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 require Python natively and thus does not pre-install a version of Python. However, the CPython team has compiled Windows installers (MSI packages) with every [release](http://www.python.org/download/releases/) for many years.

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**](http://www.python.org/dev/peps/pep-0011) for details on all unsupported platforms.

* [Windows CE](http://pythonce.sourceforge.net/) is still supported.
* The [Cygwin](http://cygwin.com/) installer offers to install the [Python interpreter](http://cygwin.com/packages/python) as well; it is located under “Interpreters.” (cf. [Cygwin package source](ftp://ftp.uni-erlangen.de/pub/pc/gnuwin32/cygwin/mirrors/cygnus/release/python), [Maintainer releases](http://www.tishler.net/jason/software/python/))

See [Python for Windows](http://www.python.org/download/windows/) for detailed information about platforms with pre-compiled installers.

**See also**

[**Python on XP**](http://www.richarddooling.com/index.php/2006/03/14/python-on-xp-7-minutes-to-hello-world/)

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

[**Installing on Windows**](http://diveintopython.net/installing_python/windows.html)

in “[Dive into Python: Python from novice to pro](http://diveintopython.net/index.html)” by Mark Pilgrim, 2004, ISBN 1-59059-356-1

[**For Windows users**](http://swaroopch.com/text/Byte_of_Python:Installing_Python#For_Windows_users)

in “Installing Python” in “[A Byte of Python](http://www.byteofpython.info/)” 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](http://www.activestate.com/activepython/)

Installer with multi-platform compatibility, documentation, PyWin32

[Enthought Python Distribution](http://www.enthought.com/products/epd.php)

Popular modules (such as PyWin32) with their respective documentation, tool suite for building extensible Python applications

Notice that these packages are likely to install *older* versions of Python.

3.3. Configuring Python

In order to run Python flawlessly, you might have to change certain environment settings in Windows.

3.3.1. Excursus: Setting environment variables

Windows has a built-in dialog for changing environment variables (following guide applies to XP classical view): Right-click the icon for your machine (usually located on your Desktop and called “My Computer”) and choose *Properties* there. Then, open the *Advanced* tab and click the *Environment Variables* button.

In short, your path is:

*My Computer ‣ Properties ‣ Advanced ‣ Environment Variables*

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).

Another way of adding variables to your environment is using the **set** command:

set PYTHONPATH=%PYTHONPATH%;C:\My\_python\_lib

To make this setting permanent, you could add the corresponding command line to your autoexec.bat. **msconfig** is a graphical interface to this file.

Viewing environment variables can also be done more straight-forward: The command prompt will expand strings wrapped into percent signs automatically:

echo %PATH%

Consult **set /?** for details on this behaviour.

**See also**

[**http://support.microsoft.com/kb/100843**](http://support.microsoft.com/kb/100843)

Environment variables in Windows NT

[**http://support.microsoft.com/kb/310519**](http://support.microsoft.com/kb/310519)

How To Manage Environment Variables in Windows XP

[**http://www.chem.gla.ac.uk/~louis/software/faq/q1.html**](http://www.chem.gla.ac.uk/~louis/software/faq/q1.html)

Setting Environment variables, Louis J. Farrugia

3.3.2. Finding the Python executable

*Changed in version 3.3.*

Besides using the automatically created start menu entry for the Python interpreter, you might want to start Python in the command prompt. As of Python 3.3, the installer has an option to set that up for you.

At the “Customize Python 3.3” screen, an option called “Add python.exe to search path” can be enabled to have the installer place your installation into the %PATH%. This allows you to type **python** to run the interpreter. Thus, you can also execute your scripts with command line options, see [*Command line*](https://docs.python.org/3.3/using/cmdline.html#using-on-cmdline)documentation.

If you don’t enable this option at install time, you can always re-run the installer to choose it.

The alternative is manually modifying the %PATH% using the directions in [*Excursus: Setting environment variables*](https://docs.python.org/3.3/using/windows.html#setting-envvars). You need to set your %PATH% environment variable to include the directory of your Python distribution, delimited by a semicolon from other entries. An example variable could look like this (assuming the first two entries are Windows’ default):

C:\WINDOWS\system32;C:\WINDOWS;C:\Python33

3.3.3. 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\.

This is how [sys.path](https://docs.python.org/3.3/library/sys.html" \l "sys.path" \o "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](https://docs.python.org/3.3/using/cmdline.html#envvar-PYTHONPATH) exists, as described in [*Environment variables*](https://docs.python.org/3.3/using/cmdline.html#using-on-envvars), 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 theHKEY\_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](https://docs.python.org/3.3/library/sys.html" \l "sys.path" \o "sys.path). (Note that all known installers only use HKLM, so HKCU is typically empty.)
* If the environment variable [PYTHONHOME](https://docs.python.org/3.3/using/cmdline.html#envvar-PYTHONHOME) is set, it is assumed as “Python Home”. Otherwise, the path of the main Python executable is used to locate a “landmark file” (Lib\os.py) to deduce the “Python Home”. If a Python home is found, the relevant sub-directories added to [sys.path](https://docs.python.org/3.3/library/sys.html" \l "sys.path" \o "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](https://docs.python.org/3.3/using/cmdline.html#envvar-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).

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 is no registry (eg, frozen .exe, some very strange installation setup) you get a path with some default, but relative, paths.

3.3.4. Executing scripts

As of Python 3.3, Python includes a launcher which facilitates running Python scripts. See [*Python Launcher for Windows*](https://docs.python.org/3.3/using/windows.html#launcher) for more information.

3.3.5. Executing scripts without the Python launcher

Without the Python launcher installed, Python scripts (files with the extension .py) will be executed by **python.exe** by default. This executable opens a terminal, which stays open even if the program uses a GUI. If you do not want this to happen, use the extension .pyw which will cause the script to be executed by **pythonw.exe** by default (both executables are located in the top-level of your Python installation directory). This suppresses the terminal window on startup.

You can also make all .py scripts execute with **pythonw.exe**, setting this through the usual facilities, for example (might require administrative rights):

1. Launch a command prompt.
2. Associate the correct file group with .py scripts:

assoc .py=Python.File

1. Redirect all Python files to the new executable:

ftype Python.File=C:\Path\to\pythonw.exe "%1" %\*

3.4. Python Launcher for Windows

*New in version 3.3.*

The Python launcher for Windows is a utility which aids in the location and execution 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.

3.4.1. Getting started

3.4.1.1. From the command-line

You should ensure the launcher is on your PATH - depending on how it was installed it may already be there, but check just in case it is not.

From a command-prompt, execute the following command:

py

You should find that the latest version of Python 2.x 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 2.x installed (e.g., 2.6 and 2.7) you will have noticed that Python 2.7 was started - to launch Python 2.6, try the command:

py -2.6

If you have a Python 3.x installed, try the command:

py -3

You should find the latest version of Python 3.x starts.

3.4.1.2. 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.

3.4.1.3. From file associations

The launcher should have been associated with Python files (i.e. .py, .pyw, .pyc, .pyo 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 are commonly used on such systems to indicate how a script should be executed. This launcher allows the same facilities to be using 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.

3.4.3. Arguments in shebThe 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. 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*](https://docs.python.org/3.3/library/windows.html#mswin-specific-services).

3.5.1. PyWin32

The [PyWin32](http://python.net/crew/mhammond/win32/) module by Mark Hammond is a collection of modules for advanced Windows-specific support. This includes utilities for:

* [Component Object Model](http://www.microsoft.com/com/) (COM)
* Win32 API calls
* Registry
* Event log
* [Microsoft Foundation Classes](http://msdn.microsoft.com/en-us/library/fe1cf721%28VS.80%29.aspx) (MFC) user interfaces
* [PythonWin](http://web.archive.org/web/20060524042422/http:/www.python.org/windows/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...?**](http://timgolden.me.uk/python/win32_how_do_i.html)

by Tim Golden

[**Python and COM**](http://www.boddie.org.uk/python/COM.html)

by David and Paul Boddie

3.5.2. cx\_Freeze

[cx\_Freeze](http://cx-freeze.sourceforge.net/) is a [distutils](https://docs.python.org/3.3/library/distutils.html" \l "module-distutils" \o "distutils: Support for building and installing Python modules into an existing Python installation.) extension (see [*Extending Distutils*](https://docs.python.org/3.3/distutils/extending.html#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.

3.5.3. WConio

Since Python’s advanced terminal handling layer, [curses](https://docs.python.org/3.3/library/curses.html#module-curses), is restricted to Unix-like systems, there is a library exclusive to Windows as well: Windows Console I/O for Python.

[WConio](http://newcenturycomputers.net/projects/wconio.html) is a wrapper for Turbo-C’s CONIO.H, used to create text user interfaces.

3.6. Compiling Python on Windows

If you want to compile CPython yourself, first thing you should do is get the [source](http://python.org/download/source/). You can download either the latest release’s source or just grab a fresh [checkout](http://docs.python.org/devguide/setup#checking-out-the-code).

The source tree contains a build solution and project files for Microsoft Visual C++, which is the compiler used to build the official Python releases. View the readme.txt in their respective directories:

| **Directory** | **MSVC version** | **Visual Studio version** |
| --- | --- | --- |
| PC/VC6/ | 6.0 | 97 |
| PC/VS7.1/ | 7.1 | 2003 |
| PC/VS8.0/ | 8.0 | 2005 |
| PC/VS9.0/ | 9.0 | 2008 |
| PCbuild/ | 10.0 | 2010 |

Note that any build directories within the PC directory are not necessarily fully supported. The PCbuild directory contains the files for the compiler used to build the official release.

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

For extension modules, consult [*Building C and C++ Extensions on Windows*](https://docs.python.org/3.3/extending/windows.html#building-on-windows).