

Extending and Embedding the Python Interpreter

This document describes how to write modules in C or C++ to extend the Python interpreter with new modules. Those modules can not only define new functions but also new object types and their methods. The document also describes how to embed the Python interpreter in another application, for use as an extension language. Finally, it shows how to compile and link extension modules so that they can be loaded dynamically (at run time) into the interpreter, if the underlying operating system supports this feature.

This document assumes basic knowledge about Python. For an informal introduction to the language, see [The Python Tutorial](#). [The Python Language Reference](#) gives a more formal definition of the language. [The Python Standard Library](#) documents the existing object types, functions and modules (both built-in and written in Python) that give the language its wide application range.

For a detailed description of the whole Python/C API, see the separate [Python/C API Reference Manual](#).

Recommended third party tools

This guide only covers the basic tools for creating extensions provided as part of this version of CPython. Third party tools like [Cython](#), [cffi](#), [SWIG](#) and [Numba](#) offer both simpler and more sophisticated approaches to creating C and C++ extensions for Python.

See also:

[Python Packaging User Guide: Binary Extensions](#)

The Python Packaging User Guide not only covers several available tools that simplify the creation of binary extensions, but also discusses the various reasons why creating an extension module may be desirable in the first place.

Creating extensions without third party tools

This section of the guide covers creating C and C++ extensions without assistance from third party tools. It is intended primarily for creators of those tools, rather than being a recommended way to create your own C extensions.

- [1. Extending Python with C or C++](#)

- 1.1. A Simple Example
- 1.2. Intermezzo: Errors and Exceptions
- 1.3. Back to the Example
- 1.4. The Module's Method Table and Initialization Function
- 1.5. Compilation and Linkage
- 1.6. Calling Python Functions from C
- 1.7. Extracting Parameters in Extension Functions
- 1.8. Keyword Parameters for Extension Functions
- 1.9. Building Arbitrary Values
- 1.10. Reference Counts
- 1.11. Writing Extensions in C++
- 1.12. Providing a C API for an Extension Module
- 2. Defining Extension Types: Tutorial
 - 2.1. The Basics
 - 2.2. Adding data and methods to the Basic example
 - 2.3. Providing finer control over data attributes
 - 2.4. Supporting cyclic garbage collection
 - 2.5. Subclassing other types
- 3. Defining Extension Types: Assorted Topics
 - 3.1. Finalization and De-allocation
 - 3.2. Object Presentation
 - 3.3. Attribute Management
 - 3.4. Object Comparison
 - 3.5. Abstract Protocol Support
 - 3.6. Weak Reference Support
 - 3.7. More Suggestions
- 4. Building C and C++ Extensions
 - 4.1. Building C and C++ Extensions with distutils
 - 4.2. Distributing your extension modules
- 5. Building C and C++ Extensions on Windows
 - 5.1. A Cookbook Approach
 - 5.2. Differences Between Unix and Windows
 - 5.3. Using DLLs in Practice

Embedding the CPython runtime in a larger application

Sometimes, rather than creating an extension that runs inside the Python interpreter as the main application, it is desirable to instead embed the CPython runtime inside a larger application. This section covers some of the details involved in doing that successfully.

- 1. Embedding Python in Another Application
 - 1.1. Very High Level Embedding

- 1.2. Beyond Very High Level Embedding: An overview
- 1.3. Pure Embedding
- 1.4. Extending Embedded Python
- 1.5. Embedding Python in C++
- 1.6. Compiling and Linking under Unix-like systems