Global and Virtual Environments, Plus Terminals

Contents

[Global and Virtual Environments, Plus Terminals 1](#_Toc86650677)

[Global and Virtual Environments 3](#_Toc86650678)

[Global and Virtual Environments 3](#_Toc86650679)

[Check the Version of the Python Interpreter 4](#_Toc86650680)

[Create a virtual environment 4](#_Toc86650681)

[Select and activate an environment 4](#_Toc86650682)

[Manually specify an interpreter# 7](#_Toc86650683)

[Environments and Terminal windows# 9](#_Toc86650684)

[Choose a debugging environment# 9](#_Toc86650685)

[Limited support for Python 2.7# 10](#_Toc86650686)

[Environment variables 10](#_Toc86650687)

[Environment variable definitions file# 10](#_Toc86650688)

[dev.env file 10](#_Toc86650689)

[prod.env file 11](#_Toc86650690)

[Variable substitution# 11](#_Toc86650691)

[Use of the PYTHONPATH variable# 12](#_Toc86650692)

[References 14](#_Toc86650693)

# Global and Virtual Environments

# Global and Virtual Environments

|  |  |
| --- | --- |
| By default, any Python interpreter that you've installed runs in its own **global environment**, which is not specific to any one project. For example, if you just run python (Windows) or python3 (macOS/Linux) at a new command prompt, you're running in that interpreter's global environment. Accordingly, any packages that you install or uninstall affect the global environment and all programs that you run within that context.  The packages installed in each virtual environment are seen only in that virtual environment and no other. Even large, complex packages with platform-dependent binaries can be corralled off from each other in virtual environments. | Although working in the global environment is an easy way to get started, that environment will, over time, become cluttered with many different packages that you've installed for different projects. Such clutter makes it difficult to thoroughly test an application against a specific set of packages with known versions, which is exactly the kind of environment you'd set up on a build server or web server.  A virtual environment is a way to have multiple, parallel instances of the Python interpreter, each with different package sets and different configurations. Each virtual environment contains a discrete copy of the Python interpreter, including copies of its support utilities. |
| For this reason, developers often create a virtual environment for a project. A virtual environment is a subfolder in a project that contains a copy of a specific interpreter. When you activate the virtual environment, any packages you install are installed only in that environment's subfolder. When you then run a Python program within that environment, you know that it's running against only those specific packages. Be aware that if you're not using a virtual environment, and you have multiple versions of Python installed and set in the path environment variable, you might need to specify the Python interpreter to use in the terminal for installing packages to the global environment. | For examples of using virtual environment in projects, see the [Python](https://code.visualstudio.com/docs/python/python-tutorial), [Django](https://code.visualstudio.com/docs/python/tutorial-django), and [Flask](https://code.visualstudio.com/docs/python/tutorial-flask) tutorials.  **Where Environments live**  Global:  ./AppData/locaPrograms/Python/Python39/python.exe  Local (venv): ~/venv/Scripts/python.exe  C:\Python\.venv\Scripts |

More Virtual Environment Use Cases:

1. You’re developing multiple projects that depend on different versions of the same packages, or you have a project that must be isolated from certain packages because of a namespace collision. This is the most standard use case.
2. You’re working in a Python environment where you can’t modify the site-packages directory. This may be because you’re working in a highly controlled environment, such as managed hosting, or on a server where the choice of interpreter (or packages used in it) can’t be changed because of production requirements.
3. You want to experiment with a specific combination of packages under highly controlled circumstances, for instance to test cross-compatibility or backward compatibility.
4. You want to run a “baseline” version of the Python interpreter on a system with no third-party packages, and only install third-party packages for each individual project as needed.

## Check the Version of the Python Interpreter

As the interpreter takes the code and helps in execution. Sometimes due to the old version of the interpreter making command doesn’t work properly and as we know the language python comes with frequent update vision with some added new features, so the user must update the interpreter with the latest release version

The packages installed in each virtual environment are seen only in that virtual environment and no other. Even large, complex packages with platform-dependent binaries can be corralled off from each other in virtual environments.

## Create a virtual environment

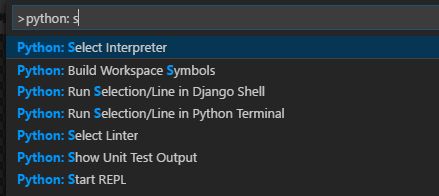
# Windows

# You can also use py -3 -m venv .venv

PS> python -m venv .venv

### Select and activate an environment

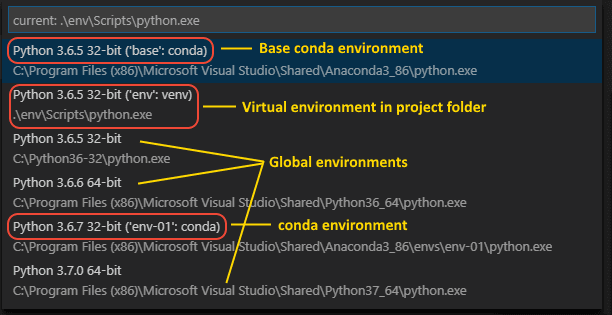
By default, the Python extension looks for and uses the first Python interpreter it finds in the system path. To select a specific environment, use the **Python: Select Interpreter** command from the **Command Palette** (Ctrl+Shift+P).



**Note**: If the Python extension doesn't find an interpreter, it issues a warning. On macOS, the extension also issues a warning if you're using the OS-installed Python interpreter, because you typically want to use an interpreter you install directly. In either case, you can disable these warnings by setting python.disableInstallationCheck to true in your user [settings](https://code.visualstudio.com/docs/getstarted/settings).

You can switch environments at any time; switching environments helps you test different parts of your project with different interpreters or library versions as needed.

The **Python: Select Interpreter** command displays a list of available global environments, conda environments, and virtual environments. (See the [Where the extension looks for environments](https://code.visualstudio.com/docs/python/environments#_where-the-extension-looks-for-environments) section for details, including the distinctions between these types of environments.) The following image, for example, shows several Anaconda and CPython installations along with a conda environment and a virtual environment (env) that's located within the workspace folder:



**Note:** On Windows, it can take a little time for VS Code to detect available conda environments. During that process, you may see "(cached)" before the path to an environment. The label indicates that VS Code is presently working with cached information for that environment.

If you have a folder or a workspace open in VS Code and you select an interpreter from the list, the Python extension will store that information internally so that the same interpreter will be used once you reopen the workspace.

The Python extension uses the selected environment for running Python code (using the **Python: Run Python File in Terminal** command), providing language services (auto-complete, syntax checking, linting, formatting, etc.) when you have a .py file open in the editor, and opening a terminal with the **Terminal: Create New Terminal** command. In the latter case, VS Code automatically activated the selected environment.

**Tip**: To prevent automatic activation of a selected environment, add "python.terminal.activateEnvironment": false to your settings.json file (it can be placed anywhere as a sibling to the existing settings).

**Tip**: If the activate command generates the message "Activate.ps1 is not digitally signed. You cannot run this script on the current system.", then you need to temporarily change the PowerShell execution policy to allow scripts to run (see [About Execution Policies](https://go.microsoft.com/fwlink/?LinkID=135170) in the PowerShell documentation): Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process

**Note**: By default, VS Code uses the interpreter selected for your workspace when debugging code. You can override this behavior by specifying a different path in the python property of a debug configuration. See [Choose a debugging environment](https://code.visualstudio.com/docs/python/environments#_choose-a-debugging-environment).

The Status Bar always shows the current interpreter.

Status Bar showing a selected interpreter

The Status Bar also reflects when no interpreter is selected.

No interpreter selected

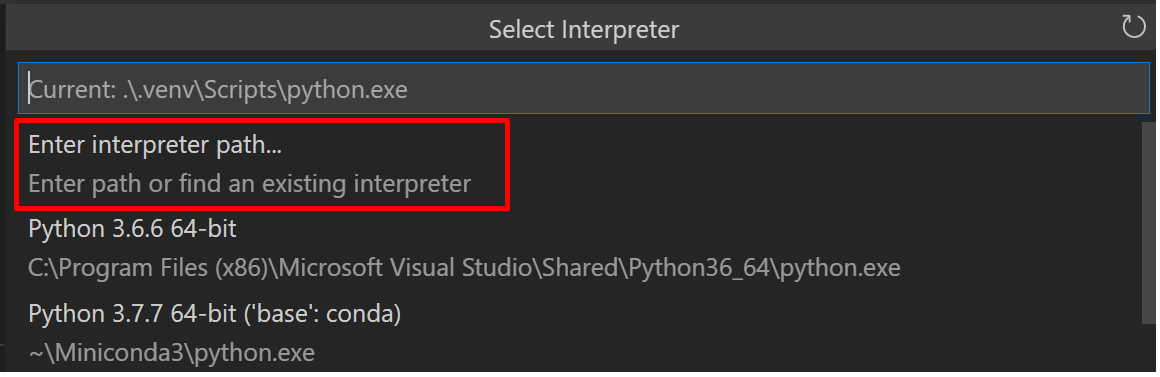
In either case, clicking this area of the Status Bar is a convenient shortcut for the **Python: Select Interpreter** command.

**Tip**: If you have any problems with VS Code recognizing a virtual environment, please [file an issue](https://github.com/microsoft/vscode-python/issues) in the extension repository so we can help determine the cause.

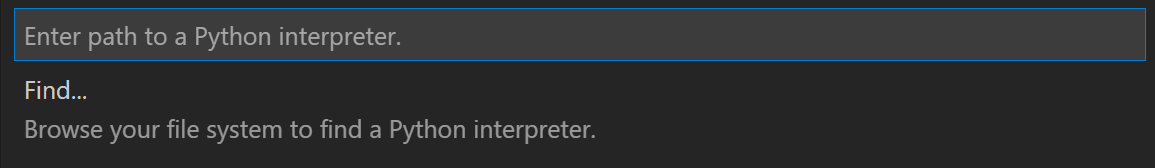
### Manually specify an interpreter[#](https://code.visualstudio.com/docs/python/environments#_manually-specify-an-interpreter)

If VS Code does not automatically locate an interpreter you want to use, you can browse for the interpreter on your file system or provide the path to it manually.

You can do so by running the **Python: Select Interpreter** command and clicking on the **Enter interpreter path...** option that shows on the top of the interpreters list:



You can then either enter the full path of the Python interpreter directly in the text box (for example, ".venv/Scripts/python.exe"), or you can click on the **Find...** button and browse your file system to find the python executable you wish to select.



If you want to manually specify a default interpreter that will be used once you first open your workspace, you can create or modify an entry for python.defaultInterpreterPath setting in your workspace settings.json with the full path to the Python executable.

For example:

* Windows:
* {
* "python.defaultInterpreterPath": "c:/python39/python.exe"

}

* macOS/Linux:
* {
* "python.defaultInterpreterPath": "/home/python39/python"

}

You can also use python.defaultInterpreterPath to point to a virtual environment, for example:

* Windows:
* {
* "python.defaultInterpreterPath": "c:/dev/ala/venv/Scripts/python.exe"

}

* macOS/Linux:
* {
* "python.defaultInterpreterPath": "/home/abc/dev/ala/venv/bin/python"

}

**Note**: Changes to the python.defaultInterpreterPath setting are not picked up after an interpreter has already been selected for a workspace; any changes to the setting will be ignored once an initial interpreter is selected for the workspace.

Additionally, if you'd like to set up a default interpreter to all of your Python applications, you can add an entry for python.defaultInterpreterPath manually inside your User Settings. To do so, open the Command Palette (Ctrl+Shift+P) and enter **Preferences: Open User Settings**. Then set python.defaultInterpreterPath, which is in the Python extension section of User Settings, with the appropriate interpreter.

You can also use an environment variable in the path setting using the syntax ${env:VARIABLE}. For example, if you've created a variable named PYTHON\_INSTALL\_LOC with a path to an interpreter, you can then use the following setting value:

"python.defaultInterpreterPath": "${env:PYTHON\_INSTALL\_LOC}",

**Note**: Variable substitution is only supported in VS Code settings files, it will not work in .env environment files.

By using an environment variable, you can easily transfer a project between operating systems where the paths are different, just be sure to set the environment variable on the operating system first.

### Environments and Terminal windows[#](https://code.visualstudio.com/docs/python/environments#_environments-and-terminal-windows)

After using **Python: Select Interpreter**, that interpreter is applied when right-clicking a file and selecting **Python: Run Python File in Terminal**. The environment is also activated automatically when you use the **Terminal: Create New Terminal** command unless you change the python.terminal.activateEnvironment setting to false.

However, launching VS Code from a shell in which a certain Python environment is activated does not automatically activate that environment in the default Integrated Terminal. Use the **Terminal: Create New Terminal** command after VS Code is running.

**Note:** conda environments cannot be automatically activated in the integrated terminal if PowerShell is set as the integrated shell. See [Integrated terminal - Configuration](https://code.visualstudio.com/docs/editor/integrated-terminal#_configuration) for how to change the shell.

Any changes you make to an activated environment within the terminal are persistent. For example, using conda install <package> from the terminal with a conda environment activated installs the package into that environment permanently. Similarly, using pip install in a terminal with a virtual environment activated adds the package to that environment.

Changing interpreters with the **Python: Select Interpreter** command doesn't affect terminal panels that are already open. You can thus activate separate environments in a split terminal: select the first interpreter, create a terminal for it, select a different interpreter, then use the split button (Ctrl+Shift+5) in the terminal title bar.

### Choose a debugging environment[#](https://code.visualstudio.com/docs/python/environments#_choose-a-debugging-environment)

By default, the debugger will use the Python interpreter you have selected with the Python extension. However, if you have a python property in the debug configuration of launch.json, that interpreter is used instead. To be more specific, VS Code will give precedence to the python property of the selected debug configuration in launch.json. If it's not defined, then it will use the path to the Python interpreter you have selected for your workspace.

For more details on debug configuration, see [Debugging configurations](https://code.visualstudio.com/docs/python/debugging).

### Limited support for Python 2.7[#](https://code.visualstudio.com/docs/python/environments#_limited-support-for-python-27)

The Python extension no longer offers IntelliSense support for Python 2.7 with [Jedi](https://pypi.org/project/jedi/) as it only supports Python 3 at this point. When using Python 2.7 with the Python extension you can customize the [language server setting](https://code.visualstudio.com/docs/python/settings-reference#_intellisense-engine-settings) to either turn off auto-completions or select Pylance as your language server, as it may provide a good experience if the code is compatible enough with Python 3.

We currently support selecting Python 2.7 as an interpreter in your workspace. Because [Python 2.7 is no longer maintained as of January 2020](https://www.python.org/doc/sunset-python-2/), we strongly suggest you to upgrade your code to Python 3 as soon as you can. You can [learn how to port your code to Python 3](https://docs.python.org/3/howto/pyporting.html) if you need help.

## Environment variables

### Environment variable definitions file[#](https://code.visualstudio.com/docs/python/environments#_environment-variable-definitions-file)

An environment variable definitions file is a simple text file containing key-value pairs in the form of environment\_variable=value, with # used for comments. Multiline values are not supported, but values can refer to any other environment variable that's already defined in the system or earlier in the file. For more information, see [Variable substitution](https://code.visualstudio.com/docs/python/environments#_variable-substitution). Environment variable definitions files can be used for scenarios such as debugging and tool execution (including linters, formatters, IntelliSense, and testing tools), but are not applied to the terminal.

By default, the Python extension looks for and loads a file named .env in the current workspace folder, then applies those definitions. The file is identified by the default entry "python.envFile": "${workspaceFolder}/.env" in your user settings (see [General settings](https://code.visualstudio.com/docs/python/settings-reference#_general-settings)). You can change the python.envFile setting at any time to use a different definitions file.

A debug configuration also contains an envFile property that also defaults to the .env file in the current workspace (see [Debugging - Set configuration options](https://code.visualstudio.com/docs/python/debugging#_set-configuration-options)). This property allows you to easily set variables for debugging purposes that replace variables specified in the default .env file.

For example, when developing a web application, you might want to easily switch between development and production servers. Instead of coding the different URLs and other settings into your application directly, you could use separate definitions files for each. For example:

### **dev.env file**

# dev.env - development configuration

# API endpoint

MYPROJECT\_APIENDPOINT=https://my.domain.com/api/dev/

# Variables for the database

MYPROJECT\_DBURL=https://my.domain.com/db/dev

MYPROJECT\_DBUSER=devadmin

MYPROJECT\_DBPASSWORD=!dfka\*\*213=

### **prod.env file**

# prod.env - production configuration

# API endpoint

MYPROJECT\_APIENDPOINT=https://my.domain.com/api/

# Variables for the database

MYPROJECT\_DBURL=https://my.domain.com/db/

MYPROJECT\_DBUSER=coreuser

MYPROJECT\_DBPASSWORD=kKKfa98\*11@

You can then set the python.envFile setting to ${workspaceFolder}/prod.env, then set the envFile property in the debug configuration to ${workspaceFolder}/dev.env.

**Note**: When environment variables are specified using multiple methods, be aware that there is an order of precedence. Environment variables contained in the .env file specified by the python.envFile setting (user or workspace) will override variables defined in the envFile specified in launch.json, as well as any env variables defined in the launch.json file itself. Similarly, environment variables defined in the envFile specified in launch.json will override env variables defined in the launch.json file.

### Variable substitution[#](https://code.visualstudio.com/docs/python/environments#_variable-substitution)

When defining an environment variable in a definitions file, you can use the value of any existing environment variable with the following general syntax:

<VARIABLE>=...${env:EXISTING\_VARIABLE}...

where ... means any other text as used in the value. The curly braces are required.

Within this syntax, the following rules apply:

* Variables are processed in the order they appear in the .env file, so you can use any variable that's defined earlier in the file.
* Single or double quotes don't affect substituted value and are included in the defined value. For example, if the value of VAR1 is abcedfg, then VAR2='${env:VAR1}' assigns the value 'abcedfg' to VAR2.
* The $ character can be escaped with a backslash, as in \$.
* You can use recursive substitution, such as PYTHONPATH=${env:PROJ\_DIR}:${env:PYTHONPATH} (where PROJ\_DIR is any other environment variable).
* You can use only simple substitution; nesting such as ${\_${env:VAR1}\_EX} is not supported.
* Entries with unsupported syntax are left as-is.

### Use of the PYTHONPATH variable[#](https://code.visualstudio.com/docs/python/environments#_use-of-the-pythonpath-variable)

The [PYTHONPATH](https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH) environment variable specifies additional locations where the Python interpreter should look for modules. In VS Code, PYTHONPATH can be set through the terminal settings (terminal.integrated.env.\*) and/or within an .env file.

When the terminal settings are used, PYTHONPATH affects any tools that are run within the terminal by a user, as well as any action the extension performs for a user that is routed through the terminal such as debugging. However, in this case when the extension is performing an action that isn't routed through the terminal, such as the use of a linter or formatter, then this setting will not have an effect on module look-up.

When PYTHONPATH is set using an .env file, it will affect anything the extension does on your behalf and actions performed by the debugger, but it will not affect tools run in the terminal.

If needed, you can set PYTHONPATH using both methods.

An example of when to use PYTHONPATH would be if you have source code in a src folder and tests in a tests folder. When running tests, however, those tests can't normally access modules in src unless you hard-code relative paths.

To solve this problem, you could add the path to src to PYTHONPATH by creating an .env file within your VS Code workspace.

PYTHONPATH=src

Then set python.envFile in your settings.json file to point to the .env file you just created. For example, if the .env file was in your workspace root, your settings.json would be set as shown:

"python.envFile": "${workspaceFolder}/.env"

The value of PYTHONPATH can contain multiple locations separated by os.pathsep: a semicolon (;) on Windows and a colon (:) on Linux/macOS. Invalid paths are ignored. If you find that your value for PYTHONPATH isn't working as expected, make sure that you're using the correct separator between locations for the operating system. For example, using a colon to separate locations on Windows, or using a semicolon to separate locations on Linux/macOS results in an invalid value for PYTHONPATH, which is ignored.

**Note**: PYTHONPATH does **not** specify a path to a Python interpreter itself. For additional information about PYTHONPATH, read the [PYTHONPATH documentation](https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH).

# References

* [Getting Started with Python in VS Code](https://code.visualstudio.com/docs/python/python-tutorial)
* [Using Python environments in VS Code](https://code.visualstudio.com/docs/python/environments#_global-virtual-and-conda-environments)
* [Check the Version of the Python Interpreter](https://www.geeksforgeeks.org/check-the-version-of-the-python-interpreter/)
* [Virtualenv and venv: Python virtual environments explained](https://www.infoworld.com/article/3239675/virtualenv-and-venv-python-virtual-environments-explained.html)
* [Environment Variable](https://www.techopedia.com/definition/15664/environment-variable)