Introduction ¶

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

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

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

Coding standards

If you're writing C code for inclusion in CPython, you **must** follow the guidelines and standards defined in **PEP 7**. These guidelines apply regardless of the version of Python you are contributing to. Following these conventions is not necessary for your own third party extension modules, unless you eventually expect to contribute them to Python.

Include Files

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

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

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

> **Note:** Since Python may define some pre-processor definitions which affect the standard headers on some systems, you *must* include `Python.h` before any standard headers are included.
>
> It is recommended to always define `PY_SSIZE_T_CLEAN` before including `Python.h`. See Parsing arguments and building values for a description of this macro.

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

and jeopardizes the portability of the user code to future Python versions, which may define additional names beginning with one of these prefixes.

The header files are typically installed with Python. On Unix, these are located in the directories *prefix*`/include/pythonversion/` and *exec_prefix*`/include/pythonversion/`, where `prefix` and `exec_prefix` are defined by the corresponding parameters to Python's **configure** script and *version* is `'%d.%d' % sys.version_info[:2]`. On Windows, the headers are installed in *prefix*`/include`, where `prefix` is the installation directory specified to the installer.

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

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

Useful macros

Several useful macros are defined in the Python header files. Many are defined closer to where they are useful (e.g. `Py_RETURN_NONE`). Others of a more general utility are defined here. This is not necessarily a complete listing.

Py_ABS(x)
: Return the absolute value of `x`.

  New in version 3.3.

Py_ALWAYS_INLINE
: Ask the compiler to always inline a static inline function. The compiler can ignore it and decides to not inline the function.

  It can be used to inline performance critical static inline functions when building Python in debug mode with function inlining disabled. For example, MSC disables function inlining when building in debug mode.

  Marking blindly a static inline function with Py_ALWAYS_INLINE can result in worse performances (due to increased code size for example). The compiler is usually smarter than the developer for the cost/benefit analysis.

  If Python is built in debug mode (if the Py_DEBUG macro is defined), the `Py_ALWAYS_INLINE` macro does nothing.

  It must be specified before the function return type. Usage:

  ```
  static inline Py_ALWAYS_INLINE int random(void) { return 4; }
  ```

  New in version 3.11.

Py_CHARMASK(c)
: Argument must be a character or an integer in the range [-128, 127] or [0, 255]. This macro returns `c` cast to an `unsigned char`.

Example:

```
Py_DEPRECATED(3.8) PyAPI_FUNC(int) Py_OldFunction(void);
```

Changed in version 3.8: MSVC support was added.

Py_GETENV(s)
Like getenv(s), but returns NULL if -E was passed on the command line (i.e. if Py_IgnoreEnvironmentFlag is set).

Py_MAX(x, y)
Return the maximum value between x and y.

New in version 3.3.

Py_MEMBER_SIZE(type, member)
Return the size of a structure (type) member in bytes.

New in version 3.6.

Py_MIN(x, y)
Return the minimum value between x and y.

New in version 3.3.

Py_NO_INLINE
Disable inlining on a function. For example, it reduces the C stack consumption: useful on LTO+PGO builds which heavily inline code (see bpo-33720).

Usage:

```
Py_NO_INLINE static int random(void) { return 4; }
```

New in version 3.11.

Py_STRINGIFY(x)
Convert x to a C string. E.g. Py_STRINGIFY(123) returns "123".

New in version 3.4.

Py_UNREACHABLE()
Use this when you have a code path that cannot be reached by design. For example, in the default: clause in a switch statement for which all possible values are covered in case statements. Use this in places where you might be tempted to put an assert(0) or abort() call.

In release mode, the macro helps the compiler to optimize the code, and avoids a warning about unreachable code. For example, the macro is implemented with __builtin_unreachable() on GCC in release mode.

A use for Py_UNREACHABLE() is following a call a function that never returns but that is not declared _Py_NO_RETURN.