

3. Using Python on Windows

This document aims to give an overview of Windows-specific behaviour you should know about when using Python on Microsoft Windows.

3.1. Installing Python

Unlike most Unix systems and services, Windows does not include a system supported installation of Python. To make Python available, the CPython team has compiled Windows installers (MSI packages) with every [release](#) for many years. These installers are primarily intended to add a per-user installation of Python, with the core interpreter and library being used by a single user. The installer is also able to install for all users of a single machine, and a separate ZIP file is available for application-local distributions.

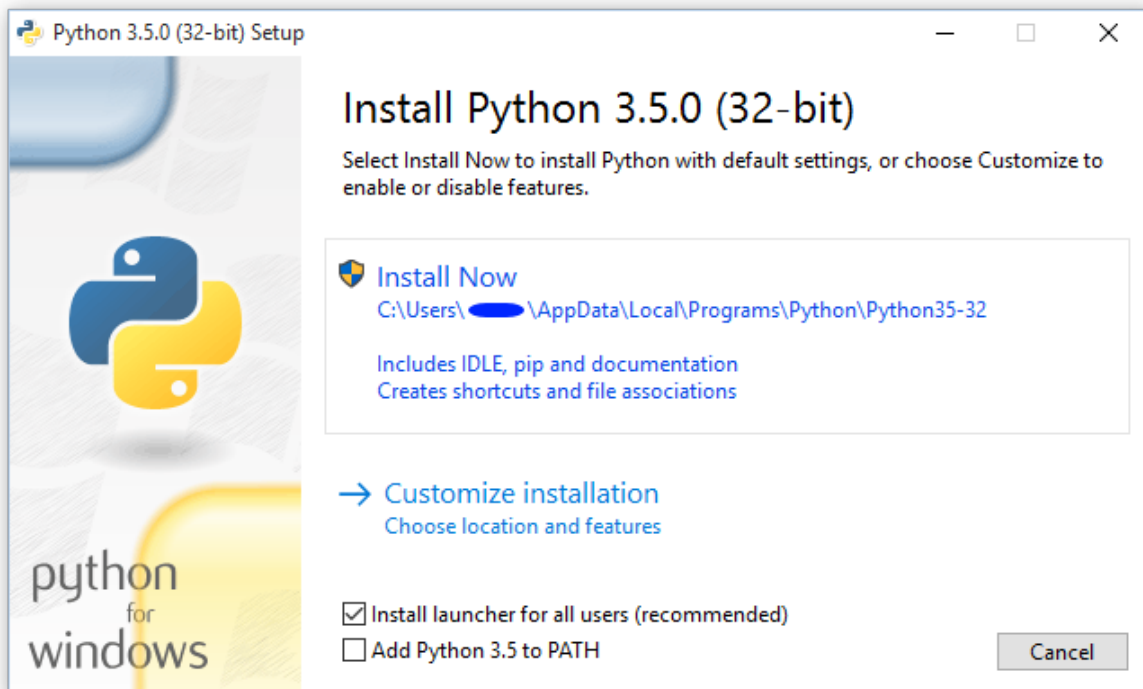
3.1.1. Supported Versions

As specified in [PEP 11](#), a Python release only supports a Windows platform while Microsoft considers the platform under extended support. This means that Python 3.6 supports Windows Vista and newer. If you require Windows XP support then please install Python 3.4.

3.1.2. Installation Steps

Four Python 3.6 installers are available for download - two each for the 32-bit and 64-bit versions of the interpreter. The *web installer* is a small initial download, and it will automatically download the required components as necessary. The *offline installer* includes the components necessary for a default installation and only requires an internet connection for optional features. See [Installing Without Downloading](#) for other ways to avoid downloading during installation.

After starting the installer, one of two options may be selected:



If you select “Install Now”:

- You will *not* need to be an administrator (unless a system update for the C Runtime Library is required or you install the [Python Launcher for Windows](#) for all users)
- Python will be installed into your user directory
- The [Python Launcher for Windows](#) will be installed according to the option at the bottom of the first page
- The standard library, test suite, launcher and pip will be installed
- If selected, the install directory will be added to your PATH
- Shortcuts will only be visible for the current user

Selecting “Customize installation” will allow you to select the features to install, the installation location and other options or post-install actions. To install debugging symbols or binaries, you will need to use this option.

To perform an all-users installation, you should select “Customize installation”. In this case:

- You may be required to provide administrative credentials or approval
- Python will be installed into the Program Files directory
- The [Python Launcher for Windows](#) will be installed into the Windows directory
- Optional features may be selected during installation
- The standard library can be pre-compiled to bytecode
- If selected, the install directory will be added to the system PATH
- Shortcuts are available for all users

3.1.3. Removing the MAX_PATH Limitation

Windows historically has limited path lengths to 260 characters. This meant that paths longer than this would not resolve and errors would result.

In the latest versions of Windows, this limitation can be expanded to approximately 32,000 characters. Your administrator will need to activate the “Enable Win32 long paths” group policy, or set the registry value `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\FileSystem@LongPathsEnabled` to 1.

This allows the `open()` function, the `os` module and most other path functionality to accept and return paths longer than 260 characters when using strings. (Use of bytes as paths is deprecated on Windows, and this feature is not available when using bytes.)

After changing the above option, no further configuration is required.

Changed in version 3.6: Support for long paths was enabled in Python.

3.1.4. Installing Without UI

All of the options available in the installer UI can also be specified from the command line, allowing scripted installers to replicate an installation on many machines without user interaction. These options may also be set without suppressing the UI in order to change some of the defaults.

To completely hide the installer UI and install Python silently, pass the `/quiet` option. To skip past the user interaction but still display progress and errors, pass the `/passive` option. The `/uninstall` option may be passed to immediately begin removing Python - no prompt will be displayed.

All other options are passed as `name=value`, where the value is usually 0 to disable a feature, 1 to enable a feature, or a path. The full list of available options is shown below.

Name	Description	Default
InstallAllUsers	Perform a system-wide installation.	0
TargetDir	The installation directory	Selected based on InstallAllUsers
DefaultAllUsersTargetDir	The default installation directory for all-user installs	%ProgramFiles%\Python X.Y or %ProgramFiles(x86)%\Python X.Y
DefaultJustForMeTargetDir	The default install directory for just-for-me installs	%LocalAppData%\Programs\PythonXY or %LocalAppData%\Programs\PythonXY-32
DefaultCustomTargetDir	The default custom install directory displayed in the UI	(empty)
AssociateFiles	Create file associations if the launcher is also installed.	1
CompileAll	Compile all .py files to .pyc.	0
PrependPath		0

Name	Description	Default
	Add install and Scripts directories to the PATH and .PY to PATHEXT	
Shortcuts	Create shortcuts for the interpreter, documentation and IDLE if installed.	1
Include_doc	Install Python manual	1
Include_debug	Install debug binaries	0
Include_dev	Install developer headers and libraries	1
Include_exe	Install python.exe and related files	1
Include_launcher	Install Python Launcher for Windows .	1
InstallLauncherAllUsers	Installs Python Launcher for Windows for all users.	1
Include_lib	Install standard library and extension modules	1
Include_pip	Install bundled pip and setuptools	1
Include_symbols	Install debugging symbols (*.pdb)	0
Include_tcltk	Install Tcl/Tk support and IDLE	1
Include_test	Install standard library test suite	1
Include_tools	Install utility scripts	1
LauncherOnly	Only installs the launcher. This will override most other options.	0
SimpleInstall	Disable most install UI	0
SimpleInstallDescription	A custom message to display when the simplified install UI is used.	(empty)

For example, to silently install a default, system-wide Python installation, you could use the following command (from an elevated command prompt):

```
python-3.6.0.exe /quiet InstallAllUsers=1 PrependPath=1 Include_test=0
```

To allow users to easily install a personal copy of Python without the test suite, you could provide a shortcut with the following command. This will display a simplified initial page and disallow customization:

```
python-3.6.0.exe InstallAllUsers=0 Include_launcher=0 Include_test=0
SimpleInstall=1 SimpleInstallDescription="Just for me, no test suite."
```

(Note that omitting the launcher also omits file associations, and is only recommended for per-user installs when there is also a system-wide installation that included the launcher.)

The options listed above can also be provided in a file named `unattend.xml` alongside the executable. This file specifies a list of options and values. When a value is provided as an at-

tribute, it will be converted to a number if possible. Values provided as element text are always left as strings. This example file sets the same options and the previous example:

```
<Options>
  <Option Name="InstallAllUsers" Value="no" />
  <Option Name="Include_launcher" Value="0" />
  <Option Name="Include_test" Value="no" />
  <Option Name="SimpleInstall" Value="yes" />
  <Option Name="SimpleInstallDescription">Just for me, no test suite</Option>
</Options>
```

3.1.5. Installing Without Downloading

As some features of Python are not included in the initial installer download, selecting those features may require an internet connection. To avoid this need, all possible components may be downloaded on-demand to create a complete *layout* that will no longer require an internet connection regardless of the selected features. Note that this download may be bigger than required, but where a large number of installations are going to be performed it is very useful to have a locally cached copy.

Execute the following command from Command Prompt to download all possible required files. Remember to substitute `python-3.6.0.exe` for the actual name of your installer, and to create layouts in their own directories to avoid collisions between files with the same name.

```
python-3.6.0.exe /layout [optional target directory]
```

You may also specify the `/quiet` option to hide the progress display.

3.1.6. Modifying an install

Once Python has been installed, you can add or remove features through the Programs and Features tool that is part of Windows. Select the Python entry and choose “Uninstall/Change” to open the installer in maintenance mode.

“Modify” allows you to add or remove features by modifying the checkboxes - unchanged checkboxes will not install or remove anything. Some options cannot be changed in this mode, such as the install directory; to modify these, you will need to remove and then reinstall Python completely.

“Repair” will verify all the files that should be installed using the current settings and replace any that have been removed or modified.

“Uninstall” will remove Python entirely, with the exception of the [Python Launcher for Windows](#), which has its own entry in Programs and Features.

3.1.7. Other Platforms

With ongoing development of Python, some platforms that used to be supported earlier are no longer supported (due to the lack of users or developers). Check [PEP 11](#) for details on all unsupported platforms.

- [Windows CE](#) is still supported.
- The [Cygwin](#) installer offers to install the Python interpreter as well (cf. [Cygwin package source](#), [Maintainer releases](#))

See [Python for Windows](#) for detailed information about platforms with pre-compiled installers.

See also:

Python on XP

“7 Minutes to “Hello World!”” by Richard Dooling, 2006

Installing on Windows

in “[Dive into Python: Python from novice to pro](#)” by Mark Pilgrim, 2004, ISBN 1-59059-356-1

For Windows users

in “Installing Python” in “[A Byte of Python](#)” by Swaroop C H, 2003

3.2. Alternative bundles

Besides the standard CPython distribution, there are modified packages including additional functionality. The following is a list of popular versions and their key features:

ActivePython

Installer with multi-platform compatibility, documentation, PyWin32

Anaconda

Popular scientific modules (such as numpy, scipy and pandas) and the conda package manager.

Canopy

A “comprehensive Python analysis environment” with editors and other development tools.

WinPython

Windows-specific distribution with prebuilt scientific packages and tools for building packages.

Note that these packages may not include the latest versions of Python or other libraries, and are not maintained or supported by the core Python team.

3.3. Configuring Python

To run Python conveniently from a command prompt, you might consider changing some default environment variables in Windows. While the installer provides an option to configure

the PATH and PATHEXT variables for you, this is only reliable for a single, system-wide installation. If you regularly use multiple versions of Python, consider using the [Python Launcher for Windows](#).

3.3.1. Excursus: Setting environment variables

Windows allows environment variables to be configured permanently at both the User level and the System level, or temporarily in a command prompt.

To temporarily set environment variables, open Command Prompt and use the **set** command:

```
C:\>set PATH=C:\Program Files\Python 3.6;%PATH%
C:\>set PYTHONPATH=%PYTHONPATH%;C:\My_python_lib
C:\>python
```

These changes will apply to any further commands executed in that console, and will be inherited by any applications started from the console.

Including the variable name within percent signs will expand to the existing value, allowing you to add your new value at either the start or the end. Modifying PATH by adding the directory containing **python.exe** to the start is a common way to ensure the correct version of Python is launched.

To permanently modify the default environment variables, click Start and search for 'edit environment variables', or open System properties, Advanced system settings and click the Environment Variables button. In this dialog, you can add or modify User and System variables. To change System variables, you need non-restricted access to your machine (i.e. Administrator rights).

Note: Windows will concatenate User variables *after* System variables, which may cause unexpected results when modifying PATH.

The [PYTHONPATH](#) variable is used by all versions of Python 2 and Python 3, so you should not permanently configure this variable unless it only includes code that is compatible with all of your installed Python versions.

See also:

<https://support.microsoft.com/kb/100843>

Environment variables in Windows NT

<https://technet.microsoft.com/en-us/library/cc754250.aspx>

The SET command, for temporarily modifying environment variables

<https://technet.microsoft.com/en-us/library/cc755104.aspx>

The SETX command, for permanently modifying environment variables

<https://support.microsoft.com/kb/310519>

How To Manage Environment Variables in Windows XP

3.3.2. Finding the Python executable

Changed in version 3.5.

Besides using the automatically created start menu entry for the Python interpreter, you might want to start Python in the command prompt. The installer has an option to set that up for you.

On the first page of the installer, an option labelled “Add Python to PATH” may be selected to have the installer add the install location into the PATH. The location of the `Scripts\` folder is also added. This allows you to type **python** to run the interpreter, and **pip** for the package installer. Thus, you can also execute your scripts with command line options, see [Command line](#) documentation.

If you don't enable this option at install time, you can always re-run the installer, select Modify, and enable it. Alternatively, you can manually modify the PATH using the directions in [Excursus: Setting environment variables](#). You need to set your PATH environment variable to include the directory of your Python installation, delimited by a semicolon from other entries. An example variable could look like this (assuming the first two entries already existed):

```
C:\WINDOWS\system32;C:\WINDOWS;C:\Program Files\Python 3.6
```

3.4. Python Launcher for Windows

New in version 3.3.

The Python launcher for Windows is a utility which aids in locating and executing of different Python versions. It allows scripts (or the command-line) to indicate a preference for a specific Python version, and will locate and execute that version.

Unlike the PATH variable, the launcher will correctly select the most appropriate version of Python. It will prefer per-user installations over system-wide ones, and orders by language version rather than using the most recently installed version.

3.4.1. Getting started

3.4.1.1. From the command-line

Changed in version 3.6.

System-wide installations of Python 3.3 and later will put the launcher on your PATH. The launcher is compatible with all available versions of Python, so it does not matter which version is installed. To check that the launcher is available, execute the following command in Command Prompt:


```
py
```

You should find that the latest version of Python you have installed is started - it can be exited as normal, and any additional command-line arguments specified will be sent directly to Python.

If you have multiple versions of Python installed (e.g., 2.7 and 3.6) you will have noticed that Python 3.6 was started - to launch Python 2.7, try the command:

```
py -2.7
```

If you want the latest version of Python 2.x you have installed, try the command:

```
py -2
```

You should find the latest version of Python 2.x starts.

If you see the following error, you do not have the launcher installed:

```
'py' is not recognized as an internal or external command,  
operable program or batch file.
```

Per-user installations of Python do not add the launcher to PATH unless the option was selected on installation.

3.4.1.2. Virtual environments

New in version 3.5.

If the launcher is run with no explicit Python version specification, and a virtual environment (created with the standard library `venv` module or the external `virtualenv` tool) active, the launcher will run the virtual environment's interpreter rather than the global one. To run the global interpreter, either deactivate the virtual environment, or explicitly specify the global Python version.

3.4.1.3. From a script

Let's create a test Python script - create a file called `hello.py` with the following contents

```
#!/python  
import sys  
sys.stdout.write("hello from Python %s\n" % (sys.version,))
```

From the directory in which `hello.py` lives, execute the command:

```
py hello.py
```

You should notice the version number of your latest Python 2.x installation is printed. Now try changing the first line to be:

```
#! python3
```

Re-executing the command should now print the latest Python 3.x information. As with the above command-line examples, you can specify a more explicit version qualifier. Assuming you have Python 2.6 installed, try changing the first line to `#! python2.6` and you should find the 2.6 version information printed.

Note that unlike interactive use, a bare “python” will use the latest version of Python 2.x that you have installed. This is for backward compatibility and for compatibility with Unix, where the command `python` typically refers to Python 2.

3.4.1.4. From file associations

The launcher should have been associated with Python files (i.e. `.py`, `.pyw`, `.pyc` files) when it was installed. This means that when you double-click on one of these files from Windows explorer the launcher will be used, and therefore you can use the same facilities described above to have the script specify the version which should be used.

The key benefit of this is that a single launcher can support multiple Python versions at the same time depending on the contents of the first line.

3.4.2. Shebang Lines

If the first line of a script file starts with `#!`, it is known as a “shebang” line. Linux and other Unix like operating systems have native support for such lines and they are commonly used on such systems to indicate how a script should be executed. This launcher allows the same facilities to be used with Python scripts on Windows and the examples above demonstrate their use.

To allow shebang lines in Python scripts to be portable between Unix and Windows, this launcher supports a number of ‘virtual’ commands to specify which interpreter to use. The supported virtual commands are:

- `/usr/bin/env python`
- `/usr/bin/python`
- `/usr/local/bin/python`
- `python`

For example, if the first line of your script starts with

```
#! /usr/bin/python
```

The default Python will be located and used. As many Python scripts written to work on Unix will already have this line, you should find these scripts can be used by the launcher without modification. If you are writing a new script on Windows which you hope will be useful on Unix, you should use one of the shebang lines starting with `/usr`.

Any of the above virtual commands can be suffixed with an explicit version (either just the major version, or the major and minor version) - for example `/usr/bin/python2.7` - which will cause that specific version to be located and used.

The `/usr/bin/env` form of shebang line has one further special property. Before looking for installed Python interpreters, this form will search the executable `PATH` for a Python executable. This corresponds to the behaviour of the Unix `env` program, which performs a `PATH` search.

3.4.3. Arguments in shebang lines

The shebang lines can also specify additional options to be passed to the Python interpreter. For example, if you have a shebang line:

```
#!/usr/bin/python -v
```

Then Python will be started with the `-v` option

3.4.4. Customization

3.4.4.1. Customization via INI files

Two `.ini` files will be searched by the launcher - `py.ini` in the current user's "application data" directory (i.e. the directory returned by calling the Windows function `SHGetFolderPath` with `CSIDL_LOCAL_APPDATA`) and `py.ini` in the same directory as the launcher. The same `.ini` files are used for both the 'console' version of the launcher (i.e. `py.exe`) and for the 'windows' version (i.e. `pyw.exe`)

Customization specified in the "application directory" will have precedence over the one next to the executable, so a user, who may not have write access to the `.ini` file next to the launcher, can override commands in that global `.ini` file)

3.4.4.2. Customizing default Python versions

In some cases, a version qualifier can be included in a command to dictate which version of Python will be used by the command. A version qualifier starts with a major version number and can optionally be followed by a period ('.') and a minor version specifier. If the minor qualifier is specified, it may optionally be followed by `-32` to indicate the 32-bit implementation of that version be used.

For example, a shebang line of `#!/python` has no version qualifier, while `#!/python3` has a version qualifier which specifies only a major version.

If no version qualifiers are found in a command, the environment variable `PY_PYTHON` can be set to specify the default version qualifier - the default value is `"2"`. Note this value could specify just a major version (e.g. `"2"`) or a major.minor qualifier (e.g. `"2.6"`), or even major.minor-32.

If no minor version qualifiers are found, the environment variable `PY_PYTHON{major}` (where `{major}` is the current major version qualifier as determined above) can be set to specify the full version. If no such option is found, the launcher will enumerate the installed Python versions and use the latest minor release found for the major version, which is likely, although not guaranteed, to be the most recently installed version in that family.

On 64-bit Windows with both 32-bit and 64-bit implementations of the same (major.minor) Python version installed, the 64-bit version will always be preferred. This will be true for both 32-bit and 64-bit implementations of the launcher - a 32-bit launcher will prefer to execute a 64-bit Python installation of the specified version if available. This is so the behavior of the launcher can be predicted knowing only what versions are installed on the PC and without regard to the order in which they were installed (i.e., without knowing whether a 32 or 64-bit version of Python and corresponding launcher was installed last). As noted above, an optional “-32” suffix can be used on a version specifier to change this behaviour.

Examples:

- If no relevant options are set, the commands `python` and `python2` will use the latest Python 2.x version installed and the command `python3` will use the latest Python 3.x installed.
- The commands `python3.1` and `python2.7` will not consult any options at all as the versions are fully specified.
- If `PY_PYTHON=3`, the commands `python` and `python3` will both use the latest installed Python 3 version.
- If `PY_PYTHON=3.1-32`, the command `python` will use the 32-bit implementation of 3.1 whereas the command `python3` will use the latest installed Python (`PY_PYTHON` was not considered at all as a major version was specified.)
- If `PY_PYTHON=3` and `PY_PYTHON3=3.1`, the commands `python` and `python3` will both use specifically 3.1

In addition to environment variables, the same settings can be configured in the .INI file used by the launcher. The section in the INI file is called `[defaults]` and the key name will be the same as the environment variables without the leading `PY_` prefix (and note that the key names in the INI file are case insensitive.) The contents of an environment variable will override things specified in the INI file.

For example:

- Setting `PY_PYTHON=3.1` is equivalent to the INI file containing:

```
[defaults]
python=3.1
```

- Setting `PY_PYTHON=3` and `PY_PYTHON3=3.1` is equivalent to the INI file containing:

```
[defaults]
python=3
python3=3.1
```

3.4.5. Diagnostics

If an environment variable `PYLAUNCH_DEBUG` is set (to any value), the launcher will print diagnostic information to `stderr` (i.e. to the console). While this information manages to be simultaneously verbose *and* terse, it should allow you to see what versions of Python were located, why a particular version was chosen and the exact command-line used to execute the target Python.

3.5. Finding modules

Python usually stores its library (and thereby your site-packages folder) in the installation directory. So, if you had installed Python to `C:\Python\`, the default library would reside in `C:\Python\Lib\` and third-party modules should be stored in `C:\Python\Lib\site-packages\`.

To completely override `sys.path`, create a `._pth` file with the same name as the DLL (`python36._pth`) or the executable (`python._pth`) and specify one line for each path to add to `sys.path`. The file based on the DLL name overrides the one based on the executable, which allows paths to be restricted for any program loading the runtime if desired.

When the file exists, all registry and environment variables are ignored, isolated mode is enabled, and `site` is not imported unless one line in the file specifies `import site`. Blank paths and lines starting with `#` are ignored. Each path may be absolute or relative to the location of the file. Import statements other than to `site` are not permitted, and arbitrary code cannot be specified.

Note that `.pth` files (without leading underscore) will be processed normally by the `site` module.

When no `._pth` file is found, this is how `sys.path` is populated on Windows:

- An empty entry is added at the start, which corresponds to the current directory.
- If the environment variable `PYTHONPATH` exists, as described in [Environment variables](#), its entries are added next. Note that on Windows, paths in this variable must be separated by semicolons, to distinguish them from the colon used in drive identifiers (`C:\` etc.).
- Additional “application paths” can be added in the registry as subkeys of `\SOFTWARE\Python\PythonCore\version\PythonPath` under both the `HKEY_CURRENT_USER` and `HKEY_LOCAL_MACHINE` hives. Subkeys which have semicolon-delimited path strings as their default value will cause each path to be added to `sys.path`. (Note that all known installers only use `HKLM`, so `HKCU` is typically empty.)
- If the environment variable `PYTHONHOME` is set, it is assumed as “Python Home”. Otherwise, the path of the main Python executable is used to locate a “landmark file” (either `Lib\os.py` or `pythonXY.zip`) to deduce the “Python Home”. If a Python home is found, the relevant sub-directories added to `sys.path` (`Lib`, `plat-win`, etc) are based on that folder. Otherwise, the core Python path is constructed from the `PythonPath` stored in the registry.

- If the Python Home cannot be located, no `PYTHONPATH` is specified in the environment, and no registry entries can be found, a default path with relative entries is used (e.g. `.\Lib;.\plat-win`, etc).

If a `pyvenv.cfg` file is found alongside the main executable or in the directory one level above the executable, the following variations apply:

- If home is an absolute path and `PYTHONHOME` is not set, this path is used instead of the path to the main executable when deducing the home location.

The end result of all this is:

- When running `python.exe`, or any other `.exe` in the main Python directory (either an installed version, or directly from the PCbuild directory), the core path is deduced, and the core paths in the registry are ignored. Other “application paths” in the registry are always read.
- When Python is hosted in another `.exe` (different directory, embedded via COM, etc), the “Python Home” will not be deduced, so the core path from the registry is used. Other “application paths” in the registry are always read.
- If Python can’t find its home and there are no registry value (frozen `.exe`, some very strange installation setup) you get a path with some default, but relative, paths.

For those who want to bundle Python into their application or distribution, the following advice will prevent conflicts with other installations:

- Include a `._pth` file alongside your executable containing the directories to include. This will ignore paths listed in the registry and environment variables, and also ignore `site` unless `import site` is listed.
- If you are loading `python3.dll` or `python36.dll` in your own executable, explicitly call `Py_SetPath()` or (at least) `Py_SetProgramName()` before `Py_Initialize()`.
- Clear and/or overwrite `PYTHONPATH` and set `PYTHONHOME` before launching `python.exe` from your application.
- If you cannot use the previous suggestions (for example, you are a distribution that allows people to run `python.exe` directly), ensure that the landmark file (`Lib\os.py`) exists in your install directory. (Note that it will not be detected inside a ZIP file, but a correctly named ZIP file will be detected instead.)

These will ensure that the files in a system-wide installation will not take precedence over the copy of the standard library bundled with your application. Otherwise, your users may experience problems using your application. Note that the first suggestion is the best, as the other may still be susceptible to non-standard paths in the registry and user site-packages.

Changed in version 3.6:

- Adds `._pth` file support and removes `applocal` option from `pyvenv.cfg`.
- Adds `pythonXX.zip` as a potential landmark when directly adjacent to the executable.

Deprecated since version 3.6:

Modules specified in the registry under Modules (not PythonPath) may be imported by `importlib.machinery.WindowsRegistryFinder`. This finder is enabled on

Windows in 3.6.0 and earlier, but may need to be explicitly added to `sys.meta_path` in the future.

3.6. Additional modules

Even though Python aims to be portable among all platforms, there are features that are unique to Windows. A couple of modules, both in the standard library and external, and snippets exist to use these features.

The Windows-specific standard modules are documented in [MS Windows Specific Services](#).

3.6.1. PyWin32

The [PyWin32](#) module by Mark Hammond is a collection of modules for advanced Windows-specific support. This includes utilities for:

- [Component Object Model](#) (COM)
- Win32 API calls
- Registry
- Event log
- [Microsoft Foundation Classes](#) (MFC) user interfaces

[PythonWin](#) is a sample MFC application shipped with PyWin32. It is an embeddable IDE with a built-in debugger.

See also:

[Win32 How Do I...?](#)

by Tim Golden

[Python and COM](#)

by David and Paul Boddie

3.6.2. cx_Freeze

[cx_Freeze](#) is a [distutils](#) extension (see [Extending Distutils](#)) which wraps Python scripts into executable Windows programs (*.exe files). When you have done this, you can distribute your application without requiring your users to install Python.

3.6.3. WConio

Since Python's advanced terminal handling layer, [curses](#), is restricted to Unix-like systems, there is a library exclusive to Windows as well: Windows Console I/O for Python.

[WConio](#) is a wrapper for Turbo-C's CONIO.H, used to create text user interfaces.

3.7. Compiling Python on Windows

If you want to compile CPython yourself, first thing you should do is get the [source](#). You can download either the latest release's source or just grab a fresh [checkout](#).

The source tree contains a build solution and project files for Microsoft Visual Studio 2015, which is the compiler used to build the official Python releases. These files are in the PCbuild directory.

Check PCbuild/readme.txt for general information on the build process.

For extension modules, consult [Building C and C++ Extensions on Windows](#).

See also:

[Python + Windows + distutils + SWIG + gcc MinGW](#)

or “Creating Python extensions in C/C++ with SWIG and compiling them with MinGW gcc under Windows” or “Installing Python extension with distutils and without Microsoft Visual C++” by Sébastien Sauvage, 2003

[MingW – Python extensions](#)

by Trent Apted et al, 2007

3.8. Embedded Distribution

New in version 3.5.

The embedded distribution is a ZIP file containing a minimal Python environment. It is intended for acting as part of another application, rather than being directly accessed by end-users.

When extracted, the embedded distribution is (almost) fully isolated from the user's system, including environment variables, system registry settings, and installed packages. The standard library is included as pre-compiled and optimized .pyc files in a ZIP, and python3.dll, python36.dll, python.exe and pythonw.exe are all provided. Tcl/tk (including all dependants, such as Idle), pip and the Python documentation are not included.

Note: The embedded distribution does not include the [Microsoft C Runtime](#) and it is the responsibility of the application installer to provide this. The runtime may have already been installed on a user's system previously or automatically via Windows Update, and can be detected by finding ucrtbase.dll in the system directory.

Third-party packages should be installed by the application installer alongside the embedded distribution. Using pip to manage dependencies as for a regular Python installation is not supported with this distribution, though with some care it may be possible to include and use pip for automatic updates. In general, third-party packages should be treated as part of the application (“vendoring”) so that the developer can ensure compatibility with newer versions before providing updates to users.

The two recommended use cases for this distribution are described below.

3.8.1. Python Application

An application written in Python does not necessarily require users to be aware of that fact. The embedded distribution may be used in this case to include a private version of Python in an install package. Depending on how transparent it should be (or conversely, how professional it should appear), there are two options.

Using a specialized executable as a launcher requires some coding, but provides the most transparent experience for users. With a customized launcher, there are no obvious indications that the program is running on Python: icons can be customized, company and version information can be specified, and file associations behave properly. In most cases, a custom launcher should simply be able to call `Py_Main` with a hard-coded command line.

The simpler approach is to provide a batch file or generated shortcut that directly calls the `python.exe` or `pythonw.exe` with the required command-line arguments. In this case, the application will appear to be Python and not its actual name, and users may have trouble distinguishing it from other running Python processes or file associations.

With the latter approach, packages should be installed as directories alongside the Python executable to ensure they are available on the path. With the specialized launcher, packages can be located in other locations as there is an opportunity to specify the search path before launching the application.

3.8.2. Embedding Python

Applications written in native code often require some form of scripting language, and the embedded Python distribution can be used for this purpose. In general, the majority of the application is in native code, and some part will either invoke `python.exe` or directly use `python3.dll`. For either case, extracting the embedded distribution to a subdirectory of the application installation is sufficient to provide a loadable Python interpreter.

As with the application use, packages can be installed to any location as there is an opportunity to specify search paths before initializing the interpreter. Otherwise, there is no fundamental differences between using the embedded distribution and a regular installation.

3.9. Other resources

See also:

[Python Programming On Win32](#)

“Help for Windows Programmers” by Mark Hammond and Andy Robinson, O’Reilly Media, 2000, ISBN 1-56592-621-8

[A Python for Windows Tutorial](#)

by Amanda Birmingham, 2004

[PEP 397 - Python launcher for Windows](#)

The proposal for the launcher to be included in the Python distribution.