Boost.Python Build and Test HOWTO

1 Requirements

Boost.Python requires Python 2.2[1] *or newer*.

2 Background

There are two basic models for combining C++ and Python:

- extending, in which the end-user launches the Python interpreter executable and imports Python “extension modules” written in C++. Think of taking a library written in C++ and giving it a Python interface so Python programmers can use it. From Python, these modules look just like regular Python modules.
- embedding, in which the end-user launches a program written in C++ that in turn invokes the Python interpreter as a library subroutine. Think of adding scriptability to an existing application.

The key distinction between extending and embedding is the location of the C++ `main()` function: in the Python interpreter executable, or in some other program, respectively. Note that even when embedding Python in another program, extension modules are often the best way to make C/C++ functionality accessible to Python code, so the use of extension modules is really at the heart of both models.

Except in rare cases, extension modules are built as dynamically-loaded libraries with a single entry point, which means you can change them without rebuilding either the other extension modules or the executable containing `main()`.

Contents

3 No-Install Quickstart

There is no need to “install Boost” in order to get started using Boost.Python. These instructions use Boost.Build projects, which will build those binaries as soon as they're needed. Your first tests may take a little longer while you wait for Boost.Python to build, but doing things this way will save you from worrying about build intricacies like which library binaries to use for a specific compiler configuration and figuring out the right compiler options to use yourself.

Note

Of course it's possible to use other build systems to build Boost.Python and its extensions, but they are not officially supported by Boost. Moreover **99% of all “I can't build Boost.Python” problems come from trying to use another build system** without first following these instructions.

If you want to use another system anyway, we suggest that you follow these instructions, and then invoke `bjam` with the

```
-a -ofilename
```

options to dump the build commands it executes to a file, so you can see what your alternate build system needs to do.

3.1 Basic Procedure

1. Get Boost; see sections 1 and 2 [Unix/Linux, Windows] of the Boost Getting Started Guide.
2. Get the `bjam` build driver. See section 5 [Unix/Linux, Windows] of the Boost Getting Started Guide.
3. cd into the `libs/python/example/quickstart/` directory of your Boost installation, which contains a small example project.
4. Invoke `bjam`. Replace the "`stage`" argument from the example invocation from section 5 of the Getting Started Guide with "`test`," to build all the test targets. Also add the argument "`--verbose-test`" to see the output generated by the tests when they are run.

   On Windows, your `bjam` invocation might look something like:

   ```
   C:\boost_1_34_0\…\quickstart> bjam toolset=msvc --verbose-test test
   ```

   and on Unix variants, perhaps,

   ```
   ~/boost_1_34_0/…/quickstart$ bjam toolset=gcc --verbose-test test
   ```

Note to Windows Users

For the sake of concision, the rest of this guide will use unix-style forward slashes in pathnames instead of the backslashes with which you may be more familiar. The forward slashes should work everywhere except in Command Prompt windows, where you should use backslashes.

If you followed this procedure successfully, you will have built an extension module called `extending` and tested it by running a Python script called `test_extending.py`. You will also have built and run a simple application called `embedding` that embeds python.

3.2 In Case of Trouble

If you're seeing lots of compiler and/or linker error messages, it's probably because Boost.Build is having trouble finding your Python installation. You might want to pass the `--debug-configuration` option to `bjam` the first few times you invoke it, to make sure that Boost.Build is correctly locating all the parts of your Python installation. If it isn't, consider Configuring Boost.Build as detailed below.

If you're still having trouble, Someone on one of the following mailing lists may be able to help:

- The Boost.Build mailing list for issues related to Boost.Build
- The Python C++ Sig for issues specifically related to Boost.Python

3.3 In Case Everything Seemed to Work

Rejoice! If you're new to Boost.Python, at this point it might be a good idea to ignore build issues for a while and concentrate on learning the library by going through the tutorial and perhaps some of the reference documentation, trying out what you've learned about the API by modifying the quickstart project.

3.4 Modifying the Example Project

If you're content to keep your extension module forever in one source file called `extending.cpp`, inside your Boost distribution, and import it forever as `extending`, then you can stop here. However, it's likely that you will want to make a few changes. There are a few things you can do without having to learn Boost.Build in depth.

The project you just built is specified in two files in the current directory: `boost-build.jam`, which tells `bjam` where it can find the interpreted code of the Boost build system, and `Jamroot`, which describes the targets you just built. These files are heavily commented, so they should be easy to modify. Take care, however, to preserve whitespace. Punctuation such as `;` will not be recognized as intended by `bjam` if it is not surrounded by whitespace.

Relocate the Project

You'll probably want to copy this project elsewhere so you can change it without modifying your Boost distribution. To do that, simply

a. copy the entire `libs/python/example/quickstart/` directory into a new directory.
b. In the new copies of `boost-build.jam` and `Jamroot`, locate the relative path near the top of the file that is clearly marked by a comment, and edit that path so that it refers to the same directory your Boost distribution as it referred to when the file was in its original location in the `libs/python/example/quickstart/` directory.

For example, if you moved the project from `/home/dave/boost_1_34_0/libs/python/example/quickstart` to `/home/dave/my-project`, you could change the first path in `boost-build.jam` from

```
../../../../tools/build/v2
```

to

```
/home/dave/boost_1_34_0/tools/build/v2
```

and change the first path in `Jamroot` from

```
../../../..
```

to

```
/home/dave/boost_1_34_0
```

Add New or Change Names of Existing Source Files

The names of additional source files involved in building your extension module or embedding application can be listed in `Jamroot` right alongside `extending.cpp` or `embedding.cpp` respectively. Just be sure to leave whitespace around each filename:

```
… file1.cpp file2.cpp file3.cpp …
```

Naturally, if you want to change the name of a source file you can tell Boost.Build about it by editing the name in `Jamroot`.

Change the Name of your Extension Module

The name of the extension module is determined by two things:

1. the name in `Jamroot` immediately following `python-extension`, and
2. the name passed to `BOOST_PYTHON_MODULE` in `extending.cpp`.

To change the name of the extension module from `extending` to `hello`, you'd edit `Jamroot`, changing

```
python-extension extending : extending.cpp ;
```

to

```
python-extension hello : extending.cpp ;
```

and you'd edit extending.cpp, changing

```
BOOST_PYTHON_MODULE(extending)
```

to

```
BOOST_PYTHON_MODULE(hello)
```

4 Installing Boost.Python on your System

Since Boost.Python is a separately-compiled (as opposed to header-only) library, its user relies on the services of a Boost.Python library binary.

If you need a regular installation of the Boost.Python library binaries on your system, the Boost Getting Started Guide will walk you through the steps of creating one. If building binaries from source, you might want to supply the `--with-python` argument to `bjam` (or the `--with-libraries=python` argument to `configure`), so only the Boost.Python binary will be built, rather than all the Boost binaries.

5 Configuring Boost.Build

As described in the Boost.Build reference manual, a file called `user-config.jam` in your home directory[6] is used to specify the tools and libraries available to the build system. You may need to create or edit `user-config.jam` to tell Boost.Build how to invoke Python, `#include` its headers, and link with its libraries.

> **Users of Unix-Variant OSes**
>
> If you are using a unix-variant OS and you ran Boost's `configure` script, it may have generated a `user-config.jam` for you.[4] If your `configure`/`make` sequence was successful and Boost.Python binaries were built, your `user-config.jam` file is probably already correct.

If you have one fairly “standard” python installation for your platform, you might not need to do anything special to describe it. If you haven't configured python in `user-config.jam` (and you don't specify `--without-python` on the Boost.Build command line), Boost.Build will automatically execute the equivalent of

```
import toolset : using ;
using python ;
```

which automatically looks for Python in the most likely places. However, that only happens when using the Boost.Python project file (e.g. when referred to by another project as in the quickstart method). If instead you are linking against separately-compiled Boost.Python binaries, you should set up a `user-config.jam` file with at least the minimal incantation above.

5.1 Python Configuration Parameters

If you have several versions of Python installed, or Python is installed in an unusual way, you may want to supply any or all of the following optional parameters to `using python`.

version
: the version of Python to use. Should be in Major.Minor format, for example, `2.3`. Do not include the subminor version (i.e. *not* `2.5.1`). If you have multiple Python versions installed, the version will usually be the only configuration argument required.

cmd-or-prefix
: preferably, a command that invokes a Python interpreter. Alternatively, the installation prefix for Python libraries and header files. Only use the alternative formulation if there is no appropriate Python executable available.

includes
: the `#include` paths for Python headers. Normally the correct path(s) will be automatically deduced from `version` and/or `cmd-or-prefix`.

libraries
: the path to Python library binaries. On MacOS/Darwin, you can also pass the path of the Python framework. Normally the correct path(s) will be automatically deduced from `version` and/or `cmd-or-prefix`.

condition
: if specified, should be a set of Boost.Build properties that are matched against the build configuration when Boost.Build selects a Python configuration to use. See examples below for details.

extension-suffix
: A string to append to the name of extension modules before the true filename extension. You almost certainly don't need to use this. Usually this suffix is only used when targeting a Windows debug build of Python, and will be set automatically for you based on the value of the `<python-debugging>` feature. However, at least one Linux distribution (Ubuntu Feisty Fawn) has a specially configured python-dbg package that claims to use such a suffix.

5.2 Examples

Note that in the examples below, case and *especially whitespace* are significant.

- If you have both python 2.5 and python 2.4 installed, `user-config.jam` might contain:

```
using python : 2.5 ;  # Make both versions of Python available

using python : 2.4 ;  # To build with python 2.4, add python=2.4
                      # to your command line.
```

  The first version configured (2.5) becomes the default. To build against python 2.4, add `python=2.4` to the `bjam` command line.

- If you have python installed in an unusual location, you might supply the path to the interpreter in the `cmd-or-prefix` parameter:

```
using python : : /usr/local/python-2.6-beta/bin/python ;
```

- If you have a separate build of Python for use with a particular toolset, you might supply that toolset in the `condition` parameter:

```
using python ;  # use for most toolsets

# Use with Intel C++ toolset
using python
     : # version
     : c:\\Devel\\Python-2.5-IntelBuild\\PCBuild\\python # cmd-or-prefix
     : # includes
     : # libraries
     : <toolset>intel # condition
     ;
```

- If you have downloaded the Python sources and built both the normal and the “python debugging” builds from source on Windows, you might see:

```
using python : 2.5 : C:\\src\\Python-2.5\\PCBuild\\python ;
using python : 2.5 : C:\\src\\Python-2.5\\PCBuild\\python_d
  : # includes
  : # libs
  : <python-debugging>on ;
```

- You can set up your user-config.jam so a bjam built under Windows can build/test both Windows and Cygwin python extensions. Just pass `<target-os>cygwin` in the `condition` parameter for the cygwin python installation:

  ```
  # windows installation
  using python ;

  # cygwin installation
  using python : : c:\\cygwin\\bin\\python2.5 : : : <target-os>cygwin ;
  ```

  when you put target-os=cygwin in your build request, it should build with the cygwin version of python:[5]

  ```
  bjam target-os=cygwin toolset=gcc
  ```

  This is supposed to work the other way, too (targeting windows python with a Cygwin bjam) but it seems as though the support in Boost.Build's toolsets for building that way is broken at the time of this writing.

- Note that because of the way Boost.Build currently selects target alternatives, you might have be very explicit in your build requests. For example, given:

  ```
  using python : 2.5 ; # a regular windows build
  using python : 2.4 : : : : <target-os>cygwin ;
  ```

  building with

  ```
  bjam target-os=cygwin
  ```

  will yield an error. Instead, you'll need to write:

  ```
  bjam target-os=cygwin/python=2.4
  ```

6 Choosing a Boost.Python Library Binary

If—instead of letting Boost.Build construct and link with the right libraries automatically—you choose to use a pre-built Boost.Python library, you'll need to think about which one to link with. The Boost.Python binary comes in both static and dynamic flavors. Take care to choose the right flavor for your application.[2]

6.1 The Dynamic Binary

The dynamic library is the safest and most-versatile choice:

- A single copy of the library code is used by all extension modules built with a given toolset.[3]
- The library contains a type conversion registry. Because one registry is shared among all extension modules, instances of a class exposed to Python in one dynamically-loaded extension module can be passed to functions exposed in another such module.

6.2 The Static Binary

It might be appropriate to use the static Boost.Python library in any of the following cases:

- You are extending python and the types exposed in your dynamically-loaded extension module don't need to be used by any other Boost.Python extension modules, and you don't care if the core library code is duplicated among them.
- You are embedding python in your application and either:
  - You are targeting a Unix variant OS other than MacOS or AIX, where the dynamically-loaded extension modules can “see” the Boost.Python library symbols that are part of the executable.
  - Or, you have statically linked some Boost.Python extension modules into your application and you don't care if any dynamically-loaded Boost.Python extension modules are able to use the types exposed by your statically-linked extension modules (and vice-versa).

7 `#include` Issues

1. If you should ever have occasion to `#include "python.h"` directly in a translation unit of a program using Boost.Python, use `#include "boost/python/detail/wrap_python.hpp"` instead. It handles several issues necessary for use with Boost.Python, one of which is mentioned in the next section.
2. Be sure not to `#include` any system headers before `wrap_python.hpp`. This restriction is actually imposed by Python, or more properly, by Python's interaction with your operating system. See http://docs.python.org/ext/simpleExample.html for details.

8 Python Debugging Builds

Python can be built in a special “python debugging” configuration that adds extra checks and instrumentation that can be very useful for developers of extension modules. The data structures used by the debugging configuration contain additional members, so **a Python executable built with python debugging enabled cannot be used with an extension module or library compiled without it, and vice-versa.**

Since pre-built “python debugging” versions of the Python executable and libraries are not supplied with most distributions of Python,[7] and we didn't want to force our users to build them, Boost.Build does not automatically enable python debugging in its `debug` build variant (which is the default). Instead there is a special build property called `python-debugging` that, when used as a build property, will define the right preprocessor symbols and select the right libraries to link with.

On unix-variant platforms, the debugging versions of Python's data structures will only be used if the symbol `Py_DEBUG` is defined. On many windows compilers, when extension modules are built with the preprocessor symbol `_DEBUG`, Python defaults to force linking with a special debugging version of the Python DLL. Since that symbol is very commonly used even when Python is not present, Boost.Python temporarily undefines _DEBUG when Python.h is #included from `boost/python/detail/wrap_python.hpp` - unless `BOOST_DEBUG_PYTHON` is defined. The upshot is that if you want “python debugging”and you aren't using Boost.Build, you should make sure `BOOST_DEBUG_PYTHON` is defined, or python debugging will be suppressed.

9 Testing Boost.Python

To run the full test suite for Boost.Python, invoke `bjam` in the `libs/python/test` subdirectory of your Boost distribution.

10 Notes for MinGW (and Cygwin with -mno-cygwin) GCC Users

If you are using a version of Python prior to 2.4.1 with a MinGW prior to 3.0.0 (with binutils-2.13.90-20030111-1), you will need to create a MinGW-compatible version of the Python library; the one shipped with Python will only work with a Microsoft-compatible linker. Follow the instructions in the “Non-Microsoft” section of the “Building Extensions: Tips And Tricks” chapter in Installing Python Modules to create `libpythonXX.a`, where `XX` corresponds to the major and minor version numbers of your Python installation.

---

[1] Note that although we tested earlier versions of Boost.Python with Python 2.2, and we don't *think* we've done anything to break compatibility, this release of Boost.Python may not have been tested with versions of Python earlier than 2.4, so we're not 100% sure that python 2.2 and 2.3 are supported.

[2] Information about how to identify the static and dynamic builds of Boost.Python:

- on Windows

- on Unix variants

[3] Because of the way most *nix platforms share symbols among dynamically-loaded objects, I'm not certain that extension modules built with different compiler toolsets will always use different copies of the Boost.Python library when loaded into the same Python instance. Not using different libraries could be a good thing if the compilers have compatible ABIs, because extension modules built with the two libraries would be interoperable. Otherwise, it could spell disaster, since an extension module and the Boost.Python library would have different ideas of such things as class layout. I would appreciate someone doing the experiment to find out what happens.

[4] `configure` overwrites the existing `user-config.jam` in your home directory (if any) after making a backup of the old version.

[5] Note that the `<target-os>cygwin` feature is different from the `<flavor>cygwin` subfeature of the `gcc` toolset, and you might need handle both explicitly if you also have a MinGW GCC installed.

[6] Windows users, your home directory can be found by typing:

```
ECHO %HOMEDRIVE%%HOMEPATH%
```

into a command prompt window.

[7] On Unix and similar platforms, a debugging python and associated libraries are built by adding `--with-pydebug` when configuring the Python build. On Windows, the debugging version of Python is generated by the "Win32 Debug" target of the Visual Studio project in the PCBuild subdirectory of a full Python source code distribution.

View document source. Generated on: 2007-07-02 13:46 UTC. Generated by Docutils from reStructuredText source.