Using Python Environments in Visual Studio Code

code.visualstudio.com/docs/python/environments

Microsoft

Version 1.83 is now available! Read about the new features and fixes from September.

Edit

Python environments in VS Code

An "environment" in Python is the context in which a Python program runs that consists of an interpreter and any number of installed packages.

> **Note**: If you'd like to become more familiar with the Python programming language, review More Python resources.

Types of Python environments

Global environments

By default, any Python interpreter installed runs in its own **global environment**. For example, if you just run `python`, `python3`, or `py` at a new terminal (depending on how you installed Python), you're running in that interpreter's global environment. Any packages that you install or uninstall affect the global environment and all programs that you run within it.

> **Tip**: In Python, it is best practice to create a workspace-specific environment, for example, by using a local environment.

Local environments

There are two types of environments that you can create for your workspace: **virtual** and **conda**. These environments allow you to install packages without affecting other environments, isolating your workspace's package installations.

Virtual environments

A **virtual environment** is a built-in way to create an environment. A virtual environment creates a folder that contains a copy (or symlink) to a specific interpreter. When you install packages into a virtual environment it will end up in this new folder, and thus isolated from other packages used by other workspaces.

> **Note**: While it's possible to open a virtual environment folder as a workspace, doing so is not recommended and might cause issues with using the Python extension.

Conda environments

A **conda environment** is a Python environment that's managed using the `conda` package manager (see Getting started with conda).Choosing between conda and virtual environments depends on your packaging needs, team standards, etc.

Python environment tools

The following table lists the various tools involved with Python environments:

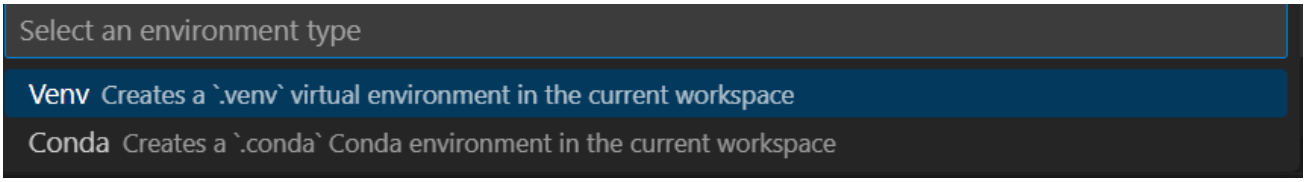
Tool	Definition and Purpose
pip	The Python package manager that installs and updates packages. It's installed with Python 3.9+ by default (unless you are on a Debian-based OS; install `python3-pip` in that case).
venv	Allows you to manage separate package installations for different projects and is installed with Python 3 by default (unless you are on a Debian-based OS; install `python3-venv` in that case)
conda	Installed with **Miniconda**. It can be used to manage both packages and virtual environments. Generally used for data science projects.

Creating environments

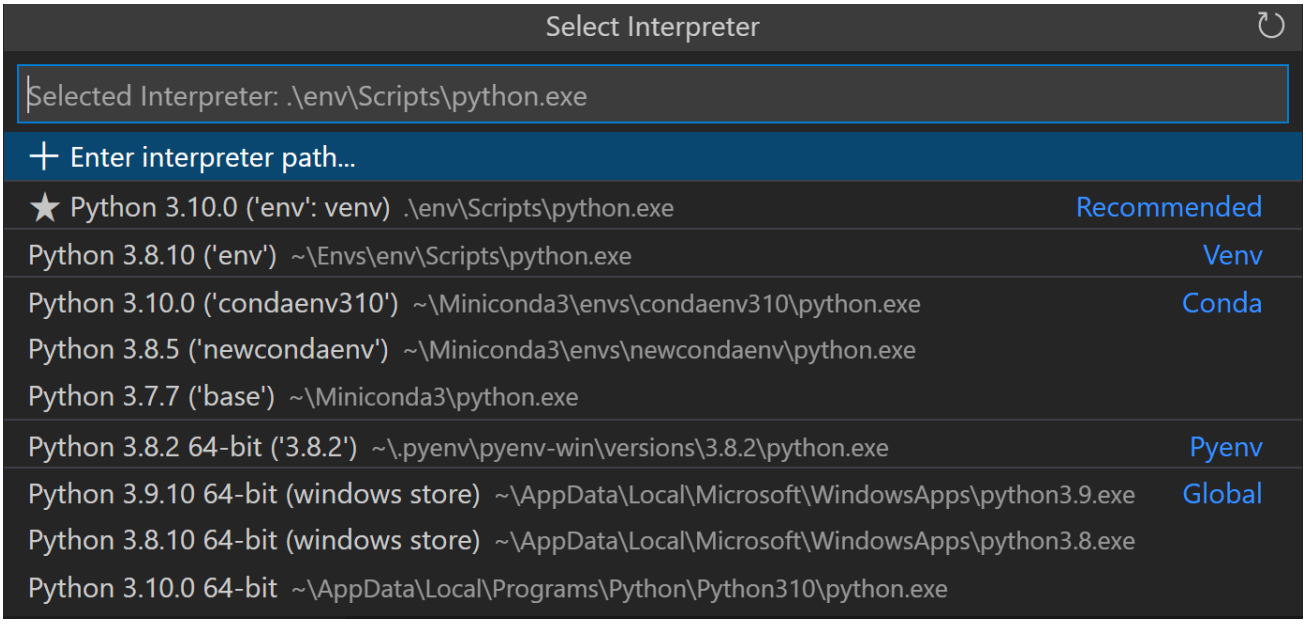
Using the Create Environment command

To create local environments in VS Code using virtual environments or Anaconda, you can follow these steps: open the Command Palette (Ctrl+Shift+P), search for the **Python: Create Environment** command, and select it.

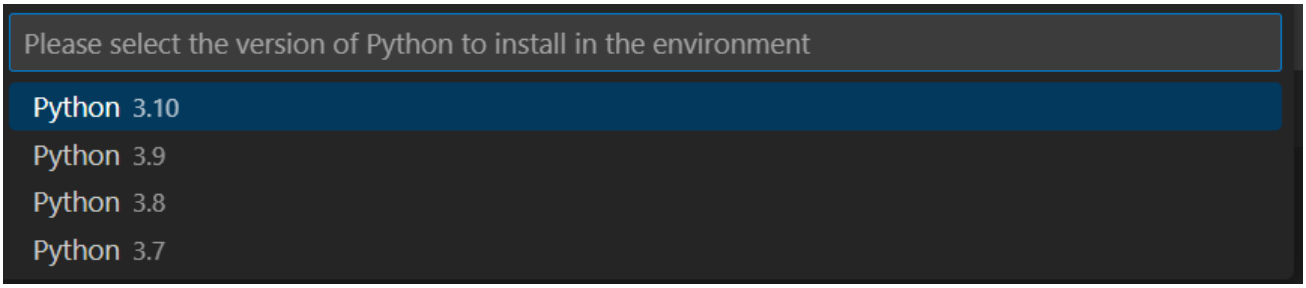
The command presents a list of environment types: **Venv** or **Conda**.



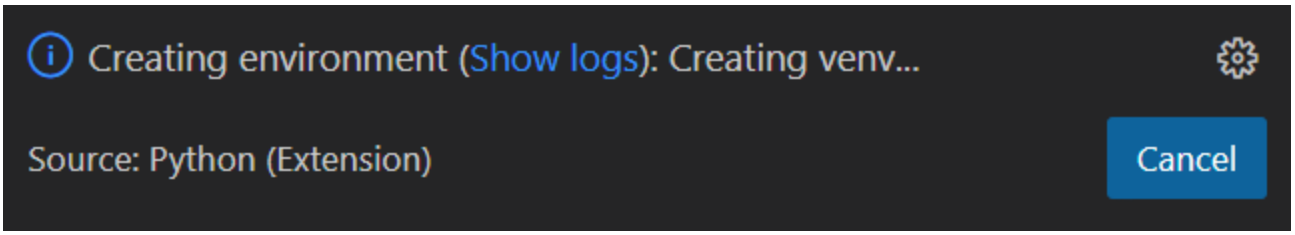
If you are creating an environment using **Venv**, the command presents a list of interpreters that can be used as a base for the new virtual environment.



If you are creating an environment using **Conda**, the command presents a list of Python versions that can be used for your project.



After selecting the desired interpreter or Python version, a notification will show the progress of the environment creation and the environment folder will appear in your workspace.



> **Note**: The command will also install necessary packages outlined in a requirements/dependencies file, such as `requirements.txt`, `pyproject.toml`, or `environment.yml`, located in the project folder. It will also add a `.gitignore` file to the virtual environment to help prevent you from accidentally committing the virtual environment to source control.

Create a virtual environment in the terminal

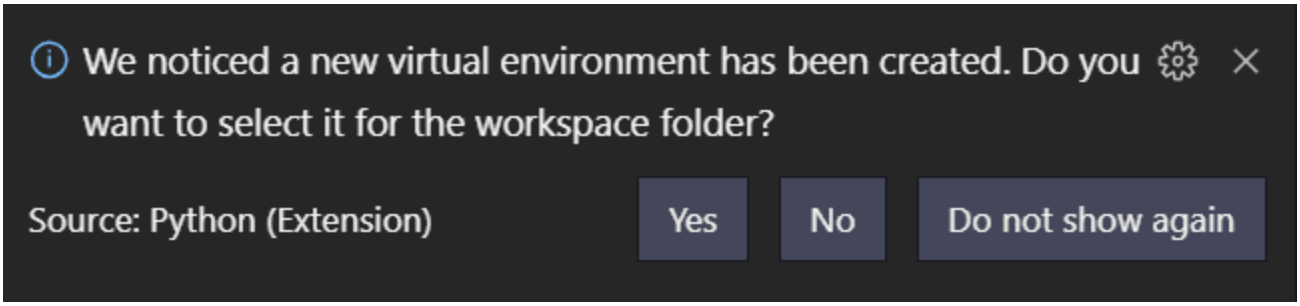
If you choose to create a virtual environment manually, use the following command (where ".venv" is the name of the environment folder):

```
# macOS/Linux
# You may need to run `sudo apt-get install python3-venv` first on Debian-based OSs
python3 -m venv .venv

# Windows
# You can also use `py -3 -m venv .venv`
python -m venv .venv
```

> **Note**: To learn more about the `venv` module, read Creation of virtual environments on Python.org.

When you create a new virtual environment, a prompt will be displayed in VS Code to allow you to select it for the workspace.



> **Tip**: Make sure to update your source control settings to prevent accidentally committing your virtual environment (in for example `.gitignore`). Since virtual environments are not portable, it typically does not make sense to commit them for others to use.

Create a conda environment in the terminal

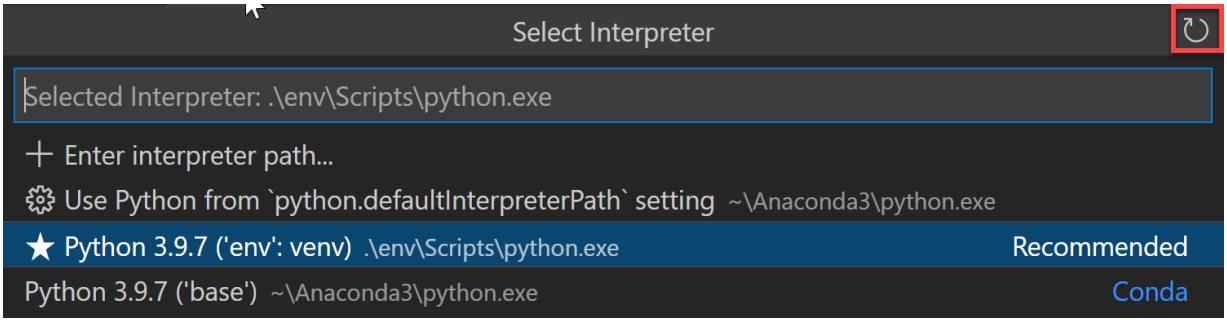
The Python extension automatically detects existing conda environments. We recommend you install a Python interpreter into your conda environment, otherwise one will be installed for you after you select the environment. For example, the following command creates a conda environment named `env-01` with a Python 3.9 interpreter and several libraries:

```
conda create -n env-01 python=3.9 scipy=0.15.0 numpy
```

> **Note**: For more information on the conda command line, you can read Conda environments.

Additional notes:

- If you create a new conda environment while VS Code is running, use the refresh icon on the top right of the **Python: Select Interpreter** window; otherwise you may not find the environment there.



- To ensure the environment is properly set up from a shell perspective, use an Anaconda prompt and activate the desired environment. Then, you can launch VS Code by entering the `code .` command. Once VS Code is open, you can select the interpreter either by using the Command Palette or by clicking on the status bar.

- Although the Python extension for VS Code doesn't currently have direct integration with conda `environment.yml` files, VS Code itself is a great YAML editor.

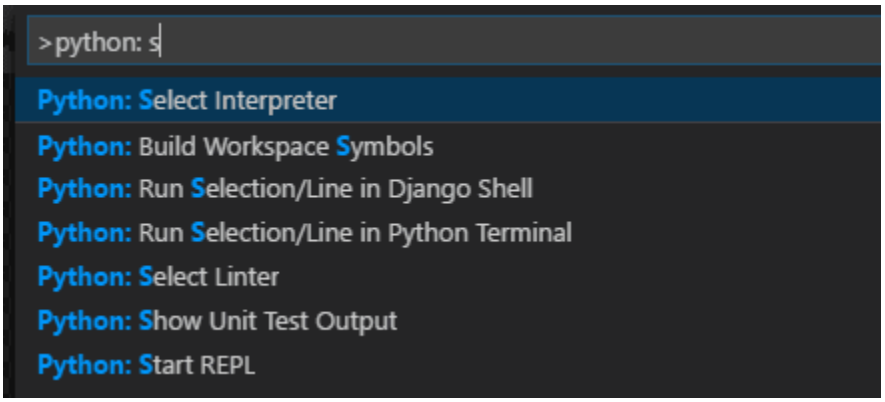
- Conda environments can't be automatically activated in the VS Code Integrated Terminal if the default shell is set to PowerShell. To change the shell, see Integrated terminal - Terminal profiles.

- You can manually specify the path to the `conda` executable to use for activation (version 4.4+). To do so, open the Command Palette (Ctrl+Shift+P) and run **Preferences: Open User Settings**. Then set `python.condaPath`, which is in the Python extension section of User Settings, with the appropriate path.

Working with Python interpreters

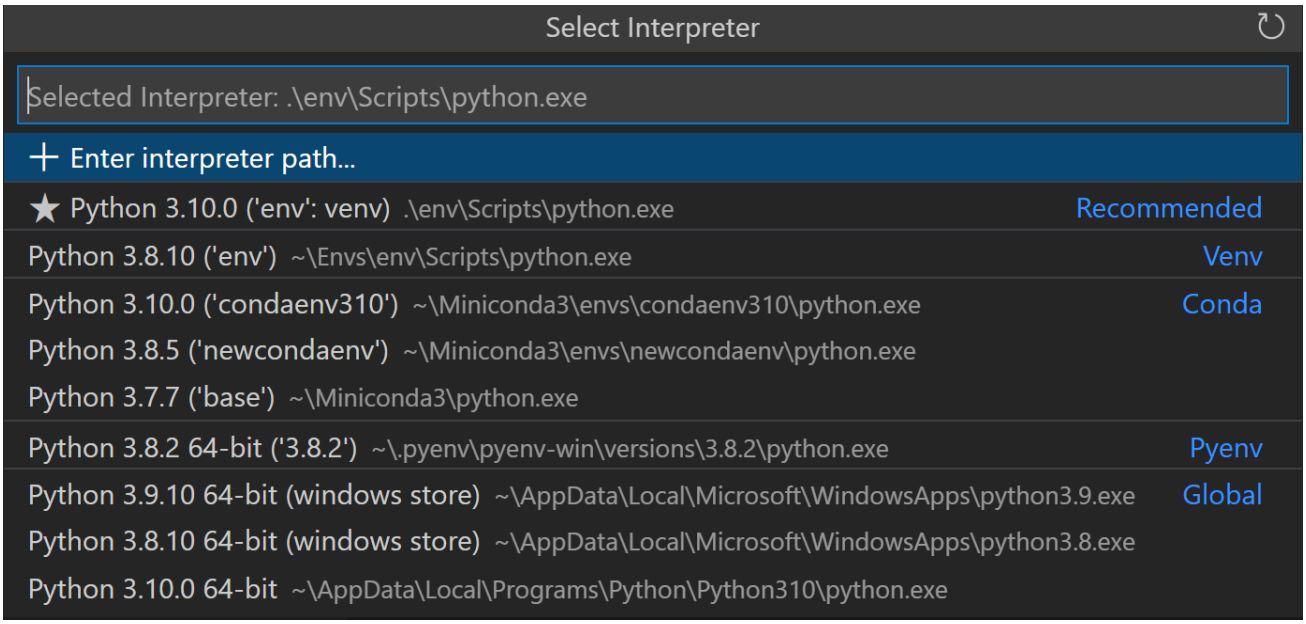
Select and activate an environment

The Python extension tries to find and then select what it deems the best environment for the workspace. If you would prefer 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 12.2 and older, the extension also issues a warning if you're using the OS-installed Python interpreter as it is known to have compatibility issues. In either case, you can disable these warnings by setting `python.disableInstallationCheck` to `true` in your user settings.

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 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. This ensures that the same interpreter will be used when you reopen the workspace.

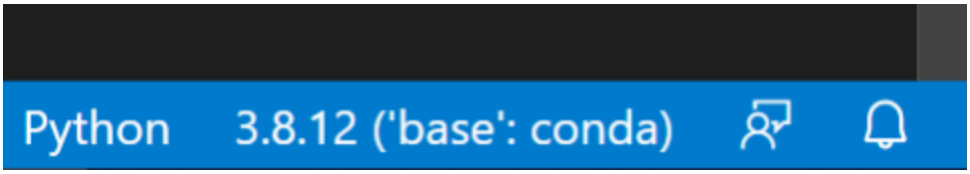
The selected environment is used by the Python extension for running Python code (using the **Python: Run Python File in Terminal** command), providing language services (auto-complete, syntax checking, linting, formatting, etc.) when you have a `.py` file open in the editor, and opening a terminal with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

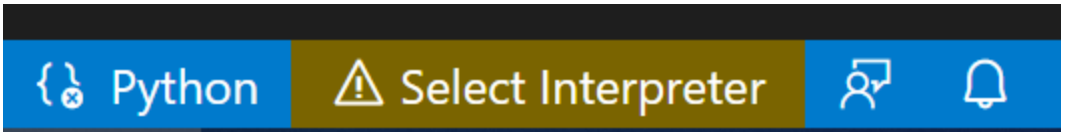
> **Tip**: If the activate command generates the message "Activate.ps1 is not digitally signed. You cannot run this script on the current system.", then you need to temporarily change the PowerShell execution policy to allow scripts to run (see About Execution Policies in the PowerShell documentation): `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.

The selected interpreter version will show on the right side of the Status Bar.



The Status Bar also reflects when no interpreter is selected.



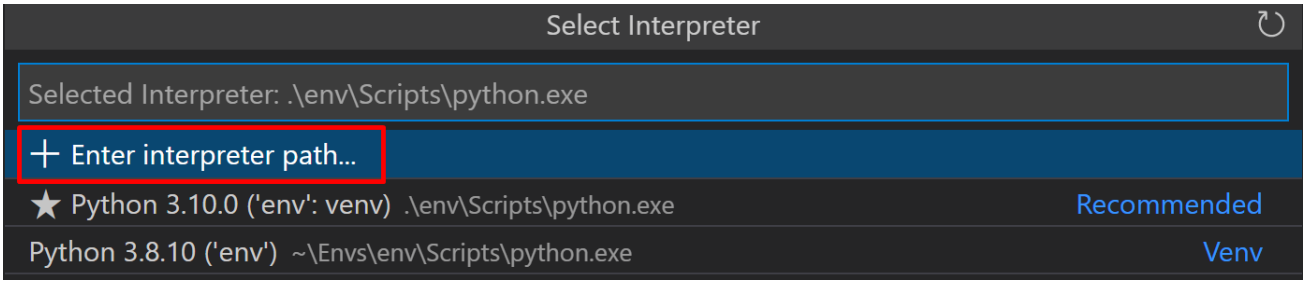
In either case, clicking this area of the Status Bar is a convenient shortcut for the **Python: Select Interpreter** command.

> **Tip**: If you have any problems with VS Code recognizing a virtual environment, please file an issue so we can help determine the cause.

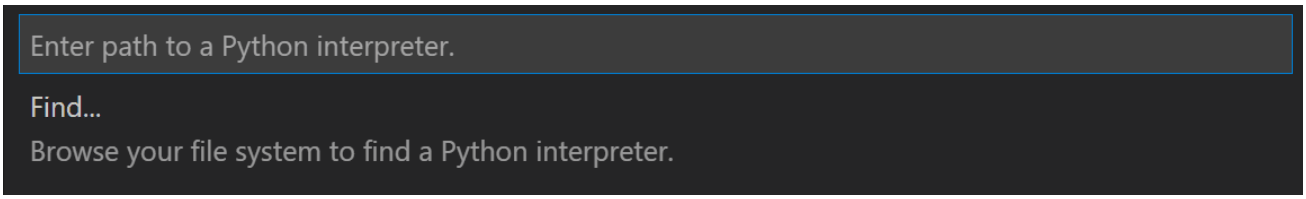
Manually specify an interpreter

If VS Code doesn't 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 select 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 select 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 when you first open your workspace, you can create or modify an entry for the `python.defaultInterpreterPath` setting.

> **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.

How the extension chooses an environment automatically

If an interpreter hasn't been specified, then the Python extension automatically selects the interpreter with the highest version in the following priority order:

1. Virtual environments located directly under the workspace folder.
2. Virtual environments related to the workspace but stored globally. For example, Pipenv or Poetry environments that are located outside of the workspace folder.

3. Globally installed interpreters. For example, the ones found in `/usr/local/bin`, `C:\\python38`, etc.

> **Note**: The interpreter selected may differ from what `python` refers to in your terminal.

If Visual Studio Code doesn't locate your interpreter automatically, you can manually specify an interpreter.

Where the extension looks for environments

The extension automatically looks for interpreters in the following locations, in no particular order:

- Standard install paths such as `/usr/local/bin`, `/usr/sbin`, `/sbin`, `c:\\python36`, etc.
- Virtual environments located directly under the workspace (project) folder.
- Virtual environments located in the folder identified by the `python.venvPath` setting (see General Python settings), which can contain multiple virtual environments. The extension looks for virtual environments in the first-level subfolders of `venvPath`.
- Virtual environments located in a `~/.virtualenvs` folder for virtualenvwrapper.
- Interpreters created by pyenv, Pipenv, and Poetry.
- Virtual environments located in the path identified by `WORKON_HOME` (as used by virtualenvwrapper).
- Conda environments found by `conda env list`. Conda environments which do not have an interpreter will have one installed for them upon selection.
- Interpreters installed in a `.direnv` folder for direnv under the workspace folder.

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`.

Please note that launching VS Code from a shell in which a specific Python environment is activated doesn't automatically activate that environment in the default Integrated Terminal.

> **Note:** conda environments cannot be automatically activated in the integrated terminal if PowerShell is set as the integrated shell. See Integrated terminal - Terminal profiles for how to change the shell.

Changing interpreters with the **Python: Select Interpreter** command doesn't affect terminal panels that are already open. Thus, you can 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

By default, the debugger will use the Python interpreter chosen with the Python extension. However, if there is a `python` property specified in the debug configuration of `launch.json`, it takes precedence. If this property is not defined, it will fall back to using the Python interpreter path selected for the workspace.

For more details on debug configuration, see Debugging configurations.

Environment variables

Environment variable definitions file

An environment variable definitions file is a text file containing key-value pairs in the form of `environment_variable=value`, with `#` used for comments. Multiline values aren't supported, but references to previously defined environment variables are allowed. Environment variable definitions files can be used for scenarios such as debugging and tool execution (including linters, formatters, IntelliSense, and testing tools), but aren't applied to the terminal.

> **Note**: Environment variable definitions files are not necessarily cross-platform. For instance, while Unix uses `:` as a path separator in environment variables, Windows uses `;`. There is no normalization of such operating system differences, and so you need to make sure any environment definitions file use values that are compatible with your operating system.

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 Python settings). You can change the `python.envFile` setting at any time to use a different definitions file.

> **Note**: Environment variable definitions files are not used in all situations where environment variables are available for use. Unless Visual Studio Code documentation states otherwise, these only affect certain scenarios as per their definition. For example, the extension doesn't use environment variable definitions files when resolving setting values.

A debug configuration also contains an `envFile` property that also defaults to the `.env` file in the current workspace (see 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. All `env` variables defined in the `launch.json` file will override variables contained in the `.env` file, specified by the `python.envFile` setting (user or workspace). Similarly, `env` variables defined in the `launch.json` file will override the environment variables defined in the `envFile` that are specified in `launch.json`.

Use of the PYTHONPATH variable

The 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 won't have an effect on module look-up.

Next steps

More Python resources

Was this documentation helpful?

6/12/2023