

Notes for Visual Studio Code

Programming Languages

Changing the language for the selected file

In VS Code, we default the language support for a file based on its filename extension. However, at times you may wish to change language modes, to do this click on the language indicator - which is located on the right hand of the Status Bar. This will bring up the **Select Language Mode** drop-down where you can select another language for the current file.

A screenshot of the Visual Studio Code status bar. It shows a blue bar with white text. On the left, it says 'Ln 33, Col 1'. In the center, it says 'UTF-8' and 'CRLF'. On the right, it says 'Markdown'. A small white triangle points upwards from the 'Markdown' text.

Tip: You can get the same drop-down by running the **Change Language Mode** command ([Ctrl+K M](#)).

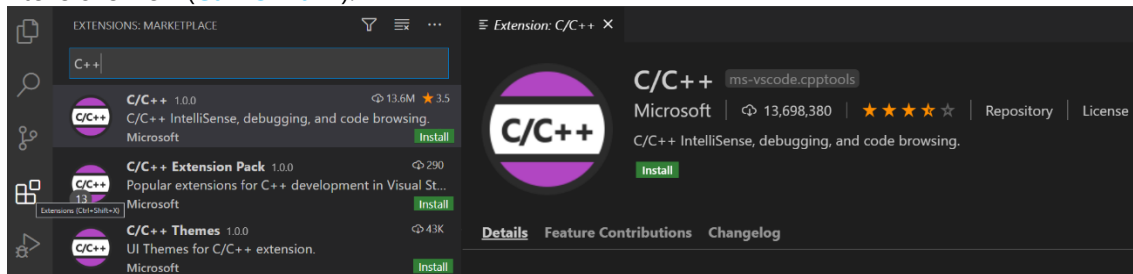
Configure VS Code for Microsoft C++/Linux

In this tutorial, you configure Visual Studio Code to use the Microsoft Visual C++ compiler and debugger on Windows.

Second, you will configure Visual Studio Code to use the [GCC](#) C++ compiler (g++) and GDB debugger on Linux. GCC stands for GNU Compiler Collection; GDB is the GNU debugger.

Prerequisites

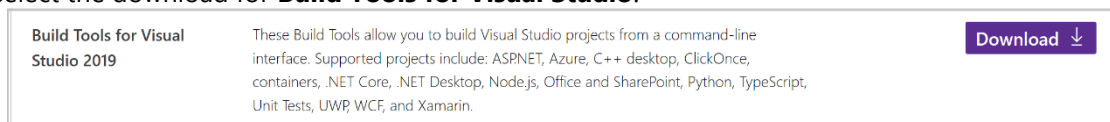
1. Install [Visual Studio Code](#).
2. Install the [C/C++ extension for VS Code](#). You can install the C/C++ extension by searching for 'c++' in the Extensions view (**Ctrl+Shift+X**).



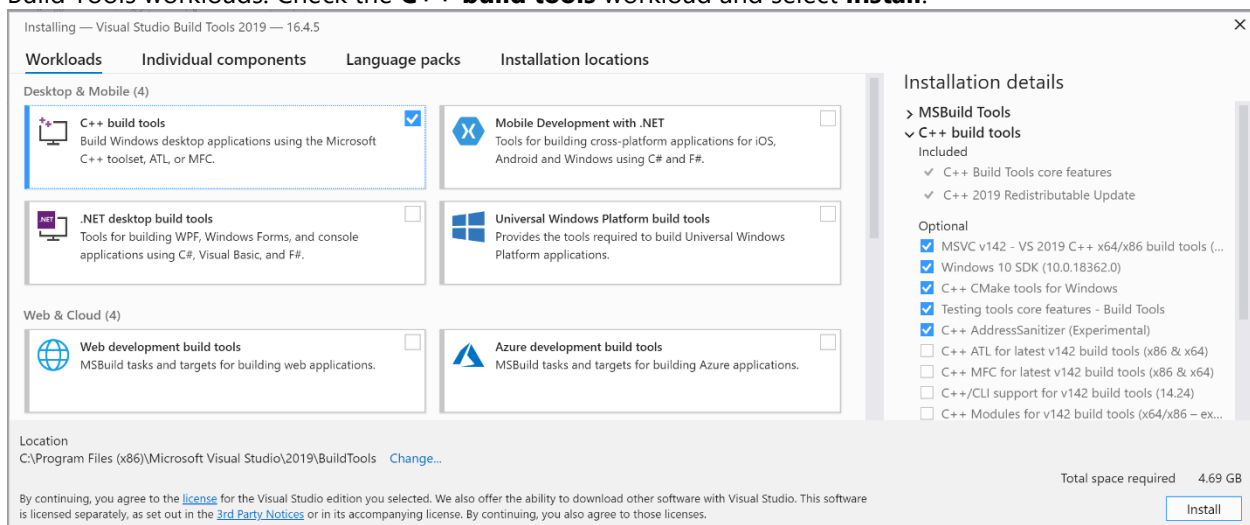
3. Install the Microsoft Visual C++ (MSVC) compiler toolset.

If you have a recent version of Visual Studio, open the Visual Studio Installer from the Windows Start menu and verify that the C++ workload is checked. If it's not installed, then check the box and click the **Modify** button in the installer.

You can **also** install just the **C++ Build Tools**, without a full Visual Studio IDE installation. From the Visual Studio [Downloads](#) page, scroll down until you see **Tools for Visual Studio** under the **All downloads** section and select the download for **Build Tools for Visual Studio**.



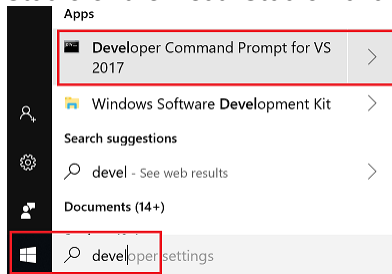
This will launch the Visual Studio Installer, which will bring up a dialog showing the available Visual Studio Build Tools workloads. Check the **C++ build tools** workload and select **Install**.



Check your Microsoft Visual C++ installation

To use MSVC from a command line or VS Code, you **must** run from a **Developer Command Prompt for Visual Studio**. An ordinary shell such as PowerShell, Bash, or the Windows command prompt **does not** have the necessary path environment variables set.

To open the Developer Command Prompt for VS, start typing 'developer' in the Windows Start menu, and you should see it appear in the list of suggestions. The exact name depends on which version of Visual Studio or the Visual Studio Build Tools you have installed. Click on the item to open the prompt.



You can **test** that you have the C++ compiler, **cl.exe**, installed correctly by typing 'cl' and you should see a copyright message with the version and basic usage description (**x86** version).

```
C:\> Developer Command Prompt for VS 2019

*****
** Visual Studio 2019 Developer Command Prompt v16.4.5
** Copyright (c) 2019 Microsoft Corporation
*****

C:\Program Files (x86)\Microsoft Visual Studio\2019\BuildTools>cl
Microsoft (R) C/C++ Optimizing Compiler Version 19.24.28316 for x86
Copyright (C) Microsoft Corporation. All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]

C:\Program Files (x86)\Microsoft Visual Studio\2019\BuildTools>
```

(x64 version)

```
C:\> x64 Native Tools Command Prompt for VS 2019

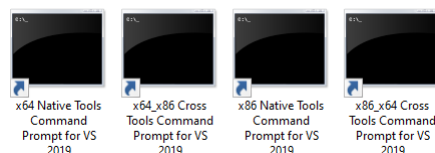
D:\temp\diffusion-cuda>cl
Microsoft (R) C/C++ Optimizing Compiler Version 19.28.29914 for x64
Copyright (C) Microsoft Corporation. All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]

D:\temp\diffusion-cuda>
```

If the Developer Command Prompt is using the BuildTools location as the starting directory (you **wouldn't** want to put projects there), **navigate** to your user folder (**C:\users\{your username}**) before you start creating new projects.

Note: Now we have **multiple versions** of cl.exe (i.e., x86 and x64), and below Command Prompt can be used to select those versions:



What environment variables does MSVC add?

Original path: (cmd.exe: `echo %Path%`, PowerShell: `$Env:Path`)

```
D:\Softwares\VMware\VMware Workstation\bin\;C:\Program Files\NVIDIA GPU Computing
Toolkit\CUDA\v11.3\bin;C:\Program Files\NVIDIA GPU Computing
Toolkit\CUDA\v11.3\libnvvp;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WI
NDOWS\System32\WindowsPowerShell\v1.0\;C:\WINDOWS\System32\OpenSSH\;C:\Program Files
(x86)\NVIDIA Corporation\PhysX\Common;C:\Program Files\NVIDIA Corporation\NVIDIA
NvDLISR;C:\Program Files\PuTTY\;C:\Program Files\NVIDIA Corporation\Nsight Compute
2021.1.1\;D:\Softwares\Git\cmd;C:\Program
Files\dotnet\;C:\Users\Yousei\AppData\Local\Microsoft\WindowsApps;C:\Users\Yousei\AppData\Local\Pro
grams\Microsoft_VS_Code\bin;
```

MSVC's path:

```
D:\Softwares\Microsoft_Visual_Studio\2019\BuildTools\VC\Tools\MSVC\14.29.30133\bin\HostX64\x64;D:\
Softwares\Microsoft_Visual_Studio\2019\BuildTools\Common7\IDE\VC\VCackages;D:\Softwares\Microso
ft_Visual_Studio\2019\BuildTools\Common7\IDE\CommonExtensions\Microsoft\TestWindow;D:\Softwares
\Microsoft_Visual_Studio\2019\BuildTools\Common7\IDE\CommonExtensions\Microsoft\TeamFoundation
\Team
Explorer;D:\Softwares\Microsoft_Visual_Studio\2019\BuildTools\MSBuild\Current\bin\Roslyn;D:\Softwares\
Microsoft_Visual_Studio\2019\BuildTools\Common7\Tools\devinit;C:\Program Files (x86)\Windows
Kits\10\bin\10.0.18362.0\x64;C:\Program Files (x86)\Windows
Kits\10\bin\x64;D:\Softwares\Microsoft_Visual_Studio\2019\BuildTools\MSBuild\Current\Bin;C:\Windows\
Microsoft.NET\Framework64\v4.0.30319;D:\Softwares\Microsoft_Visual_Studio\2019\BuildTools\Common7
\IDE\;D:\Softwares\Microsoft_Visual_Studio\2019\BuildTools\Common7\Tools\;D:\Softwares\VMware\VM
ware Workstation\bin\;C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.3\bin;C:\Program
Files\NVIDIA GPU Computing
Toolkit\CUDA\v11.3\libnvvp;C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem;C:\WI
NDOWS\System32\WindowsPowerShell\v1.0\;C:\WINDOWS\System32\OpenSSH\;C:\Program Files
(x86)\NVIDIA Corporation\PhysX\Common;C:\Program Files\NVIDIA Corporation\NVIDIA
NvDLISR;C:\Program Files\PuTTY\;C:\Program Files\NVIDIA Corporation\Nsight Compute
2021.1.1\;D:\Softwares\Git\cmd;C:\Program
Files\dotnet\;C:\Users\Yousei\AppData\Local\Microsoft\WindowsApps;C:\Users\Yousei\AppData\Local\Pro
grams\Microsoft_VS_Code\bin;;D:\Softwares\Microsoft_Visual_Studio\2019\BuildTools\Common7\IDE\Co
mmonExtensions\Microsoft\CMake\CMake\bin;D:\Softwares\Microsoft_Visual_Studio\2019\BuildTools\Co
mmon7\IDE\CommonExtensions\Microsoft\CMake\Ninja
```

Ensure GCC is installed (For Linux)

Although you'll use VS Code to edit your source code, you'll compile the source code on Linux using the **g++** compiler. You'll also use **GDB** to debug. These tools are not installed by default on Ubuntu, so you have to install them. Fortunately, that's easy.

First, check to see whether GCC is already installed. To verify whether it is, open a Terminal window and enter the following command:

```
gcc -v
```

If GCC isn't installed, run the following command from the terminal window to update the Ubuntu package lists. An out-of-date Linux distribution can sometimes interfere with attempts to install new packages.

```
sudo apt-get update
```

Next install the GNU compiler tools and the GDB debugger with this command:

```
sudo apt-get install build-essential gdb
```

Note: If for some reason you can't run VS Code from a **Developer Command Prompt**, you can find a workaround for building C++ projects with VS Code in **Run VS Code outside a Developer Command Prompt**.

Create Hello World

From the **Developer Command Prompt**, create an empty folder called "projects" where you can **store** all your VS Code **projects**, then create a **subfolder** called "helloworld", navigate into it, and **open** VS Code (**code**) in that folder (.) by entering the following commands:

```
mkdir projects
cd projects
mkdir helloworld
cd helloworld
code .
```

For Linux:

From the terminal window, create an empty folder called **projects** to store your VS Code projects. Then create a subfolder called **helloworld**, navigate into it, and open VS Code in that folder by entering the following commands:

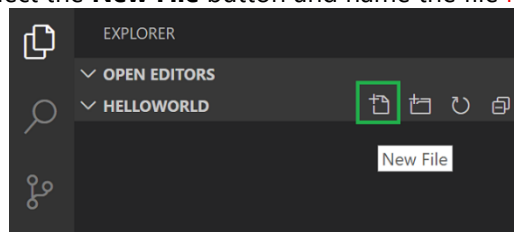
```
mkdir projects
cd projects
mkdir helloworld
cd helloworld
code .
```

The "**code .**" command opens VS Code in the **current** working folder, which becomes your "**workspace**". As you go through the tutorial, you **will** see three files created in a **.vscode** folder in the workspace:

- **tasks.json** (build instructions)
- **launch.json** (debugger settings)
- **c_cpp_properties.json** (compiler path and IntelliSense settings)

Add a source code file

In the File Explorer title bar, select the **New File** button and name the file **helloworld.cpp**.



Paste in the following source code:

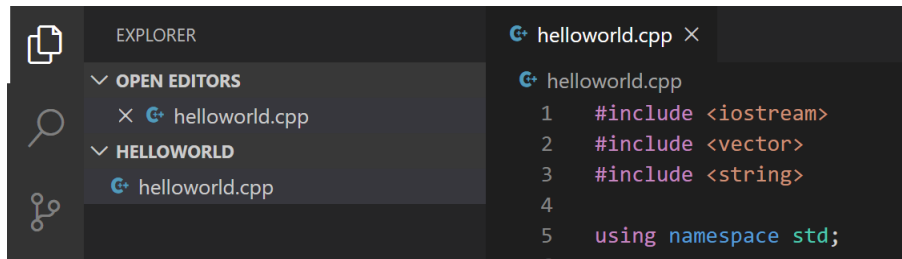
```
#include <iostream>
#include <vector>
#include <string>

using namespace std;

int main()
{
    vector<string> msg {"Hello", "C++", "World", "from", "VS Code", "and the C++ extension!"};

    for (const string& word : msg)
    {
        cout << word << " ";
    }
    cout << endl;
}
```

Now press **Ctrl+S** to save the file. Notice how the file you just added appears in the **File Explorer** view (**Ctrl+Shift+E**) in the side bar of VS Code:

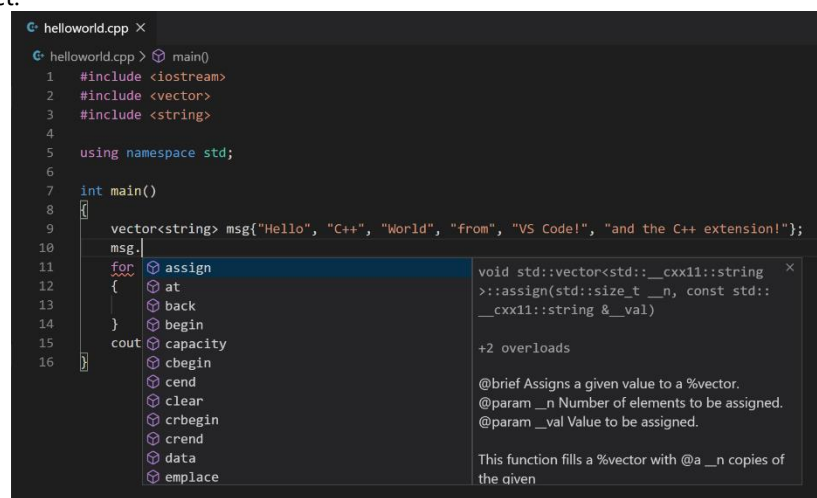


You can also enable **Auto Save** to automatically save your file changes, by checking **Auto Save** in the main **File** menu.

The Activity Bar on the far left lets you open different views such as **Search**, **Source Control**, and **Run**. You'll look at the **Run** view later in this tutorial. You can find out more about the other views in the VS Code [User Interface documentation](#).

Explore IntelliSense

In your new **helloworld.cpp** file, hover over **vector** or **string** to see type information. After the declaration of the **msg** variable, start typing **msg.** as you would when calling a member function. You should immediately see a completion list that shows all the member functions, and a window that shows the type information for the **msg** object:



You can press the **Tab** key to **insert** the selected member; then, when you add the opening parenthesis, you will see information about any arguments that the function requires.

Build helloworld.cpp

Next, you will create a **tasks.json** file to **tell** VS Code how to **build (compile)** the program. This task will invoke the Microsoft C++ compiler to create **an executable file** based on the source code.

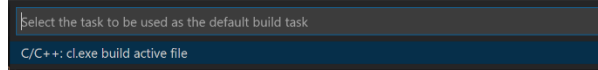
For Linux:

Next, you'll create a **tasks.json** file to tell VS Code how to build (compile) the program. This task will invoke the g++ compiler to create an executable file from the source code.

It's **important** to have **helloworld.cpp** **open** in the editor because the next step uses the **active file** in the editor for context to create the build task in the next step.

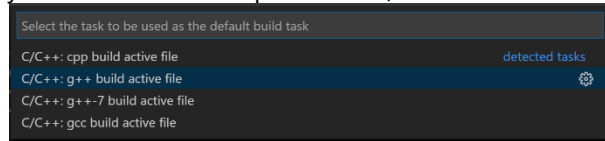
From the main menu, choose **Terminal > Configure Default Build Task**. In the dropdown, which will display a tasks dropdown listing various predefined build tasks for C++ compilers. Choose **cl.exe build**

active file, which will build the file that is currently displayed (**active**) in the editor (If you already have tasks.json file, you may see some same dropdown lists).



For Linux:

From the main menu, choose **Terminal > Configure Default Build Task**. A dropdown appears showing various predefined build tasks for C++ compilers. Choose **C/C++: g++ build active file**. (If you already have tasks.json file, you may see some same dropdown lists)



This will **create** a **tasks.json** file in a **.vscode** folder and open it in the editor. Your **new tasks.json** file should look similar to the JSON below:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "shell",
      "label": "cl.exe build active file",
      "command": "cl.exe",
      "args": [
        "/zi",
        "/EHsc",
        "/Fe:",
        "${fileDirname}\\${fileBasenameNoExtension}.exe",
        "${file}"
      ],
      "problemMatcher": ["$msCompile"],
      "group": {
        "kind": "build",
        "isDefault": true
      }
    }
  ]
}
```

For Linux:

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "shell",
      "label": "g++ build active file",
      "command": "/usr/bin/g++",
      "args": ["-g", "${file}", "-o", "${fileDirname}/${fileBasenameNoExtension}"],
      "options": {
        "cwd": "/usr/bin"
      },
      "problemMatcher": ["$gcc"],
      "group": {
        "kind": "build",
        "isDefault": true
      }
    }
  ]
}
```

Note: You can learn more about **tasks.json** variables in the [variables reference](#).

The **command** setting specifies the program to run; in this case that is "cl.exe" (or g++). The **args** array specifies the command-line arguments that will be passed to cl.exe (or g++). These arguments **must** be specified in the **order** expected by the compiler.

This task tells the C++ compiler (or g++) to take the **active file** (**\${file}**), compile it, and create an executable file (**/Fe:** switch in C++ compiler) in the **current directory** (**\${fileDirname}**) with the **same name** as the **active file** but with the **.exe** extension (**\${fileBasenameNoExtension}.exe**), resulting in **helloworld.exe** for our example.

The **label** value is what you will see in the **tasks list**; you can name this whatever you like.

The **problemMatcher** value selects the output parser to use for finding errors and warnings in the compiler output. For **cl.exe**, you'll get the **best** results if you use the **\$msCompile** problem matcher.

The **"isDefault": true** value in the **group** object specifies that this task will be run when you press **Ctrl+Shift+B**.

This property is for **convenience only**; if you set it to false, you can still run it from the Terminal menu with **Tasks: Run Build Task**.

(Note: \${fileDirname}/output; add the **"-lm"** in the back of "args" for the **math library** to be linked)

Supplement materials

You should refer to the Compiler options [links](#). (cl.exe)

For example:

Option	Purpose
/openmp	Enables #pragma omp in source code.
/Zi	Generates complete debugging information.
/Zs	Checks syntax only.
/Fe: <i>pathname</i>	Renames the executable file.
/Od	Disables optimization.
/O1	Creates small code.
/O2	Creates fast code.
/EH	Specifies the model of exception handling.
/EHa	Enable C++ exception handling (with SEH exceptions).
/EHc	extern "C" defaults to nothrow .
/EHr	Always generate noexcept runtime termination checks.
/EHs	Enable C++ exception handling (no SEH exceptions).
/std:c++17	C++17 standard ISO/IEC 14882:2017.
/nologo	Suppresses display of sign-on banner.
/I <i>directory</i>	Searches a directory for include files.
/Fo: <i>pathname</i>	Specifies an object (.obj) file name or directory to be used
/D	Defines constants and macros.
/MT	Compiles to create a multithreaded executable file, by using LIBCMT.lib.
/Wall	Enable all warnings, including warnings that are disabled by default.

Like (Note there is a colon (:) in the rear of **Fe**, to help identify the files' path)

```
"args": [  
  "-Zi",  
  "-EHsc",  
  "-openmp",  
  "-Fe:",  
  "${workspaceFolder}\\${workspaceFolderBasename}.exe",  
  "${workspaceFolder}\\*.c"  
],
```

For gcc:

Option	Purpose
-fopenmp	Enables #pragma omp in source code.
-g	Produce debugging information in the operating system's native format.
-I	e.g., -I dir. This means Adding the directory dir to the list of directories to be searched for header files during preprocessing.

-L	e.g., -L dir. Add directory [dir] to the list of directories to be searched for [-l].
-l	<p>e.g., -l library. Search the library named [library] when linking.</p> <p>The linker searches and processes libraries and object files in the order they are specified. Thus, [foo.o -lz bar.o] searches library [z] after file [foo.o] but before [bar.o]. If bar.o refers to functions in [z], those functions may not be loaded.</p> <p>The linker searches a standard list of directories plus any that you specify with [-L] for the library, which is actually a file named [liblibrary.a]=[a file name library surrounded with lib and .a].</p>
-o	e.g., -o file1. This means Placing the primary output in file file1.

Like

```

"args": [
  "--diagnostics-color=always",
  // "-O3", // O1/O2/O3
  "-g", // Produce debugging information in the operating system's native format
  "-Wall", // Turns on all optional warnings which are desirable for normal code
  // "-Werror", // Make all warnings into hard errors. Source code which triggers warnings will be rejected
  "-fopenmp", // Enables #pragma omp in source code
  "--std=c++17", // c++11/c++17
  /**
   * Below are header files: (can be from multiples folders)
   */
  // e.g., -I dir. This means Adding the directory dir to the list of directories to be searched for header files during preprocessing
  "${workspaceFolder}/inc/",
  "-I",
  "${workspaceFolder}/Codes/",
  // "-L", // e.g., -L dir. Add directory [dir] to the list of directories to be searched for [-l].
  // "${workspaceFolder}/vcpkg_installed/x64-linux/lib",
  /**
   * Below are source files: (can be from multiples folders)
   */
  "${workspaceFolder}/Codes/*.c",
  "${workspaceFolder}/src/*.c",
  /**
   * Below are library names:
   * e.g., -l library. Search the library named [library] when linking.
   *
   * The linker searches and processes libraries and object files in the order they are specified.
   * Thus, [foo.o -lz bar.o] searches library [z] after file [foo.o] but before [bar.o]. If bar.o refers to functions in [z], those functions may not be loaded.
   *
   * The linker searches a standard list of directories plus any that you specify with [-L] for the library, which is actually a file named [liblibrary.a]=[a file name library surrounded with lib and .a].
   */
  // "-l",
  // "-fat",
  /**
   * Below is the executable file's name
   */
  "-o", // e.g., -o file1. This means placing the primary output in file [file1].
  "${workspaceFolder}/output/entry"
],

```

Running the build

1. Go back to [helloworld.cpp](#). Your task builds the **active file** and you want to build [helloworld.cpp](#).
2. To run the build task defined in [tasks.json](#), press **Ctrl+Shift+B** or from the **Terminal** main menu choose **Tasks: Run Build Task**.
3. When the task starts, you should see the Integrated Terminal panel appear below the source code editor. After the task completes, the terminal shows output from the compiler that indicates whether the build succeeded or failed. For a successful C++ (or g++) build, the output looks something like this:

```

OUTPUT  TERMINAL  DEBUG CONSOLE  PROBLEMS
1: Task - cl.exe build act  +  [ ]  [ ]  ^  x

> Executing task: cl.exe /Zi /EHsc /Fe: c:\Users\ [redacted] \projects\helloworld\helloworld.exe c:\Users\ [redacted] \projects\helloworld\helloworld.cpp <

Microsoft (R) C/C++ Optimizing Compiler Version 19.24.28316 for x86
Copyright (C) Microsoft Corporation. All rights reserved.

helloworld.cpp
Microsoft (R) Incremental Linker Version 14.24.28316.0
Copyright (C) Microsoft Corporation. All rights reserved.

/debug
/out:c:\Users\ [redacted] \projects\helloworld\helloworld.exe
helloworld.obj

Terminal will be reused by tasks, press any key to close it.

```

cl.exe



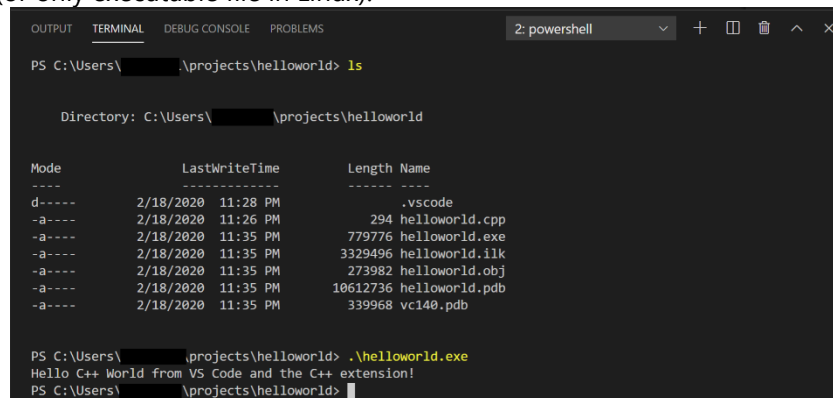
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 1: Task - g++ build acti + [icon] [icon] [icon]
> Executing task: /usr/bin/g++ -g /home/projects/helloworld/helloworld.cpp -o /home/projects/helloworld/helloworld <

Terminal will be reused by tasks, press any key to close it.
```

g++

If the build fails due to not finding **cl.exe**, or lacking an include path, make sure you have started VS Code from the **Developer Command Prompt for Visual Studio**.

1. Create a **new terminal** using the **+** button and you'll have a new terminal (running PowerShell or else) with the **helloworld** folder as the working directory. Run **ls** (or **DIR**) and you should now see the executable **helloworld.exe** along with various intermediate C++ output and debugging files (**helloworld.obj**, **helloworld.pdb**) (or only executable file in Linux).



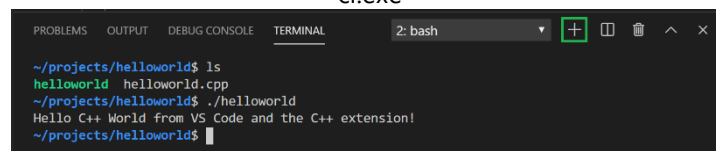
```
OUTPUT TERMINAL DEBUG CONSOLE PROBLEMS 2: powershell + [icon] [icon] [icon] x
PS C:\Users\ [redacted] \projects\helloworld> ls

Directory: C:\Users\ [redacted] \projects\helloworld

Mode                LastWriteTime         Length Name
----                -
d-----          2/18/2020 11:28 PM             .vscode
-a----          2/18/2020 11:26 PM             294 helloworld.cpp
-a----          2/18/2020 11:35 PM          779776 helloworld.exe
-a----          2/18/2020 11:35 PM        3329496 helloworld.ilc
-a----          2/18/2020 11:35 PM        273982 helloworld.obj
-a----          2/18/2020 11:35 PM        10612736 helloworld.pdb
-a----          2/18/2020 11:35 PM        339968 vc140.pdb

PS C:\Users\ [redacted] \projects\helloworld> .\helloworld.exe
Hello C++ World from VS Code and the C++ extension!
PS C:\Users\ [redacted] \projects\helloworld>
```

cl.exe



```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL 2: bash + [icon] [icon] [icon] x
~/projects/helloworld$ ls
helloworld  helloworld.cpp
~/projects/helloworld$ ./helloworld
Hello C++ World from VS Code and the C++ extension!
~/projects/helloworld$
```

g++

2. You can **run helloworld** in the terminal by typing **.\helloworld.exe**. (or **./output/helloworld.exe**)

Modifying tasks.json

You can **modify** your **tasks.json** to **build multiple** C++ files by using an argument like **"\${workspaceFolder}/**/*.cpp"** instead of **\${file}**. This will build all **.cpp** files in your current folder. You can **also modify** the output filename by replacing **"\${fileDirname}\\${fileBasenameNoExtension}.exe"** with a hard-coded filename (for example **"\${workspaceFolder}\myProgram.exe"**).

For Linux:

You can **modify** your **tasks.json** to **build multiple** C++ files by using an argument like **"\${workspaceFolder}/*.cpp"** instead of **\${file}** (**only** one main function). You can **also modify** the output filename by replacing **"\${fileDirname}/\${fileBasenameNoExtension}"** with a hard-coded filename (for example 'helloworld.out').

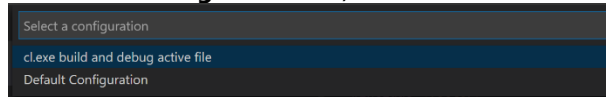
Debug helloworld.cpp

Next, you'll **create** a **launch.json** file to **configure** VS Code to launch the **Microsoft C++ debugger** when you press **F5** to **debug** the program. From the main menu, choose **Run > Add Configuration...** and then choose **C++ (Windows)**.

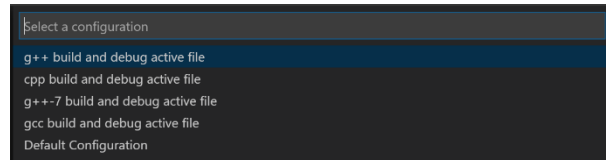
For Linux:

Next, you'll create a **launch.json** file to configure VS Code to launch the **GDB debugger** when you press **F5** to debug the program. From the main menu, choose **Run > Add Configuration...** and then choose **C++ (GDB/LLDB)**.

You'll then see a dropdown for various predefined debugging configurations. Choose **cl.exe build and debug active file** (or **g++ build and debug active file**).



cl.exe



g++

VS Code creates a **launch.json** file, opens it in the editor, and **builds** and **runs** 'helloworld'.

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "cl.exe build and debug active file",
      "type": "cppvsdbg",
      "request": "launch",
      "program": "${fileDirname}\\${fileBasenameNoExtension}.exe",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "preLaunchTask": "cl.exe build active file"
    }
  ]
}
```

The **program** setting specifies the program you **want to** debug. Here it is set to the **active file folder** **\${fileDirname}** and **active filename** with the **.exe** extension **\${fileBasenameNoExtension}.exe**, which if **helloworld.cpp** is the active file will be **helloworld.exe**.

(Note: **\${fileDirname}/output**)

For Linux:

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "g++ build and debug active file",
      "type": "cppdbg",
      "request": "launch",
      "program": "${fileDirname}/${fileBasenameNoExtension}",
      "args": [],
      "stopAtEntry": false,
      "cwd": "${workspaceFolder}",
      "environment": [],
      "externalConsole": false,
      "MIMode": "gdb",
      "setupCommands": [
        {
          "description": "Enable pretty-printing for gdb",
          "text": "-enable-pretty-printing",
          "ignoreFailures": true
        }
      ],
      "preLaunchTask": "g++ build active file",
      "miDebuggerPath": "/usr/bin/gdb"
    }
  ]
}
```

In the JSON above, **program** specifies the program you want to debug. Here it is set to the **active file folder** **\${fileDirname}** and **active filename** without an extension **\${fileBasenameNoExtension}**, which if **helloworld.cpp** is the active file will be **helloworld**.

By default, the C++ extension **won't add** any breakpoints to your source code and the **stopAtEntry** value is set to **false**.

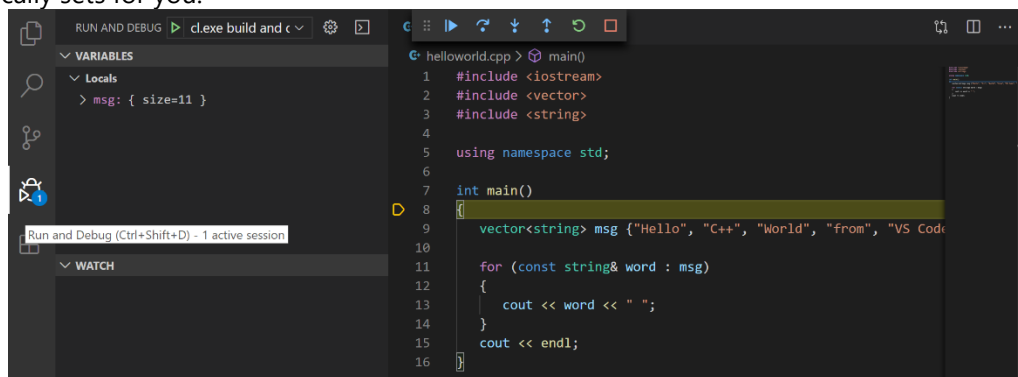
Change the **stopAtEntry** value to true to cause the debugger to **stop** on the **main** method when you start debugging.

Start a debugging session

1. Go back to **helloworld.cpp** so that it is the active file.

2. Press **F5** or from the main menu choose **Run > Start Debugging**. Before you start stepping through the source code, let's take a moment to notice several changes in the user interface:

- The Integrated Terminal appears at the bottom of the source code editor. In the **Debug Output** tab, you see output that indicates the debugger is **up** and **running**.
- The editor **highlights** the first statement in the **main** method. This is a breakpoint that the C++ extension automatically sets for you:



- The Run view on the left shows debugging information. You'll see an example later in the tutorial.
- At the top of the code editor, a debugging control panel appears. You can move this around the screen by grabbing the dots on the left side.

Step through the code

Now you're ready to start stepping through the code.

Click or press the **Step over** icon in the debugging control panel until the **for (const string& word : msg)** statement is highlighted.



1. The **Step Over** command skip over all the internal function calls within the **vector** and **string** classes that are invoked when the **msg** variable is created and initialized. Notice the change in the **Variables** window on the left. In this case, the errors are expected because, although the variable names for the loop are now visible to the debugger, the statement has not executed yet, so there is nothing to read at this point. The contents of **msg** are visible, however, because that statement has completed.

2. Press **Step over** again to advance to the next statement in this program (skipping over all the internal code that is executed to initialize the loop). Now, the **Variables** window shows information about the loop variables.

3. Press **Step over** again to execute the **cout** statement (display in the **DEBUG CONSOLE**). **Note** As of the March 2019 version of the extension, no output is displayed until the loop completes. It's not vs codes behaviour, your operating system buffers output before printing to the console, this is entirely normal and fairly universal across all platforms

4. If you like, you can keep pressing **Step over** until all the words in the vector have been printed to the console. But if you are curious, try pressing the **Step Into** button to step through source code in the C++ standard library!

```
167 template<class _Elem,  
168         class _Traits,  
169         class _Alloc> inline  
170     basic_ostream<_Elem, _Traits>& operator<<(  
171         basic_ostream<_Elem, _Traits>& _Ostr,  
172         const basic_string<_Elem, _Traits, _Alloc>& _Str)  
173     { // insert a string  
174         return (_Insert_string(_Ostr, _Str.data(), _Str.size()));  
175     }  
176  
177     // sto* NARROW CONVERSIONS  
178  
179     inline int stoi(const string& _Str, size_t * _Idx = nullptr,  
180                    int Base = 10)
```

To return to your own code, one way is to keep pressing **Step over**. Another way is to set a breakpoint in your code by switching to the **helloworld.cpp** tab in the code editor, putting the insertion point somewhere on the **cout** statement inside the loop, and pressing **F9**. A red dot appears in the gutter on the left to indicate that a breakpoint has been set on this line.

```
12     ... for (const string& word : msg)  
13     {  
14         ... cout << word << " ";  
15     }  
16 }
```

Then press **F5** to start execution from the current line in the standard library header. Execution will break on **cout**. If you like, you can press **F9** again to toggle off the breakpoint.

Set a watch

Sometimes you might want to keep track of the value of a variable as your program executes. You can do this by setting a **watch** on the variable.

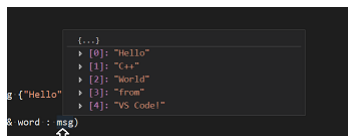
1. Place the insertion point inside the loop. In the **Watch** window, click the plus sign and in the text box, type **word**, which is the name of the loop variable. Now view the Watch window as you step through the loop.



```
18     vector<string> msg {"Hello", "C++",  
19                        "World", "VS Code!"};  
20     for (const string& word : msg)  
21     {  
22         cout << word << " ";  
23     }  
24     cout << endl;
```

2. Add another watch by adding this statement before the loop: **int i = 0**. Then, inside the loop, add this statement: **++i**. Now add a watch for **i** as you did in the previous step.

3. To quickly view the value of any variable while execution is paused on a breakpoint, you can hover over it with the mouse pointer.

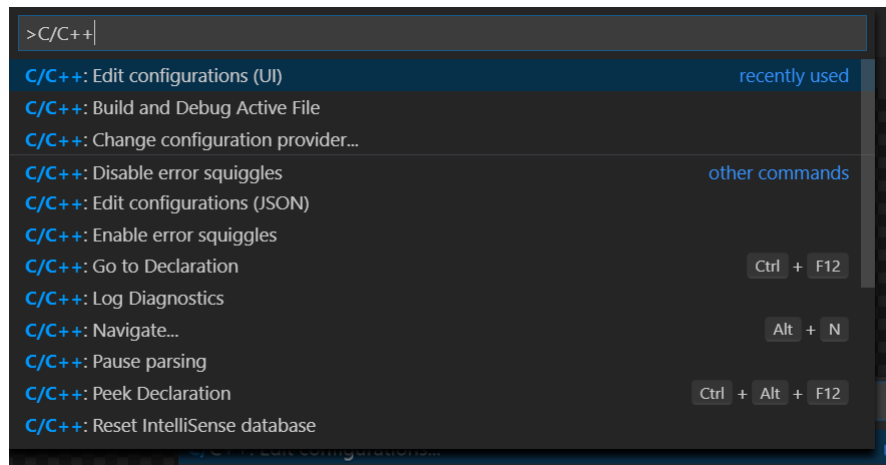


```
g ("Hello"  
% word : msg)
```

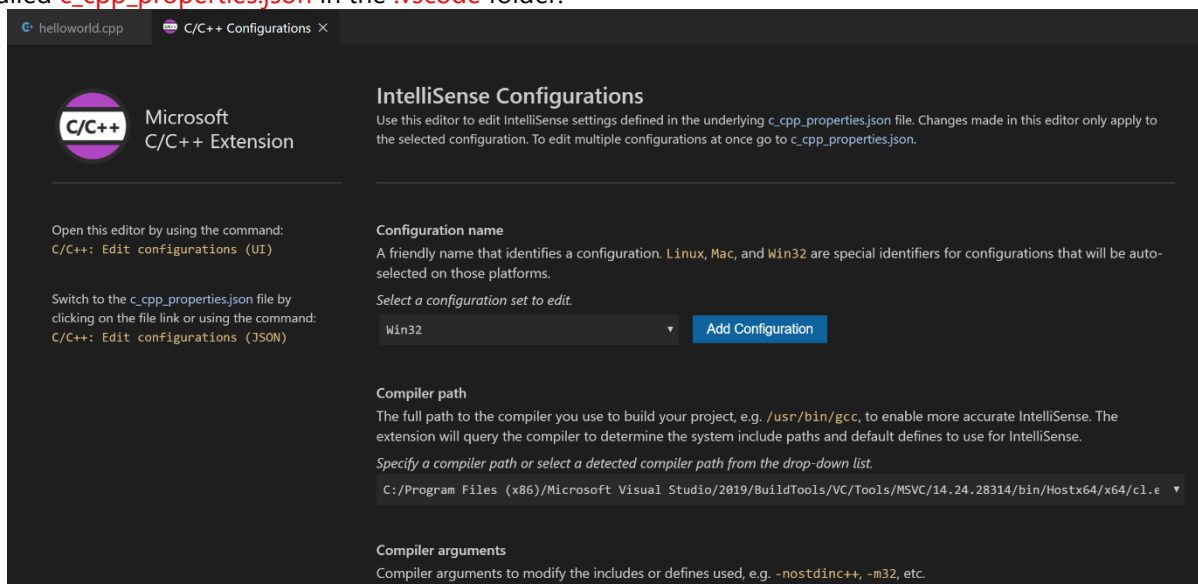
C/C++ configurations

If you want **more control** over the C/C++ extension, you can **create** a **c_cpp_properties.json** file, which will **allow** you to change settings such as the **path** to the compiler, **include paths**, C++ **standard** (default is C++17), and more.

You can view the C/C++ configuration UI by running the command **C/C++: Edit Configurations (UI)** from the Command Palette (**Ctrl+Shift+P**).



This opens the **C/C++ Configurations** page. When you make changes here, VS Code **writes** them to a file called **c_cpp_properties.json** in the **.vscode** folder.



Visual Studio Code places these settings in **.vscode\c_cpp_properties.json**. If you open that file directly, it should look something like this:

```
{
  "configurations": [
    {
      "name": "Win32",
      "includePath": ["${workspaceFolder}/**"],
      "defines": ["_DEBUG", "UNICODE", "_UNICODE"],
      "windowsSdkVersion": "10.0.18362.0",
      "compilerPath": "C:/Program Files (x86)/Microsoft Visual Studio/2019/BuildTools/VC/Tools/MSVC/14.24.28314/bin/Hostx64/x64/cl.exe",
      "cStandard": "c11",
      "cppStandard": "c++17",
      "intelliSenseMode": "msvc-x64"
    }
  ],
  "version": 4
}
```

cl.exe

```
{
  "configurations": [
    {
      "name": "Linux",
      "includePath": ["${workspaceFolder}/**"],
      "defines": [],
      "compilerPath": "/usr/bin/gcc",
      "cStandard": "c11",
      "cppStandard": "c++17",
      "intelliSenseMode": "clang-x64"
    }
  ],
  "version": 4
}
```

g++

You **only** need to add to the **Include path** array setting if your program includes header files that are **not** in your workspace **or** in the standard library path.

1. Editing features

The C/C++ extension for VS Code has many features that help you write code, understand it, and navigate around in your source files. To provide the best experience, the extension needs to know where it can find each header file referenced in your code. By default, the extension searches the current source directory, its sub-directories, and some platform-specific locations. If a referenced header file can't be found, VS Code displays a green squiggle underneath each `#include` directive that references it.

To specify additional include directories to be searched, place your cursor over any `#include` directive that displays a green squiggle, then click the lightbulb action when it appears. This opens the file `c_cpp_properties.json` for editing; here you can specify additional include directories for each platform configuration individually by adding more directories to the 'browse.path' property.

2. Code formatting

The C/C++ extension for Visual Studio Code supports source code formatting using clang-format which is included with the extension. Note you can also get the latest Clang-Format extension from the [Visual Studio Marketplace](#).

You can format an entire file with **Format Document** (**Shift+Alt+F**) or just the current selection with **Format Selection** (**Ctrl+K Ctrl+F**) in right-click context menu. You can also configure auto-formatting with the following [settings](#):

- `editor.formatOnSave` - to format when you save your file.
- `editor.formatOnType` - to format as you type (triggered on the `;` character).

By default, the clang-format style is set to "file" which means it looks for a `.clang-format` file inside your workspace. If the `.clang-format` file is found, formatting is applied according to the settings specified in the file. If no `.clang-format` file is found in your workspace, formatting is applied based on a default style specified in the `C_Cpp.clang_format_fallbackStyle` [setting](#) instead. Currently, the default formatting style is "Visual Studio" which is an approximation of the default code formatter in Visual Studio.

To use a different version of clang-format than the one that ships with the extension, change the `C_Cpp.clang_format_path` [setting](#) to the path where the clang-format binary is installed.

For example, on the Windows platform:

```
"C_Cpp.clang_format_path": "C:\\Program Files (x86)\\LLVM\\bin\\clang-format.exe"
```

Compiler path

The `compilerPath` setting is an important setting in your configuration. The extension uses it to infer the path to the C++ standard library header files. When the extension knows where to find those files, it can provide useful features like smart completions and **Go to Definition** navigation.

The C/C++ extension attempts to populate `compilerPath` with the default compiler location based on what it finds on your system. The extension looks in several common compiler locations.

The `compilerPath` search order is:

- First check for the Microsoft Visual C++ compiler
- Then look for g++ on Windows Subsystem for Linux (WSL)
- Then g++ for Mingw-w64.

If you have g++ or WSL installed, you **might** need to change `compilerPath` to match the preferred compiler for your project. For Microsoft C++, the path should look something like this, depending on which specific version you have installed: "C:/Program Files (x86)/Microsoft Visual Studio/2017/BuildTools/VC/Tools/MSVC/14.16.27023/bin/Hostx64/x64/cl.exe".

Reusing your C++ configuration

VS Code is now configured to use the Microsoft C++ compiler (or gcc on Linux). The configuration applies to the current workspace. To **reuse the configuration**, just `copy` the JSON files to a `.vscode` folder in a new project folder (workspace) and change the names of the source file(s) and executable as needed.

Run VS Code outside the Developer Command Prompt

In certain circumstances, it isn't possible to run VS Code from **Developer Command Prompt for Visual Studio** (for example, in Remote Development through SSH scenarios). In that case, you can automate initialization of **Developer Command Prompt for Visual Studio** during the build using the following `tasks.json` configuration:


```
{
  "version": "2.0.0",
  "windows": {
    "options": {
      "shell": {
        "executable": "cmd.exe",
        "args": [
          "/C",
          // The path to VsDevCmd.bat depends on the version of Visual Studio you have installed.
          "\"C:/Program Files (x86)/Microsoft Visual Studio/2019/Community/Common7/Tools/VsDevCmd.bat\"",
          "&&"
        ]
      }
    }
  },
  "tasks": [
    {
      "type": "shell",
      "label": "cl.exe build active file",
      "command": "cl.exe",
      "args": [
        "/Zi",
        "/EHsc",
        "/Fe:",
        "${fileDirname}\\${fileBasenameNoExtension}.exe",
        "${file}"
      ],
      "problemMatcher": ["$msCompile"],
      "group": {
        "kind": "build",
        "isDefault": true
      }
    }
  ]
}
```

Note: The path to **VsDevCmd.bat** might be different depending on the **Visual Studio version** or **installation path** (like in D:). You can find the path to **VsDevCmd.bat** by opening a Command Prompt and running `dir "\\VsDevCmd*" /s`.

Troubleshooting

The term 'cl.exe' is not recognized

If you see the error "The term 'cl.exe' is not recognized as the name of a cmdlet, function, script file, or operable program.", this usually means you are running VS Code **outside** of a **Developer Command Prompt for Visual Studio** and VS Code doesn't know the path to the **cl.exe** compiler.

You **can always check** that you are running VS Code in the context of the Developer Command Prompt by opening a new Terminal (**Ctrl+Shift+**) and typing 'cl' to verify **cl.exe** is available to VS Code.

The most common cause of errors (such as **undefined _main**, or **attempting to link with file built for unknown-unsupported file format**, and so on) occurs when **helloworld.cpp** is **not** the **active file** when you start a build or start debugging. This is because the compiler is trying to compile something that isn't source code, like your **launch.json**, **tasks.json**, or **c_cpp_properties.json** file.

Variables Reference

Visual Studio Code supports variable substitution in **Debugging** and **Task** configuration files as well as some select settings. Variable substitution is supported inside some key and value strings in **launch.json** and **tasks.json** files using **\${variableName}** syntax.

Predefined variables

The following predefined variables are supported:

- **\${workspaceFolder}** - the path of the folder opened in VS Code
- **\${workspaceFolderBasename}** - the name of the folder opened in VS Code without any slashes (/)
- **\${file}** - the **current** opened file

- **`${fileWorkspaceFolder}`** - the current opened file's workspace folder
- **`${relativeFile}`** - the current opened file relative to **`workspaceFolder`**
- **`${relativeFileDirname}`** - the current opened file's dirname relative to **`workspaceFolder`**
- **`${fileBasename}`** - the current opened file's basename
- **`${fileBasenameNoExtension}`** - the current opened file's basename with no file extension
- **`${fileDirname}`** - the current opened file's dirname
- **`${fileExtname}`** - the current opened file's extension
- **`${cwd}`** - the task runner's current working directory on startup
- **`${lineNumber}`** - the current selected line number in the active file
- **`${selectedText}`** - the current selected text in the active file
- **`${execPath}`** - the path to the running VS Code executable
- **`${defaultBuildTask}`** - the name of the default build task
- **`${pathSeparator}`** - the character used by the operating system to separate components in file paths

Predefined variables examples

Supposing that you have the following requirements:

1. A file located at **`/home/your-username/your-project/folder/file.ext`** opened in your editor;
2. The directory **`/home/your-username/your-project`** opened as your root workspace.

So you will have the following values for each variable:

- **`${workspaceFolder}`** - **`/home/your-username/your-project`**
- **`${workspaceFolderBasename}`** - **`your-project`**
- **`${file}`** - **`/home/your-username/your-project/folder/file.ext`**
- **`${fileWorkspaceFolder}`** - **`/home/your-username/your-project`**
- **`${relativeFile}`** - **`folder/file.ext`**
- **`${relativeFileDirname}`** - **`folder`**
- **`${fileBasename}`** - **`file.ext`**
- **`${fileBasenameNoExtension}`** - **`file`**
- **`${fileDirname}`** - **`/home/your-username/your-project/folder`**
- **`${fileExtname}`** - **`.ext`**
- **`${lineNumber}`** - line number of the cursor
- **`${selectedText}`** - text selected in your code editor
- **`${execPath}`** - location of Code.exe
- **`${pathSeparator}`** - **`/`** on macOS or linux, **`\\`** on Windows

Tip: Use IntelliSense inside string values for **`tasks.json`** and **`launch.json`** to get a full list of predefined variables.

Common questions

How can I know a variable's actual value?

One easy way to check a variable's runtime value is to create a VS Code **task** to output the variable value to the console. For example, to see the resolved value for **`${workspaceFolder}`**, you can create and run (**Terminal > Run Task**) the following simple 'echo' task in **`tasks.json`**: (**Note** we need to **Terminal > Configure Tasks** to let Visual Studio Code know this new task)

```
{
  "version": "2.0.0",
  "tasks": [
    {
      "label": "echo",
      "type": "shell",
      "command": "echo ${workspaceFolder}"
    }
  ]
}
```

Getting Started with Python in VS Code

In this tutorial, you use **Python 3** to create the simplest Python "Hello World" application in Visual Studio Code. By using the Python extension, you make VS Code into a great lightweight Python IDE (which you may find a productive alternative to PyCharm).

This tutorial introduces you to VS Code as a Python environment, primarily how to edit, run, and debug code through the following tasks:

- Write, run, and debug a Python "Hello World" Application
- Learn how to install packages by creating Python virtual environments
- Write a simple Python script to plot figures within VS Code

This tutorial is not intended to teach you Python itself. Once you are familiar with the basics of VS Code, you can then follow any of the [programming tutorials on python.org](https://python.org/programming-tutorials/) within the context of VS Code for an introduction to the language.

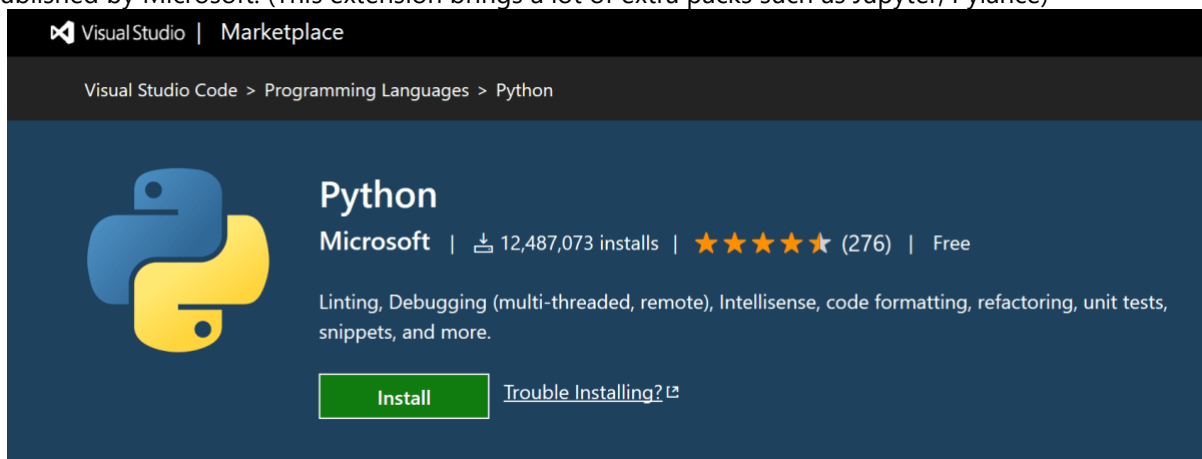
Prerequisites

To successfully complete this tutorial, you need to first setup your Python development environment. Specifically, this tutorial requires:

- VS Code
- VS Code Python extension
- Python 3

Install Visual Studio Code and the Python Extension

1. If you have not already done so, install [VS Code](https://code.visualstudio.com/).
2. Next, install the [Python extension for VS Code](https://marketplace.visualstudio.com/items?itemName=ms-python.python) from the Visual Studio Marketplace. For additional details on installing extensions, see [Extension Marketplace](https://code.visualstudio.com/docs/editor/extension-marketplace). The Python extension is named Python and it's published by Microsoft. (This extension brings a lot of extra packs such as Jupyter, Pylance)



Install a Python interpreter

Along with the Python extension, you need to install a Python interpreter. Which interpreter you use is dependent on your specific needs, but some guidance is provided below.

Windows

Install [Python from python.org](https://python.org/). You can typically use the Download Python button that appears first on the page to download the latest version.

Note: If you don't have admin access, an additional option for installing Python on Windows is to use the Microsoft Store. The Microsoft Store provides installs of Python 3.7, Python 3.8, Python 3.9, and Python 3.10. Be aware that you might have compatibility issues with some packages using this method.

For additional information about using Python on Windows, see [Using Python on Windows at Python.org](#).

macOS

The system install of Python on macOS is not supported. Instead, an installation through [Homebrew](#) is recommended. To install Python using Homebrew on macOS use `brew install python3` at the Terminal prompt.

Note On macOS, make sure the location of your VS Code installation is included in your PATH environment variable. See these setup instructions for more information.

Linux

The built-in Python 3 installation on Linux works well, but to install other Python packages you must install pip with [get-pip.py](#).

Other options

- **Data Science:** If your primary purpose for using Python is Data Science, then you might consider a download from [Anaconda](#). Anaconda provides not just a Python interpreter, but many useful libraries and tools for data science.
- **Windows Subsystem for Linux:** If you are working on Windows and want a Linux environment for working with Python, the [Windows Subsystem for Linux](#) (WSL) is an option for you. If you choose this option, you'll also want to install the [Remote - WSL extension](#). For more information about using WSL with VS Code, see [VS Code Remote Development](#) or try the [Working in WSL tutorial](#), which will walk you through setting up WSL, installing Python, and creating a Hello World application running in WSL.

Verify the Python installation

To verify that you've installed Python successfully on your machine, run one of the following commands (depending on your operating system):

- Linux/macOS: open a Terminal Window and type the following command:

```
python3 --version
```

- Windows: open a command prompt and run the following command: (if you installed **py launcher** in C:\Windows)

```
py -3 --version
```

Open a command prompt and run the following command: (directly open Python with the version you want, here it is Python 3.10)

```
"C:\Program Files\Python310\python" or "C:\Program Files\Python310\python" -V or & "C:/Program Files/Python310/python.exe" -V
```

 (**Note** when using **PowerShell** and the path using `'`', you need add `&` at head)

If the installation was successful, the output window should show the version of Python that you installed.

Note You can use the `py -0` command in the VS Code integrated terminal to view the versions of python installed on your machine. The default interpreter is identified by an asterisk (*).

Start VS Code in a project (workspace) folder

Using a command prompt or terminal, create an empty folder called "hello", navigate into it, and open VS Code (code) in that folder (.) by entering the following commands:

```
mkdir hello
cd hello
code .
```

Note: If you're using an Anaconda distribution, be sure to use an Anaconda command prompt.

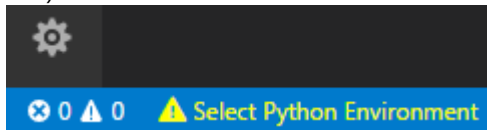
By starting VS Code in a folder, that folder becomes your "workspace". VS Code stores settings that are specific to that workspace in `.vscode/settings.json`, which are separate from user settings that are stored globally.

Alternately, you can run VS Code through the operating system UI, then use **File > Open Folder** to open the project folder.

Select a Python interpreter

Python is an *interpreted language*, and in order to run Python code and get Python IntelliSense, you must tell VS Code which interpreter to use.

From within VS Code, select a Python 3 interpreter by opening the **Command Palette** (Ctrl+Shift+P), start typing the **Python: Select Interpreter** command to search, then select the command. You can also use the **Select Python Environment** option on the Status Bar if available (it may already show a selected interpreter, too):



The command presents a list of available interpreters that VS Code can find automatically, including virtual environments. If you don't see the desired interpreter, see [Configuring Python environments](#).

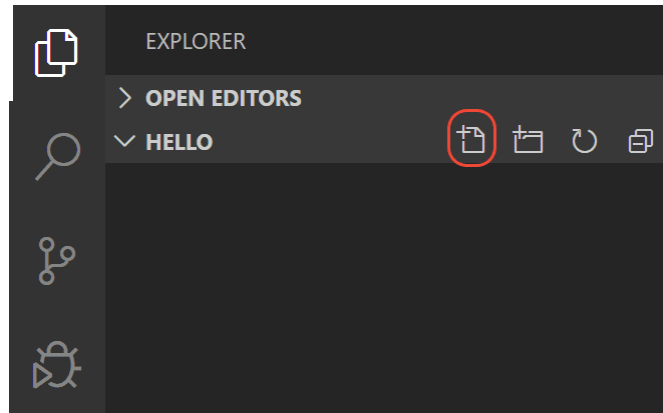
Note: When using an Anaconda distribution, the correct interpreter should have the suffix `('base':conda)`, for example `Python 3.7.3 64-bit ('base':conda)`.

Selecting an interpreter sets which interpreter will be used by the Python extension for that workspace.

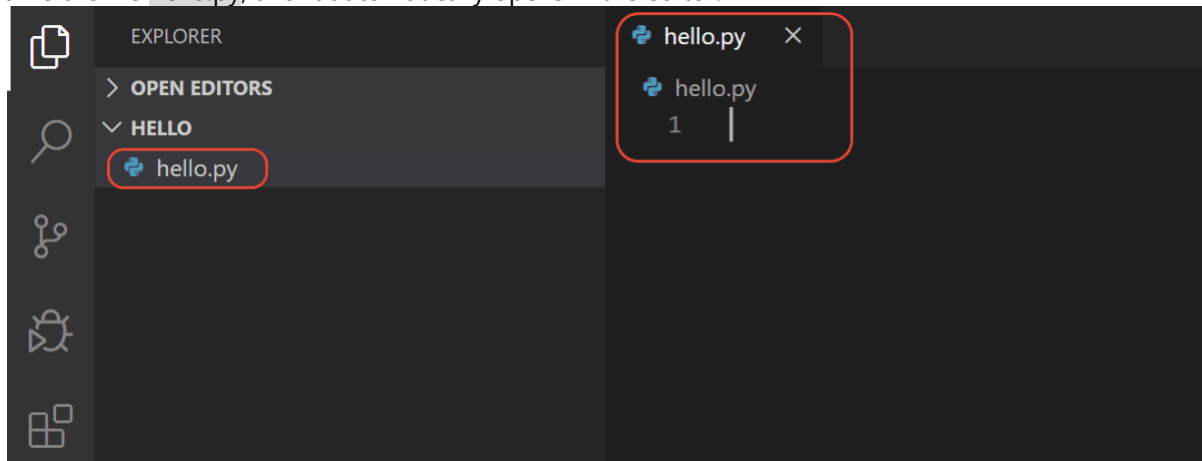
Note: If you select an interpreter without a workspace folder open, VS Code sets `python.defaultInterpreterPath` in User scope instead, which sets the default interpreter for VS Code in general. The user setting makes sure you always have a default interpreter for Python projects. The workspace settings lets you override the user setting.

Create a Python Hello World source code file

From the File Explorer toolbar, select the **New File** button on the `hello` folder:



Name the file `hello.py`, and it automatically opens in the editor:



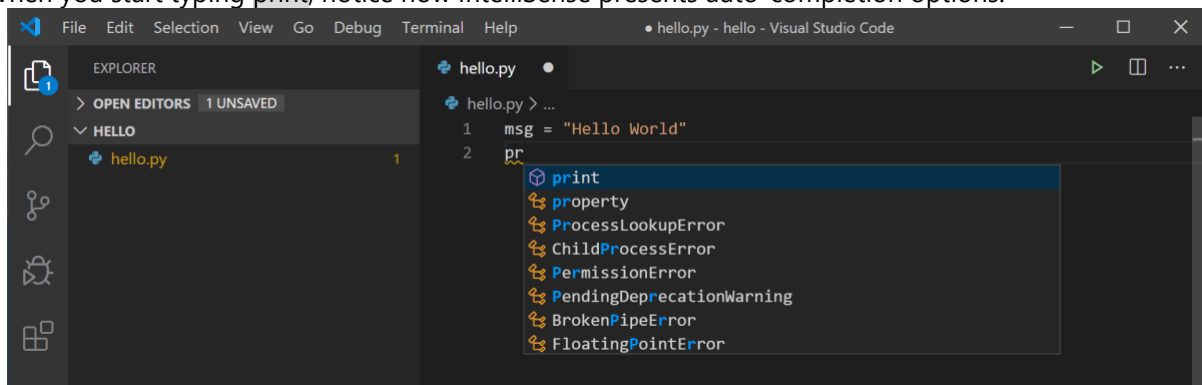
By using the `.py` file extension, you tell VS Code to interpret this file as a Python program, so that it evaluates the contents with the Python extension and the selected interpreter.

Note: The File Explorer toolbar also allows you to create folders within your workspace to better organize your code. You can use the **New folder** button to quickly create a folder.

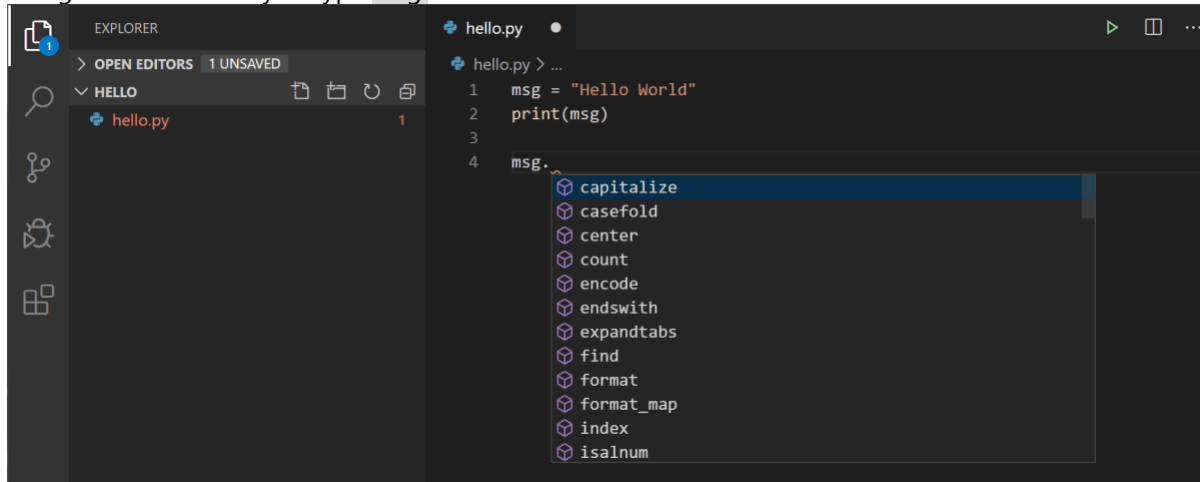
Now that you have a code file in your Workspace, enter the following source code in `hello.py`:

```
msg = "Hello World"
print(msg)
```

When you start typing `print`, notice how IntelliSense presents auto-completion options.



IntelliSense and auto-completions work for standard Python modules as well as other packages you've installed into the environment of the selected Python interpreter. It also provides completions for methods available on object types. For example, because the `msg` variable contains a string, IntelliSense provides string methods when you type `msg.`:



Feel free to experiment with IntelliSense some more, but then revert your changes so you have only the `msg` variable and the `print` call, and save the file (`Ctrl+S`).

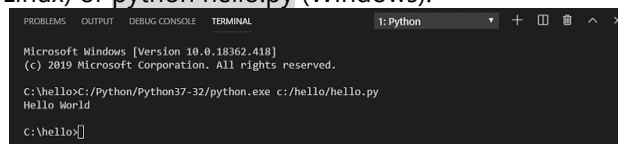
For full details on editing, formatting, and refactoring, see [Editing code](#). The Python extension also has full support for [Linting](#).

Run Hello World

It's simple to run `hello.py` with Python. Just click the `Run Python File in Terminal` play button in the top-right side of the editor.

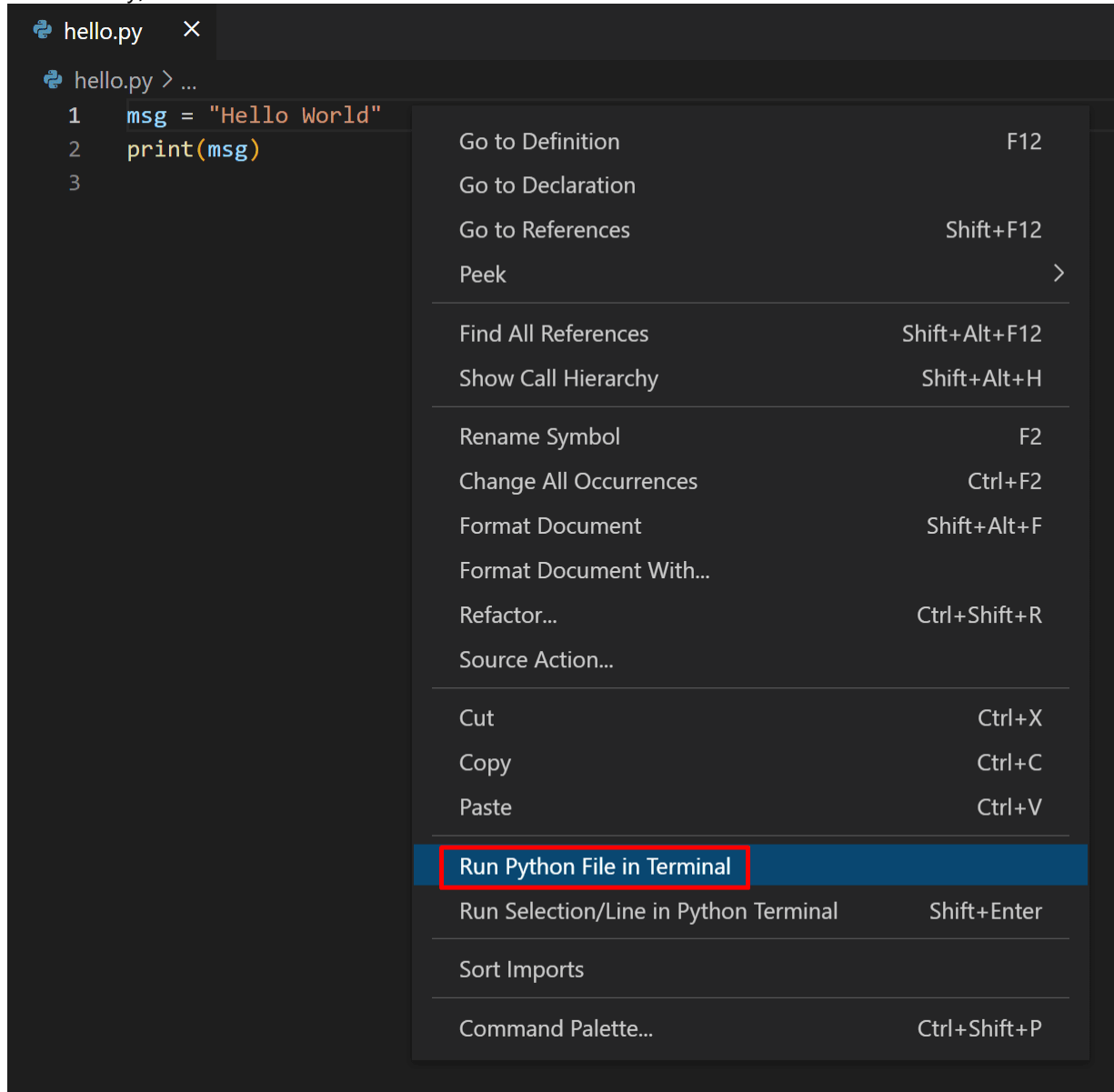


The button opens a terminal panel in which your Python interpreter is automatically activated, then runs `python3 hello.py` (macOS/Linux) or `python hello.py` (Windows):



There are three other ways you can run Python code within VS Code:

- Right-click anywhere in the editor window and select **Run Python File in Terminal** (which saves the file automatically):



- Select one or more lines, then press **Shift+Enter** or right-click and select **Run Selection/Line in Python Terminal**. This command is convenient for testing just a part of a file.
- From the Command Palette (**Ctrl+Shift+P**), select the **Python: Start REPL** command to open a REPL terminal for the currently selected Python interpreter. In the REPL, you can then enter and run lines of code one at a time.

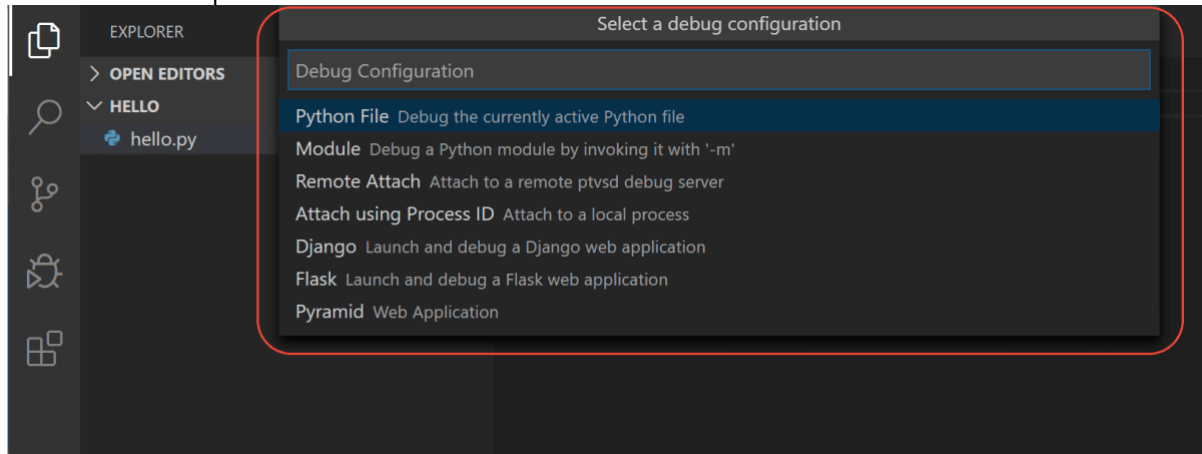
Configure and run the debugger

Let's now try debugging our simple Hello World program.

First, set a breakpoint on line 2 of `hello.py` by placing the cursor on the `print` call and pressing **F9**. Alternately, just click in the editor's left gutter, next to the line numbers. When you set a breakpoint, a red circle appears in the gutter.


```
hello.py x
hello.py > ...
1 msg = "Hello World"
2 print(msg)
```

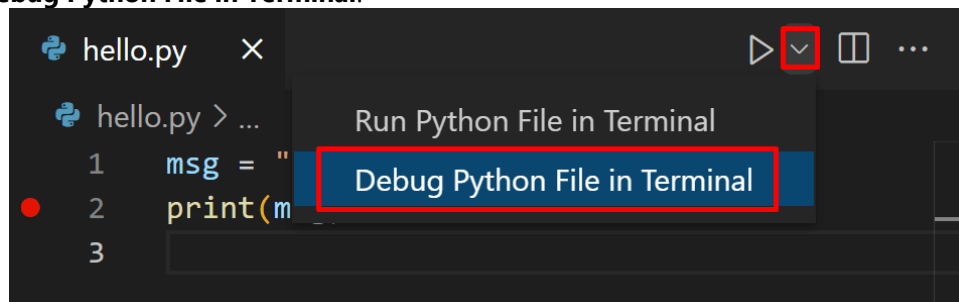
Next, to initialize the debugger, press **F5**. Since this is your first time debugging this file, a configuration menu will open from the Command Palette allowing you to select the type of debug configuration you would like for the opened file.



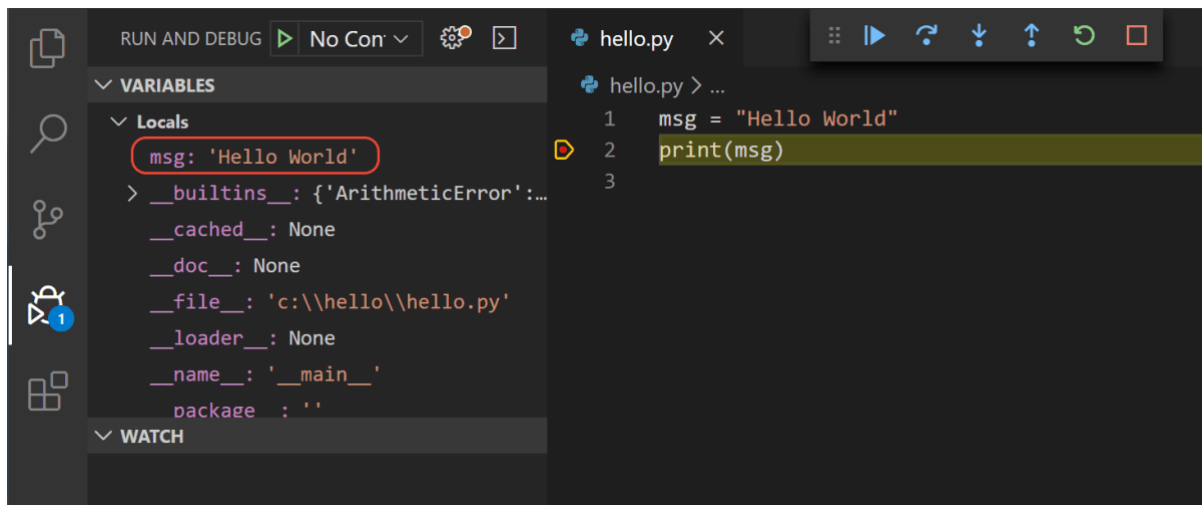
Note: VS Code uses JSON files for all of its various configurations; `launch.json` is the standard name for a file containing debugging configurations.

These different configurations are fully explained in **Debugging configurations**; for now, just select **Python File**, which is the configuration that runs the current file shown in the editor using the currently selected Python interpreter.

You can also start the debugger by clicking on the down-arrow next to the run button on the editor, and selecting **Debug Python File in Terminal**.



The debugger will stop at the first line of the file breakpoint. The current line is indicated with a yellow arrow in the left margin. If you examine the **Local** variables window at this point, you will see now defined `msg` variable appears in the **Local** pane.



A debug toolbar appears along the top with the following commands from left to right: continue (F5), step over (F10), step into (F11), step out (Shift+F11), restart (Ctrl+Shift+F5), and stop (Shift+F5).



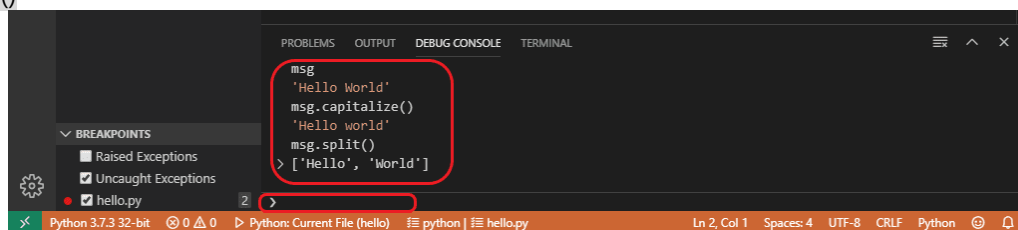
The Status Bar also changes color (orange in many themes) to indicate that you're in debug mode. The **Python Debug Console** also appears automatically in the lower right panel to show the commands being run, along with the program output.

To continue running the program, select the continue command on the debug toolbar (F5). The debugger runs the program to the end.

Tip Debugging information can also be seen by hovering over code, such as variables. In the case of `msg`, hovering over the variable will display the string `Hello world` in a box above the variable.

You can **also work** with variables in the **Debug Console** (If you don't see it, select **Debug Console** in the lower right area of VS Code, or select it from the ... menu.) Then try entering the following lines, one by one, at the > prompt at the bottom of the console:

```
msg
msg.capitalize()
msg.split()
```



Select the blue **Continue** button on the toolbar again (or press F5) to run the program to completion. "Hello World" appears in the **Python Debug Console** if you switch back to it, and VS Code exits debugging mode once the program is complete.

If you restart the debugger, the debugger again stops on the first breakpoint.

To stop running a program before it's complete, use the red square stop button on the debug toolbar (Shift+F5), or use the **Run > Stop debugging** menu command.

For full details, see **Debugging configurations**, which includes notes on how to use a specific Python interpreter for debugging.

Tip: Use Logpoints instead of print statements: Developers often litter source code with `print` statements to quickly inspect variables without necessarily stepping through each line of code in a debugger. In VS Code, you can instead use **Logpoints**. A Logpoint is like a breakpoint except that it logs a message to the console and doesn't stop the program. For more information, see [Logpoints](#) in the main VS Code debugging article.

Install and use packages

Let's now run an example that's a little more interesting. In Python, packages are how you obtain any number of useful code libraries, typically from [PyPI](#). For this example, you use the `matplotlib` and `numpy` packages to create a graphical plot as is commonly done with data science. (Note that `matplotlib` cannot show graphs when running in the Windows Subsystem for Linux as it lacks the necessary UI support.)

Return to the **Explorer** view (the top-most icon on the left side, which shows files), create a new file called `standardplot.py`, and paste in the following source code:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 20, 100) # Create a list of evenly-spaced numbers over the range
plt.plot(x, np.sin(x))      # Plot the sine of each x point
plt.show()                  # Display the plot
```

Tip: If you enter the above code by hand, you may find that auto-completions change the names after the `as` keywords when you press **Enter** at the end of a line. To avoid this, type a space, then **Enter**.

Next, try running the file in the debugger using the "Python: Current file" configuration as described in the last section.

Unless you're using an Anaconda distribution or have previously installed the `matplotlib` package, you should see the message, "**ModuleNotFoundError: No module named 'matplotlib'**". Such a message indicates that the required package isn't available in your system.

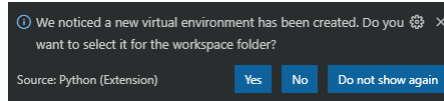
To install the `matplotlib` package (which also installs `numpy` as a dependency), stop the debugger and use the Command Palette to run **Terminal: Create New Terminal** (**Ctrl+Shift+`**). This command opens a command prompt for your selected interpreter.

A best practice among Python developers is to avoid installing packages into a global interpreter environment. You instead use a project-specific `virtual environment` that contains a copy of a global interpreter. Once you activate that environment, any packages you then install are isolated from other environments. Such isolation **reduces many complications** that can arise from conflicting package versions. To create a **virtual environment** and install the required packages, enter the following commands as appropriate for your operating system:

Note: For additional information about virtual environments, see **Environments**.

1. Create and activate the virtual environment

Note: When you create a new virtual environment, you should be prompted by VS Code to set it as the default for your workspace folder. If selected, the environment will automatically be activated when you open a new terminal.



For Windows

```
py -3 -m venv .venv
.venv\scripts\activate
```

& "C:/Program Files/Python310/python.exe" -m venv .venv (**Note** when using **PowerShell** and the path using `"`, you need add `'` at head)
.venv\scripts\activate

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):

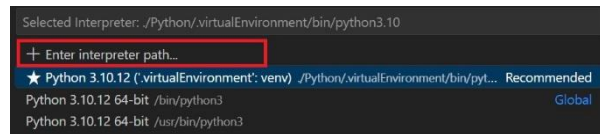
```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process
```

Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope Process

For macOS/Linux

```
python3 -m venv .venv
source .venv/bin/activate
```

2. Select your new environment by using the **Python: Select Interpreter** command from the **Command Palette**.



3. Install the packages

```
# Don't use with Anaconda distributions because they include matplotlib already.

# macOS
python3 -m pip install matplotlib

# Windows (may require elevation)
python -m pip install matplotlib

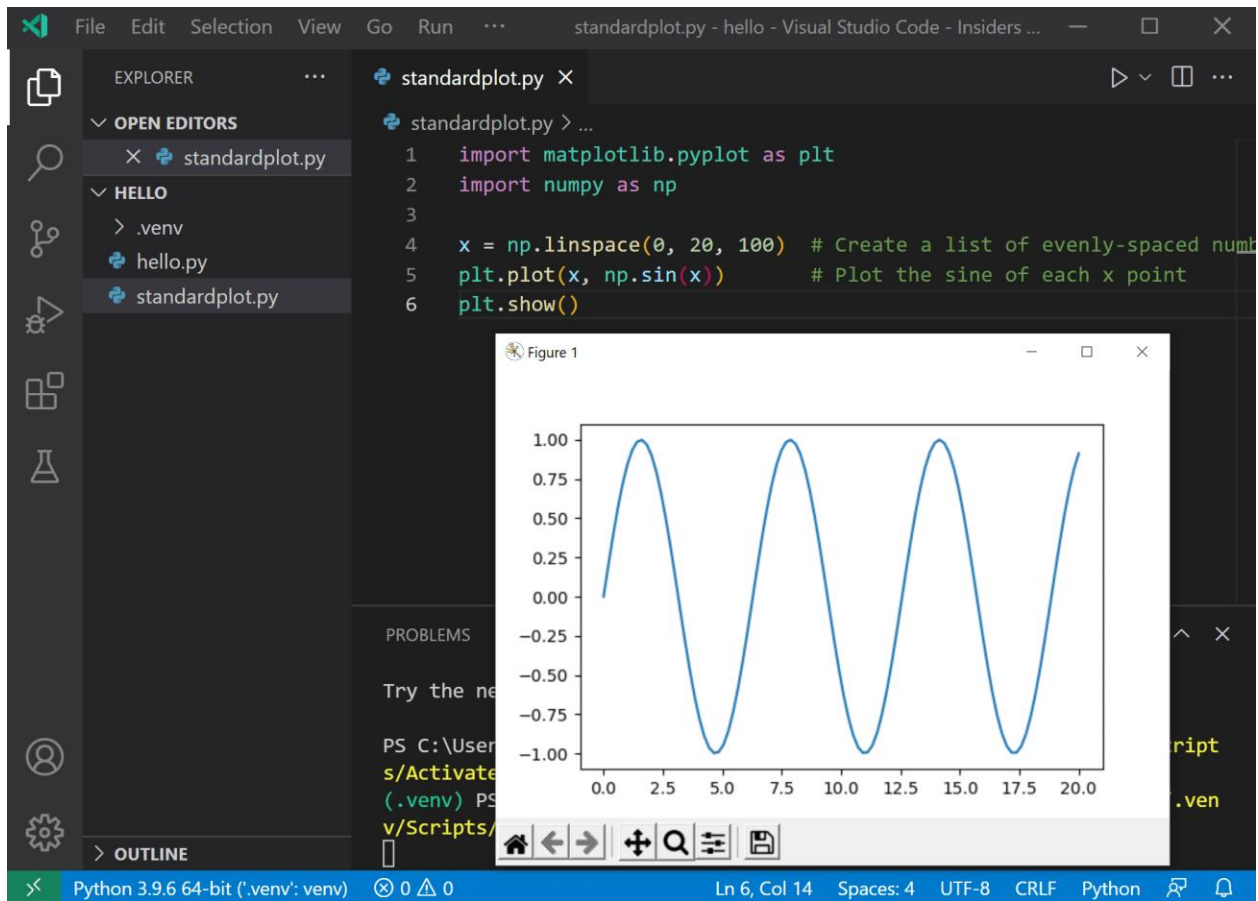
# Linux (Debian)
apt-get install python3-tk
python3 -m pip install matplotlib
```

Note: (the detail underneath)

c:/Users/youse/Files/Codes/python/helloworld/.venv/Scripts/python.exe -m pip list

c:/Users/youse/Files/Codes/python/helloworld/.venv/Scripts/python.exe -m pip install matplotlib

4. Rerun the program now (with or without the debugger) and after a few moments a plot window appears with the output:



5. Once you are finished, type `deactivate` in the terminal window to deactivate the virtual environment.

For additional examples of creating and activating a virtual environment and installing packages, see the **Django tutorial** and the **Flask tutorial**.

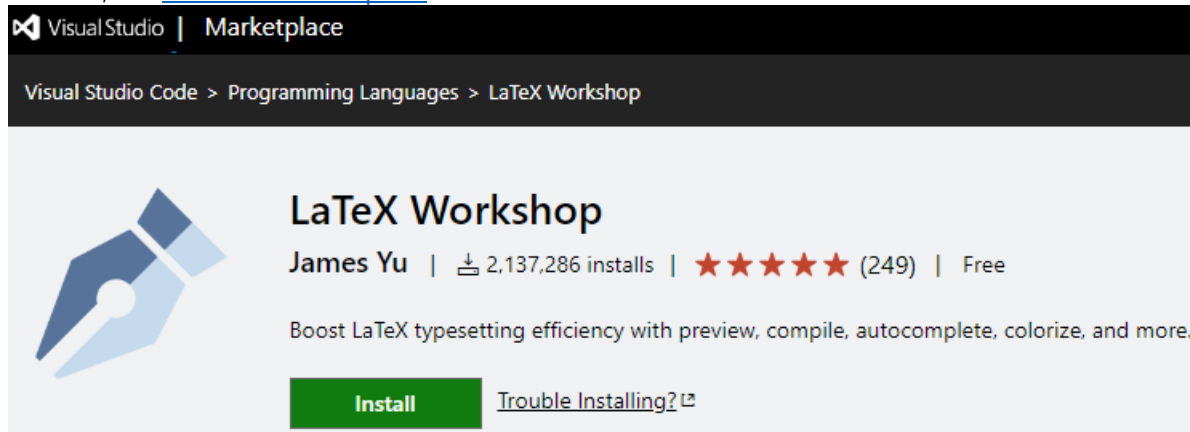
Getting Started with LaTeX-Workshop in VS Code

In this tutorial, you use **LaTeX-Workshop** to Boost LaTeX typesetting efficiency with preview, compile, autocomplete, colorize, and more.

Install Visual Studio Code and the LaTeX-Workshop

3. If you have not already done so, install [VS Code](#).

4. Next, install the LaTeX-Workshop from the Visual Studio Marketplace. For additional details on installing extensions, see [Extension Marketplace](#).



Reference the Manual of LaTeX-Workshop

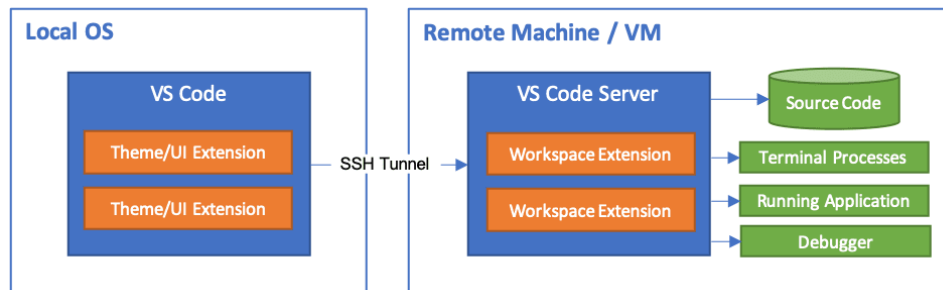
The manual of the extension is maintained as a [wiki](#).

latexmk only supports the paths in English.

Remote Development using SSH

The **Visual Studio Code Remote - SSH** extension allows you to open a remote folder on any remote machine, virtual machine, or container with a running SSH server and take full advantage of VS Code's feature set. Once connected to a server, you can interact with files and folders anywhere on the remote filesystem.

No source code needs to be on your local machine to gain these benefits since the extension runs commands and other extensions directly on the remote machine.



This lets VS Code provide a **local-quality development experience** - including full IntelliSense (completions), code navigation, and debugging - **regardless of where your code is hosted**.

Getting started

Note: After reviewing this topic, you can get started with the introductory [SSH tutorial](#).

System requirements

Local: A supported [OpenSSH compatible SSH client](#) must also be installed.

Installing a supported SSH client

OS	Instructions
Windows 10 1803+ / Server 2016/2019 1803+	Install the Windows OpenSSH Client .
Earlier Windows	Install Git for Windows .
macOS	Comes pre-installed.
Debian/Ubuntu	Run <code>sudo apt-get install openssh-client</code>
RHEL / Fedora / CentOS	Run <code>sudo yum install openssh-clients</code>

VS Code will look for the `ssh` command in the PATH. Failing that, on Windows it will attempt to find `ssh.exe` in the default Git for Windows install path. You can also specifically tell VS Code where to find the SSH client by adding the `remote.SSH.path` property to `settings.json`.

Remote SSH host: A running [SSH server](#) on:

- x86_64 Debian 8+, Ubuntu 16.04+, CentOS / RHEL 7+.
- ARMv7l (AArch32) Raspberry Pi OS (previously called Raspbian) Stretch/9+ (32-bit).
- ARMv8l (AArch64) Ubuntu 18.04+ (64-bit).
- Windows 10 / Server 2016/2019 (1803+) using the [official OpenSSH Server](#).
- macOS 10.14+ (Mojave) SSH hosts with [Remote Login enabled](#).
- 1 GB RAM is required for remote hosts, but at least 2 GB RAM and a 2-core CPU is recommended.

Installing a supported SSH server

OS	Instructions	Details
----	--------------	---------

Debian 8+ / Ubuntu 16.04+	Run <code>sudo apt-get install openssh-server</code>	See the Ubuntu SSH documentation for details.
RHEL / CentOS 7+	Run <code>sudo yum install openssh-server</code> && <code>sudo systemctl start sshd.service</code> && <code>sudo systemctl enable sshd.service</code>	See the RedHat SSH documentation for details.
SuSE 12+ / openSUSE 42.3+	In Yast, go to Services Manager, select "sshd" in the list, and click Enable . Next go to Firewall, select the Permanent configuration, and under services check sshd .	See the SuSE SSH documentation for details.
Windows 10 1803+ / Server 2016/2019 1803+	Install the Windows OpenSSH Server .	
macOS 10.14+ (Mojave)	Enable Remote Login .	

Installation

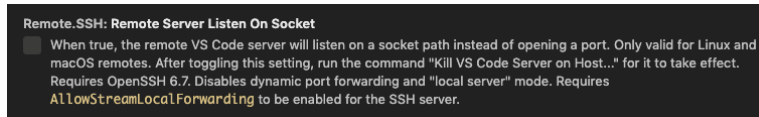
To get started, you need to:

1. Install an [OpenSSH compatible SSH client](#) if one is not already present.
2. Install Visual Studio Code or Visual Studio Code Insiders.
3. Install the [Remote-SSH extension](#). If you plan to work with other remote extensions in VS Code, you may choose to install the [Remote Development extension pack](#).

SSH host setup

1. If you do not have an SSH host set up, follow the directions for [Linux](#), [Windows 10 / Server \(1803+\)](#), or [macOS](#) SSH host or create a [VM on Azure](#).
2. **Optional:** If your Linux or macOS SSH host will be accessed by multiple users at the same time, consider enabling **Remote.SSH: Remote Server Listen On Socket** in VS Code [User settings](#) for improved security.

In the Settings editor:



See the [Tips and Tricks](#) article for details.

3. **Optional:** While password-based authentication is supported, we recommend setting up **key based authentication** for your host. See the [Tips and Tricks](#) article for details.

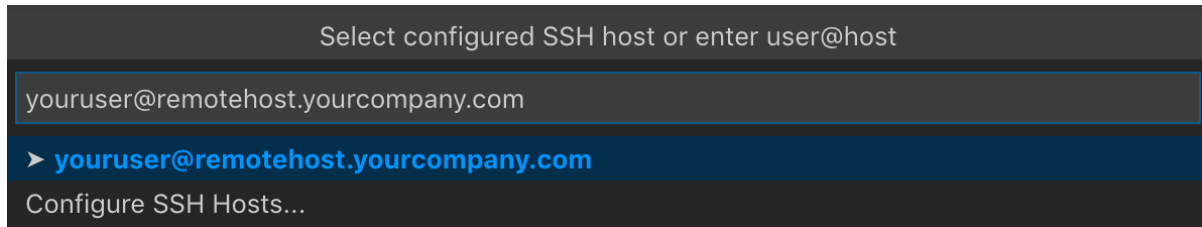
Connect to a remote host

To connect to a remote host for the first time, follow these steps:

1. Verify you can connect to the SSH host by running the following command from a terminal / PowerShell window replacing `user@hostname` as appropriate.

```
ssh user@hostname
# Or for Windows when using a domain / AAD account
ssh user@domain@hostname
```

2. In VS Code, select **Remote-SSH: Connect to Host...** from the Command Palette (**F1**, **Ctrl+Shift+P**) and use the same `user@hostname` as in step 1.



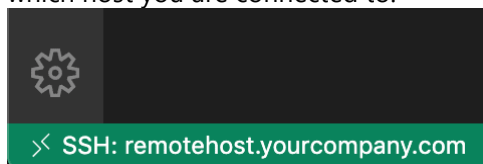
3. If VS Code cannot automatically detect the type of server you are connecting to, you will be asked to select the type manually.

4. After a moment, VS Code will connect to the SSH server and set itself up. VS Code will keep you up-to-date using a progress notification and you can see a detailed log in the **Remote - SSH** output channel.

Tip: Connection hanging or failing? See [troubleshooting tips](#) for information on resolving common problems.

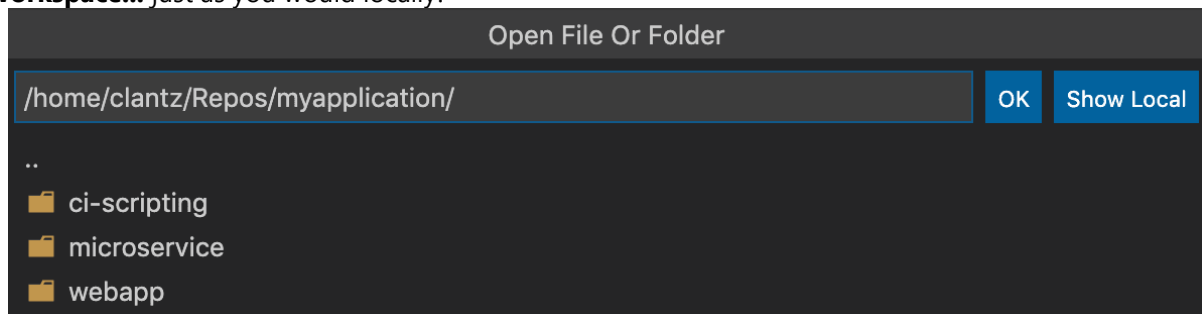
If you see errors about SSH file permissions, see the section on [Fixing SSH file permission errors](#).

5. After you are connected, you'll be in an empty window. You can always refer to the Status bar to see which host you are connected to.



Clicking on the Status bar item will provide a list of remote commands while you are connected.

6. You can then open any folder or workspace on the remote machine using **File > Open...** or **File > Open Workspace...** just as you would locally!



From here, install any extensions you want to use when connected to the host and start editing!

Note: On ARMv7l / ARMv8l **glibc** SSH hosts, some extensions may not work due to x86 compiled native code inside the extension.

Open a folder on a remote SSH host in a container

If you are using a Linux or macOS SSH host, you can use the Remote - SSH and [Dev Containers](#) extensions together to open a folder on your remote host inside of a container. You do not even need to have a Docker client installed locally.

To do so:

1. Follow the [installation](#) steps for the Dev Containers extension on your remote host.

2. **Optional:** Set up SSH [key based authentication](#) to the server so you do not need to enter your password multiple times.

3. Follow the [quick start](#) for the Remote - SSH extension to connect to a host and open a folder there.
4. Use the **Dev Containers: Reopen in Container** command from the Command Palette (**F1, Ctrl+Shift+P**). The rest of the [Dev Containers quick start](#) applies as-is. You can learn more about the [Dev Containers extension in its documentation](#). You can also see the [Develop on a remote Docker host](#) article for other options if this model does not meet your needs. (**Dev Containers: Attach to Running Container** can also work. see [Attach to a running container](#))

Disconnect from a remote host

To close the connection when you finish editing files on the remote host, choose **File > Close Remote Connection** to disconnect from the host. The default configuration does not include a keyboard shortcut for this command. You can also simply exit VS Code to close the remote connection.

Visual Studio Code Tips and Tricks

Command line

VS Code has a powerful command line interface (CLI) which allows you to customize how the editor is launched to support various scenarios.

Make sure the VS Code binary is on your path so you can simply type 'code' to launch VS Code. See the platform specific setup topics if VS Code is added to your environment path during installation.

open code with current directory

code .

open the current directory in the most recently used code window

code -r .

create a new window

code -n

change the language

code --locale=es

open diff editor

code --diff <file1> <file2>

open file at specific line and column <file:line[:character]>

code --goto package.json:10:5

see help options

code --help

disable all extensions

code --disable-extensions .

Miscellaneous

<Windows.h>

We can find **void Sleep(DWORD dwMilliseconds)** in window OS.