencoding()`). Returns `0` at EOF or a negative number upon failure.

int **`(*PyOS_InputHook)`**(void)

Can be set to point to a function with the prototype `int func(void)`. The function will be called when Python's interpreter prompt is about to become idle and wait for user input from the terminal. The return value is ignored. Overriding this hook can be used to integrate the interpreter's prompt with other event loops, as done in the `Modules/_tkinter.c` in the Python source code.

char* **`(*PyOS_ReadlineFunctionPointer)`**(FILE *, FILE *, const char *)

Can be set to point to a function with the prototype `char *func(FILE *stdin, FILE *stdout, char *prompt)`, overriding the default function used to read a single line of input at the interpreter's prompt. The function is expected to output the string *prompt* if it's not *NULL*, and then read a line of input from the provided standard input file, returning the resulting string. For example, The `readline` module sets this hook to provide line-editing and tab-completion features.

The result must be a string allocated by `PyMem_RawMalloc()` or `PyMem_RawRealloc()`, or *NULL* if an error occurred.

Changed in version 3.4: The result must be allocated by `PyMem_RawMalloc()` or `PyMem_RawRealloc()`, instead of being allocated by `PyMem_Malloc()` or `PyMem_Realloc()`.

struct _node* **PyParser_SimpleParseString**(const char **str*, int *start*)

This is a simplified interface to `PyParser_SimpleParseStringFlagsFilename()` below, leaving *filename* set to *NULL* and *flags* set to `0`.

struct _node* **PyParser_SimpleParseStringFlags**(const char **str*, int *start*, int *flags*)

This is a simplified interface to `PyParser_SimpleParseStringFlagsFilename()` below, leaving *filename* set to *NULL*.

struct _node* **PyParser_SimpleParseStringFlagsFilename**(const char **str*, const char **filename*, int *start*, int *flags*)

Parse Python source code from *str* using the start token *start* according to the *flags* argument. The result can be used to create a code object which can be evaluated efficiently. This is useful if a code fragment must be evaluated many times. *filename* is decoded from the filesystem encoding (`sys.getfilesystemencoding()`).

struct _node* **PyParser_SimpleParseFile**(FILE **fp*, const char **filename*, int *start*)

This is a simplified interface to `PyParser_SimpleParseFileFlags()` below, leaving *flags* set to `0`.

struct _node* **PyParser_SimpleParseFileFlags**(FILE **fp*, const char **filename*, int *start*, int *flags*)

Similar to `PyParser_SimpleParseStringFlagsFilename()`, but the Python source code is read from *fp* instead of an in-memory string.

PyObject* **PyRun_String**(const char **str*, int *start*, PyObject **globals*, PyObject **locals*)

Return value: New reference.

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

PyObject* **PyRun_StringFlags**(const char **str*, int *start*, PyObject **globals*, PyObject **locals*, PyCompilerFlags **flags*)

Return value: New reference.

Execute Python source code from *str* in the context specified by the objects *globals* and *locals* with the compiler flags specified by *flags*. *globals* must be a dictionary; *locals* can be any object that implements the mapping protocol. The parameter *start* specifies the start token that should be used to parse the source code.

Returns the result of executing the code as a Python object, or *NULL* if an exception was raised.

PyObject* **PyRun_File**(FILE *fp*, const char *filename*, int *start*, PyObject *globals*, PyObject *locals*)

Return value: New reference.

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

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

Return value: New reference.

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

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

Return value: New reference.

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

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

Return value: New reference.

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

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

Return value: New reference.

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

PyObject* **Py_CompileStringFlags**(const char *str*, const char *filename*, int *start*, PyCompilerFlags *flags*)

Return value: New reference.

This is a simplified interface to `Py_CompileStringExFlags()` below, with *optimize* set to `-1`.

PyObject* **Py_CompileStringObject**(const char *str*, PyObject *filename*, int *start*, PyCompilerFlags *flags*, int *optimize*)

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

The integer *optimize* specifies the optimization level of the compiler; a value of `-1` selects the optimization level of the interpreter as given by `-O` options. Explicit levels are `0` (no optimization; `__debug__` is true), `1` (asserts are removed, `__debug__` is false) or `2` (docstrings are removed too).

New in version 3.4.

PyObject* **`Py_CompileStringExFlags`**(const char **str*, const char **filename*, int *start*, PyCompilerFlags **flags*, int *optimize*)

Like `Py_CompileStringObject()`, but *filename* is a byte string decoded from the filesystem encoding (`os.fsdecode()`).

New in version 3.2.

PyObject* **`PyEval_EvalCode`**(PyObject **co*, PyObject **globals*, PyObject **locals*)

Return value: New reference.

This is a simplified interface to `PyEval_EvalCodeEx()`, with just the code object, and global and local variables. The other arguments are set to *NULL*.

PyObject* **`PyEval_EvalCodeEx`**(PyObject **co*, PyObject **globals*, PyObject **locals*, PyObject ***args*, int *argcount*, PyObject ***kws*, int *kwcount*, PyObject ***defs*, int *defcount*, PyObject **kwdefs*, PyObject **closure*)

Evaluate a precompiled code object, given a particular environment for its evaluation. This environment consists of a dictionary of global variables, a mapping object of local variables, arrays of arguments, keywords and defaults, a dictionary of default values for keyword-only arguments and a closure tuple of cells.

`PyFrameObject`

The C structure of the objects used to describe frame objects. The fields of this type are subject to change at any time.

PyObject* **`PyEval_EvalFrame`**(PyFrameObject **f*)

Evaluate an execution frame. This is a simplified interface to `PyEval_EvalFrameEx()`, for backward compatibility.

PyObject* **`PyEval_EvalFrameEx`**(PyFrameObject **f*, int *throwflag*)

This is the main, unvarnished function of Python interpretation. It is literally 2000 lines long. The code object associated with the execution frame *f* is executed, interpreting bytecode and executing calls as needed. The additional *throwflag* parameter can mostly be ignored - if true, then it causes an exception to immediately be thrown; this is used for the `throw()` methods of generator objects.

Changed in version 3.4: This function now includes a debug assertion to help ensure that it does not silently discard an active exception.

int **PyEval_MergeCompilerFlags**(PyCompilerFlags **cf*)

This function changes the flags of the current evaluation frame, and returns true on success, false on failure.

int **Py_eval_input**

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

int **Py_file_input**

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

int **Py_single_input**

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

struct **PyCompilerFlags**

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

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

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

int **CO_FUTURE_DIVISION**

This bit can be set in *flags* to cause division operator `/` to be interpreted as "true division" according to **PEP 238**.