Python on macOS README

Authors: Jack Jansen (2004-07)
Ronald Oussoren (2010-04)
Ned Deily (2012-06)

This document provides a quick overview of some macOS specific features in the Python distribution.

macOS specific arguments to configure

- `--enable-framework[=DIR]`

  If this argument is specified the build will create a Python.framework rather than a traditional Unix install. See the section Building and using a framework-based Python on macOS for more information on frameworks.

  If the optional directory argument is specified the framework is installed into that directory. This can be used to install a python framework into your home directory:

  ```
  $ ./configure --enable-framework=/Users/ronald/Library/Frameworks
  $ make && make install
  ```

  This will install the framework itself in `/Users/ronald/Library/Frameworks`, the applications in a subdirectory of `/Users/ronald/Applications` and the command-line tools in `/Users/ronald/bin`.

- `--with-framework-name=NAME`

  Specify the name for the python framework, defaults to `Python`. This option is only valid when `--enable-framework` is specified.

- `--enable-universalsdk[=PATH]`

  Create a universal binary build of Python. This can be used with both regular and framework builds.

  The optional argument specifies which macOS SDK should be used to perform the build. In most cases on current systems, you do not need to specify PATH or you can just use `/`; the default MacOSX SDK for the active Xcode or Command Line Tools developer directory will be used. See the macOS `xcrun` man page for more information. Current versions of macOS and Xcode no longer install system header files in their traditional locations, like `/usr/include` and `/System/Library/Frameworks`; instead they are found within a MacOSX SDK. The Apple-supplied build tools handle this transparently and current versions of Python now handle this as well. So it is no longer necessary, and since macOS 10.14, no longer possible to force the installation of system headers with `xcode-select`.

- `--with-universal-archs=VALUE`

  Specify the kind of universal binary that should be created. This option is only valid when `--enable-universalsdk` is specified. The default is `32-bit` if building with a SDK that supports PPC, otherwise defaults to `intel`. Note that `intel` means a universal build of both 32-bit and 64-bit binaries and that may not be what you want; for example, as of macOS 10.15 Catalina, 32-bit execution is no longer supported by the operating system. Thus it is best to either explicitly specify values for `--with-universal-archs`:

  ```
  --enable-universalsdk --with-universal-archs=intel-64
  ```

  or avoid using either.

Building and using a universal binary of Python on macOS

1. What is a universal binary

A universal binary build of Python contains object code for more than one CPU architecture. A universal macOS executable file or library combines the architecture-specific code into one file and can therefore run at native speed on all supported architectures. Universal files were introduced in macOS 10.4 to add support for Intel-based Macs to the existing PowerPC (PPC) machines. In macOS 10.5 support was extended to 64-bit Intel and 64-bit PPC architectures. It is possible to build Python with various combinations of architectures depending on the build tools and macOS version in use. Note that PPC support was removed in macOS 10.7 and 32-bit Intel support was removed in macOS 10.15. So currently as of macOS 10.15, macOS only supports one execution architecture, 64-bit Intel (`x86_64`).

2. How do I build a universal binary

You can enable universal binaries by specifying the "--enable-universalsdk" flag to configure:

```
$ ./configure --enable-universalsdk
$ make
$ make install
```

This flag can be used with a framework build of python, but also with a classic unix build. Universal builds were first supported with macOS 10.4 with Xcode 2.1 and the 10.4u SDK. Starting with Xcode 3 and macOS 10.5, more configurations are available.

In general, universal builds depend on specific features provided by the Apple-supplied compilers and other build tools included in Apple's Xcode development tools. You should install Xcode or the command line tools component appropriate for the macOS release you are running on. See the Python Developer's Guide (https://devguide.python.org/setup/) for more information.

2.1 Flavors of universal binaries

It is possible to build a number of flavors of the universal binary build, the default is a 32-bit only binary (i386 and ppc) in build environments that support ppc (10.4 with Xcode 2, 10.5 and 10.6 with Xcode 3) or an Intel-32/-64-bit binary (i386 and X86_64) in build environments that do not support ppc (Xcode 4 on 10.6 and later systems). The flavor can be specified using the configure option `--with-universal-archs=VALUE`. The following values are available:

- `universal2`: `arm64`, `x86_64`
- `intel`: `i386`, `x86_64`
- `intel-32`: `i386`
- `intel-64`: `x86_64`
- `32-bit`: `ppc`, `i386`
- `3-way`: `i386`, `x86_64`, `ppc`
- `64-bit`: `ppc64`, `x86_64`
- `all`: `ppc`, `ppc64`, `i386`, `x86_64`

To build a universal binary that includes a 64-bit architecture, you must build on a system running macOS 10.5 or later. The `all` and `64-bit` flavors can only be built with a 10.5 SDK because `ppc64` support was only included with macOS 10.5. Although legacy `ppc` support was included with Xcode 3 on macOS 10.6, it was removed in Xcode 4, versions of which were released on macOS 10.6 and which is the standard for macOS 10.7. To summarize, the following combinations of SDKs and universal-archs flavors are available:

- 10.4u SDK with Xcode 2 supports `32-bit` only
- 10.5 SDK with Xcode 3.1.x supports all flavors
- 10.6 SDK with Xcode 3.2.x supports `intel`, `intel-32`, `intel-64`, `3-way`, and `32-bit`
- 10.6 SDK with Xcode 4 supports `intel`, `intel-32`, and `intel-64`
- 10.7 through 10.14 SDKs support `intel`, `intel-32`, and `intel-64`
- 10.15 and later SDKs support `intel-64` only
- 11.0 and later SDKs support `universal2`

The makefile for a framework build will also install `python3.x-32` binaries when the universal architecture includes at least one 32-bit architecture (that is, for all flavors but `64-bit` and `intel-64`). It will also install `python3.x-intel64` binaries in the `universal2` case to allow easy execution with the Rosetta 2 Intel emulator on Apple Silicon Macs.

Running a specific architecture

You can run code using a specific architecture using the `arch` command:

```
$ arch -i386 python
```

Or to explicitly run in 32-bit mode, regardless of the machine hardware:

```
$ arch -i386 -ppc python
```

Using `arch` is not a perfect solution as the selected architecture will not automatically carry through to subprocesses launched by programs and tests under that Python. If you want to ensure that Python interpreters launched in subprocesses also run in 32-bit-mode if the main interpreter does, use a `python3.x-32` binary and use the value of `sys.executable` as the `subprocess Popen` executable value.

Likewise, use `python3.x-intel64` to force execution in `x86_64` mode with `universal2` binaries.

Building and using a framework-based Python on macOS

1. Why would I want a framework Python instead of a normal static Python?

The main reason is because you want to create GUI programs in Python. With the exception of X11/XDarwin-based GUI toolkits all GUI programs need to be run from a macOS application bundle (".app").

While it is technically possible to create a .app without using frameworks you will have to do the work yourself if you really want this.

A second reason for using frameworks is that they put Python-related items in only two places: "/Library/Framework/Python.framework" and "/Applications/Python <VERSION>" where `<VERSION>` can be e.g. "3.8", "2.7", etc. This simplifies matters for users installing Python from a binary distribution if they want to get rid of it again. Moreover, due to the way frameworks work, users without admin privileges can install a binary distribution in their home directory without recompilation.

2. How does a framework Python differ from a normal static Python?

In everyday use there is no difference, except that things are stored in a different place. If you look in /Library/Frameworks/Python.framework you will see lots of relative symlinks, see the Apple documentation for details. If you are used to a normal unix Python file layout go down to Versions/Current and you will see the familiar bin and lib directories.

3. Do I need extra packages?

Yes, probably. If you want Tkinter support you need to get the macOS AquaTk distribution, this is installed by default on macOS 10.4 or later. Be aware, though, that the Cocoa-based AquaTk's supplied starting with macOS 10.6 have proven to be unstable. If possible, you should consider installing a newer version before building on macOS 10.6 or later, such as the ActiveTcl 8.6. See https://www.python.org/download/mac/tcltk/. If you are building with an SDK, ensure that the newer Tcl and Tk frameworks are seen in the SDK's `Library/Frameworks` directory; you may need to manually create symlinks to their installed location, `/Library/Frameworks`. If you want wxPython you need to get that. If you want Cocoa you need to get PyObjC.

4. How do I build a framework Python?

This directory contains a Makefile that will create a couple of python-related applications (full-blown macOS .app applications, that is) in "/Applications/Python <VERSION>", and a hidden helper application Python.app inside the Python.framework, and unix tools including "python" into /usr/local/bin. In addition it has a target "installmacsubtree" that installs the relevant portions of the Mac subtree into the Python.framework.

It is normally invoked indirectly through the main Makefile, as the last step in the sequence

1. ./configure --enable-framework
2. make
3. make install

This sequence will put the framework in `/Library/Framework/Python.framework`, the applications in `/Applications/Python <VERSION>` and the unix tools in `/usr/local/bin`.

Installing in another place, for instance `$HOME/Library/Frameworks` if you have no admin privileges on your machine, is possible. This can be accomplished by configuring with `--enable-framework=$HOME/Library/Frameworks`. The other two directories will then also be installed in your home directory, at `$HOME/Applications/Python-<VERSION>` and `$HOME/bin`.

If you want to install some part, but not all, read the main Makefile. The frameworkinstall is composed of a couple of sub-targets that install the framework itself, the Mac subtree, the applications and the unix tools.

There is an extra target frameworkinstallextras that is not part of the normal frameworkinstall which installs the Tools directory into "/Applications/Python <VERSION>", this is useful for binary distributions.

What do all these programs do?

"IDLE.app" is an integrated development environment for Python: editor, debugger, etc.

"Python Launcher.app" is a helper application that will handle things when you double-click a .py, .pyc or .pyw file. For the first two it creates a Terminal window and runs the scripts with the normal command-line Python. For the latter it runs the script in the Python.app interpreter so the script can do GUI-things. Keep the `Option` key depressed while dragging or double-clicking a script to set runtime options. These options can be set persistently through Python Launcher's preferences dialog.

The program `pythonx.x` runs python scripts from the command line. Previously, various compatibility aliases were also installed, including `pythonwx.x` which in early releases of Python on macOS was required to run GUI programs. As of 3.4.0, the `pythonwx.x` aliases are no longer installed.

How do I create a binary distribution?

Download and unpack the source release from https://www.python.org/download/. Go to the directory `Mac/BuildScript`. There you will find a script `build-installer.py` that does all the work. This will download and build a number of 3rd-party libraries, configures and builds a framework Python, installs it, creates the installer package files and then packs this in a DMG image. The script also builds an HTML copy of the current Python documentation set for this release for inclusion in the framework. The installer package will create links to the documentation for use by IDLE, pydoc, shell users, and Finder user.

The script will build a universal binary so you'll therefore have to run this script on macOS 10.4 or later and with Xcode 2.1 or later installed. However, the Python build process itself has several build dependencies not available out of the box with macOS 10.4 so you may have to install additional software beyond what is provided with Xcode 2. It should be possible to use SDKs and/or older versions of Xcode to build installers that are compatible with older systems on a newer system but this may not be completely foolproof so the resulting executables, shared libraries, and `.so` bundles should be carefully examined and tested on all supported systems for proper dynamic linking dependencies. It is safest to build the distribution on a system running the minimum macOS version supported.

All of this is normally done completely isolated in /tmp/_py, so it does not use your normal build directory nor does it install into /.

Because of the way the script locates the files it needs you have to run it from within the BuildScript directory. The script accepts a number of command-line arguments, run it with --help for more information.

Configure warnings

The configure script sometimes emits warnings like the one below:

```
configure: WARNING: libintl.h: present but cannot be compiled
configure: WARNING: libintl.h:     check for missing prerequisite headers?
configure: WARNING: libintl.h: see the Autoconf documentation
configure: WARNING: libintl.h:     section "Present But Cannot Be Compiled"
configure: WARNING: libintl.h: proceeding with the preprocessor's result
configure: WARNING: libintl.h: in the future, the compiler will take precedence
configure: WARNING:     ## ------------------------------------- ##
configure: WARNING:     ## Report this to https://bugs.python.org/ ##
configure: WARNING:     ## ------------------------------------- ##
```

This almost always means you are trying to build a universal binary for Python and have libraries in `/usr/local` that don't contain the required architectures. Temporarily move `/usr/local` aside to finish the build.

Uninstalling a framework install, including the binary installer

Uninstalling a framework can be done by manually removing all bits that got installed. That's true for both installations from source and installations using the binary installer. macOS does not provide a central uninstaller.

The main bit of a framework install is the framework itself, installed in `/Library/Frameworks/Python.framework`. This can contain multiple versions of Python, if you want to remove just one version you have to remove the version-specific subdirectory: `/Library/Frameworks/Python.framework/Versions/X.Y`. If you do that, ensure that `/Library/Frameworks/Python.framework/Versions/Current` is a symlink that points to an installed version of Python.

A framework install also installs some applications in `/Applications/Python X.Y`,

And lastly a framework installation installs files in `/usr/local/bin`, all of them symbolic links to files in `/Library/Frameworks/Python.framework/Versions/X.Y/bin`.

Weak linking support

The CPython sources support building with the latest SDK while targeting deployment to macOS 10.9. This is done through weak linking of symbols introduced in macOS 10.10 or later and checking for their availability at runtime.

This requires the use of Apple's compiler toolchain on macOS 10.13 or later.

The basic implementation pattern is:

- `HAVE_<FUNCTION>` is a macro defined (or not) by the configure script
- `HAVE_<FUNCTION>_RUNTIME` is a macro defined in the relevant source files. This expands to a call to `__builtin_available` when using a new enough Apple compiler, and to a true value otherwise.
- Use `HAVE_<FUNCTION>_RUNTIME` before calling `<function>`. This macro *must* be used a the sole expression in an if statement:

  ```
  if (HAVE_<FUNCTION>_RUNTIME) {
    /* <function> is available */
  }
  ```

  Or:

  ```
  if (HAVE_<FUNCTION>_RUNTIME) {} else {
          /* <function> is not available */
  ```

  System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Mac\(cpython-main)(Mac)README.rst, line 392); *backlink*

  Inline emphasis start-string without end-string.

  System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Mac\(cpython-main)(Mac)README.rst, line 393)

  Definition list ends without a blank line; unexpected unindent.

  ```
  }
  ```

Using other patterns (such as `!HAVE_<FUNCTION>_RUNTIME`) is not supported by Apple's compilers.

Resources

- https://www.python.org/downloads/macos/
- https://www.python.org/community/sigs/current/pythonmac-sig/
- https://devguide.python.org/