

Contents

Python in Visual Studio

Overview

[Overview of Python support](#)

[Tour the Visual Studio IDE](#)

[Data science and analytical applications workload](#)

Installation

[Installation](#)

[Install Python interpreters](#)

Quickstarts

[Create a web app with Flask](#)

[Create a project from a template](#)

[Open and run Python code in a folder](#)

[Create a project from existing code](#)

[Create a project from a repository](#)

[Create a project from a Cookiecutter template](#)

Tutorials

Python in Visual Studio

[0 - Install Python support](#)

[1 - Create a new Python project](#)

[2 - Writing and running code](#)

[3 - Using the interactive REPL window](#)

[4 - Running code in the debugger](#)

[5 - Installing packages and managing Python environments](#)

[6 - Working with Git](#)

Learn Django in Visual Studio

[1 - Create a project and solution](#)

[2 - Create a Django app](#)

[3 - Serve static files and add pages](#)

[4 - Use the Django Web Project template](#)

[5 - Authenticate users](#)

[6 - Use the Polls Django Web Project template](#)

[Learn Flask in Visual Studio](#)

[1 - Create a project and solution](#)

[2 - Create a Flask app](#)

[3 - Serve static files and add pages](#)

[4 - Use the Flask Web Project template](#)

[5 - Use the Polls Flask Web Project template](#)

[Concepts](#)

[Python projects](#)

[Python projects](#)

[Web project templates](#)

[Django web project template](#)

[Azure cloud service template](#)

[Azure SDK for Python](#)

[How-to guides](#)

[Manage Python environments](#)

[Manage Python environments](#)

[Select an interpreter for a project](#)

[Using requirements.txt for dependencies](#)

[Search paths](#)

[Environment window reference](#)

[Configure web apps for IIS](#)

[Edit Python code](#)

[Edit Python code](#)

[Format code](#)

[Refactor code](#)

[Use PyLint](#)

[Define custom menu commands](#)

[Interactive Python \(REPL\)](#)

[Interactive Python \(REPL\)](#)

[Using IPython REPL](#)

[Debugging](#)

[Debugging](#)

[Debugging code on remote Linux computers](#)

[Publishing to Azure App Service on Linux](#)

[Interacting with C++](#)

[Creating a C++ extension for Python](#)

[Python/C++ mixed-mode debugging](#)

[Symbols for mixed-mode debugging](#)

[Profiling](#)

[Unit testing](#)

[Using the Cookiecutter extension](#)

[Reference](#)

[Item templates](#)

[Options](#)

Work with Python in Visual Studio on Windows

6/13/2019 • 12 minutes to read • [Edit Online](#)

Python is a popular programming language that is reliable, flexible, easy to learn, free to use on all operating systems, and supported by both a strong developer community and many free libraries. Python supports all manners of development, including web applications, web services, desktop apps, scripting, and scientific computing, and is used by many universities, scientists, casual developers, and professional developers alike. You can learn more about the language on python.org and [Python for Beginners](#).

Visual Studio is a powerful Python IDE on Windows. Visual Studio provides [open-source](#) support for the Python language through the **Python Development** and **Data Science** workloads (Visual Studio 2017 and later) and the free Python Tools for Visual Studio extension (Visual Studio 2015 and earlier).

Python is not presently supported in Visual Studio for Mac, but is available on Mac and Linux through Visual Studio Code (see [questions and answers](#)).

To get started:

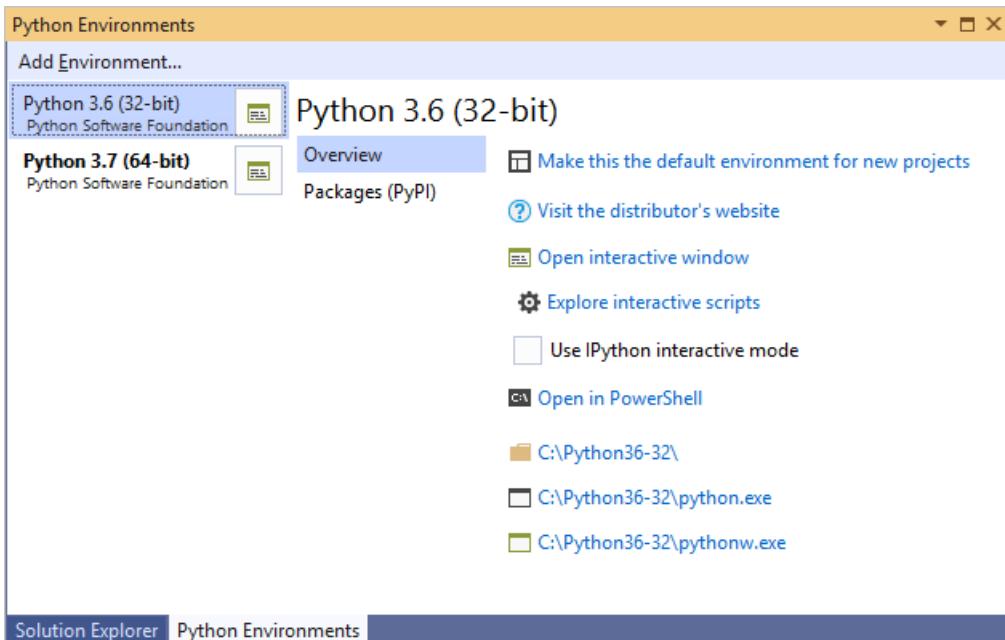
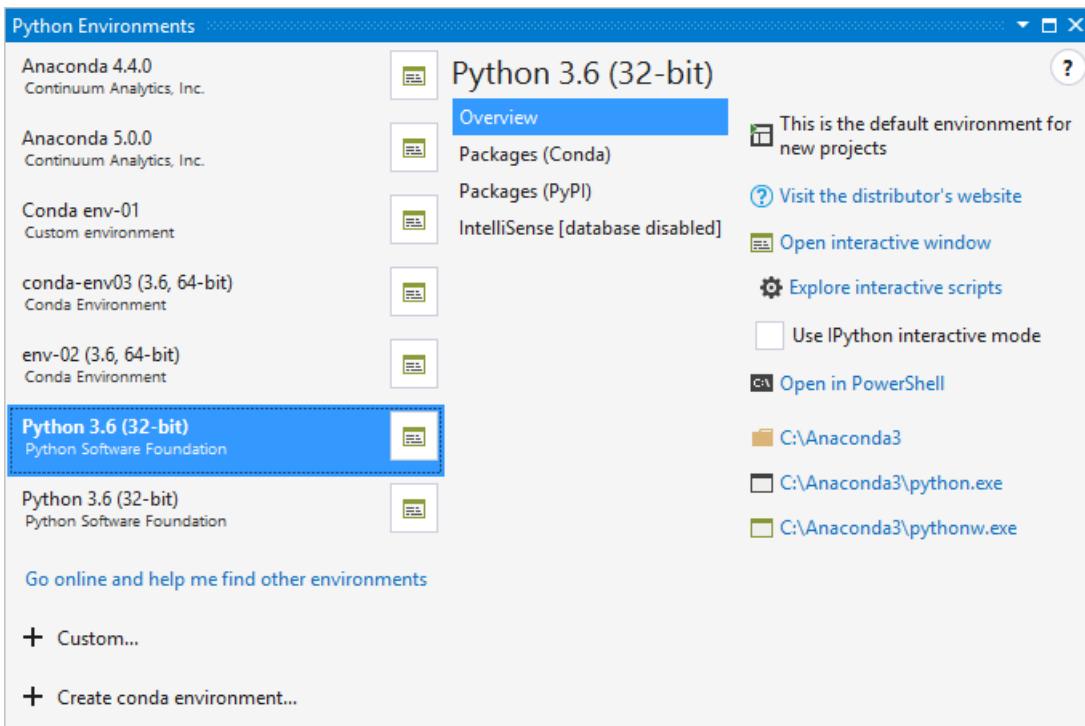
- Follow the [installation instructions](#) to set up the Python workload.
- Familiarize yourself with the Python capabilities of Visual Studio through the sections in this article.
- Go through one or more of the Quickstarts to create a project. If you're unsure, start with [Create a web app with Flask](#).
- Go through one or more of the Quickstarts to create a project. If you're unsure, start with [Quickstart: Open and run Python code in a folder](#) or [Create a web app with Flask](#).
- Follow the [Work with Python in Visual Studio](#) tutorial for a full end-to-end experience.

NOTE

Visual Studio supports Python version 2.7, as well as version 3.5 and greater. While it is possible to use Visual Studio to edit code written in other versions of Python, those versions are not officially supported and features such as IntelliSense and debugging might not work.

Support for multiple interpreters

Visual Studio's **Python Environments** window (shown below in a wide, expanded view) gives you a single place to manage all of your global Python environments, conda environments, and virtual environments. Visual Studio automatically detects installations of Python in standard locations, and allows you to configure custom installations. With each environment, you can easily manage packages, open an interactive window for that environment, and access environment folders.



Use the **Open interactive window** command to run Python interactively within the context of Visual Studio. Use the **Open in PowerShell** command to open a separate command window in the folder of the selected environment. From that command window you can run any python script.

For more information:

- [Manage Python environments](#)
- [Python Environments reference](#)

Rich editing, IntelliSense, and code comprehension

Visual Studio provides a first-class Python editor, including syntax coloring, auto-complete across all your code and libraries, code formatting, signature help, refactoring, linting, and type hints. Visual Studio also provides unique features like class view, **Go to Definition**, **Find All References**, and code snippets. Direct integration with the [Interactive window](#) helps you quickly develop Python code that's already saved in a file.

The screenshot shows a code editor window with Python code. The cursor is at the end of the line 'ss_map.a|'. A dropdown menu is open, showing suggestions: 'add_planet' (highlighted in blue), 'data', and 'date'. The code in the editor is:

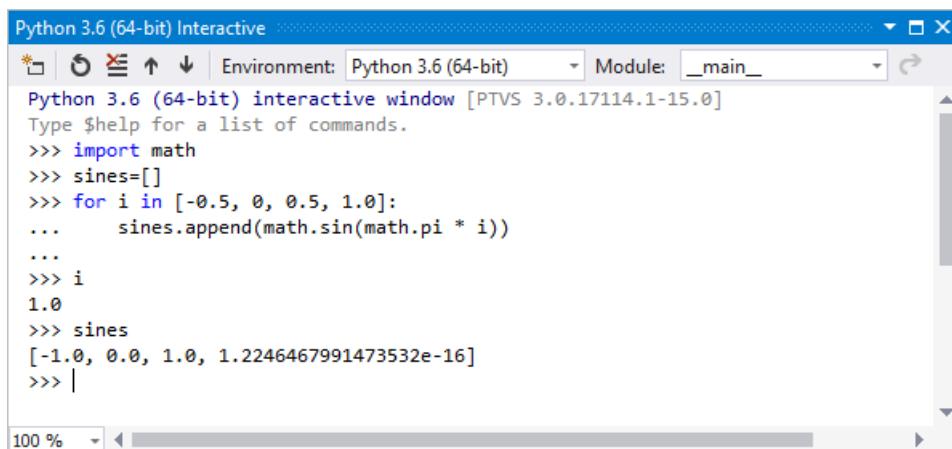
```
def main():
    for date in daterange(2015, 2016, step=timedelta(days=7)):
        # Create a map of the solar system on a particular day
        ss_map = SolarSystemMap(100, 27, date)
        ss_map.a|
```

For more information:

- Docs: [Edit Python code](#)
- Docs: [Format code](#)
- Docs: [Refactor code](#)
- Docs: [Use a linter](#)
- General Visual Studio feature docs: [Features of the code editor](#)

Interactive window

For every Python environment known to Visual Studio, you can easily open the same interactive (REPL) environment for a Python interpreter directly within Visual Studio, rather than using a separate command prompt. You can easily switch between environments as well. (To open a separate command prompt, select your desired environment in the **Python Environments** window, then select the **Open in PowerShell** command as explained earlier under [Support for multiple interpreters](#).)



Visual Studio also provides tight integration between the Python code editor and the **Interactive** window. The **Ctrl+Enter** keyboard shortcut conveniently sends the current line of code (or code block) in the editor to the **Interactive** window, then moves to the next line (or block). **Ctrl+Enter** lets you easily step through code without having to run the debugger. You can also send selected code to the **Interactive** window with the same keystroke, and easily paste code from the **Interactive** window into the editor. Together, these capabilities allow you to work out details for a segment of code in the **Interactive** window and easily save the results in a file in the editor.

Visual Studio also supports IPython/Jupyter in the REPL, including inline plots, .NET, and Windows Presentation Foundation (WPF).

For more information:

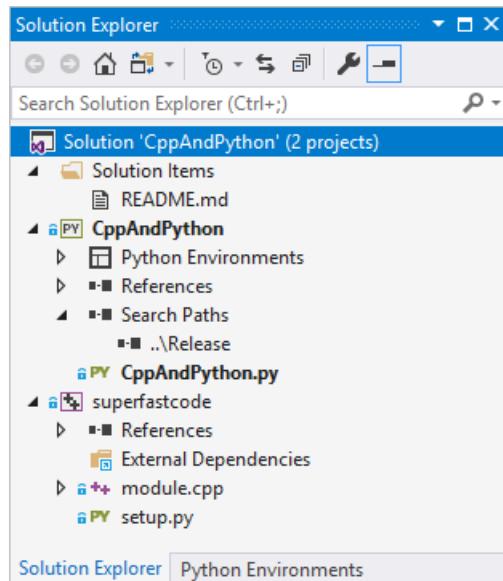
- [Interactive window](#)
- [IPython in Visual Studio](#)

Project system, and project and item templates

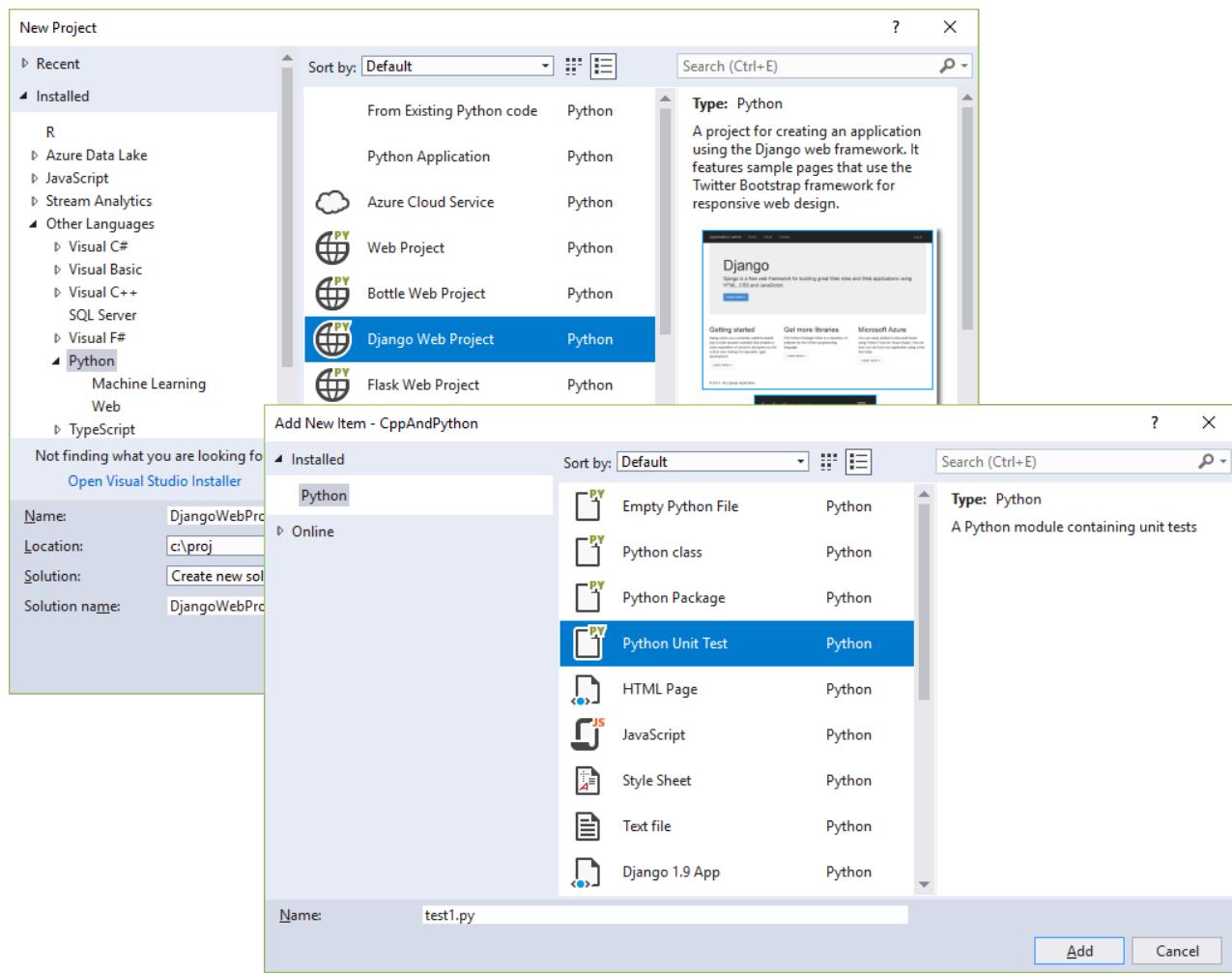
NOTE

Visual Studio 2019 supports opening a folder containing Python code and running that code without creating Visual Studio project and solution files. For more information, see [Quickstart: Open and run Python code in a folder](#). There are, however, benefits to using a project file, as explained in this section.

Visual Studio helps you manage the complexity of a project as it grows over time. A *Visual Studio project* is much more than a folder structure: it includes an understanding of how different files are used and how they relate to each other. Visual Studio helps you distinguish app code, test code, web pages, JavaScript, build scripts, and so on, which then enable file-appropriate features. A Visual Studio solution, moreover, helps you manage multiple related projects, such as a Python project and a C++ extension project.



Project and item templates automate the process of setting up different types of projects and files, saving you valuable time and relieving you from managing intricate and error-prone details. Visual Studio provides templates for web, Azure, data science, console, and other types of projects, along with templates for files like Python classes, unit tests, Azure web configuration, HTML, and even Django apps.

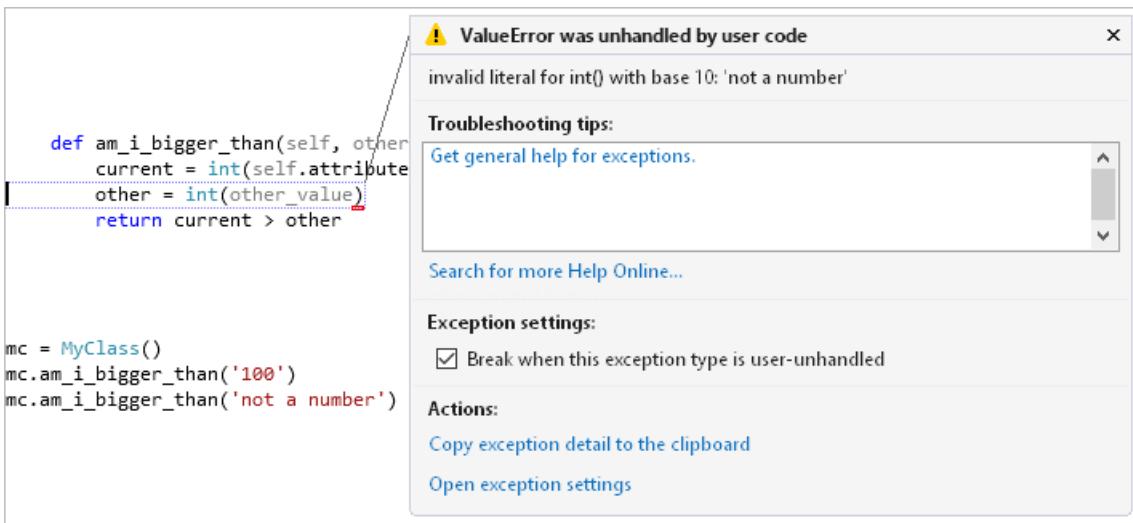


For more information:

- Docs: [Manage Python projects](#)
- Docs: [Item templates reference](#)
- Docs: [Python project templates](#)
- Docs: [Work with C++ and Python](#)
- General Visual Studio feature docs: [Project and item templates](#)
- General Visual Studio feature docs: [Solutions and projects in Visual Studio](#)

Full-featured debugging

One of Visual Studio's strengths is its powerful debugger. For Python in particular, Visual Studio includes Python/C++ mixed-mode debugging, remote debugging on Linux, debugging within the **Interactive** window, and debugging Python unit tests.



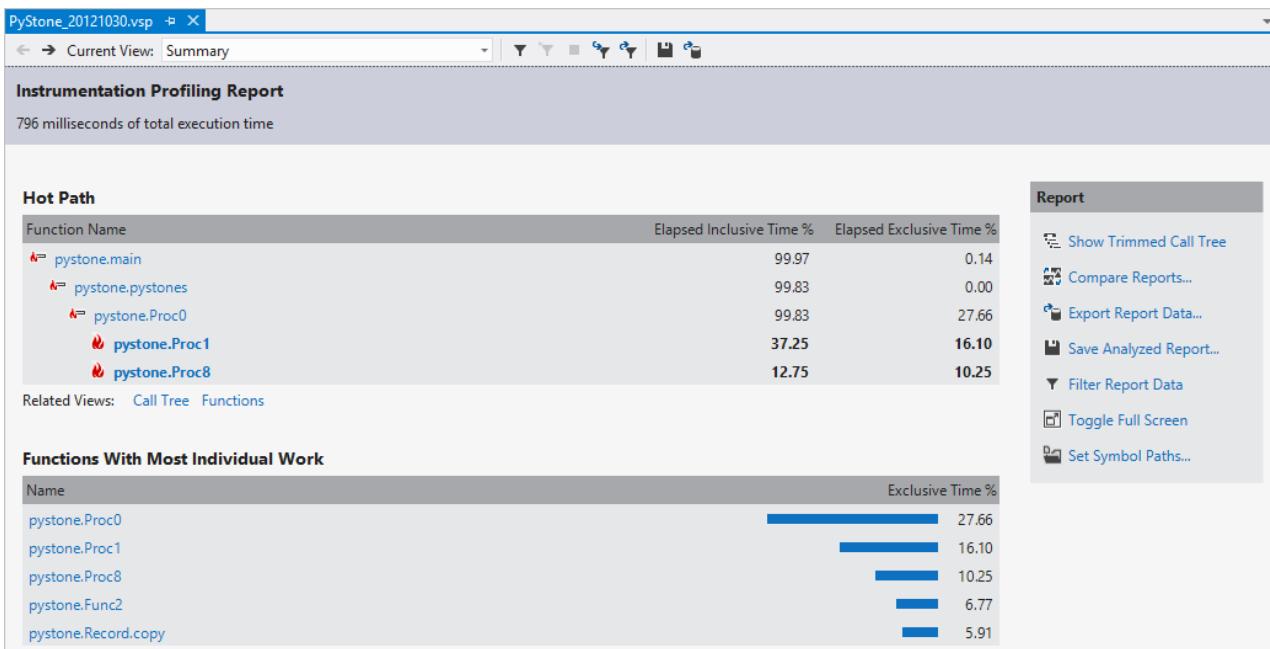
In Visual Studio 2019, you can run and debug code without having a Visual Studio project file. See [Quickstart: Open and run Python code in a folder](#) for an example.

For more information:

- Docs: [Debug Python](#)
- Docs: [Python/C++ mixed-mode debugging](#)
- Docs: [Remote debugging on Linux](#)
- General Visual Studio feature docs: [Feature tour of the Visual Studio Debugger](#)

Profiling tools with comprehensive reporting

Profiling explores how time is being spent within your application. Visual Studio supports profiling with CPython-based interpreters and includes the ability to compare performance between different profiling runs.

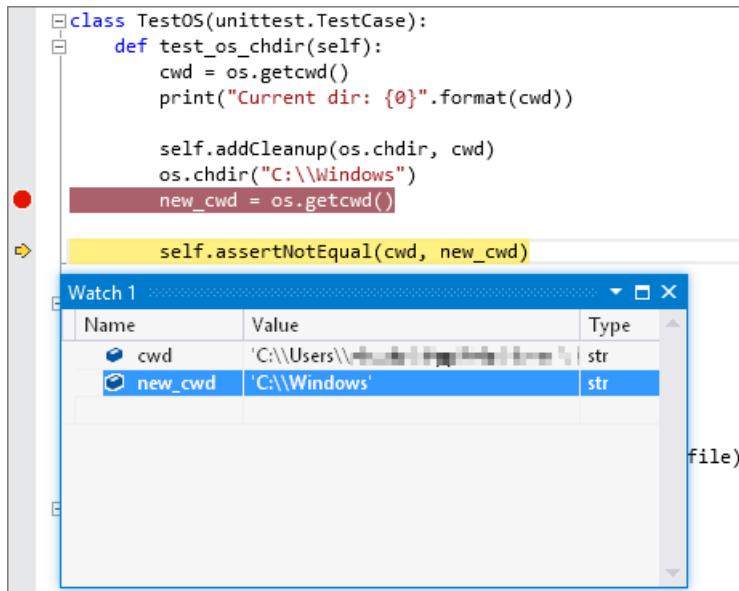


For more information:

- Docs: [Python profiling tools](#)
- General Visual Studio feature docs: [Profiling Feature Tour](#). (Not all Visual Studio profiling features are available for Python).

Unit testing tools

Discover, run, and manage tests in Visual Studio **Test Explorer**, and easily debug unit tests.



For more information:

- Docs: [Unit testing tools for Python](#)
- General Visual Studio feature docs: [Unit test your code](#).

Azure SDK for Python

The Azure libraries for Python simplify consuming Azure services from Windows, Mac OS X, and Linux apps. You can use them to create and manage Azure resources, as well as to connect to Azure services.

For more information, see [Azure SDK for Python](#) and [Azure libraries for Python](#) .

Questions and answers

Q. Is Python support available with Visual Studio for Mac?

A. Not at this time, but you can up vote the request on [Developer Community](#). The [Visual Studio for Mac](#) documentation identifies the current types of development that it does support. In the meantime, Visual Studio Code on Windows, Mac, and Linux [works well with Python through available extensions](#).

Q. What can I use to build UI with Python?

A. The main offering in this area is the [Qt Project](#), with bindings for Python known as [PySide \(the official binding\)](#) (also see [PySide downloads](#)) and [PyQt](#). At present, Python support in Visual Studio does not include any specific tools for UI development.

Q. Can a Python project produce a stand-alone executable?

A. Python is generally an interpreted language, with which code is run on demand in a suitable Python-capable environment such as Visual Studio and web servers. Visual Studio itself does not at present provide the means to create a stand-alone executable, which essentially means a program with an embedded Python interpreter. However, the Python community supplied different means to create executables as described on [StackOverflow](#). CPython also supports being embedded within a native application, as described on the blog post, [Using CPython's embeddable zip file](#).

Feature support

Python features can be installed in the following editions of Visual Studio as described in the [installation guide](#):

- [Visual Studio 2019 \(all editions\)](#)
 - Visual Studio 2017 (all editions)
 - Visual Studio 2015 (all editions)
 - Visual Studio 2013 Community Edition
 - Visual Studio 2013 Express for Web, Update 2 or higher
 - Visual Studio 2013 Express for Desktop, Update 2 or higher
 - Visual Studio 2013 (Pro edition or higher)
 - Visual Studio 2012 (Pro edition or higher)
 - Visual Studio 2010 SP1 (Pro edition or higher; .NET 4.5 required)

Visual Studio 2015 and earlier are available at visualstudio.microsoft.com/vs/older-downloads/.

IMPORTANT

Features are fully supported and maintained for only the latest version of Visual Studio. Features are available in older versions but are not actively maintained.

PROJECT SYSTEM	2017+	2015	2013 COMM	2013 DESKTOP	2013 WEB	2013 PRO+	2012 PRO+	2010 SP1 PRO+
Show all files	✓	✓	✓	✓	✓	✓	✓	✓
Source control	✓	✓	✓	✓	✓	✓	✓	✓
Git integration	✓	✓	✓	✓	✓	✓	✓ ¹	✗

EDITING	2017+	2015	2013 COMM	2013 DESKTOP	2013 WEB	2013 PRO+	2012 PRO+	2010 SP1 PRO+
Refactor - extract method	✓	✓	✓	✓	✓	✓	✓	✓
Refactor - add/remove import	✓	✓	✓	✓	✓	✓	✓	✓
PyLint	✓	✓	✓	✓	✓	✓	✓	✓

INTERACTIVE WINDOW	2017+	2015	2013 COMM	2013 DESKTOP	2013 WEB	2013 PRO+	2012 PRO+	2010 SP1 PRO+
Interactive window	✓	✓	✓	✓	✓	✓	✓	✓
IPython with inline graphs	✓	✓	✓	✓	✓	✓	✓	✓

DESKTOP	2017+	2015	2013 COMM	2013 DESKTOP	2013 WEB	2013 PRO+	2012 PRO+	2010 SP1 PRO+
Console/Windows application	✓	✓	✓	✓	✓	✓	✓	✓
IronPython WPF (with XAML designer)	✓	✓	✓	✓	✓	✓	✓	✓
IronPython Windows Forms	✓	✓	✓	✓	✓	✓	✓	✓

WEB	2017+	2015	2013 COMM	2013 DESKTOP	2013 WEB	2013 PRO+	2012 PRO+	2010 SP1 PRO+
Django web project	✓	✓	✓	✗	✓	✓	✓	✓

WEB	2017+	2015	2013 COMM	2013 DESKTOP	2013 WEB	2013 PRO+	2012 PRO+	2010 SP1 PRO+
Bottle web project	✓	✓	✓	✗	✓	✓	✓	✓
Flask web project	✓	✓	✓	✗	✓	✓	✓	✓
Generic web project	✓	✓	✓	✗	✓	✓	✓	✓

AZURE	2017+	2015	2013 COMM	2013 DESKTOP	2013 WEB	2013 PRO+	2012 PRO+	2010 SP1 PRO+
Deploy to web site	✓	✓	✓	✗	✓	✓	✓	✓ ²
Deploy to web role	✓	✓	✓	✗	✓ ⁴	✓ ⁴	✓ ³	✗
Deploy to worker role	?	?	?	✗	✓ ⁴	✓ ⁴	✓ ³	✗
Run in Azure emulator	?	?	?	✗	✓ ⁴	✓ ⁴	✓ ³	✗
Remote debugging	✓	✓	✓	✗	✓ ⁶	✓ ⁸	✓ ⁸	✗
Attach Server Explorer	✓	✓	✓	✗	✓ ⁷	✓ ⁷	✗	✗

DJANGO TEMPLATES	2017+	2015	2013 COMM	2013 DESKTOP	2013 WEB	2013 PRO+	2012 PRO+	2010 SP1 PRO+
Debugging	✓	✓	✓	✗	✓	✓	✓	✓
Auto-complete	✓	✓	✓	✗	✓ ⁵	✓ ⁵	✓	✓
Auto-complete for CSS and JavaScript	✓	✓	✓	✗	✓ ⁵	✓ ⁵	✗	✗

DEBUGGING	2017+	2015	2013 COMM	2013 DESKTOP	2013 WEB	2013 PRO+	2012 PRO+	2010 SP1 PRO+
Debugging	✓	✓	✓	✓	✓	✓	✓	✓
Debugging without a project	✓	✓	✓	✓	✓	✓	✓	✓
Debugging - attach to editing	✓	✓	✓	✓	✗	✓	✓	✓
Mixed-mode debugging	✓	✓	✓	✓	✓	✓	✓	✗
Remote debugging (Windows, Mac OS X, Linux)	✓	✓	✓	✓	✗	✓	✓	✓
Debug Interactive window	✓	✓	✓	✓	✓	✓	✓	✓

PROFILING	2017+	2015	2013 COMM	2013 DESKTOP	2013 WEB	2013 PRO+	2012 PRO+	2010 SP1 PRO+
Profiling	✓	✓	✓	✗	✗	✓	✓	✓

TEST	2017+	2015	2013 COMM	2013 DESKTOP	2013 WEB	2013 PRO+	2012 PRO+	2010 SP1 PRO+
Test explorer	✓	✓	✓	✓	✓	✓	✓	✗
Run test	✓	✓	✓	✓	✓	✓	✓	✗
Debug test	✓	✓	✓	✓	✓	✓	✓	✗

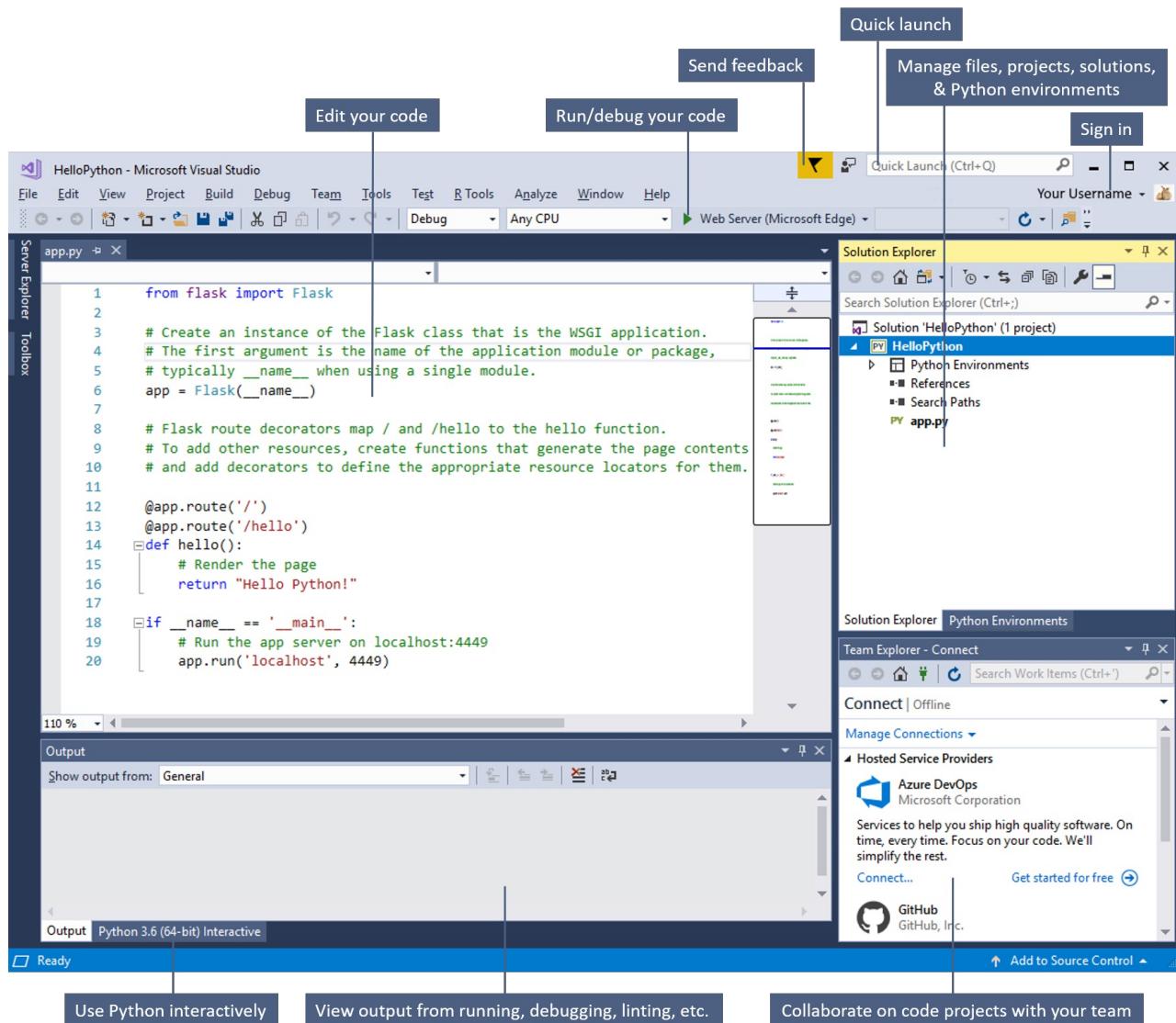
1. Git support for Visual Studio 2012 is available in the Visual Studio Tools for Git extension, available on the [Visual Studio Marketplace](#).
2. Deployment to Azure Web Site requires [Azure SDK for .NET 2.1 - Visual Studio 2010 SP1](#). Later versions don't support Visual Studio 2010.

3. Support for Azure Web Role and Worker Role requires [Azure SDK for .NET 2.3 - VS 2012](#) or later.
4. Support for Azure Web Role and Worker Role requires [Azure SDK for .NET 2.3 - VS 2013](#) or later.
5. Django template editor in Visual Studio 2013 has some known issues that are resolved by installing Update 2.
6. Requires Windows 8 or later. Visual Studio 2013 Express for Web doesn't have the **Attach to Process** dialog, but Azure Web Site remote debugging is still possible using the **Attach Debugger (Python)** command in **Server Explorer**. Remote debugging requires [Azure SDK for .NET 2.3 - Visual Studio 2013](#) or later.
7. Requires Windows 8 or later. **Attach Debugger (Python)** command in **Server Explorer** requires [Azure SDK for .NET 2.3 - Visual Studio 2013](#) or later.
8. Requires Windows 8 or later.

Welcome to the Visual Studio IDE | Python

7/11/2019 • 5 minutes to read • [Edit Online](#)

The Visual Studio *integrated development environment* is a creative launching pad for Python (and other languages) that you can use to edit, debug, and test code, and then publish an app. An integrated development environment (IDE) is a feature-rich program that can be used for many aspects of software development. Over and above the standard editor and debugger that most IDEs provide, Visual Studio includes code completion tools, interactive REPL environments, and other features to ease the software development process.



This image shows Visual Studio with an open Python project and several key tool windows you'll likely use:

- **Solution Explorer** (top right) lets you view, navigate, and manage your code files. **Solution Explorer** can help organize your code by grouping the files into [solutions and projects](#).
 - Alongside **Solution Explorer** is **Python Environments**, where you manage the different Python interpreters that are installed on your computer.
 - You can also open and run Python code in a folder without creating Visual Studio project and solution files. For more information, see [Quickstart: Open and run Python code in a folder](#).
- The [editor window](#) (center), where you'll likely spend a majority of your time, displays file contents. This is where you [edit Python code](#), navigate within your code structure, and set breakpoints during debugging

sessions. With Python, you can also select code and press Ctrl+Enter to run that code in an [interactive REPL window](#).

- The [Output window](#) (bottom center) is where Visual Studio sends notifications such as debugging and error messages, warnings, publishing status messages, and more. Each message source has its own tab.
 - A [Python Interactive REPL window](#) appears in the same area as the Output window.
- [Team Explorer](#) (bottom right) lets you track work items and share code with others using version control technologies such as [Git](#) and [Team Foundation Version Control \(TFVC\)](#).

Editions

Visual Studio is available for Windows and Mac; however, Python support is available only on Visual Studio for Windows.

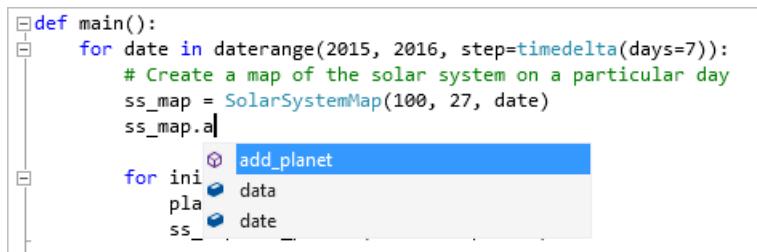
There are three editions of Visual Studio on Windows: Community, Professional, and Enterprise. See [Compare Visual Studio IDEs](#) to learn about which features are supported in each edition.

Popular productivity features

Some of the popular features in Visual Studio that help you to be more productive as you develop software include:

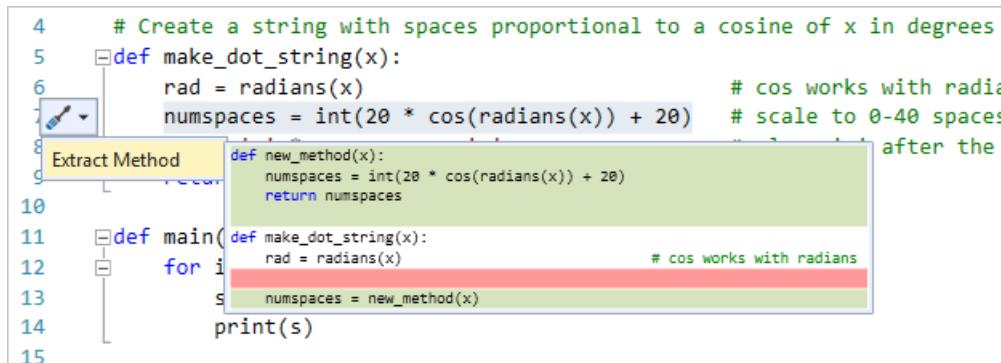
- [IntelliSense](#)

IntelliSense is a term for a set of features that displays information about your code directly in the editor and, in some cases, write small bits of code for you. It's like having basic documentation inline in the editor, which saves you from having to look up type information elsewhere. IntelliSense features vary by language, and the [Editing Python code](#) article has details for Python. The following illustration shows how IntelliSense displays a member list for a type:



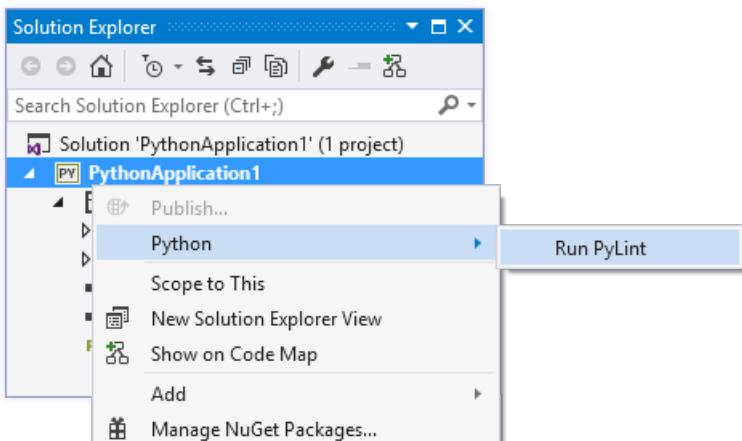
- [Refactoring](#)

By right-clicking on a piece of code and selecting **Quick actions and Refactorings**, Visual Studio provides you with operations such as intelligent renaming of variables, extracting one or more lines of code into a new method, changing the order of method parameters, and more.



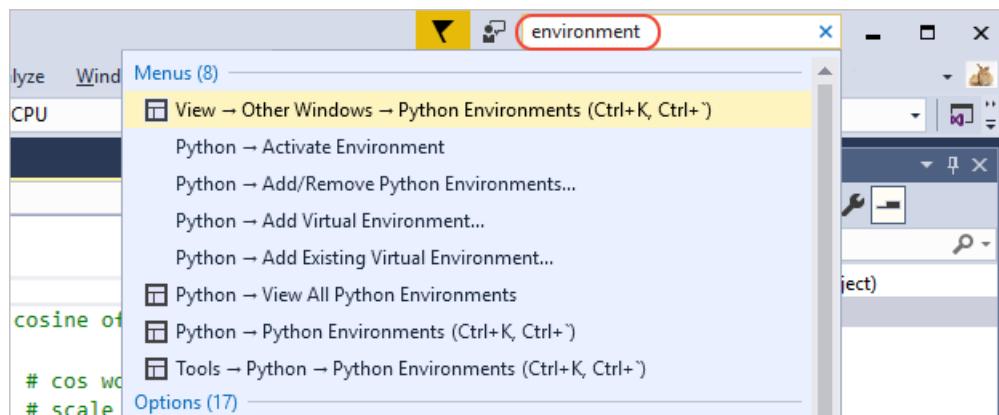
- [Linting](#)

Linting checks for errors and common problems in your Python code, encouraging you with good Python coding patterns.



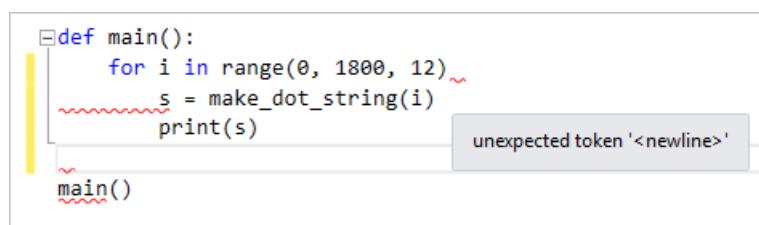
- Search box

Visual Studio can seem overwhelming at times with so many menus, options, and properties. The search box is a great way to rapidly find what you need in Visual Studio. When you start typing the name of something you're looking for, Visual Studio lists results that take you exactly where you need to go. If you need to add functionality to Visual Studio, for example to add support for an additional programming language, the search box provides results that open Visual Studio Installer to install a workload or individual component.



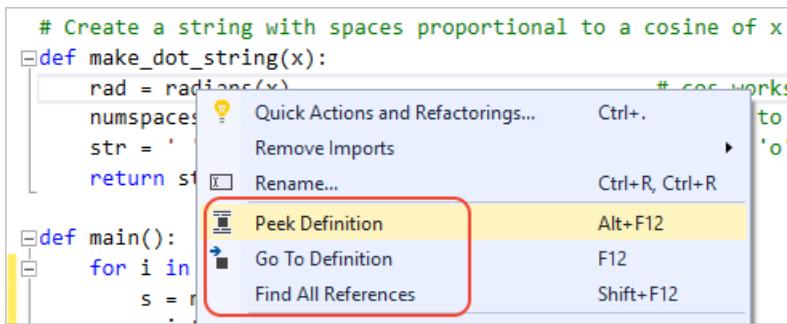
- Squiggles and Quick Actions

Squiggles are wavy underlines that alert you to errors or potential problems in your code as you type. These visual clues enable you to fix problems immediately without waiting for the error to be discovered during build or when you run the program. If you hover over a squiggle, you see additional information about the error. A light bulb may also appear in the left margin with actions, known as Quick Actions, to fix the error.



- Go To and Peek Definition

The **Go To Definition** feature takes you directly to the location where a function or type is defined. The **Peek Definitions** command displays the definition in a window without opening a separate file. The **Find All References** command also provides a helpful way to discover where any given identifier is both defined and used.



Powerful features for Python

- Run code without a project

Starting in Visual Studio 2019, you can open a folder containing Python code to enjoy features like IntelliSense and debugging without having to create a Visual Studio project for the code.

- Collaborate using Visual Studio

Visual Studio Live Share enables you to collaboratively edit and debug with others in real time, regardless of what programming language you're using or app types you're building.

- Python Interactive REPL

Visual Studio provides an interactive read-evaluate-print loop (REPL) window for each of your Python environments, which improves upon the REPL you get with `python.exe` on the command line. In the **Interactive** window you can enter arbitrary Python code and see immediate results.

The screenshot shows the Python 3.6 (64-bit) Interactive window in Visual Studio. The window title is "Python 3.6 (64-bit) Interactive". The environment dropdown shows "Python 3.6 (64-bit)" and the module dropdown shows "_main_". The window displays the following Python session:

```

Python 3.6 (64-bit) interactive window [PTVS 3.0.17114.1-15.0]
Type $help for a list of commands.
>>> import math
>>> sines=[]
>>> for i in [-0.5, 0, 0.5, 1.0]:
...     sines.append(math.sin(math.pi * i))
...
>>> i
1.0
>>> sines
[-1.0, 0.0, 1.0, 1.2246467991473532e-16]
>>>

```

- Debugging

Visual Studio provides a comprehensive debugging experience for Python, including attaching to running processes, evaluating expressions in the **Watch** and **Immediate** windows, inspecting local variables, breakpoints, step in/out/over statements, **Set Next Statement**, and more. You can also debug remote Python code running on Linux computers.

The screenshot shows a Microsoft Visual Studio Preview window with the title "guessing-game.py (Debugging) - Microsoft Visual Studio Preview". The menu bar includes File, Edit, View, Project, Build, Debug, Team, Tools, Test, Analyze, Window, Help, and Kraig Brockschmidt. The toolbar has icons for back, forward, search, and other debugging functions. The status bar at the bottom shows "Process: [25590] python3 @ tcp://40.83.191 - Lifecycle Events - Thread: [1] [140017000875776] MainThread".

The code editor displays the following Python script:

```
print('Well, {0}, I am thinking of a number between 1 and 20.'.format(name))

while guesses_made < 6:
    guess = int(input('Take a guess: '))
    guesses_made += 1
    if guess < number:
        print('Your guess is too low.')
    if guess > number:
        print('Your guess is too high.')
    if guess == number:
        break
if guess == number:
    print('Good job, {0}! You guessed my number in {1} guesses!'.format(name, guesses_made))
else:
    print('The number I was thinking of was {0}'.format(number))
```

The Locals window shows the following variable values:

Name	Type
guesses_made	int
guess	int
name	str
builtins	module
loader	SourceFile
package	NoneType
spec	NoneType
doc	NoneType
file	str

The Call Stack window shows the current stack trace:

Name	Lang
guessing-game module line 14	Pyth

- Interacting with C++

Many libraries created for Python are written in C++ for optimal performance. Visual Studio provides rich facilities for developing C++ extensions, including [mixed-mode debugging](#).

The screenshot shows the PyCharm IDE interface with two open files:

- main.py**: Contains Python code for thread communication and native method calls.
- TestModule.cpp**: Contains C++ code for a natmod_NatObjObject struct and its methods.

The Locals tool window displays the following variable values:

Name	Type	Value
[Globals]		
a_natobj	<natmod.NatObj object at 0x0000000000000000>	
[C++ view]	{ob_base={ob_refcnt=4 ob_type=natmod.pyd!NatObj}}	
ob_base	{ob_refcnt=4 ob_type=0x703 natmod.pyd!NatObj}	
o	0x028e5fc8 {ob_refcnt=4 ob_type=natmod.pyd!NatObj}	
i	42	int
s	0x703520e4 "NatObj"	const char *
hidden	0.0000000000	float
i	42	int
o	<natmod.NatObj object at 0x0000000000000000>	
s	'NatObj'	const char *
n	(123, 'foo', True)	tuple
natmod	<module object at 0x029893f module>	

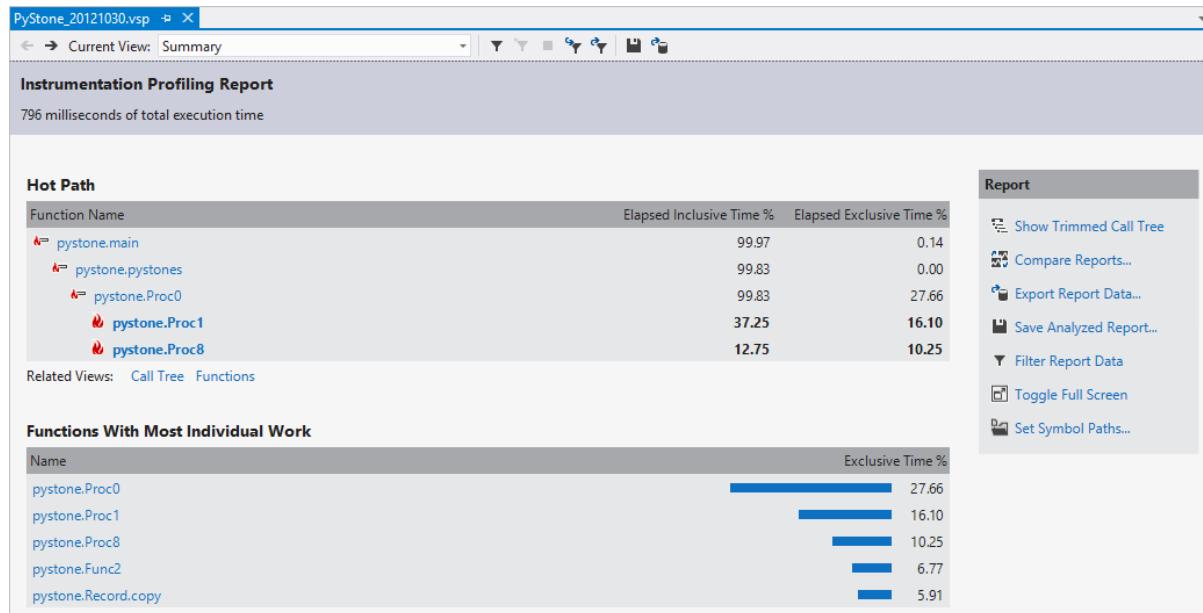
The Call Stack tool window shows the following call stack:

Name	Language
natmod.pyd!natmod_NatObj_frob(natmod_NatObjObject*)	C++
[Python to Native Transition]	
main.py!python_called_from_native Line 163	Python
[Native to Python Transition]	
natmod.pyd!natmod_callback(_object * self, _object * a)	C++
[Python to Native Transition]	
main.py!python_calling_native Line 157	Python
main.py!python_calling_python Line 153	Python
main.py!<module> Line 166	Python
[Native to Python Transition]	
python.exe!_tmainCRTStartup() Line 552	C
kernel32.dll!756485430	Unknown
[Frames below may be incorrect and/or missing, no symbols available]	

- Profiling

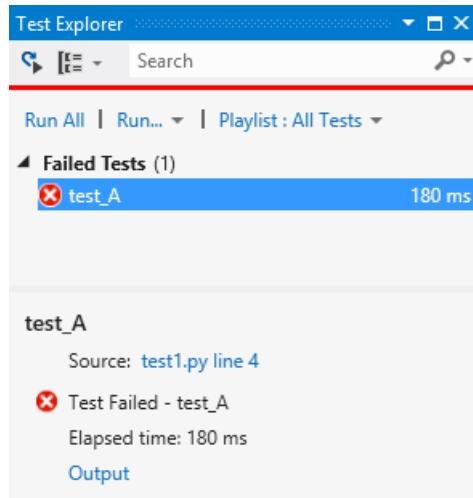
When using a CPython-based interpreter, you can evaluate the performance of your Python code within

Visual Studio.



- Unit Testing

Visual Studio provides integrated support for discovering, running, and debugging unit tests all in the context of the IDE.



Next steps

Explore Python in Visual Studio further by following one of the following quickstarts or tutorials:

[Quickstart: Create a web app with Flask](#)

[Work with Python in Visual Studio](#)

[Get started with the Django web framework in Visual Studio](#)

[Get started with the Flask web framework in Visual Studio](#)

See also

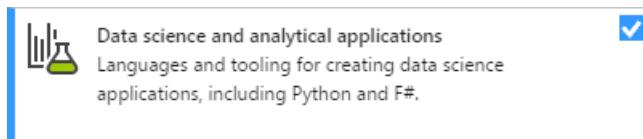
- Discover [more Visual Studio features](#)
- Visit [visualstudio.microsoft.com](#)
- Read [The Visual Studio blog](#)

Install data science support in Visual Studio

4/9/2019 • 3 minutes to read • [Edit Online](#)

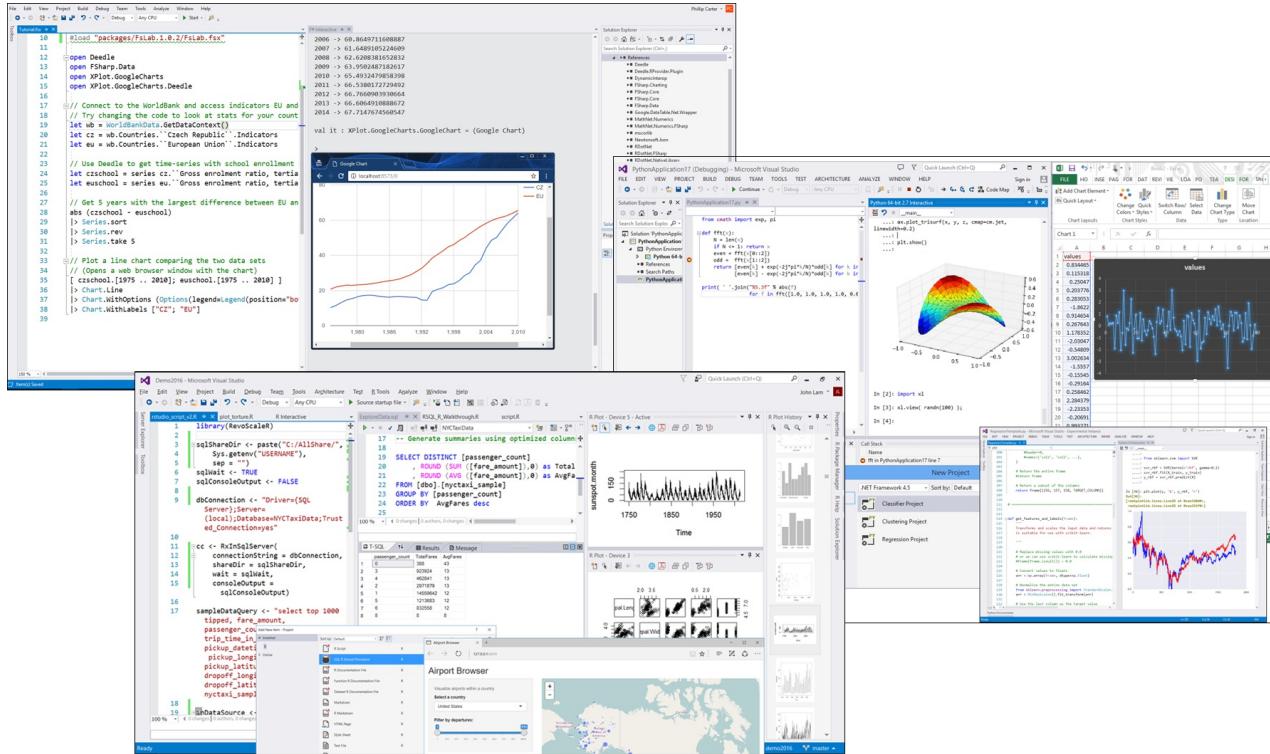
The Data Science and Analytical Applications workload, which you select and install through the Visual Studio installer, brings together several languages and their respective runtime distributions:

- [Python and Anaconda](#)
- [F# with the .NET framework](#)
- [R and Microsoft R Client](#)
- [Python](#)
- [F# with the .NET framework](#)



Python and R are two of the primary scripting languages used for data science. Both languages are easy to learn and are supported by a rich ecosystem of packages. Those packages address a wide range of scenarios such as data acquisition, cleaning, model training, deployment, and plotting. F# is also a powerful functional-first .NET language that's suited for a wide variety of data processing tasks.

Python is a primary scripting language used for data science. Python is easy to learn and is supported by a rich ecosystem of packages. Those packages address a wide range of scenarios such as data acquisition, cleaning, model training, deployment, and plotting. F# is also a powerful functional-first .NET language that's suited for a wide variety of data processing tasks. (For the R language we recommend [Azure Notebooks](#).)



Workload options

By default, the workload installs the following options, which you can modify in the summary section for the

workload in the Visual Studio installer:

- F# desktop language support
- Python:
 - Python language support
 - Python web support
- F# language support
- Python:
 - Python language support
 - [Anaconda3 64-bit](#), a Python distro that includes extensive data science libraries and a Python interpreter.
 - Python web support
 - Cookiecutter template support
- R:
 - R language support
 - Runtime support for R development tools
 - [Microsoft R Client](#) (Microsoft's fully compatible, community-supported R interpreter with ScaleR libraries for faster computation on single nodes or clusters. You can also use any R from [CRAN](#).)

SQL Server integration

SQL Server supports using both Python and R to do advanced analytics directly inside SQL Server. R support is included with SQL Server 2016 and later; Python support is available in SQL Server 2017 CTP 2.0 and later.

SQL Server supports using Python to do advanced analytics directly inside SQL Server. Python support is available in SQL Server 2017 CTP 2.0 and later.

You enjoy the following advantages by running your code where your data already lives:

- **Elimination of data movement:** Instead of moving data from the database to your application or model, you can build applications in the database. This capability eliminates barriers of security, compliance, governance, integrity, and a host of similar issues related to moving vast amounts of data around. You can also consume datasets that couldn't fit into the memory of a client machine.
- **Easy deployment:** Once you have a model ready, deploying it to production is a simple matter of embedding it in a T-SQL script. Any SQL client application written in any language can then take advantage of the models and intelligence through a stored procedure call. No specific language integrations are necessary.
- **Enterprise-grade performance and scale:** You can use SQL Server's advanced capabilities like in-memory table and column store indexes with the high-performance scalable APIs in the RevoScale packages. The elimination of data movement also means that you avoid client memory constraints as your data grows or you wish to increase the performance of the application.
- **Rich extensibility:** You can install and run any of the latest open source packages in SQL Server to build deep learning and AI applications on huge amounts of data in SQL Server. Installing a package in SQL Server is as simple as installing a package on your local machine.
- **Wide availability at no additional cost:** Language integrations are available in all editions of SQL Server 2017 and later, including the Express edition.

To take full advantage of SQL Server integration, use the Visual Studio installer to install the **Data storage and processing** workload with the **SQL Server Data Tools** option. The latter option enables SQL IntelliSense, syntax highlighting, and deployment.

 Data storage and processing
Connect, develop, and test data solutions with SQL Server, Azure Data Lake, or Hadoop.

▼ Data storage and processing

Optional

- SQL Server Data Tools
 Azure Data Lake and Stream Analytics Tools
 .NET Framework 4 – 4.6 development tools
 Redgate SQL Search
 F# desktop language support

For more information:

- [Work with SQL Server and R](#)
- [In-database Advanced Analytics with R in SQL Server 2016 \(blog\)](#)
- [Python in SQL Server 2017: enhanced in-database machine learning \(blog\)](#)

Additional services and SDKs

In addition to what's in the Data Science and Analytics Applications workload directly, the Azure Notebooks service and the Azure SDK for Python are also helpful for data science.

The Azure SDK for Python makes it easy to consume and manage Microsoft Azure services from applications running on Windows, Mac, and Linux. For more information, see [Azure SDK for Python](#)

Azure Notebooks (currently in preview) provides free online access to Jupyter notebooks running in the cloud on Microsoft Azure. The service includes sample notebooks in Python, R, and F# to get you started. Visit notebooks.azure.com.

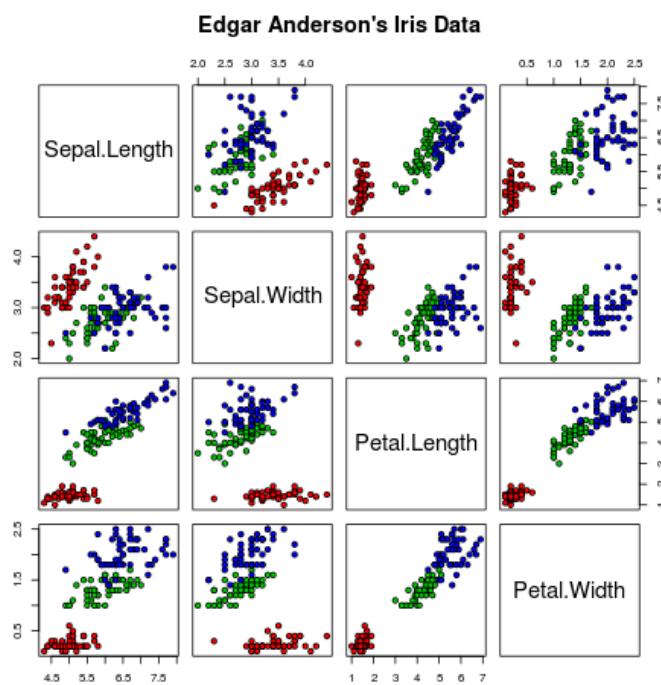


File Edit View Insert Cell Kernel Data Widgets Help | R O

A scatterplot matrix

The good old Iris data (yet again)

```
In [4]: pairs(iris[1:4], main="Edgar Anderson's Iris Data", pch=21,  
           bg = c("red", "green3", "blue")[unclass(iris$Species)])
```



How to install Python support in Visual Studio on Windows

4/26/2019 • 6 minutes to read • [Edit Online](#)

To install Python support for Visual Studio (also known as Python Tools for Visual Studio or PTVS), follow the instructions in the section that matches your version of Visual Studio:

- [Visual Studio 2017 and Visual Studio 2019](#)
- [Visual Studio 2015](#)
- [Visual Studio 2013 and earlier](#)

To quickly test Python support after following the installation steps, open the **Python Interactive** window by pressing **Alt+I** and entering `2+2`. If you don't see the output of `4`, recheck your steps.

TIP

The Python workload includes the helpful Cookiecutter extension that provides a graphical user interface to discover templates, input template options, and create projects and files. For details, see [Use Cookiecutter](#).

NOTE

Python support is not presently available in Visual Studio for Mac, but is available on Mac and Linux through Visual Studio Code. See [questions and answers](#).

Visual Studio 2019 and Visual Studio 2017

1. Download and run the latest Visual Studio installer. If you have Visual Studio installed already, run the Visual Studio Installer, select the **Modify** option (see [Modify Visual Studio](#)) and go to step 2.

[Install Visual Studio 2019 Community](#)

TIP

The Community edition is for individual developers, classroom learning, academic research, and open source development. For other uses, install [Visual Studio 2019 Professional](#) or [Visual Studio 2019 Enterprise](#).

2. The installer presents you with a list of workloads, which are groups of related options for specific development areas. For Python, select the **Python development** workload.



Optional: if you're working with data science, also consider the **Data science and analytical applications** workload. This workload includes support for the Python, R, and F# languages. For more information, see [Data science and analytical applications workload](#).

NOTE

The Python and Data Science workloads are available only with Visual Studio 2017 version 15.2 and later.

Optional: if you're working with data science, also consider the **Data science and analytical applications** workload. This workload includes support for the Python and F# languages. For more information, see [Data science and analytical applications workload](#).

3. On the right side of the installer, chose additional options if desired. Skip this step to accept the default options.

Summary

✓ Python development
Included
✓ Python language support
Optional
<input checked="" type="checkbox"/> Cookiecutter template support
<input checked="" type="checkbox"/> Python web support
<input checked="" type="checkbox"/> Python 3 64-bit (3.6.0)
<input type="checkbox"/> Python IoT support
<input type="checkbox"/> Python native development tools
<input type="checkbox"/> Azure Cloud Services core tools
<input type="checkbox"/> Python 2 64-bit (2.7.13)
<input type="checkbox"/> Anaconda3 64-bit (4.3.0.1)
<input type="checkbox"/> Anaconda2 64-bit (4.3.0.1)
<input type="checkbox"/> Python 3 32-bit (3.6.0)
<input type="checkbox"/> Python 2 32-bit (2.7.13)
<input type="checkbox"/> Anaconda3 32-bit (4.3.0.1)
<input type="checkbox"/> Anaconda2 32-bit (4.3.0.1)

Installation details

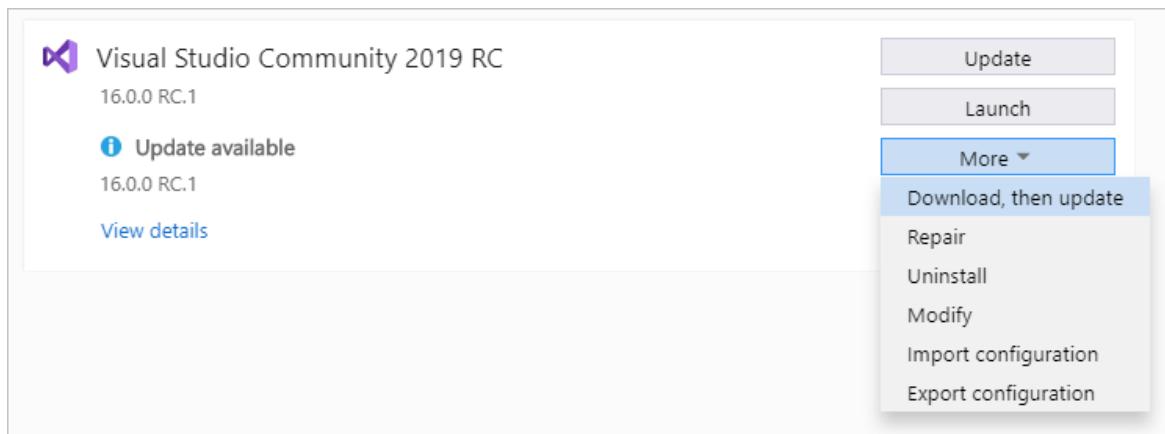
✓ Python development
Included
✓ Python language support
Optional
<input checked="" type="checkbox"/> Python miniconda
<input checked="" type="checkbox"/> Python web support
<input checked="" type="checkbox"/> Python 3 64-bit (3.7.2)
<input checked="" type="checkbox"/> Live Share - Preview
<input type="checkbox"/> Python native development tools
<input checked="" type="checkbox"/> Azure Cloud Services core tools
<input type="checkbox"/> Python 2 64-bit (2.7.15)
<input type="checkbox"/> Python 3 32-bit (3.7.2)
<input type="checkbox"/> Python 2 32-bit (2.7.15)

OPTION**DESCRIPTION**

OPTION	DESCRIPTION
Python distributions	<p>Choose any combination of the available options, such as 32-bit and 64-bit variants of the Python 2, Python 3, Miniconda, Anaconda2, and Anaconda3 distributions that you plan to work with. Each includes the distribution's interpreter, runtime, and libraries.</p> <p>Anaconda, specifically, is an open data science platform that includes a wide range of pre-installed packages. (You can return to the Visual Studio installer at any time to add or remove distributions.) Note: If you've installed a distribution outside of the Visual Studio installer, there's no need to check the equivalent option here. Visual Studio automatically detects existing Python installations. See The Python Environments window. Also, if a newer version of Python is available than what's shown in the installer, you can install that version separately and Visual Studio will detect it.</p>
Cookiecutter template support	<p>Installs the Cookiecutter graphical UI to discover templates, input template options, and create projects and files. See Use the Cookiecutter extension.</p>
Python web support	<p>Installs tools for web development including HTML, CSS, and JavaScript editing support, along with templates for projects using the Bottle, Flask, and Django frameworks. See Python web project templates.</p>
Python IoT support	<p>Supports Windows IoT Core development using Python.</p>
Python native development tools	<p>Installs the C++ compiler and other necessary components to develop native extensions for Python. See Create a C++ extension for Python. Also install the Desktop development with C++ workload for full C++ support.</p>
Azure Cloud Services core tools	<p>Provides additional support for developer Azure Cloud Services in Python. See Azure cloud service projects.</p>
OPTION	DESCRIPTION
Python distributions	<p>Choose any combination of the available options, such as 32-bit and 64-bit variants of the Python 2, Python 3, Miniconda, Anaconda2, and Anaconda3 distributions that you plan to work with. Each includes the distribution's interpreter, runtime, and libraries.</p> <p>Anaconda, specifically, is an open data science platform that includes a wide range of pre-installed packages. (You can return to the Visual Studio installer at any time to add or remove distributions.) Note: If you've installed a distribution outside of the Visual Studio installer, there's no need to check the equivalent option here. Visual Studio automatically detects existing Python installations. See The Python Environments window. Also, if a newer version of Python is available than what's shown in the installer, you can install that version separately and Visual Studio will detect it.</p>

OPTION	DESCRIPTION
Cookiecutter template support	Installs the Cookiecutter graphical UI to discover templates, input template options, and create projects and files. See Use the Cookiecutter extension .
Python web support	Installs tools for web development including HTML, CSS, and JavaScript editing support, along with templates for projects using the Bottle, Flask, and Django frameworks. See Python web project templates .
Python native development tools	Installs the C++ compiler and other necessary components to develop native extensions for Python. See Create a C++ extension for Python . Also install the Desktop development with C++ workload for full C++ support.
Azure Cloud Services core tools	Provides additional support for developer Azure Cloud Services in Python. See Azure cloud service projects .

4. After installation, the installer provides options to modify, launch, repair, or uninstall Visual Studio. The **Modify** button changes to **Update** when updates to Visual Studio are available for any installed components. (The **Modify** option is then available on the drop-down menu.) You can also launch Visual Studio and the installer from the Windows **Start** menu by searching on "Visual Studio".



Troubleshooting

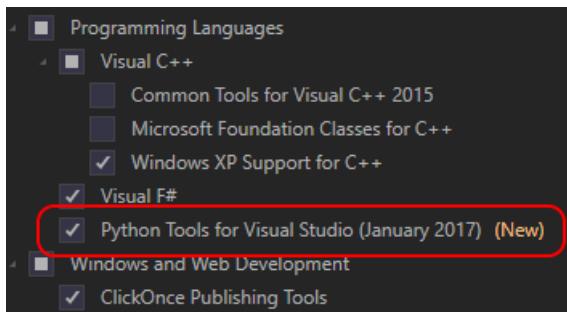
If you encounter problems installing or running Python in Visual Studio, try the following:

- Determine whether the same error occurs using the Python CLI, that is, running `python.exe` from a command prompt.
- Use the **Repair** option in the Visual Studio installer.
- Repair or reinstall Python through **Settings > Apps & features** in Windows.

Example error: Failed to start interactive process: System.ComponentModel.Win32Exception (0x80004005): Unknown error (0xc0000135) at Microsoft.PythonTools.Repl.PythonInteractiveEvaluator.d__43.MoveNext().

Visual Studio 2015

1. Run the Visual Studio installer through **Control Panel > Programs and Features**, selecting **Microsoft Visual Studio 2015** and then **Change**.
2. In the installer, select **Modify**.
3. Select **Programming Languages > Python Tools for Visual Studio** and then **Next**.



4. Once Visual Studio setup is complete, [install a Python interpreter of your choice](#). Visual Studio 2015 supports only Python 3.5 and earlier; later versions generate a message like **Unsupported Python version 3.6**). If you already have an interpreter installed and Visual Studio doesn't detect it automatically, see [Manually identify an existing environment](#).

Visual Studio 2013 and earlier

1. Install the appropriate version of Python Tools for Visual Studio for your version of Visual Studio:
 - Visual Studio 2013: [PTVS 2.2 for Visual Studio 2013](#). The **File > New Project** dialog in Visual Studio 2013 gives you a shortcut for this process.
 - Visual Studio 2012: [PTVS 2.1 for Visual Studio 2012](#)
 - Visual Studio 2010: [PTVS 2.1 for Visual Studio 2010](#)
2. [Install a Python interpreter of your choice](#). If you already have an interpreter installed and Visual Studio doesn't detect it automatically, see [Manually identify an existing environment](#).

Install locations

By default, Python support is installed for all users on a computer.

For Visual Studio 2019 and Visual Studio 2017, the Python workload is installed in
`%ProgramFiles(x86)%\Microsoft Visual Studio\<VS_version>\<VS_edition>\Common7\IDE\Extensions\Microsoft\Python` where `<VS_version>` is 2019 or 2017 and
`<VS_edition>` is Community, Professional, or Enterprise.

For Visual Studio 2015 and earlier, installation paths are as follows:

- 32-bit:
 - Path: `%Program Files(x86)%\Microsoft Visual Studio <VS_ver>\Common7\IDE\Extensions\Microsoft\Python Tools for Visual Studio\<PTVS_ver>`
 - Registry location of path: `HKEY_LOCAL_MACHINE\Software\Microsoft\PythonTools\<VS_ver>\InstallDir`
- 64-bit:
 - Path: `%Program Files%\Microsoft Visual Studio <VS_ver>\Common7\IDE\Extensions\Microsoft\Python Tools for Visual Studio\<PTVS_ver>`
 - Registry location of path:
`HKEY_LOCAL_MACHINE\Software\Wow6432Node\Microsoft\PythonTools\<VS_ver>\InstallDir`

where:

- `<VS_ver>` is:
 - 14.0 for Visual Studio 2015
 - 12.0 for Visual Studio 2013
 - 11.0 for Visual Studio 2012

- 10.0 for Visual Studio 2010
- <PTVS_ver> is a version number, such as 2.2, 2.1, 2.0, 1.5, 1.1, or 1.0.

User-specific installations (1.5 and earlier)

Python Tools for Visual Studio 1.5 and earlier allowed installation for the current user only, in which case the installation path is %LocalAppData%\Microsoft\VisualStudio\<VS_ver>\Extensions\Microsoft\Python Tools for Visual Studio\<PTVS_ver> where <VS_ver> and <PTVS_ver> are the same as described above.

Install Python interpreters

6/13/2019 • 3 minutes to read • [Edit Online](#)

By default, installing the Python development workload in Visual Studio 2017 and later also installs Python 3 (64-bit). You can optionally choose to install 32-bit and 64-bit versions of Python 2 and Python 3, along with Miniconda (Visual Studio 2019) or Anaconda 2/Anaconda 3 (Visual Studio 2017), as described in [Installation](#).

Alternately, you can install standard python interpreters from the **Add Environment** dialog. Select the **Add Environment** command in the **Python Environments** window or the Python toolbar, select the **Python installation** tab, indicate which interpreters to install, and select **Install**.

You can also manually install any of the interpreters listed in the table below outside of the Visual Studio installer. For example, if you installed Anaconda 3 before installing Visual Studio, you don't need to install it again through the Visual Studio installer. You can also install an interpreter manually if, for example, a newer version of available that doesn't yet appear in the Visual Studio installer.

NOTE

Visual Studio supports Python version 2.7, as well as version 3.5 and greater. While it is possible to use Visual Studio to edit code written in other versions of Python, those versions are not officially supported and features such as IntelliSense and debugging might not work.

For **Visual Studio 2015 and earlier**, you must manually install one of the interpreters.

Visual Studio (all versions) automatically detects each installed Python interpreter and its environment by checking the registry according to [PEP 514 - Python registration in the Windows registry](#). Python installations are typically found under **HKEY_LOCAL_MACHINE\SOFTWARE\Python** (32-bit) and **HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Python** (64-bit), then within nodes for the distribution such as **PythonCore** (CPython) and **ContinuumAnalytics** (Anaconda).

If Visual Studio does not detect an installed environment, see [Manually identify an existing environment](#).

Visual Studio shows all known environments in the **Python Environments** window, and automatically detects updates to existing interpreters.

INTERPRETER	DESCRIPTION
CPython	The "native" and most commonly-used interpreter, available in 32-bit and 64-bit versions (32-bit recommended). Includes the latest language features, maximum Python package compatibility, full debugging support, and interop with IPython . See also: Should I use Python 2 or Python 3? . Note that Visual Studio 2015 and earlier do not support Python 3.6+ and can give errors like Unsupported python version 3.6 . Use Python 3.5 or earlier instead.
IronPython	A .NET implementation of Python, available in 32-bit and 64-bit versions, providing C#/F#/Visual Basic interop, access to .NET APIs, standard Python debugging (but not C++ mixed-mode debugging), and mixed IronPython/C# debugging. IronPython, however, does not support virtual environments.

INTERPRETER	DESCRIPTION
Anaconda	An open data science platform powered by Python, and includes the latest version of CPython and most of the difficult-to-install packages. We recommend it if you can't otherwise decide.
PyPy	A high-performance tracing JIT implementation of Python that's good for long-running programs and situations where you identify performance issues but cannot find other resolutions. Works with Visual Studio but with limited support for advanced debugging features.
Jython	An implementation of Python on the Java Virtual Machine (JVM). Similar to IronPython, code running in Jython can interact with Java classes and libraries, but may not be able to use many libraries intended for CPython. Works with Visual Studio but with limited support for advanced debugging features.

Developers that want to provide new forms of detection for Python environments, see [PTVS Environment Detection](#) (github.com).

Move an interpreter

If you move an existing interpreter to a new location using the file system, Visual Studio doesn't automatically detect the change.

- If you originally specified the location of the interpreter through the **Python Environments** window, then edit its environment using the **Configure** tab in that window to identify the new location. See [Manually identify an existing environment](#).
- If you installed the interpreter using an installer program, then use the following steps to reinstall the interpreter in the new location:
 1. Restore the Python interpreter to its original location.
 2. Uninstall the interpreter using its installer, which clears the registry entries.
 3. Reinstall the interpreter at the desired location.
 4. Restart Visual Studio, which should auto-detect the new location in place of the old location.

Following this process ensures that the registry entries that identify the interpreter's location, which Visual Studio uses, are properly updated. Using an installer also handles any other side effects that may exist.

See also

- [Manage Python environments](#)
- [Select an interpreter for a project](#)
- [Use requirements.txt for dependencies](#)
- [Search paths](#)
- [Python Environments window reference](#)

Quickstart: Create your first Python web app using Visual Studio

6/25/2019 • 7 minutes to read • [Edit Online](#)

In this 5-10 minute introduction to Visual Studio as a Python IDE, you create a simple Python web application based on the Flask framework. You create the project through discrete steps that help you learn about Visual Studio's basic features.

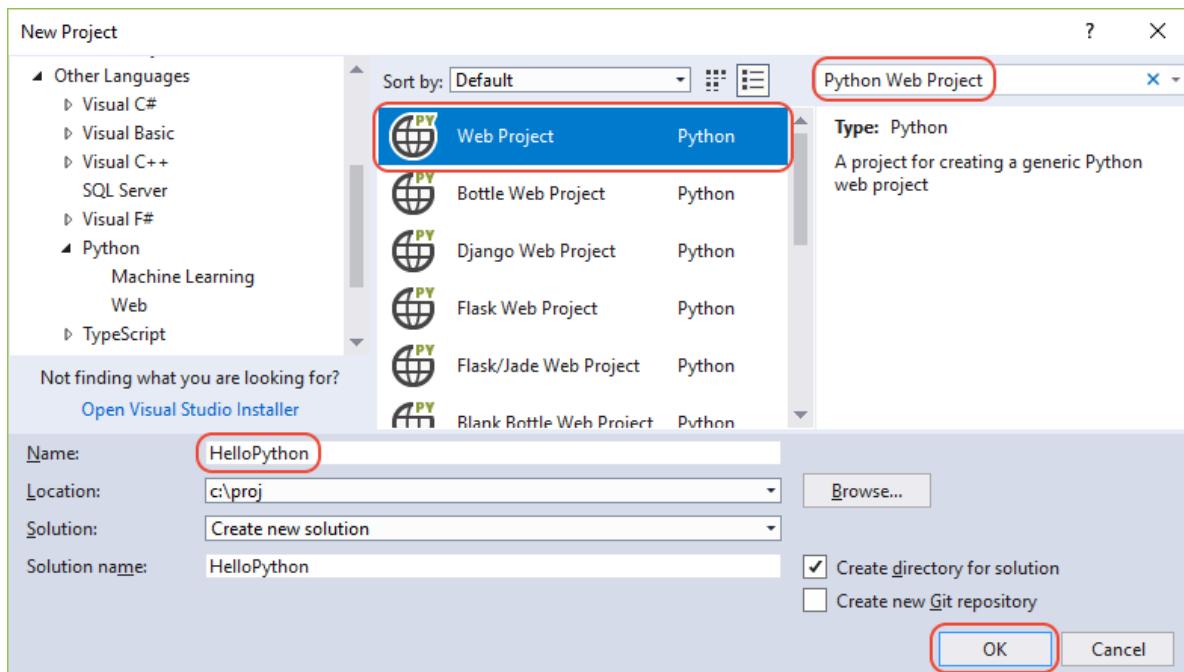
If you haven't already installed Visual Studio, go to the [Visual Studio downloads](#) page to install it for free. In the installer, make sure to select the **Python development** workload.

If you haven't already installed Visual Studio, go to the [Visual Studio downloads](#) page to install it for free. In the installer, make sure to select the **Python development** workload.

Create the project

The following steps create an empty project that serves as a container for the application:

1. Open Visual Studio 2017.
2. From the top menu bar, choose **File > New > Project**.
3. In the **New Project** dialog box, enter "Python Web Project" in the search field on the upper right, choose **Web project** in the middle list, give the project a name like "HelloPython", then choose **OK**.

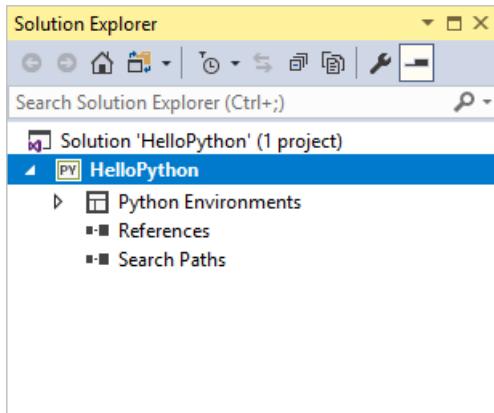


If you don't see the Python project templates, run the **Visual Studio Installer**, select **More > Modify**, select the **Python development** workload, then choose **Modify**.

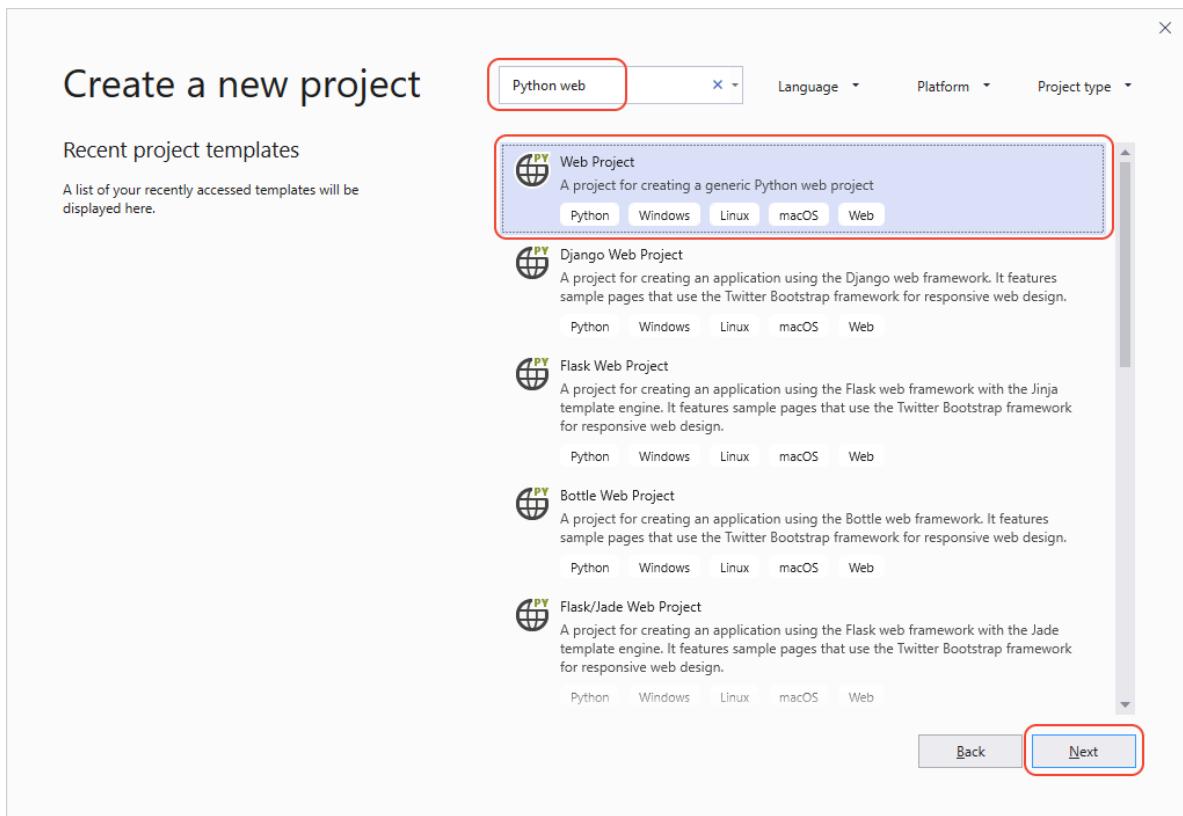


4. The new project opens in **Solution Explorer** in the right pane. The project is empty at this point because it

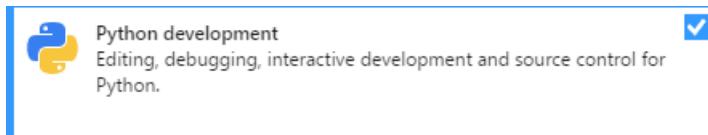
contains no other files.



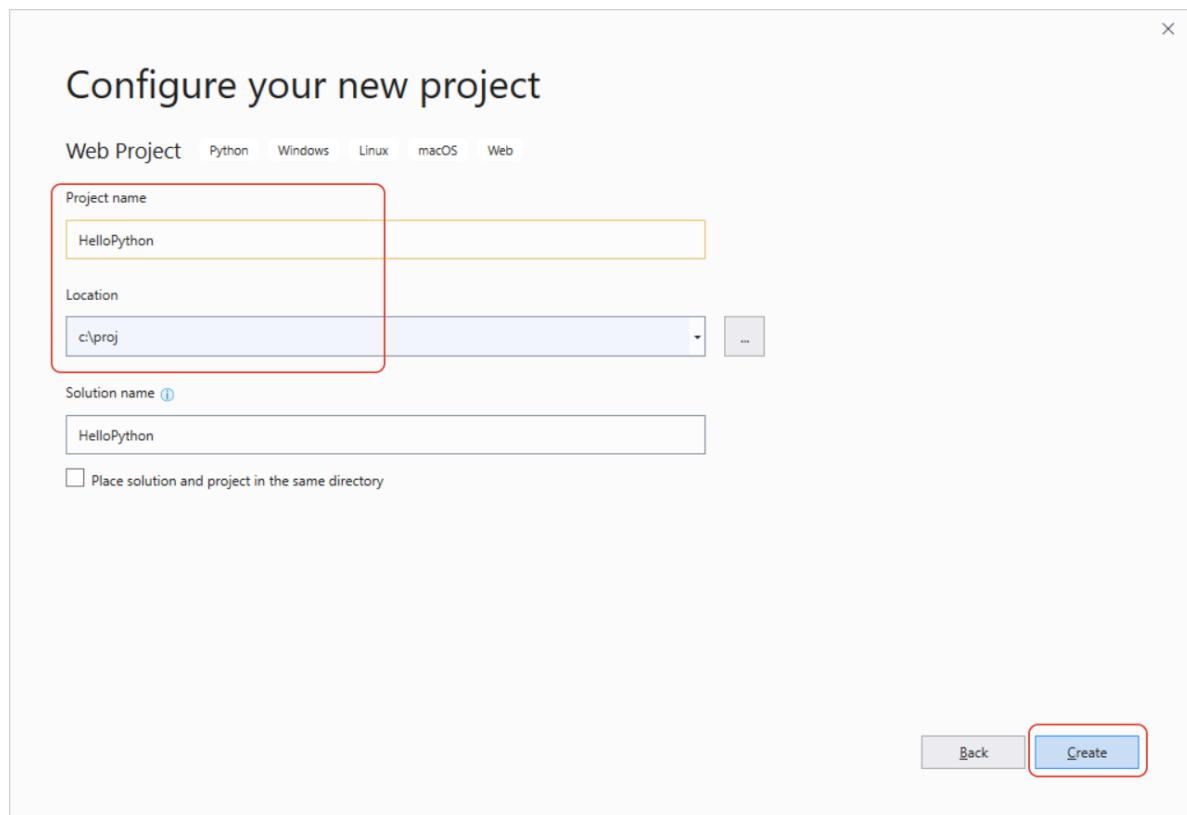
1. Open Visual Studio 2019.
2. On the start screen, select **Create a new project**.
3. In the **Create a new project** dialog box, enter "Python web" in the search field at the top, choose **Web Project** in the middle list, then select **Next**:



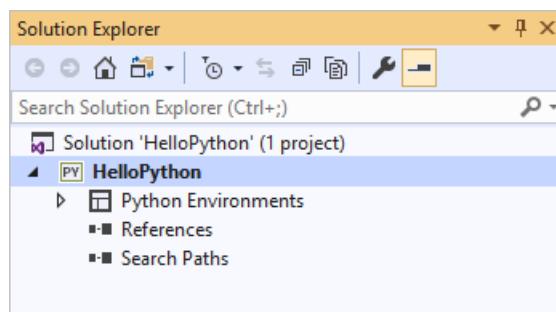
If you don't see the Python project templates, run the **Visual Studio Installer**, select **More > Modify**, select the **Python development** workload, then choose **Modify**.



4. In the **Configure your new project** dialog that follows, enter "HelloPython" for **Project name**, specify a location, and select **Create**. (The **Solution name** is automatically set to match the **Project name**.)



5. The new project opens in **Solution Explorer** in the right pane. The project is empty at this point because it contains no other files.



Question: What's the advantage of creating a project in Visual Studio for a Python application?

Answer: Python applications are typically defined using only folders and files, but this simple structure can become burdensome as applications become larger and perhaps involve auto-generated files, JavaScript for web applications, and so on. A Visual Studio project helps manage this complexity. The project (a `.pyproj` file) identifies all the source and content files associated with your project, contains build information for each file, maintains the information to integrate with source-control systems, and helps you organize your application into logical components.

Question: What is the "solution" shown in Solution Explorer?

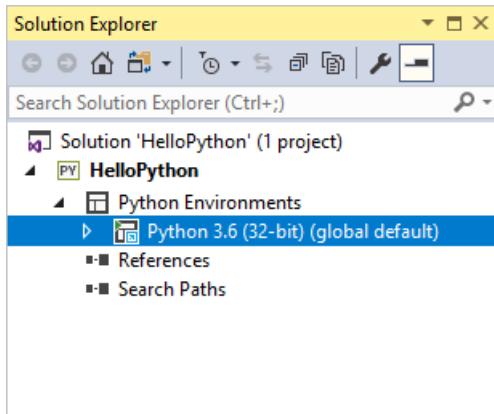
Answer: A Visual Studio solution is a container that helps you manage for one or more related projects as a group, and stores configuration settings that aren't specific to a project. Projects in a solution can also reference one another, such that running one project (a Python app) automatically builds a second project (such as a C++ extension used in the Python app).

Install the Flask library

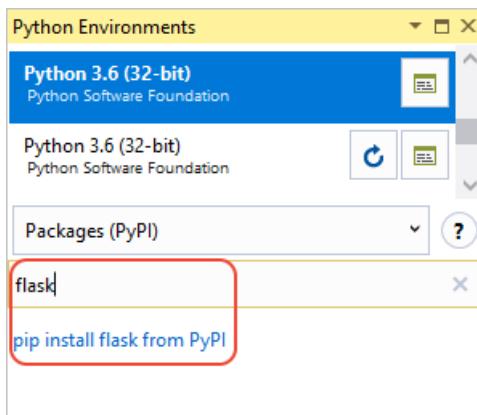
Web apps in Python almost always use one of the many available Python libraries to handle low-level details like routing web requests and shaping responses. For this purpose, Visual Studio provides a variety of templates for web apps, one of which you use later in this Quickstart.

Here, you use the following steps to install the Flask library into the default "global environment" that Visual Studio uses for this project.

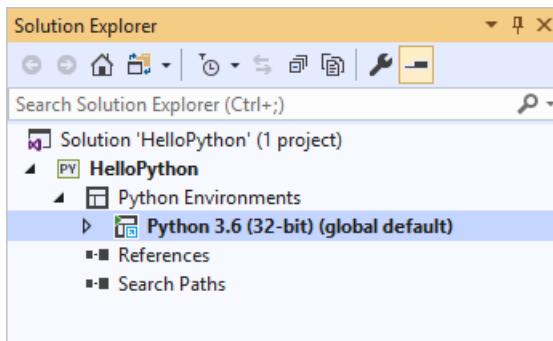
1. Expand the **Python Environments** node in the project to see the default environment for the project.



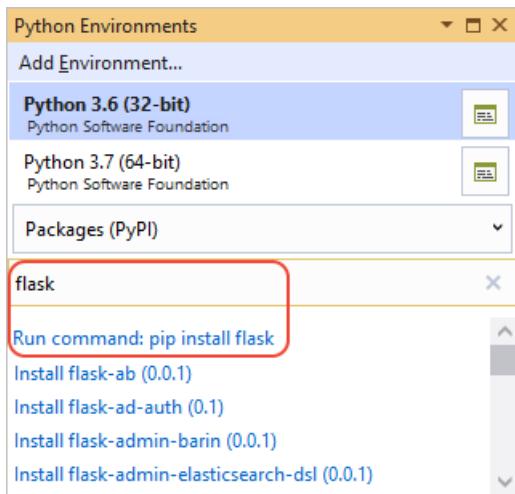
2. Right-click the environment and select **Install Python Package**. This command opens the **Python Environments** window on the **Packages** tab.
3. Enter "flask" in the search field and select **pip install flask from PyPI**. Accept any prompts for administrator privileges and observe the **Output** window in Visual Studio for progress. (A prompt for elevation happens when the packages folder for the global environment is located within a protected area like *C:\Program Files*.)



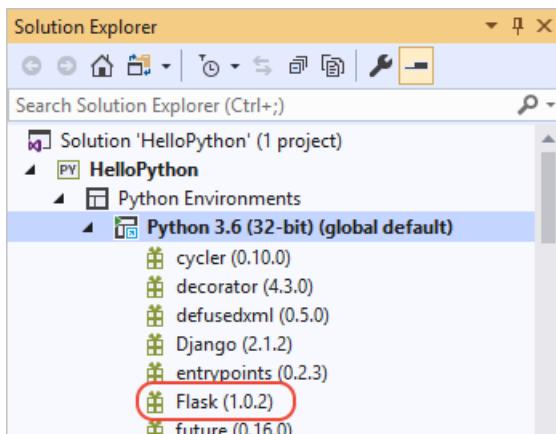
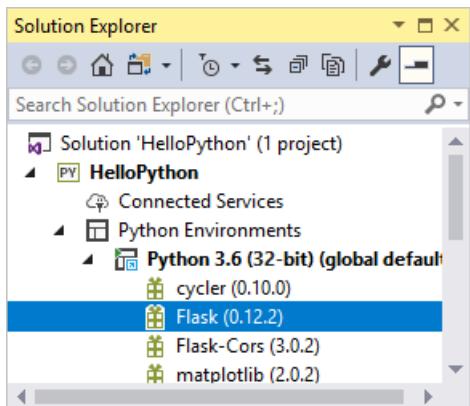
1. Expand the **Python Environments** node in the project to see the default environment for the project.



2. Right-click the environment and select **Manage Python Packages...**. This command opens the **Python Environments** window on the **Packages (PyPI)** tab.
3. Enter "flask" in the search field. If **Flask** appears below the search box, you can skip this step. Otherwise select **Run command: pip install flask**. Accept any prompts for administrator privileges and observe the **Output** window in Visual Studio for progress. (A prompt for elevation happens when the packages folder for the global environment is located within a protected area like *C:\Program Files*.)



- Once installed, the library appears in the environment in **Solution Explorer**, which means that you can make use of it in Python code.



NOTE

Instead of installing libraries in the global environment, developers typically create a "virtual environment" in which to install libraries for a specific project. Visual Studio templates typically offer this option, as discussed in [Quickstart - Create a Python project using a template](#).

Question: Where do I learn more about other available Python packages?

Answer: Visit the [Python Package Index](#).

Add a code file

You're now ready to add a bit of Python code to implement a minimal web app.

1. Right-click the project in **Solution Explorer** and select **Add > New Item**.
2. In the dialog that appears, select **Empty Python File**, name it *app.py*, and select **Add**. Visual Studio automatically opens the file in an editor window.
3. Copy the following code and paste it into *app.py*:

```
from flask import Flask

# Create an instance of the Flask class that is the WSGI application.
# The first argument is the name of the application module or package,
# typically __name__ when using a single module.
app = Flask(__name__)

# Flask route decorators map / and /hello to the hello function.
# To add other resources, create functions that generate the page contents
# and add decorators to define the appropriate resource locators for them.

@app.route('/')
@app.route('/hello')
def hello():
    # Render the page
    return "Hello Python!"

if __name__ == '__main__':
    # Run the app server on localhost:4449
    app.run('localhost', 4449)
```

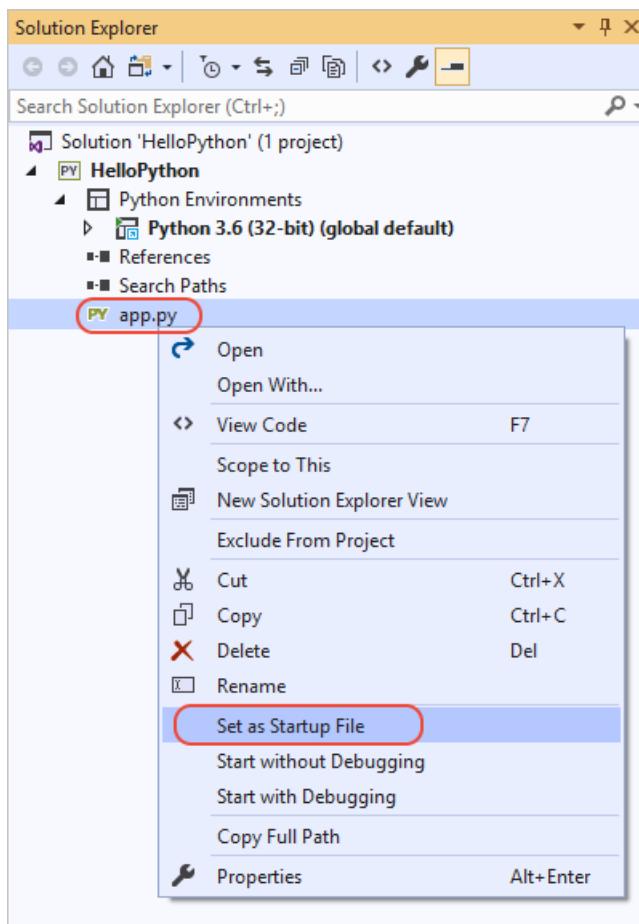
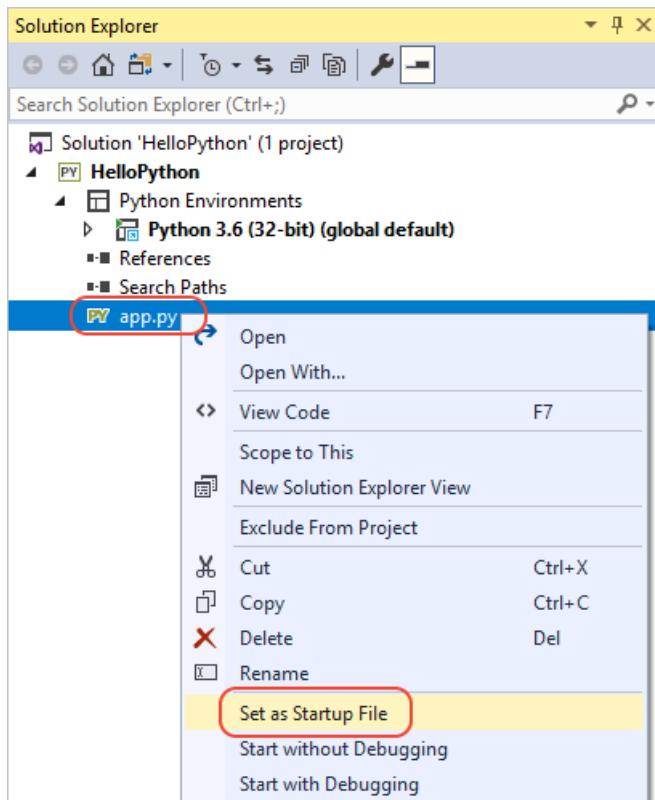
4. You may have noticed that the **Add > New Item** dialog box displays many other types of files you can add to a Python project, including a Python class, a Python package, a Python unit test, *web.config* files, and more. In general, these item templates, as they're called, are a great way to quickly create files with useful boilerplate code.

Question: Where can I learn more about Flask?

Answer: Refer to the Flask documentation, starting with the [Flask Quickstart](#).

Run the application

1. Right-click *app.py* in **Solution Explorer** and select **Set as startup file**. This command identifies the code file to launch in Python when running the app.



2. Right-click the project in **Solution Explorer** and select **Properties**. Then select the **Debug** tab and set the **Port Number** property to `4449`. This step ensures that Visual Studio launches a browser with `localhost:4449` to match the `app.run` arguments in the code.
3. Select **Debug > Start Without Debugging (Ctrl+F5)**, which saves changes to files and runs the app.
4. A command window appears with the message "`* Running in https://localhost:4449/`", and a browser window should open to `localhost:4449` where you see the message, "Hello, Python!" The GET request

also appears in the command window with a status of 200.

If a browser does not open automatically, start the browser of your choice and navigate to `localhost:4449`.

If you see only the Python interactive shell in the command window, or if that window flashes on the screen briefly, ensure that you set `app.py` as the startup file in step 1 above.

5. Navigate to `localhost:4449/hello` to test that the decorator for the `/hello` resource also works. Again, the GET request appears in the command window with a status of 200. Feel free to try some other URL as well to see that they show 404 status codes in the command window.
6. Close the command window to stop the app, then close the browser window.

Question: What's the difference between the Start Without Debugging command and Start Debugging?

Answer: You use **Start Debugging** to run the app in the context of the [Visual Studio debugger](#), allowing you to set breakpoints, examine variables, and step through your code line by line. Apps may run slower in the debugger because of the various hooks that make debugging possible. **Start Without Debugging**, in contrast, runs the app directly as if you ran it from the command line, with no debugging context, and also automatically launches a browser and navigates to the URL specified in the project properties' **Debug** tab.

Next steps

Congratulations on running your first Python app from Visual Studio, in which you've learned a little about using Visual Studio as a Python IDE!

[Deploy the app to Azure App Service](#)

Because the steps you followed in this Quickstart are fairly generic, you've probably guessed that they can and should be automated. Such automation is the role of Visual Studio project templates. Go through [Quickstart - Create a Python project using a template](#) for a demonstration that creates a web app similar to the one you created in this article, but with fewer steps.

To continue with a fuller tutorial on Python in Visual Studio, including using the interactive window, debugging, data visualization, and working with Git, go through [Tutorial: Get started with Python in Visual Studio](#).

To explore more that Visual Studio has to offer, select the links below.

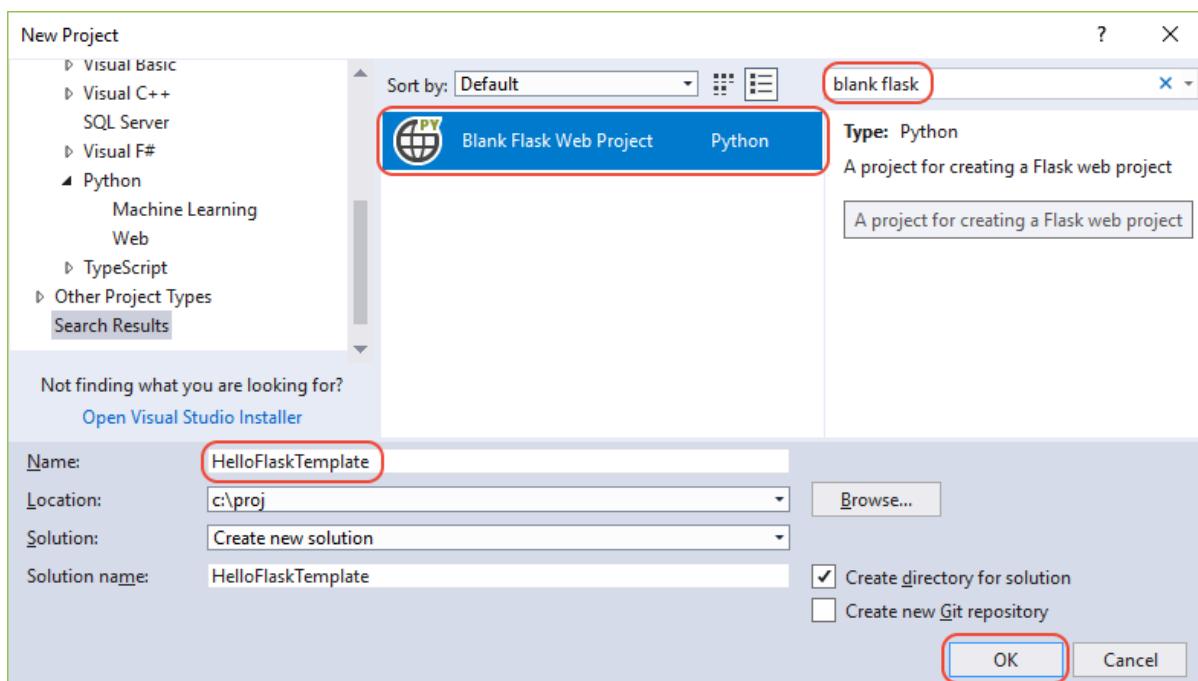
- Learn about [Python web app templates in Visual Studio](#).
- Learn about [Python debugging](#)
- Learn more about the [Visual Studio IDE](#) in general.

Quickstart: Create a Python project from a template in Visual Studio

4/9/2019 • 3 minutes to read • [Edit Online](#)

Once you've [installed Python support in Visual Studio](#), it's easy to create a new Python project using a variety of templates. In this Quickstart, you create a simple Flask app using a template. The resulting project is similar to the project you create manually through [Quickstart - Create a web app with Flask](#).

1. Start Visual Studio.
2. From the top menu bar, choose **File > New > Project**, then in the **New Project** dialog search for "blank flask", select the **Blank Flask Web Project** template in the middle list, give the project a name, and select **OK**:



3. Visual Studio prompts you with a dialog that says **This project requires external packages**. This dialog appears because the template includes a `requirements.txt` file specifying a dependency on Flask. Visual Studio can install the packages automatically, and gives you the option to install them into a *virtual environment*. Using a virtual environment is recommended over installing into a global environment, so select **Install into a virtual environment** to continue.

This project requires external packages

We can download and install these packages for you automatically, but we need to know whether you want to install them for just this project or for all your projects.

→ [Install into a virtual environment...](#)

Packages will only be available for this project (recommended)

→ [Install into Python 3.6 \(32-bit\)](#)

Packages will be shared by all projects

→ [I will install them myself](#)

[Hide required packages](#)

flask

4. Visual Studio displays the **Add Virtual Environment** dialog. Accept the default and select **Create**, then consent to any elevation requests.

TIP

When you begin a project, it's highly recommended to create a virtual environment right away, as most Visual Studio templates invite you to do. Virtual environments maintain your project's exact requirements over time as you add and remove libraries. You can then easily generate a *requirements.txt* file, which you use to reinstall those dependencies on other development computers (as when using source control) and when deploying the project to a production server. For more information on virtual environments and their benefits, see [Use virtual environments](#) and [Manage required packages with requirements.txt](#).

5. After Visual Studio creates that environment, look in **Solution Explorer** to see that you have an *app.py* file along with *requirements.txt*. Open *app.py* to see that the template has provided code like that in [Quickstart - Create a web app with Flask](#), with a few added sections. All of the code shown below is created by the template, so you don't need to paste any into *app.py* yourself.

The code begins with the necessary imports:

```
from flask import Flask
app = Flask(__name__)
```

Next is the following line that can be helpful when deploying an app to a web host:

```
wsgi_app = app.wsgi_app
```

Then comes route decorator on a simple function that defines a view:

```
@app.route('/')
def hello():
    """Renders a sample page."""
    return "Hello World!"
```

Finally, the startup code below allows you to set the host and port through environment variables rather than hard-coding them. Such code allows you to easily control the configuration on both development and production machines without changing the code:

```
if __name__ == '__main__':
    import os
    HOST = os.environ.get('SERVER_HOST', 'localhost')
    try:
        PORT = int(os.environ.get('SERVER_PORT', '5555'))
    except ValueError:
        PORT = 5555
    app.run(HOST, PORT)
```

6. Select **Debug > Start without Debugging** to run the app and open a browser to `localhost:5555`.

Question: What other Python templates does Visual Studio offer?

Answer: With the Python workload installed, Visual Studio provides a variety of project templates including ones for the [Flask, Bottle, and Django web frameworks](#), Azure cloud services, different machine learning scenarios, and even a template to create a project from an existing folder structure containing a Python app. You access these through the **File > New > Project** dialog box by selecting the **Python** language node and its child nodes.

Visual Studio also provides a variety of file or *item templates* to quickly create a Python class, a Python package, a Python unit test, `web.config` files, and more. When you have a Python project open, you access item templates through the **Project > Add New Item** menu command. See the [item templates](#) reference.

Using templates can save you significant time when starting a project or creating a file, and are also a great way to learn about different app types and code structures. It's helpful to take a few minutes to create projects and items from the various templates to familiarize yourself with what they offer.

Question: Can I also use Cookiecutter templates?

Answer: Yes! In fact, Visual Studio provides direct integration with Cookiecutter, which you can learn about through [Quickstart: Create a project from a Cookiecutter template](#).

Next steps

[Tutorial: Work with Python in Visual Studio](#)

See also

- [Manually identify an existing Python interpreter](#)
- [Install Python support in Visual Studio 2015 and earlier](#)
- [Install locations](#)

Quickstart: Open and run Python code in a folder

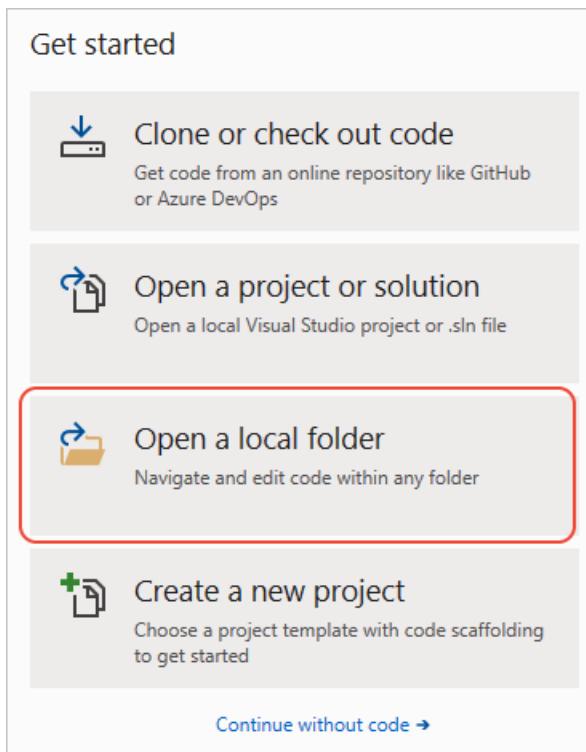
4/9/2019 • 3 minutes to read • [Edit Online](#)

Once you've [installed Python support in Visual Studio 2019](#), it's easy to run existing Python code in Visual Studio 2019 without creating a Visual Studio project.

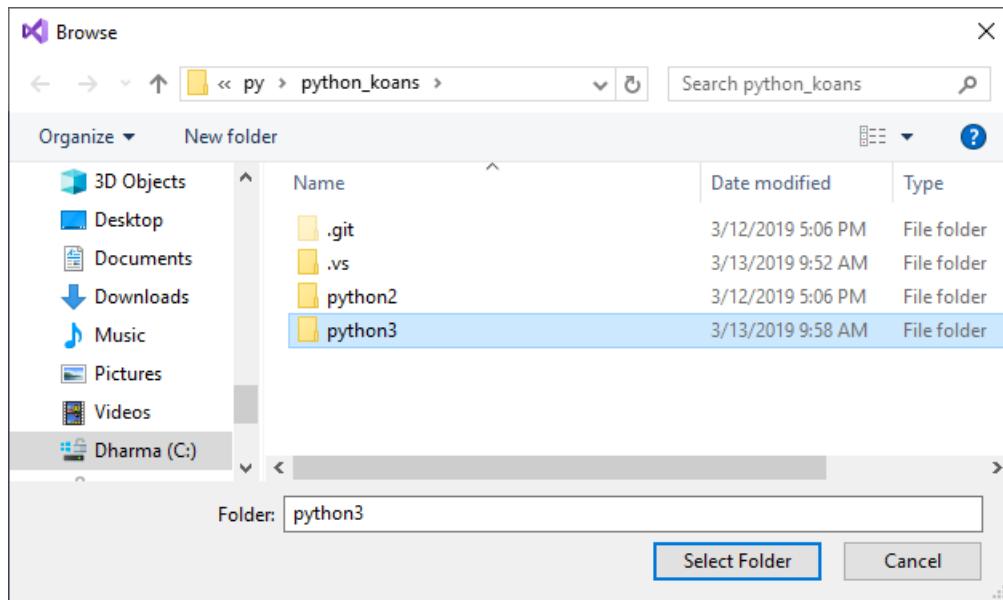
NOTE

Visual Studio 2017 and earlier require you to create a Visual Studio project to run Python code, which you can easily do using a built-in project template. See [Quickstart: Create a Python project from existing code](#)

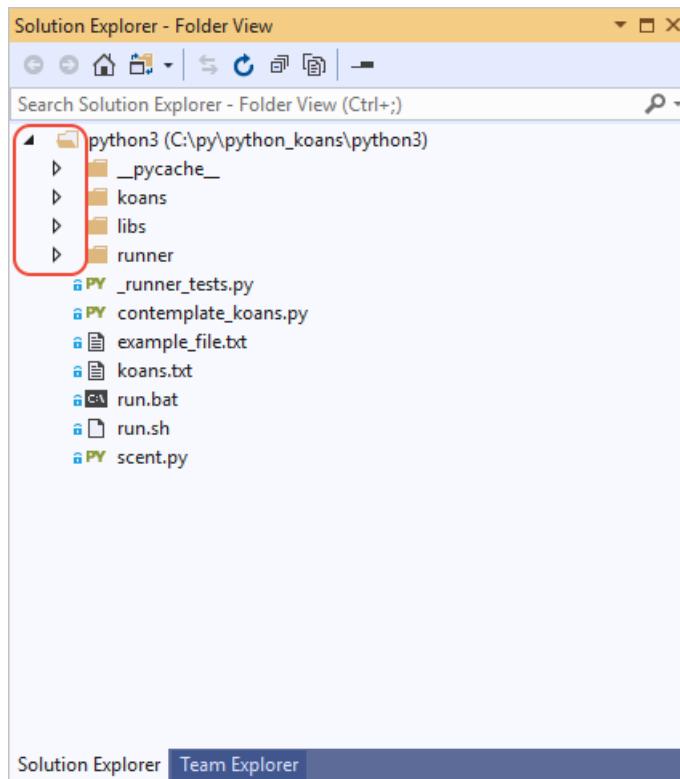
1. For this walkthrough, you can use any folder with Python code that you like. To follow along with the example shown here, clone the gregmalcolm/python_koans GitHub repository to your computer using the command `git clone https://github.com/gregmalcolm/python_koans` in an appropriate folder.
2. Launch Visual Studio 2019 and in the start window, select **Open** at the bottom of the **Get started** column. Alternately, if you already have Visual Studio running, select the **File > Open > Folder** command instead.



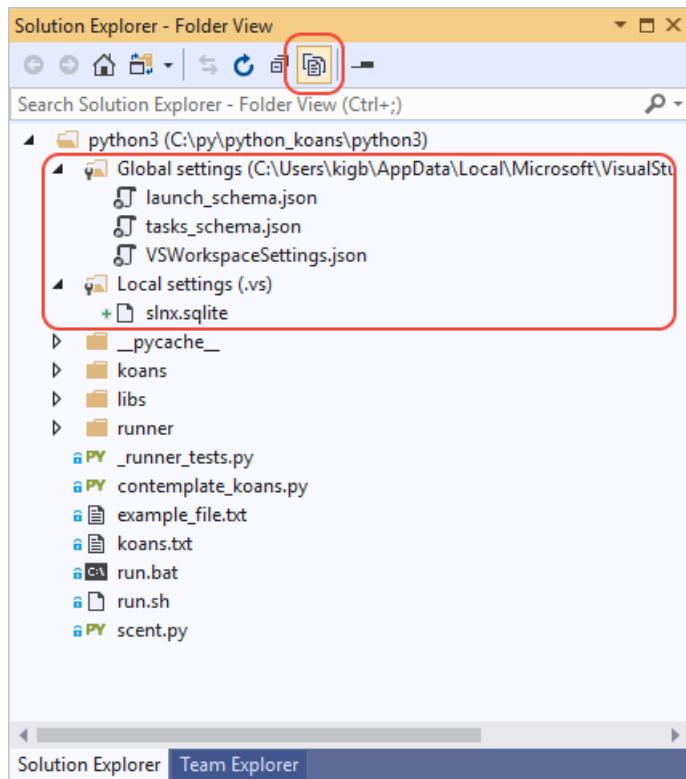
3. Navigate to the folder containing your Python code, then choose **Select Folder**. If you're using the python_koans code, make sure to select the `python3` folder within the clone folder.



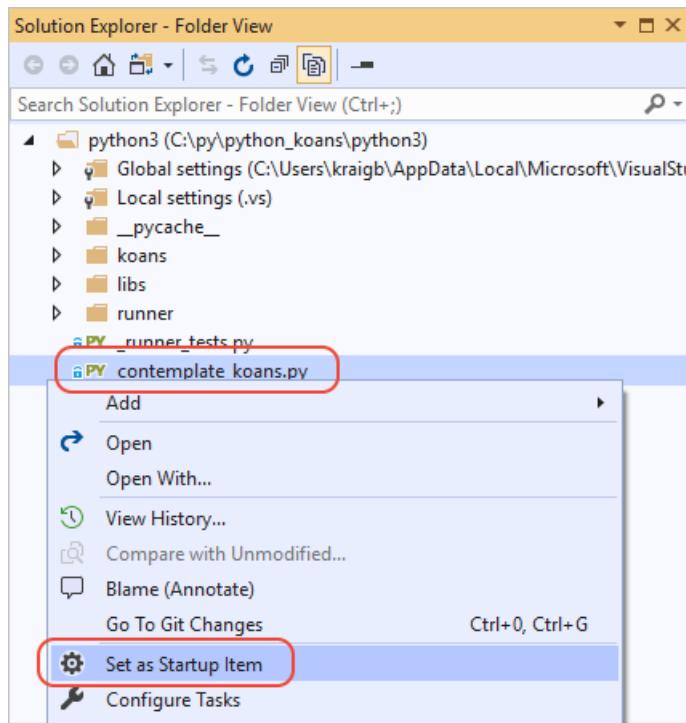
4. Visual Studio displays the folder in **Solution Explorer** in what's called **Folder View**. You can expand and collapse folders using the arrows on the left edges of the folder names:



5. When opening a Python folder, Visual Studio creates several hidden folders to manage settings related to the project. To see these folders (and any other hidden files and folders, such as the `.git` folder), select the **Show All Files** toolbar button:



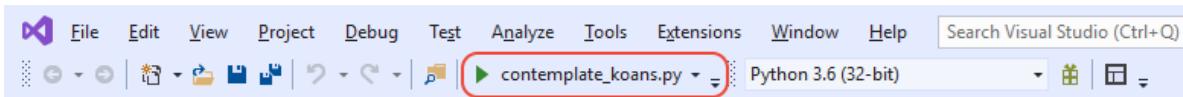
6. To run the code, you first need to identify the startup or primary program file. In the example shown here, the startup file *contemplate-koans.py*. Right-click that file and select **Set as Startup Item**.



IMPORTANT

If your startup item is not located in the root of the folder you opened, you must also add a line to the launch configuration JSON file as described in the section, [Set a working directory](#).

7. Run the code by pressing **Ctrl+F5** or selecting **Debug > Start without Debugging**. You can also select the toolbar button that shows the startup item with a play button, which runs code in the Visual Studio debugger. In all cases, Visual Studio detects that your startup item is a Python file, so it automatically runs the code in the default Python environment. (That environment is shown to the right of the startup item on the toolbar.)



8. The program's output appears in a separate command window:

```
C:\Python36-32\python.exe

Thinking AboutAsserts
test_assert_truth has damaged your karma.

You have not yet reached enlightenment ...
AssertionError: False is not true

Please meditate on the following code:
File "D:\py\python_koans\python3\koans\about_asserts.py", line 17, in test_assert_truth
    self.assertTrue(False) # This should be True

You have completed 0 (0 %) koans and 0 (out of 37) lessons.
You are now 302 koans and 37 lessons away from reaching enlightenment.

Beautiful is better than ugly.
```

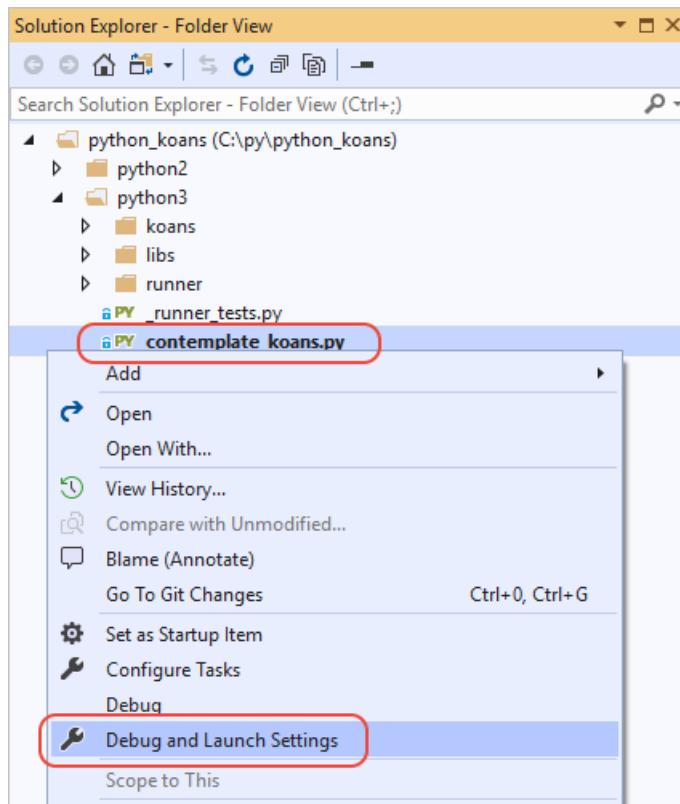
9. To run the code in a different environment, select that environment from the drop-down control on the toolbar, then launch the startup item again.
10. To close the folder in Visual Studio, select the **File > Close folder** menu command.

Set a working directory

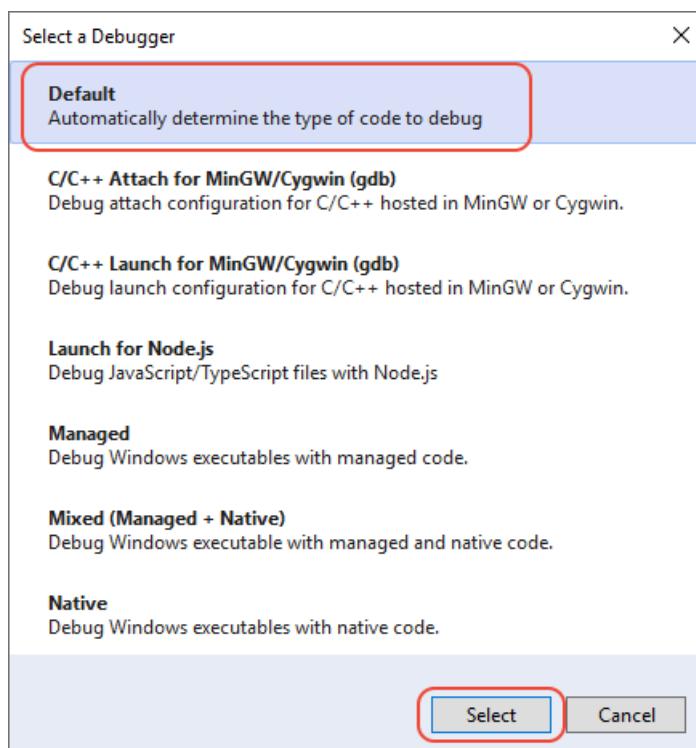
By default, Visual Studio runs a Python project opened as a folder in the root of that same folder. The code in your project, however, might assume that Python is being run in a subfolder. For example, suppose you open the root folder of the `python_koans` repository and then set the `python3/contemplate-koans.py` file as startup item. If you then run the code, you see an error that the `koans.txt` file cannot be found. This error happens because `contemplate-koans.py` assumes that Python is being run in the `python3` folder rather than the repository root.

In such cases, you must also add a line to the launch configuration JSON file to specify the working directory:

1. Right-click the Python (.py) startup file in **Solution Explorer** and select **Debug and Launch Settings**.



2. In the **Select debugger** dialog box that appears, select **Default** and then choose **Select**.



NOTE

If you don't see **Default** as a choice, be sure that you right-clicked a Python .py file when selecting the **Debug and Launch Settings** command. Visual Studio uses the file type to determine while debugger options to display.

3. Visual Studio opens a file named *launch.json*, which is located in the hidden .vs folder. This file describes the debugging context for the project. To specify a working directory, add a value for `"workingDirectory"`, as in `"workingDirectory": "python3"` for python-koans example:

```
{  
  "version": "0.2.1",  
  "defaults": {},  
  "configurations": [  
    {  
      "type": "python",  
      "interpreter": "(default)",  
      "interpreterArguments": "",  
      "scriptArguments": "",  
      "env": {},  
      "nativeDebug": false,  
      "webBrowserUrl": "",  
      "project": "python3\\contemplate_koans.py",  
      "name": "contemplate_koans.py",  
      "workingDirectory": "python3"  
    }  
  ]  
}
```

4. Save the file and launch the program again, which now runs in the specified folder.

Next steps

[Tutorial: Work with Python in Visual Studio](#)

See also

- [Quickstart: Create a Python project from existing code](#)
- [Quickstart: Create a Python project from a repository](#)
- [Manually identify an existing Python interpreter](#)

Quickstart: Create a Python project from existing code

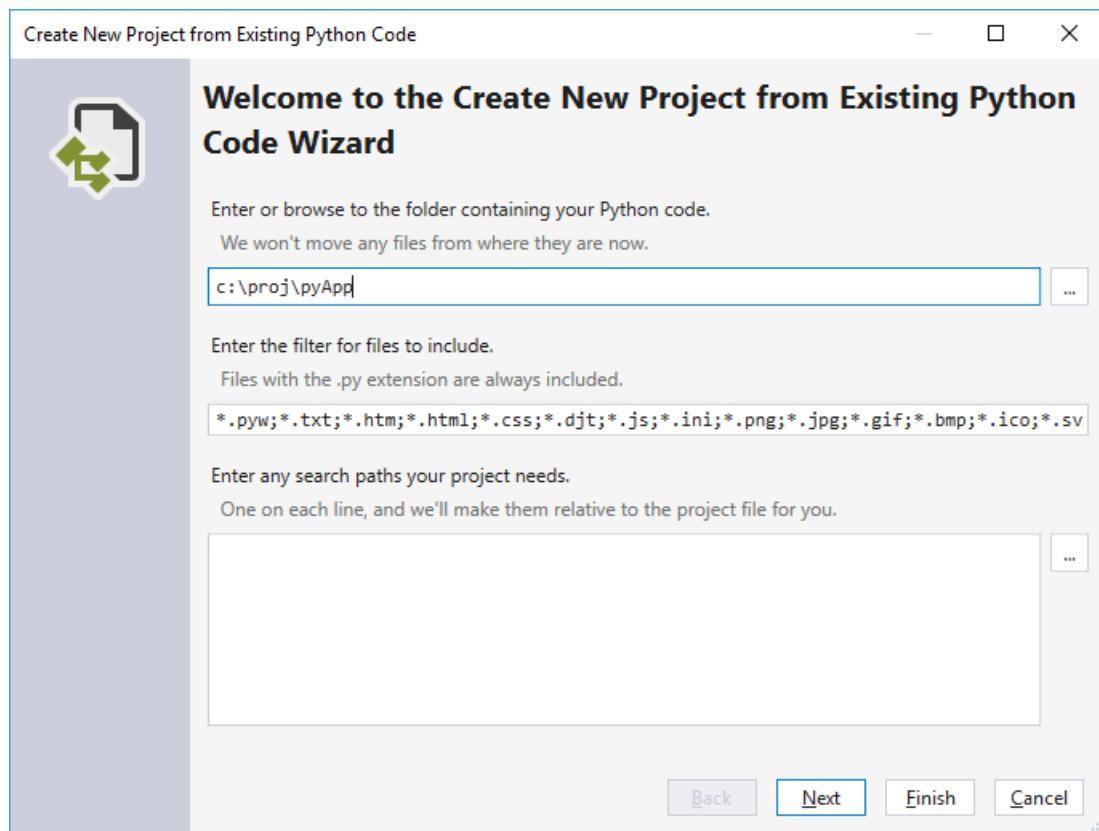
4/9/2019 • 2 minutes to read • [Edit Online](#)

Once you've [installed Python support in Visual Studio](#), it's easy to bring existing Python code into a Visual Studio project.

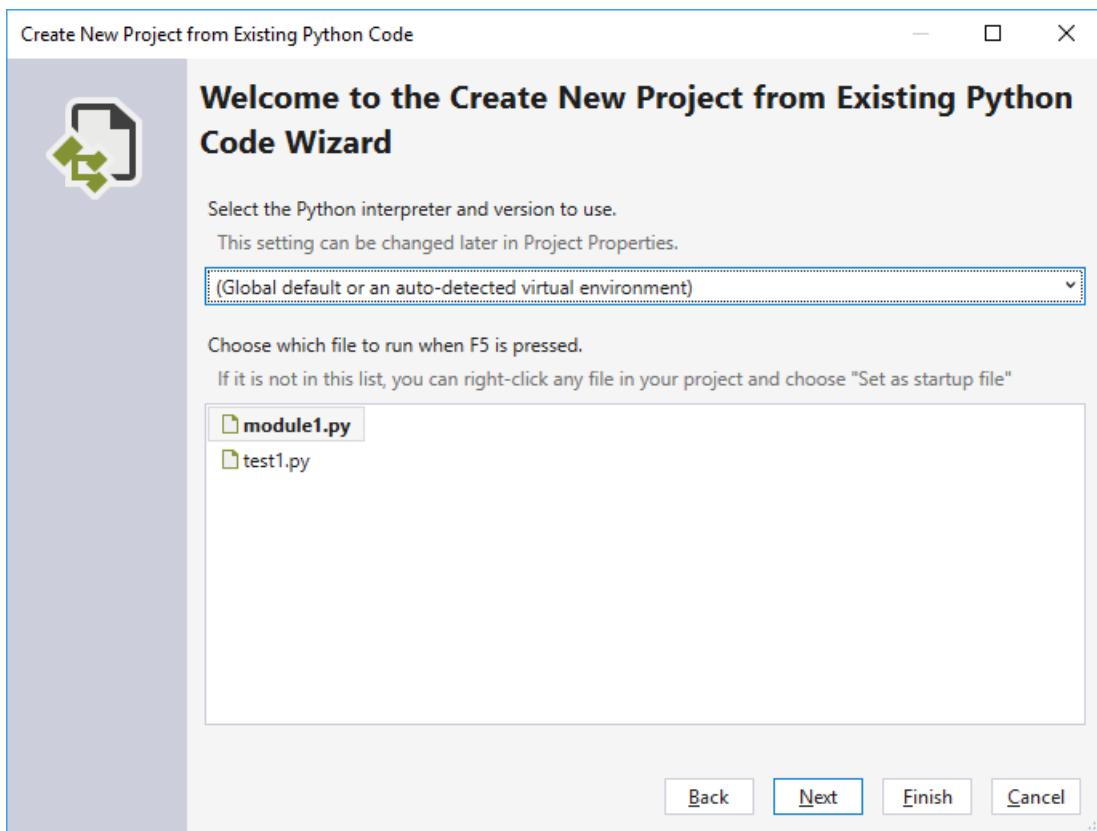
IMPORTANT

The process described here does not move or copy the original source files. If you want to work with a copy, duplicate the folder first.

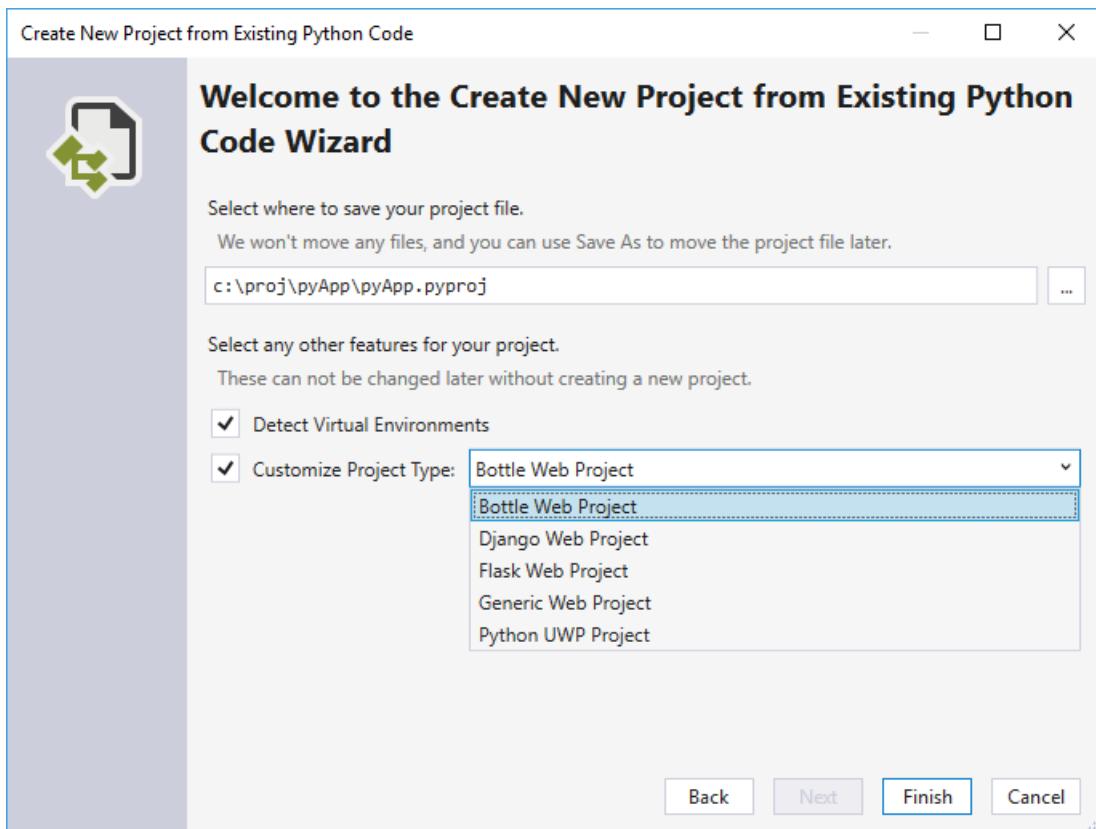
1. Launch Visual Studio and select **File > New > Project**.
2. In the **New Project** dialog, search for "Python", select the **From Existing Python code** template, give the project a name and location, and select **OK**.
3. In the wizard that appears, set the path to your existing code, set a filter for file types, and specify any search paths that your project requires, then select **Next**. If you don't know what search paths are, leave that field blank.



4. In the next dialog, select the startup file for your project and select **Next**. (If desired, select an environment; otherwise accept the defaults.) Note that the dialog shows only files in the root folder; if the file you want is in a subfolder, leave the startup file blank and set it later in **Solution Explorer** (described below).



5. Select the location in which to save the project file (which is a `.pyproj` file on disk). If applicable, you can also include auto-detection of virtual environments and customize the project for different web frameworks. If you're unsure of these options, leave them set to the defaults.



6. Select **Finish** and Visual Studio creates the project and opens it in **Solution Explorer**. If you want to move the `.pyproj` file elsewhere, select it in **Solution Explorer** and choose **File > Save As**. This action updates file references in the project but does not move any code files.
7. To set a different startup file, locate the file in **Solution Explorer**, right-click, and select **Set as Startup File**.

If desired, run the program by pressing **Ctrl+F5** or selecting **Debug > Start without Debugging**.

Next steps

[Tutorial: Work with Python in Visual Studio](#)

See also

- [Manually identify an existing Python interpreter](#)
- [Install Python support in Visual Studio 2015 and earlier](#)
- [Install locations](#)

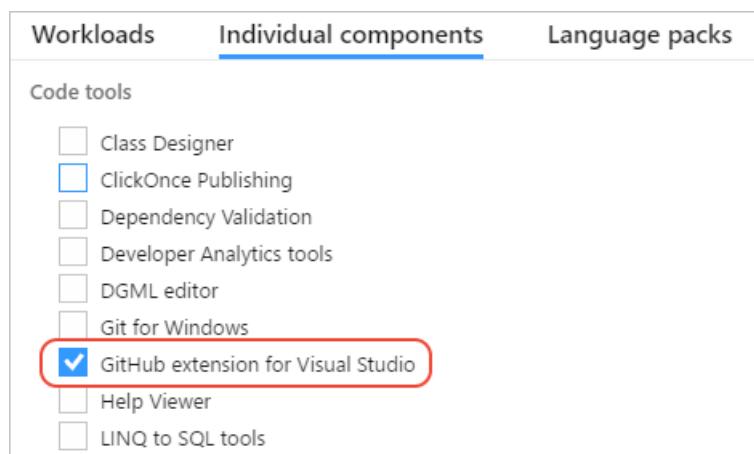
Quickstart: Clone a repository of Python code in Visual Studio

4/26/2019 • 3 minutes to read • [Edit Online](#)

Once you've [installed Python support in Visual Studio](#), you can add the GitHub Extension for Visual Studio. The extension lets you easily clone a repository of Python code and create a project from it from within the IDE. You can always clone repositories on the command line as well, and then work with them in Visual Studio.

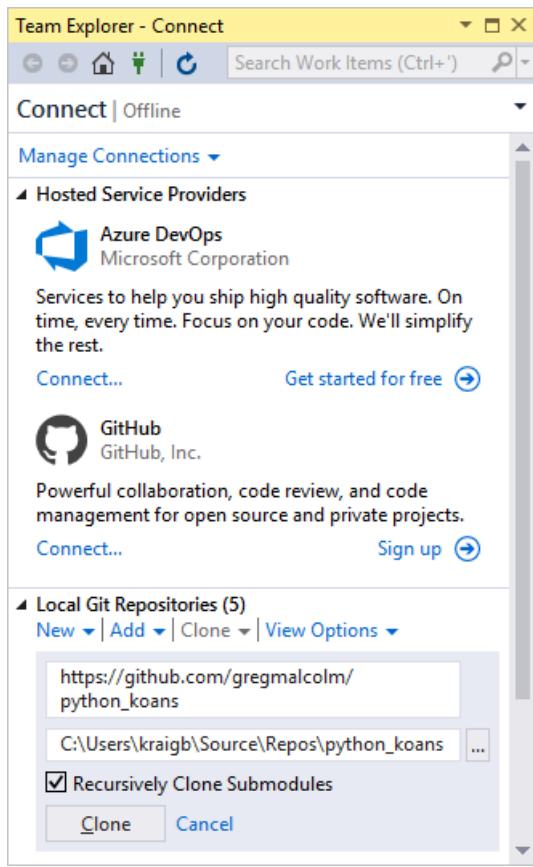
Install the GitHub Extension for Visual Studio

To work with GitHub repositories from within VS, you need to install the GitHub Extension for Visual Studio. To do so, run the Visual Studio installer, select **Modify**, and select the **Individual components** tab. Scroll down to the **Code tools** section, select **GitHub extension for Visual Studio**, and select **Modify**.



Work with GitHub in Visual Studio

1. Launch Visual Studio.
2. Select **View > Team Explorer** to open the **Team Explorer** window in which you can connect to GitHub or Azure Repos, or clone a repository. (If you don't see the **Connect** page shown below, select the plug icon on the top toolbar, which takes you to that page.)

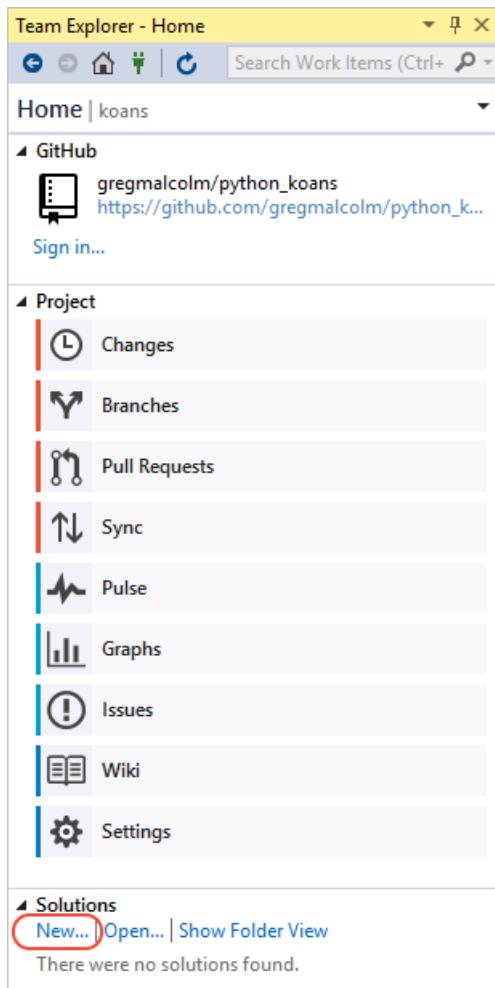


3. Under **Local Git Repositories**, select the **Clone** command, then enter `https://github.com/gregmalcolm/python_koans` in the URL field, enter a folder for the cloned files, and select the **Clone** button.

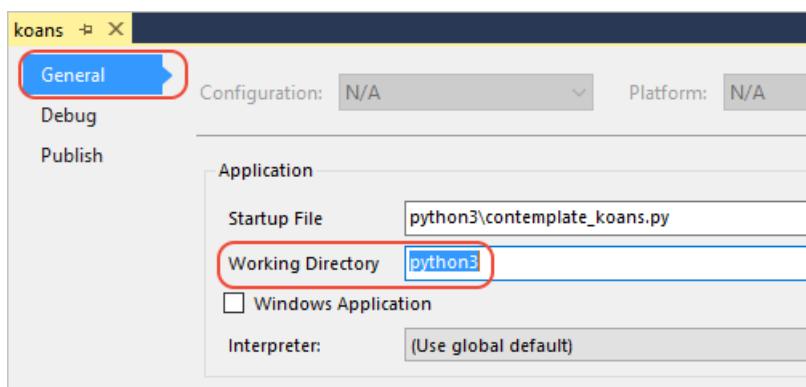
TIP

The folder you specify in **Team Explorer** is the exact folder to receive the cloned files. Unlike the `git clone` command, creating a clone in **Team Explorer** does not automatically create a subfolder with the name of the repository.

4. When cloning is complete, the repository name appears in the **Local Git Repositories** list. Double-click that name to navigate to the repository dashboard in **Team Explorer**.
5. Under **Solutions**, select **New**.



6. In the **New Project** dialog that appears, navigate to the **Python** language (or search on "Python"), select **From Existing Python Code**, specify a name for the project, set **Location** to the same folder as the repository, and select **OK**. In the wizard that appears, select **Finish**.
7. Select **View > Solution Explorer** from the menu.
8. In **Solution Explorer**, expand the **python3** node, right-click **contemplate_koans.py**, and select **Set as Startup File**. This step tells Visual Studio which file it should use when running the project.
9. Select **Project > Koans Properties** from the menu, select the **General** tab, and set **Working Directory** to "python3". This step is necessary because by default Visual Studio sets the working directory to the project root rather than the location of the startup file (*python3\contemplate_koans.py*, which you can see in the project properties as well). The program code looks for a file *koans.txt* in the working folder, so without changing this value you see a runtime error.



10. Press **Ctrl+F5** or select **Debug > Start without Debugging** to run the program. If you see a **FileNotFoundException** for *koans.txt*, check the working directory setting as described in the previous step.

11. When the program runs successfully, it displays an assertion error on line 17 of `python3/koans/about_asserts.py`. This is intentional: the program is designed to teach Python by having you correct all the intentional errors. (More details are found on [Ruby Koans](#), which inspired Python Koans.)

```
C:\Windows\system32\cmd.exe

Thinking AboutAsserts
test_assert_truth has damaged your karma.

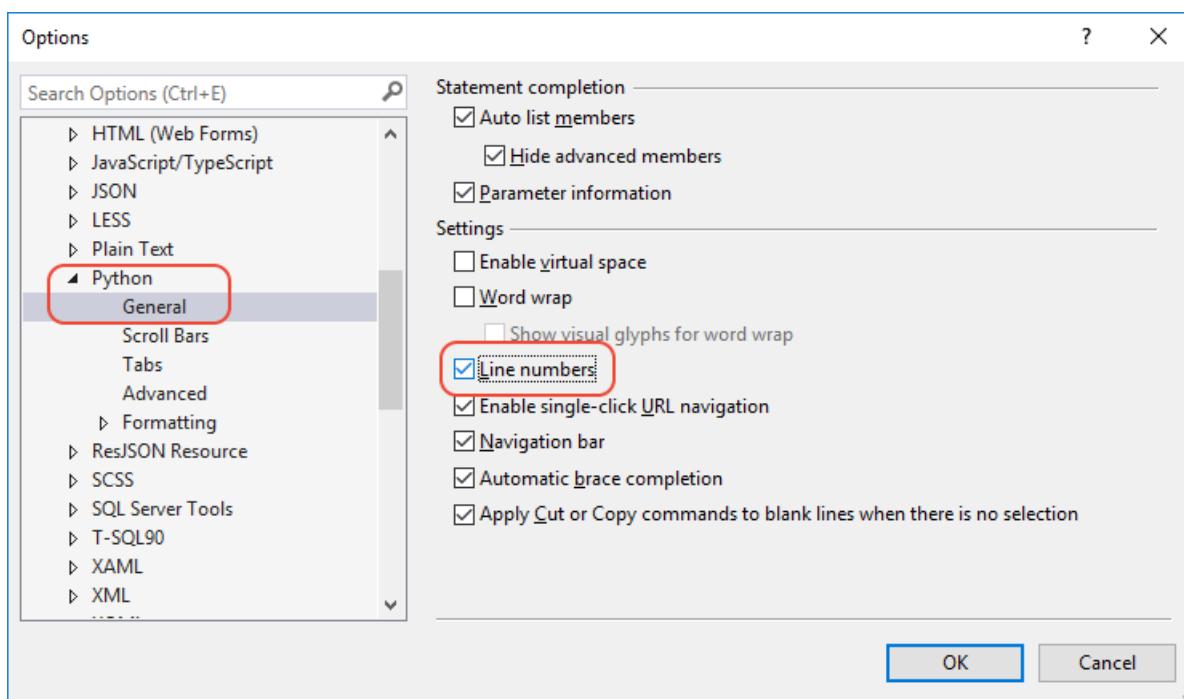
You have not yet reached enlightenment ...
AssertionError: False is not true

Please meditate on the following code:
File "c:\repo\koans\python3\koans\about_asserts.py", line 17, in test_assert_truth
    self.assertTrue(False) # This should be True

You have completed 0 (0 %) koans and 0 (out of 37) lessons.
You are now 302 koans and 37 lessons away from reaching enlightenment.

Beautiful is better than ugly.
Press any key to continue . . .
```

12. Open `python3/koans/about_asserts.py` by navigating to it in **Solution Explorer** and double-clicking the file. Notice that line numbers do not appear by default in the editor. To change this, select **Tools > Options**, select **Show all settings** at the bottom of the dialog, then navigate to **Text Editor > Python > General** and select **Line numbers**:



13. Correct the error by changing the `False` argument on line 17 to `True`. The line should read as follows:

```
self.assertTrue(True) # This should be True
```

14. Run the program again. If Visual Studio warns you about errors, respond with **Yes** to continue running the code. You then see that the first check passes and the program stops on the next koan. Continue correcting the errors and the program again as you want.

IMPORTANT

In this Quickstart, you created a direct clone of the *python_koans* repository on GitHub. Such a repository is protected by its author from direct changes, so attempting to commit changes to the repository fails. In practice, developers instead fork such a repository to their own GitHub account, make changes there, and then create pull requests to submit those changes to the original repository. When you have your own fork, use its URL instead of the original repository URL used earlier.

Next steps

[Tutorial: Work with Python in Visual Studio](#)

See also

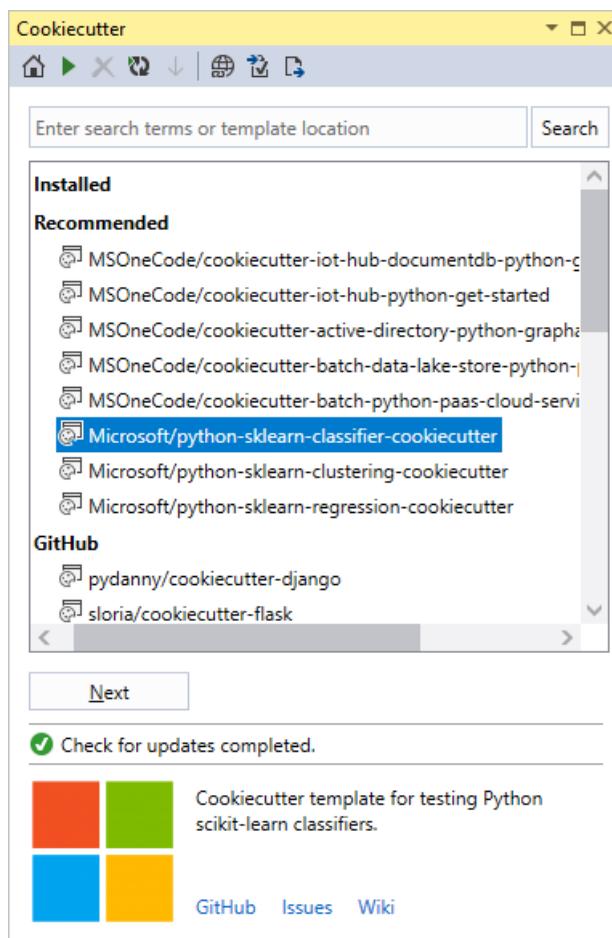
- [Manually identify an existing Python interpreter](#)
- [How to install Python support in Visual Studio on Windows](#)
- [Install locations](#)

Quickstart: Create a project from a Cookiecutter template

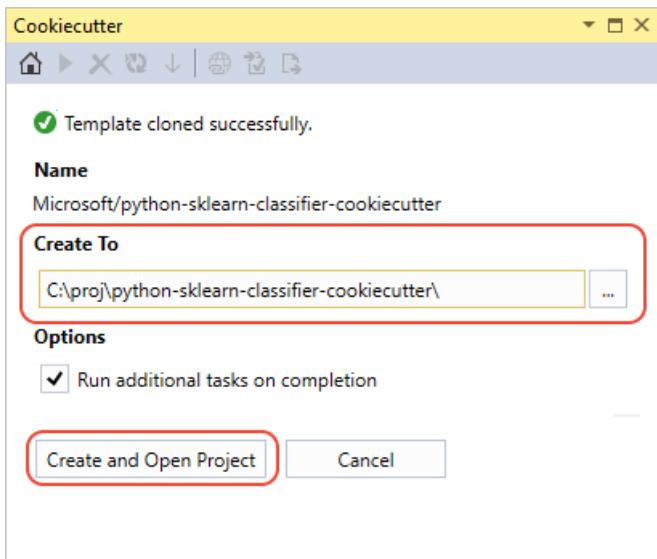
4/9/2019 • 2 minutes to read • [Edit Online](#)

Once you've [installed Python support in Visual Studio](#), it's easy to create a new project from a Cookiecutter template, including many that are published to GitHub. [Cookiecutter](#) provides a graphical user interface to discover templates, input template options, and create projects and files. It's included with Visual Studio 2017 and later and can be installed separately in earlier versions of Visual Studio.

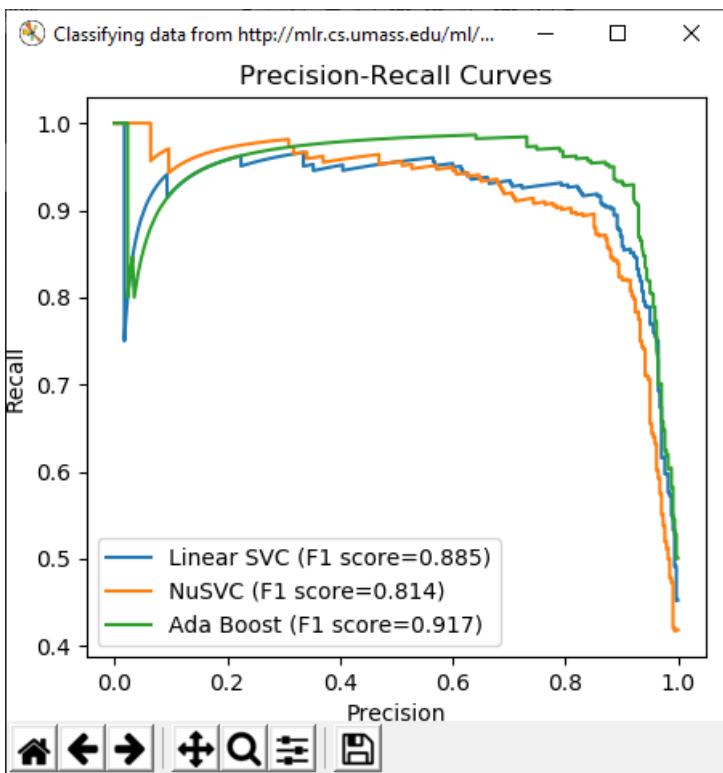
1. For this quickstart, first install the Anaconda3 Python distribution, which includes the necessary Python packages for the Cookiecutter template shown here. Run the Visual Studio installer, select **Modify**, expand the options for **Python development** on the right side, and select **Anaconda3** (either 32-bit or 64-bit). Note that installation may take some time depending on your Internet speed, but this is the simplest way to install the needed packages.
2. Launch Visual Studio.
3. Select **File > New > From Cookiecutter**. This command opens a window in Visual Studio where you can browse templates.



4. Selected the **Microsoft/python-sklearn-classifier-cookiecutter** template, then select **Next**. (The process may take several minutes the first time you use a particular template, as Visual Studio installs the required Python packages.)
5. In the next step, set a location for the new project in the **Create To** field, then select **Create and Open Project**.



6. When the process is complete, you see the message **Successfully created files using template...**. The project is opened in Solution Explorer automatically.
7. Press **Ctrl+F5** or select **Debug > Start without Debugging** to run the program.



Next steps

[Tutorial: Work with Python in Visual Studio](#)

See also

- [Use the Cookiecutter extension](#)
- [Manually identify an existing Python interpreter](#)
- [Install Python support in Visual Studio 2015 and earlier](#)
- [Install locations](#)

Install Python support in Visual Studio

4/9/2019 • 2 minutes to read • [Edit Online](#)

NOTE

Python support is presently available only on Visual Studio for Windows; on Mac and Linux, Python support is available through [Visual Studio Code](#).

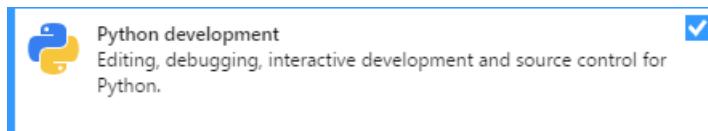
1. Download and run the latest Visual Studio installer for Windows (Python support is present in release 15.2 and later). If you have Visual Studio installed already, run the Visual Studio installer and go to step 2.

[Install Visual Studio Community](#)

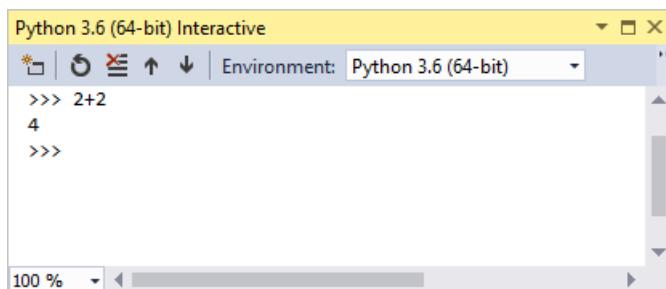
TIP

The Community edition is for individual developers, classroom learning, academic research, and open source development. For other uses, install [Visual Studio Professional](#) or [Visual Studio Enterprise](#).

2. The installer presents you with a list of workloads, which are groups of related options for specific development areas. For Python, select the **Python development** workload and select **Install**:



3. To quickly test Python support, launch Visual Studio, press **Alt+I** to open the **Python Interactive** window, and enter `2+2`. If you don't see the output of `4`, recheck your steps.



Next step

[Step 1: Create a Python project](#)

See also

- [Manually identify an existing Python interpreter](#)
- [Install Python support in Visual Studio 2015 and earlier](#)
- [Install locations](#)

Tutorial: Work with Python in Visual Studio

4/9/2019 • 3 minutes to read • [Edit Online](#)

Python is a popular programming language that is reliable, flexible, easy to learn, free to use on all operating systems, and supported by both a strong developer community and many free libraries. The language supports all manners of development, including web applications, web services, desktop apps, scripting, and scientific computing and is used by many universities, scientists, casual developers, and professional developers alike.

Visual Studio provides first-class language support for Python. This tutorial guides you through the following steps:

- [Step 0: Installation](#)
- [Step 1: Create a Python project \(this article\)](#)
- [Step 2: Write and run code to see Visual Studio IntelliSense at work](#)
- [Step 3: Create more code in the Interactive REPL window](#)
- [Step 4: Run the completed program in the Visual Studio debugger](#)
- [Step 5: Install packages and manage Python environments](#)
- [Step 6: Work with Git](#)

Prerequisites

- Visual Studio 2017 with the Python workload installed. For instructions, see [Work with Python in Visual Studio - Step 0](#).
- Visual Studio 2019 with the Python workload installed. For instructions, see [Work with Python in Visual Studio - Step 0](#).

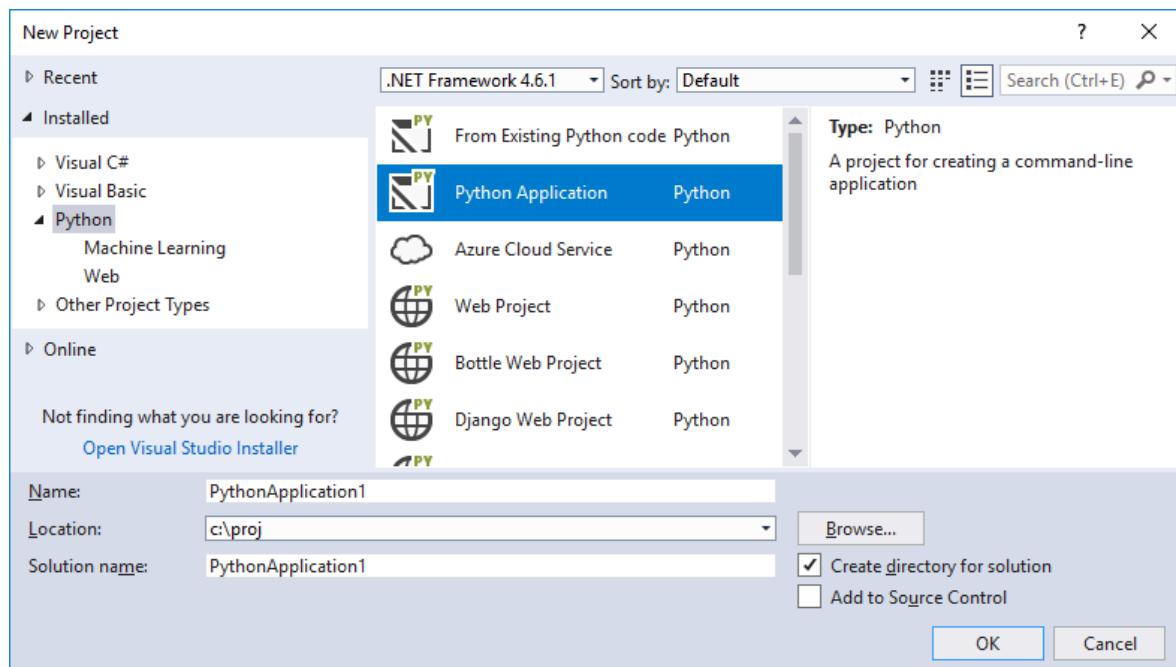
You can also use an earlier version of Visual Studio with the Python Tools for Visual Studio installed. See [Install Python support in Visual Studio](#).

Step 1: Create a new Python project

A *project* is how Visual Studio manages all the files that come together to produce a single application, including source code, resources, configurations, and so on. A project formalizes and maintains the relationship between all the project's files as well as external resources that are shared between multiple projects. As such, projects allow your application to effortlessly expand and grow much easier than simply managing a project's relationships in ad hoc folders, scripts, text files, and even your own mind.

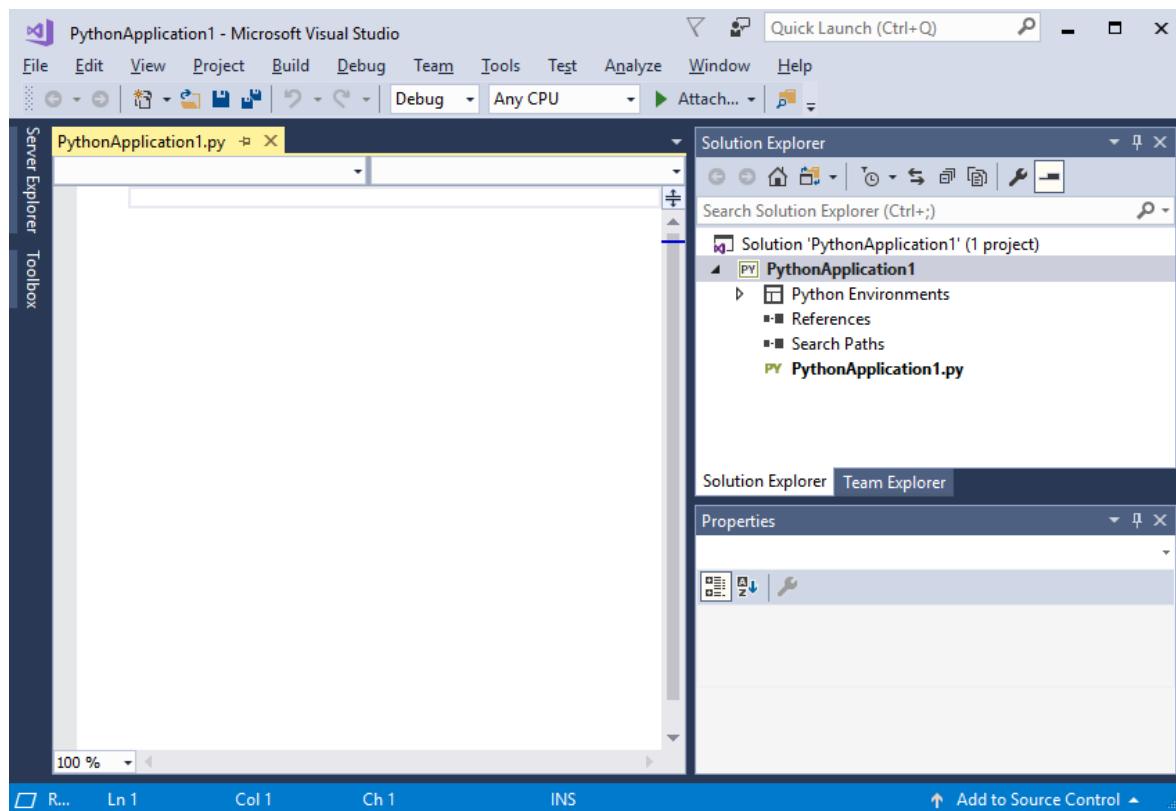
In this tutorial you begin with a simple project containing a single, empty code file.

1. In Visual Studio, select **File > New > Project (Ctrl+Shift+N)**, which brings up the **New Project** dialog. Here you browse templates across different languages, then select one for your project and specify where Visual Studio places files.
2. To view Python templates, select **Installed > Python** on the left, or search for "Python". Using search is a great way to find a template when you can't remember its location in the languages tree.

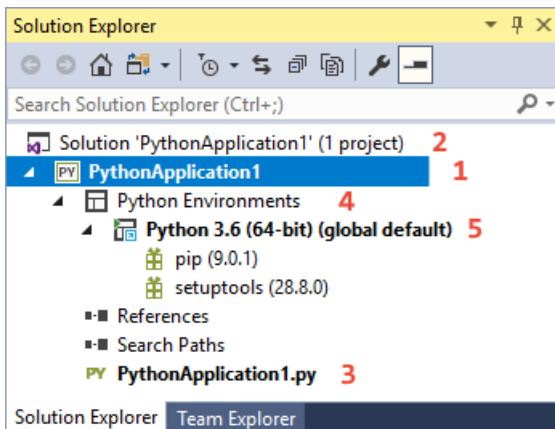


Notice how Python support in Visual Studio includes a number of project templates, including web applications using the Bottle, Flask, and Django frameworks. For the purposes of this walkthrough, however, let's start with an empty project.

3. Select the **Python Application** template, specify a name for the project, and select **OK**.
4. After a few moments, Visual Studio shows the project structure in the **Solution Explorer** window (1). The default code file is open in the editor (2). The **Properties** window (3) also appears to show additional information for any item selected in **Solution Explorer**, including its exact location on disk.



5. Take a few moments to familiarize yourself with **Solution Explorer**, which is where you browse files and folders in your project.



(1) Highlighted in bold is your project, using the name you gave in the **New Project** dialog. On disk, this project is represented by a `.pyproj` file in your project folder.

(2) At the top level is a *solution*, which by default has the same name as your project. A solution, represented by a `.sln` file on disk, is a container for one or more related projects. For example, if you write a C++ extension for your Python application, that C++ project could reside within the same solution. The solution might also contain a project for a web service, along with projects for dedicated test programs.

(3) Under your project you see source files, in this case only a single `.py` file. Selecting a file displays its properties in the **Properties** window. Double-clicking a file opens it in whatever way is appropriate for that file.

(4) Also under the project is the **Python Environments** node. When expanded, you see the Python interpreters that are available to you. Expand an interpreter node to see the libraries that are installed into that environment (5).

Right-click any node or item in **Solution Explorer** to access a menu of applicable commands. For example, the **Rename** command allows you to change the name of any node or item, including the project and the solution.

Next step

[Write and run code](#)

Go deeper

- [Python projects in Visual Studio](#).
- [Learn about the Python language on python.org](#)
- [Python for Beginners \(python.org\)](#)

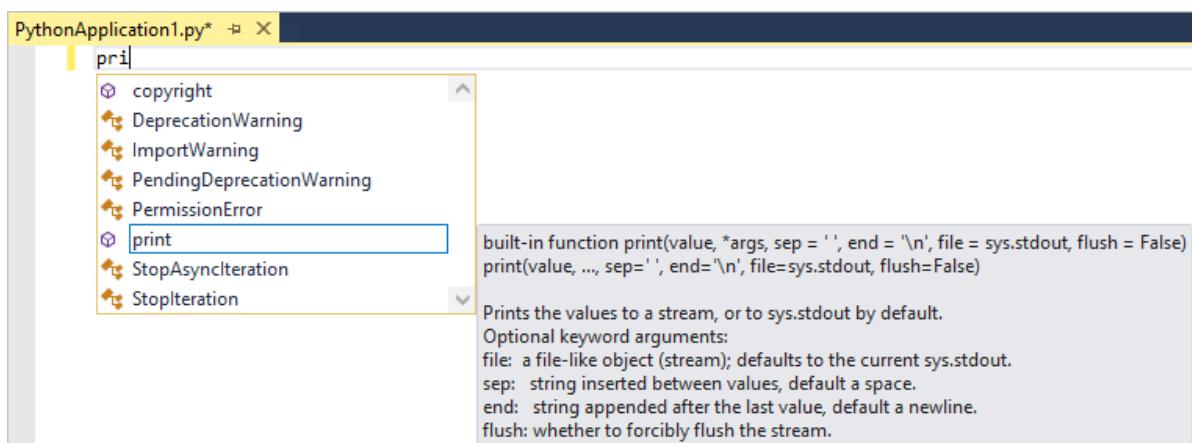
Step 2: Write and run code

4/9/2019 • 2 minutes to read • [Edit Online](#)

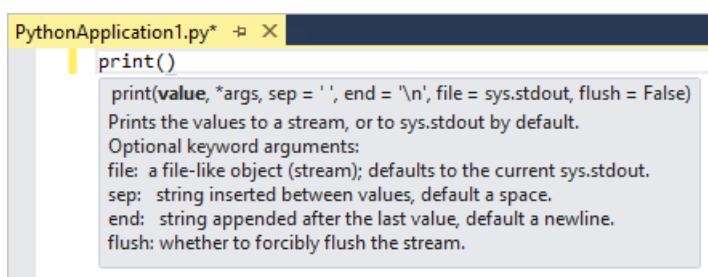
Previous step: Create a new Python project

Although **Solution Explorer** is where you manage project files, the *editor* window is typically where you work with the *contents* of files, like source code. The editor is contextually aware of the type of file you're editing, including the programming language (based on the file extension), and offers features appropriate to that language such as syntax coloring and auto-completion using IntelliSense.

1. After creating a new "Python Application" project, a default empty file named *PythonApplication1.py* is open in the Visual Studio editor.
2. In the editor, start typing `print("Hello, Visual Studio")` and notice how Visual Studio IntelliSense displays auto-completion options along the way. The outlined option in the drop-down list is the default completion that's used when you press the **Tab** key. Completions are most helpful when longer statements or identifiers are involved.



3. IntelliSense shows different information depending on the statement you're using, the function you're calling, and so forth. With the `print` function, typing `()` after `print` to indicate a function call displays full usage information for that function. The IntelliSense pop up also shows the current argument in boldface (**value** as shown here):



4. Complete the statement so it matches the following:

```
print("Hello, Visual Studio")
```

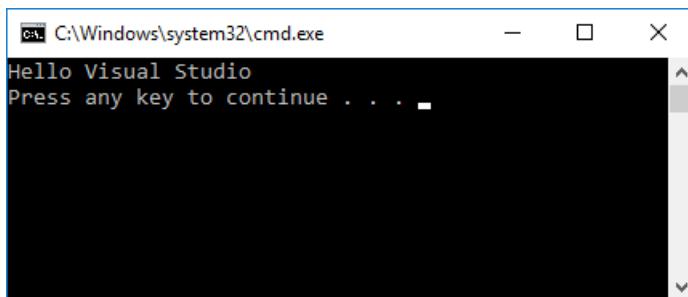
5. Notice the syntax coloration that differentiates the statement `print` from the argument `"Hello Visual Studio"`. Also, temporarily delete the last `"` on the string and notice how Visual Studio shows a red underline for code that contains syntax errors. Then replace the `"` to correct the code.

```
PythonApplication1.py* ▸ X
print("Hello Visual Studio")
~
```

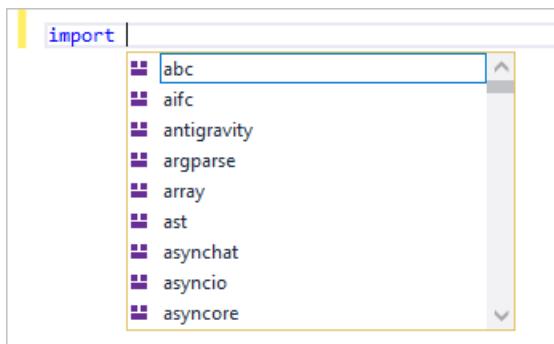
TIP

Because one's development environment is a very personal matter, Visual Studio gives you complete control over Visual Studio's appearance and behavior. Select the **Tools > Options** menu command and explore the settings under the **Environment** and **Text Editor** tabs. By default you see only a limited number of options; to see every option for every programming language, select **Show all settings** at the bottom of the dialog box.

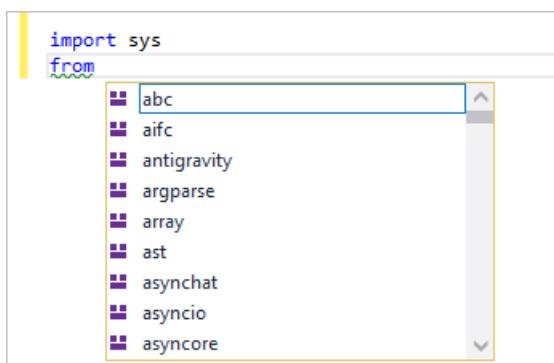
6. Run the code you've written to this point by pressing **Ctrl+F5** or selecting **Debug > Start without Debugging** menu item. Visual Studio warns you if you still have errors in your code.
7. When you run the program, a console window appears displaying the results, just as if you'd run a Python interpreter with *PythonApplication1.py* from the command line. Press a key to close the window and return to the Visual Studio editor.



8. In addition to completions for statements and functions, IntelliSense provide completions for Python `import` and `from` statements. These completions help you easily discover what modules are available in your environment and the members of those modules. In the editor, delete the `print` line and start typing `import .` A list of modules appears when you type the space:



9. Complete the line by typing or selecting `sys`.
10. On the next line, type `from` to again see a list of modules:



11. Select or type `math`, then continue typing with a space and `import`, which displays the module members:

The screenshot shows a code editor with the following code in the main pane:

```
import sys
from math import
```

A dropdown menu titled "Import all members from the module" is open over the word "import". The menu lists several mathematical functions starting with an asterisk (*), indicating they are available for import. The listed functions are:

- * (selected)
- acos
- acosh
- asin
- asinh
- atan
- atan2
- atanh
- BuiltinImporter

12. Finish by importing the `sin`, `cos`, and `radians` members, noticing the auto-completions available for each. When you're done, your code should appear as follows:

```
import sys
from math import cos, radians
```

TIP

Completions work with substrings as you type, matching parts of words, letters at the beginning of words, and even skipped characters. See [Edit code - Completions](#) for details.

13. Add a little more code to print the cosine values for 360 degrees:

```
for i in range(360):
    print(cos(radians(i)))
```

14. Run the program again with **Ctrl+F5** or **Debug > Start without Debugging**. Close the output window when you're done.

Next step

[Use the interactive REPL window](#)

Go deeper

- [Edit code](#)
- [Format code](#)
- [Refactor code](#)
- [Use PyLint](#)

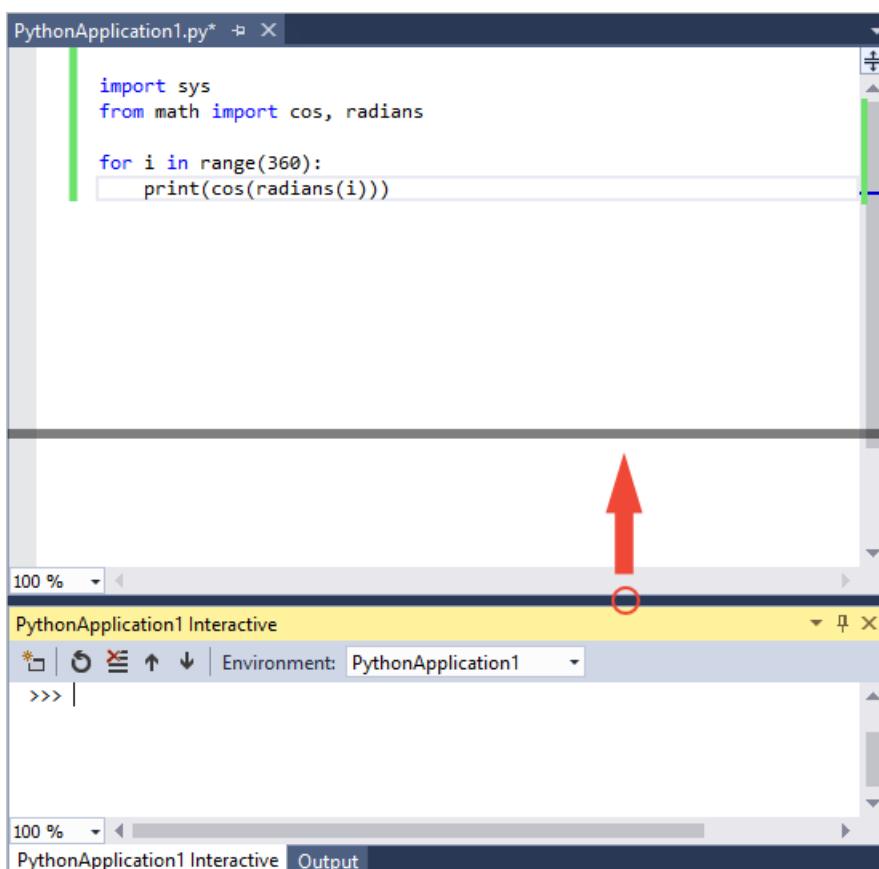
Step 3: Use the Interactive REPL window

4/9/2019 • 4 minutes to read • [Edit Online](#)

Previous step: Write and run code

The Visual Studio **Interactive** window for Python provides a rich read-evaluate-print-loop (REPL) experience that greatly shortens the usual edit-build-debug cycle. The **Interactive** window provides all the capabilities of the REPL experience of the Python command line. It also makes it very easy to exchange code with source files in the Visual Studio editor, which is otherwise cumbersome with the command line.

1. Open the **Interactive** window by right-clicking the project's Python environment in **Solution Explorer** (such as **Python 3.6 (32-bit)** shown in an earlier graphic) and selecting **Open Interactive Window**. You can alternately select **View > Other Windows > Python Interactive Windows** from the main Visual Studio menu.
2. The **Interactive** window opens below the editor with the standard `>>>` Python REPL prompt. The **Environment** drop-down list allows you to select a specific interpreter to work with. Oftentimes you also want to make the **Interactive** window larger, which you can do by dragging the separator between the two windows:

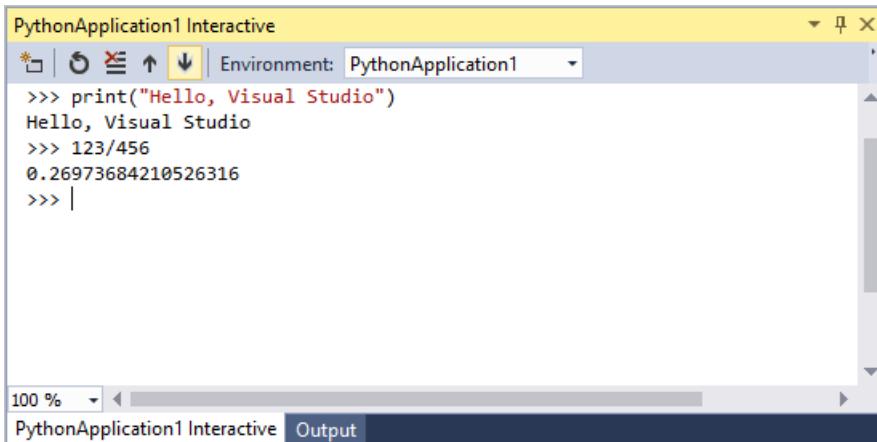


TIP

You can resize all of the windows in Visual Studio by dragging the bordering separators. You can also drag windows out independently of the Visual Studio frame, and rearrange them however you like within the frame. For complete details, see [Customize window layouts](#).

3. Enter a few statements like `print("Hello, Visual Studio")` and expressions like `123/456` to see immediate

results:

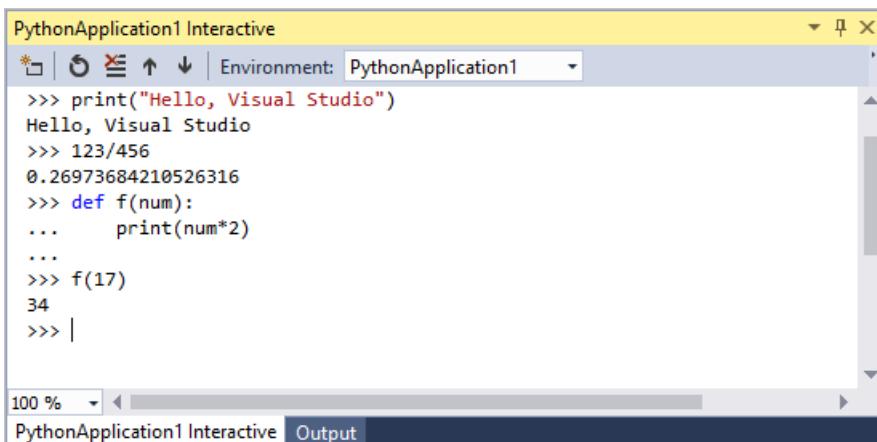


The screenshot shows the Python Application1 Interactive window. The title bar says "PythonApplication1 Interactive". The environment dropdown is set to "PythonApplication1". The code input area contains the following Python code:

```
>>> print("Hello, Visual Studio")
Hello, Visual Studio
>>> 123/456
0.26973684210526316
>>> |
```

The output area at the bottom shows the results of the code execution.

- When you start writing a multiline statement, like a function definition, the **Interactive** window shows Python's ... prompt for continuing lines, which, unlike the command-line REPL, provides automatic indentation:



The screenshot shows the Python Application1 Interactive window. The title bar says "PythonApplication1 Interactive". The environment dropdown is set to "PythonApplication1". The code input area contains the following Python code:

```
>>> print("Hello, Visual Studio")
Hello, Visual Studio
>>> 123/456
0.26973684210526316
>>> def f(num):
...     print(num*2)
...
>>> f(17)
34
>>> |
```

The output area at the bottom shows the results of the code execution.

- The **Interactive** window provides a full history of everything you've entered, and improves upon the command-line REPL with multiline history items. For example, you can easily recall the entire definition of the `f` function as a single unit and easily change the name to `make_double`, rather than re-creating the function line by line.
- Visual Studio can send multiple lines of code from an editor window to the **Interactive** window. This capability allows you to maintain code in a source file and easily send select parts of it to the **Interactive** window. You can then work with such code fragments in the rapid REPL environment rather than having to run the whole program. To see this feature, first replace the `for` loop in the *PythonApplication1.py* file with the following:

```
# Create a string with spaces proportional to a cosine of x in degrees
def make_dot_string(x):
    return ' ' * int(20 * cos(radians(x))) + 'o'
```

- Select only the `import` and `from` statements in the *.py* file, right-click, and select **Send to Interactive** (or press **Ctrl+Enter**). The code fragment is immediately pasted into the **Interactive** window and run. Now select the `make_dot_string` function and repeat the same command, which again runs that code fragment. Because the code defines a function, you can quickly test that function by calling it a few times:

The screenshot shows the Python Application 1 Interactive window. The code in the editor is:

```
>>> from math import sin, cos, radians
>>> # Create a string with spaces proportional to a cosine of x in degrees
... def make_dot_string(x):
...     return ' ' * int(20 * cos(radians(x))) + 'o'
...
>>> make_dot_string(90)
' o'
>>> make_dot_string(180)
'o'
>>> make_dot_string(135)
' o'
>>> |
```

TIP

Using **Ctrl+Enter** in the editor *without* a selection runs the current line of code in the **Interactive** window and automatically places the caret on the next line. With this feature, pressing **Ctrl+Enter** repeatedly provides a convenient way to step through your code that is not possible with only the Python command line. It also lets you step through your code without running the debugger and without necessarily starting your program from the beginning.

8. You can also copy and paste multiple lines of code into the **Interactive** window from any source, such as the snippet below, which is difficult to do with the Python command-line REPL. When pasted, the **Interactive** window runs that code as if you'd typed it in:

```
for i in range(360):
    s = make_dot_string(i)
    print(s)
```

The screenshot shows the Python Application 1 Interactive window. The code in the editor is:

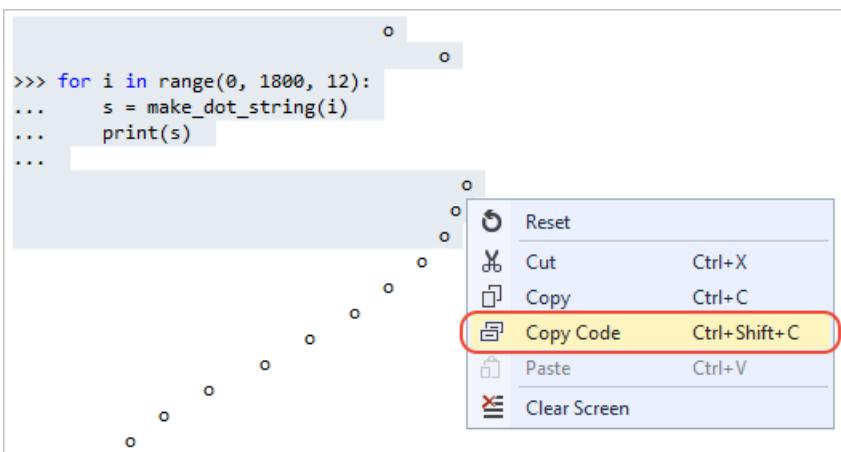
```
>>> for i in range(360):
...     s = make_dot_string(i)
...     print(s)
...

```

The output in the window shows a vertical column of 'o' characters, indicating the code is running correctly.

9. As you can see, this code works fine but its output isn't very inspiring. A different step value in the `for` loop would show more of the cosine wave. Fortunately, because the entire `for` loop is in the REPL history as a single unit, it's easy to go back and make whatever changes you want and then test the function again. Press the up arrow to first recall the `for` loop. Then press the left or right arrows to start navigating in the code (until you do so, the up and down arrows continue to cycle through the history). Navigate to and change the `range` specification to `range(0, 360, 12)`. Then press **Ctrl+Enter** (anywhere in the code) to run the whole statement again:

10. Repeat the process to experiment with different step settings until you find a value you like best. You can also make the wave repeat by lengthening the range, for example, `range(0, 1800, 12)`.
 11. When you're satisfied with code you've written in the **Interactive** window, select it, right-click and select **Copy Code (Ctrl+Shift+C)**, and then paste into the editor. Notice how this special feature of Visual Studio automatically omits any output as well as the `>>>` and `...` prompts. For example, the image below shows using the **Copy Code** command on a selection that includes prompts and output:



When you paste into the editor, you get only the code:

```
for i in range(0, 1800, 12):
    s = make_dot_string(i)
    print(s)
```

If you want to copy the exact contents of the **Interactive** window, including prompts and output, just use the standard **Copy** command.

12. What you've just done is use the rapid REPL environment of the **Interactive** window to work out the details for a small piece of code, then you conveniently added that code to your project's source file. When you now run the code again with **Ctrl+F5** (or **Debug > Start without Debugging**), you see the exact results you wanted.

Next step

[Run code in the debugger](#)

Go deeper

- [Use the Interactive window](#)
- [Use IPython REPL](#)

Step 4: Run code in the debugger

4/9/2019 • 6 minutes to read • [Edit Online](#)

Previous step: [Use the Interactive REPL window](#)

In addition to managing projects, providing a rich editing experience, and the **Interactive** window, Visual Studio provides full-featured debugging for Python code. In the debugger, you can run your code step by step, including every iteration of a loop. You can also pause the program whenever certain conditions are true. At any point when the program is paused in the debugger, you can examine the entire program state and change the value of variables. Such actions are essential for tracking down program bugs, and also provide very helpful aids for carefully following the exact program flow.

1. Replace the code in the *PythonApplication1.py* file with the following. This variation of the code expands `make_dot_string` so that you can examine its discrete steps in the debugger. It also places the `for` loop into a `main` function and runs it explicitly by calling that function:

```
from math import cos, radians

# Create a string with spaces proportional to a cosine of x in degrees
def make_dot_string(x):
    rad = radians(x)                      # cos works with radians
    numspaces = int(20 * cos(radians(x)) + 20)  # scale to 0-40 spaces
    st = ' ' * numspaces + 'o'             # place 'o' after the spaces
    return st

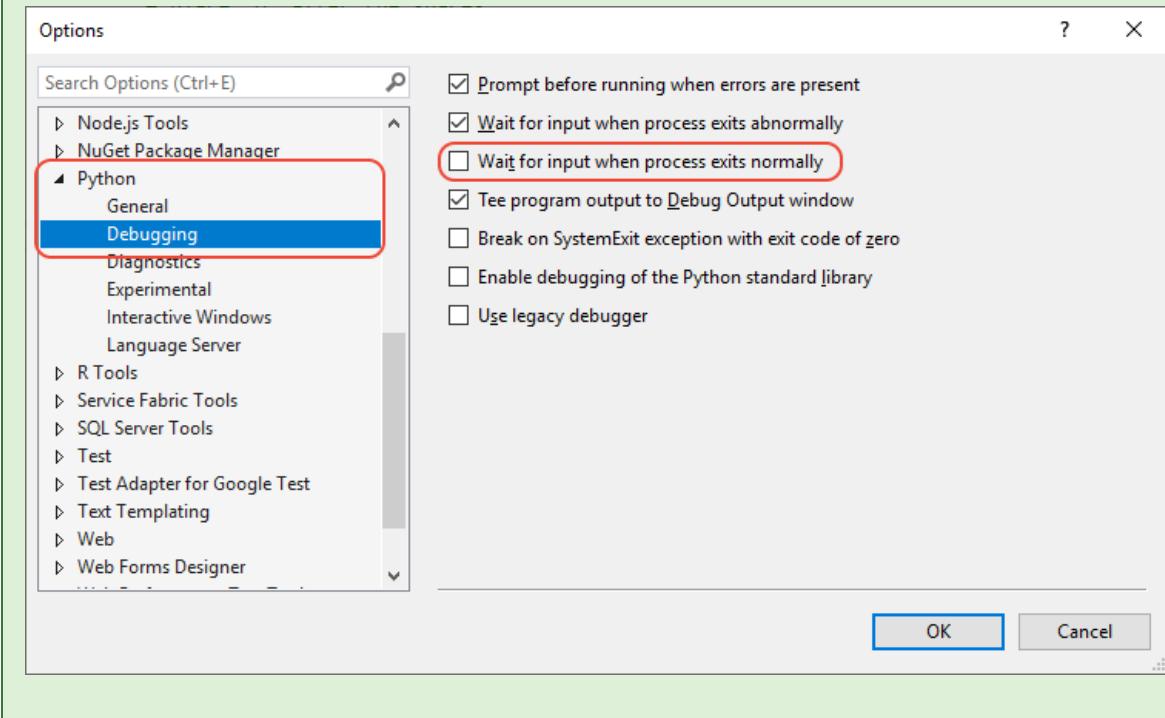
def main():
    for i in range(0, 1800, 12):
        s = make_dot_string(i)
        print(s)

main()
```

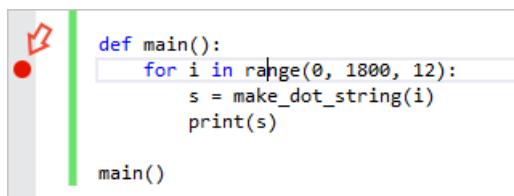
2. Check that the code works properly by pressing **F5** or selecting the **Debug > Start Debugging** menu command. This command runs the code in the debugger, but because you haven't done anything to pause the program while it's running, it just prints a wave pattern for a few iterations. Press any key to close the output window.

TIP

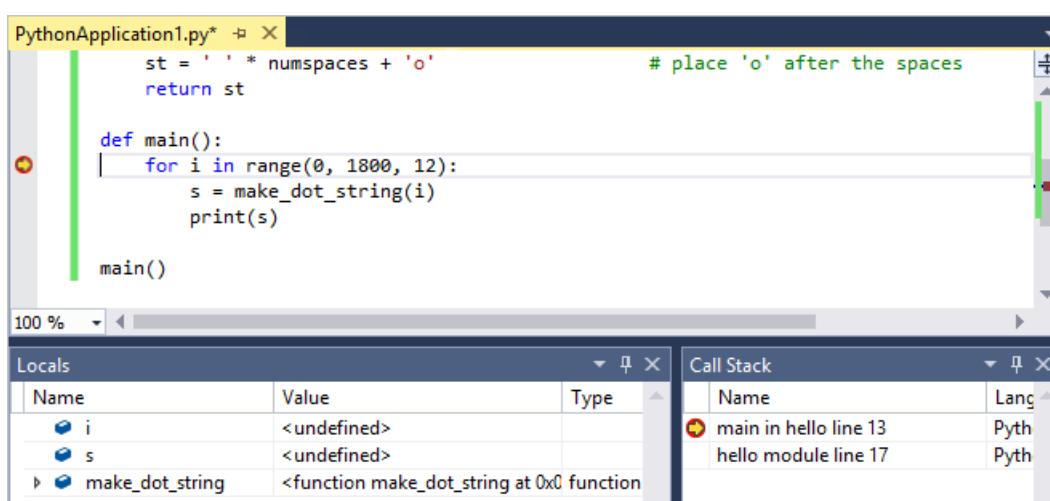
To close the output window automatically when the program completes, select the **Tools > Options** menu command, expand the **Python** node, select **Debugging**, and then clear the option **Wait for input when process exits normally**:



3. Set a breakpoint on the `for` statement by clicking once in the gray margin by that line, or by placing the caret in that line and using the **Debug > Toggle Breakpoint** command (**F9**). A red dot appears in the gray margin to indicate the breakpoint (as noted by the arrow below):



4. Start the debugger again (**F5**) and see that running the code stops on the line with that breakpoint. Here you can inspect the call stack and examine variables. Variables that are in-scope appear in the **Autos** window when they're defined; you can also switch to the **Locals** view at the bottom of that window to show all variables that Visual Studio finds in the current scope (including functions), even before they're defined:



5. Observe the debugging toolbar (shown below) along the top of the Visual Studio window. This toolbar

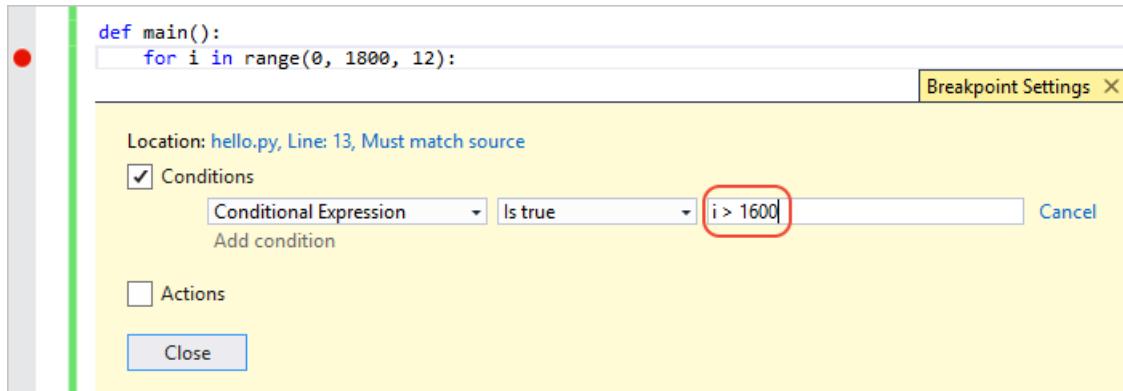
provides quick access to the most common debugging commands (which can also be found on the **Debug** menu):



The buttons from left to right as follows:

- **Continue (F5)** runs the program until the next breakpoint or until program completion.
 - **Break All (Ctrl+Alt+Break)** pauses a long-running program.
 - **Stop Debugging (Shift+F5)** stops the program wherever it is, and exits the debugger.
 - **Restart (Ctrl+Shift+F5)** stops the program wherever it is, and restarts it from the beginning in the debugger.
 - **Show Next Statement (Alt+Num *)** switches to the next line of code to run. This is most helpful when you navigate around within your code during a debugging session and want to quickly return to the point where the debugger is paused.
 - **Step Into (F11)** runs the next line of code, entering into called functions.
 - **Step Over (F10)** runs the next line of code without entering into called functions.
 - **Step Out (Shift+F11)** runs the remainder of the current function and pauses in the calling code.
6. Step over the `for` statement using **Step Over**. *Stepping* means that the debugger runs the current line of code, including any function calls, and then immediately pauses again. Notice how the variable `i` is now defined in the **Locals** and **Autos** windows.
 7. Step over the next line of code, which calls `make_dot_string` and pauses. **Step Over** here specifically means that the debugger runs the whole of `make_dot_string` and pauses when it returns. The debugger does not stop inside that function unless a separate breakpoint exists there.
 8. Continue stepping over the code a few more times and observe how the values in the **Locals** or **Autos** window change.
 9. In the **Locals** or **Autos** window, double-click in the **Value** column for either the `i` or `s` variables to edit the value. Press **Enter** or click outside that value to apply any changes.
 10. Continue stepping through the code using **Step Into**. **Step Into** means that the debugger enters inside any function call for which it has debugging information, such as `make_dot_string`. Once inside `make_dot_string` you can examine its local variables and step through its code specifically.
 11. Continue stepping with **Step Into** and notice that when you reach the end of the `make_dot_string`, the next step returns to the `for` loop with the new return value in the `s` variable. As you step again to the `print` statement, notice that **Step Into** on `print` does not enter into that function. This is because `print` is not written in Python but is rather native code inside the Python runtime.
 12. Continue using **Step Into** until you're again partway into `make_dot_string`. Then use **Step Out** and notice that you return to the `for` loop. With **Step Out**, the debugger runs the remainder of the function and then automatically pauses in the calling code. This is very helpful when you've stepped through some portion of a lengthy function that you wish to debug, but don't need to step through the rest and don't want to set an explicit breakpoint in the calling code.
 13. To continue running the program until the next breakpoint is reached, use **Continue (F5)**. Because you have a breakpoint in the `for` loop, you break on the next iteration.
 14. Stepping through hundreds of iterations of a loop can be tedious, so Visual Studio lets you add a *condition* to a breakpoint. The debugger then pauses the program at the breakpoint only when the condition is met. For example, you can use a condition with the breakpoint on the `for` statement so that it pauses only when the value of `i` exceeds 1600. To set this condition, right-click the red breakpoint dot and select **Conditions** (**Alt+F9 > C**). In the **Breakpoint Settings** popup that appears, enter `i > 1600` as the expression and select

Close. Press **F5** to continue and observe that the program runs many iterations before the next break.



15. To run the program to completion, disable the breakpoint by right-clicking and selecting **Disable breakpoint (Ctrl+F9)**. Then select **Continue** (or press **F5**) to run the program. When the program ends, Visual Studio stops its debugging session and returns to its editing mode. Note that you can also delete the breakpoint by clicking its dot, but this also deletes any condition you've set.

TIP

In some situations, such as a failure to launch the Python interpreter itself, the output window may appear only briefly and then close automatically without giving you a chance to see any errors messages. If this happens, right-click the project in **Solution Explorer**, select **Properties**, select the **Debug** tab, then add `-i` to the **Interpreter Arguments** field. This argument causes the interpreter to go into interactive mode after a program completes, thereby keeping the window open until you enter **Ctrl+Z > Enter** to exit.

Next step

[Install packages in your Python environment](#)

Go deeper

- [Debugging](#)
- [Debugging in Visual Studio](#) provides full documentation of Visual Studio's debugging features.

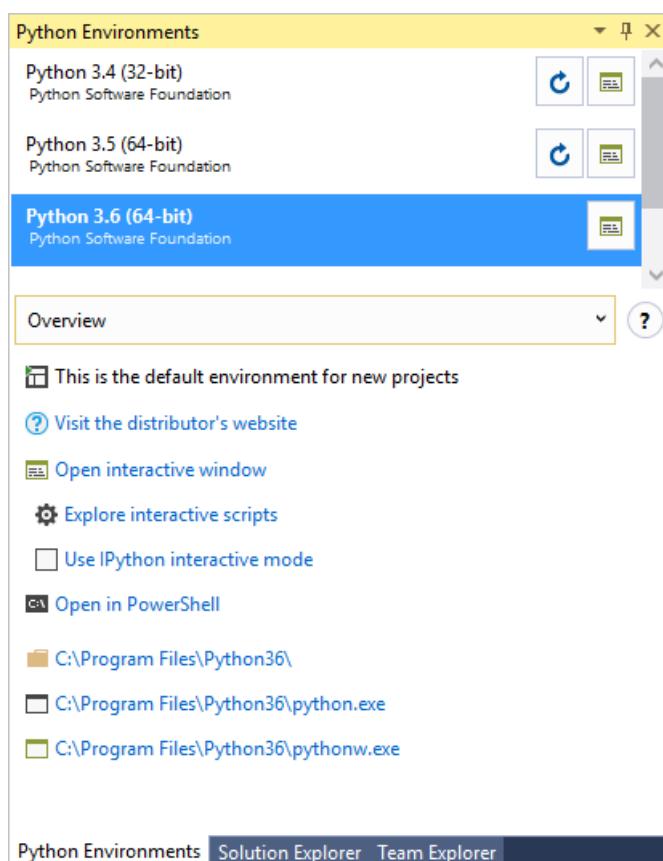
Step 5: Install packages in your Python environment

4/9/2019 • 2 minutes to read • [Edit Online](#)

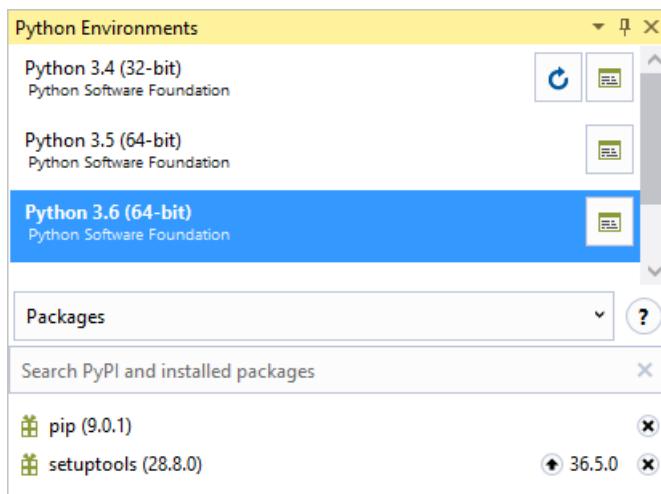
Previous step: [Run code in the debugger](#)

The Python developer community has produced thousands of useful packages that you can incorporate into your own projects. Visual Studio provides a UI to manage packages in your Python environments.

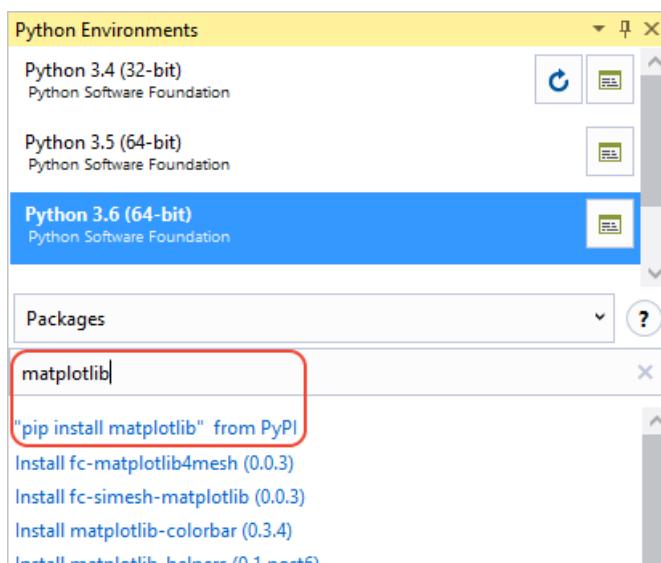
1. Select the **View > Other Windows > Python Environments** menu command. The **Python Environments** window opens as a peer to **Solution Explorer** and shows the different environments available to you. The list includes both environments that you installed using the Visual Studio installer and those you installed separately. The environment in bold is the default environment that's used for new projects.



2. The environment's **Overview** tab provides quick access to an **Interactive** window for that environment along with the environment's installation folder and interpreters. For example, select **Open interactive window** and an **Interactive** window for that specific environment appears in Visual Studio.
3. Select the **Packages** tab and you see a list of packages that are currently installed in the environment.

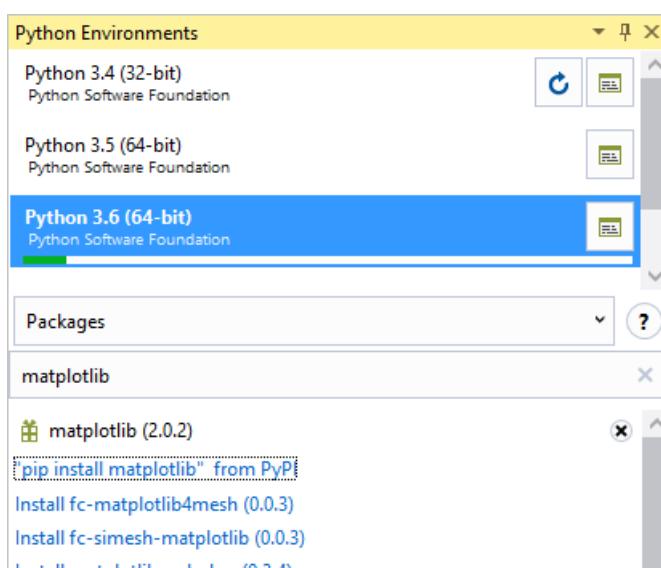


4. Install `matplotlib` by entering its name into the search field, then select the **pip install**



5. Consent to elevation if prompted to do so.

6. After the package is installed, it appears in the **Python Environments** window. The **X** to the right of the package uninstalls it.



A small progress bar may appear underneath the environment to indicate that Visual Studio is building its IntelliSense database for the newly-installed package. The **IntelliSense** tab also shows more detailed information. Note that until that database is complete, IntelliSense features like auto-completion and syntax checking won't be active in the editor for that package.

Note that Visual Studio 2017 version 15.6 and later uses a different and faster method for working with IntelliSense, and displays a message to that effect on the **IntelliSense** tab.

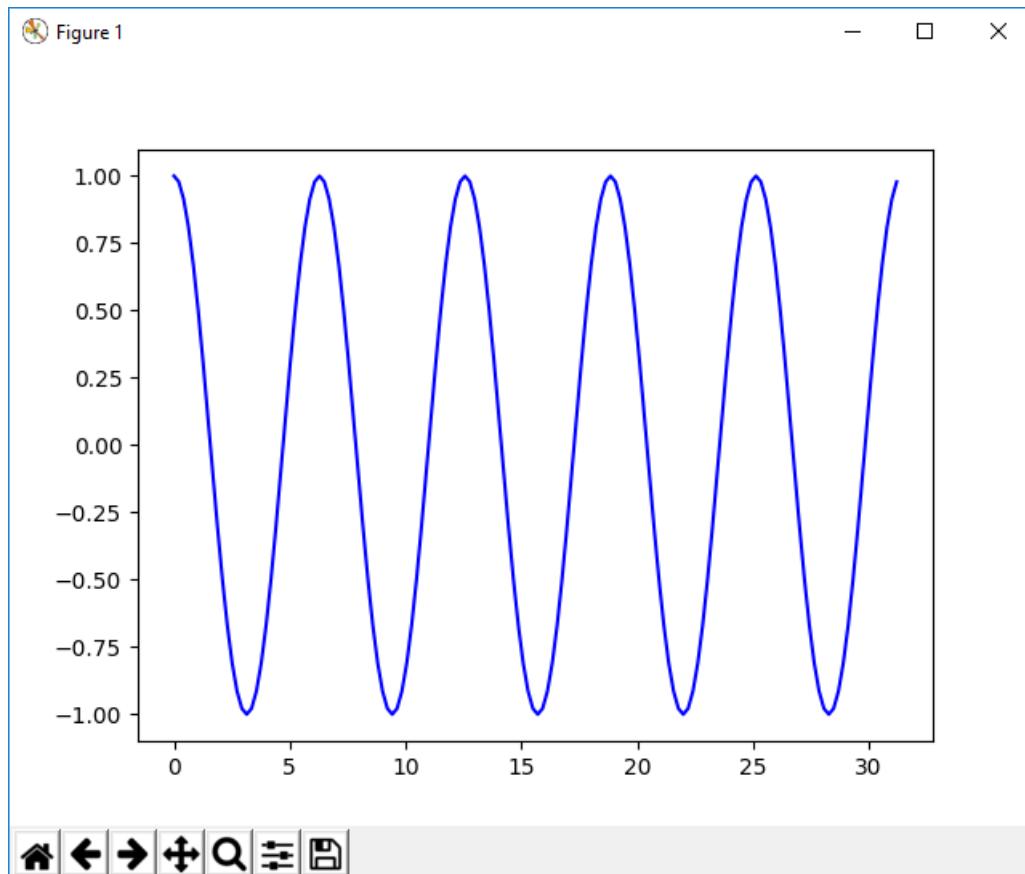
7. Create a new project with **File > New > Project**, selecting the **Python Application** template. In the code file that appears, paste the following code, which creates a cosine wave like the previous tutorial steps, only this time plotted graphically:

```
from math import radians
import numpy as np      # installed with matplotlib
import matplotlib.pyplot as plt

def main():
    x = np.arange(0, radians(1800), radians(12))
    plt.plot(x, np.cos(x), 'b')
    plt.show()

main()
```

8. Run the program with (**F5**) or without the debugger (**Ctrl+F5**) to see the output:



Next step

[Work with Git](#)

Go deeper

- [Python environments](#)

Step 6: Work with Git

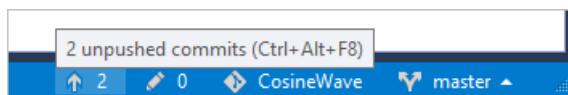
4/9/2019 • 2 minutes to read • [Edit Online](#)

Previous step: [Install packages and manage your Python environment](#)

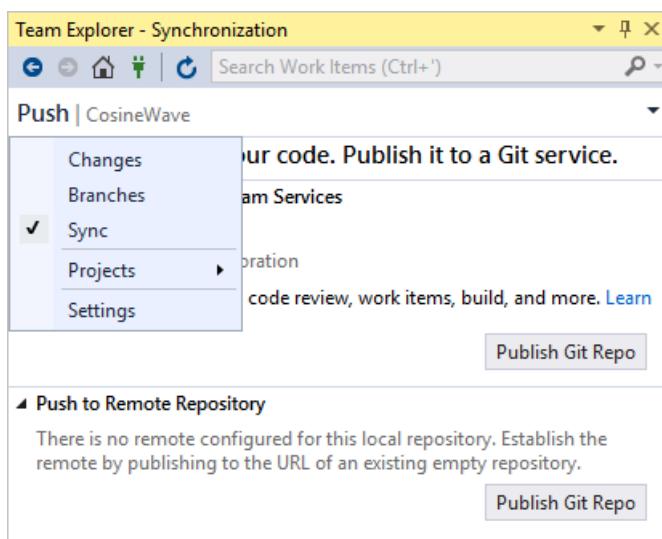
Visual Studio provides direct integration with local Git repositories and remote repositories on services like GitHub and Azure Repos. The integration includes cloning a repository, committing changes, and managing branches.

This article provides a basic overview of creating a local Git repository for an existing project, and familiarizing yourself with some of Visual Studio's Git-related features.

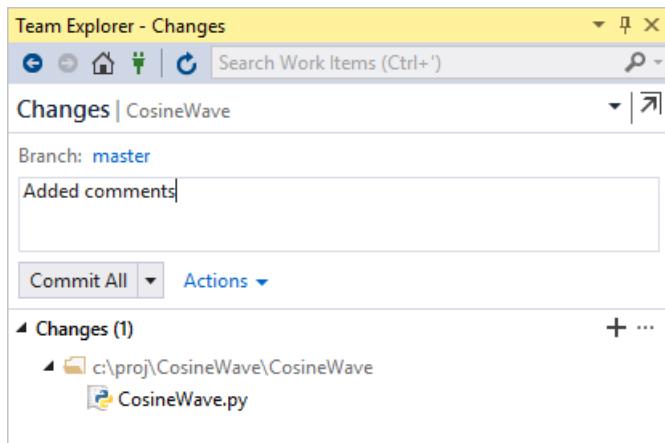
1. With a project open in Visual Studio, such as the project from the [previous step](#), right-click the solution and select **Add Solution to Source Control**. Visual Studio creates a local Git repository that contains your project code.
2. When Visual Studio detects that the project is managed in a Git repository Git-related controls appear along the bottom right corner of the Visual Studio window. The controls show pending commits, changes, the name of the repository, and the branch. Hover over the controls to see additional information.



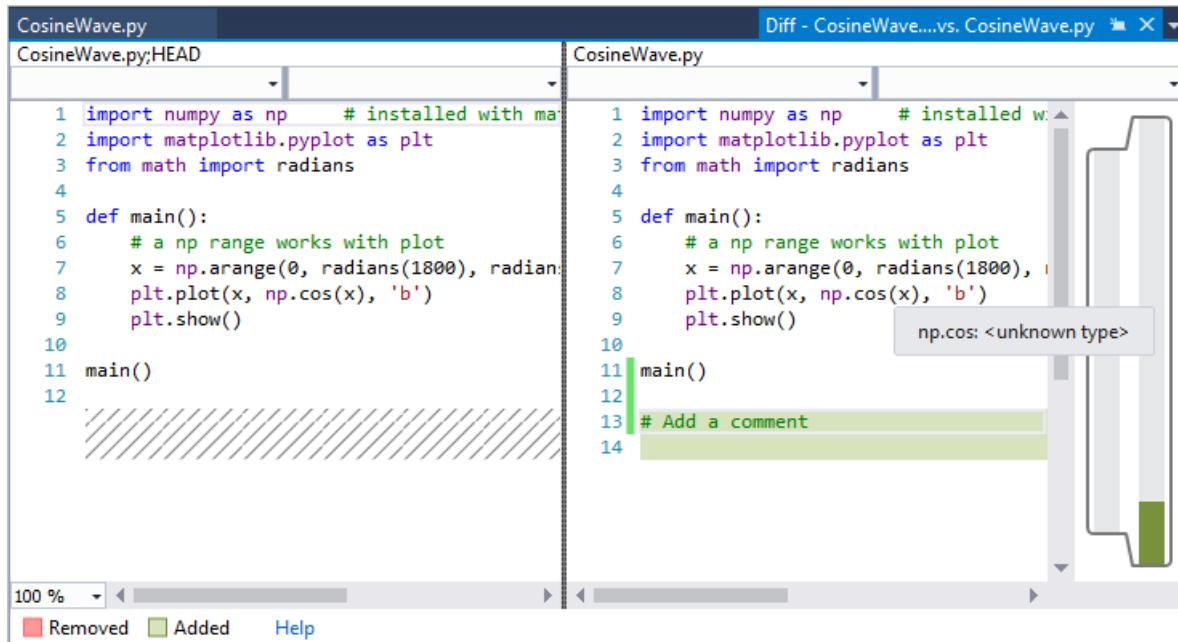
3. When you create a new repository or select any of the Git controls, Visual Studio opens the **Team Explorer** window. (You can open the window at any time with the **View > Team Explorer** menu command.) The window has three main panes, which you switch between using the drop-down on the **Team Explorer** header. The **Sync** pane, which provides publishing operations, also appears when you select the **Push** control (the up arrow icon):



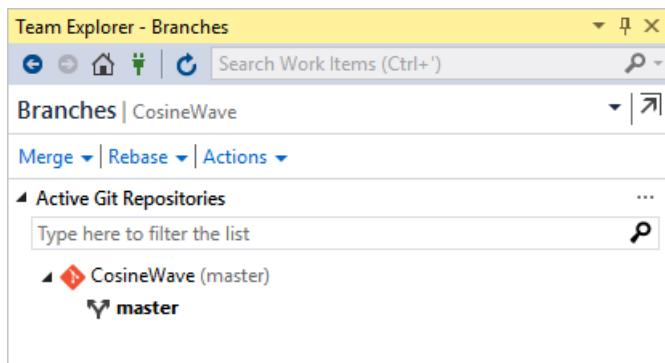
4. Select **Changes** (or the Git control with the pencil icon) to review uncommitted changes and to commit them when desired.



Double-click a file in the **Changes** list to open a diff view for that file:



5. Select **Branches** (or the Git control with a branch name) to examine branches and perform merge and rebase operations:



6. Selecting the Git control with the repository name (**CosineWave** in a previous image), **Team Explorer** shows a **Connect** interface with which you can quickly switch to another repository entirely.
7. When using a local repository, committed changes go directly into the repository. If you're connected to a remote repository, select the drop-down header in **Team Explorer**, choose **Sync** to switch to the **Synchronization** section, and work with the **Pull** and **Fetch** commands presented there.

Go deeper

For a short walkthrough of creating a project from a remote Git repository, see [Quickstart: Clone a repository of](#)

[Python code in Visual Studio](#).

For a much more comprehensive tutorial, including handling merge conflicts, reviewing code with pull requests, rebasing, and cherry-picking changes between branches, see [Get started with Git and Azure Repos](#).

Tutorial review

Congratulations on completing this tutorial on Python in Visual Studio. In this tutorial you've learned how to:

- Create projects and view a project's contents.
- Use the code editor and run a project.
- Use the **Interactive** window to develop new code and easily copy that code into the editor.
- Run a completed program in the Visual Studio debugger.
- Install packages and manage Python environments.
- Work with code in a Git repository.

From here, explore the Concepts and How-to guides, including the following articles:

- [Create a C++ extension for Python](#)
- [Publish to Azure App Service](#)
- [Profiling](#)
- [Unit testing](#)

Tutorial: Get started with the Django web framework in Visual Studio

4/9/2019 • 13 minutes to read • [Edit Online](#)

Django is a high-level Python framework designed for rapid, secure, and scalable web development. This tutorial explores the Django framework in the context of the project templates that Visual Studio provides to streamline the creation of Django-based web apps.

In this tutorial, you learn how to:

- Create a basic Django project in a Git repository using the "Blank Django Web Project" template (step 1)
- Create a Django app with one page and render that page using a template (step 2)
- Serve static files, add pages, and use template inheritance (step 3)
- Use the Django Web Project template to create an app with multiple pages and responsive design (step 4)
- Authenticate users (step 5)
- Use the Polls Django Web Project template to create an app that uses models, database migrations, and customizations to the administrative interface (step 6)

Prerequisites

- Visual Studio 2017 or later on Windows with the following options:
 - The **Python development** workload (**Workload** tab in the installer). For instructions, see [Install Python support in Visual Studio](#).
 - **Git for Windows** and **GitHub Extension for Visual Studio** on the **Individual components** tab under **Code tools**.

Django project templates are also included with all earlier versions of Python Tools for Visual Studio, though details may differ from what's discussed in this tutorial (especially different with earlier versions of the Django framework).

Python development is not presently supported in Visual Studio for Mac. On Mac and Linux, use the [Python extension in Visual Studio Code](#).

"Visual Studio projects" and "Django projects"

In Django terminology, a "Django project" is composed of several site-level configuration files along with one or more "apps" that you deploy to a web host to create a full web application. A Django project can contain multiple apps, and the same app can be in multiple Django projects.

A Visual Studio project, for its part, can contain the Django project along with multiple apps. For the sake of simplicity, whenever this tutorial refers to just a "project," it's referring to the Visual Studio project. When it refers to the "Django project" portion of the web application, it uses "Django project" specifically.

Over the course of this tutorial you'll create a single Visual Studio solution that contains three separate Django projects, each of which contains a single Django app. By keeping the projects in the same solution, you can easily switch back and forth between different files for comparison.

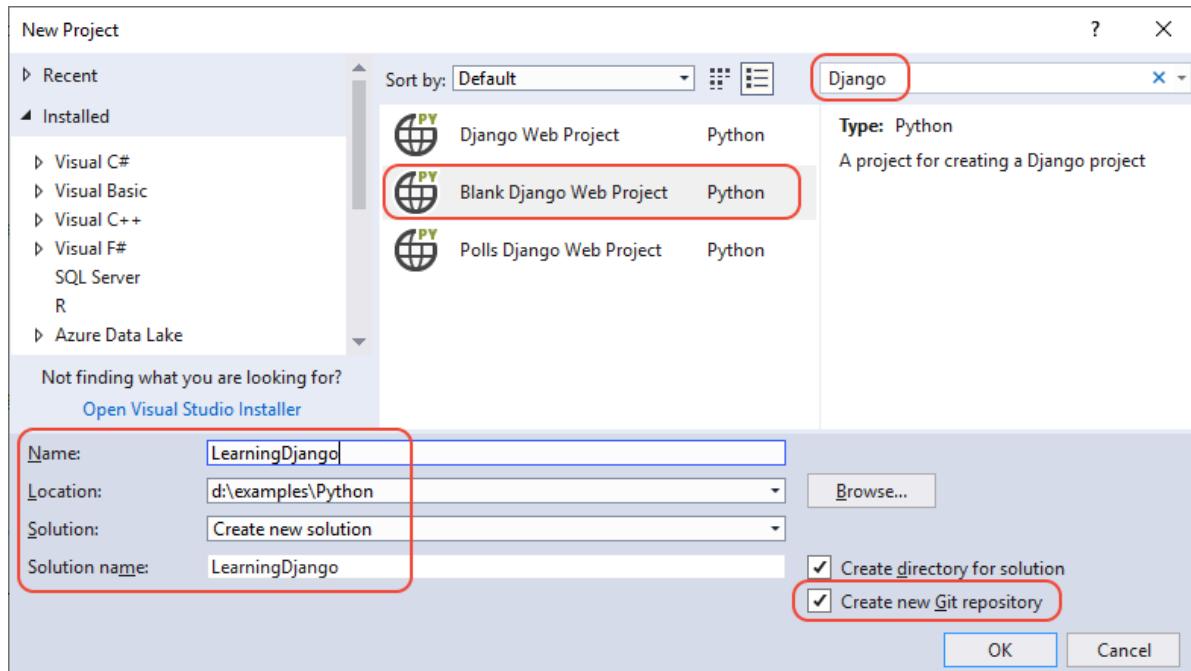
Step 1-1: Create a Visual Studio project and solution

When working with Django from the command line, you typically start a project by running the

`django-admin startproject <project_name>` command. In Visual Studio, using the "Blank Django Web Project"

template provides the same structure within a Visual Studio project and solution.

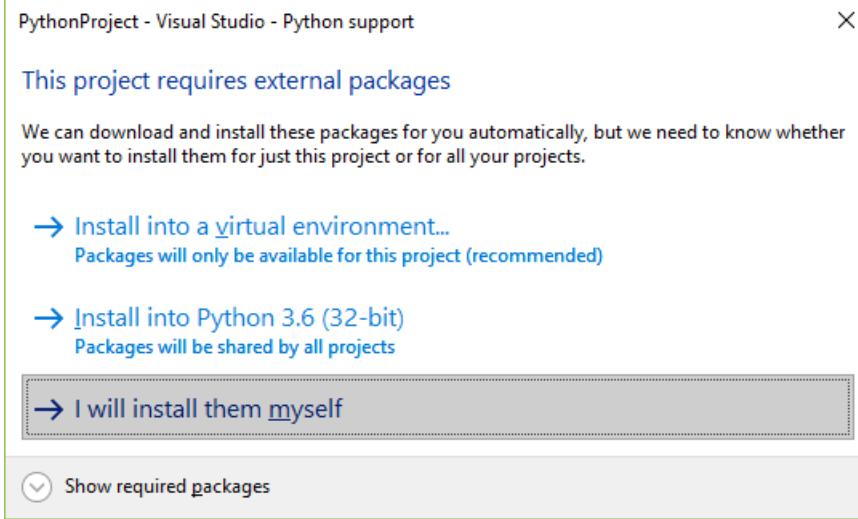
1. In Visual Studio, select **File > New > Project**, search for "Django", and select the **Blank Django Web Project** template. (The template is also found under **Python > Web** in the left-hand list.)



2. In the fields at the bottom of the dialog, enter the following information (as shown in the previous graphic), then select **OK**:

- **Name:** set the name of the Visual Studio project to **BasicProject**. This name is also used for the Django project.
- **Location:** specify a location in which to create the Visual Studio solution and project.
- **Solution:** leave this control set to default **Create new solution** option.
- **Solution name:** set to **LearningDjango**, which is appropriate for the solution as a container for multiple projects in this tutorial.
- **Create directory for solution:** Leave set (the default).
- **Create new Git repository:** Select this option (which is clear by default) so that Visual Studio creates a local Git repository when it creates the solution. If you don't see this option, run the Visual Studio installer and add the **Git for Windows** and **GitHub Extension for Visual Studio** on the **Individual components** tab under **Code tools**.

3. After a moment, Visual Studio prompts you with a dialog saying **This project requires external packages** (shown below). This dialog appears because the template includes a *requirements.txt* file referencing the latest Django 1.x package. (Select **Show required packages** to see the exact dependencies.)



4. Select the option **I will install them myself**. You create the virtual environment shortly to make sure it's excluded from source control. (The environment can always be created from *requirements.txt*.)

Step 1-2: Examine the Git controls and publish to a remote repository

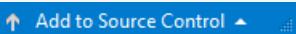
Because you selected the **Create new Git repository** in the **New Project** dialog, the project is already committed to local source control as soon as the creation process is complete. In this step, you familiarize yourself with Visual Studio's Git controls and the **Team Explorer** window in which you work with source control.

1. Examine the Git controls on the bottom corner of the Visual Studio main window. From left to right, these controls show unpushed commits, uncommitted changes, the name of the repository, and the current branch:

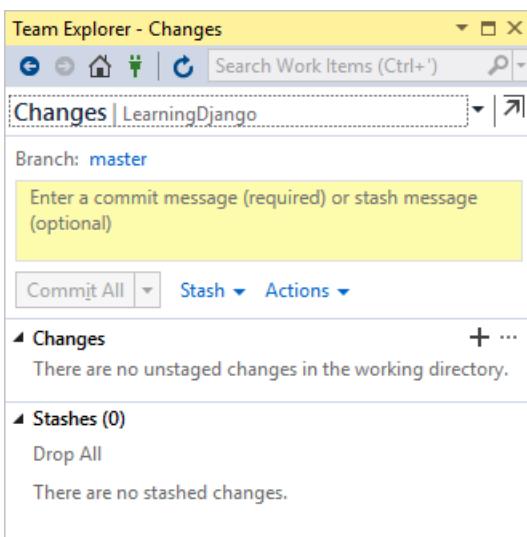


NOTE

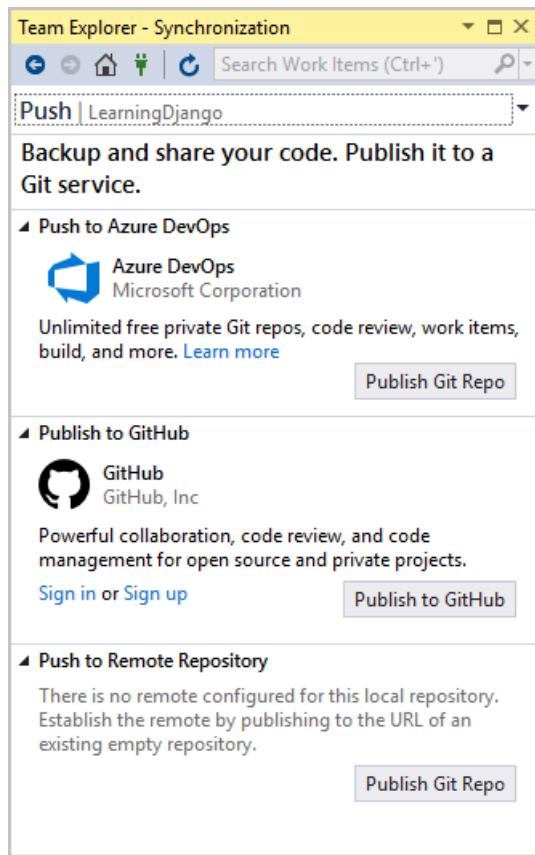
If you don't select the **Create new Git repository** in the **New Project** dialog, the Git controls show only an **Add to source control** command that creates a local repository.



2. Select the changes button, and Visual Studio opens its **Team Explorer** window on the **Changes** page. Because the newly created project is already committed to source control automatically, you don't see any pending changes.

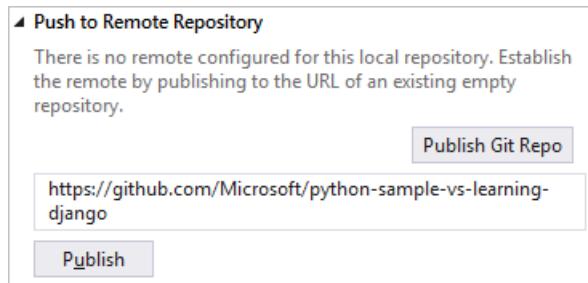


3. On the Visual Studio status bar, select the unpushed commits button (the up arrow with **2**) to open the **Synchronization** page in **Team Explorer**. Because you have only a local repository, the page provides easy options to publish the repository to different remote repositories.



You can choose whichever service you want for your own projects. This tutorial shows the use of GitHub, where the completed sample code for the tutorial is maintained in the [Microsoft/python-sample-vs-learning-django](https://github.com/Microsoft/python-sample-vs-learning-django) repository.

4. When selecting any of the **Publish** controls, **Team Explorer** prompts you for more information. For example, when publishing the sample for this tutorial, the repository itself had to be created first, in which case the **Push to Remote Repository** option was used with the repository's URL.



If you don't have an existing repository, the **Publish to GitHub** and **Push to Azure DevOps** options let you create one directly from within Visual Studio.

5. As you work through this tutorial, get into the habit of periodically using the controls in Visual Studio to commit and push changes. This tutorial reminds you at appropriate points.

TIP

To quickly navigate within **Team Explorer**, select the header (that reads **Changes** or **Push** in the images above) to see a pop-up menu of the available pages.

Question: What are some advantages of using source control from the beginning of a project?

Answer: First of all, using source control from the start, especially if you also use a remote repository, provides a regular offsite backup of your project. Unlike maintaining a project just on a local file system, source control also provides a complete change history and the easy ability to revert a single file or the whole project to a previous state. That change history helps determine the cause of regressions (test failures). Furthermore, source control is essential if multiple people are working on a project, as it manages overwrites and provides conflict resolution. Finally, source control, which is fundamentally a form of automation, sets you up well for automating builds, testing, and release management. It's really the first step in using DevOps for a project, and because the barriers to entry are so low, there's really no reason to not use source control from the beginning.

For further discussion on source control as automation, see [The Source of Truth: The Role of Repositories in DevOps](#), an article in MSDN Magazine written for mobile apps that applies also to web apps.

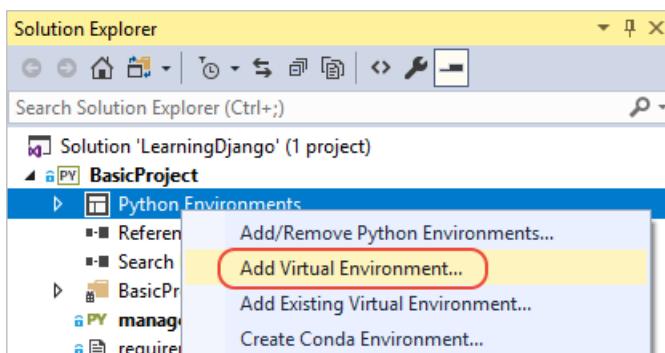
Question: Can I prevent Visual Studio from auto-committing a new project?

Answer: Yes. To disable auto-commit, go to the **Settings** page in **Team Explorer**, select **Git > Global settings**, clear the option labeled **Commit changes after merge by default**, then select **Update**.

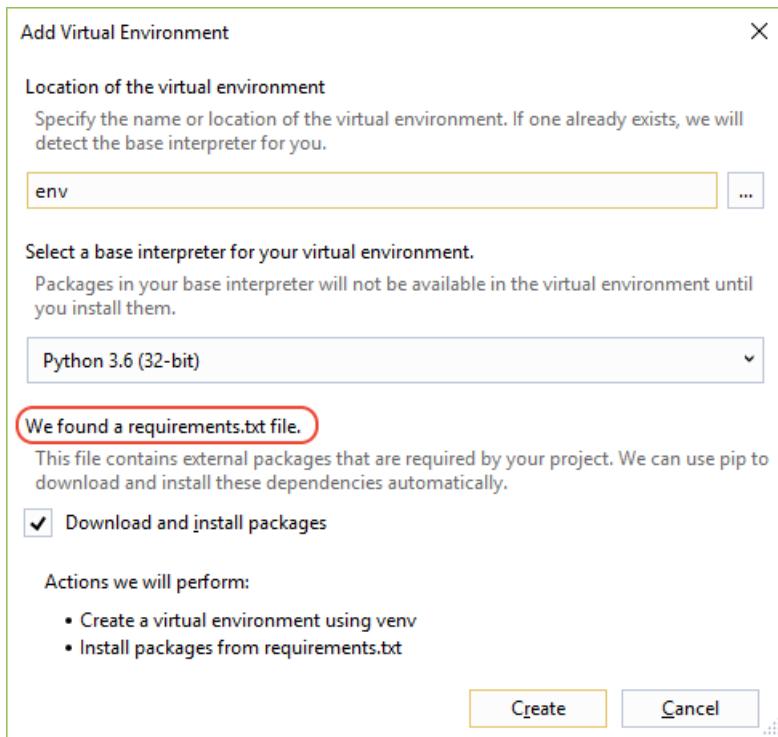
Step 1-3: Create the virtual environment and exclude it from source control

Now that you've configured source control for your project, you can create the virtual environment that contains the necessary Django packages for the project. You can then use **Team Explorer** to exclude the environment's folder from source control.

1. In **Solution Explorer**, right-click the **Python Environments** node and select **Add Virtual Environment**.



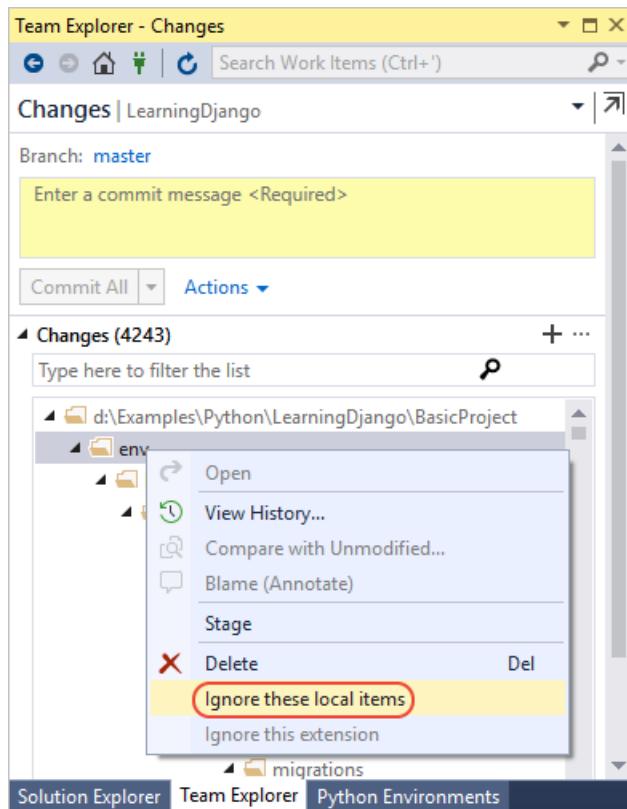
2. An **Add Virtual Environment** dialog appears, with a message saying **We found a requirements.txt file**. This message indicates that Visual Studio uses that file to configure the virtual environment.



3. Select **Create** to accept the defaults. (You can change the name of the virtual environment if you want, which just changes the name of its subfolder, but `env` is a standard convention.)
4. Consent to administrator privileges if prompted, then be patient for a few minutes while Visual Studio downloads and installs packages, which for Django means expanding several thousand files in about as many subfolders! You can see progress in the Visual Studio **Output** window. While you're waiting, ponder the Question sections that follow.
5. On the Visual Studio Git controls (on the status bar), select the changes indicator (that shows **99***) which opens the **Changes** page in **Team Explorer**.

Creating the virtual environment brought in thousands of changes, but you don't need to include any of them in source control because you (or anyone else cloning the project) can always recreate the environment from *requirements.txt*.

To exclude the virtual environment, right-click the `env` folder and select **Ignore these local items**.



6. After excluding the virtual environment, the only remaining changes are to the project file and `.gitignore`. The `.gitignore` file contains an added entry for the virtual environment folder. You can double-click the file to see a diff.
7. Enter a commit message and select the **Commit All** button, then push the commits to your remote repository if you like.

Question: Why do I want to create a virtual environment?

Answer: A virtual environment is a great way to isolate your app's exact dependencies. Such isolation avoids conflicts within a global Python environment, and aids both testing and collaboration. Over time, as you develop an app, you invariably bring in many helpful Python packages. By keeping packages in a project-specific virtual environment, you can easily update the project's `requirements.txt` file that describes that environment, which is included in source control. When the project is copied to any other computers, including build servers, deployment servers, and other development computers, it's easy to recreate the environment using only `requirements.txt` (which is why the environment doesn't need to be in source control). For more information, see [Use virtual environments](#).

Question: How do I remove a virtual environment that's already committed to source control?

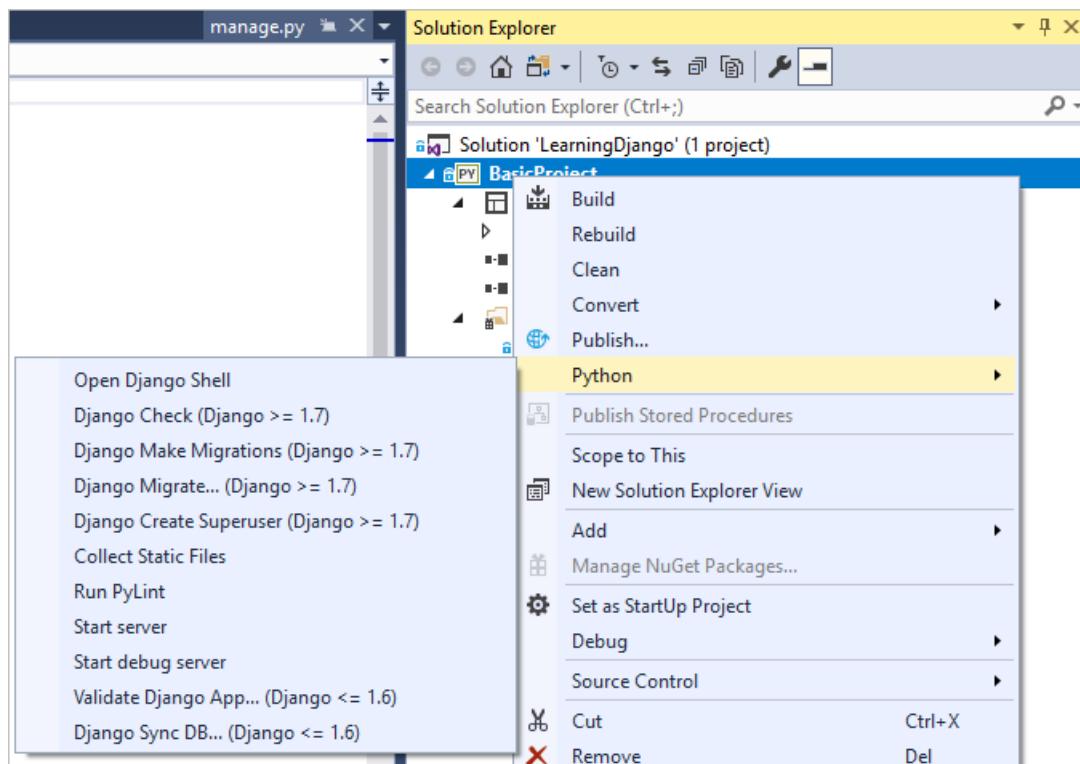
Answer: First, edit your `.gitignore` file to exclude the folder: find the section at the end with the comment `# Python Tools for Visual Studio (PTVS)` and add a new line for the virtual environment folder, such as `/BasicProject/env`. (Because Visual Studio doesn't show the file in **Solution Explorer**, open it directly using the **File > Open > File** menu command. You can also open the file from **Team Explorer**: on the **Settings** page, select **Repository Settings**, go to the **Ignore & Attributes Files** section, then select the **Edit** link next to `.gitignore`.)

Second, open a command window, navigate to the folder like `BasicProject` that contains the virtual environment folder such as `env`, and run `git rm -r env`. Then commit those changes from the command line (`git commit -m 'Remove venv'`) or commit from the **Changes** page of **Team Explorer**.

Step 1-4: Examine the boilerplate code

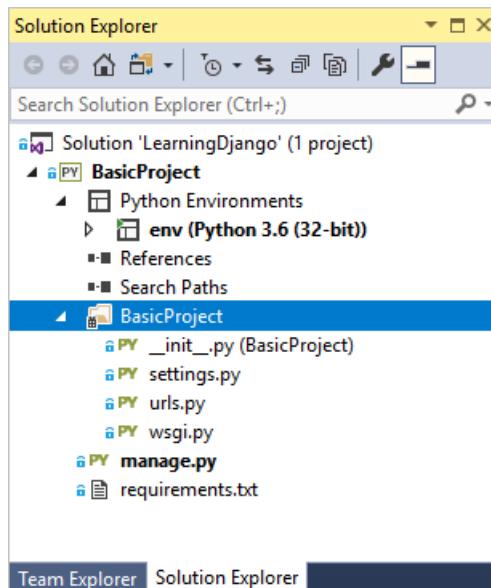
Once project creation completes, examine the boilerplate Django project code (which is again the same as generated by the CLI command `django-admin startproject <project_name>`).

1. In your project root is `manage.py`, the Django command-line administrative utility that Visual Studio automatically sets as the project startup file. You run the utility on the command line using `python manage.py <command> [options]`. For common Django tasks, Visual Studio provides convenient menu commands. Right-click the project in **Solution Explorer** and select **Python** to see the list. You encounter some of these commands in the course of this tutorial.



2. In your project is a folder named the same as the project. It contains the basic Django project files:

- `__init__.py`: an empty file that tells Python that this folder is a Python package.
- `wsgi.py`: an entry point for WSGI-compatible web servers to serve your project. You typically leave this file as-is as it provides the hooks for production web servers.
- `settings.py`: contains settings for Django project, which you modify in the course of developing a web app.
- `urls.py`: contains a table of contents for the Django project, which you also modify in the course of development.



3. As noted earlier, the Visual Studio template also adds a `requirements.txt` file to your project specifying the Django package dependency. The presence of this file is what invites you to create a virtual environment when first creating the project.

Question: Can Visual Studio generate a requirements.txt file from a virtual environment after I install other packages?

Answer: Yes. Expand the **Python Environments** node, right-click your virtual environment, and select the **Generate requirements.txt** command. It's good to use this command periodically as you modify the environment, and commit changes to `requirements.txt` to source control along with any other code changes that depend on that environment. If you set up continuous integration on a build server, you should generate the file and commit changes whenever you modify the environment.

Step 1-5: Run the empty Django project

1. In Visual Studio, select **Debug > Start Debugging (F5)** or use the **Web Server** button on the toolbar (the browser you see may vary):



2. Running the server means running the command `manage.py runserver <port>`, which starts Django's built-in development server. If Visual Studio says **Failed to start debugger** with a message about having no startup file, right-click **manage.py** in **Solution Explorer** and select **Set as Startup File**.
3. When you start the server, you see a console window open that also displays the server log. Visual Studio automatically opens a browser to `http://localhost:<port>`. Because the Django project has no apps, however, Django shows only a default page to acknowledge that what you have so far is working fine:

It worked!
Congratulations on your first Django-powered page.

Next, start your first app by running `python manage.py startapp [app_label]`.

You're seeing this message because you have `DEBUG = True` in your Django settings file and you haven't configured any URLs. Get to work!

4. When you're done, stop the server by closing the console window, or by using the **Debug > Stop Debugging** command in Visual Studio.

Question: Is Django a web server as well as a framework?

Answer: Yes and no. Django does have a built-in web server that's used for development purposes. This web server is what gets used when you run the web app locally, such as when debugging in Visual Studio. When you deploy to a web host, however, Django uses the host's web server instead. The `wsgi.py` module in the Django project takes care of hooking into the production servers.

Question: What's the difference between using the Debug menu commands and the server commands on the project's Python submenu?

Answer: In addition to the **Debug** menu commands and toolbar buttons, you can also launch the server using the **Python > Run server** or **Python > Run debug server** commands on the project's context menu. Both commands open a console window in which you see the local URL (`localhost:port`) for the running server. However, you must manually open a browser with that URL, and running the debug server does not automatically start the Visual Studio debugger. You can attach a debugger to the running process later, if you want, using the **Debug > Attach to Process** command.

Next steps

At this point, the basic Django project does not contain any apps. You create an app in the next step. Because you

typically work with Django apps more than the Django project, you won't need to know much more about the boilerplate files at present.

[Create a Django app with views and page templates](#)

Go deeper

- Django project code: [Writing your first Django app, part 1](#) (docs.djangoproject.com)
- Administrative utility: [django-admin and manage.py](#) (docs.djangoproject.com)
- Tutorial source code on GitHub: [Microsoft/python-sample-vs-learning-django](#)

Step 2: Create a Django app with views and page templates

4/9/2019 • 14 minutes to read • [Edit Online](#)

Previous step: [Create a Visual Studio project and solution](#)

What you have so far in the Visual Studio project are only the site-level components of a Django *project*, which can run one or more Django *apps*. The next step is to create your first app with a single page.

In this step you now learn how to:

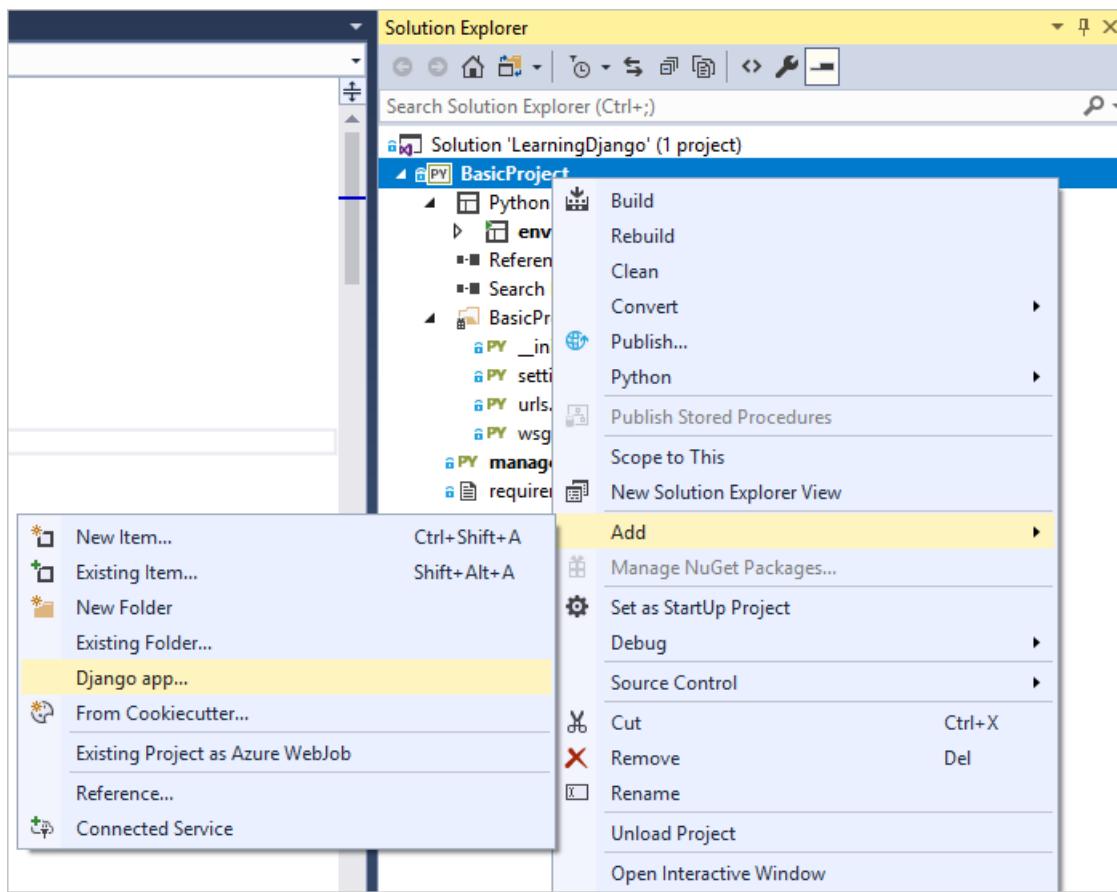
- Create a Django app with a single page (step 2-1)
- Run the app from the Django project (step 2-2)
- Render a view using HTML (step 2-3)
- Render a view using a Django page template (step 2-4)

Step 2-1: Create an app with a default structure

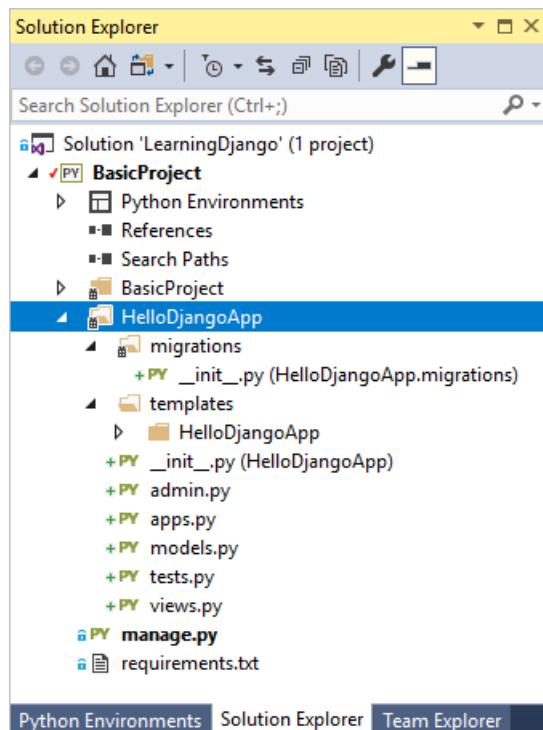
A Django app is a separate Python package that contains a set of related files for a specific purpose. A Django project can contain any number of apps, which reflects the fact that a web host can serve any number of separate entry points from a single domain name. For example, a Django project for a domain like contoso.com might contain one app for `www.contoso.com`, a second app for support.contoso.com, and a third app for docs.contoso.com. In this case, the Django project handles site-level URL routing and settings (in its `urls.py` and `settings.py` files), while each app has its own distinct styling and behavior through its internal routing, views, models, static files, and administrative interface.

A Django app typically begins with a standard set of files. Visual Studio provides item templates to initialize a Django app within a Django project, along with an integrated menu command that serves the same purpose:

- Templates: In **Solution Explorer**, right-click the project and select **Add > New item**. In the **Add New Item** dialog, select the **Django 1.9 App** template, specify the app name in the **Name** field, and select **OK**.
- Integrated command: In **Solution Explorer**, right-click the project and select **Add > Django app**. This command prompts you for a name and creates a Django 1.9 app.



Using either method, create an app with the name "HelloDjangoApp". The result is a folder in your project with that name that contains items as described in the table that follows.



ITEM	DESCRIPTION
__init__.py	The file that identifies the app as a package.

ITEM	DESCRIPTION
migrations	A folder in which Django stores scripts that update the database to align with changes to the models. Django's migration tools then apply the necessary changes to any previous version of the database so that it matches the current models. Using migrations, you keep your focus on your models and let Django handle the underlying database schema. Migrations are discussed in step 6; for now, the folder simply contains an <code>__init__.py</code> file (indicating that the folder defines its own Python package).
templates	A folder for Django page templates containing a single file <code>index.html</code> within a folder matching the app name. (In Visual Studio 2017 15.7 and earlier, the file is contained directly under <code>templates</code> and step 2-4 instructs you to create the subfolder.) Templates are blocks of HTML into which views can add information to dynamically render a page. Page template "variables," such as <code>{{ content }}</code> in <code>index.html</code> , are placeholders for dynamic values as explained later in this article (step 2). Typically Django apps create a namespace for their templates by placing them in a subfolder that matches the app name.
admin.py	The Python file in which you extend the app's administrative interface (see step 6), which is used to seed and edit data in a database. Initially, this file contains only the statement, <code>from django.contrib import admin</code> . By default, Django includes a standard administrative interface through entries in the Django project's <code>settings.py</code> file, which you can turn on by uncommenting existing entries in <code>urls.py</code> .
apps.py	A Python file that defines a configuration class for the app (see below, after this table).
models.py	Models are data objects, identified by functions, through which views interact with the app's underlying database (see step 6). Django provides the database connection layer so that apps don't need to concern themselves with those details. The <code>models.py</code> file is a default place in which to create your models, and initially contains only the statement, <code>from django.db import models</code> .
tests.py	A Python file that contains the basic structure of unit tests.
views.py	Views are what you typically think of as web pages, which take an HTTP request and return an HTTP response. Views typically render as HTML that web browsers know how to display, but a view doesn't necessarily have to be visible (like an intermediate form). A view is defined by a Python function whose responsibility is to render the HTML to send to the browser. The <code>views.py</code> file is a default place in which to create views, and initially contains only the statement, <code>from django.shortcuts import render</code> .

The contents of `app.py` appears as follows when using the name "HelloDjangoApp":

```
from django.apps import AppConfig

class HelloDjangoAppConfig(AppConfig):
    name = 'HelloDjango'
```

Question: Is creating a Django app in Visual Studio any different from creating an app on the command line?

Answer: Running the **Add > Django app** command or using **Add > New Item** with a Django app template produces the same files as the Django command `manage.py startapp <app_name>`. The benefit to creating the app in Visual Studio is that the app folder and all its files are automatically integrated into the project. You can use the same Visual Studio command to create any number of apps in your project.

Step 2-2: Run the app from the Django project

At this point, if you run the project again in Visual Studio (using the toolbar button or **Debug > Start Debugging**), you still see the default page. No app content appears because you need to define an app-specific page and add the app to the Django project:

1. In the *HelloDjangoApp* folder, modify *views.py* to match the code below, which defines a view named "index":

```
from django.shortcuts import render
from django.http import HttpResponse

def index(request):
    return HttpResponse("Hello, Django!")
```

2. In the *BasicProject* folder (created in step 1), modify *urls.py* to at least match the following code (you can retain the instructive comments if you like):

```
from django.conf.urls import include, url
import HelloDjangoApp.views

# Django processes URL patterns in the order they appear in the array
urlpatterns = [
    url(r'^$', HelloDjangoApp.views.index, name='index'),
    url(r'^home$', HelloDjangoApp.views.index, name='home'),
]
```

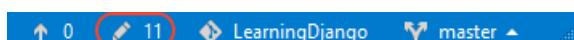
Each URL pattern describes the views to which Django routes specific site-relative URLs (that is, the portion that follows `https://www.domain.com/`). The first entry in `urlPatterns` that starts with the regular expression `^$` is the routing for the site root, `/`. The second entry, `^home$` specifically routes `/home`. You can have any number of routings to the same view.

3. Run the project again to see the message **Hello, Django!** as defined by the view. Stop the server when you're done.

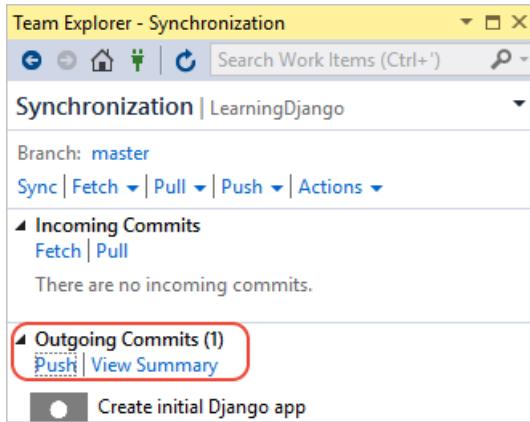
Commit to source control

Because you've made changes to your code and have tested them successfully, now is a great time to review and commit your changes to source control. Later steps in this tutorial remind you of appropriate times to commit to source control again, and refer you back to this section.

1. Select the changes button along the bottom of Visual Studio (circled below), which navigates to **Team Explorer**.



2. In **Team Explorer**, enter a commit message like "Create initial Django app" and select **Commit All**. When the commit is complete, you see a message **Commit <hash> created locally. Sync to share your changes with the server**. If you want to push changes to your remote repository, select **Sync**, then select **Push** under **Outgoing Commits**. You can also accumulate multiple local commits before pushing to remote.



Question: What is the 'r' prefix before the routing strings for?

Answer: The 'r' prefix on a string in Python means "raw," which instructs Python to not escape any characters within the string. Because regular expressions use many special characters, using the 'r' prefix makes those strings much easier to read than if they contained a number of '\' escape characters.

Question: What do the ^ and \$ characters mean in the URL routing entries?

Answer: In the regular expressions that define URL patterns, '^' means "start of line" and '\$' means "end of line," where again the URLs are relative to the site root (the part that follows `https://www.domain.com/`). The regular expression `^$` effectively means "blank" and therefore matches the full URL `https://www.domain.com/` (nothing added to the site root). The pattern `^home$` matches exactly `https://www.domain.com/home/`. (Django doesn't use the trailing / in pattern matching.)

If you don't use a trailing \$ in a regular expression, as with `^home`, then URL pattern matches *any* URL that begins with "home" such as "home", "homework", "homestead", and "home192837".

To experiment with different regular expressions, try online tools such as regex101.com at pythex.org.

Step 2-3: Render a view using HTML

The `index` function that you have so far in `views.py` generates nothing more than a plain-text HTTP response for the page. Most real-world web pages, of course, respond with rich HTML pages that often incorporate live data. Indeed, the primary reason to define a view using a function is so you can generate that content dynamically.

Because the argument to `HttpResponse` is just a string, you can build up any HTML you like within a string. As a simple example, replace the `index` function with the following code (keeping the existing `from` statements), which generates an HTML response using dynamic content that's updated every time you refresh the page:

```
from datetime import datetime

def index(request):
    now = datetime.now()

    html_content = "<html><head><title>Hello, Django</title></head><body>"
    html_content += "<strong>Hello Django!</strong> on " + now.strftime("%A, %d %B, %Y at %X")
    html_content += "</body></html>"

    return HttpResponse(html_content)
```

Run the project again to see a message like "**Hello Django!**" on Monday, 16 April, 2018 at 16:28:10". Refresh the page to update the time and confirm that the content is being generated with each request. Stop the server when you're done.

TIP

A shortcut to stopping and restarting the project is to use the **Debug > Restart** menu command (**Ctrl+Shift+F5**) or the **Restart** button on the debugging toolbar:



Step 2-4: Render a view using a page template

Generating HTML in code works fine for very small pages, but as pages get more sophisticated you typically want to maintain the static HTML parts of your page (along with references to CSS and JavaScript files) as "page templates" into which you then insert dynamic, code-generated content. In the previous section, only the date and time from the `now.strftime` call is dynamic, which means all the other content can be placed in a page template.

A Django page template is a block of HTML that can contain any number of replacement tokens called "variables" that are delineated by `{{` and `}}`, as in `{{ content }}`. Django's templating module then replaces variables with dynamic content that you provide in code.

The following steps demonstrate the use of page templates:

- Under the *BasicProject* folder, which contains the Django project, open `settings.py` file and add the app name, "HelloDjangoApp", to the `INSTALLED_APPS` list. Adding the app to the list tells the Django project that there's a folder of that name containing an app:

```
INSTALLED_APPS = [
    'HelloDjangoApp',
    # Other entries...
]
```

- Also in `settings.py`, make sure the `TEMPLATES` object contains the following line (included by default), which instructs Django to look for templates in an installed app's *templates* folder:

```
'APP_DIRS': True,
```

- In the *HelloDjangoApp* folder, open the `templates/HelloDjangoApp/index.html` page template file (or `templates/index.html` in VS 2017 15.7 and earlier), to observe that it contains one variable, `{{ content }}`:

```
<html>
<head><title></title></head>

<body>

{{ content }}

</body>
</html>
```

- In the *HelloDjangoApp* folder, open `views.py` and replace the `index` function with the following code that uses the `django.shortcuts.render` helper function. The `render` helper provides a simplified interface for working with page templates. Be sure to keep all existing `from` statements.

```

from django.shortcuts import render # Added for this step

def index(request):
    now = datetime.now()

    return render(
        request,
        "HelloDjangoApp/index.html", # Relative path from the 'templates' folder to the template file
        # "index.html", # Use this code for VS 2017 15.7 and earlier
        {
            'content': "<strong>Hello Django!</strong> on " + now.strftime("%A, %d %B, %Y at %X")
        }
    )

```

The first argument to `render`, as you can see, is the request object, followed by the relative path to the template file within the app's *templates* folder. A template file is named for the view it supports, if appropriate. The third argument to `render` is then a dictionary of variables that the template refers to. You can include objects in the dictionary, in which case a variable in the template can refer to `{{ object.property }}`.

- Run the project and observe the output. You should see a similar message to that seen in step 2-2, indicating that the template works.

Observe, however, that the HTML you used in the `content` property renders only as plain text because the `render` function automatically escapes that HTML. Automatic escaping prevent accidental vulnerabilities to injection attacks: developers often gather input from one page and use it as a value in another through a template placeholder. Escaping also serves as a reminder that it's again best to keep HTML in the page template and out of the code. Fortunately, it's a simple matter to create additional variables where needed. For example, change *index.html* with *templates* to match the following markup, which adds a page title and keeps all formatting in the page template:

```

<html>
    <head>
        <title>{{ title }}</title>
    </head>
    <body>
        <strong>{{ message }}</strong>{{ content }}
    </body>
</html>

```

Then write the `index` view function as follows, to provide values for all the variables in the page template:

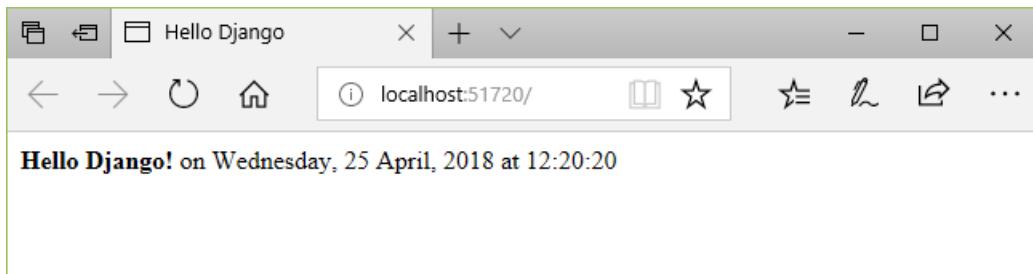
```

def index(request):
    now = datetime.now()

    return render(
        request,
        "HelloDjangoApp/index.html", # Relative path from the 'templates' folder to the template file
        # "index.html", # Use this code for VS 2017 15.7 and earlier
        {
            'title' : "Hello Django",
            'message' : "Hello Django!",
            'content' : " on " + now.strftime("%A, %d %B, %Y at %X")
        }
    )

```

- Stop the server and restart the project, and observe that the page now renders properly:



7. Visual Studio 2017 version 15.7 and earlier: As a final step, move your templates into a subfolder named the same as your app, which creates a namespace and avoids potential conflicts with other apps you might add to the project. (The templates in VS 2017 15.8+ do this for you automatically.) That is, create a subfolder in `templates` named `HelloDjangoApp`, move `index.html` into that subfolder, and modify the `index` view function to refer to the template's new path, `HelloDjangoApp/index.html`. Then run the project, verify that the page renders properly, and stop the server.
8. Commit your changes to source control and update your remote repository, if desired, as described under [step 2-2](#).

Question: Do page templates have to be in a separate file?

Answer: Although templates are usually maintained in separate HTML files, you can also use an inline template. Using a separate file is recommended, however, to maintain a clean separation between markup and code.

Question: Must templates use the .html file extension?

Answer: The `.html` extension for page template files is entirely optional, because you always identify the exact relative path to the file in the second argument to the `render` function. However, Visual Studio (and other editors) typically give you features like code completion and syntax coloration with `.html` files, which outweighs the fact that page templates are not strictly HTML.

In fact, when you're working with a Django project, Visual Studio automatically detects when the HTML file you're editing is actually a Django template, and provides certain auto-complete features. For example, when you start typing a Django page template comment, `{#`, Visual Studio automatically gives you the closing `#}` characters. The **Comment Selection** and **Uncomment Selection** commands (on the **Edit > Advanced** menu and on the toolbar) also use template comments instead of HTML comments.

Question: When I run the project, I see an error that the template cannot be found. What's wrong?

Answer: If you see errors that the template cannot be found, make sure you added the app to the Django project's `settings.py` in the `INSTALLED_APPS` list. Without that entry, Django won't know to look in the app's `templates` folder.

Question: Why is template namespacing important?

Answer: When Django looks for a template referred to in the `render` function, it uses whatever file it finds first that matches the relative path. If you have multiple Django apps in the same project that use the same folder structures for templates, it's likely that one app will unintentionally use a template from another app. To avoid such errors, always create a subfolder under an app's `templates` folder that matches the name of the app to avoid any and all duplication.

Next steps

[Serve static files, add pages, and use template inheritance](#)

Go deeper

- [Writing your first Django app, part 1 - views](#) (docs.djangoproject.com)
- For more capabilities of Django templates, such as includes and inheritance, see [The Django template language](#) (docs.djangoproject.com)

- Regular expression training on inLearning (LinkedIn)
- Tutorial source code on GitHub: [Microsoft/python-sample-vs-learning-django](https://github.com/Microsoft/python-sample-vs-learning-django)

Step 3: Serve static files, add pages, and use template inheritance

4/9/2019 • 9 minutes to read • [Edit Online](#)

Previous step: [Create a Django app with views and page templates](#)

In the previous steps of this tutorial, you've learned how to create a minimal Django app with a single page of self-contained HTML. Modern web apps, however, are typically composed of many pages, and make use of shared resources like CSS and JavaScript files to provide consistent styling and behavior.

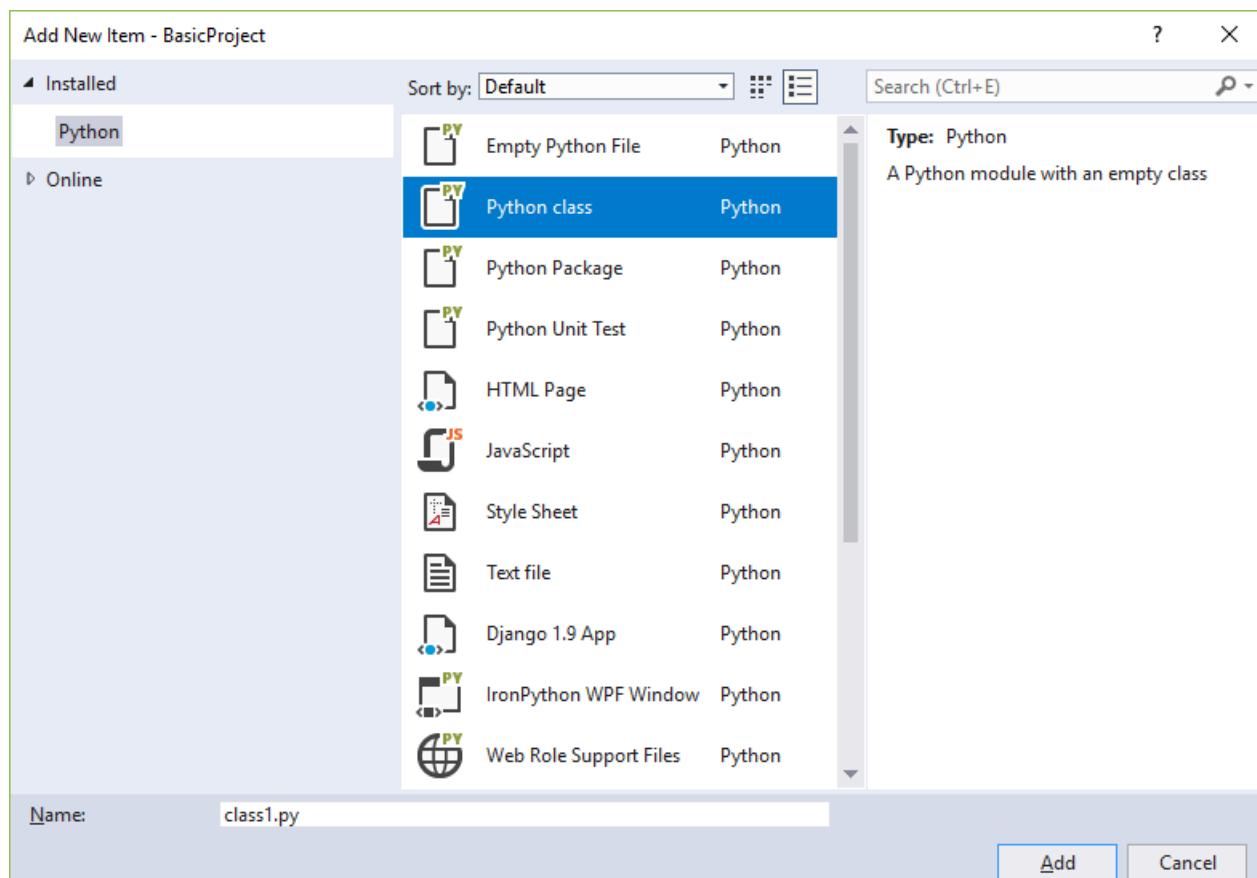
In this step, you learn how to:

- Use Visual Studio item templates to quickly add new files of different types with convenient boilerplate code (step 3-1)
- Configure the Django project to serve static files (step 3-2)
- Add additional pages to the app (step 3-3)
- Use template inheritance to create a header and nav bar that's used across pages (step 3-4)

Step 3-1: Become familiar with item templates

As you develop a Django app, you typically add many more Python, HTML, CSS, and JavaScript files. For each file type (as well as other files like `web.config` that you may need for deployment), Visual Studio provides convenient [item templates](#) to get you started.

To see available templates, go to **Solution Explorer**, right-click the folder in which you want to create the item, select **Add > New Item**:



To use a template, select the desired template, specify a name for the file, and select **OK**. Adding an item in this manner automatically adds the file to your Visual Studio project and marks the changes for source control.

Question: How does Visual Studio know which item templates to offer?

Answer: The Visual Studio project file (`.pyproj`) contains a project type identifier that marks it as a Python project. Visual Studio uses this type identifier to show only those item templates that are suitable for the project type. This way, Visual Studio can supply a rich set of item templates for many project types without asking you to sort through them every time.

Step 3-2: Serve static files from your app

In a web app built with Python (using any framework), your Python files always run on the web host's server and are never transmitted to a user's computer. Other files, however, such as CSS and JavaScript, are used exclusively by the browser, so the host server simply delivers them as-is whenever they're requested. Such files are referred to as "static" files, and Django can deliver them automatically without you needing to write any code.

A Django project is configured by default to serve static files from the app's `static` folder, thanks to these lines in the Django project's `settings.py`:

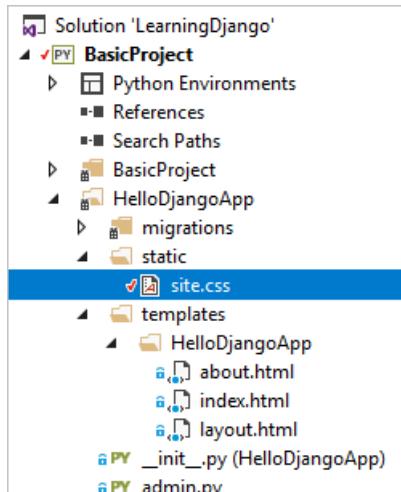
```
# Static files (CSS, JavaScript, Images)
# https://docs.djangoproject.com/en/1.9/howto/static-files/

STATIC_URL = '/static/'

STATIC_ROOT = posixpath.join(*(BASE_DIR.split(os.path.sep) + ['static']))
```

You can organize files using any folder structure within `static` that you like, and then use relative paths within that folder to refer to the files. To demonstrate this process, the following steps add a CSS file to the app, then use that stylesheet in the `index.html` template:

1. In **Solution Explorer**, right-click the **HelloDjangoApp** folder in the Visual Studio project, select **Add > New folder**, and name the folder `static`.
2. Right-click the **static** folder and select **Add > New item**. In the dialog that appears, select the **Stylesheet** template, name the file `site.css`, and select **OK**. The `site.css` file appears in the project and is opened in the editor. Your folder structure should appear similar to the following image:



3. Replace the contents of `site.css` with the following code and save the file:

```
.message {  
    font-weight: 600;  
    color: blue;  
}
```

- Replace the contents of the app's `templates/HelloDjangoApp/index.html` file with the following code, which replaces the `` element used in step 2 with a `` that references the `message` style class. Using a style class in this way gives you much more flexibility in styling the element. (If you haven't moved `index.html` into a subfolder in `templates` when using VS 2017 15.7 and earlier, refer to [template namespacing](#) in step 2-4.)

```
<html>  
    <head>  
        <title>{{ title }}</title>  
        {% load staticfiles %} <!-- Instruct Django to load static files -->  
        <link rel="stylesheet" type="text/css" href="{% static 'site.css' %}" />  
    </head>  
    <body>  
        <span class="message">{{ message }}</span>{{ content }}  
    </body>  
</html>
```

- Run the project to observe the results. Stop the server when done, and commit your changes to source control if you like (as explained in [step 2](#)).

Question: What is the purpose of the `{% load staticfiles %}` tag?

Answer: The `{% load staticfiles %}` line is required before referring to static files in elements like `<head>` and `<body>`. In the example shown in this section, "staticfiles" refers to a custom Django template tag set, which is what allows you to use the `{% static %}` syntax to refer to static files. Without `{% load staticfiles %}`, you'll see an exception when the app runs.

Question: Are there any conventions for organizing static files?

Answer: You can add other CSS, JavaScript, and HTML files in your `static` folder however you want. A typical way to organize static files is to create subfolders named `fonts`, `scripts`, and `content` (for stylesheets and any other files). In each case, remember to include those folders in the relative path to the file in `{% static %}` references.

Step 3-3: Add a page to the app

Adding another page to the app means the following:

- Add a Python function that defines the view.
- Add a template for the page's markup.
- Add the necessary routing to the Django project's `urls.py` file.

The following steps add an "About" page to the "HelloDjangoApp" project, and links to that page from the home page:

- In **Solution Explorer**, right-click the `templates/HelloDjangoApp` folder, select **Add > New item**, select the **HTML Page** item template, name the file `about.html`, and select **OK**.

TIP

If the **New Item** command doesn't appear on the **Add** menu, make sure that you've stopped the server so that Visual Studio exits debugging mode.

- Replace the contents of `about.html` with the following markup (you replace the explicit link to the home page with a simple navigation bar in step 3-4):

```
<html>
  <head>
    <title>{{ title }}</title>
    {% load staticfiles %}
    <link rel="stylesheet" type="text/css" href="{% static 'site.css' %}" />
  </head>
  <body>
    <div><a href="home">Home</a></div>
    {{ content }}
  </body>
</html>
```

- Open the app's `views.py` file and add a function named `about` that uses the template:

```
def about(request):
    return render(
        request,
        "HelloDjangoApp/about.html",
        {
            'title' : "About HelloDjangoApp",
            'content' : "Example app page for Django."
        }
    )
```

- Open the Django project's `urls.py` file and add the following line to the `urlPatterns` array:

```
url(r'^about$', HelloDjangoApp.views.about, name='about'),
```

- Open the `templates/HelloDjangoApp/index.html` file and add the following line below the `<body>` element to link to the About page (again, you replace this link with a nav bar in step 3-4):

```
<div><a href="about">About</a></div>
```

- Save all the files using the **File > Save All** menu command, or just press **Ctrl+Shift+S**. (Technically, this step isn't needed as running the project in Visual Studio saves files automatically. Nevertheless, it's a good command to know about!)

- Run the project to observe the results and check navigation between pages. Close the server when done.

Question: I tried using "index" for the link to the home page, but it didn't work. Why?

Answer: Even though the view function in `views.py` is named `index`, the URL routing patterns in the Django project's `urls.py` file does not contain a regular expression that matches the string "index". To match that string, you need to add another entry for the pattern `^index$`.

As shown in the next section, it's much better to use the `{% url '<pattern_name>' %}` tag in the page template to refer to the `name` of a pattern, in which case Django creates the proper URL for you. For example, replace

`<div>Home</div>` in `about.html` with `<div>Home</div>`. The use of 'index' works here because the first URL pattern in `urls.py` is, in fact, named 'index' (by virtue of the `name='index'` argument). You can also use 'home' to refer to the second pattern.

Step 3-4: Use template inheritance to create a header and nav bar

Instead of having explicit navigation links on each page, modern web apps typically use a branding header and a

navigation bar that provides the most important page links, popup menus, and so on. To make sure the header and nav bar are the same across all pages, however, you don't want to repeat the same code in every page template. You instead want to define the common parts of all your pages in one place.

Django's templating system provides two means for reusing specific elements across multiple templates: includes and inheritance.

- *Includes* are other page templates that you insert at a specific place in the referring template using the syntax `{% include <template_path> %}`. You can also use a variable if you want to change the path dynamically in code. Includes are typically used in the body of a page to pull in the shared template at a specific location on the page.
- *Inheritance* uses the `{% extends <template_path> %}` at the beginning of a page template to specify a shared base template that the referring template then builds upon. Inheritance is commonly used to define a shared layout, nav bar, and other structures for an app's pages, such that referring templates need only add or modify specific areas of the base template called *blocks*.

In both cases, `<template_path>` is relative to the app's *templates* folder (`../` or `./` are also allowed).

A base template delineates blocks using `{% block <block_name> %}` and `{% endblock %}` tags. If a referring template then uses tags with the same block name, its block content override that of the base template.

The following steps demonstrate inheritance:

1. In the app's *templates/HelloDjangoApp* folder, create a new HTML file (using the **Add > New item** context menu or **Add > HTML Page**) called *layout.html*, and replace its contents with the markup below. You can see that this template contains a block named "content" that is all that the referring pages need to replace:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>{{ title }}</title>
    {% load staticfiles %}
    <link rel="stylesheet" type="text/css" href="{% static 'site.css' %}" />
</head>

<body>
    <div class="navbar">
        <a href="/" class="navbar-brand">Hello Django</a>
        <a href="{% url 'home' %}" class="navbar-item">Home</a>
        <a href="{% url 'about' %}" class="navbar-item">About</a>
    </div>

    <div class="body-content">
        {% block content %}{% endblock %}
        <hr/>
        <footer>
            <p>&copy; 2018</p>
        </footer>
    </div>
</body>
</html>
```

2. Add the following styles to the app's *static/site.css* file (this walkthrough isn't attempting to demonstrate responsive design here; these styles are simply to generate an interesting result):

```

.navbar {
    background-color: lightslategray;
    font-size: 1em;
    font-family: 'Trebuchet MS', 'Lucida Sans Unicode', 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;
    color: white;
    padding: 8px 5px 8px 5px;
}

.navbar a {
    text-decoration: none;
    color: inherit;
}

.navbar-brand {
    font-size: 1.2em;
    font-weight: 600;
}

.navbar-item {
    font-variant: small-caps;
    margin-left: 30px;
}

.body-content {
    padding: 5px;
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}

```

3. Modify `templates/HelloDjangoApp/index.html` to refer to the base template and override the content block. You can see that by using inheritance, this template becomes simple:

```

{% extends "HelloDjangoApp/layout.html" %}
{% block content %}
<span class="message">{{ message }}</span>{{ content }}
{% endblock %}

```

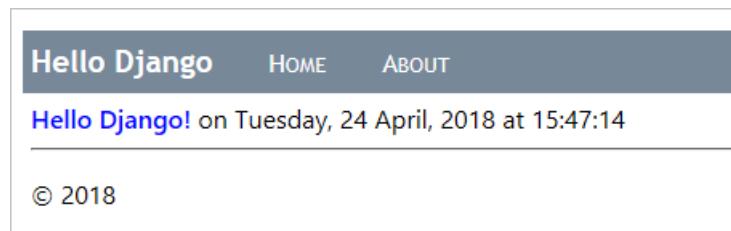
4. Modify `templates/HelloDjangoApp/about.html` to also refer to the base template and override the content block:

```

{% extends "HelloDjangoApp/layout.html" %}
{% block content %}
{{ content }}
{% endblock %}

```

5. Run the server to observe the results. Close the server when done.



6. Because you'd made substantial changes to the app, it's again a good time to [commit your changes to source control](#).

Next steps

Use the full Django Web Project template

Go deeper

- [Deploy the web app to Azure App Service](#)
- [Writing your first Django app, part 3 \(views\)](#) (docs.djangoproject.com)
- For more capabilities of Django templates, such as control flow, see [The Django template language](#) (docs.djangoproject.com)
- For complete details on using the `{% url %}` tag, see [url](#) within the [Built-in template tags and filters for Django templates reference](#) (docs.djangoproject.com)
- Tutorial source code on GitHub: [Microsoft/python-sample-vs-learning-django](#)

Step 4: Use the full Django Web Project template

4/9/2019 • 7 minutes to read • [Edit Online](#)

Previous step: [Serve static files, add pages, and use template inheritance](#)

Now that you've explored the basics of Django by building an app upon the "Blank Django Web Project" template in Visual Studio, you can easily understand the fuller app that's produced by the "Django Web Project" template.

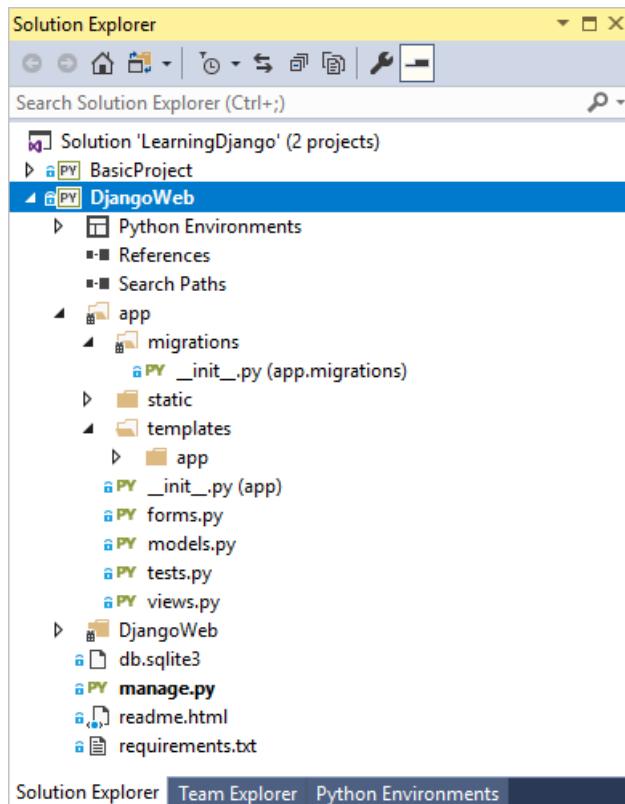
In this step you now:

- Create a fuller Django web app using the "Django Web Project" template and examine the project structure (step 4-1)
- Understand the views and page templates created by the project template, which consist of three pages that inherit from a base page template and that employs static JavaScript libraries like jQuery and Bootstrap (step 4-2)
- Understand the URL routing provided by the template (step 4-3)

The template also provides basic authentication, which is covered in Step 5.

Step 4-1: Create a project from the template

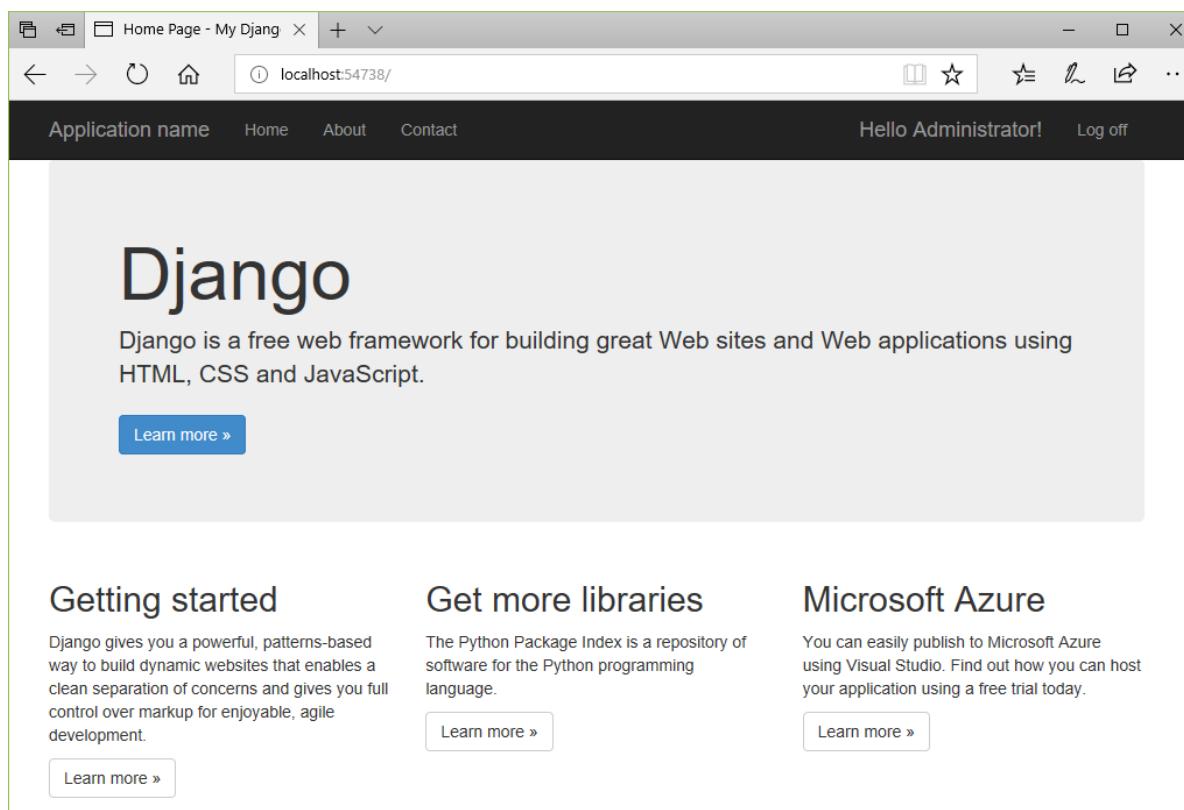
1. In Visual Studio, go to **Solution Explorer**, right-click the **LearningDjango** solution created earlier in this tutorial, and select **Add > New Project**. (Alternately, if you want to use a new solution, select **File > New > Project** instead.)
2. In the new project dialog, search for and select the **Django Web Project** template, call the project "DjangoWeb", and select **OK**.
3. Because the template again includes a *requirements.txt* file, Visual Studio asks where to install those dependencies. Choose the option, **Install into a virtual environment**, and in the **Add Virtual Environment** dialog select **Create** to accept the defaults.
4. Once Visual Studio finishes setting up the virtual environment, follow the instructions in the displayed *readme.html* to create a Django super user (that is, an administrator). Just right-click the Visual Studio project and select the **Python > Django Create Superuser** command, then follow the prompts. Make sure to record your username and password as you use it when exercising the authentication features of the app.
5. Set the **DjangoWeb** project to be the default for the Visual Studio solution by right-clicking that project in **Solution Explorer** and selecting **Set as Startup Project**. The startup project, which is shown in bold, is what's run when you start the debugger.



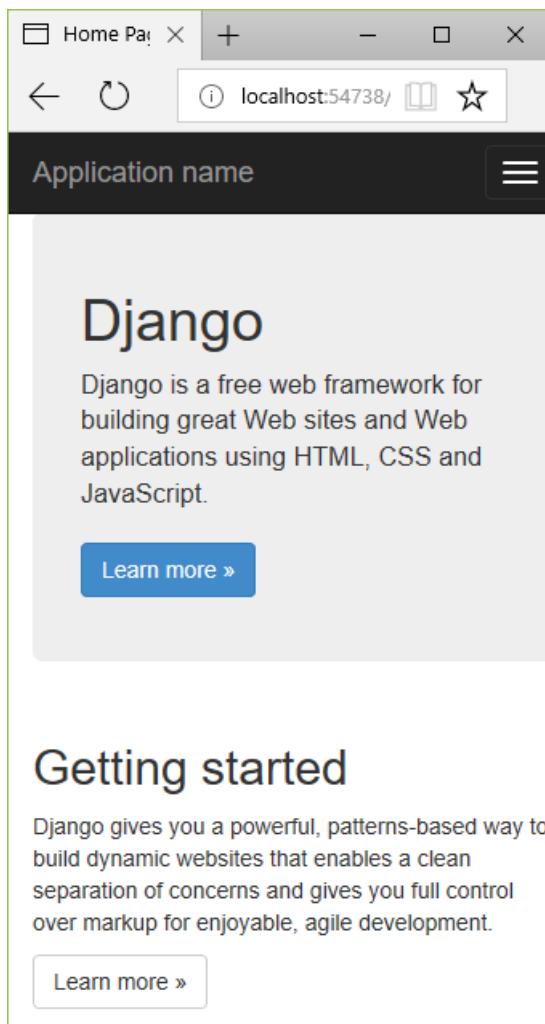
6. Select **Debug > Start Debugging (F5)** or use the **Web Server** button on the toolbar to run the server:

▶ Web Server (Microsoft Edge) ▾

7. The app created by the template has three pages, Home, About, and Contact, which you navigate between using the nav bar. Take a minute or two to examine different parts of the app. To authenticate with the app through the **Log in** command, use the superuser credentials created earlier.



8. The app created by the "Django Web Project" template uses Bootstrap for responsive layout that accommodates mobile form factors. To see this responsiveness, resize the browser to a narrow view so that the content renders vertically and the nav bar turns into a menu icon:



9. You can leave the app running for the sections that follow.

If you want to stop the app and [commit changes to source control](#), first open the **Changes** page in **Team Explorer**, right-click the folder for the virtual environment (probably **env**), and select **Ignore these local items**.

Examine what the template creates

At the broadest level, the "Django Web Project" template creates the following structure:

- Files in the project root:
 - *manage.py*, the Django administrative utility.
 - *db.sqlite3*, a default SQLite database.
 - *requirements.txt* containing a dependency on Django 1.x.
 - *readme.html*, a file that's displayed in Visual Studio after creating the project. As noted in the previous section, follow the instructions here to create a super user (administrator) account for the app.
- The *app* folder contains all the app files, including views, models, tests, forms, templates, and static files (see step 4-2). You typically rename this folder to use a more distinctive app name.
- The *DjangoWeb* (Django project) folder contains the typical Django project files: *__init__.py*, *settings.py*, *urls.py*, and *wsgi.py*. By using the project template, *settings.py* is already configured for the app and the database file, and *urls.py* is already configured with routes to all the app pages, including the login form.

Question: Is it possible to share a virtual environment between Visual Studio projects?

Answer: Yes, but do so with the awareness that different projects likely use different packages over time, and therefore a shared virtual environment must contain all the packages for all projects that use it.

Nevertheless, to use an existing virtual environment, do the following:

1. When prompted to install dependencies in Visual Studio, select **I will install them myself** option.
2. In **Solution Explorer**, right-click the **Python Environments** node and select **Add Existing Virtual Environment**.
3. Navigate to and select the folder containing the virtual environment, then select **OK**.

Step 4-2: Understand the views and page templates created by the project template

As you observe when you run the project, the app contains three views: Home, About, and Contact. The code for these views is found in the `app/views` folder. Each view function simply calls `django.shortcuts.render` with the path to a template and a simple dictionary object. For example, the About page is handled by the `about` function:

```
def about(request):
    """Renders the about page."""
    assert isinstance(request, HttpRequest)
    return render(
        request,
        'app/about.html',
        {
            'title': 'About',
            'message': 'Your application description page.',
            'year': datetime.now().year,
        }
    )
```

Templates are located in the app's `templates/app` folder (and you typically want to rename `app` to the name of your real app). The base template, `layout.html`, is the most extensive. It refers to all the necessary static files (JavaScript and CSS), defines a block named "content" that other pages override, and provides another block named "scripts". The following annotated excerpts from `layout.html` show these specific areas:

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />

    <!-- Define a viewport for Bootstrap's responsive rendering -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{ title }} - My Django Application</title>

    {% load staticfiles %}
    <link rel="stylesheet" type="text/css" href="{% static 'app/content/bootstrap.min.css' %}" />
    <link rel="stylesheet" type="text/css" href="{% static 'app/content/site.css' %}" />
    <script src="{% static 'app/scripts/modernizr-2.6.2.js' %}"></script>
</head>
<body>
    <!-- Navbar omitted -->

    <div class="container body-content">

        <!-- "content" block that pages are expected to override -->
        {% block content %}{% endblock %}
            <hr/>
            <footer>
                <p>&copy; {{ year }} - My Django Application</p>
            </footer>
    </div>

    <!-- Additional scripts; use the "scripts" block to add page-specific scripts. -->
    <script src="{% static 'app/scripts/jquery-1.10.2.js' %}"></script>
    <script src="{% static 'app/scripts/bootstrap.js' %}"></script>
    <script src="{% static 'app/scripts/respond.js' %}"></script>
    {% block scripts %}{% endblock %}

</body>
</html>

```

The individual page templates, *about.html*, *contact.html*, and *index.html*, each extend the base template *layout.html*. *about.html* is the simplest and shows the `{% extends %}` and `{% block content %}` tags:

```

{% extends "app/layout.html" %}

{% block content %}

<h2>{{ title }}.</h2>
<h3>{{ message }}</h3>

<p>Use this area to provide additional information.</p>

{% endblock %}

```

index.html and *contact.html* use the same structure and provide lengthier content in the "content" block.

In the *templates/app* folder is also a fourth page *login.html*, along with *loginpartial.html* that's brought into *layout.html* using `{% include %}`. These template files are discussed in step 5 on authentication.

Question: Can `{% block %}` and `{% endblock %}` be indented in the Django page template?

Answer: Yes, Django page templates work fine if you indent block tags, perhaps to align them within their appropriate parent elements. They're not indented in the page templates generated by the Visual Studio project template so that you can clearly see where they are placed.

Step 4-3: Understand the URL routing created by the template

The Django project's `urls.py` file as created by the "Django Web Project" template contains the following code:

```
from datetime import datetime
from django.conf.urls import url
import django.contrib.auth.views

import app.forms
import app.views

urlpatterns = [
    url(r'^$', app.views.home, name='home'),
    url(r'^contact$', app.views.contact, name='contact'),
    url(r'^about$', app.views.about, name='about'),
    url(r'^login/$',
        django.contrib.auth.views.login,
        {
            'template_name': 'app/login.html',
            'authentication_form': app.forms.BootstrapAuthenticationForm,
            'extra_context':
            {
                'title': 'Log in',
                'year': datetime.now().year,
            }
        },
        name='login'),
    url(r'^logout$',
        django.contrib.auth.views.logout,
        {
            'next_page': '/',
        },
        name='logout'),
]
```

The first three URL patterns map directly to the `home`, `contact`, and `about` views in the app's `views.py` file. The patterns `^login$/` and `^logout$`, on the other hand, use built-in Django views instead of app-defined views. The calls to the `url` method also include extra data to customize the view. Step 5 explores these calls.

Question: In the project I created, why does the "about" URL pattern uses '^about' instead of '^about\$' as shown here?

Answer: The lack of the trailing '\$' in the regular expression was a simple oversight in many versions of the project template. The URL pattern works perfectly well for a page named "about", but without the trailing '\$' the URL pattern also matches URLs like "about=django", "about09876", "aboutoflaughte" and so on. The trailing '\$' is shown here to create a URL pattern that matches *only* "about".

Next steps

[Authenticate users in Django](#)

Go deeper

- [Deploy the web app to Azure App Service](#)
- [Writing your first Django app, part 4 - forms and generic views](#) (docs.djangoproject.com)
- Tutorial source code on GitHub: [Microsoft/python-sample-vs-learning-django](https://github.com/Microsoft/python-sample-vs-learning-django)

Step 5: Authenticate users in Django

4/9/2019 • 5 minutes to read • [Edit Online](#)

Previous step: [Use the full Django Web Project template](#)

Because authentication is a common need for web apps, the "Django Web Project" template includes a basic authentication flow. (The "Polls Django Web Project" template discussed in step 6 of this tutorial also includes the same flow.) When using any of the Django project templates, Visual Studio includes all the necessary modules for authentication in the Django project's `settings.py`.

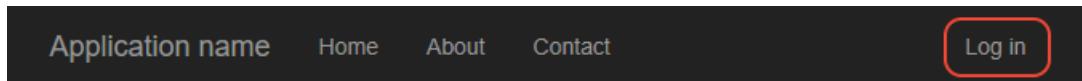
In this step you learn:

- How to use the authentication flow provided in the Visual Studio templates (step 5-1)

Step 5-1: Use the authentication flow

The following steps exercise the authentication flow and describe the parts of the project that are involved:

1. If you've not already followed the instructions in the `readme.html` file in the project root to create a super user (administrator) account, do so now.
2. Run the app from Visual Studio using **Debug > Start Debugging (F5)**. When the app appears in the browser, observe that **Log in** appears on the upper right of the nav bar.



3. Open `templates/app/layout.html` and observe that the `<div class="navbar ...>` element contains the tag `{% include app/loginpartial.html %}`. The `{% include %}` tag instructs Django's templating system to pull in the contents of the included file at this point in the containing template.
4. Open `templates/app/loginpartial.html` and observe how it uses the conditional tag `{% if user.is_authenticated %}` along with an `{% else %}` tag to render different UI elements depending on whether the user has authenticated:

```
{% if user.is_authenticated %}  
  <form id="logoutForm" action="/logout" method="post" class="navbar-right">  
    {% csrf_token %}  
    <ul class="nav navbar-nav navbar-right">  
      <li><span class="navbar-brand">Hello {{ user.username }}!</span></li>  
      <li><a href="javascript:document.getElementById('logoutForm').submit()">Log off</a></li>  
    </ul>  
  </form>  
  
  {% else %}  
  
    <ul class="nav navbar-nav navbar-right">  
      <li><a href="{% url 'login' %}">Log in</a></li>  
    </ul>  
  
  {% endif %}
```

5. Because no user is authenticated when you first start the app, this template code renders only the "Log in" link to the relative path "login". As specified in `urls.py` (as shown in the previous section), that route is mapped to the `django.contrib.auth.views.login` view. That view receives the following data:

```
{
    'template_name': 'app/login.html',
    'authentication_form': app.forms.BootstrapAuthenticationForm,
    'extra_context':
    {
        'title': 'Log in',
        'year': datetime.now().year,
    }
}
```

Here, `template_name` identifies the template for the login page, in this case `templates/app/login.html`. The `extra_context` property is added to the default context data given to the template. Finally, `authentication_form` specifies a form class to use with the login; in the template it appears as the `form` object. The default value is `AuthenticationForm` (from `django.contrib.auth.views`); the Visual Studio project template instead uses the form defined in the app's `forms.py` file:

```
from django import forms
from django.contrib.auth.forms import AuthenticationForm
from django.utils.translation import gettext_lazy as _

class BootstrapAuthenticationForm(AuthenticationForm):
    """Authentication form which uses bootstrap CSS."""
    username = forms.CharField(max_length=254,
                               widget=forms.TextInput({
                                   'class': 'form-control',
                                   'placeholder': 'User name'}))
    password = forms.CharField(label=_('Password'),
                               widget=forms.PasswordInput({
                                   'class': 'form-control',
                                   'placeholder':'Password'}))
```

As you can see, this form class derives from `AuthenticationForm` and specifically overrides the `username` and `password` fields to add placeholder text. The Visual Studio template includes this explicit code on the assumption that you likely want to customize the form, such as adding password strength validation.

- When you navigate to the login page, then, the app renders the `login.html` template. The variables `{{ form.username }}` and `{{ form.password }}` render the `CharField` forms from `BootstrapAuthenticationForm`. There's also a built-in section to show validation errors, and a ready-made element for social logins if you choose to add those services.

```

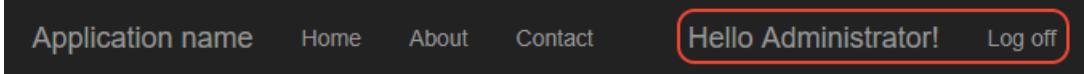
{% extends "app/layout.html" %}

{% block content %}

<h2>{{ title }}</h2>
<div class="row">
    <div class="col-md-8">
        <section id="loginForm">
            <form action"." method="post" class="form-horizontal">
                {% csrf_token %}
                <h4>Use a local account to log in.</h4>
                <hr />
                <div class="form-group">
                    <label for="id_username" class="col-md-2 control-label">User name</label>
                    <div class="col-md-10">
                        {{ form.username }}
                    </div>
                </div>
                <div class="form-group">
                    <label for="id_password" class="col-md-2 control-label">Password</label>
                    <div class="col-md-10">
                        {{ form.password }}
                    </div>
                </div>
                <div class="form-group">
                    <div class="col-md-offset-2 col-md-10">
                        <input type="hidden" name="next" value="/" />
                        <input type="submit" value="Log in" class="btn btn-default" />
                    </div>
                </div>
                {% if form.errors %}
                    <p class="validation-summary-errors">Please enter a correct user name and password.</p>
                {% endif %}
            </form>
        </section>
    </div>
    <div class="col-md-4">
        <section id="socialLoginForm"></section>
    </div>
</div>

{% endblock %}

```

7. When you submit the form, Django attempts to authenticate your credentials (such as the super user's credentials). If authentication fails, you remain on the current page but `form.errors` set to true. If authentication is successful, Django navigates to the relative URL in the "next" field, `<input type="hidden" name="next" value="/" />`, which in this case is the home page (`/`).
 8. Now, when the home page is rendered again, the `user.is_authenticated` property is true when the `loginpartial.html` template is rendered. As a result, you see a **Hello (username)** message and **Log off**. You can use `user.is_authenticated` in other parts of the app to check authentication.
- 
9. To check whether the authenticated user is authorized to access specific resources, you need to retrieve user-specific permissions from your database. For more information, see [Using the Django authentication system](#) (Django docs).
 10. The super user or administrator, in particular, is authorized to access the built-in Django administrator interfaces using the relative URLs `/admin/` and `/admin/doc/`. To enable these interfaces, do the following:
 - a. Install the `docutils` Python package into your environment. A great way to do this is to add "docutils"

to your `requirements.txt` file, then in **Solution Explorer**, expand the project, expand the **Python Environments** node, then right-click the environment you're using and select **Install from requirements.txt**.

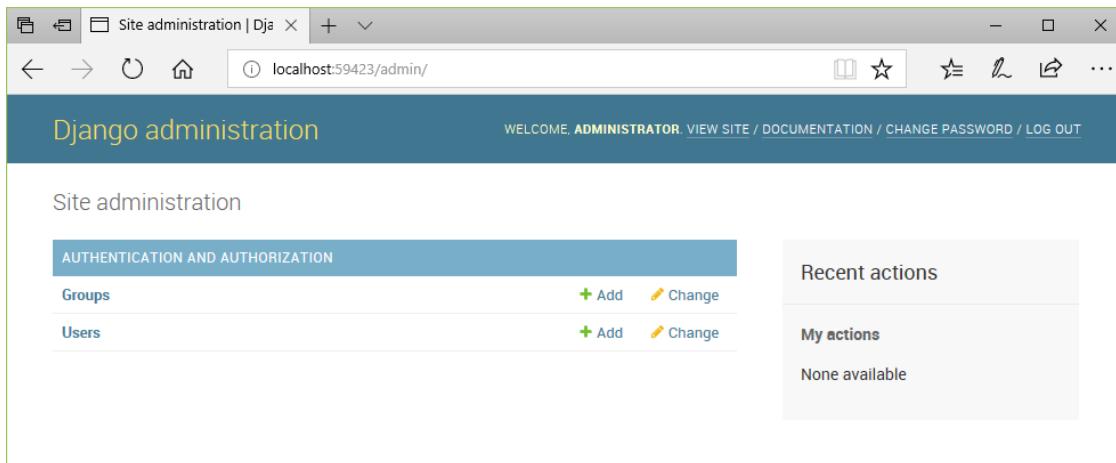
- b. Open the Django project's `urls.py` and remove the default comments from the following entries:

```
from django.conf.urls import include
from django.contrib import admin
admin.autodiscover()

# ...
urlpatterns = [
    # ...
    url(r'^admin/doc/', include('django.contrib.admindocs.urls')),
    url(r'^admin/', include(admin.site.urls)),
]
```

- c. In the Django project's `settings.py` file, navigate to the `INSTALLED_APPS` collection and add `'django.contrib.admindocs'`.

- d. When you restart the app, you can navigate to `/admin/` and `/admin/doc/` and perform tasks like creating additional user accounts.



11. The final part to the authentication flow is logging off. As you can see in `loginpartial.html`, the **Log off** link simply does a POST to the relative URL `/login`, which is handled by the built-in view `django.contrib.auth.views.logout`. This view doesn't display any UI and just navigates to the home page (as shown in `urls.py` for the `"^logout$"` pattern). If you want to display a logoff page, first change the URL pattern as follows to add a "template_name" property and remove the "next_page" property:

```
url(r'^logout$',
    django.contrib.auth.views.logout,
{
    'template_name': 'app/loggedoff.html',
    # 'next_page': '/',
},
name='logout')
```

Then create `templates/app/loggedoff.html` with the following (minimal) contents:

```
{% extends "app/layout.html" %}
{% block content %}
<h3>You have been logged off</h3>
{% endblock %}
```

The result appears as follows:

A screenshot of a Django application's login page. At the top, there is a dark navigation bar with the text "Application name" and links for "Home", "About", and "Contact". On the right side of the navigation bar is a "Log in" button. Below the navigation bar, the main content area has a light gray background. In the center, the text "You have been logged off" is displayed in a large, dark font. At the bottom of the content area, there is a thin horizontal line followed by the copyright notice "© - My Django Application".

12. When you're all done, stop the server and once again commit your changes to source control.

Question: What is the purpose of the `{% csrf_token %}` tag that appears in the <form> elements?

Answer: The `{% csrf_token %}` tag includes Django's built-in [cross-site request forgery \(csrf\) protection](#) (Django docs). You typically add this tag to any element that involves POST, PUT, or DELETE request methods, such as a form. The template rendering function (`render`) then inserts the necessary protection.

Next steps

[Use the Polls Django Web Project template](#)

Go deeper

- [User authentication in Django](#) (docs.djangoproject.com)
- Tutorial source code on GitHub: [Microsoft/python-sample-vs-learning-django](#)

Step 6: Use the Polls Django Web Project template

4/9/2019 • 18 minutes to read • [Edit Online](#)

Previous step: [Authenticate users in Django](#)

Having understood Visual Studio's "Django Web Project" template, you can now look at the third Django template, "Polls Django Web Project", which builds upon the same code base and demonstrates working with a database.

In this step you learn how to:

- Create a project from the template and initialize the database (step 6-1)
- Understand data models (step 6-2)
- Apply migrations (step 6-3)
- Understand the views and page templates created by the project template (step 6-4)
- Create a custom administration interface (step 6-5)

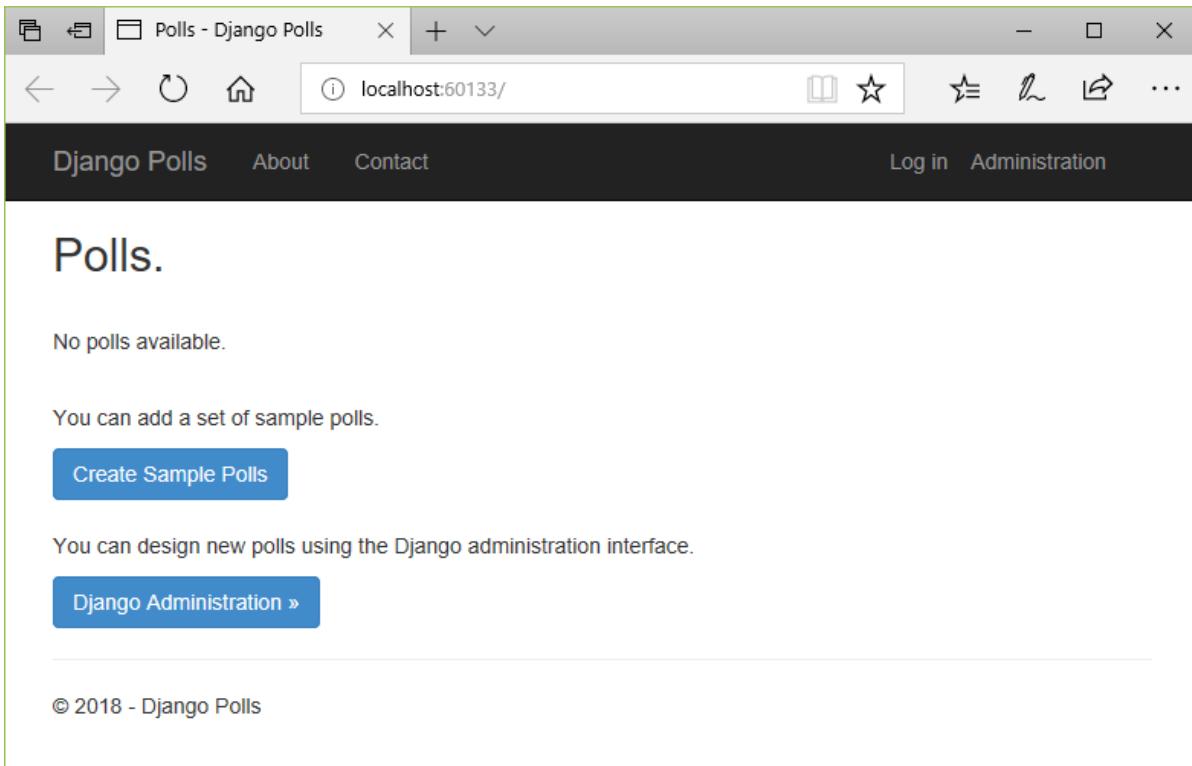
A project created using this template is similar to what you get by following the [Writing your first Django app](#) tutorial in the Django docs. The web app consists of a public site that lets people view polls and vote in them, along with a custom administrative interface through which you can manage polls. It uses the same authentication system as the "Django Web Project" template and makes more use of the database by implementing Django models as explored in the following sections.

Step 6-1: Create the project and initialize the database

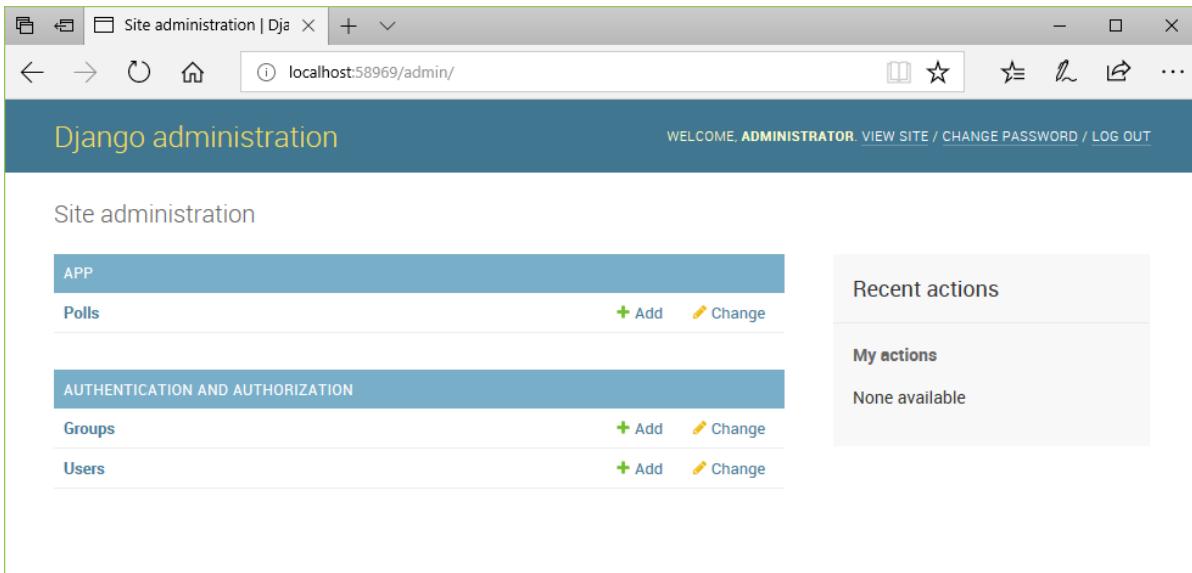
1. In Visual Studio, go to **Solution Explorer**, right-click the **LearningDjango** solution created earlier in this tutorial, and select **Add > New Project**. (Alternately, if you want to use a new solution, select **File > New > Project** instead.)
2. In the new project dialog, search for and select the **Polls Django Web Project** template, call the project "DjangoPolls", and select **OK**.
3. Like the other project templates in Visual Studio, the "Polls Django Web Project" template includes a *requirements.txt* file, Visual Studio prompts asks where to install those dependencies. Choose the option, **Install into a virtual environment**, and in the **Add Virtual Environment** dialog select **Create** to accept the defaults.
4. Once Python finishes setting up the virtual environment, follow the instructions in the displayed *readme.html* to initialize the database and create a Django super user (that is, an administrator). The steps are to first right-click the **DjangoPolls** project in **Solution Explorer**, select the **Python > Django Migrate** command, then right-click the project again, select the **Python > Django Create Superuser** command, and follow the prompts. (If you try to create a super user first, you'll see an error because the database has not been initialized.)
5. Set the **DjangoPolls** project to be the default for the Visual Studio solution by right-clicking that project in **Solution Explorer** and selecting **Set as Startup Project**. The startup project, which is shown in bold, is what's run when you start the debugger.
6. Select **Debug > Start Debugging (F5)** or use the **Web Server** button on the toolbar to run the server:

 7. The app created by the template has three pages, Home, About, and Contact, which you navigate between using the top nav bar. Take a minute or two to examine different parts of the app (the About and Contact

pages are very similar to the "Django Web Project" and aren't discussed further).



8. Also select the **Administration** link in the nav bar, which displays a login screen to demonstrate that the administrative interface is authorized only to authenticated administrators. Use the super user credentials and you're routed to the "/admin" page, which is enabled by default when using this project template.



9. You can leave the app running for the sections that follow.

If you want to stop the app and [commit changes to source control](#), first open the **Changes** page in **Team Explorer**, right-click the folder for the virtual environment (probably **env**), and select **Ignore these local items**.

Examine the project contents

As noted before, much of what's in a project created from the "Polls Django Web Project" template should be familiar if you've explored the other project templates in Visual Studio. The additional steps in this article summarize the more significant changes and additions, namely data models and additional views.

Question: What does the Django Migrate command do?

Answer: the **Django Migrate** command specifically runs the `manage.py migrate` command, which runs any scripts in the `app/migrations` folder that haven't been run previously. In this case, the command runs the `0001_initial.py` script in that folder to set up the necessary schema in the database.

The migration script itself is created by the `manage.py makemigrations` command, which scans the app's `models.py` file, compares it to the current state of the database, and then generates the necessary scripts to migrate the database schema to match the current models. This feature of Django is very powerful as you update and modify your models over time. By generating and running migrations, you keep the models and the database in sync with little difficulty.

You work with a migration in step 6-3 later in this article.

Step 6-2: Understand data models

The models for the app, named Poll and Choice, are defined in `app/models.py`. Each is a Python class that derives from `django.db.models.Model` and uses methods of the `models` class like `CharField` and `IntegerField` to define fields in the model, which map to database columns.

```
from django.db import models
from django.db.models import Sum

class Poll(models.Model):
    """A poll object for use in the application views and repository."""
    text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')

    def total_votes(self):
        """Calculates the total number of votes for this poll."""
        return self.choice_set.aggregate(Sum('votes'))['votes__sum']

    def __unicode__(self):
        """Returns a string representation of a poll."""
        return self.text

class Choice(models.Model):
    """A poll choice object for use in the application views and repository."""
    poll = models.ForeignKey(Poll)
    text = models.CharField(max_length=200)
    votes = models.IntegerField(default=0)

    def votes_percentage(self):
        """Calculates the percentage of votes for this choice."""
        total = self.poll.total_votes()
        return self.votes / float(total) * 100 if total > 0 else 0

    def __unicode__(self):
        """Returns a string representation of a choice."""
        return self.text
```

As you can see, a Poll maintains a description in its `text` field and a publication date in `pub_date`. These fields are the only ones that exist for the Poll in the database; the `total_votes` field is calculated at runtime.

A Choice is related to a Poll through the `poll` field, contains a description in `text`, and maintains a count for that choice in `votes`. The `votes_percentage` field is calculated at runtime and is not found in the database.

The full list of field types is `CharField` (limited text) `TextField` (unlimited text), `EmailField`, `URLField`, `DateTimeField`, `IntegerField`, `DecimalField`, `BooleanField`, `ForeignKey`, and `ManyToMany`. Each field takes some attributes, like `max_length`. The `blank=True` attribute means the field is optional; `null=True` means that a value is optional. There is also a `choices` attribute that limits values to values in an array of data value/display value tuples. (See the [Model field reference](#) in the Django documentation.)

You can confirm exactly what's stored in the database by examining the `db.sqlite3` file in the project using a tool like the [SQLite browser](#). In the database, you see that a foreign key field like `poll` in the `Choice` model is stored as `poll_id`; Django handles the mapping automatically.

In general, working with your database in Django means working exclusively through your models so that Django can manage the underlying database on your behalf.

Seed the database from `samples.json`

Initially, the database contains no polls. You can use the administrative interface at the "/admin" URL to add polls manually, and you can also visit the "/seed" page on the running site to add seed the database with polls defined in the app's `samples.json` file.

The Django project's `urls.py` has an added URL pattern, `url(r'^seed$', app.views.seed, name='seed')`. The `seed` view in `app/views.py` loads the `samples.json` file and creates the necessary model objects. Django then automatically creates the matching records in the underlying database.

Note the use of the `@login_required` decorator to indicate the authorization level for the view.

```
@login_required
def seed(request):
    """Seeds the database with sample polls."""
    samples_path = path.join(path.dirname(__file__), 'samples.json')
    with open(samples_path, 'r') as samples_file:
        samples_polls = json.load(samples_file)

    for sample_poll in samples_polls:
        poll = Poll()
        poll.text = sample_poll['text']
        poll.pub_date = timezone.now()
        poll.save()

        for sample_choice in sample_poll['choices']:
            choice = Choice()
            choice.poll = poll
            choice.text = sample_choice
            choice.votes = 0
            choice.save()

    return HttpResponseRedirect(reverse('app:home'))
```

To see the effect, run the app first to see that no polls yet exist. Then visit the "/seed" URL, and when the app returns to the home page you should see that polls have become available. Again, feel free to examine the raw `db.sqlite3` file with a tool like the [SQLite browser](#).

The screenshot shows the Django Polls application's homepage. At the top, there is a dark navigation bar with the text "Django Polls", "About", "Contact", "Hello Administrator!", "Log off", and "Administration". Below the navigation bar, the main content area has a title "Polls." followed by three poll options:

- "Have you tried R Tools for Visual Studio?"
- "How do you commute to work/school?"
- "What is your favorite season?"

At the bottom of the page, there is a footer with the text "© 2018 - Django Polls".

Question: Is it possible to initialize the database using the Django administrative utility?

Answer: Yes, you can use the [django-admin loaddata command](#) to accomplish the same task as the seeding page in the app. When working on a full web app, you might use a combination of the two methods: initialize a database from the command line, then convert the seed page here to an API to which you can send any other arbitrary JSON rather than relying on a hard-coded file.

Step 6-3: Use migrations

When you ran the `manage.py makemigrations` command (using the context menu in the Visual Studio) after creating the project, Django created the file `app/migrations/0001_initial.py`. This file contains a script that creates the initial database tables.

Because you'll inevitably make changes to your models over time, Django makes it easy to keep the underlying database schema up to date with those models. The general workflow is as follows:

1. Make changes to the models in your `models.py` file.
2. In Visual Studio, right-click the project in **Solution Explorer** and select the **Python > Django Make Migrations** command. As described earlier, this command generates scripts in `app/migrations` to migrate the database from its current state to the new state.
3. To apply the scripts to the actual database, right-click the project again and select **Python > Django Migrate**.

Django tracks which migrations have been applied to any given database, such that when you run the `migrate` command, Django applies whichever migrations are needed. If you create a new, empty database, for example, running the `migrate` command brings it up to date with your current models by applying every migration script. Similarly, if you make multiple model changes and generate migrations on a development computer, you can then apply the cumulative migrations to your production database by running the `migrate` command on your production server. Django again applies only those migration scripts that have been generated since the last migration of the production database.

To see the effect of changing a model, try the following steps:

1. Add an optional `author` field to the `Poll` model in `app/models.py` by adding the following line after the `pub_date` field to add an optional `author` field:

```
author = models.CharField(max_length=100, blank=True)
```
2. Save the file, then right-click the **DjangoPolls** project in **Solution Explorer** and select the **Python > Django Make Migrations** command.
3. Select the **Project > Show All Files** command to see the newly generated script in the **migrations** folder, whose name starts with `002_auto_`. Right-click that file and select **Include In Project**. You can then select **Project > Show All Files** again to restore the original view. (See the second question below for details on this step.)
4. If desired, open that file to examine how Django scripts the change from the previous model state to the new state.
5. Right-click the Visual Studio project again and select **Python > Django Migrate** to apply the changes to the database.
6. If desired, open the database in an appropriate viewer to confirm the change.

Overall, Django's migration feature means that you need never manage your database schema manually. Just make changes to your models, generate the migration scripts, and apply them with the `migrate` command.

Question: What happens if I forget to run the `migrate` command after making changes to models?

Answer: If the models don't match what's in the database, Django fails at runtime with appropriate exceptions. For example, if you forget to migrate the model change shown in the previous section, you see an error **no such column: app_poll.author**:

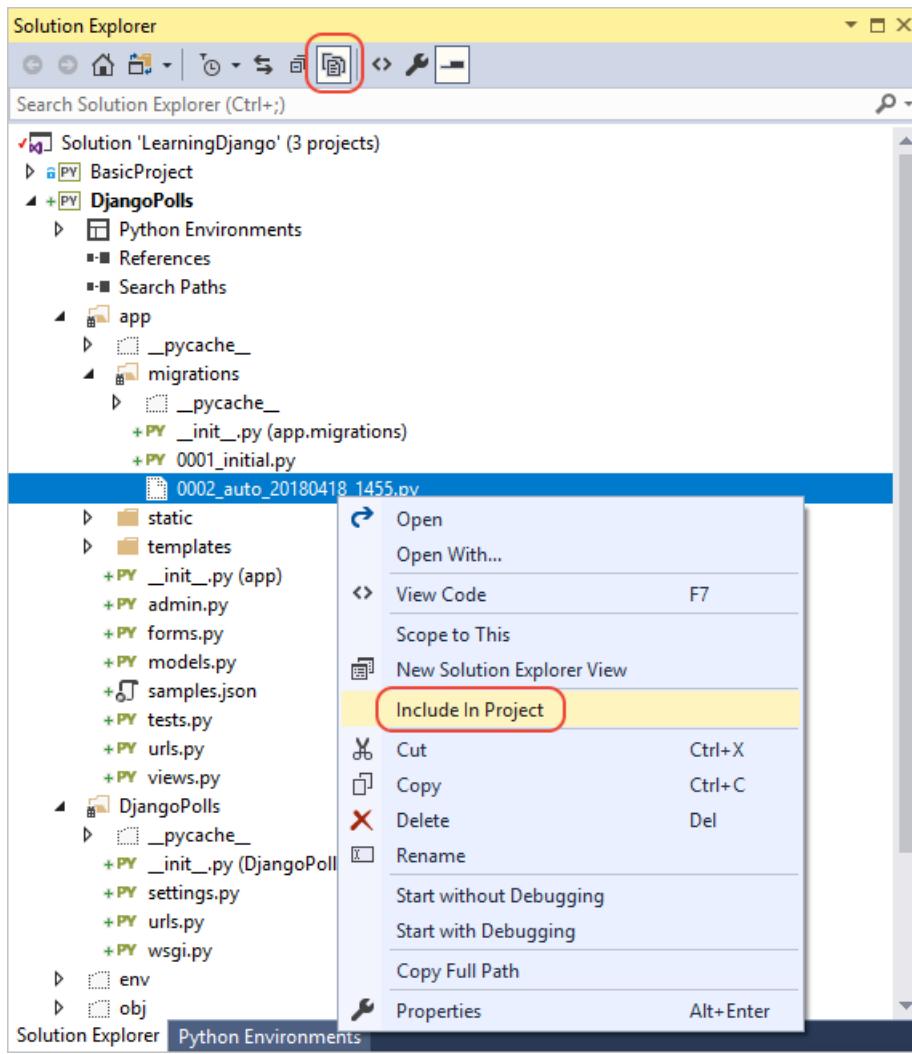
OperationalError at /

no such column: app_poll.author

```
Request Method: GET
Request URL: http://localhost:56408/
Django Version: 1.11.12
Exception Type: OperationalError
Exception Value: no such column: app_poll.author
    Exception: D:\Examples\Python\LearningDjango\ DjangoPolls\env\lib\site-
    Location: packages\django\db\backends\sqlite3\base.py in execute, line 328
Python Executable: D:\Examples\Python\LearningDjango\ DjangoPolls\env\Scripts\python.exe
    Python Version: 3.6.1
    Python Path: ['D:\\\\Examples\\\\Python\\\\LearningDjango\\\\DjangoPolls',
    'D:\\\\Examples\\\\Python\\\\LearningDjango\\\\DjangoPolls',
    'D:\\\\Examples\\\\Python\\\\LearningDjango\\\\DjangoPolls\\\\env\\\\Scripts\\\\python36.zip',
    'C:\\\\Anaconda3\\\\DLLs',
    'C:\\\\Anaconda3\\\\lib',
    'C:\\\\Anaconda3',
```

Question: Why doesn't Solution Explorer show newly generated scripts after running Django Make Migrations?

Answer: Although newly generated scripts exist in the *app/migrations* folder and are applied when running the **Django Migrate** command, they don't appear automatically in **Solution Explorer** because they've not been added to the Visual Studio project. To make them visible, first select the **Project > Show All Files** menu command or the toolbar button outlined in the image below. This command causes **Solution Explorer** to show all files in the project folder, using a dotted outline icon for items that haven't been added to the project itself. Right-click the files you want to add and select **Include In Project**, which also includes them in source control with your next commit.



Question: Can I see what migrations would be applied before running the migrate command?

Answer: Yes, use the [django-admin showmigrations](#) command.

Step 6-4: Understand the views and page templates created by the project template

Most of the views generated by the "Polls Django Web Project" template, such as the views for the About and Contact pages, are quite similar to views created by the "Django Web Project" template you worked with earlier in this tutorial. What's different in the Polls app is that its home page makes use of the models, as do several added pages for voting and viewing poll results.

To begin with, the first line in the Django project's `urlpatterns` array in `urls.py` file is more than just a simple routing to an app view. Instead, it pulls in the app's own `urls.py` file:

```
from django.conf.urls import url, include
import app.views

urlpatterns = [
    url(r'^$', include('app.urls', namespace="app")),
    # ...
]
```

The `app/urls.py` file then contains some more interesting routing code (explanatory comments added):

```

urlpatterns = [
    # Home page routing
    url(r'^$', 
        app.views.PollListView.as_view(
            queryset=Poll.objects.order_by('-pub_date')[:5],
            context_object_name='latest_poll_list',
            template_name='app/index.html'),
        name='home'), 

    # Routing for a poll page, which use URLs in the form <poll_id>/,
    # where the id number is captured as a group named "pk".
    url(r'^(?P<pk>\d+)/$', 
        app.views.PollDetailView.as_view(
            template_name='app/details.html'),
        name='detail'), 

    # Routing for <poll_id>/results pages, again using a capture group
    # named pk.
    url(r'^(?P<pk>\d+)/results/$', 
        app.views.PollResultsView.as_view(
            template_name='app/results.html'),
        name='results'), 

    # Routing for <poll_id>/vote pages, with the capture group named
    # poll_id this time, which becomes an argument passed to the view.
    url(r'^(?P<poll_id>\d+)/vote/$', app.views.vote, name='vote'),
]

]

```

If you're not familiar with the more complex regular expressions used here, you can paste the expression into [regex101.com](#) for an explanation in plain language. (You'll need to escape the forward slashes `/` by adding a back slash, `\` before them; escaping isn't necessary in Python because of the `r` prefix on the string, meaning "raw").

In Django, the syntax `?P<name>pattern` creates a group named `name`, which gets passed as arguments to views in the order they appear. In the code shown earlier, `PollsDetailView` and `PollsResultsView` receive an argument named `pk` and `app.views.vote` receives an argument named `poll_id`.

You can also see that most of the views are not just direct references to a view function in `app/views.py`. Instead, most refer to a class in that same file that derives from `django.views.generic.ListView` or `django.views.generic.DetailView`. The base classes provide the `as_view` methods, which take a `template_name` argument to identify the template. The `ListView` base class, as used for the home page, also expects a `queryset` property containing the data and a `context_object_name` property with the variable name by which you want to refer to the data in the template, in this case `latest_poll_list`.

Now you can examine the `PollListView` for the home page, which is defined as follows in `app/views.py`:

```

class PollListView(ListView):
    """Renders the home page, with a list of all polls."""
    model = Poll

    def get_context_data(self, **kwargs):
        context = super(PollListView, self).get_context_data(**kwargs)
        context['title'] = 'Polls'
        context['year'] = datetime.now().year
        return context

```

All that's done here is to identify the model that the view works with (`Poll`), and overrides the `get_context_data` method to add `title` and `year` values to the context.

The core of the template (`templates/app/index.html`) is as follows:

```

{% if latest_poll_list %}


|                                                            |
|------------------------------------------------------------|
| <a href="{% url 'app:detail' poll.id %}">{{poll.text}}</a> |
|------------------------------------------------------------|


{% else %}

{% endif %}

```

Put simply, the template receives the list of Poll objects in `latest_poll_list`, and then iterates through that list to create a table row that contains a link to each poll using the poll's `text` value. In the `{% url %}` tag, "app:detail" refers to the url pattern in `app/urls.py` named "detail", using `poll.id` as an argument. The effect of this is that Django creates a URL using the appropriate pattern and uses that for the link. This bit of future-proofing means that you can change that URL pattern at any time and the generated links automatically update to match.

The `PollDetailView` and `PollResultsView` classes in `app/views.py` (not shown here) look almost identical to `PollListView` except that they derive from `DetailView` instead. Their respective templates, `app/templates/details.html` and `app/templates/results.html` then place the appropriate fields from the models within various HTML controls. One unique piece in `details.html` is that the choices for a poll are contained within an HTML form that when submitted does a POST to the /vote URL. As seen earlier, that URL pattern is routed to `app.views.vote`, which is implemented as follows (note the `poll_id` argument, which is again a named group in the regular expression used in the routing for this view):

```

def vote(request, poll_id):
    """Handles voting. Validates input and updates the repository."""
    poll = get_object_or_404(Poll, pk=poll_id)
    try:
        selected_choice = poll.choice_set.get(pk=request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        return render(request, 'app/details.html', {
            'title': 'Poll',
            'year': datetime.now().year,
            'poll': poll,
            'error_message': "Please make a selection.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
        return HttpResponseRedirect(reverse('app:results', args=(poll.id,)))

```

Here, the view doesn't have its own corresponding template like the other pages. Instead, it validates the selected poll, showing a 404 if the poll doesn't exist (just in case someone enters a URL like "vote/1a2b3c"). It then makes sure the voted choice is valid for the poll. If not, the `except` block just renders the details page again with an error message. If the choice is valid, then the view tallies the vote and redirects to the results page.

Step 6-5: Create a custom administration interface

The last pieces of the "Polls Django Web Project" template are custom extensions to the default Django administrative interface, as shown earlier in this article under step 6-1. The default interface provides for user and group management, but nothing more. The Polls project template adds features that allow you to manage polls as well.

First of all, the URL patterns in the Django project's `urls.py` has `url(r'^admin/', include(admin.site.urls)),` included by default; the "admin/doc" pattern is also included but commented out.

The app then contains the file `admin.py`, which Django automatically runs when you visit the administrative interface thanks to the inclusion of `django.contrib.admin` in the `INSTALLED_APPS` array of `settings.py`. The code in that file, as provided by the project template, is as follows:

```
from django.contrib import admin
from app.models import Choice, Poll

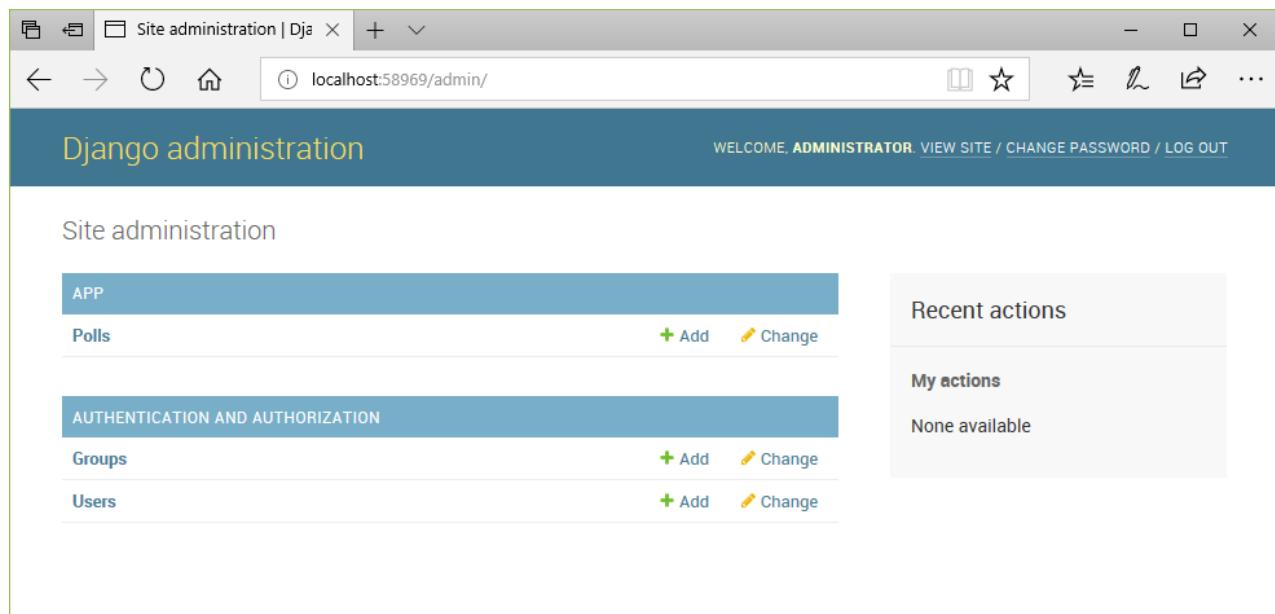
class ChoiceInline(admin.TabularInline):
    """Choice objects can be edited inline in the Poll editor."""
    model = Choice
    extra = 3

class PollAdmin(admin.ModelAdmin):
    """Definition of the Poll editor."""
    fieldsets = [
        (None, {'fields': ['text']}),
        ('Date information', {'fields': ['pub_date']}),
    ]
    inlines = [ChoiceInline]
    list_display = ('text', 'pub_date')
    list_filter = ['pub_date']
    search_fields = ['text']
    date_hierarchy = 'pub_date'

admin.site.register(Poll, PollAdmin)
```

As you can see, the `PollAdmin` class derives from `django.contrib.admin.ModelAdmin` and customizes a number of its fields using names from the `Poll` model, which it manages. These fields are described on [ModelAdmin options](#) in the Django documentation.

The call to `admin.site.register` then connects that class to the model (`Poll`) and includes it on the admin interface. The overall result is shown below:



Next steps

NOTE

If you've been committing your Visual Studio solution to source control throughout the course of this tutorial, now is a good time to do another commit. Your solution should match the tutorial source code on GitHub: [Microsoft/python-sample-vs-learning-django](https://github.com/Microsoft/python-sample-vs-learning-django).

You've now explored the entirety of the "Blank Django Web Project", "Django Web Project", and "Polls Django Web Project" templates in Visual Studio. You've learned all the basics of Django such as using views and templates, and have explored routing, authentication, and using database models. You should now be able to create a web app of your own with any views and models that you need.

Running a web app on your development computer is just one step in making the app available to your customers. Next steps may include the following tasks:

- Deploy the web app to a production server, such as Azure App Service. See [Publish to Azure App Service](#).
- Customize the 404 page by creating a template named *templates/404.html*. When present, Django uses this template instead of its default one. For more information, see [Error views](#) in the Django documentation.
- Write unit tests in *tests.py*; the Visual Studio project templates provide starting points for these, and more information can be found on [Writing your first Django app, part 5 - testing](#) and [Testing in Django](#) in the Django documentation.
- Change the app from SQLite to a production-level data store such as PostgreSQL, MySQL, and SQL Server (all of which can be hosted on Azure). As described on [When to use SQLite](#) (sqlite.org), SQLite works fine for low to medium traffic sites with fewer than 100K hits/day, but is not recommended for higher volumes. It's also limited to a single computer, so it cannot be used in any multi-server scenario such as load-balancing and geo-replication. For information on Django's support for other databases, see [Database setup](#). You can also use the [Azure SDK for Python](#) to work with Azure storage services like tables and blobs.
- Set up a continuous integration/continuous deployment pipeline on a service like Azure DevOps. In addition to working with source control (via Azure Repos or GitHub, or elsewhere), you can configure an Azure DevOps Project to automatically run your unit tests as a pre-requisite for release, and also configure the pipeline to deploy to a staging server for additional tests before deploying to production. Azure DevOps, furthermore, integrates with monitoring solutions like App Insights and closes the whole cycle with agile planning tools. For more information, see [Create a CI/CD pipeline for Python with the Azure DevOps project](#) and also the general [Azure DevOps documentation](#).

Tutorial: Get started with the Flask web framework in Visual Studio

4/9/2019 • 14 minutes to read • [Edit Online](#)

[Flask](#) is a lightweight Python framework for web applications that provides the basics for URL routing and page rendering.

Flask is called a "micro" framework because it doesn't directly provide features like form validation, database abstraction, authentication, and so on. Such features are instead provided by special Python packages called *Flask extensions*. The extensions integrate seamlessly with Flask so that they appear as if they are part of Flask itself. For example, Flask itself doesn't provide a page template engine. Templating is provided by extensions such as Jinja and Jade, as demonstrated in this tutorial.

In this tutorial, you learn how to:

- Create a basic Flask project in a Git repository using the "Blank Flask Web Project" template (step 1)
- Create a Flask app with one page and render that page using a template (step 2)
- Serve static files, add pages, and use template inheritance (step 3)
- Use the Flask Web Project template to create an app with multiple pages and responsive design (step 4)
- Use the Polls Flask Web Project template to create an polling app that uses a variety of storage options (Azure storage, MongoDB, or memory).

Over the course of these steps you create a single Visual Studio solution that contains three separate projects. You create the project using different Flask project templates that are included with Visual Studio. By keeping the projects in the same solution, you can easily switch back and forth between different files for comparison.

NOTE

This tutorial differs from the [Flask Quickstart](#) in that you learn more about Flask as well as how to use the different Flask project templates that provide a more extensive starting point for your own projects. For example, the project templates automatically install the Flask package when creating a project, rather than needing you to install the package manually as shown in the Quickstart.

Prerequisites

- Visual Studio 2017 or later on Windows with the following options:
 - The **Python development** workload (**Workload** tab in the installer). For instructions, see [Install Python support in Visual Studio](#).
 - **Git for Windows** and **GitHub Extension for Visual Studio** on the **Individual components** tab under **Code tools**.

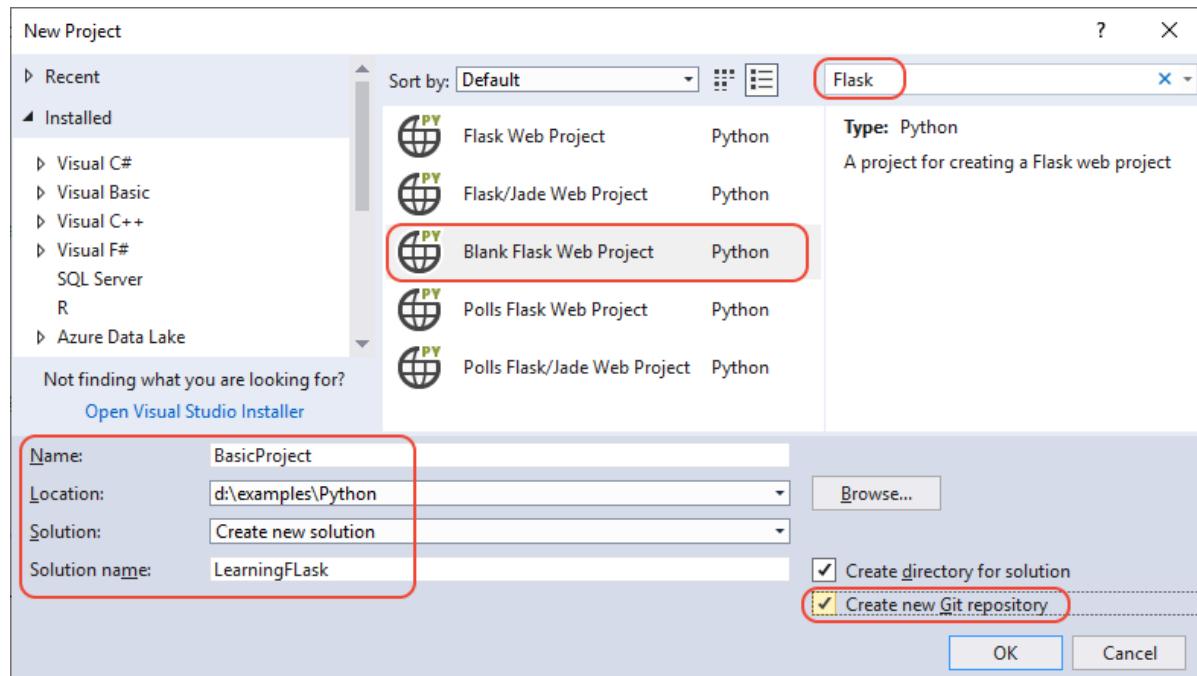
Flask project templates are included with all earlier versions of Python Tools for Visual Studio, though details may differ from what's discussed in this tutorial.

Python development is not presently supported in Visual Studio for Mac. On Mac and Linux, use the [Python extension in Visual Studio Code](#).

Step 1-1: Create a Visual Studio project and solution

1. In Visual Studio, select **File > New > Project**, search for "Flask", and select the **Blank Flask Web Project**

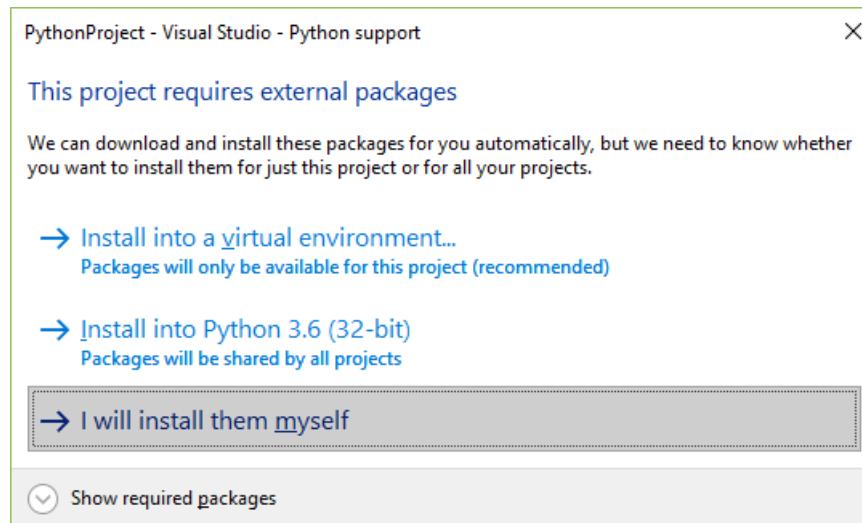
template. (The template is also found under **Python > Web** in the left-hand list.)



2. In the fields at the bottom of the dialog, enter the following information (as shown in the previous graphic), then select **OK**:

- **Name:** set the name of the Visual Studio project to **BasicProject**. This name is also used for the Flask project.
- **Location:** specify a location in which to create the Visual Studio solution and project.
- **Solution name:** set to **LearningFlask**, which is appropriate for the solution as a container for multiple projects in this tutorial.
- **Create directory for solution:** Leave set (the default).
- **Create new Git repository:** Select this option (which is clear by default) so that Visual Studio creates a local Git repository when it creates the solution. If you don't see this option, run the Visual Studio installer and add the **Git for Windows** and **GitHub Extension for Visual Studio** on the **Individual components** tab under **Code tools**.

3. After a moment, Visual Studio prompts you with a dialog saying **This project requires external packages** (shown below). This dialog appears because the template includes a *requirements.txt* file referencing the latest Flask 1.x package. (Select **Show required packages** to see the exact dependencies.)

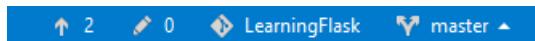


4. Select the option **I will install them myself**. You create the virtual environment shortly to make sure it's excluded from source control. (The environment can always be created from *requirements.txt*.)

Step 1-2: Examine the Git controls and publish to a remote repository

Because you selected the **Create new Git repository** in the **New Project** dialog, the project is already committed to local source control as soon as the creation process is complete. In this step, you familiarize yourself with Visual Studio's Git controls and the **Team Explorer** window in which you work with source control.

1. Examine the Git controls on the bottom corner of the Visual Studio main window. From left to right, these controls show unpushed commits, uncommitted changes, the name of the repository, and the current branch:

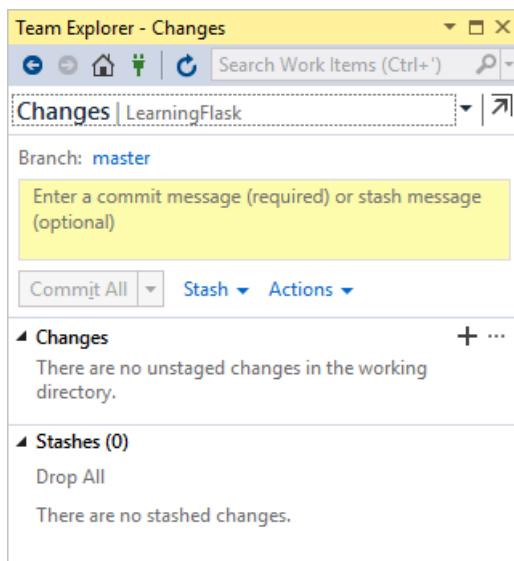


NOTE

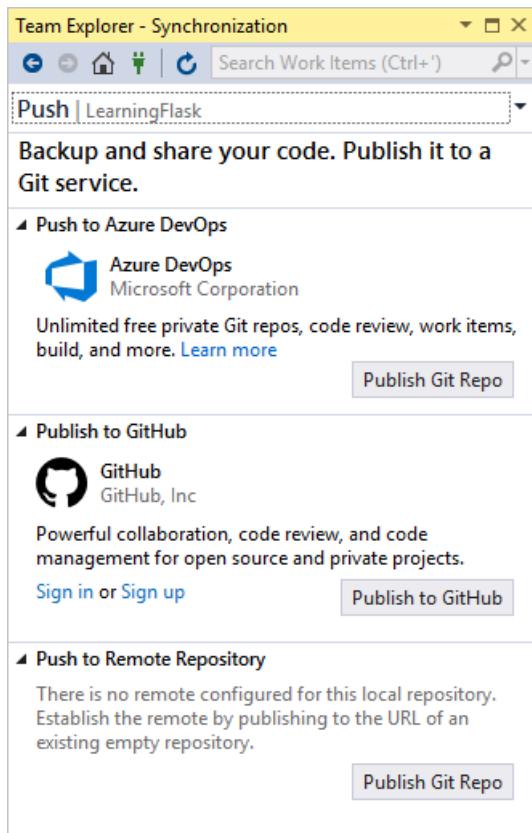
If you don't select the **Create new Git repository** in the **New Project** dialog, the Git controls show only an **Add to source control** command that creates a local repository.



2. Select the changes button, and Visual Studio opens its **Team Explorer** window on the **Changes** page. Because the newly created project is already committed to source control automatically, you don't see any pending changes.

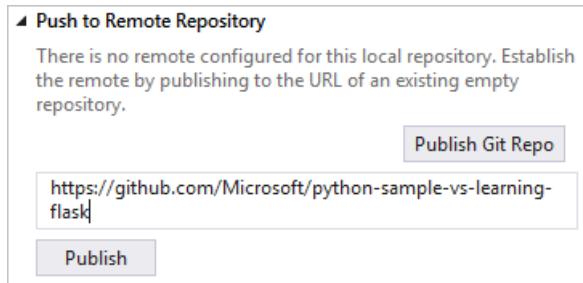


3. On the Visual Studio status bar, select the unpushed commits button (the up arrow with 2) to open the **Synchronization** page in **Team Explorer**. Because you have only a local repository, the page provides easy options to publish the repository to different remote repositories.



You can choose whichever service you want for your own projects. This tutorial shows the use of GitHub, where the completed sample code for the tutorial is maintained in the [Microsoft/python-sample-vs-learning-flask](#) repository.

- When selecting any of the **Publish** controls, **Team Explorer** prompts you for more information. For example, when publishing the sample for this tutorial, the repository itself had to be created first, in which case the **Push to Remote Repository** option was used with the repository's URL.



If you don't have an existing repository, the **Publish to GitHub** and **Push to Azure DevOps** options let you create one directly from within Visual Studio.

- As you work through this tutorial, get into the habit of periodically using the controls in Visual Studio to commit and push changes. This tutorial reminds you at appropriate points.

TIP

To quickly navigate within **Team Explorer**, select the header (that reads **Changes** or **Push** in the images above) to see a pop-up menu of the available pages.

Question: What are some advantages of using source control from the beginning of a project?

Answer: First of all, using source control from the start, especially if you also use a remote repository, provides a regular offsite backup of your project. Unlike maintaining a project just on a local file system, source control also provides a complete change history and the easy ability to revert a single file or the whole project to a previous state. That change history helps determine the cause of regressions (test failures). Furthermore, source control is

essential if multiple people are working on a project, as it manages overwrites and provides conflict resolution. Finally, source control, which is fundamentally a form of automation, sets you up well for automating builds, testing, and release management. It's really the first step in using DevOps for a project, and because the barriers to entry are so low, there's really no reason to not use source control from the beginning.

For further discussion on source control as automation, see [The Source of Truth: The Role of Repositories in DevOps](#), an article in MSDN Magazine written for mobile apps that applies also to web apps.

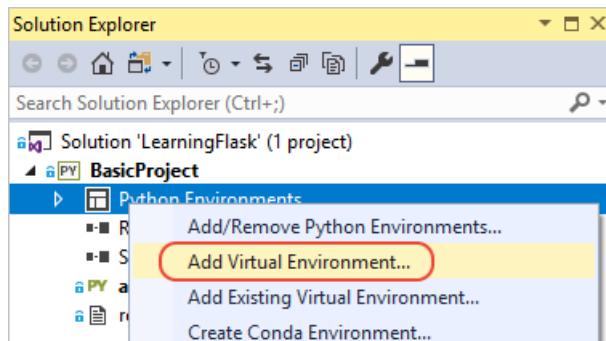
Question: Can I prevent Visual Studio from auto-committing a new project?

Answer: Yes. To disable auto-commit, go to the **Settings** page in **Team Explorer**, select **Git > Global settings**, clear the option labeled **Commit changes after merge by default**, then select **Update**.

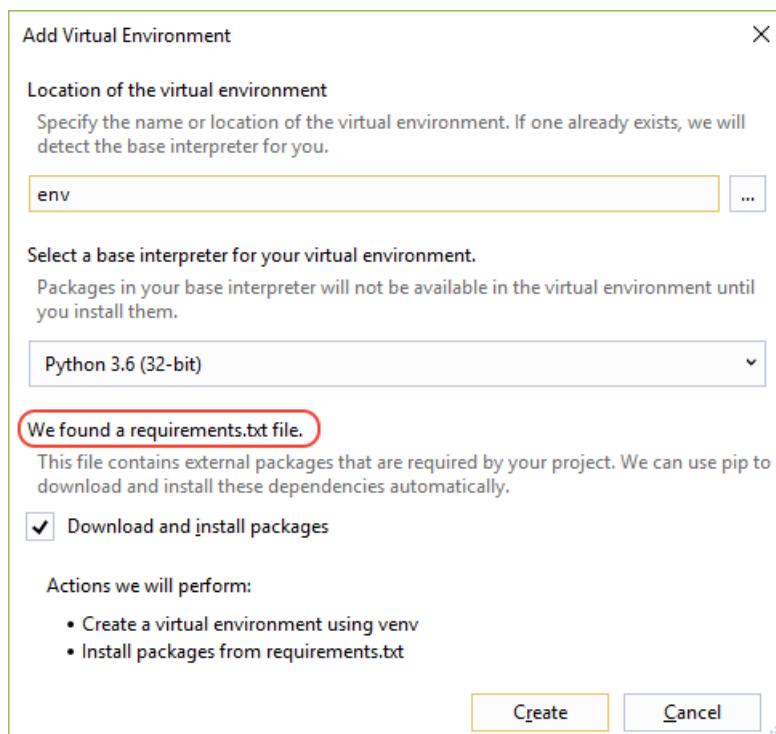
Step 1-3: Create the virtual environment and exclude it from source control

Now that you've configured source control for your project, you can create the virtual environment the necessary Flask packages that the project requires. You can then use **Team Explorer** to exclude the environment's folder from source control.

1. In **Solution Explorer**, right-click the **Python Environments** node and select **Add Virtual Environment**.



2. An **Add Virtual Environment** dialog appears, with a message saying **We found a requirements.txt file**. This message indicates that Visual Studio uses that file to configure the virtual environment.



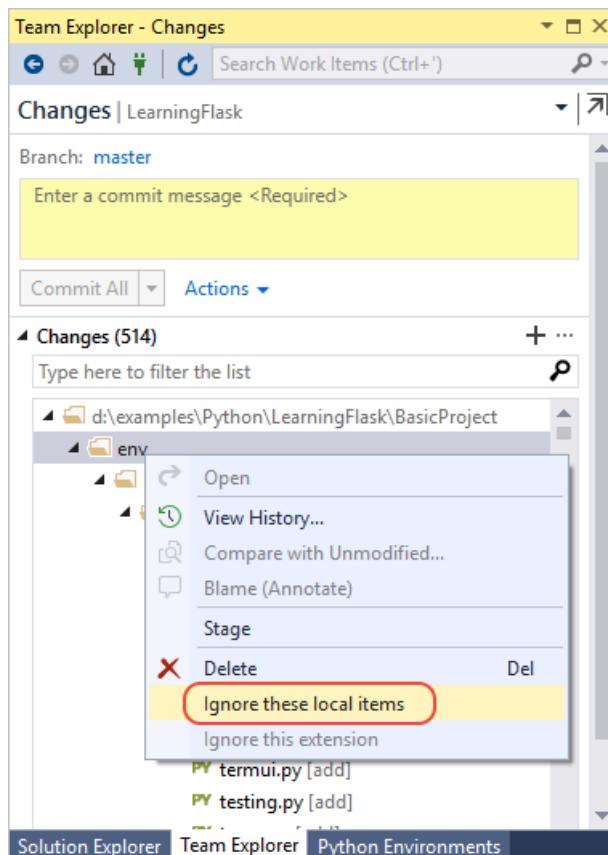
3. Select **Create** to accept the defaults. (You can change the name of the virtual environment if you want,

which just changes the name of its subfolder, but `env` is a standard convention.)

4. Consent to administrator privileges if prompted, then be patient for a few minutes while Visual Studio downloads and installs packages, which for Flask and its dependencies means expanding about a thousand files in over 100 subfolders. You can see progress in the Visual Studio **Output** window. While you're waiting, ponder the Question sections that follow. You can also see a description of Flask's dependencies on the [Flask installation](#) page (flask.pocoo.org).
5. On the Visual Studio Git controls (on the status bar), select the changes indicator (that shows **99***) which opens the **Changes** page in **Team Explorer**.

Creating the virtual environment brought in hundreds of changes, but you don't need to include any of them in source control because you (or anyone else cloning the project) can always recreate the environment from `requirements.txt`.

To exclude the virtual environment, right-click the `env` folder and select **Ignore these local items**.



6. After excluding the virtual environment, the only remaining changes are to the project file and `.gitignore`. The `.gitignore` file contains an added entry for the virtual environment folder. You can double-click the file to see a diff.
7. Enter a commit message and select the **Commit All** button, then push the commits to your remote repository if you like.

Question: Why do I want to create a virtual environment?

Answer: A virtual environment is a great way to isolate your app's exact dependencies. Such isolation avoids conflicts within a global Python environment, and aids both testing and collaboration. Over time, as you develop an app, you invariably bring in many helpful Python packages. By keeping packages in a project-specific virtual environment, you can easily update the project's `requirements.txt` file that describes that environment, which is included in source control. When the project is copied to any other computers, including build servers, deployment servers, and other development computers, it's easy to recreate the environment using only `requirements.txt` (which is why the environment doesn't need to be in source control). For more information, see [Use virtual environments](#).

Question: How do I remove a virtual environment that's already committed to source control?

Answer: First, edit your `.gitignore` file to exclude the folder: find the section at the end with the comment

```
# Python Tools for Visual Studio (PTVS)
```

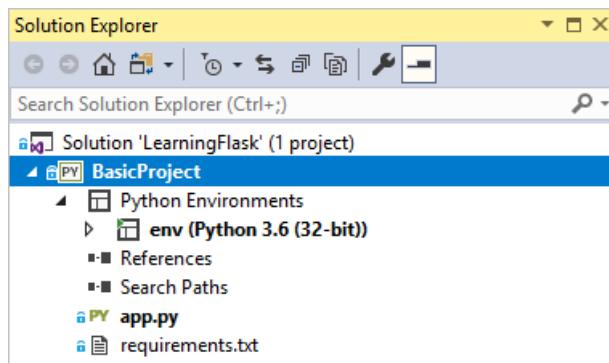
and add a new line for the virtual environment folder, such as
`/BasicProject/env`. (Because Visual Studio doesn't show the file in **Solution Explorer**, open it directly using the **File > Open > File** menu command. You can also open the file from **Team Explorer**: on the **Settings** page, select **Repository Settings**, go to the **Ignore & Attributes Files** section, then select the **Edit** link next to `.gitignore`.)

Second, open a command window, navigate to the folder like `BasicProject` that contains the virtual environment folder such as `env`, and run `git rm -r env`. Then commit those changes from the command line (

```
git commit -m 'Remove venv'
```

Step 1-4: Examine the boilerplate code

- Once project creation completes, you see the solution and project in **Solution Explorer**, where the project contains only two files, `app.py` and `requirements.txt`:



- As noted earlier, the `requirements.txt` file specifies the Flask package dependency. The presence of this file is what invites you to create a virtual environment when first creating the project.
- The single `app.py` file contains three parts. First is an `import` statement for Flask, creating an instance of the `Flask` class, which is assigned to the variable `app`, and then assigning a `wsgi_app` variable (which is useful when deploying to a web host, but not used at present):

```
from flask import Flask
app = Flask(__name__)

# Make the WSGI interface available at the top level so wfastcgi can get it.
wsgi_app = app.wsgi_app
```

- The second part, at the end of the file, is a bit of optional code that starts the Flask development server with specific host and port values taken from environment variables (defaulting to localhost:5555):

```
if __name__ == '__main__':
    import os
    HOST = os.environ.get('SERVER_HOST', 'localhost')
    try:
        PORT = int(os.environ.get('SERVER_PORT', '5555'))
    except ValueError:
        PORT = 5555
    app.run(HOST, PORT)
```

- Third is a short bit of code that assigns a function to a URL route, meaning that the function provides the resource identified by the URL. You define routes using Flask's `@app.route` decorator, whose argument is the relative URL from the site root. As you can see in the code, the function here returns only a text string, which is enough for a browser to render. In the steps that follow you render richer pages with HTML.

```
@app.route('/')
def hello():
    """Renders a sample page."""
    return "Hello World!"
```

Question: What is the purpose of the name argument to the Flask class?

Answer: The argument is the name of the app's module or package, and tells Flask where to look for templates, static files, and other resources that belong to the app. For apps contained in a single module, `__name__` is always the proper value. It's also important for extensions that need debugging information. For more information, and additional arguments, see the [Flask class documentation](#) (flask.pocoo.org).

Question: Can a function have more than one route decorator?

Answer: Yes, you can use as many decorators as you want if the same function serves multiple routes. For example, to use the `hello` function for both "/" and "/hello", use the following code:

```
@app.route('/')
@app.route('/hello')
def hello():
    """Renders a sample page."""
    return "Hello World!"
```

Question: How does Flask work with variable URL routes and query parameters?

Answer: In a route, you mark any variable with `<variable_name>`, and Flask passes the variable to the function using a named argument. The variable can be part of the URL path or in a query parameter. For example, a route in the form of `'/hello/<name>'` generates a string argument called `name` to the function, and using `?message=<msg>` in the route parses the value given for the "message=" query parameter and passes it to the function as `msg`:

```
@app.route('/hello/<name>?message=<msg>')
def hello(name, msg):
    return "Hello " + name + "! Message is " + msg + "."
```

To change the type, prefix the variable with `int`, `float`, `path` (which accepts slashes to delineate folder names), and `uuid`. For details, see [Variable rules](#) in the Flask documentation.

Query parameters are also available through the `request.args` property, specifically through the `request.args.get` method. For more information, see [The Request object](#) in the Flask documentation.

Question: Can Visual Studio generate a requirements.txt file from a virtual environment after I install other packages?

Answer: Yes. Expand the **Python Environments** node, right-click your virtual environment, and select the **Generate requirements.txt** command. It's good to use this command periodically as you modify the environment, and commit changes to `requirements.txt` to source control along with any other code changes that depend on that environment. If you set up continuous integration on a build server, you should generate the file and commit changes whenever you modify the environment.

Step 1-5: Run the project

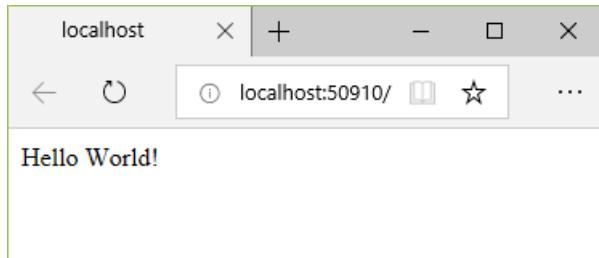
1. In Visual Studio, select **Debug > Start Debugging (F5)** or use the **Web Server** button on the toolbar (the browser you see may vary):



2. Either command assigns a random port number to the PORT environment variable, then runs

`python app.py`. The code starts the app using that port within Flask's development server. If Visual Studio says **Failed to start debugger** with a message about having no startup file, right-click **app.py** in **Solution Explorer** and select **Set as Startup File**.

- When the server starts, you see a console window open that displays the server log. Visual Studio then automatically opens a browser to `http://localhost:<port>`, where you should see the message rendered by the `hello` function:



- When you're done, stop the server by closing the console window, or by using the **Debug > Stop Debugging** command in Visual Studio.

Question: What's the difference between using the Debug menu commands and the server commands on the project's Python submenu?

Answer: In addition to the **Debug** menu commands and toolbar buttons, you can also launch the server using the **Python > Run server** or **Python > Run debug server** commands on the project's context menu. Both commands open a console window in which you see the local URL (localhost:port) for the running server. However, you must manually open a browser with that URL, and running the debug server does not automatically start the Visual Studio debugger. You can attach a debugger to the running process later, if you want, using the **Debug > Attach to Process** command.

Next steps

At this point, the basic Flask project contains the startup code and page code in the same file. It's best to separate these two concerns, and to also separate HTML and data by using templates.

[Create a Flask app with views and page templates](#)

Go deeper

- [Flask Quickstart](#) (flask.pocoo.org)
- Tutorial source code on GitHub: [Microsoft/python-sample-vs-learning-flask](https://github.com/Microsoft/python-sample-vs-learning-flask)

Step 2: Create a Flask app with views and page templates

4/18/2019 • 7 minutes to read • [Edit Online](#)

Previous step: [Create a Visual Studio project and solution](#)

What you have from step 1 of this tutorial is a Flask app with one page and all the code in a single file. To allow for future development, it's best to refactor the code and create a structure for page templates. In particular, you want to separate code for the app's views from other aspects like startup code.

In this step you now learn how to:

- Refactor the app's code to separate views from startup code (step 2-1)
- Render a view using a page template (step 2-2)

Step 2-1: Refactor the project to support further development

In the code created by the "Blank Flask Web Project" template, you have a single `app.py` file that contains startup code alongside a single view. To allow for further development of an app with multiple views and templates, it's best to separate these concerns.

1. In your project folder, create an app folder called `HelloFlask` (right-click the project in **Solution Explorer** and select **Add > New Folder**.)
2. In the `HelloFlask` folder, create a file named `__init__.py` with the following contents that creates the `Flask` instance and loads the app's views (created in the next step):

```
from flask import Flask
app = Flask(__name__)

import HelloFlask.views
```

3. In the `HelloFlask` folder, create a file named `views.py` with the following contents. The name `views.py` is important because you used `import HelloFlask.views` within `__init__.py`; you'll see an error at run time if the names don't match.

```
from flask import Flask
from HelloFlask import app

@app.route('/')
@app.route('/home')
def home():
    return "Hello Flask!"
```

In addition to renaming the function and route to `home`, this code contains the page rendering code from `app.py` and imports the `app` object that's declared in `__init__.py`.

4. Create a subfolder in `HelloFlask` named `templates`, which remains empty for now.
5. In the project's root folder, rename `app.py` to `runserver.py`, and make the contents match the following code:

```

import os
from HelloFlask import app      # Imports the code from HelloFlask/__init__.py

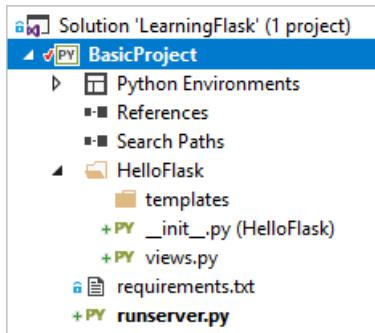
if __name__ == '__main__':
    HOST = os.environ.get('SERVER_HOST', 'localhost')

    try:
        PORT = int(os.environ.get('SERVER_PORT', '5555'))
    except ValueError:
        PORT = 5555

    app.run(HOST, PORT)

```

6. Your project structure should look like the following image:



7. Select **Debug > Start Debugging (F5)** or use the **Web Server** button on the toolbar (the browser you see may vary) to start the app and open a browser. Try both the / and /home URL routes.
8. You can also set breakpoints at various parts of the code and restart the app to follow the startup sequence. For example, set a breakpoint on the first lines of `runserver.py` and `HelloFlask_init_.py`, and on the `return "Hello Flask!"` line in `views.py`. Then restart the app (**Debug > Restart, Ctrl+F5**, or the toolbar button shown below) and step through (**F10**) the code, or run from each breakpoint using **F5**.

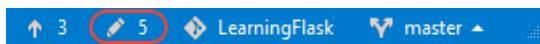


9. Stop the app when you're done.

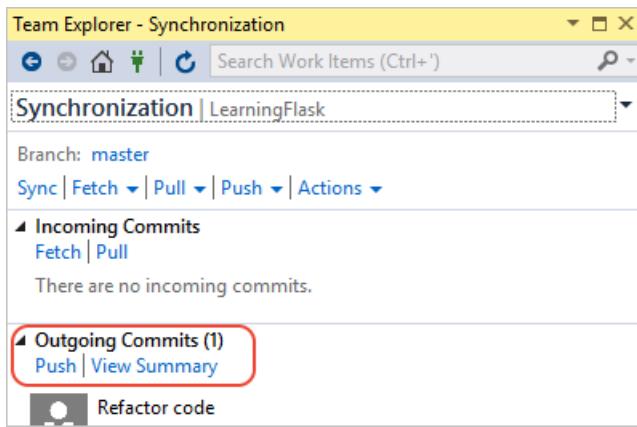
Commit to source control

Because you've made changes to your code and have tested them successfully, now is a great time to review and commit your changes to source control. Later steps in this tutorial remind you of appropriate times to commit to source control again, and refer you back to this section.

1. Select the changes button along the bottom of Visual Studio (circled below), which navigates to **Team Explorer**.



2. In **Team Explorer**, enter a commit message like "Refactor code" and select **Commit All**. When the commit is complete, you see a message **Commit <hash> created locally. Sync to share your changes with the server.** If you want to push changes to your remote repository, select **Sync**, then select **Push** under **Outgoing Commits**. You can also accumulate multiple local commits before pushing to remote.



Question: How frequently should one commit to source control?

Answer: Committing changes to source control creates a record in the change log and a point to which you can revert the repository if necessary. Each commit can also be examined for its specific changes. Because commits in Git are inexpensive, it's better to do frequent commits than to accumulate a larger number of changes into a single commit. Clearly, you don't need to commit every small change to individual files. Typically you make a commit when adding a feature, changing a structure like you've done in this step, or done some code refactoring. Also check with others in your team for the granularity of commits that work best for everyone.

How often you commit and how often you push commits to a remote repository are two different concerns. You may accumulate multiple commits in your local repository before pushing them to the remote repository. Again, how frequently you commit depends on how your team wants to manage the repository.

Step 2-2: Use a template to render a page

The `home` function that you have so far in `views.py` generates nothing more than a plain-text HTTP response for the page. However, most real-world web pages, respond with rich HTML pages that often incorporate live data. Indeed, the primary reason to define a view using a function is to generate content dynamically.

Because the return value for the view is just a string, you can build up any HTML you like within a string, using dynamic content. However, because it's best to separate markup from data, it's much better to place the markup in a template and keep the data in code.

1. For starters, edit `views.py` to contain the following code that uses inline HTML for the page with some dynamic content:

```
from datetime import datetime
from flask import render_template
from HelloFlask import app

@app.route('/')
@app.route('/home')
def home():
    now = datetime.now()
    formatted_now = now.strftime("%A, %d %B, %Y at %X")

    html_content = "<html><head><title>Hello Flask</title></head><body>"
    html_content += "<strong>Hello Flask!</strong> on " + formatted_now
    html_content += "</body></html>"

    return html_content
```

2. Run the app and refresh the page a few times to see that the date/time is updated. Stop the app when you're done.
3. To convert the page rendering to use a template, create a file named `index.html` in the `templates` folder with

the following content, where `{{ content }}` is a placeholder or replacement token (also called a *template variable*) for which you supply a value in the code:

```
<html>
  <head><title>Hello Flask</title></head>

  <body>
    {{ content }}
  </body>
</html>
```

4. Modify the `home` function to use `render_template` to load the template and supply a value for "content", which is done using a named argument matching the name of the placeholder. Flask automatically looks for templates in the *templates* folder, so the path to the template is relative to that folder:

```
def home():
    now = datetime.now()
    formatted_now = now.strftime("%A, %d %B, %Y at %X")

    return render_template(
        "index.html",
        content = "<strong>Hello, Flask!</strong> on " + formatted_now)
```

5. Run the app and see the results, and observe that the inline HTML in the `content` value doesn't render as HTML because the templating engine (Jinja) automatically escapes HTML content. Automatic escaping prevents accidental vulnerabilities to injection attacks: developers often gather input from one page and use it as a value in another through a template placeholder. Escaping also serves as a reminder that it's again best to keep HTML out of the code.

Accordingly, review *templates\index.html* to contain distinct placeholders for each piece of data within the markup:

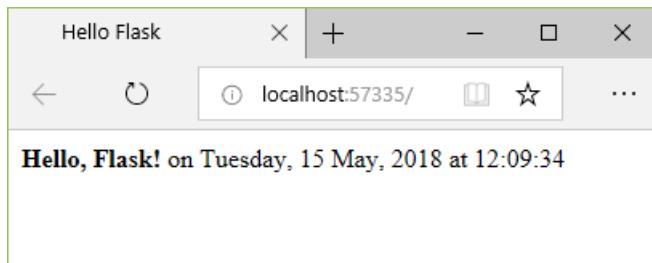
```
<html>
  <head>
    <title>{{ title }}</title>
  </head>
  <body>
    <strong>{{ message }}</strong>{{ content }}
  </body>
</html>
```

Then update the `home` function to provide values for all the placeholders:

```
def home():
    now = datetime.now()
    formatted_now = now.strftime("%A, %d %B, %Y at %X")

    return render_template(
        "index.html",
        title = "Hello Flask",
        message = "Hello, Flask!",
        content = " on " + formatted_now)
```

6. Run the app again to see the properly rendered output.



7. Commit your changes to source control and update your remote repository, if desired, as described under [step 2-1](#).

Question: Do page templates have to be in a separate file?

Answer: Although templates are usually maintained in separate HTML files, you can also use an inline template. Using a separate file is recommended, however, to maintain a clean separation between markup and code.

Question: Must templates use the .html file extension?

Answer: The `.html` extension for page template files is entirely optional, because you always identify the exact relative path to the file in the first argument to the `render_template` function. However, Visual Studio (and other editors) typically give you features like code completion and syntax coloration with `.html` files, which outweighs the fact that page templates are not strictly HTML.

In fact, when you're working with a Flask project, Visual Studio automatically detects when the HTML file you're editing is actually a Flask template, and provides certain auto-complete features. For example, when you start typing a Flask page template comment, `{#`, Visual Studio automatically gives you the closing `#}` characters. The **Comment Selection** and **Uncomment Selection** commands (on the **Edit > Advanced** menu and on the toolbar) also use template comments instead of HTML comments.

Question: When I run the project, I see an error that the template cannot be found. What's wrong?

Answer: If you see errors that the template cannot be found, make sure you added the app to the Flask project's `settings.py` in the `INSTALLED_APPS` list. Without that entry, Flask won't know to look in the app's `templates` folder.

Question: Can templates be organized into further subfolders?

Answer: Yes, you can use subfolders and then refer to the relative path under `templates` in calls to `render_template`. Doing so is a great way to effectively create namespaces for your templates.

Next steps

[Serve static files, add pages, and use template inheritance](#)

Go deeper

- [Flask Quickstart - Rendering Templates](#) (`flask.pocoo.org`)
- Tutorial source code on GitHub: [Microsoft/python-sample-vs-learning-flask](#)

Step 3: Serve static files, add pages, and use template inheritance

4/9/2019 • 9 minutes to read • [Edit Online](#)

Previous step: Create a Flask app with views and page templates

In the previous steps of this tutorial, you've learned how to create a minimal Flask app with a single page of self-contained HTML. Modern web apps, however, are typically composed of many pages, and make use of shared resources like CSS and JavaScript files to provide consistent styling and behavior.

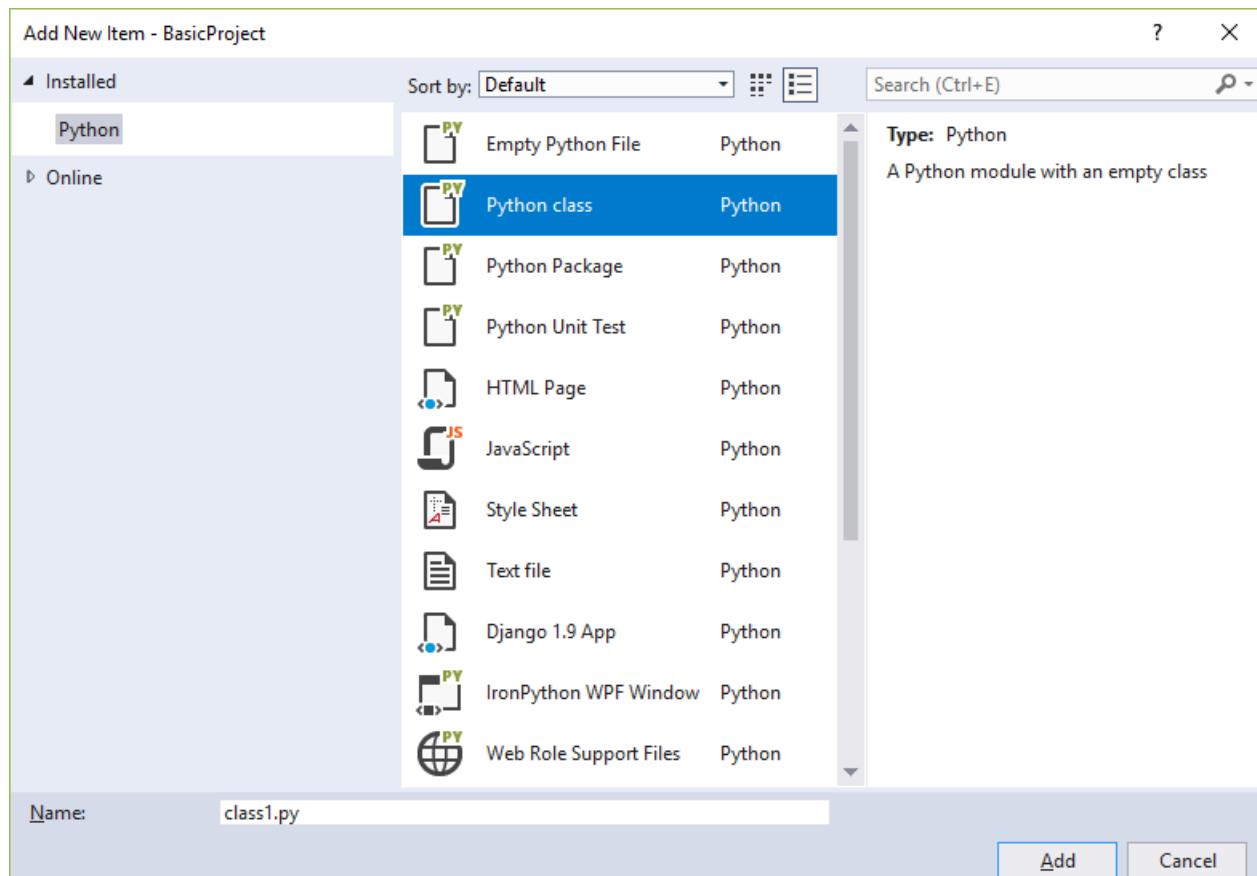
In this step, you learn how to:

- Use Visual Studio item templates to quickly add new files of different types with convenient boilerplate code (step 3-1)
- Serve static files from code (step 3-2, optional)
- Add additional pages to the app (step 3-3)
- Use template inheritance to create a header and nav bar that's used across pages (step 3-4)

Step 3-1: Become familiar with item templates

As you develop a Flask app, you typically add many more Python, HTML, CSS, and JavaScript files. For each file type (as well as other files like `web.config` that you may need for deployment), Visual Studio provides convenient [item templates](#) to get you started.

To see available templates, go to **Solution Explorer**, right-click the folder in which you want to create the item, select **Add > New Item**:



To use a template, select the desired template, specify a name for the file, and select **OK**. Adding an item in this manner automatically adds the file to your Visual Studio project and marks the changes for source control.

Question: How does Visual Studio know which item templates to offer?

Answer: The Visual Studio project file (`.pyproj`) contains a project type identifier that marks it as a Python project. Visual Studio uses this type identifier to show only those item templates that are suitable for the project type. This way, Visual Studio can supply a rich set of item templates for many project types without asking you to sort through them every time.

Step 3-2: Serve static files from your app

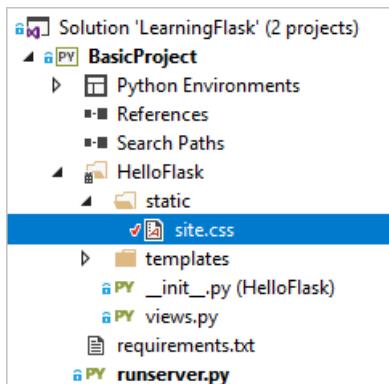
In a web app built with Python (using any framework), your Python files always run on the web host's server and are never transmitted to a user's computer. Other files, however, such as CSS and JavaScript, are used exclusively by the browser, so the host server simply delivers them as-is whenever they're requested. Such files are referred to as "static" files, and Flask can deliver them automatically without you needing to write any code. Within HTML files, for example, you can just refer to static files using a relative path in the project. The first section in this step adds a CSS file to your existing page template.

When you need to deliver a static file from code, such as through an API endpoint implementation, Flask provides a convenient method that lets you refer to files using relative paths within a folder named `static` (in the project root). The second section in this step demonstrates that method using a simple static data file.

In either case, you can organize files under `static` however you like.

Use a static file in a template

1. In **Solution Explorer**, right-click the **HelloFlask** folder in the Visual Studio project, select **Add > New folder**, and name the folder `static`.
2. Right-click the `static` folder and select **Add > New item**. In the dialog that appears, select the **Stylesheet** template, name the file `site.css`, and select **OK**. The `site.css` file appears in the project and is opened in the editor. Your folder structure should appear similar to the following image:



3. Replace the contents of `site.css` with the following code and save the file:

```
.message {  
    font-weight: 600;  
    color: blue;  
}
```

4. Replace the contents of the app's `templates/index.html` file with the following code, which replaces the `` element used in step 2 with a `` that references the `message` style class. Using a style class in this way gives you much more flexibility in styling the element.

```

<html>
  <head>
    <title>{{ title }}</title>
    <link rel="stylesheet" type="text/css" href="/static/site.css" />
  </head>
  <body>
    <span class="message">{{ message }}</span>{{ content }}
  </body>
</html>

```

- Run the project to observe the results. Stop the app when done, and commit your changes to source control if you like (as explained in [step 2](#)).

Serve a static file from code

Flask provides a function called `serve_static_file` that you can call from code to refer to any file within the project's *static* folder. The following process creates a simple API endpoint that returns a static data file.

- If you haven't done so already, create a *static* folder: in **Solution Explorer**, right-click the **HelloFlask** folder in the Visual Studio project, select **Add > New folder**, and name the folder `static`.
- In the *static* folder, create a static JSON data file named *data.json* with the following contents (which are meaningless sample data):

```
{
  "01": {
    "note" : "Data is very simple because we're demonstrating only the mechanism."
  }
}
```

- In *views.py*, add a function with the route `/api/data` that returns the static data file using the `send_static_file` method:

```

@app.route('/api/data')
def get_data():
    return app.send_static_file('data.json')

```

- Run the app and navigate to the `/api/data` endpoint to see that the static file is returned. Stop the app when you're done.

Question: Are there any conventions for organizing static files?

Answer: You can add other CSS, JavaScript, and HTML files in your *static* folder however you want. A typical way to organize static files is to create subfolders named *fonts*, *scripts*, and *content* (for stylesheets and any other files).

Question: How do I handle URL variables and query parameters in an API?

Answer: See the answer in step 1-4 for [Question: How does Flask work with variable URL routes and query parameters?](#)

Step 3-3: Add a page to the app

Adding another page to the app means the following:

- Add a Python function that defines the view.
- Add a template for the page's markup.
- Add the necessary routing to the Flask project's *urls.py* file.

The following steps add an "About" page to the "HelloFlask" project, and links to that page from the home page:

1. In **Solution Explorer**, right-click the **templates** folder, select **Add > New item**, select the **HTML Page** item template, name the file `about.html`, and select **OK**.

TIP

If the **New Item** command doesn't appear on the **Add** menu, make sure that you've stopped the app so that Visual Studio exits debugging mode.

2. Replace the contents of `about.html` with the following markup (you replace the explicit link to the home page with a simple navigation bar in step 3-4):

```
<html>
  <head>
    <title>{{ title }}</title>
    <link rel="stylesheet" type="text/css" href="/static/site.css" />
  </head>
  <body>
    <div><a href="home">Home</a></div>
    {{ content }}
  </body>
</html>
```

3. Open the app's `views.py` file and add a function named `about` that uses the template:

```
@app.route('/about')
def about():
    return render_template(
        "about.html",
        title = "About HelloFlask",
        content = "Example app page for Flask.")
```

4. Open the `templates/index.html` file and add the following line immediately within the `<body>` element to link to the About page (again, you replace this link with a nav bar in step 3-4):

```
<div><a href="about">About</a></div>
```

5. Save all the files using the **File > Save All** menu command, or just press **Ctrl+Shift+S**. (Technically, this step isn't needed as running the project in Visual Studio saves files automatically. Nevertheless, it's a good command to know about!)

6. Run the project to observe the results and check navigation between pages. Stop the app when done.

Question: Does the name of a page function matter to Flask?

Answer: No, because it's the `@app.route` decorator that determines the URLs for which Flask calls the function to generate a response. Developers typically match the function name to the route, but such matching isn't required.

Step 3-4: Use template inheritance to create a header and nav bar

Instead of having explicit navigation links on each page, modern web apps typically use a branding header and a navigation bar that provides the most important page links, popup menus, and so on. To make sure the header and nav bar are the same across all pages, however, you don't want to repeat the same code in every page template. You instead want to define the common parts of all your pages in one place.

Flask's templating system (Jinja by default) provides two means for reusing specific elements across multiple templates: includes and inheritance.

- *Includes* are other page templates that you insert at a specific place in the referring template using the syntax `{% include <template_path> %}`. You can also use a variable if you want to change the path dynamically in code. Includes are typically used in the body of a page to pull in the shared template at a specific location on the page.
- *Inheritance* uses the `{% extends <template_path> %}` at the beginning of a page template to specify a shared base template that the referring template then builds upon. Inheritance is commonly used to define a shared layout, nav bar, and other structures for an app's pages, such that referring templates need only add or modify specific areas of the base template called *blocks*.

In both cases, `<template_path>` is relative to the app's *templates* folder (`../` or `./` are also allowed).

A base template delineates *blocks* using `{% block <block_name> %}` and `{% endblock %}` tags. If a referring template then uses tags with the same block name, its block content overrides that of the base template.

The following steps demonstrate inheritance:

1. In the app's *templates* folder, create a new HTML file (using the **Add > New item** context menu or **Add > HTML Page**) called *layout.html*, and replace its contents with the markup below. You can see that this template contains a block named "content" that is all that the referring pages need to replace:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <title>{{ title }}</title>
    <link rel="stylesheet" type="text/css" href="/static/site.css" />
</head>

<body>
    <div class="navbar">
        <a href="/" class="navbar-brand">Hello Flask</a>
        <a href="{{ url_for('home') }}" class="navbar-item">Home</a>
        <a href="{{ url_for('about') }}" class="navbar-item">About</a>
    </div>

    <div class="body-content">
        {% block content %}
        {% endblock %}
        <hr/>
        <footer>
            <p>&copy; 2018</p>
        </footer>
    </div>
</body>
</html>
```

2. Add the following styles to the app's *static/site.css* file (this walkthrough isn't attempting to demonstrate responsive design here; these styles are simply to generate an interesting result):

```

.navbar {
    background-color: lightslategray;
    font-size: 1em;
    font-family: 'Trebuchet MS', 'Lucida Sans Unicode', 'Lucida Grande', 'Lucida Sans', Arial, sans-serif;
    color: white;
    padding: 8px 5px 8px 5px;
}

.navbar a {
    text-decoration: none;
    color: inherit;
}

.navbar-brand {
    font-size: 1.2em;
    font-weight: 600;
}

.navbar-item {
    font-variant: small-caps;
    margin-left: 30px;
}

.body-content {
    padding: 5px;
    font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
}

```

3. Modify `templates/index.html` to refer to the base template and override the content block. You can see that by using inheritance, this template becomes simple:

```

{% extends "layout.html" %}
{% block content %}
<span class="message">{{ message }}</span>{{ content }}
{% endblock %}

```

4. Modify `templates/about.html` to also refer to the base template and override the content block:

```

{% extends "layout.html" %}
{% block content %}
{{ content }}
{% endblock %}

```

5. Run the server to observe the results. Close the server when done.



6. Because you made substantial changes to the app, it's again a good time to [commit your changes to source control](#).

Next steps

[Use the full Flask Web Project template](#)

Go deeper

- [Deploy the web app to Azure App Service](#)
- For more capabilities of Jinja templates, such as control flow, see [Jinja Template Designer Documentation](#) (jinja.pocoo.org)
- For details on using `url_for`, see [url_for](#) within the Flask Application object documentation (flask.pocoo.org)
- Tutorial source code on GitHub: [Microsoft/python-sample-vs-learning-flask](#)

Step 4: Use the full Flask Web Project template

4/18/2019 • 6 minutes to read • [Edit Online](#)

Previous step: [Serve static files, add pages, and use template inheritance](#)

Now that you've explored the basics of Flask by building an app upon the "Blank Flask App Project" template in Visual Studio, you can easily understand the fuller app that's produced by the "Flask Web Project" template.

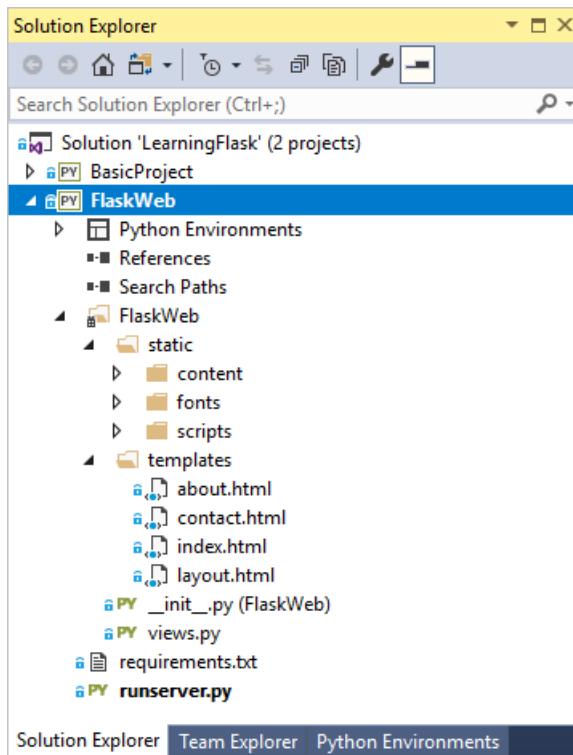
In this step you now:

- Create a fuller Flask web app using the "Flask Web Project" template and examine the project structure (step 4-1)
- Understand the views and page templates created by the project template, which consist of three pages that inherit from a base page template and that employs static JavaScript libraries like jQuery and Bootstrap (step 4-2)
- Understand the URL routing provided by the template (step 4-3)

This article applies also to the "Flask/Jade Web Project" template, which produces an app that's identical to that of the "Flask Web Project" using the Jade templating engine instead of Jinja. Additional details are included at the end of this article.

Step 4-1: Create a project from the template

1. In Visual Studio, go to **Solution Explorer**, right-click the **LearningFlask** solution created earlier in this tutorial, and select **Add > New Project**. (Alternately, if you want to use a new solution, select **File > New > Project** instead.)
2. In the new project dialog, search for and select the **Flask Web Project** template, call the project "FlaskWeb", and select **OK**.
3. Because the template again includes a *requirements.txt* file, Visual Studio asks where to install those dependencies. Choose the option, **Install into a virtual environment**, and in the **Add Virtual Environment** dialog select **Create** to accept the defaults.
4. Once Visual Studio finishes setting up the virtual environment, set the **FlaskWeb** project to be the default for the Visual Studio solution by right-clicking that project in **Solution Explorer** and selecting **Set as Startup Project**. The startup project, which is shown in bold, is what's run when you start the debugger.



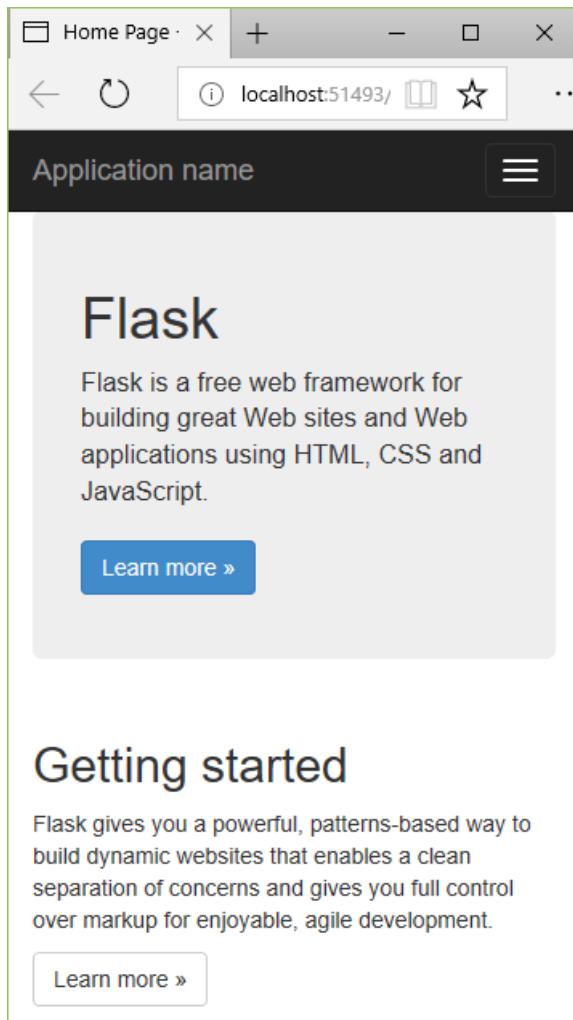
5. Select **Debug > Start Debugging (F5)** or use the **Web Server** button on the toolbar to run the server:



6. The app created by the template has three pages, Home, About, and Contact, which you navigate between using the nav bar. Take a minute or two to examine different parts of the app. To authenticate with the app through the **Log in** command, use the superuser credentials created earlier.

The screenshot shows a Microsoft Edge browser window displaying the 'Home Page - My Flask' page. The address bar shows 'localhost:51493/'. The page content includes a large 'Flask' logo, a brief description: 'Flask is a free web framework for building great Web sites and Web applications using HTML, CSS and JavaScript.', and a 'Learn more »' button. Below this, there are three sections: 'Getting started', 'Get more libraries', and 'Microsoft Azure', each with a 'Learn more »' button.

7. The app created by the "Flask Web Project" template uses Bootstrap for responsive layout that accommodates mobile form factors. To see this responsiveness, resize the browser to a narrow view so that the content renders vertically and the nav bar turns into a menu icon:



8. You can leave the app running for the sections that follow.

If you want to stop the app and [commit changes to source control](#), first open the **Changes** page in **Team Explorer**, right-click the folder for the virtual environment (probably **env**), and select **Ignore these local items**.

Examine what the template creates

The "Flask Web Project" template creates the structure below. The contents are very similar to what you created in previous steps. The difference is that the "Flask Web Project" template contains more structure in the *static* folder because it includes jQuery and Bootstrap for responsive design. The template also adds a Contact page. Overall, if you've followed the previous steps in this tutorial, everything from the template should be familiar.

- Files in the project root:
 - *runserver.py*, a script to run the app in a development server.
 - *requirements.txt* containing a dependency on Flask 0.x.
- The *FlaskWeb* folder contains all the app files:
 - *__init__.py* marks the app code as a Python module, creates the Flask object, and imports the app's views.
 - *views.py* contains the code to render pages.
 - The *static* folder contains subfolders named *content* (CSS files), *fonts* (font files), and *scripts* (JavaScript files).
 - The *templates* folder contains a *layout.html* base template along with *about.html*, *contact.html*, and *index.html* for specific pages that each extend *layout.html*.

Question: Is it possible to share a virtual environment between Visual Studio projects?

Answer: Yes, but do so with the awareness that different projects likely use different packages over time, and therefore a shared virtual environment must contain all the packages for all projects that use it.

Nevertheless, to use an existing virtual environment, do the following:

1. When prompted to install dependencies in Visual Studio, select **I will install them myself** option.
2. In **Solution Explorer**, right-click the **Python Environments** node and select **Add Existing Virtual Environment**.
3. Navigate to and select the folder containing the virtual environment, then select **OK**.

Step 4-2: Understand the views and page templates created by the project template

As you observe when you run the project, the app contains three views: Home, About, and Contact. The code for these views is found in the *FlaskWeb/views.py*. Each view function simply calls `flask.render_template` with the path to a template and a variable list of arguments for the values to give to the template. For example, the About page is handled by the `about` function (whose decorator provides the URL routing):

```
@app.route('/about')
def about():
    """Renders the about page."""
    return render_template(
        'about.html',
        title='About',
        year=datetime.now().year,
        message='Your application description page.'
    )
```

The `home` and `contact` functions are nearly identical, with similar decorators and slightly different arguments.

Templates are located in the app's *templates* folder. The base template, *layout.html*, is the most extensive. It refers to all the necessary static files (JavaScript and CSS), defines a block named "content" that other pages override, and provides another block named "scripts". The following annotated excerpts from *layout.html* show these specific areas:

```

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />

    <!-- Define a viewport for Bootstrap's responsive rendering -->
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>{{ title }} - My Flask Application</title>

    <link rel="stylesheet" type="text/css" href="/static/content/bootstrap.min.css" />
    <link rel="stylesheet" type="text/css" href="/static/content/site.css" />
    <script src="/static/scripts/modernizr-2.6.2.js"></script>
</head>

<body>
    <!-- Navbar omitted -->

    <div class="container body-content">
        <!-- "content" block that pages are expected to override -->
        {% block content %}{% endblock %}
        <hr />
        <footer>
            <p>&copy; {{ year }} - My Flask Application</p>
        </footer>
    </div>

    <!-- Additional scripts; use the "scripts" block to add page-specific scripts. -->
    <script src="/static/scripts/jquery-1.10.2.js"></script>
    <script src="/static/scripts/bootstrap.js"></script>
    <script src="/static/scripts/respond.js"></script>
    {% block scripts %}{% endblock %}
</body>
</html>

```

The individual page templates, *about.html*, *contact.html*, and *index.html*, each extend the base template *layout.html*. *about.html* is the simplest and shows the `{% extends %}` and `{% block content %}` tags:

```

{% extends "app/layout.html" %}

{% block content %}

<h2>{{ title }}.</h2>
<h3>{{ message }}</h3>

<p>Use this area to provide additional information.</p>

{% endblock %}

```

index.html and *contact.html* use the same structure and provide lengthier content in the "content" block.

The Flask/Jade Web Project template

As noted at the beginning of this article, Visual Studio provide a "Flask/Jade Web Project" template, which creates an application that's visually identical to what's produced by the "Flask Web Project". The primary difference is that it uses the Jade templating engine, which is an extension to Jinja that implements the same concepts with a more succinct language. Specifically, Jade uses keywords instead of tags enclosed in `{% %}` delimiters, for example, and lets you refer to CSS styles and HTML elements using keywords.

To enable Jade, the project template first includes the `pyjade` package in *requirements.txt*.

The app's `__init__.py` file contains a line to

```
app.jinja_env.add_extension('pyjade.ext.jinja.PyJadeExtension')
```

In the `templates` folder, you see `jade` files instead of `.html` templates, and the views in `views.py` refer to these files in their calls to `flask.render_template`. Otherwise the views code is the same.

Opening one of the `jade` files, you can see the more succinct expression of a template. For example, here's the contents of `templates/layout.jade` as created by the "Flask/Jade Web Project" template:

```
doctype html
html
  head
    meta(charset='utf-8')
    meta(name='viewport', content='width=device-width, initial-scale=1.0')
    title #{title} - My Flask/Jade Application
    link(rel='stylesheet', type='text/css', href='/static/content/bootstrap.min.css')
    link(rel='stylesheet', type='text/css', href='/static/content/site.css')
    script(src='/static/scripts/modernizr-2.6.2.js')
  body
    .navbar.navbar-inverse.navbar-fixed-top
      .container
        .navbar-header
          button.navbar-toggle(type='button', data-toggle='collapse', data-target='.navbar-collapse')
          span.icon-bar
          span.icon-bar
          span.icon-bar
        a.navbar-brand(href='/') Application name
        .navbar-collapse.collapse
          ul.nav.navbar-nav
            li
              a(href='/') Home
            li
              a(href='/about') About
            li
              a(href='/contact') Contact
      .container.body-content
        block content
        hr
        footer
          p &copy; #{year} - My Flask/Jade Application

        script(src='/static/scripts/jquery-1.10.2.js')
        script(src='/static/scripts/bootstrap.js')
        script(src='/static/scripts/respond.js')

    block scripts
```

And here's the contents of `templates/about.jade`, showing the use of `#{<name>}` for placeholders:

```
extends layout

block content
  h2 #{title}.
  h3 #{message}
  p Use this area to provide additional information.
```

Feel free to experiment with both Jinja and Jade syntaxes to see which one works best for you.

Next steps

[The Polls Flask Web Project template](#)

Go deeper

- [Writing your first Flask app, part 4 - forms and generic views](#) (docs.djangoproject.com)
- [Jade on GitHub \(Documentation\)](#) (github.com)
- Tutorial source code on GitHub: [Microsoft/python-sample-vs-learning-flask](#)

Step 5: Use the Polls Flask Web Project template

4/9/2019 • 11 minutes to read • [Edit Online](#)

Previous step: [Use the full Flask Web Project template](#)

Having understood Visual Studio's "Flask Web Project" template, you can now look at the third Flask template, "Polls Flask Web Project", which builds upon the same code base.

In this step you learn how to:

- Create a project from the template and initialize the database (step 5-1)
- Understand the data models (step 5-2)
- Understand the backing data stores (step 5-3)
- Understand the poll detail and results views (step 5-4)

Visual Studio also provides the "Polls Flask/Jade Web Project" template which produces an identical app, but uses the Jade extension for the Jinja templating engine. For details, see [Step 4 - The Flask/Jade Web Project template](#).

Step 5-1: Create the project

1. In Visual Studio, go to **Solution Explorer**, right-click the **LearningFlask** solution created earlier in this tutorial, and select **Add > New Project**. (Alternately, if you want to use a new solution, select **File > New > Project** instead.)
2. In the new project dialog, search for and select the **Polls Flask Web Project** template, call the project "FlaskPolls", and select **OK**.
3. Like the other project templates in Visual Studio, the "Polls Flask Web Project" template includes a *requirements.txt* file, Visual Studio asks where to install those dependencies. Choose the option, **Install into a virtual environment**, and in the **Add Virtual Environment** dialog select **Create** to accept the defaults. (This template requires Flask as well as the azure-storage and pymongo packages; the "Polls Flask/Jade Web Project" also requires pyjade.)
4. Set the **FlaskPolls** project to be the default for the Visual Studio solution by right-clicking that project in **Solution Explorer** and selecting **Set as Startup Project**. The startup project, which is shown in bold, is what's run when you start the debugger.
5. Select **Debug > Start Debugging (F5)** or use the **Web Server** button on the toolbar to run the server:



6. The app created by the template has three pages, Home, About, and Contact, which you navigate between using the top nav bar. Take a minute or two to examine different parts of the app (the About and Contact pages are very similar to the "Flask Web Project" and aren't discussed further).

Polls.

No polls available.

[Create Sample Polls](#)

© 2018 - Flask Polls

7. On the home page, the **Create Sample Polls** button initializes the app's data store with three different polls that are described in *models/samples.json* page. By default, the app uses an in-memory database (as shown on the About page), which is reset each time the app is restarted. The app also contains code to work with Azure Storage and Mongo DB, as described later in this article.
8. Once you've initialized the data store, you can vote in the different polls as shown on the home page (the nav bar and footer are omitted for brevity):

Polls.

[What is your favorite season?](#)

[How do you commute to work/school?](#)

[Have you tried Node.js Tools for Visual Studio \(<http://nodejstools.codeplex.com>\) ?](#)

9. Selecting a poll displays its specific choices:

What is your favorite season?

- Winter
- Spring
- Summer
- Fall

[Vote](#)

10. Once you vote, the app shows a results page and allows you to vote again:

What is your favorite season?



11. You can leave the app running for the sections that follow.

If you want to stop the app and [commit changes to source control](#), first open the **Changes** page in **Team Explorer**, right-click the folder for the virtual environment (probably **env**), and select **Ignore these local items**.

Examine the project contents

As noted before, much of what's in a project created from the "Polls Flask Web Project" template (and the "Polls Flask/Jade Web Project" template) should be familiar if you've explored the other project templates in Visual Studio. The additional steps in this article summarize the more significant changes and additions, namely data models and additional views.

Step 5-2: Understand the data models

The data models for the app are Python classes named `Poll` and `Choice`, which are defined in `models/__init__.py`. A `Poll` represents a question, for which a collection of `Choice` instances represent the available answers. A `Poll` also maintains the total number of votes (for any choice) and a method to calculate statistics that are used to generate views:

```

class Poll(object):
    """A poll object for use in the application views and repository."""
    def __init__(self, key=u'', text=u''):
        """Initializes the poll."""
        self.key = key
        self.text = text
        self.choices = []
        self.total_votes = None

    def calculate_stats(self):
        """Calculates some statistics for use in the application views."""
        total = 0
        for choice in self.choices:
            total += choice.votes
        for choice in self.choices:
            choice.votes_percentage = choice.votes / float(total) * 100 \
                if total > 0 else 0
        self.total_votes = total

class Choice(object):
    """A poll choice object for use in the application views and repository."""
    def __init__(self, key=u'', text=u'', votes=0):
        """Initializes the poll choice."""
        self.key = key
        self.text = text
        self.votes = votes
        self.votes_percentage = None

```

These data models are generic abstractions that allow the app's views to work against different types of backing data stores, which are described in the next step.

Step 5-3: Understand the backing data stores

The app created by the "Polls Flask Web Project" template can run against a data store in memory, in Azure table storage, or in a Mongo DB database.

The data storage mechanism works as follows:

1. The repository type is specified through the `REPOSITORY_NAME` environment variable, which can be set to "memory", "azuretablestore", or "mongodb". A bit of code in `settings.py` retrieves the name, using "memory" as the default. If you want to change the backing store, you have to set the environment variable and restart the app.

```

from os import environ
REPOSITORY_NAME = environ.get('REPOSITORY_NAME', 'memory')

```

2. The `settings.py` code then initializes a `REPOSITORY_SETTINGS` object. If you want to use Azure table store or Mondo DB, you must first initialize those data stores elsewhere, then set the necessary environment variables that tell the app how to connect to the store:

```

if REPOSITORY_NAME == 'azuretablestorage':
    REPOSITORY_SETTINGS = {
        'STORAGE_NAME': environ.get('STORAGE_NAME', ''),
        'STORAGE_KEY': environ.get('STORAGE_KEY', ''),
        'STORAGE_TABLE_POLL': environ.get('STORAGE_TABLE_POLL', 'polls'),
        'STORAGE_TABLE_CHOICE': environ.get('STORAGE_TABLE_CHOICE', 'choices'),
    }
elif REPOSITORY_NAME == 'mongodb':
    REPOSITORY_SETTINGS = {
        'MONGODB_HOST': environ.get('MONGODB_HOST', None),
        'MONGODB_DATABASE': environ.get('MONGODB_DATABASE', 'polls'),
        'MONGODB_COLLECTION': environ.get('MONGODB_COLLECTION', 'polls'),
    }
elif REPOSITORY_NAME == 'memory':
    REPOSITORY_SETTINGS = {}
else:
    raise ValueError('Unknown repository.')

```

3. In `views.py`, the app calls a factory method to initialize a `Repository` object using the data store's name and settings:

```

from FlaskPolls.models import PollNotFound
from FlaskPolls.models.factory import create_repository
from FlaskPolls.settings import REPOSITORY_NAME, REPOSITORY_SETTINGS

repository = create_repository(REPOSITORY_NAME, REPOSITORY_SETTINGS)

```

4. The `factory.create_repository` method is found in `models\factory.py`, which just imports the appropriate repository module, then creates a `Repository` instance:

```

def create_repository(name, settings):
    """Creates a repository from its name and settings. The settings
    is a dictionary where the keys are different for every type of repository.
    See each repository for details on the required settings."""
    if name == 'azuretablestorage':
        from .azuretablestorage import Repository
    elif name == 'mongodb':
        from .mongodb import Repository
    elif name == 'memory':
        from .memory import Repository
    else:
        raise ValueError('Unknown repository.')

    return Repository(settings)

```

5. The implementations of the `Repository` class that are specific to each data store can be found in `models\azuretablestorage.py`, `models\mongodb.py`, and `models\memory.py`. The Azure storage implementation uses the azure-storage package; the Mongo DB implementation uses the pymongo package. As noted in step 5-1, both packages are included in the project template's `requirements.txt` file. Exploring the details is left as an exercise for the reader.

In short, the `Repository` class abstracts the specifics of data store, and the app uses environment variables at run time to select and configure which of three implementations to use.

The following steps add support for a different data store than the three provided by the project template, if so desired:

1. Copy `memory.py` to a new file so that you have the basic interface for the `Repository` class.
2. Modify the implementation of the class as suits the data store you're using.

3. Modify `factory.py` to add another `elif` case that recognizes the name for your added data store and imports the appropriate module.
4. Modify `settings.py` to recognize another name in the `REPOSITORY_NAME` environment variable and to initialize `REPOSITORY_SETTINGS` accordingly.

Seed the data store from `samples.json`

Initially, any chosen data store contains no polls, so the app's home page shows the message **No polls available** along with the **Create Sample Polls** button. Once you select the button, however, the view changes to display available polls. This switch happens through conditional tags in `templates\index.html` (some blank lines omitted for brevity):

```
{% extends "layout.html" %}
{% block content %}
<h2>{{title}}.</h2>

{% if polls %}
<table class="table table-hover">
  <tbody>
    {% for poll in polls %}
      <tr>
        <td>
          <a href="/poll/{{poll.key}}">{{poll.text}}</a>
        </td>
      </tr>
    {% endfor %}
  </tbody>
</table>
{% else %}
<p>No polls available.</p>
<br />
<form action="/seed" method="post">
  <button class="btn btn-primary" type="submit">Create Sample Polls</button>
</form>
{% endif %}
{% endblock %}
```

The `polls` variable in the template comes from a call to `repository.get_polls`, which returns nothing until the data store is initialized.

Selecting the **Create Sample Polls** button navigates to the `/seed` URL. The handler for that route is defined in `views.py`:

```
@app.route('/seed', methods=['POST'])
def seed():
    """Seeds the database with sample polls."""
    repository.add_sample_polls()
    return redirect('/')
```

The call to `repository.add_sample_polls()` ends up in one of the specific `Repository` implementations for your chosen data store. Each implementation calls the `_load_samples_json` method found in `models_init_.py` to load the `models\samples.json` file into memory, then iterates through that data to create the necessary `Poll` and `Choice` objects in the data store.

Once that process is complete, the `redirect('/')` statement in the `seed` method navigates back to the home page. Because `repository.get_polls` now returns a data object, the conditional tags in `templates\index.html` now renders a table containing the polls.

Question: How does one add new polls to the app?

Answer: The app as provided through the project template doesn't include a facility for adding or editing polls. You can modify `models\samples.json` to create new initialization data, but doing would mean resetting the data store. To implement editing features, you need to extend the `Repository` class interface with methods to create the necessary `Choice` and `Poll` instances, then implement a UI in additional pages that use those methods.

Step 5-4: Understand the poll detail and results views

Most of the views generated by the "Polls Flask Web Project" and "Polls Flask/Jade Web Project" templates, such as the views for the About and Contact pages, are quite similar to views created by the "Flask Web Project" (or "Flask/Jade Web Project") template you worked with earlier in this tutorial. In the previous section you also learned how the home page is implemented to show either the initialization button or the list of polls.

What remains here is to examine the voting (details) and results view of an individual poll.

When you select a poll from the home page, the app navigates to the URL `/poll/<key>` where `key` is the unique identifier for a poll. In `views.py` you can see that the `details` function is assigned to handle that URL routing for both GET and requests. You can also see that using `<key>` in the URL route both maps any route of that form to the same function and generates an argument to the function of that same name:

```
@app.route('/poll/<key>', methods=['GET', 'POST'])
def details(key):
    """Renders the poll details page."""
    error_message = ''
    if request.method == 'POST':
        try:
            choice_key = request.form['choice']
            repository.increment_vote(key, choice_key)
            return redirect('/results/{0}'.format(key))
        except KeyError:
            error_message = 'Please make a selection.'

    return render_template(
        'details.html',
        title='Poll',
        year=datetime.now().year,
        poll=repository.get_poll(key),
        error_message=error_message,
    )
```

To show a poll (GET requests), this function simply calls upon `templates\details.html`, which iterates over the poll's `choices` array, creating a radio button for each.

```

{% extends "layout.html" %}

{% block content %}

<h2>{{poll.text}}</h2>
<br />

{% if error_message %}
<p class="text-danger">{{error_message}}</p>
{% endif %}

<form action="/poll/{{poll.key}}" method="post">
    {% for choice in poll.choices %}
        <div class="radio">
            <label>
                <input type="radio" name="choice" id="choice{{choice.key}}" value="{{choice.key}}"/>
                {{ choice.text }}
            </label>
        </div>
    {% endfor %}
    <br />
    <button class="btn btn-primary" type="submit">Vote</button>
</form>

{% endblock %}

```

Because the **Vote** button has `type="submit"`, selecting it generates a POST request back to the same URL that's routed to the `details` function once more. This time, however, it extracts the choice from the form data and redirects to `/results/<choice>`.

The `/results/<key>` URL is then routed to the `results` function in `views.py`, which then calls the poll's `calculate_stats` method and employs `templates\results.html` for the rendering:

```

@app.route('/results/<key>')
def results(key):
    """Renders the results page."""
    poll = repository.get_poll(key)
    poll.calculate_stats()
    return render_template(
        'results.html',
        title='Results',
        year=datetime.now().year,
        poll=poll,
    )

```

The `results.html` template, for its part, simply iterates through the poll's choices and generates a progress bar for each:

```

{% extends "layout.html" %}

{% block content %}

<h2>{{poll.text}}</h2>
<br />

{% for choice in poll.choices %}
<div class="row">
    <div class="col-sm-5">{{choice.text}}</div>
    <div class="col-sm-5">
        <div class="progress">
            <div class="progress-bar" role="progressbar" aria-valuenow="{{choice.votes}}" aria-valuemin="0"
aria-valuemax="{{poll.total_votes}}" style="width: {{choice.votes_percentage}}%;">{{choice.votes}}
```

Next steps

NOTE

If you've been committing your Visual Studio solution to source control throughout the course of this tutorial, now is a good time to do another commit. Your solution should match the tutorial source code on GitHub: [Microsoft/python-sample-vs-learning-flask](#).

You've now explored the entirety of the "Blank Flask Web Project", "Flask[/Jade] Web Project", and "Polls Flask[/Jade] Web Project" templates in Visual Studio. You've learned all the basics of Flask such as using views, templates, and routing, and have seen how to use backing data stores. You should now be able to get started on a web app of your own with any views and models that you need.

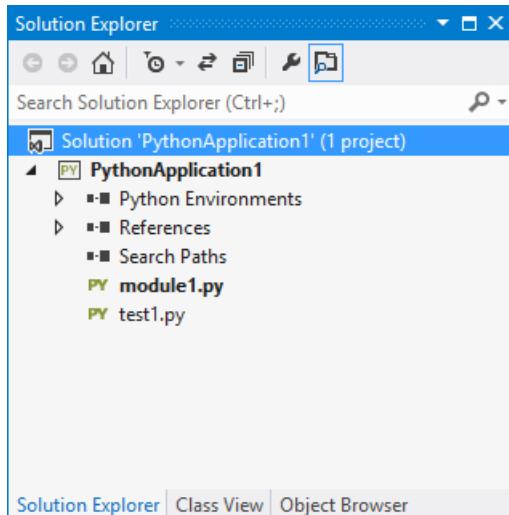
Running a web app on your development computer is just one step in making the app available to your customers. Next steps may include the following tasks:

- Deploy the web app to a production server, such as Azure App Service. See [Publish to Azure App Service](#).
- Add a repository implementation that uses another production-level data store such as PostgreSQL, MySQL, and SQL Server (all of which can be hosted on Azure). You can also use the [Azure SDK for Python](#) to work with Azure storage services like tables and blobs as well as Cosmos DB.
- Set up a continuous integration/continuous deployment pipeline on a service like Azure DevOps. In addition to working with source control (via Azure Repos or GitHub, or elsewhere), you can configure an Azure DevOps Project to automatically run your unit tests as a pre-requisite for release, and also configure the pipeline to deploy to a staging server for additional tests before deploying to production. Azure DevOps, furthermore, integrates with monitoring solutions like App Insights and closes the whole cycle with agile planning tools. For more information, see [Create a CI/CD pipeline for Python with Azure DevOps Projects](#) and also the general [Azure DevOps documentation](#).

Python projects in Visual Studio

4/26/2019 • 11 minutes to read • [Edit Online](#)

Python applications are typically defined using only folders and files, but this structure can become complex as applications become larger and perhaps involve auto-generated files, JavaScript for web applications, and so on. A Visual Studio project helps manage this complexity. The project (a `.pyproj` file) identifies all the source and content files associated with your project, contains build information for each file, maintains the information to integrate with source-control systems, and helps you organize your application into logical components.



In addition, projects are always managed within a Visual Studio *solution*, which can contain any number of projects that might reference one another. For example, a Python project can reference a C++ project that implements an extension module. With this relationship, Visual Studio automatically builds the C++ project (if necessary) when you start debugging the Python project. (For a general discussion, see [Solutions and projects in Visual Studio](#).)

Visual Studio provides a variety of Python project templates to quickly set up a number of application structures, including a template to create a project from an existing folder tree and a template to create a clean, empty project. See [Project templates](#) for an index.

TIP

Visual Studio 2019 supports opening a folder containing Python code and running that code without creating Visual Studio project and solution files. For more information, see [Quickstart: Open and run Python code in a folder](#). There are, however, benefits to using a project file, as explained in this section.

TIP

Without a project, all versions of Visual Studio work well with Python code. For example, you can open a Python file by itself and enjoy auto-complete, IntelliSense, and debugging (by right-clicking in the editor and selecting **Start with Debugging**). Because such code always uses the default global environment, however, you may see incorrect completions or errors if the code is meant for a different environment. Furthermore, Visual Studio analyzes all files and packages in the folder from which the single file is opened, which could consume considerable CPU time.

It's a simple matter to create a Visual Studio project from existing code, as described in [Create a project from existing files](#).



Deep Dive: Use source control with Python projects
(youtube.com, 8m 55s).

Add files, assign a startup file, and set environments

As you develop your application, you typically need to add new files of different types to the project. Adding such files is done by right-clicking the project and selecting **Add > Existing Item** with which you browse for a file to add, or **Add > New Item**, which brings up a dialog with a variety of item templates. As described on the [item templates](#) reference, options include empty Python files, a Python class, a unit test, and various files related to web applications. You can explore these options with a test project to learn what's available in your version of Visual Studio.

Each Python project has one assigned start-up file, shown in boldface in **Solution Explorer**. The start-up file is the file that's run when you start debugging (**F5** or **Debug > Start Debugging**) or when you run your project in the **Interactive** window (**Shift+Alt+F5** or **Debug > Execute Project in Python Interactive**). To change it, right-click the new file and select **Set as Startup Item** (or **Set as Startup File** in older versions of Visual Studio).

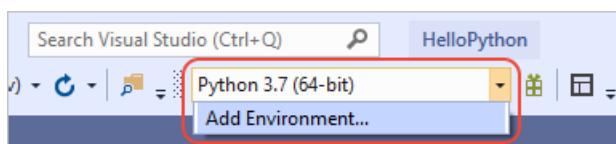
TIP

If you remove the selected startup file from a project and don't select a new one, Visual Studio doesn't know what Python file to start with when you try to run the project. In this case, Visual Studio 2017 version 15.6 and later shows an error; earlier versions either open an output window with the Python interpreter running, or you see the output window appear but then disappear almost immediately. If you encounter any of these behaviors, check that you have an assigned startup file.

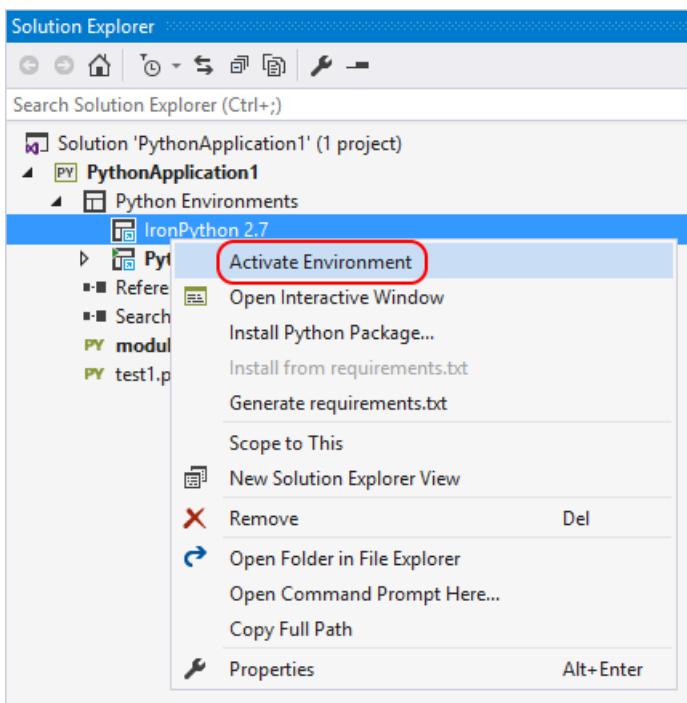
If you want to keep the output window open for any reason, right-click your project, select **Properties**, select the **Debug** tab, then add `-i` to the **Interpreter Arguments** field. This argument causes the interpreter to go into interactive mode after a program completes, thereby keeping the window open until you enter **Ctrl+Z > Enter** to exit.

A new project is always associated with the default global Python environment. To associate the project with a different environment (including virtual environments), right-click the **Python Environments** node in the project, select **Add/Remove Python Environments**, and select the ones you want.

A new project is always associated with the default global Python environment. To associate the project with a different environment (including virtual environments), right-click the **Python Environments** node in the project, select **Add Environment..**, and select the ones you want. You can also use the environments drop-down control on the toolbar to select an environment or add another one to the project.

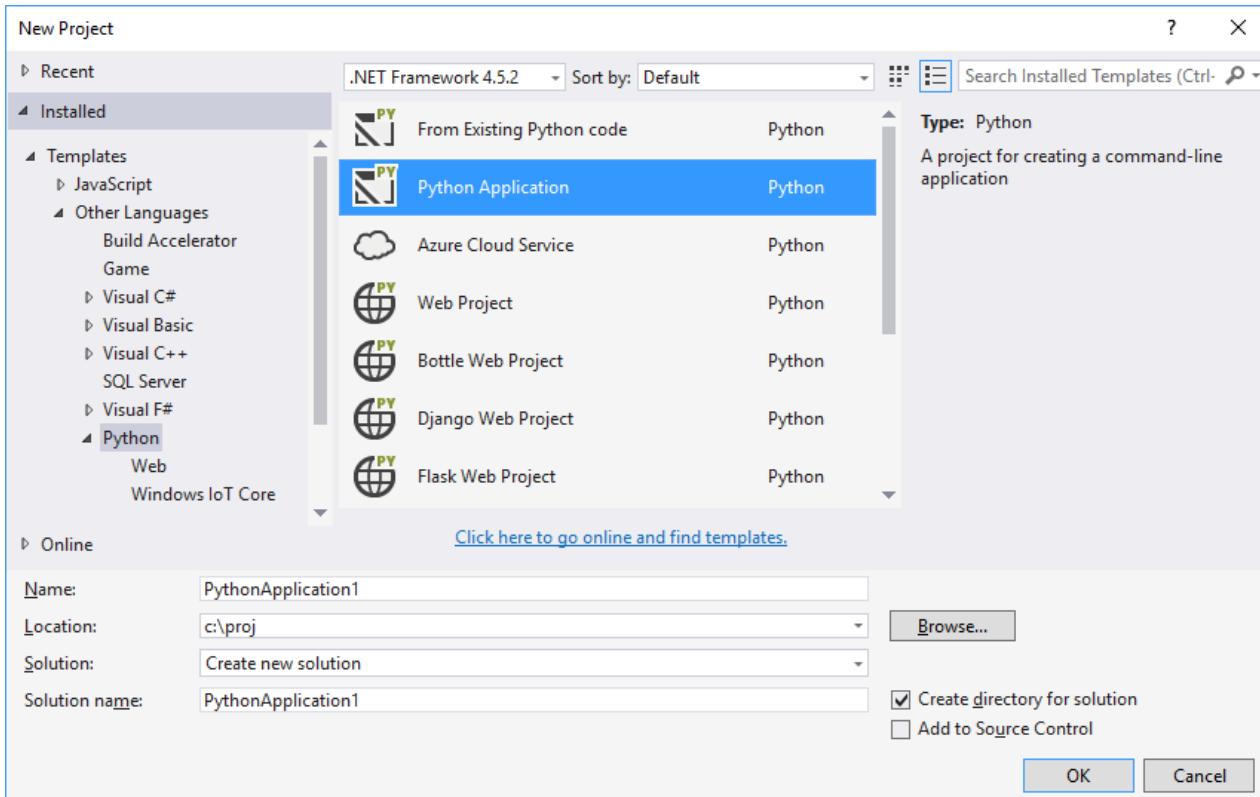


To change the active environment, right-click the desired environment in **Solution Explorer** and select **Activate Environment** as shown below. For more information, see [Select an environment for a project](#).



Project templates

Visual Studio gives you a number of ways to set up a Python project, either from scratch or from existing code. To use a template, select the **File > New > Project** menu command or right-click the solution in **Solution Explorer** and select **Add > New Project**, both of which bring up the **New Project** dialog below. To see Python-specific templates, either search on "Python" or select the **Installed > Python** node:



The following table summarizes the templates available in Visual Studio 2017 and later (not all templates are available in all previous versions):

TEMPLATE	DESCRIPTION
----------	-------------

TEMPLATE	DESCRIPTION
From existing Python code	Creates a Visual Studio project from existing Python code in a folder structure.
Python Application	A basic project structure for a new Python application with a single, empty source file. By default, the project runs in the console interpreter of the default global environment, which you can change by assigning a different environment .
Azure cloud service	A project for an Azure cloud service written in Python.
Web projects	Projects for web apps based on various frameworks including Bottle, Django, and Flask.
IronPython Application	Similar to the Python Application template, but uses IronPython by default enabling .NET interop and mixed-mode debugging with .NET languages.
IronPython WPF Application	A project structure using IronPython with Windows Presentation Foundation XAML files for the application's user interface. Visual Studio provides a XAML UI designer, code-behind can be written in Python, and the application runs without displaying a console.
IronPython Silverlight Web Page	An IronPython project that runs in a browser using Silverlight. The application's Python code is included in the web page as script. A boilerplate script tag pulls down some JavaScript code that initializes IronPython running inside of Silverlight, from which your Python code can interact with the DOM.
IronPython Windows Forms Application	A project structure using IronPython with UI created using code with Windows Forms. The application runs without displaying a console.
Background Application (IoT)	Supports deploying Python projects to run as background services on devices. Visit the Windows IoT Dev Center for more information.
Python Extension Module	This template appears under Visual C++ if you've installed the Python native development tools with the Python workload in Visual Studio 2017 or later (see Installation). It provides the core structure for a C++ extension DLL, similar to what's described on Create a C++ extension for Python .

NOTE

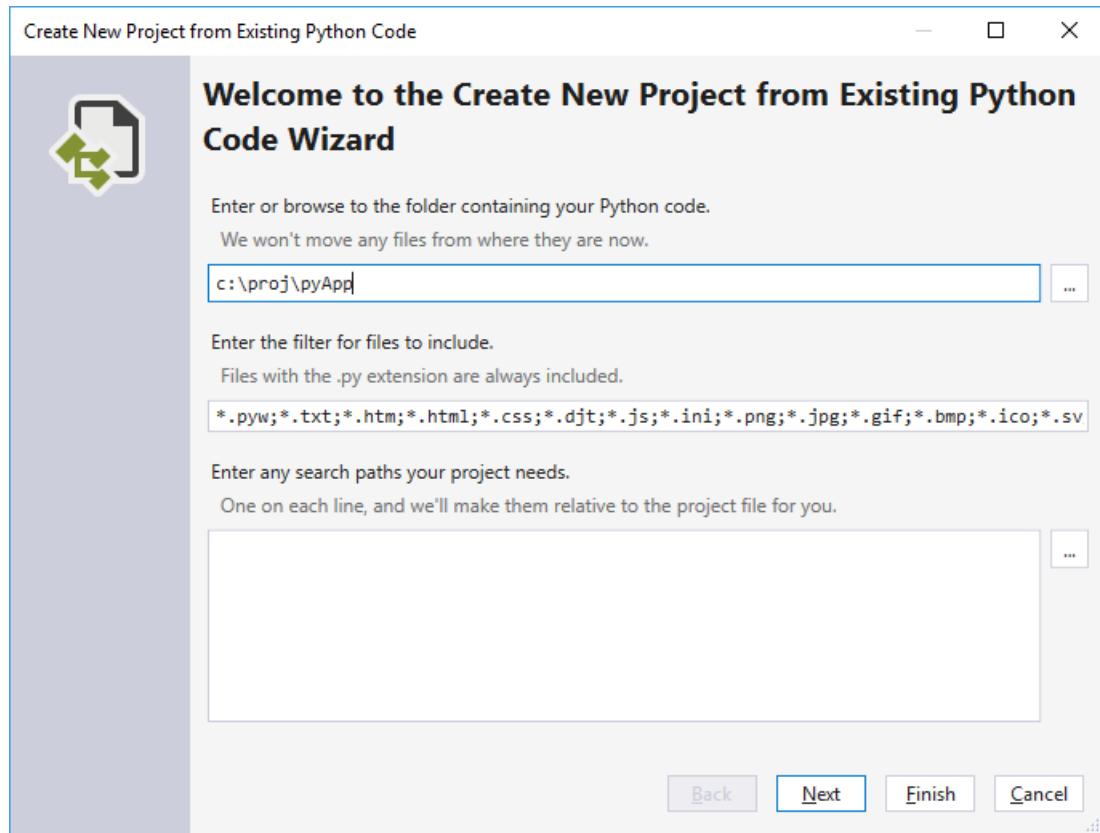
Because Python is an interpreted language, Python projects in Visual Studio don't produce a stand-alone executable like other compiled language projects (C#, for example). For more information, see [questions and answers](#).

Create a project from existing files

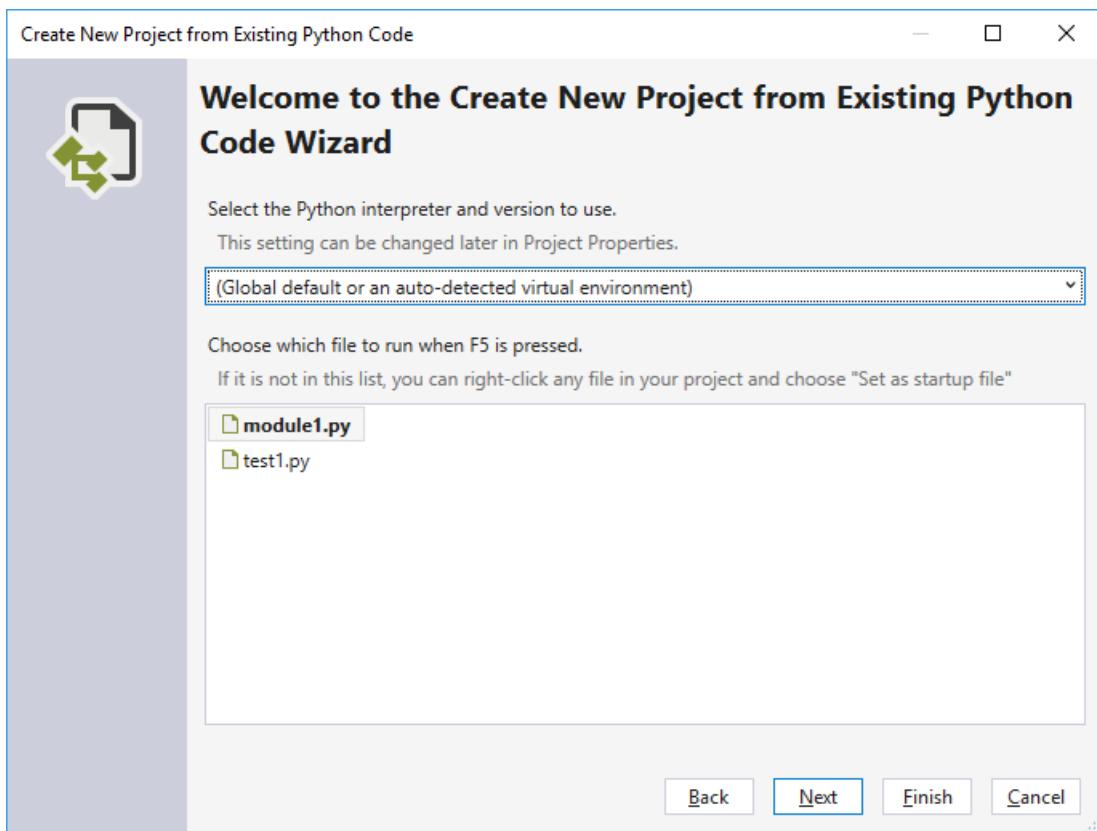
IMPORTANT

The process described here does not move or copy the original source files. If you want to work with a copy, duplicate the folder first.

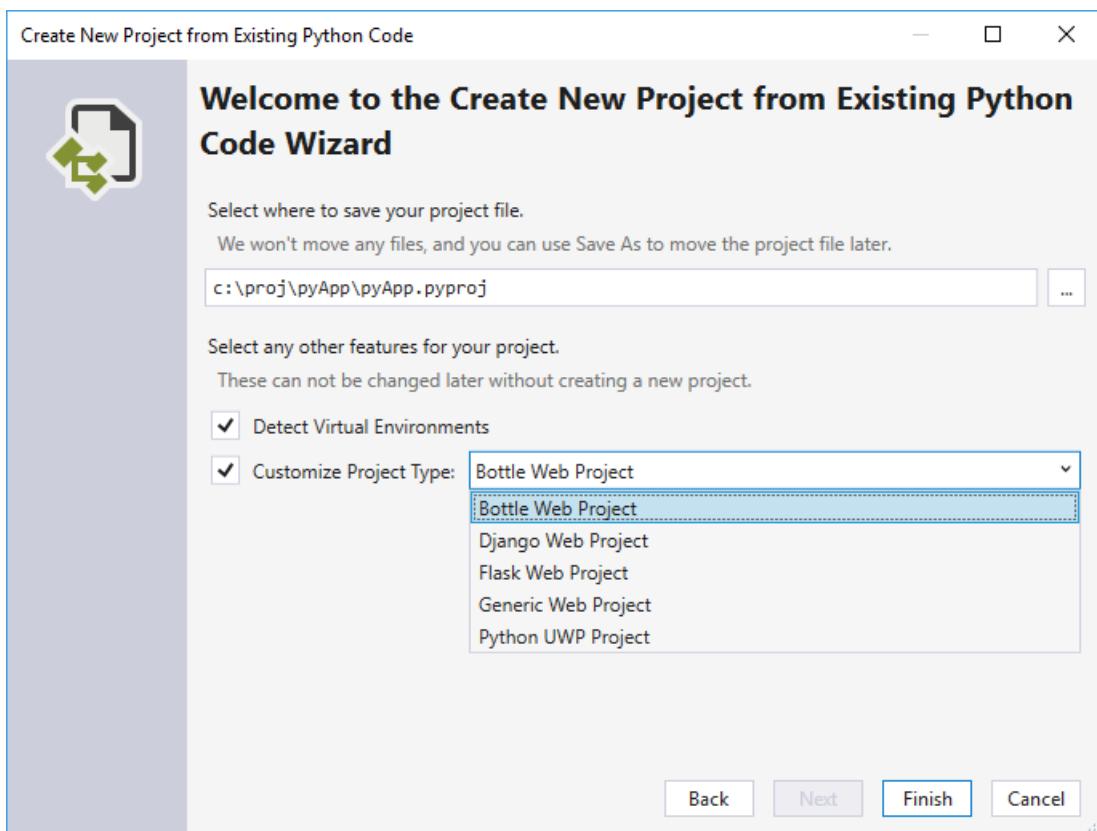
1. Launch Visual Studio and select **File > New > Project**.
2. In the **New Project** dialog, search for "Python", select the **From Existing Python code** template, give the project a name and location, and select **OK**.
3. In the wizard that appears, set the path to your existing code, set a filter for file types, and specify any search paths that your project requires, then select **Next**. If you don't know what search paths are, leave that field blank.



4. In the next dialog, select the startup file for your project and select **Next**. (If desired, select an environment; otherwise accept the defaults.) Note that the dialog shows only files in the root folder; if the file you want is in a subfolder, leave the startup file blank and set it later in **Solution Explorer** (described below).



5. Select the location in which to save the project file (which is a `.pyproj` file on disk). If applicable, you can also include auto-detection of virtual environments and customize the project for different web frameworks. If you're unsure of these options, leave them set to the defaults.



6. Select **Finish** and Visual Studio creates the project and opens it in **Solution Explorer**. If you want to move the `.pyproj` file elsewhere, select it in **Solution Explorer** and choose **File > Save As**. This action updates file references in the project but does not move any code files.
7. To set a different startup file, locate the file in **Solution Explorer**, right-click, and select **Set as Startup File**.

Linked files

Linked files are files that are brought into a project but typically reside outside of the application's project folders. They appear in **Solution Explorer** as normal files with an overlaid shortcut icon: 

Linked files are specified in the *.pyproj* file using the `<Compile Include="...">` element. Linked files are implicit if they use a relative path outside of the directory structure, or explicit if they use paths within **Solution Explorer**:

```
<Compile Include="..\test2.py">
  <Link>MyProject\test2.py</Link>
</Compile>
```

Linked files are ignored under any of the following conditions:

- The linked file contains Link metadata and the path specified in the Include attribute lives within the project directory
- The linked file duplicates a file that exists within the project hierarchy
- The linked file contains Link metadata and the Link path is a relative path outside of the project hierarchy
- The link path is rooted

Work with linked files

To add an existing item as a link, right-click the folder in the project where you wish to add the file, then select **Add** > **Existing Item**. In the dialog that appears, select a file and choose **Add as Link** from the drop-down on the **Add** button. Provided that there are no conflicting files, this command creates a link in the selected folder. However, the link is not added if there is already a file with the same name or a link to that file already exists in the project.

If you attempt to link to a file that already exists in the project folders, it is added as a normal file and not as a link. To convert a file into a link, select **File** > **Save As** to save the file to a location outside of the project hierarchy; Visual Studio automatically converts it into a link. Similarly, a link can be converted back by using **File** > **Save As** to save the file somewhere within the project hierarchy.

If you move a linked file in **Solution Explorer**, the link is moved but the actual file is unaffected. Similarly, deleting a link removes the link without affecting the file.

Linked files cannot be renamed.

References

Visual Studio projects support adding references to projects and extensions, which appear under the **References** node in **Solution Explorer**:



Extension references typically indicate dependencies between projects and are used to provide IntelliSense at design time or linking at compile time. Python projects use references in a similar fashion, but due to the dynamic nature of Python they are primarily used at design time to provide improved IntelliSense. They can also be used for deployment to Microsoft Azure to install additional dependencies.

Extension modules

A reference to a *.pyd* file enables IntelliSense for the generated module. Visual Studio loads the *.pyd* file into the Python interpreter and introspects its types and functions. It also attempts to parse the doc strings for functions to

provide signature help.

If at any time the extension module is updated on disk, Visual Studio reanalyzes the module in the background. This action has no effect on runtime behavior but some completions aren't available until analysis is complete.

You may also need to add a [search path](#) to the folder containing the module.

.NET projects

When working with IronPython, you can add references to .NET assemblies to enable IntelliSense. For .NET projects in your solution, right-click the **References** node in your Python project, select **Add Reference**, select the **Projects** tab, and browse to the desired project. For DLLs that you've downloaded separately, select the **Browse** tab instead and browse to the desired DLL.

Because references in IronPython are not available until a call to `clr.AddReference('<AssemblyName>')` is made, you also need to add an appropriate `clr.AddReference` call to the assembly, typically at the beginning of your code. For example, the code created by the **IronPython Windows Forms Application** project template in Visual Studio includes two calls at the top of the file:

```
import clr
clr.AddReference('System.Drawing')
clr.AddReference('System.Windows.Forms')

from System.Drawing import *
from System.Windows.Forms import *

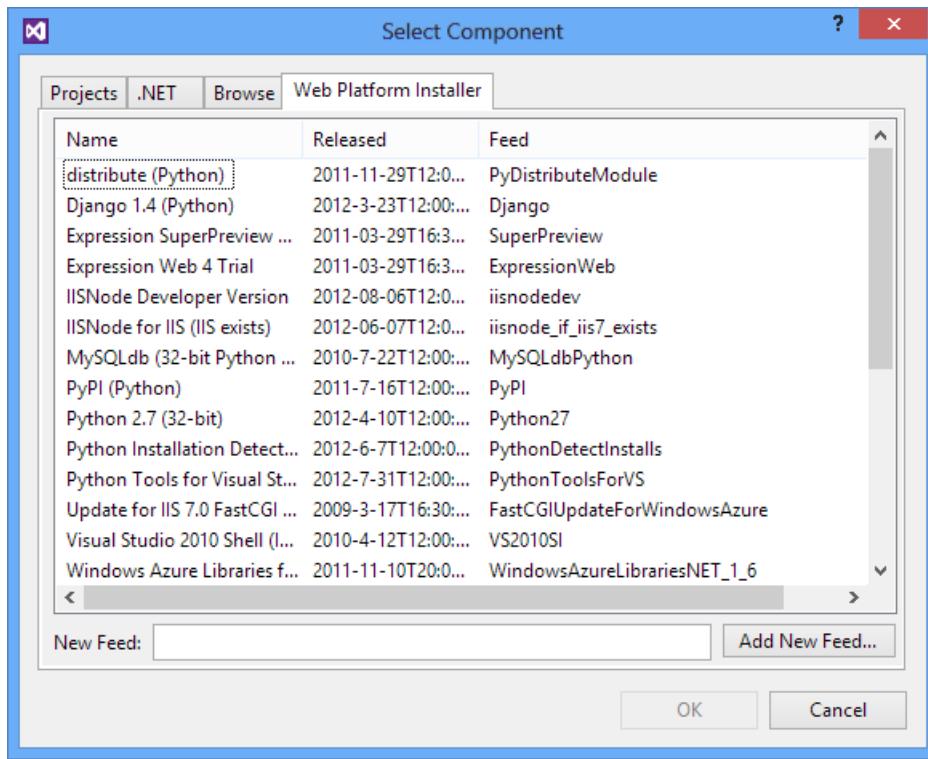
# Other code omitted
```

WebPI projects

You can add references to WebPI product entries for deployment to Microsoft Azure Cloud Services where you can install additional components via the WebPI feed. By default, the feed displayed is Python-specific and includes Django, CPython, and other core components. You can also select your own feed as shown below. When publishing to Microsoft Azure, a setup task installs all of the referenced products.

IMPORTANT

WebPI projects is not available in Visual Studio 2017 or Visual Studio 2019.



Python web application project templates

4/9/2019 • 8 minutes to read • [Edit Online](#)

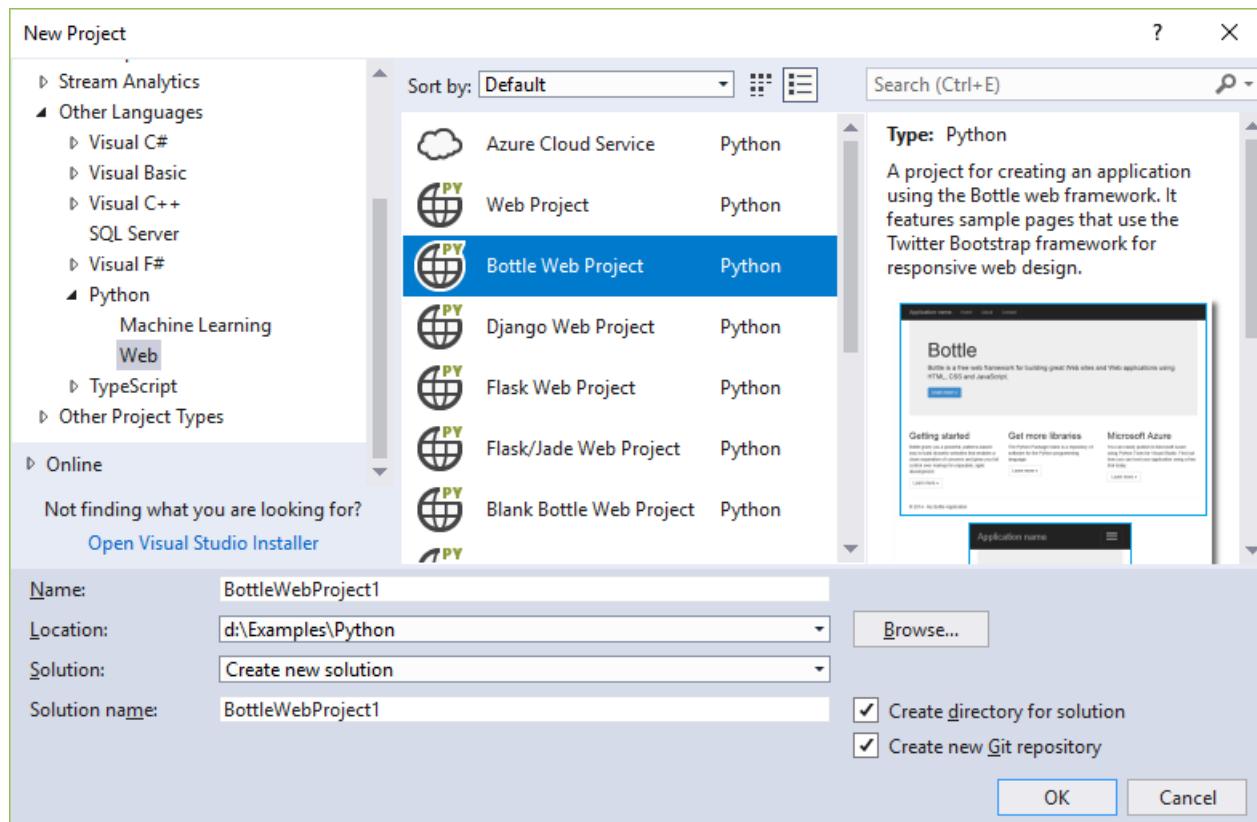
Python in Visual Studio supports developing web projects in Bottle, Flask, and Django frameworks through project templates and a debug launcher that can be configured to handle various frameworks. These templates include a `requirements.txt` file to declare the necessary dependencies. When creating a project from one of these templates, Visual Studio prompts you to install those packages (see [Install project requirements](#) later in this article).

You can also use the generic **Web Project** template for other frameworks such as Pyramid. In this case, no frameworks are installed with the template. Instead, install the necessary packages into the environment you're using for the project (see [Python environments window - Package tab](#)).

For information on deploying a Python web app to Azure, see [Publish to Azure App Service](#).

Use a project template

You create a project from a template using **File > New > Project**. To see templates for web projects, select **Python > Web** on the left side of the dialog box. Then select a template of your choice, providing names for the project and solution, set options for a solution directory and Git repository, and select **OK**.



The generic **Web Project** template, mentioned earlier, provides only an empty Visual Studio project with no code and no assumptions other than being a Python project. For details on the **Azure Cloud Service** template, see [Azure cloud service projects for Python](#).

All the other templates are based on the Bottle, Flask, or Django web frameworks, and fall into three general groups as described in the following sections. The apps created by any of these templates contain sufficient code to run and debug the app locally. Each one also provides the necessary [WSGI app object](#) (`python.org`) for use with production web servers.

Blank group

All **Blank <framework> Web Project** templates create a project with more or less minimal boilerplate code and the necessary dependencies declared in a *requirements.txt* file.

TEMPLATE	DESCRIPTION
Blank Bottle Web Project	Generates a minimal app in <i>app.py</i> with a home page for <code>/</code> and a <code>/hello/<name></code> page that echoes <code><name></code> using a very short inline page template.
Blank Django Web Project	Generates a Django project with the core Django site structure but no Django apps. For more information, see Django templates and Learn Django Step 1 .
Blank Flask Web Project	Generates a minimal app with a single "Hello World!" page for <code>/</code> . This app is similar to the result of following the detailed steps in Quickstart: Use Visual Studio to create your first Python web app . Also see Learn Flask Step 1 .

Web group

All **<Framework> Web Project** templates create a starter web app with an identical design regardless of the chosen framework. The app has Home, About, and Contact pages, along with a nav bar and responsive design using Bootstrap. Each app is appropriately configured to serve static files (CSS, JavaScript, and fonts), and uses a page template mechanism appropriate for the framework.

TEMPLATE	DESCRIPTION
Bottle Web Project	Generates an app whose static files are contained in the <i>static</i> folder and handled through code in <i>app.py</i> . Routing for the individual pages is contained in <i>routes.py</i> , and the <i>views</i> folder contains the page templates.
Django Web Project	Generates a Django project and a Django app with three pages, authentication support, and a SQLite database (but no data models). For more information, see Django templates and Learn Django Step 4 .
Flask Web Project	Generates an app whose static files are contained in the <i>static</i> folder. Code in <i>views.py</i> handles routing, with page templates using the Jinja engine contained in the <i>templates</i> folder. The <i>runserver.py</i> file provides startup code. See Learn Flask Step 4 .
Flask/Jade Web Project	Generates the same app as with the Flask Web Project template but using the Jade extension for the Jinja templating engine.

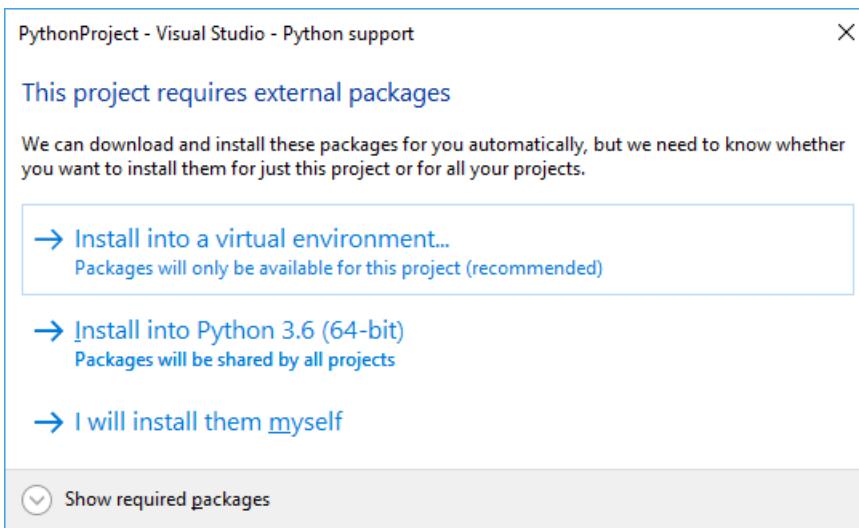
Polls group

The **Polls <framework> Web Project** templates create a starter web app through which users can vote on different poll questions. Each app builds upon the structure of the **Web** project templates to use a database to manage the polls and user responses. The apps include appropriate data models and a special app page (`/seed`) that loads polls from a *samples.json* file.

TEMPLATE	DESCRIPTION
Polls Bottle Web Project	Generates an app that can run against an in-memory database, MongoDB, or Azure Table Storage, which is configured using the <code>REPOSITORY_NAME</code> environment variable. The data models and data store code are contained in the <i>models</i> folder, and the <i>settings.py</i> file contains code to determine which data store is used.
Polls Django Web Project	Generates a Django project and a Django app with three pages and a SQLite database. Includes customizations to the Django administrative interface to allow an authenticated administrator to create and manage polls. For more information, see Django templates and Learn Django Step 6 .
Polls Flask Web Project	Generates an app that can run against an in-memory database, MongoDB, or Azure Table Storage, which is configured using the <code>REPOSITORY_NAME</code> environment variable. The data models and data store code are contained in the <i>models</i> folder, and the <i>settings.py</i> file contains code to determine which data store is used. The app uses the Jinja engine for page templates. See Learn Flask Step 5 .
Polls Flask/Jade Web Project	Generates the same app as with the Polls Flask Web Project template but using the Jade extension for the Jinja templating engine.

Install project requirements

When creating a project from a framework-specific template, a dialog appears to help you install the necessary packages using pip. We also recommend using a [virtual environment](#) for web projects so that the correct dependencies are included when you publish your web site:



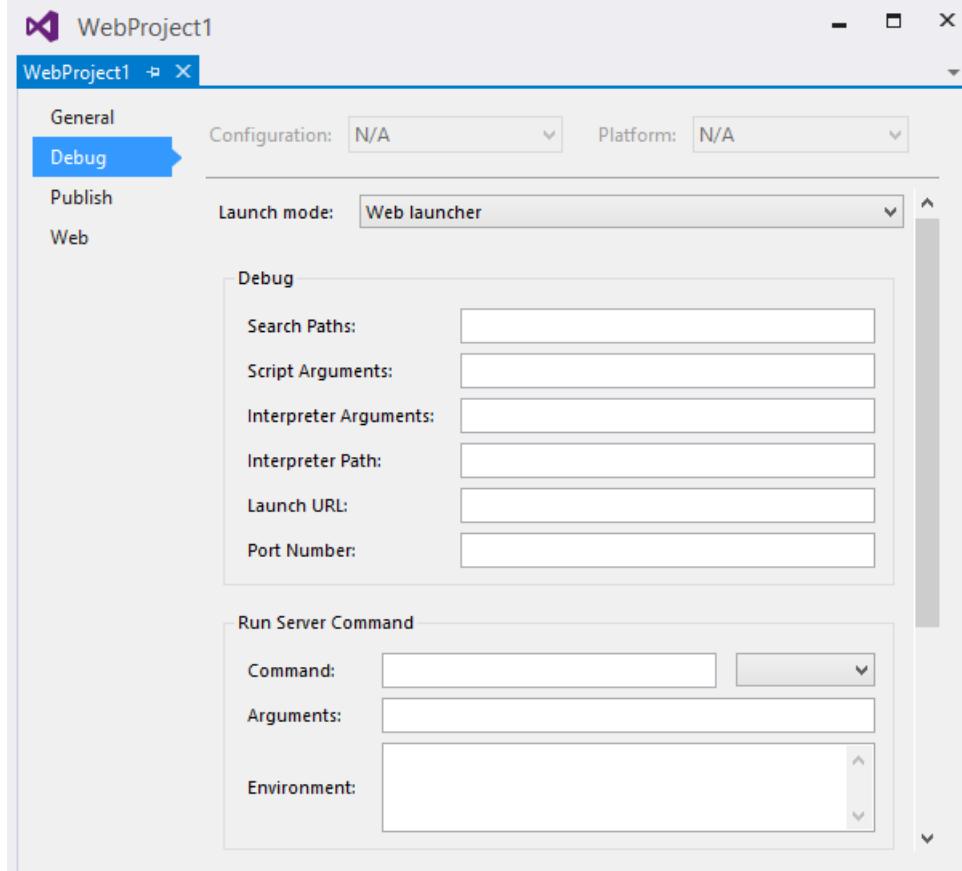
If you're using source control, you typically omit the virtual environment folder as that environment can be recreated using only *requirements.txt*. The best way to exclude the folder is to first select the **I will install them myself** in the prompt shown above, then disable auto-commit before creating the virtual environment. For details, see [Learn Django Tutorial - Steps 1-2 and 1-3](#) and [Learn Flask Tutorial - Steps 1-2 and 1-3](#).

When deploying to Microsoft Azure App Service, select a version of Python as a [site extension](#) and manually install packages. Also, because Azure App Service does **not** automatically install packages from a *requirements.txt* file when deployed from Visual Studio, follow the configuration details on [aka.ms/PythonOnAppService](#).

Microsoft Azure Cloud Services *does* support the `requirements.txt` file. See [Azure cloud service projects](#) for details.

Debugging

When a web project is started for debugging, Visual Studio starts a local web server on a random port and opens your default browser to that address and port. To specify additional options, right-click the project, select **Properties**, and select the **Web Launcher** tab:



In the **Debug** group:

- **Search Paths, Script Arguments, Interpreter Arguments, and Interpreter Path:** these options are the same as for [normal debugging](#).
- **Launch URL:** specifies the URL that is opened in your browser. It defaults to `localhost`.
- **Port Number:** the port to use if none is specified in the URL (Visual Studio selects one automatically by default). This setting allows you to override the default value of the `SERVER_PORT` environment variable, which is used by the templates to configure the port the local debug server listens on.

The properties in the **Run Server Command** and **Debug Server Command** groups (the latter is below what's shown in the image) determine how the web server is launched. Because many frameworks require the use of a script outside of the current project, the script can be configured here and the name of the startup module can be passed as a parameter.

- **Command:** can be a Python script (`*.py` file), a module name (as in, `python.exe -m module_name`), or a single line of code (as in, `python.exe -c "code"`). The value in the drop-down indicates which of these types is intended.
- **Arguments:** these arguments are passed on the command line following the command.
- **Environment:** a newline-separated list of `<NAME>=<VALUE>` pairs specifying environment variables. These variables are set after all properties that may modify the environment, such as the port number and search paths, and so may overwrite these values.

Any project property or environment variable can be specified with MSBuild syntax, for example:

`$(StartupFile) --port $(SERVER_PORT)`. `$(StartupFile)` is the relative path to the startup file and `{StartupModule}` is the importable name of the startup file. `$(SERVER_HOST)` and `$(SERVER_PORT)` are normal environment variables that are set by the **Launch URL** and **Port Number** properties, automatically, or by the **Environment** property.

NOTE

Values in **Run Server Command** are used with the **Debug > Start Server** command or **Ctrl+F5**; values in the **Debug Server Command** group are used with the **Debug > Start Debug Server** command or **F5**.

Sample Bottle configuration

The **Bottle Web Project** template includes boilerplate code that does the necessary configuration. An imported bottle app may not include this code, however, in which case the following settings launch the app using the installed `bottle` module:

- **Run Server Command** group:

- **Command:** `bottle` (module)
- **Arguments:** `--bind=%SERVER_HOST%:%SERVER_PORT% {StartupModule}:app`

- **Debug Server Command** group:

- **Command:** `bottle` (module)
- **Arguments:** `--debug --bind=%SERVER_HOST%:%SERVER_PORT% {StartupModule}:app`

The `--reload` option is not recommended when using Visual Studio for debugging.

Sample Pyramid configuration

Pyramid apps are currently best created using the `pcreate` command-line tool. Once an app has been created, it can be imported using the **From existing Python code** template. After doing so, select the **Generic Web Project** customization to configure the options. These settings assume that Pyramid is installed into a virtual environment at `..\env`.

- **Debug** group:

- **Server Port:** 6543 (or whatever is configured in the `.ini` files)

- **Run Server Command** group:

- **Command:** `..\env\scripts\pserv-script.py` (script)
- **Arguments:** `Production.ini`

- **Debug Server Command** group:

- **Command:** `..\env\scripts\pserv-script.py` (script)
- **Arguments:** `Development.ini`

TIP

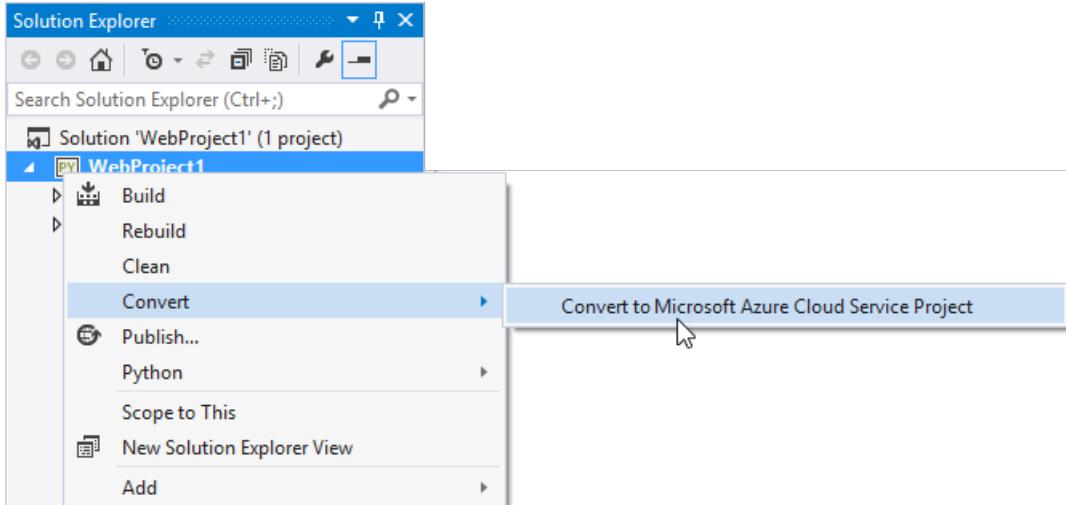
You likely need to configure the **Working Directory** property of your project because Pyramid apps are typically one folder below the project root.

Other configurations

If you have settings for another framework that you would like to share, or if you'd like to request settings for another framework, open an [issue on GitHub](#).

Convert a project to Azure Cloud Service

The **Convert to Microsoft Azure Cloud Service Project** command (image below) adds a cloud service project to your solution. This project includes the deployment settings and configuration for the virtual machines and services to be used. Use the **Publish** command on the cloud project to deploy to Cloud Services; the **Publish** command on the Python project still deploys to Web Sites. For more information, see [Azure cloud service projects](#).



See also

- [Python item templates reference](#)
- [Publish to Azure App Service](#)

Django web project template

4/9/2019 • 2 minutes to read • [Edit Online](#)

Django is a high-level Python framework designed for rapid, secure, and scalable web development. Python support in Visual Studio provides several project templates to set up the structure of a Django-based web application. To use a template in Visual Studio, select **File > New > Project**, search for "Django", and select from the **Blank Django Web Project**, **Django Web Project**, and **Polls Django Web Project** templates. See the [Learn Django tutorial](#) for a walkthrough of all the templates.

Visual Studio provides full IntelliSense for Django projects:

- Context variables passed into the template:

```
<form action="{% url 'app:vote' poll.id %}" method="post">
  {% csrf_token %}
  {% for choice in | %}
    <div class="r choice>
      <label>
        <input type="radio" value="choice" id="choice{{ forloop.counter }}"
        {{ ch poll title year }}</label>
    </div>
  {% endfor %}
  <br />
  <button class="btn btn-primary" type="submit">Vote</button>
</form>
```

The screenshot shows a Django template snippet within a code editor. A tooltip is displayed over the variable 'poll', which is highlighted in blue. The tooltip lists several context variables: choice, error_message, poll, title, and year. The 'poll' entry is currently selected.

- Tagging and filtering for both built-ins and user-defined:

```
{% if error_message %}
<p class="text-danger">
  {{error_message|}}</p>
{% endif %}
```

The screenshot shows a Django template snippet within a code editor. A tooltip is displayed over the filter '|'. The tooltip lists several built-in filters: timesince, timeuntil, title, truncatechars, truncatewords, truncatewords_html, unlocalize, unordered_list, and upper. The 'upper' entry is currently selected.

- Syntax coloring for embedded CSS and JavaScript:

A screenshot of the Visual Studio code editor showing a CSS file. A tooltip is open over a color value in a style block:

```
<style>
    body {
        font-family: 'Segoe UI', Tahoma, Geneva, Verdana, sans-serif;
        background-color: #fff;
    }
</style>
```

The tooltip shows a color palette with several color names listed. The color `#fff` is highlighted in blue, indicating it is selected.

- ActiveBorder
- ActiveCaption
- aliceblue
- antiquewhite
- AppWorkspace
- aqua
- aquamarine
- **azure**
- Background

A screenshot of the Visual Studio code editor showing a JavaScript file. A tooltip is open over a variable name in a script block:

```
<script type="text/javascript" language="javascript">
    var text = 'hello';
    alert(tex
</sc>alert([String message])
```

The tooltip shows a list of available variables. The variable `text` is highlighted in blue, indicating it is selected.

- SVGTextContentElement
- SVGTextElement
- SVGTextPathElement
- SVGTextPositioningElement
- **text** (global variable)
- Text
- TextEvent
- TextMetrics
- TextRange

Visual Studio also provides full [debugging support](#) for Django projects:

The screenshot shows the Visual Studio interface with several windows open:

- index.html**: The main code editor window displaying a Django template with syntax highlighting for HTML and Python.
- Watch 1**: A table showing variables and their values, including `poll.text` (unicode), `poll.id` (int), `poll.pub_date` (datetime.datetime), and `day` (int).
- Call Stack**: A list of function calls and their stack frames, showing the call chain from the current line 14 back to the base template line 840.
- Status Bar**: Shows the zoom level (100%), the current selection (<table.table table-hover> <tbody> <tr> <td>), and the tabs for Watch 1, Autos, Locals, Call Stack, Breakpoints, Immediate Window, and Output.

It's typical for Django projects to be managed through their `manage.py` file, which is an assumption that Visual Studio follows. If you stop using that file as the entry point, you essentially break the project file. In that case you need to [recreate the project from existing files](#) without marking it as a Django project.

Django management console

The Django management console is accessed through various commands on the **Project** menu or by right-clicking the project in **Solution Explorer**.

- **Open Django Shell**: opens a shell in your application context that enables you to manipulate your models:

The screenshot shows the Django Management Console window with the title "Django Management Console - DjangoWebProject1". It contains an interactive Python shell session:

```

Django Management Console - DjangoWebProject1
☰ _main_
>>> from app.models import Poll, Choice
>>> from django.utils import timezone
>>> p = Poll(text="Favorite way to host Django on Azure?", pub_date=timezone.now())
>>> p.save()
>>> p.choice_set.create(text='Cloud Service', votes=0)
<Choice: Cloud Service>
>>> p
<Poll: Favorite way to host Django on Azure?>
>>> |

```

- **Django Sync DB**: executes `manage.py syncdb` in an **Interactive** window:

```
Django Management Console - DjangoWebProject1
X ɔ ■ _main_
Executing manage.py syncdb
Creating tables ...
Creating table auth_permission
Creating table auth_group_permissions
Creating table auth_group
Creating table auth_user_groups
Creating table auth_user_user_permissions
Creating table auth_user
Creating table django_content_type
Creating table django_session
Creating table django_site

You just installed Django's auth system, which means you don't have any
superusers defined.
Would you like to create one now? (yes/no):
```

- **Collect Static:** executes `manage.py collectstatic --noinput` to copy all the static files to the path specified by `STATIC_ROOT` in your `settings.py`.

```
Django Management Console - DjangoWebProject1
X ɔ ■ <disconnected>
Executing manage.py collectstatic --noinput

0 static files copied to 'c:\projects\temp2\DjangoWebProject1\static', 91
unmodified.
The Python REPL process has exited
>>> |
```

- **Validate:** executes `manage.py validate`, which reports any validation errors in the installed models specified by `INSTALLED_APPS` in your `settings.py`:

```
Django Management Console - DjangoWebProject1
X ɔ ■ <disconnected>
Executing manage.py validate
0 errors found
The Python REPL process has exited
>>>
```

See also

- [Learn Django tutorial](#)
- [Publish to Azure App Service](#)

Azure cloud service projects for Python

4/9/2019 • 5 minutes to read • [Edit Online](#)

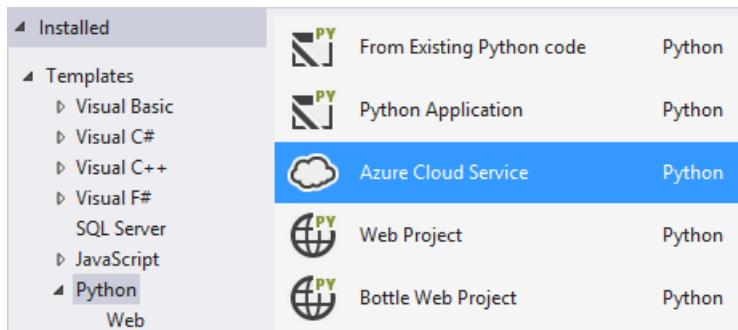
Visual Studio provides templates to help you get started creating Azure Cloud Services using Python.

A [cloud service](#) consists of any number of *worker roles* and *web roles*, each of which performs a conceptually separate task but can be separately replicated across virtual machines as needed for scaling. Web roles provide hosting for front-end web applications. Where Python is concerned, any web framework that supports WSGI can be used to write such an application (as supported by the [Web project template](#)). Worker roles are intended for long-running processes that do not interact directly with users. They typically make use of the packages within the "azure" package, which is installed with `pip install azure`.

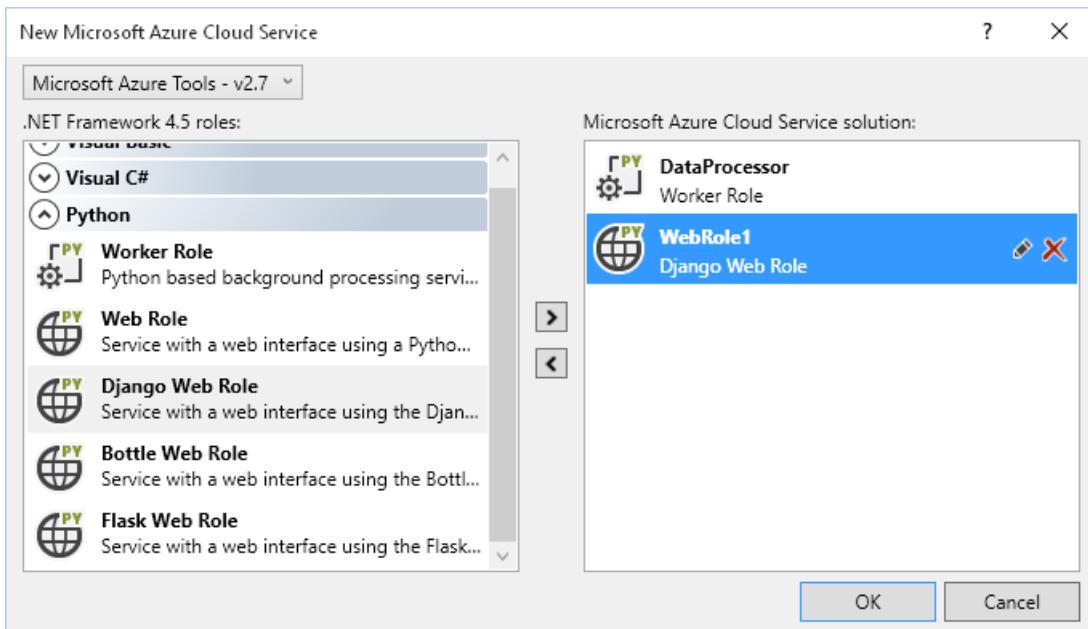
This article contains details about the project template and other support in Visual Studio 2017 and later (earlier versions are similar, but with some differences). For more about working with Azure from Python, visit the [Azure Python Developer Center](#).

Create a project

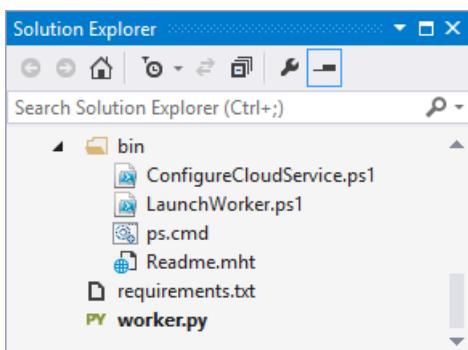
1. Install the [Azure .NET SDK for Visual Studio](#), which is required to use the cloud service template.
2. In Visual Studio, select **File > New > Project**, then search for "Azure Python" and select **Azure Cloud Service** from the list:



3. Select one or more roles to include. Cloud projects may combine roles written in different languages, so you can easily write each part of your application in the most suitable language. To add new roles to the project after completing this dialog, right-click **Roles** in **Solution Explorer** and select one of the items under **Add**.



4. As the individual role projects are created, you may be prompted to install additional Python packages, such as the Django, Bottle, or Flask frameworks if you selected a role that uses one of those.
5. After adding a new role to your project, configuration instructions appear. Configuration changes are usually unnecessary, but may be useful for future customization of your projects. Note that when adding multiple roles at the same time, only the instructions for the last role remain open. However, you can find the instructions and troubleshooting tips for the other roles in their respective *readme.mht* files, located either in the role's root or in the *bin* folder.
6. A project's *bin* folder also contains one or two PowerShell scripts that are used to configure the remote virtual machine, including installing Python, any *requirements.txt* file in your project, and setting up IIS if necessary. You may edit these files as desired to your deployment, though most common options can be managed in other ways (see [Configure role deployment](#) below). We do not suggest removing these files, as a legacy configuration script is used instead if the files are not available.



To add these configuration scripts to a new project, right-click the project, select **Add > New Item**, and select either **Web Role Support Files** or **Worker Role Support Files**.

Configure role deployment

The PowerShell scripts in a role project's *bin* folder control the deployment of that role and may be edited to customize the configuration:

- *ConfigureCloudService.ps1* is used for web and worker roles, typically to install and configure dependencies and set the Python version.
- *LaunchWorker.ps1* is used only for worker roles and is used to change startup behavior, add command-line arguments, or add environment variables.

Both files contain instructions for customization. You can also install your own version of Python by adding another task to the main cloud service project's *ServiceDefinition.csdef* file, setting the `PYTHON` variable to its installed *python.exe* (or equivalent) path. When `PYTHON` is set, Python is not installed from NuGet.

Additional configuration can be accomplished as follows:

1. Install packages using `pip` by updating the *requirements.txt* file in the root directory of your project. The *ConfigureCloudService.ps1* script installs this file on deployment.
2. Set environment variables by modifying your *web.config* file (web roles) or the `Runtime` section of your *ServiceDefinition.csdef* file (worker roles).
3. Specify the script and arguments to use for a worker role by modifying the command line in the `Runtime/EntryPoint` section of your *ServiceDefinitions.csdef* file.
4. Set the main handler script for a web role through the *web.config* file.

Test role deployment

While writing your roles, you can test your cloud project locally using the Cloud Service Emulator. The emulator is included with the Azure SDK Tools and is a limited version of the environment used when your cloud service is published to Azure.

To start the emulator, first ensure your cloud project is the startup project in your solution by right-clicking and selecting **Set as startup project**. Then select **Debug > Start Debugging (F5)** or **Debug > Start without Debugging (Ctrl+F5)**.

Note that due to limitations in the emulator it is not possible to debug your Python code. We thus recommend you debug roles by running them independently, and then use the emulator for integration testing before publishing.

Deploy a role

To open the **Publish** wizard, select the role project in **Solution Explorer** and select **Build > Publish** from the main menu, or right-click the project and select **Publish**.

The publishing process involves two phases. First, Visual Studio creates a single package containing all the roles for your cloud service. This package is what's deployed to Azure, which initializes one or more virtual machines for each role and deploys the source.

As each virtual machine activates, it executes the *ConfigureCloudService.ps1* script and installs any dependencies. This script by default installs a recent version of Python from [NuGet](#) and any packages specified in a *requirements.txt* file.

Finally, worker roles execute *LaunchWorker.ps1*, which starts running your Python script; web roles initialize IIS and begin handling web requests.

Dependencies

For Cloud Services, the *ConfigureCloudService.ps1* script uses `pip` to install a set of Python dependencies. Dependencies should be specified in a file named *requirements.txt* (customizable by modifying *ConfigureCloudService.ps1*). The file is executed with `pip install -r requirements.txt` as part of initialization.

Note that cloud service instances do not include C compilers, so all libraries with C extensions must provide pre-compiled binaries.

`pip` and its dependencies, as well as the packages in *requirements.txt*, are downloaded automatically and may count as chargeable bandwidth usage. See [Manage required packages](#) for details on managing *requirements.txt* files.

Troubleshooting

If your web or worker role does not behave correctly after deployment, check the following:

- Your Python project includes a `bin\` folder with (at least):
 - `ConfigureCloudService.ps1`
 - `LaunchWorker.ps1` (for worker roles)
 - `ps.cmd`
- Your Python project includes a `requirements.txt` file listing all dependencies (or alternately, a collection of wheel files).
- Enable Remote Desktop on your cloud service and investigate the log files.
- Logs for `ConfigureCloudService.ps1` and `LaunchWorker.ps1` are stored in `C:\Resources\Directory%RoleId%\DiagnosticStore\LogFiles` folder on the remote computer.
- Web roles may write additional logs to a path configured in `web.config`, namely the path in the `WSGI_LOG` appSetting. Most regular IIS or FastCGI logging also works.
- Currently, the `LaunchWorker.ps1.log` file is the only way to view output or errors displayed by your Python worker role.

How to create and manage Python environments in Visual Studio

4/26/2019 • 12 minutes to read • [Edit Online](#)

A Python *environment* is a context in which you run Python code and includes global, virtual, and conda environments. An environment consists of an interpreter, a library (typically the Python Standard Library), and a set of installed packages. These components together determine which language constructs and syntax are valid, what operating-system functionality you can access, and which packages you can use.

In Visual Studio on Windows, you use the **Python Environments** window, as described in this article, to manage environments and select one as the default for new projects. Other aspects of environments are found in the following articles:

- For any given project, you can [select a specific environment](#) rather than use the default.
- For details on creating and using virtual environments for Python projects, see [Use virtual environments](#).
- If you want to install packages in an environment, refer to the [Packages tab reference](#).
- To install another Python interpreter, see [Install Python interpreters](#). In general, if you download and run an installer for a mainline Python distribution, Visual Studio detects that new installation and the environment appears in the **Python Environments** window and can be selected for projects.

If you're new to Python in Visual Studio, the following articles also provide from general background:

- [Work with Python in Visual Studio](#)
- [Install Python support in Visual Studio](#)

NOTE

You can't manage environments for Python code that is opened only as a folder using the **File > Open > Folder** command. Instead, [Create a Python project from existing code](#) to enjoy the environment features of Visual Studio.

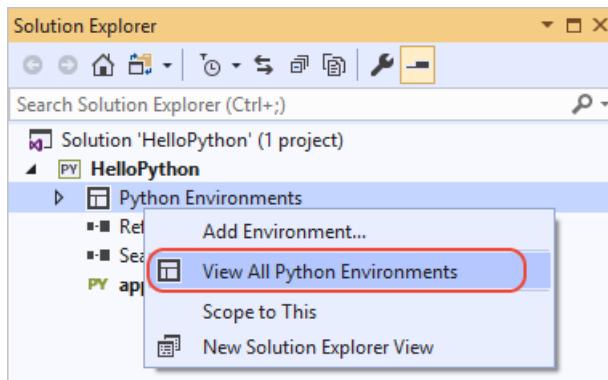
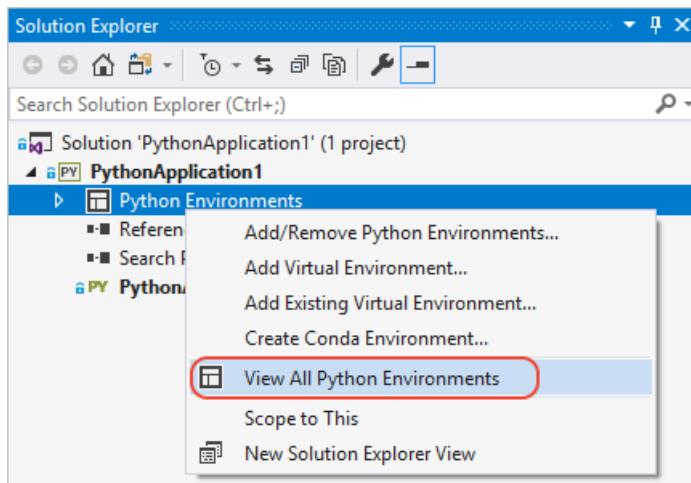
NOTE

You can manage environments for Python code that is opened as a folder using the **File > Open > Folder** command. The Python toolbar allows you switch between all detected environments, and also add a new environment. The environment information is stored in the PythonSettings.json file in the Workspace .vs folder.

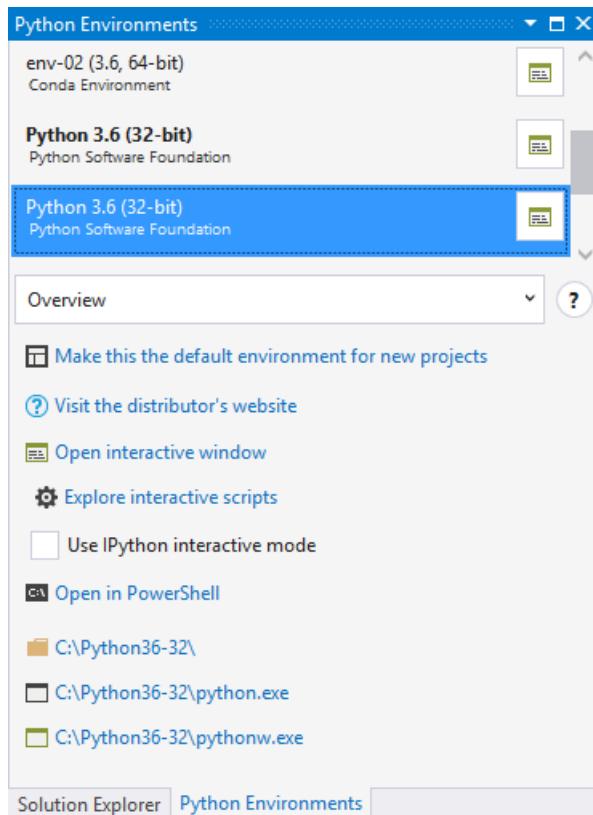
The Python Environments window

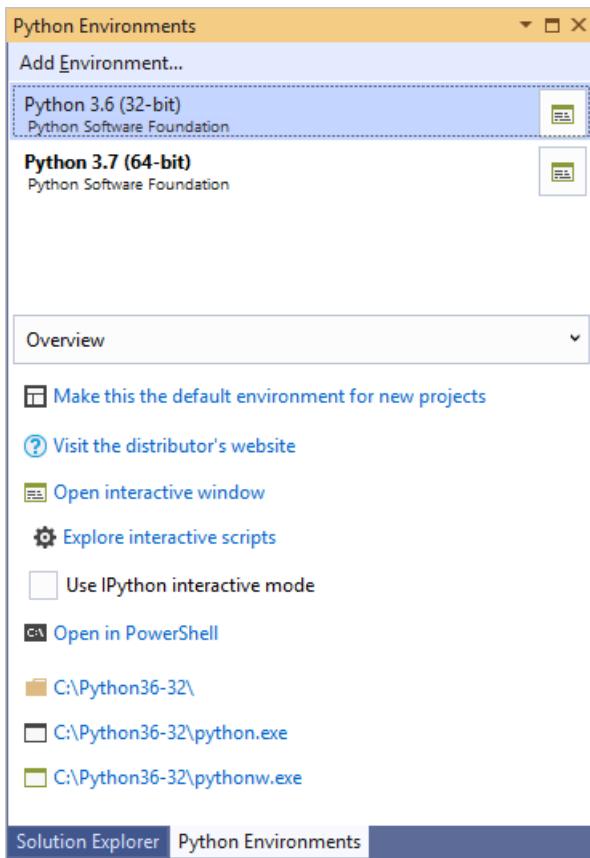
The environments that Visual Studio knows about are displayed in the **Python Environments** window. To open the window, use one of the following methods:

- Select the **View > Other Windows > Python Environments** menu command.
- Right-click the **Python Environments** node for a project in **Solution Explorer** and select **View All Python Environments**:



In either case, the **Python Environments** window appears alongside **Solution Explorer**:





Visual Studio looks for installed global environments using the registry (following [PEP 514](#)), along with virtual environments and conda environments (see [Types of environments](#)). If you don't see an expected environment in the list, see [Manually identify an existing environment](#).

When you select an environment in the list, Visual Studio displays various properties and commands for that environment on the **Overview** tab. For example, you can see in the image above that the interpreter's location is C:\Python36-32. The four commands at the bottom of the **Overview** tab each open a command prompt with the interpreter running. For more information, see [Python Environments window tab reference - Overview](#).

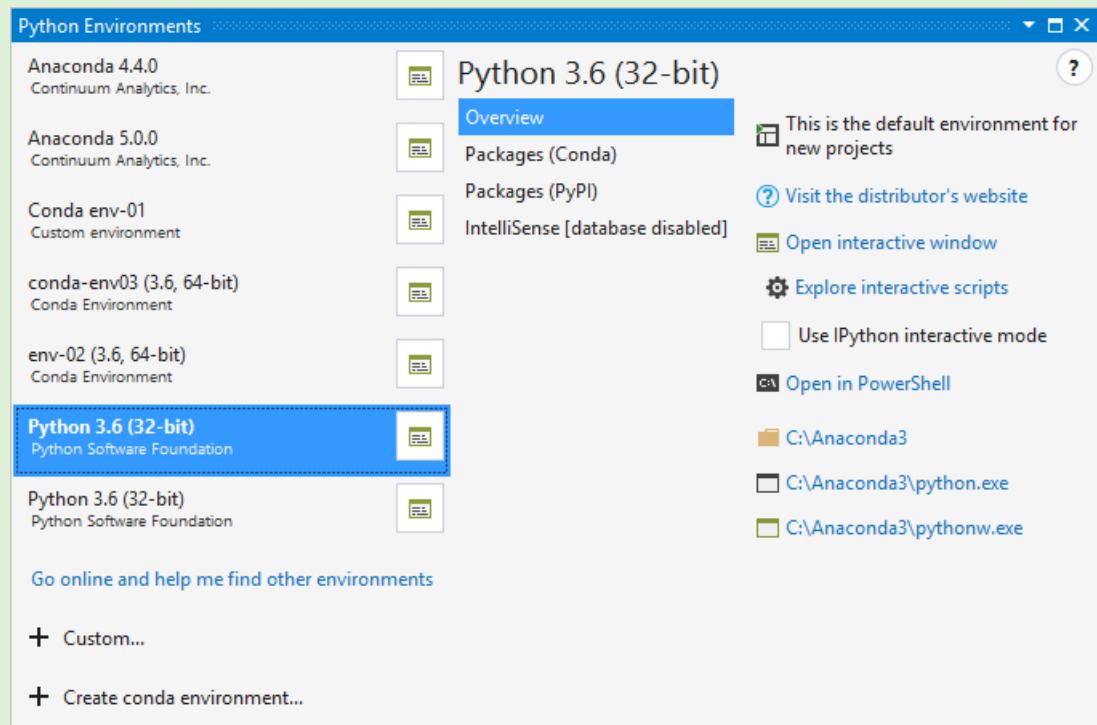
Use the drop-down list below the list of environments to switch to different tabs such as **Packages**, and **IntelliSense**. These tabs are also described in the [Python Environments window tab reference](#).

Selecting an environment doesn't change its relation to any projects. The default environment, shown in boldface in the list, is the one that Visual Studio uses for any new projects. To use a different environment with new projects, use the **Make this the default environment for new projects** command. Within the context of a project you can always select a specific environment. For more information, see [Select an environment for a project](#).

To the right of each listed environment is a control that opens an **Interactive** window for that environment. (In Visual Studio 2017 15.5 and earlier, another control appears that refreshes the IntelliSense database for that environment. See [Environments window tab reference](#) for details about the database.)

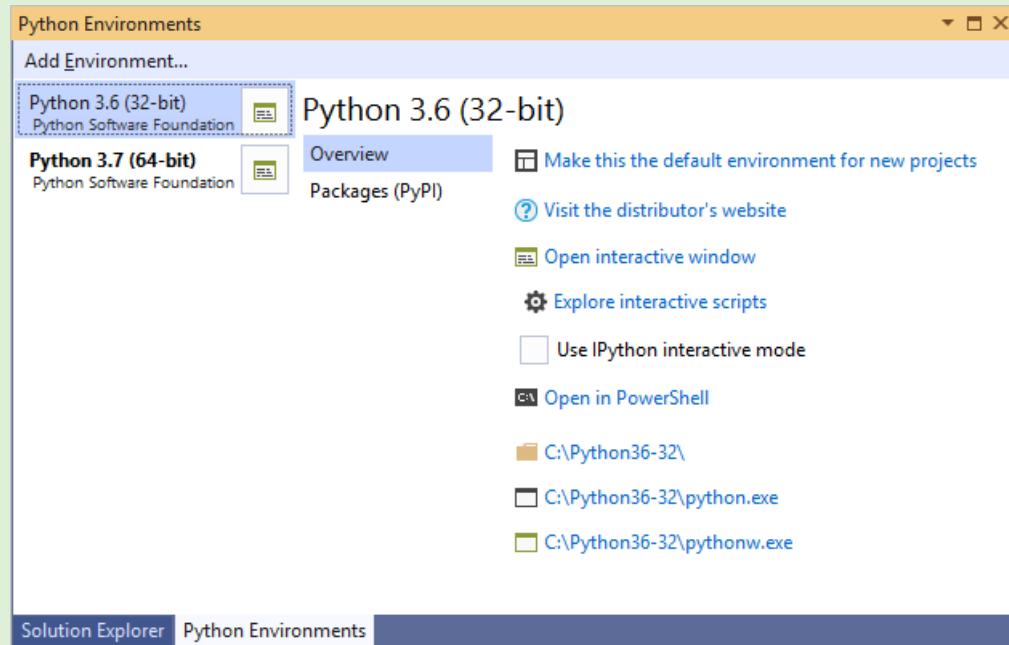
TIP

When you expand the **Python Environments** window wide enough, you get a fuller view of your environments that you may find more convenient to work with.



TIP

When you expand the **Python Environments** window wide enough, you get a fuller view of your environments that you may find more convenient to work with.



NOTE

Although Visual Studio respects the system-site-packages option, it doesn't provide a way to change it from within Visual Studio.

What if no environments appear?

If no environments appear, it means Visual Studio failed to detect any Python installations in standard locations. For example, you may have installed Visual Studio 2017 or later but cleared all the interpreter options in the installer options for the Python workload. Similarly, you may have installed Visual Studio 2015 or earlier but did not install an interpreter manually (see [Install Python interpreters](#)).

If you know you have a Python interpreter on your computer but Visual Studio (any version) did not detect it, then use the **+ Custom** command to specify its location manually. See the next section, [Manually identify an existing environment](#).

TIP

Visual Studio detects updates to an existing interpreter, such as upgrading Python 2.7.11 to 2.7.14 using the installers from [python.org](#). During the installation process, the older environment disappears from the **Python Environments** list before the update appears in its place.

However, if you manually move an interpreter and its environment using the file system, Visual Studio won't know the new location. For more information, see [Move an interpreter](#).

Types of environments

Visual Studio can work with global, virtual, and conda environments.

Global environments

Each Python installation (for example, Python 2.7, Python 3.6, Python 3.7, Anaconda 4.4.0, etc., see [Install Python interpreters](#)) maintains its own *global environment*. Each environment is composed of the specific Python interpreter, its standard library, a set of pre-installed packages, and any additional packages you install while that environment is activated. Installing a package into a global environment makes it available to all projects using that environment. If the environment is located in a protected area of the file system (within `c:\program files`, for example), then installing packages requires administrator privileges.

Global environments are available to all projects on the computer. In Visual Studio, you select one global environment as the default, which is used for all projects unless you specifically choose a different one for a project. For more information, see [Select an environment for a project](#).

Virtual environments

Although working in a global environment is an easy way to get started, that environment will, over time, become cluttered with many different packages that you've installed for different projects. Such clutter makes it difficult to thoroughly test an application against a specific set of packages with known versions, which is exactly the kind of environment you'd set up on a build server or web server. Conflicts can also occur when two projects require incompatible packages or different versions of the same package.

For this reason, developers often create a *virtual environment* for a project. A virtual environment is a subfolder in a project that contains a copy of a specific interpreter. When you activate the virtual environment, any packages you install are installed only in that environment's subfolder. When you then run a Python program within that environment, you know that it's running against only those specific packages.

Visual Studio provides direct support for creating a virtual environment for a project. For example, if you open a project that contains a `requirements.txt`, or create a project from a template that includes that file, Visual Studio prompts you to automatically create a virtual environment and install those dependencies.

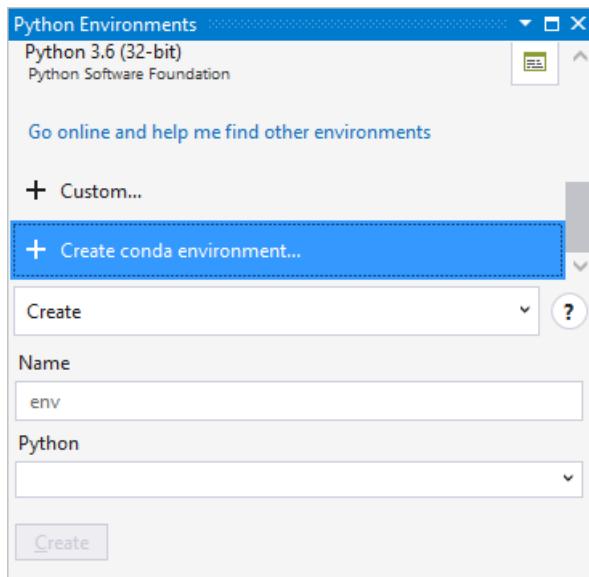
At any time within an open project, you can create a new virtual environment. In **Solution Explorer**, expand the project node, right-click **Python Environments**, and select "Add Virtual Environment." For more information, see [Create a virtual environment](#).

Visual Studio also provides a command to generate a `requirements.txt` file from a virtual environment, making it easy to recreate the environment on other computers. For more information, see [Use virtual environments](#).

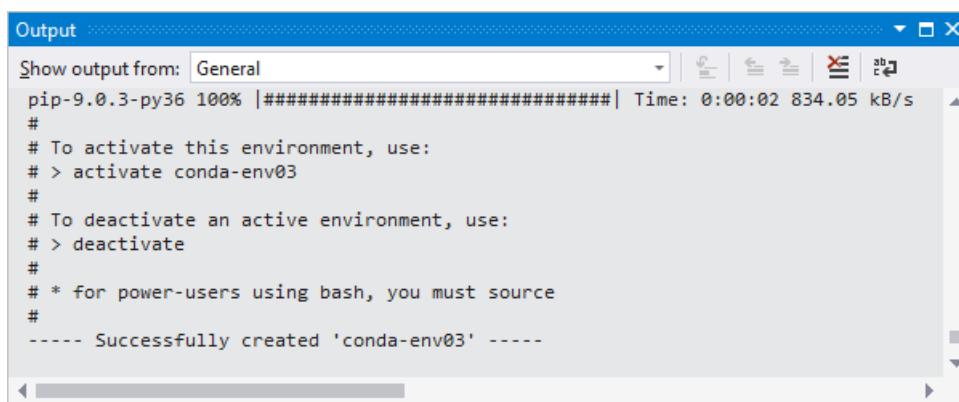
Conda environments

A conda environment is one created using the `conda` tool, or with integrated conda management in Visual Studio 2017 version 15.7 and higher. (Requires Anaconda or Miniconda, which are available through the Visual Studio installer, see [Installation](#).)

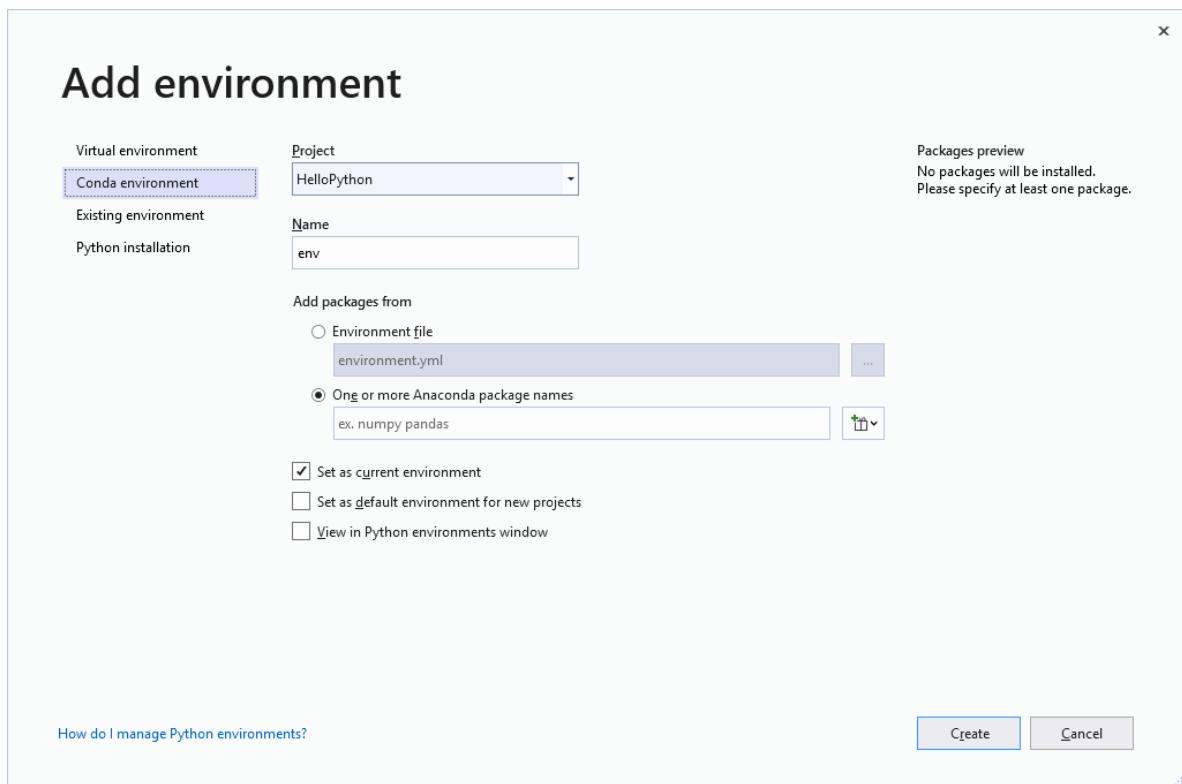
1. Select **+ Create conda environment** in the **Python Environments** window, which opens a **Create new conda environment** tab:



2. Enter a name for the environment in the **Name** field, select a base Python interpreter in the **Python** field, and select **Create**.
3. The **Output** window shows progress for the new environment, with a few CLI instructions once creation is complete:



4. Within Visual Studio, you can activate a conda environment for a project as you would any other environment as described on [Select an environment for a project](#).
 5. To install packages in the environment, use the [Packages tab](#).
1. Select **+ Add Environment** in the **Python Environments** window (or from the Python toolbar), which opens the **Add environment** dialog box. In that dialog, select the **Conda environment** tab:



2. Configure the following fields:

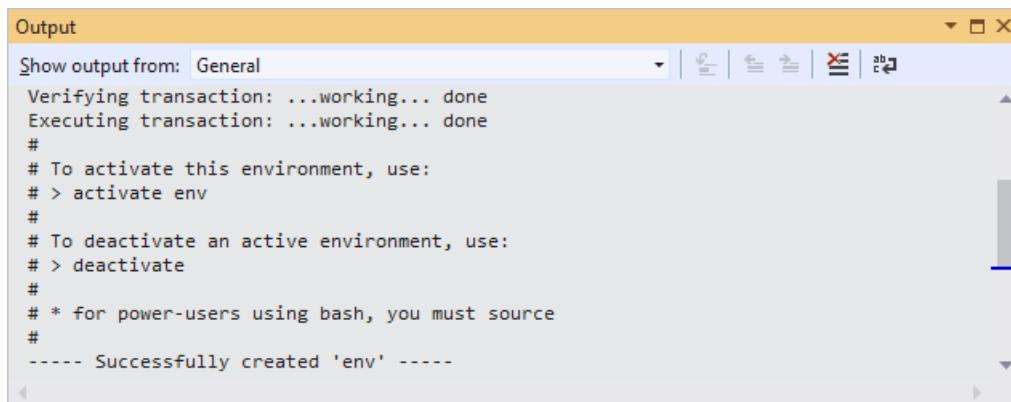
FIELD	DESCRIPTION
Project	The project in which to create the environment (if you have multiple projects in the same Visual Studio solution).
Name	The name for the conda environment.
Add packages from	Choose Environment file if you have an <i>environment.yml</i> file describing your dependencies, or choose One or more Anaconda package names and list at least one Python package or a Python version in the field below. The package list instructs conda to create a Python environment. To install the latest version of Python, use <code>python</code> ; to install a specific version, use <code>python=,major>.<minor></code> as in <code>python=3.7</code> . You can also use the package button to select Python versions and common packages from a series of menus.
Set as current environment	Activates the new environment in the selected project after the environment is created.
Set as default environment for new projects	Automatically sets and activates the conda environment in any new projects created in Visual Studio. This option is the same as using the Make this the default environment for new projects in the Python Environments window.
View in Python Environments window	Specifies whether to show the Python Environments window after creating the environment.

IMPORTANT

When creating a conda environment, be sure to specify at least one Python version or Python package using either `environments.yml` or the package list, which ensures that the environment contains a Python runtime. Otherwise, Visual Studio ignores the environment: the environment doesn't appear anywhere the **Python Environments** window, isn't set as the current environment for a project, and isn't available as a global environment.

If you happen to create a conda environment without a Python version, use the `conda info` command to see the locations of conda environment folders, then manually remove the subfolder for the environment from that location.

3. Select **Create**, and observe progress in the **Output** window. The output includes with a few CLI instructions once creation is complete:



```
Output
Show output from: General
Verifying transaction: ...working... done
Executing transaction: ...working... done
#
# To activate this environment, use:
# > activate env
#
# To deactivate an active environment, use:
# > deactivate
#
# * for power-users using bash, you must source
#
----- Successfully created 'env' -----
```

4. Within Visual Studio, you can activate a conda environment for a project as you would any other environment as described on [Select an environment for a project](#).
5. To install additional packages in the environment, use the [Packages tab](#).

NOTE

For best results with conda environments, use conda 4.4.8 or later (conda versions are different from Anaconda versions). You can install suitable versions of Miniconda (Visual Studio 2019) and Anaconda (Visual Studio 2017) through the Visual Studio installer.

To see the conda version, where conda environments are stored, and other information, run `conda info` at an Anaconda command prompt (that is, a command prompt where Anaconda is in the path):

```
conda info
```

Your conda environment folders appear as follows:

```
envs directories : C:\Users\user\.conda\envs
                  c:\anaconda3\envs
                  C:\Users\user\AppData\Local\conda\conda\envs
```

Because conda environments are not stored with a project, they act similarly to global environments. For example, installing a new package into a conda environment makes that package available to all projects using that environment.

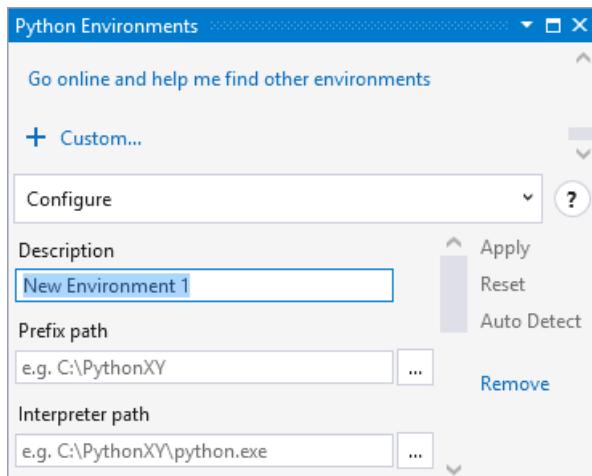
For Visual Studio 2017 version 15.6 and earlier, you can use conda environments by pointing to them manually as described under [Manually identify an existing environment](#).

Visual Studio 2017 version 15.7 and later detects conda environments automatically and displays them in the **Python Environments** window as described in the next section.

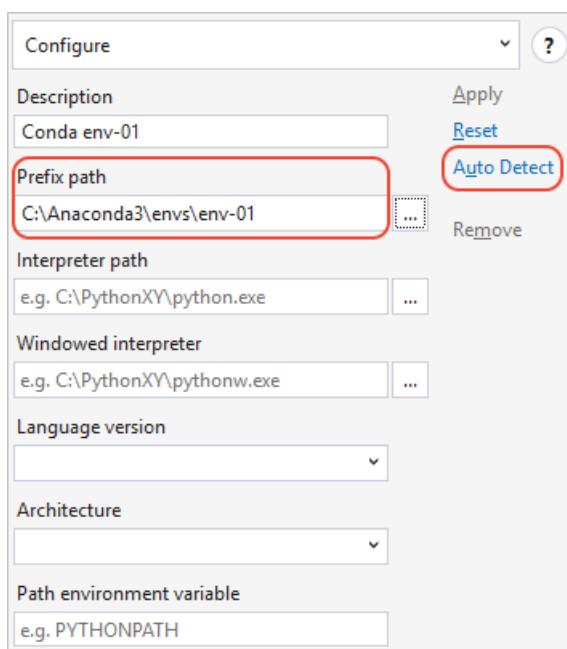
Manually identify an existing environment

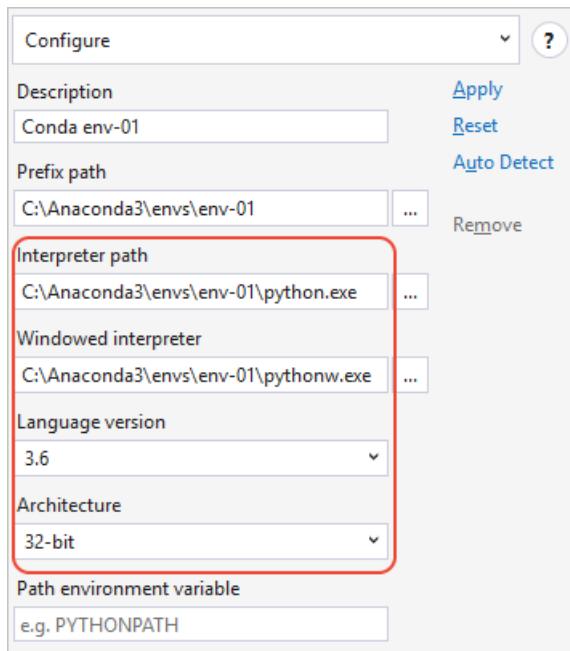
Use the following steps to identify an environment that's installed in a non-standard location (including conda environments in Visual Studio 2017 version 15.6 and earlier):

1. Select + **Custom** in the **Python Environments** window, which opens the **Configure** tab:

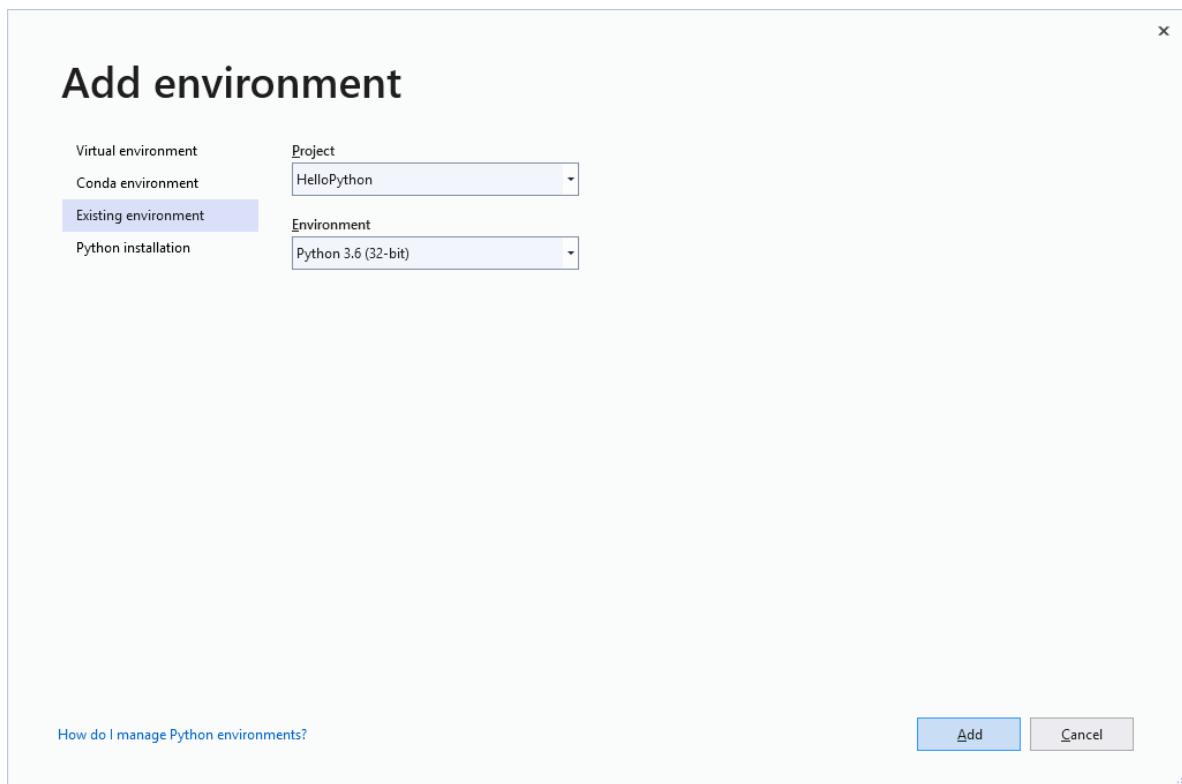


2. Enter a name for the environment in the **Description** field.
3. Enter or browse (using ...) to the path of the interpreter in the **Prefix path** field.
4. If Visual Studio detects a Python interpreter at that location (such as the path shown below for a conda environment), it enables the **Auto Detect** command. Selecting **Auto Detect** completes the remaining fields. You can also complete those fields manually.

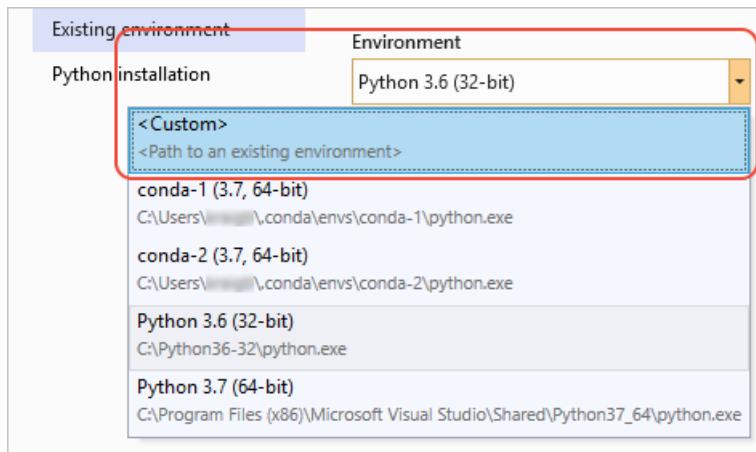




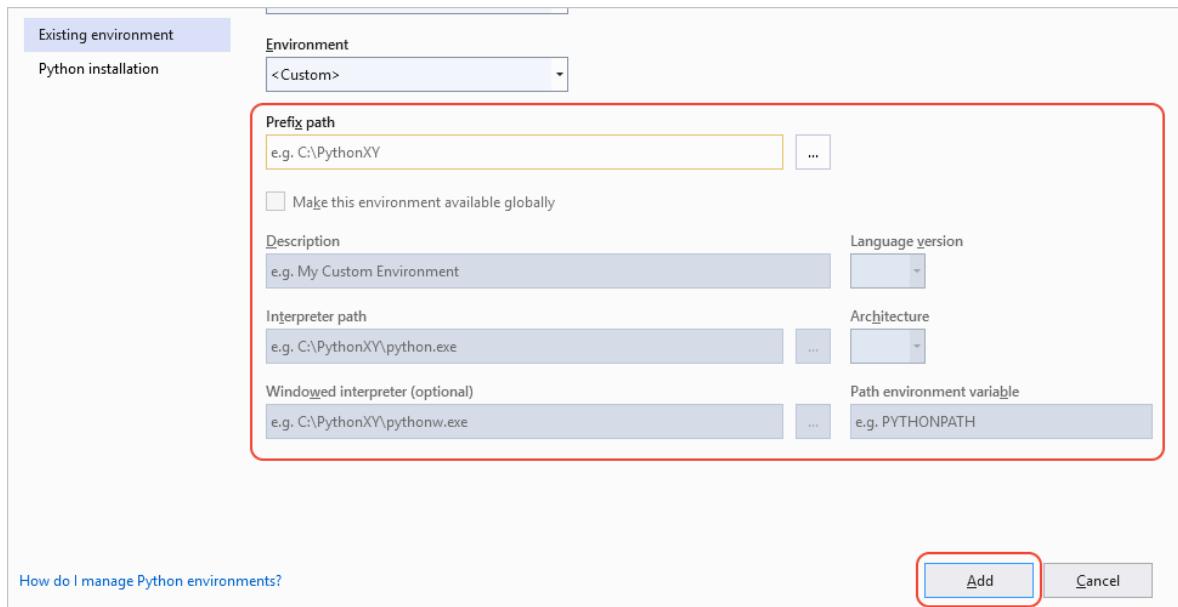
5. Once the fields contain the values you want, select **Apply** to save the configuration. You can now use the environment like any other within Visual Studio.
6. If you need to remove a manually identified environment, select the **Remove** command on the **Configure** tab. Auto-detected environments do not provide this option. For more information, see [Configure tab](#).
1. Select **+ Add Environment** in the **Python Environments** window (or from the Python toolbar), which opens the **Add environment** dialog box. In that dialog, select the **Existing environment** tab:



2. Select the **Environment** drop-down, then select **Custom**:



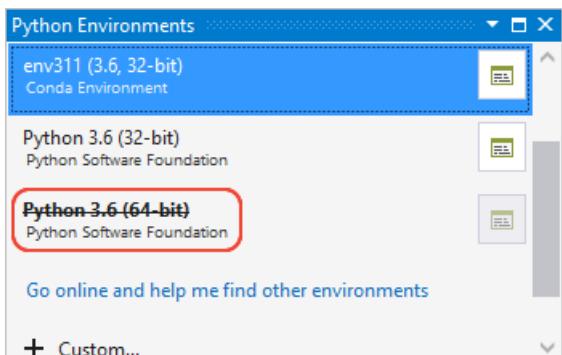
3. In the provided fields in the dialog box, enter or browse (using ...) to the path of the interpreter under **Prefix path**, which fills in most of the other fields. After reviewing those values and modifying as necessary, select **Add**.

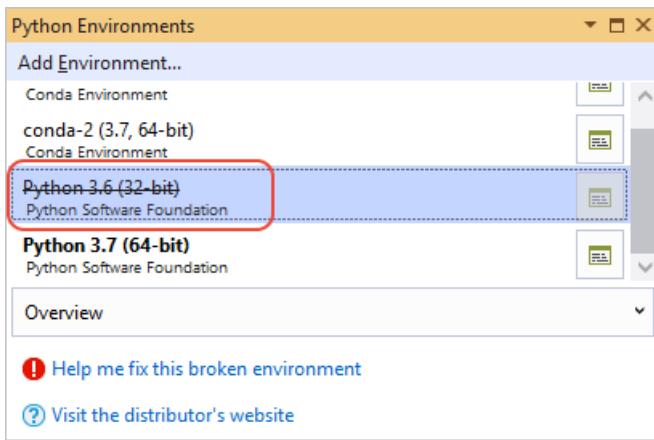


4. Details of the environment can be reviewed and modified at any time in the **Python Environments** window. In that window, select the environment, then select the **Configure** tab. After making changes, select the **Apply** command. You can also remove the environment using the **Remove** command (not available for auto-detected environments). For more information, see [Configure tab](#).

Fix or delete invalid environments

If Visual Studio finds registry entries for an environment, but the path to the interpreter is invalid, then the **Python Environments** window shows the name with a strikeout font:

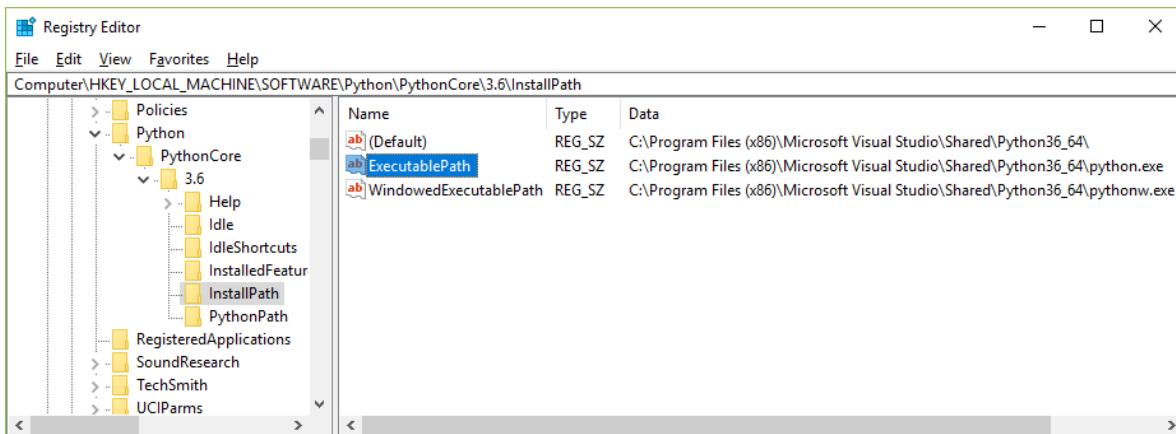




To correct an environment you wish to keep, first try using its installer's **Repair** process. The installers for standard Python 3.x, for example, include that option.

To correct an environment that doesn't have a repair option, or to remove an invalid environment, use the following steps to modify the registry directly. Visual Studio automatically updates the **Python Environments** window when you make changes to the registry.

1. Run `regedit.exe`.
2. Navigate to **HKEY_LOCAL_MACHINE\SOFTWARE\Python** for 32-bit interpreters, or **HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Python** for 64-bit interpreters. For IronPython, look for **IronPython** instead.
3. Expand the node that matches the distribution, such as **Python Core** for CPython or **ContinuumAnalytics** for Anaconda. For IronPython, expand the version number node.
4. Inspect the values under the **InstallPath** node:



- If the environment still exists on your computer, change the value of **ExecutablePath** to the correct location. Also correct the **(Default)** and **WindowedExecutablePath** values as necessary.
- If the environment no longer exists on your computer and you want to remove it from the **Python Environments** window, delete the parent node of **InstallPath**, such as **3.6** in the image above.

See also

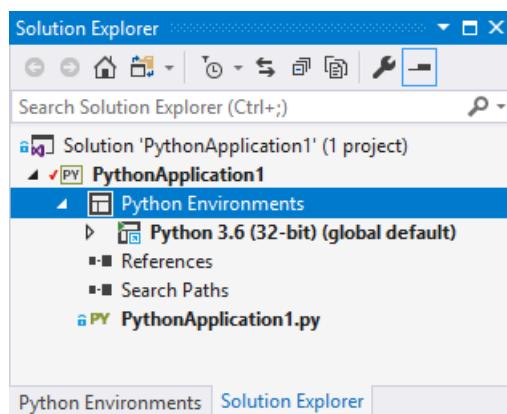
- [Install Python interpreters](#)
- [Select an interpreter for a project](#)
- [Use requirements.txt for dependencies](#)
- [Search paths](#)
- [Python Environments window reference](#)

How to select a Python environment for a project

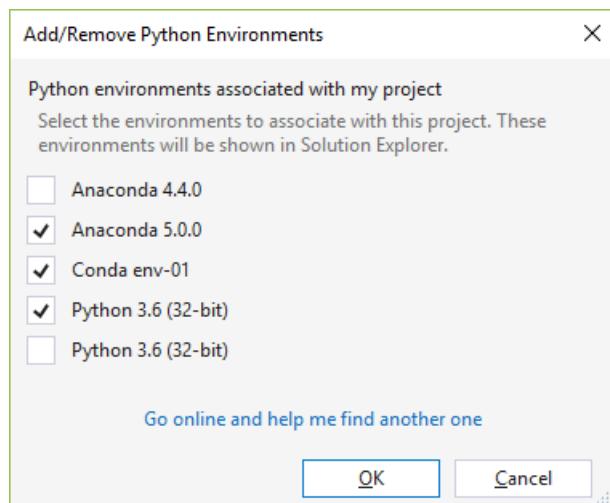
4/9/2019 • 9 minutes to read • [Edit Online](#)

All code in a Python project runs within the context of a specific environment, such as a global Python environment, an Anaconda environment, a virtual environment, or a conda environment. Visual Studio also uses that environment for debugging, import and member completions, syntax checking, and any other tasks that require language services that are specific to the Python version and a set of installed packages.

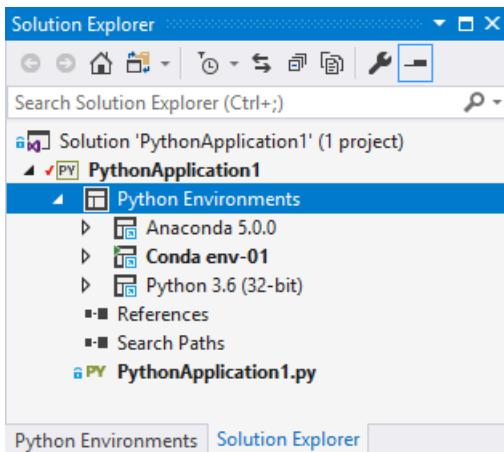
All new Python projects in Visual Studio are initially configured to use the default global environment, which appears under the **Python Environments** node in **Solution Explorer**:



To change the environment for a project, right-click the **Python Environments** node and select **Add/Remove Python Environments**. From the displayed list, which includes global, virtual, and conda environments, select all the ones you want to appear under the **Python Environments** node:



Once you select **OK**, all the selected environments appear under the **Python Environments** node. The currently activated environment appears in bold:



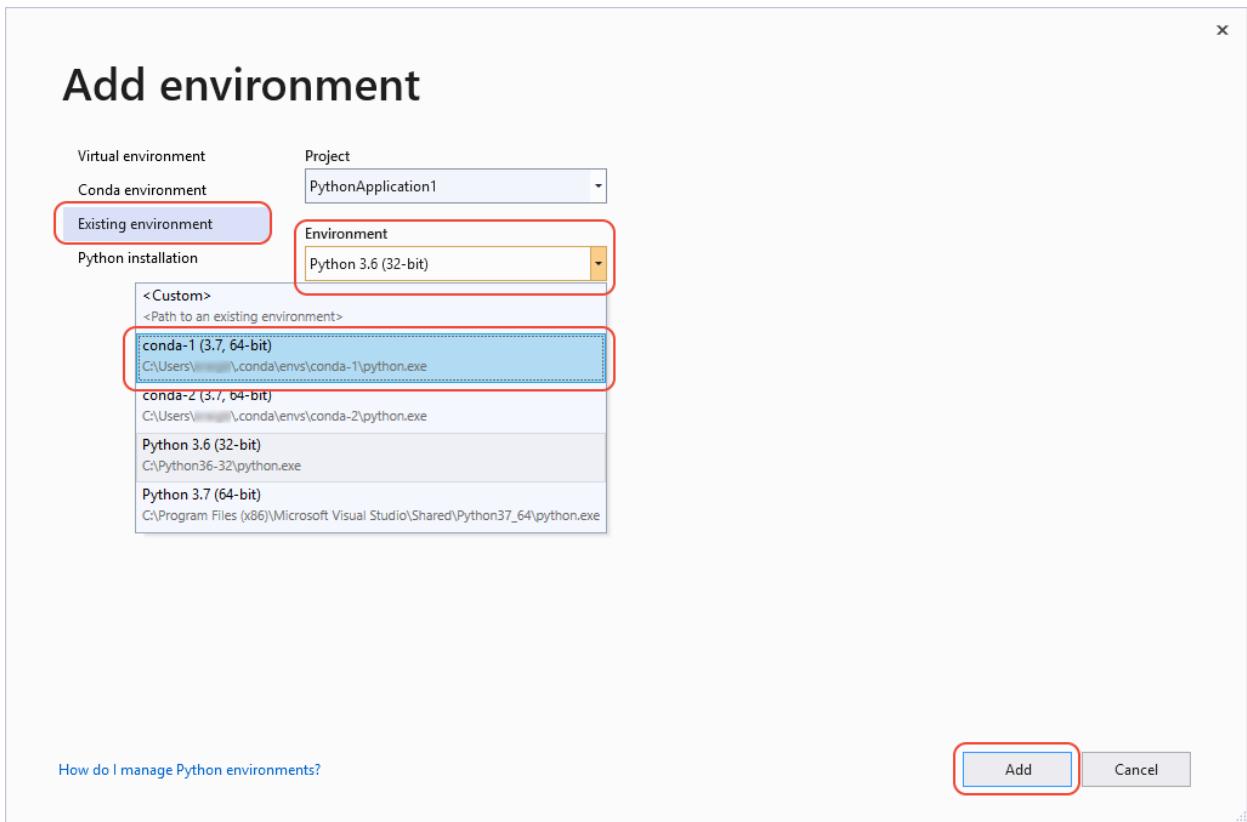
To quickly activate a different environment, right-click that environment name and select **Activate environment**. Your choice is saved with the project and that environment is activated whenever you open the project in the future. If you clear all the options in the **Add/Remove Python Environments** dialog, Visual Studio activates the global default environment.

The context menu on the **Python Environments** node also provides additional commands:

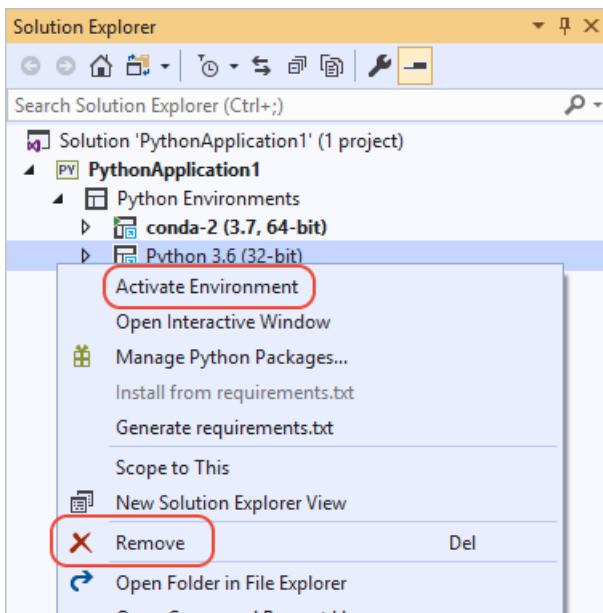
COMMAND	DESCRIPTION
Add Virtual Environment	Begins the process of creating a new virtual environment in the project. See Create a virtual environment .
Add Existing Virtual Environment	Prompts you to select a folder containing a virtual environment and adds it to the list under Python Environments , but does not activate it. See Activate an existing virtual environment .
Create Conda environment	Switches to the Python Environments window in which you enter a name for the environment and specify its base interpreter. See Conda environments .

To change the environment for a project, right-click the **Python Environments** node and select **Add Environment**, or select **Add Environment** from the environment drop-down in the Python toolbar.

Once in the **Add Environment** dialog box, select the **Existing environment** tab, then select a new environment from the **Environment** drop down list:



If you already added an environment other than the global default to a project, you may need to activate a newly added environment. Right-click that environment under the **Python Environments** node and select **Activate Environment**. To remove an environment from the project, select **Remove**.



Use virtual environments

A virtual environment is a unique combination of a specific Python interpreter and a specific set of libraries that is different from other global and conda environments. A virtual environment is specific to a project and is maintained in a project folder. That folder contains the environment's installed libraries along with a `pyvenv.cfg` file that specifies the path to the environment's *base interpreter* elsewhere on the file system. (That is, a virtual environment doesn't contain a copy of the interpreter, only a link to it.)

A benefit to using a virtual environment is that as you develop project over time, the virtual environment always reflects the project's exact dependencies. (A shared global environment, on the other hand, contains any number of libraries whether you use them in your project or not.) You can then easily create a `requirements.txt` file from

the virtual environment, which is then used to reinstall those dependencies on another development or production computer. For more information, see [Manage required packages with requirements.txt](#).

When you open a project in Visual Studio that contains a *requirements.txt* file, Visual Studio automatically gives you the option to recreate the virtual environment. On computers where Visual Studio isn't installed, you can use `pip install -r requirements.txt` to restore the packages.

Because a virtual environment contains a hard-coded path to the base interpreter, and because you can recreate the environment using *requirements.txt*, you typically omit the entire virtual environment folder from source control.

The following sections explain how to activate an existing virtual environment in a project and how to create a new virtual environment.

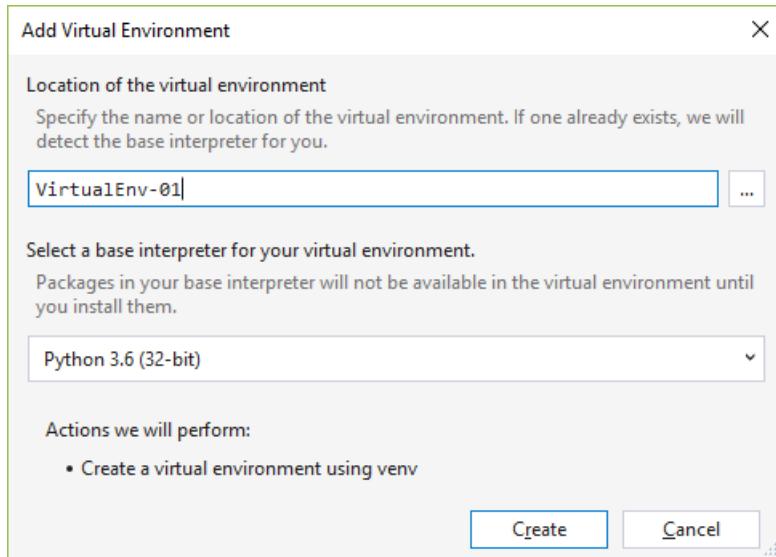
In Visual Studio, a virtual environment can be activated for a project like any other through the **Python Environments** node in **Solution Explorer**.

Once a virtual environment is added to your project, it appears in the **Python Environments** window. You can then activate it like any other environment, and you can manage its packages.

Create a virtual environment

You can create a new virtual environment directly in Visual Studio as follows:

1. Right-click **Python Environments** in **Solution Explorer** and select **Add Virtual Environment**, which brings up the following dialog box:

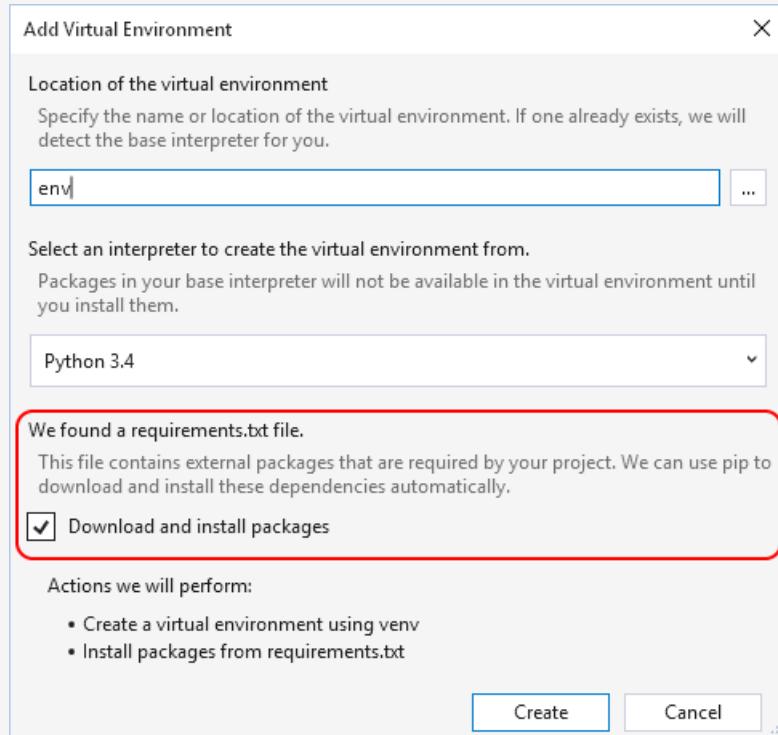


2. In the **Location of the virtual environment** field, specify a path for the virtual environment. If you specify a name only, the virtual environment is created within the current project in a subfolder with that name.
3. Select an environment as the base interpreter and select **Create**. Visual Studio displays a progress bar while it configures the environment and downloads any necessary packages. Upon completion, the virtual environment appears in the **Python Environments** window for the containing project.
4. The virtual environment is not activated by default. To activate it for the project, right-click it and select **Activate Environment**.

NOTE

If the location path identifies an existing virtual environment, Visual Studio detects the base interpreter automatically (using the *orig-prefix.txt* file in the environment's *lib* directory) and changes the **Create** button to **Add**.

If a *requirements.txt* file exists when adding a virtual environment, the **Add Virtual Environment** dialog displays an option to install the packages automatically, making it easy to recreate an environment on another computer:



Either way, the result is the same as if you'd used the **Add Existing Virtual Environment** command.

Activate an existing virtual environment

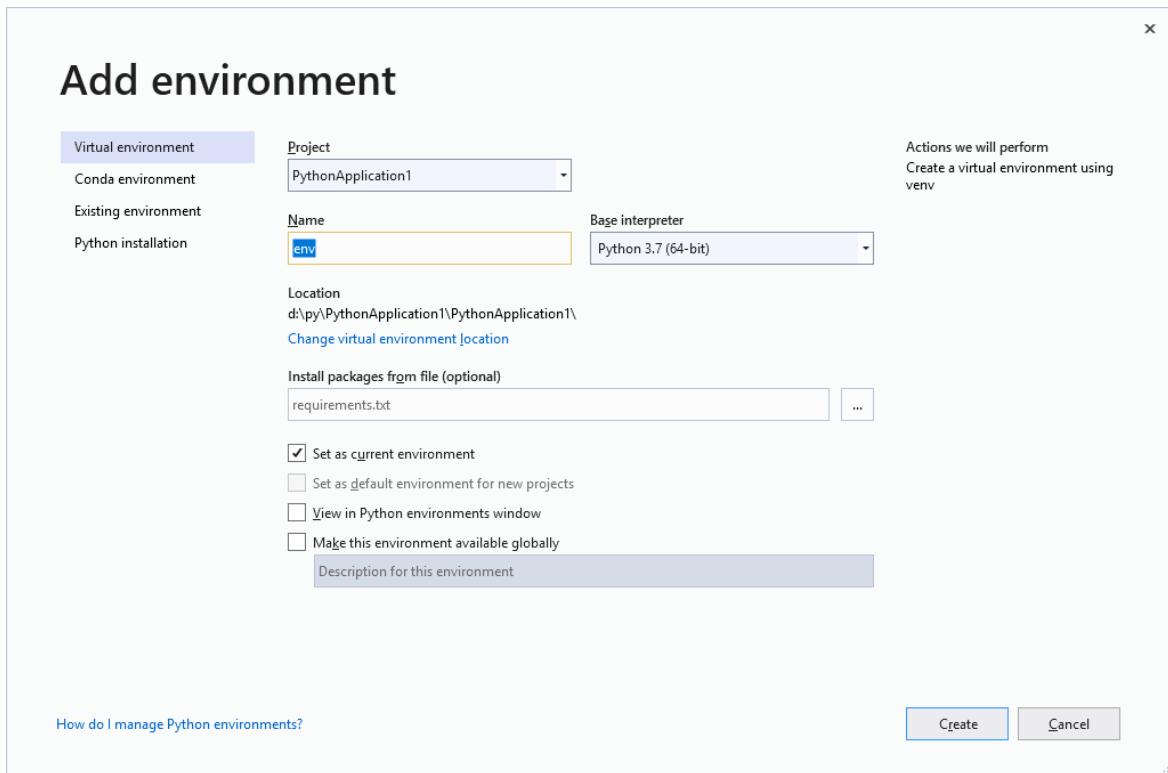
If you've already created a virtual environment elsewhere, you can activate it for a project as follows:

1. Right-click **Python Environments** in **Solution Explorer** and select **Add Existing Virtual Environment**.
2. In the **Browse** dialog that appears, navigate to and select the folder that contains the virtual environment, and select **OK**. If Visual Studio detects a *requirements.txt* file in that environment, it asks whether to install those packages.
3. After a few moments, the virtual environment appears under the **Python Environments** node in **Solution Explorer**. The virtual environment is not activated by default, so right-click it and select **Activate Environment**.

Create a virtual environment

You can create a new virtual environment directly in Visual Studio as follows:

1. Right-click **Python Environments** in **Solution Explorer** and select **Add Environment**, or select **Add Environment** from the environments drop down list on the Python toolbar. In the **Add Environment** dialog that appears, select the **Virtual Environment** tab:



2. Specify a name for the virtual environment, select a base interpreter, and verify its location. Under **Install packages from file**, provide the path to a *requirements.txt* file if desired.
3. Review the other options in the dialog:

OPTION	DESCRIPTION
Set as current environment	Activates the new environment in the selected project after the environment is created.
Set as default environment for new projects	Automatically sets and activates the virtual environment in any new projects created in Visual Studio. When using this option, the virtual environment should be placed in a location outside of a specific project.
View in Python Environments window	Specifies whether to open the Python Environments window after creating the environment.
Make this environment available globally	Specifies whether the virtual environment also acts as a global environment. When using this option, the virtual environment should be placed in a location outside of a specific project.

4. Select **Create** to finalize the virtual environment. Visual Studio displays a progress bar while it configures the environment and downloads any necessary packages. Upon completion, the virtual environment is activated and appears in the **Python Environments** node in **Solution Explorer** and the **Python Environments** window for the containing project.

Activate an existing virtual environment

If you've already created a virtual environment elsewhere, you can activate it for a project as follows:

1. Right-click **Python Environments** in **Solution Explorer** and select **Add Environment**.
2. In the **Browse** dialog that appears, navigate to and select the folder that contains the virtual environment, and select **OK**. If Visual Studio detects a *requirements.txt* file in that environment, it asks whether to

install those packages.

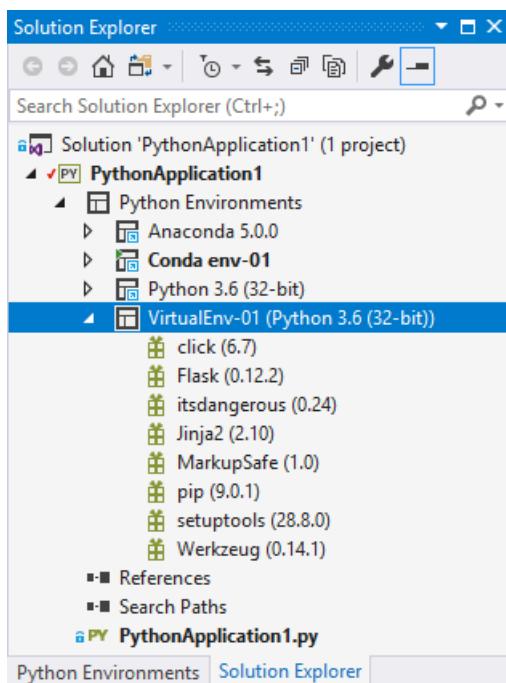
3. After a few moments, the virtual environment appears under the **Python Environments** node in **Solution Explorer**. The virtual environment is not activated by default, so right-click it and select **Activate Environment**.

Remove a virtual environment

1. In **Solution Explorer**, right-click the virtual environment and select **Remove**.
2. Visual Studio asks whether to remove or delete the virtual environment. Selecting **Remove** makes it unavailable to the project but leaves it on the file system. Selecting **Delete** both removes the environment from the project and deletes it from the file system. The base interpreter is unaffected.

View installed packages

In Solution Explorer, expand any specific environment's node to quickly view the packages that are installed in that environment (meaning that you can import and use those packages in your code when the environment is active):



To install new packages, right-click the environment and select **Install Python Package** to switch to the appropriate **Packages** tab in the **Python Environments** window. Enter a search term (usually the package name) and Visual Studio displays matching packages.

To install new packages, right-click the environment and select **Manage Python Packages** (or use the package button on the Python toolbar) to switch to the appropriate **Packages** tab in the **Python Environments** window. Once in the **Packages** tab, enter a search term (usually the package name) and Visual Studio displays matching packages.

Within Visual Studio, packages (and dependencies) for most environments are downloaded from the [Python Package Index \(PyPI\)](#), where you can also search for available packages. Visual Studio's status bar and output window show information about the install. To uninstall a package, right-click it and select **Remove**.

The conda package manager generally uses <https://repo.continuum.io/pkgs/> as the default channel, but other channels are available. For more information see [Manage Channels](#) (docs.conda.io).

Be aware that the displayed entries may not always be accurate, and installation and uninstallation may not be reliable or available. Visual Studio uses the pip package manager if available, and downloads and installs it when required. Visual Studio can also use the easy_install package manager. Packages installed using `pip` or

`easy_install` from the command line are also displayed.

Also note that Visual Studio does not presently support using `conda` to install packages into a conda environment. Use `conda` from the command line instead.

TIP

A common situation where pip fails to install a package is when the package includes source code for native components in `*.pyd` files. Without the required version of Visual Studio installed, pip cannot compile these components. The error message displayed in this situation is **error: Unable to find vcvarsall.bat**. `easy_install` is often able to download pre-compiled binaries, and you can download a suitable compiler for older versions of Python from <https://aka.ms/VCPython27>. For more details, see [How to deal with the pain of "unable to find vcvarsallbat"](#) on the Python tools team blog.

See also

- [Manage Python environments in Visual Studio](#)
- [Use requirements.txt for dependencies](#)
- [Search paths](#)
- [Python Environments window reference](#)

Manage required packages with requirements.txt

4/9/2019 • 2 minutes to read • [Edit Online](#)

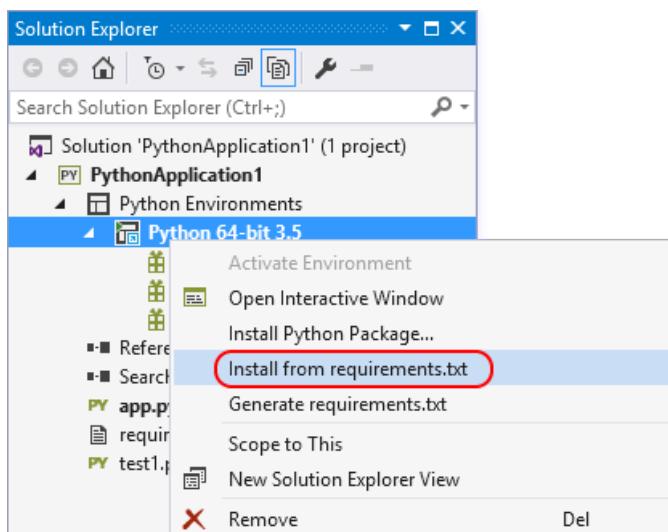
If you share a project with others, use a build system, or plan to copy the project to any other location where you need to restore an environment, you need to specify the external packages that the project requires. The recommended approach is to use a [requirements.txt file](#) ([readthedocs.org](#)) that contains a list of commands for pip that installs the required versions of dependent packages. The most common command is

```
pip freeze > requirements.txt
```

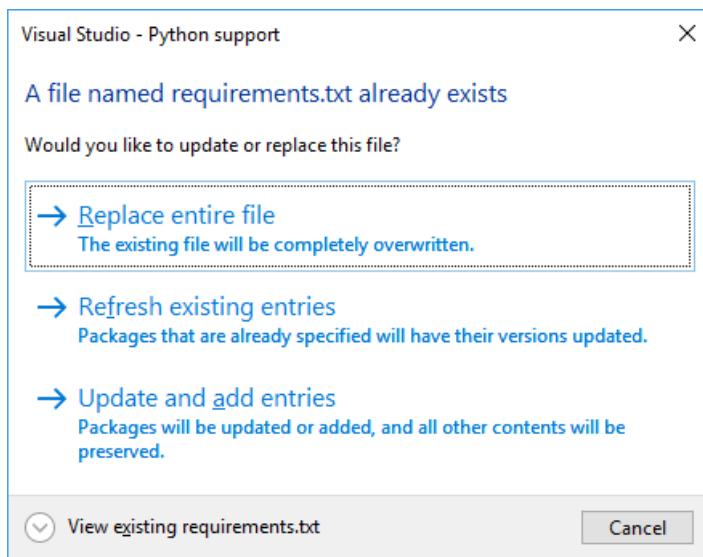
, which records an environment's current package list into *requirements.txt*.

Technically, any filename may be used to track requirements (by using `-r <full path to file>` when installing a package), but Visual Studio provides specific support for *requirements.txt*:

- If you've loaded a project that contains *requirements.txt* and wish to install all the packages listed in that file, expand the **Python Environments** node in **Solution Explorer**, then right-click an environment node and select **Install from requirements.txt**:



- If you want to install the dependencies in a virtual environment, create and activate that environment first, then use the **Install from requirements.txt** command. For more information on creating a virtual environment, see [Use virtual environments](#).
- If you already have all the necessary packages installed in an environment, you can right-click that environment in **Solution Explorer** and select **Generate requirements.txt** to create the necessary file. If the file already exists, a prompt appears for how to update it:



- **Replace entire file** removes all items, comments, and options that exist.
- **Refresh existing entries** detects package requirements and updates the version specifiers to match the version you currently have installed.
- **Update and add entries** refreshes any requirements that are found, and adds all other packages to the end of the file.

Because *requirements.txt* files are intended to freeze the requirements of an environment, all installed packages are written with precise versions. Using precise versions ensures that you can easily reproduce your environment on another computer. Packages are included even if they were installed with a version range, as a dependency of another package, or with an installer other than pip.

If a package cannot be installed by pip and it appears in a *requirements.txt* file, the entire installation fails. In this case, manually edit the file to exclude this package or to use [pip's options](#) to refer to an installable version of the package. For example, you may prefer to use `pip wheel` to compile a dependency and add the

`--find-links <path>` option to your *requirements.txt*:

```
C:\Project>pip wheel azure
Downloading/unpacking azure
  Running setup.py (path:C:\Project\env\build\azure\setup.py) egg_info for package azure

Building wheels for collected packages: azure
  Running setup.py bdist_wheel for azure
  Destination directory: c:\project\wheelhouse
Successfully built azure
Cleaning up...

C:\Project>type requirements.txt
--find-links wheelhouse
--no-index
azure==0.8.0

C:\Project>pip install -r requirements.txt -v
Downloading/unpacking azure==0.8.0 (from -r requirements.txt (line 3))
  Local files found: C:/Project/wheelhouse/azure-0.8.0-py3-none-any.whl
Installing collected packages: azure
Successfully installed azure
Cleaning up...
  Removing temporary dir C:\Project\env\build...
```

See also

- [Manage Python environments in Visual Studio](#)

- [Select an interpreter for a project](#)
- [Search paths](#)
- [Python Environments window reference](#)

How Visual Studio uses Python search paths

4/9/2019 • 2 minutes to read • [Edit Online](#)

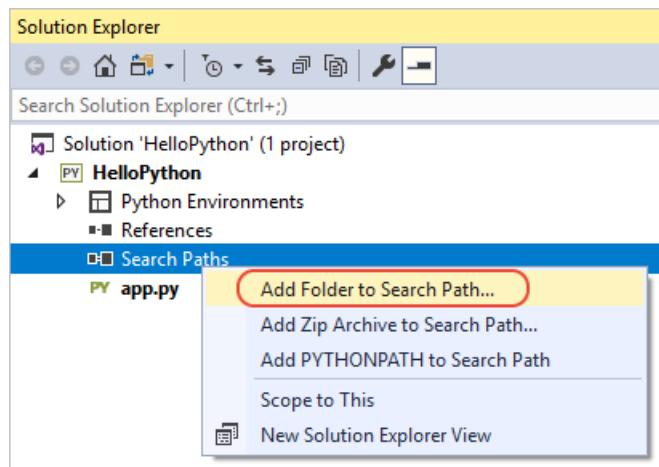
With typical Python usage, the `PYTHONPATH` environment variable (or `IRONPYTHONPATH`, etc.) provides the default search path for module files. That is, when you use an `from <name> import...` or `import <name>` statement, Python searches the following locations in order for a matching name:

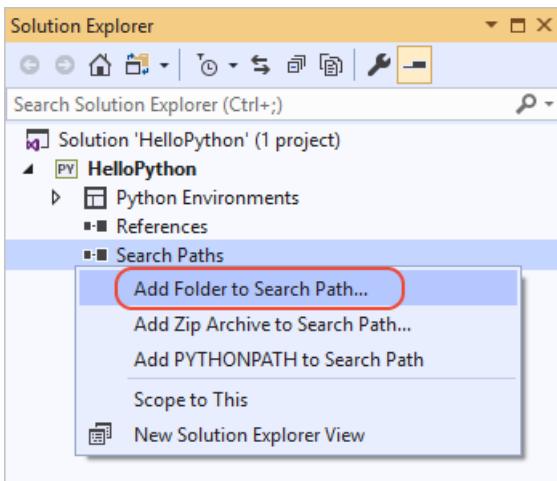
1. Python's built-in modules.
2. The folder containing the Python code you're running.
3. The "module search path" as defined by the applicable environment variable. (See [The Module Search Path](#) and [Environment variables](#) in the core Python documentation.)

Visual Studio ignores the search path environment variable, however, even when the variable is set for the entire system. It's ignored, in fact, precisely *because* it's set for the entire system and thus raises certain questions that cannot be answered automatically: Are the referenced modules meant for Python 2.7 or Python 3.6+? Are they going to override standard library modules? Is the developer aware of this behavior or is it a malicious hijacking attempt?

Visual Studio thus provides a means to specify search paths directly in both environments and projects. Code that you run or debug in Visual Studio receives search paths in the value of `PYTHONPATH` (and other equivalent variables). By adding search paths, Visual Studio inspects the libraries in those locations and builds IntelliSense databases for them when needed (Visual Studio 2017 version 15.5 and earlier; constructing the database may take some time depending on the number of libraries).

To add a search path, go to **Solution Explorer**, expand your project node, right-click on **Search Paths**, select **Add Folder to Search Path**:





This command displays a browser in which you then select the folder to include.

If your `PYTHONPATH` environment variable already includes the folder(s) you want, use the **Add PYTHONPATH to Search Paths** as a convenient shortcut.

Once folders are added to the search paths, Visual Studio uses those paths for any environment associated with the project. (You may see errors if the environment is based on Python 3 and you attempt to add a search path to Python 2.7 modules.)

Files with a `.zip` or `.egg` extension can also be added as search paths by selecting **Add Zip Archive to Search Path** command. As with folders, the contents of these files are scanned and made available to IntelliSense.

See also

- [Manage Python environments in Visual Studio](#)
- [Select an interpreter for a project](#)
- [Use requirements.txt for dependencies](#)
- [Python Environments window reference](#)

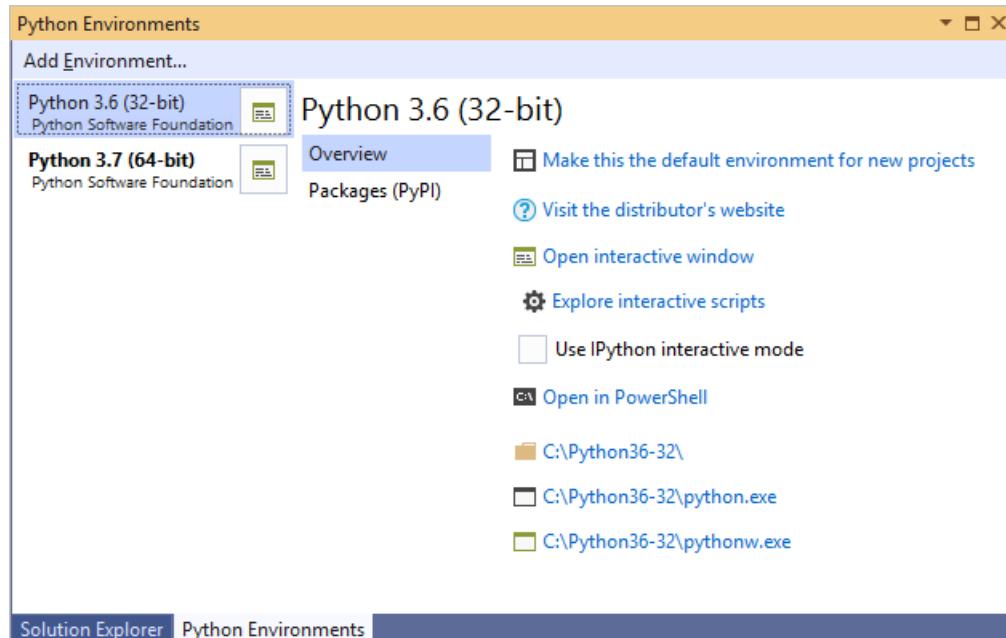
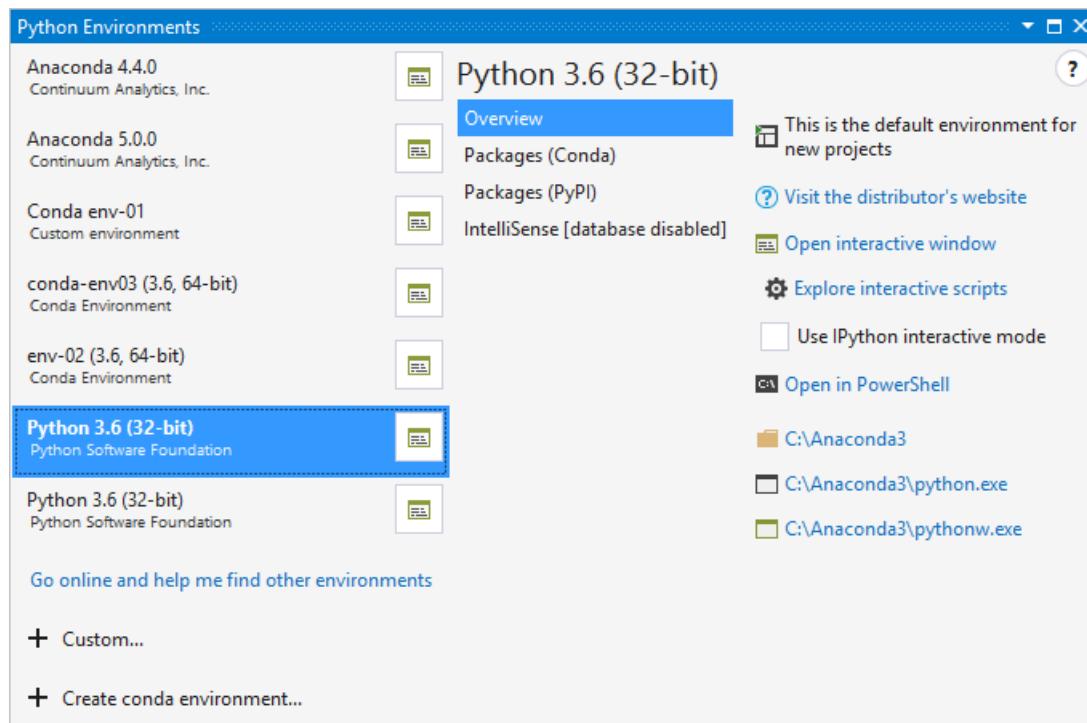
Python Environments window tabs reference

4/9/2019 • 7 minutes to read • [Edit Online](#)

To open the **Python Environments** window:

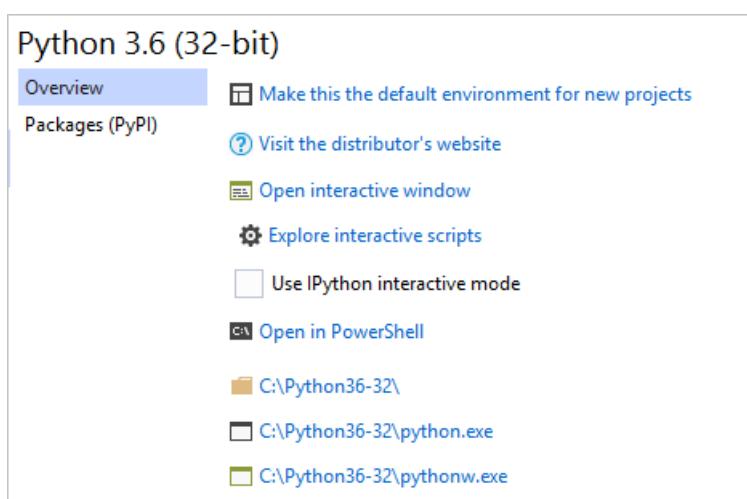
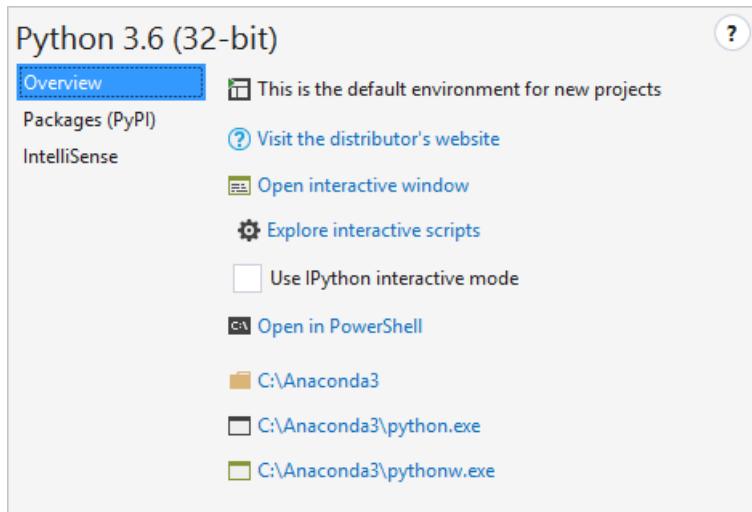
- Select the **View > Other Windows > Python Environments** menu command.
- Right-click the **Python Environments** node for a project in **Solution Explorer** and select **View All Python Environments**.

If you expand the **Python Environments** window wide enough, these options are shown as tabs, which you may find more convenient to work with. For clarity, the tabs in this article are shown in the expanded view.



Overview tab

Provides basic information and commands for the environment:



COMMAND	DESCRIPTION
Make this environment the default for new projects	Sets the active environment, which may cause Visual Studio (2017 version 15.5 and earlier) to briefly become non-responsive while it loads the IntelliSense database. Environments with many packages may be non-responsive for longer.
Visit the distributor's website	Opens a browser to the URL provided by the Python distribution. Python 3.x, for example, goes to python.org.
Open interactive window	Opens the interactive (REPL) window for this environment within Visual Studio, applying any startup scripts (see below).
Explore interactive scripts	See startup scripts .
Use IPython interactive mode	When set, opens the Interactive window with IPython by default. This enables inline plots as well as the extended IPython syntax such as <code>name?</code> to view help and <code>!command</code> for shell commands. This option is recommended when using an Anaconda distribution, as it requires extra packages. For more information, see Use IPython in the Interactive window .
Open in PowerShell	Starts the interpreter in a PowerShell command window.

COMMAND	DESCRIPTION
(Folder and program links)	Provide you quick access to the environment's installation folder, the <code>python.exe</code> interpreter, and the <code>pythonw.exe</code> interpreter. The first opens in Windows Explorer, the latter two open a console window.

Startup scripts

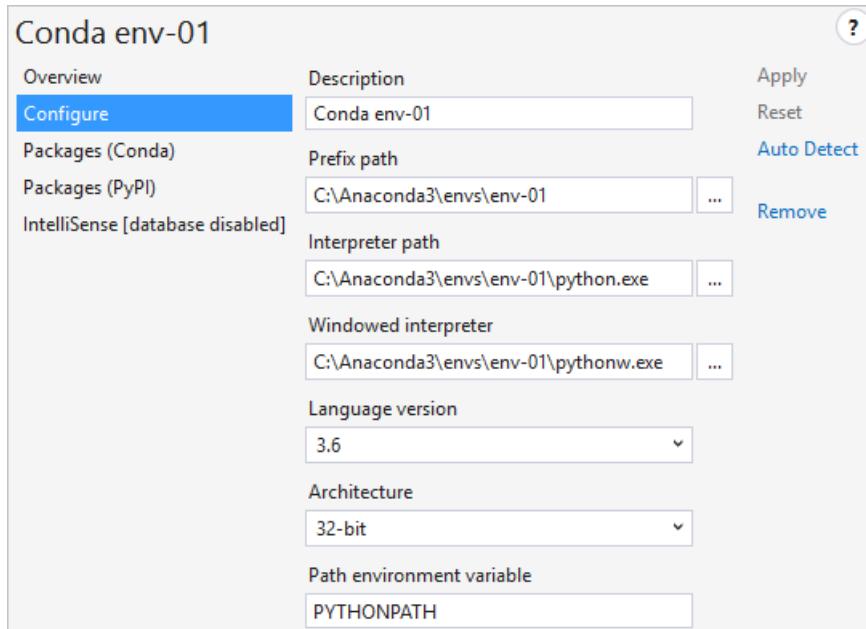
As you use interactive windows in your everyday workflow, you likely develop helper functions that you use regularly. For example, you may create a function that opens a DataFrame in Excel, and then save that code as a startup script so that it's always available in the **Interactive** window.

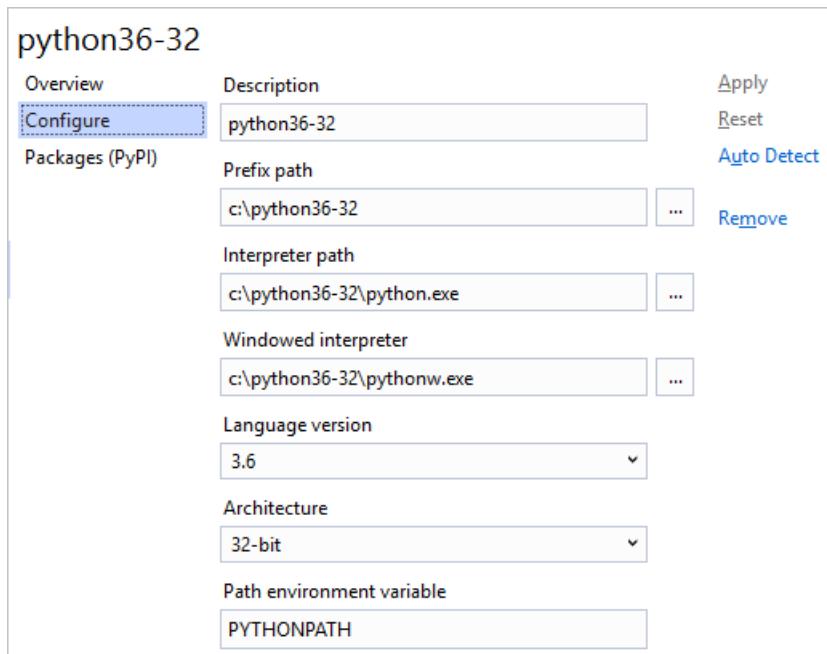
Startup scripts contain code that the **Interactive** window loads and runs automatically, including imports, function definitions, and literally anything else. Such scripts are referenced in two ways:

- When you install an environment, Visual Studio creates a folder `Documents\Visual Studio <version>\Python Scripts\<environment>` where `<version>` is the Visual Studio version (such as 2017 or 2019) and `<environment>` matches the name of the environment. You can easily navigate to the environment-specific folder with the **Explore interactive scripts** command. When you start the **Interactive** window for that environment, it loads and runs whatever `.py` files are found here in alphabetical order.
- The **Scripts** control in **Tools > Options > Python > Interactive Windows** tab (see [Interactive windows options](#)) is intended to specify an additional folder for startup scripts that are loaded and run in all environments. However, this feature doesn't work at present.

Configure tab

If available, the **Configure** tab contains details as described in the table below. If this tab isn't present, it means that Visual Studio is managing all the details automatically.





FIELD	DESCRIPTION
Description	The name to give the environment.
Prefix path	The base folder location of the interpreter. By filling this value and clicking Auto Detect , Visual Studio attempts to fill in the other fields for you.
Interpreter path	The path to the interpreter executable, commonly the prefix path followed by python.exe
Windowed interpreter	The path to the non-console executable, often the prefix path followed by pythonw.exe .
Library path (if available)	Specifies the root of the standard library, but this value may be ignored if Visual Studio is able to request a more accurate path from the interpreter.
Language version	Selected from the drop-down menu.
Architecture	Normally detected and filled in automatically, otherwise specifies 32-bit or 64-bit .
Path environment variable	The environment variable that the interpreter uses to find search paths. Visual Studio changes the value of the variable when starting Python so that it contains the project's search paths. Typically this property should be set to PYTHONPATH , but some interpreters use a different value.

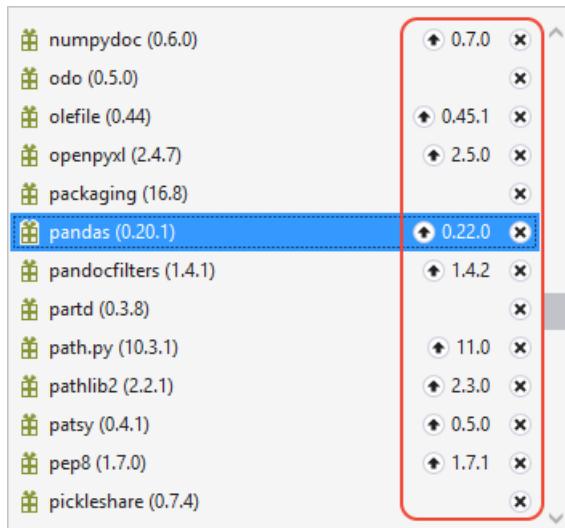
Packages tab

Also labeled "pip" in earlier versions.

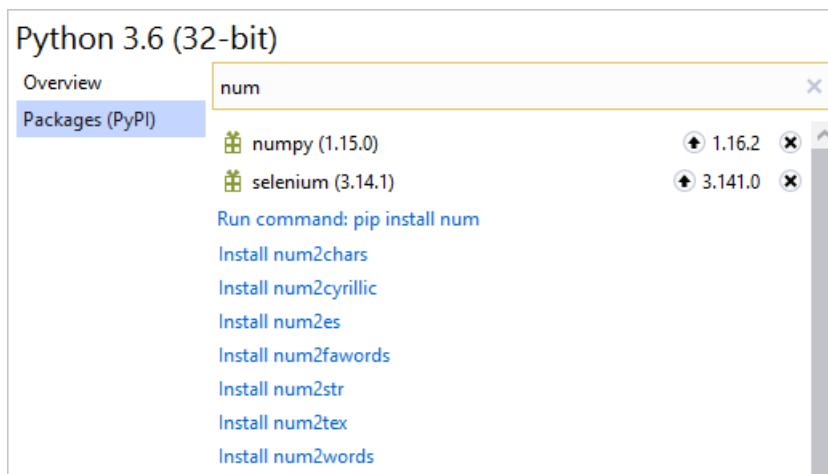
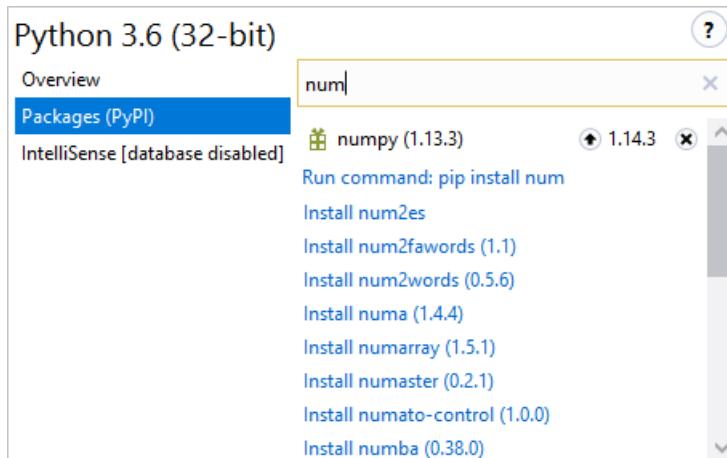
Manages the packages installed in the environment using pip (the **Packages (PyPI)** tab) or conda (the **Packages (Conda)** tab, for conda environments in Visual Studio 2017 version 15.7 and later). In this tab you can also search for and install new packages, including their dependencies.

Packages that are already installed appear with controls to update (an up arrow) and uninstall (the X in a circle)

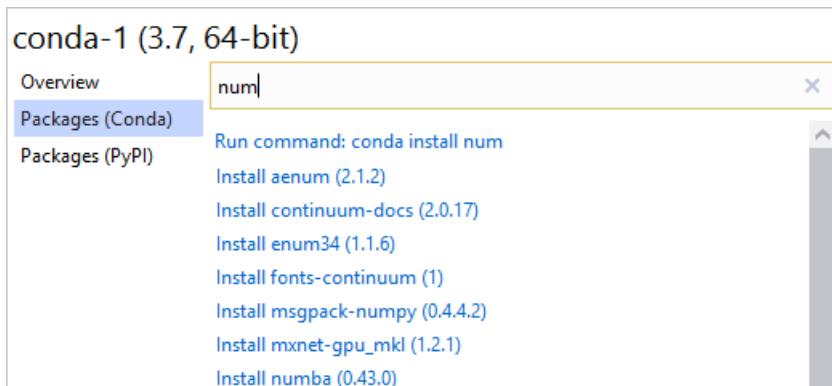
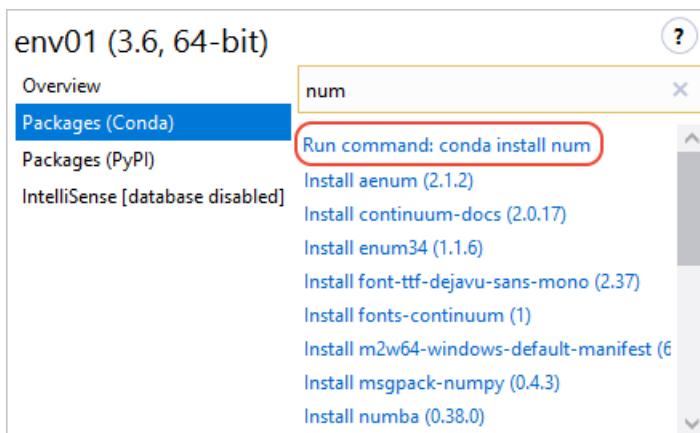
the package:



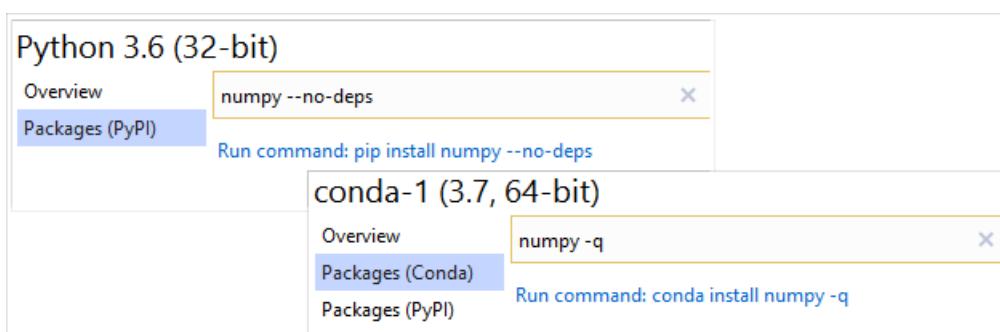
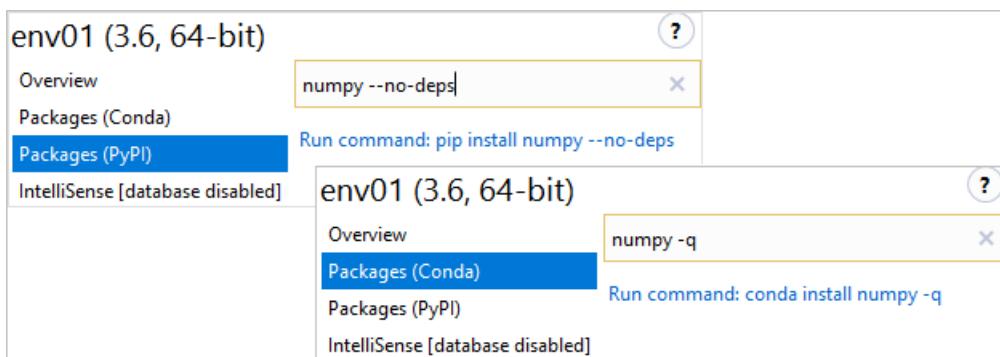
Entering a search term filters the list of installed packages as well as packages that can be installed from PyPI.



As you can see in the image above, the search results show a number of packages that match the search term; the first entry in the list, however, is a command to run `pip install <name>` directly. If you're on the **Packages (Conda)** tab, you instead see `conda install <name>`:



In both cases, you can customize the install by adding arguments in the search box after the name of the package. When you include arguments, the search results shows **pip install** or **conda install** followed by the contents of the search box:

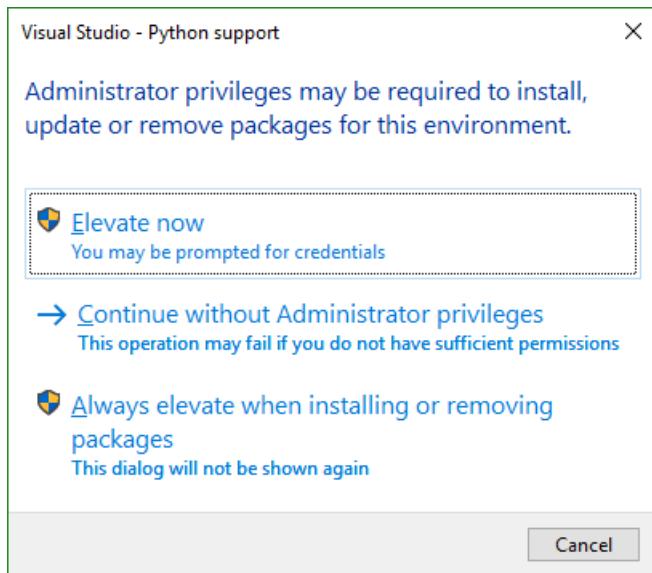


Installing a package creates subfolders within the environment's *Lib* folder on the file system. For example, if you have Python 3.6 installed in *c:\Python36*, packages are installed in *c:\Python36\Lib*; if you have Anaconda3 installed in *c:\Program Files\Anaconda3* then packages are installed in *c:\Program Files\Anaconda3\Lib*. For conda environments, packages are installed in that environment's folder.

Grant administrator privileges for package install

When installing packages into an environment that's located in a protected area of the file system, such as *c:\Program Files\Anaconda3\Lib*, Visual Studio must run `pip install` elevated to allow it to create package

subfolders. When elevation is required, Visual Studio displays the prompt, **Administrator privileges may be required to install, update or remove packages for this environment**:



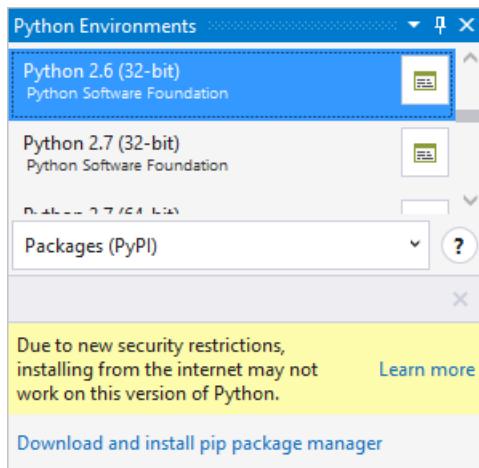
Elevate now grants administrative privileges to pip for a single operation, subject also to any operating system prompts for permissions. Selecting **Continue without Administrator privileges** attempts to install the package, but pip fails when trying to create folders with output such as **error: could not create 'C:\Program Files\Anaconda3\Lib\site-packages\png.py': Permission denied.**

Selecting **Always elevate when installing or removing packages** prevents the dialog from appearing for the environment in question. To make the dialog appear again, go to **Tools > Options > Python > General** and select the button, **Reset all permanently hidden dialogs**.

In that same **Options** tab, you can also select **Always run pip as administrator** to suppress the dialog for all environments. See [Options - General tab](#).

Security restrictions with older versions of Python

When using Python 2.6, 3.1 and 3.2, Visual Studio shows the warning, **Due to new security restrictions, installing from the internet may not work on this version of Python:**



The reason for the warning is that with these older versions of Python, `pip install` doesn't contain support for the Transport Security Layer (TLS) 1.2, which is required for downloading packages from the package source, pypi.org. Custom Python builds may support TLS 1.2 in which case `pip install` might work.

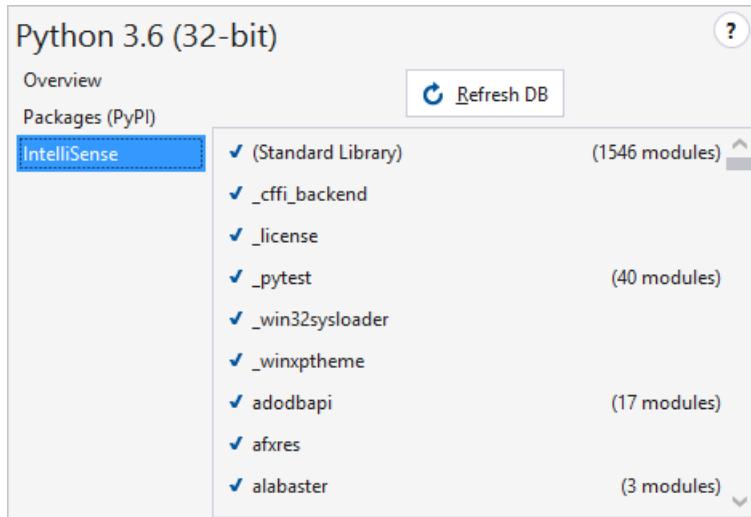
It may be possible to download the appropriate *get-pip.py* for a package from bootstrap.pypa.io, manually download a package from pypi.org, and then install the package from that local copy.

The recommendation, however, is to simply upgrade to Python 2.7 or 3.3+, in which case the warning does not

appear.

IntelliSense tab

Shows the current status of the IntelliSense completion database:



- In Visual Studio 2017 version 15.5 and earlier, IntelliSense completions depend on a database that's been compiled for that library. Building the database is done in the background when a library is installed, but can take some time and may not be complete when you start writing code.
- Visual Studio 2017 version 15.6 and later uses a faster method to provide completions that do not depend on the database by default. For this reason the tab is labeled **IntelliSense [database disabled]**. You can enable the database by clearing the option **Tools > Options > Python > Experimental > Use new style**.

IntelliSense for environments.

When Visual Studio detects a new environment (or you add one), it automatically begins to compile the database by analyzing the library source files. This process can take anywhere from a minute to an hour or more depending on what's installed. (Anaconda, for example, comes with many libraries and takes some time to compile the database.) Once complete, you get detailed IntelliSense and don't need to refresh the database again (with the **Refresh DB** button) until you install more libraries.

Libraries for which data haven't been compiled are marked with a !; if an environment's database isn't complete, a ! also appears next to it in the main environment list.

See also

- [Manage Python environments in Visual Studio](#)
- [Select an interpreter for a project](#)
- [Use requirements.txt for dependencies](#)
- [Search paths](#)

Configure Python web apps for IIS

4/9/2019 • 4 minutes to read • [Edit Online](#)

When using Internet Information Services (IIS) as a web server on a Windows computer (including [Windows virtual machines on Azure](#)), Python apps must include specific settings in their `web.config` files so that IIS can properly process Python code. The computer itself must also have Python installed along with any packages the web app requires.

NOTE

This article previously contained guidance for configuring Python on Azure App Service on Windows. The Python extensions and Windows hosts used in that scenario have been deprecated in favor of Azure App Service on Linux. For more information, see [Publishing Python Apps to Azure App Service \(Linux\)](#). The previous article, however, is still available on [Managing App Service on Windows with the Python extensions](#).

Install Python on Windows

To run a web app, first install your required version of Python directly on the Windows host machine as described on [Install Python interpreters](#).

Record the location of the `python.exe` interpreter for later steps. For convenience, you can add that location to your PATH environment variable.

Install packages

When using a dedicated host, you can use the global Python environment to run your app rather than a virtual environment. Accordingly, you can install all of your app's requirements into the global environment simply by running `pip install -r requirements.txt` at a command prompt.

Set `web.config` to point to the Python interpreter

Your app's `web.config` file instructs the IIS (7+) web server running on Windows about how it should handle Python requests through either `HttpPlatform` (recommended) or `FastCGI`. Visual Studio versions 2015 and earlier make these modifications automatically. When using Visual Studio 2017 and later, you must modify `web.config` manually.

Configure the `HttpPlatform` handler

The `HttpPlatform` module passes socket connections directly to a standalone Python process. This pass-through allows you to run any web server you like, but requires a startup script that runs a local web server. You specify the script in the `<httpPlatform>` element of `web.config`, where the `processPath` attribute points to the site extension's Python interpreter and the `arguments` attribute points to your script and any arguments you want to provide:

```

<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <system.webServer>
    <handlers>
      <add name="PythonHandler" path="*" verb="*" modules="httpPlatformHandler" resourceType="Unspecified"/>
    </handlers>
    <httpPlatform processPath="c:\python36-32\python.exe"
      arguments="c:\home\site\wwwroot\runserver.py --port %HTTP_PLATFORM_PORT%"
      stdoutLogEnabled="true"
      stdoutLogFile="c:\home\LogFiles\python.log"
      startupTimeLimit="60"
      processesPerApplication="16">
      <environmentVariables>
        <environmentVariable name="SERVER_PORT" value="%HTTP_PLATFORM_PORT%" />
      </environmentVariables>
    </httpPlatform>
  </system.webServer>
</configuration>

```

The `HTTP_PLATFORM_PORT` environment variable shown here contains the port that your local server should listen on for connections from localhost. This example also shows how to create another environment variable, if desired, in this case `SERVER_PORT`.

Configure the FastCGI handler

FastCGI is an interface that works at the request level. IIS receives incoming connections and forwards each request to a WSGI app running in one or more persistent Python processes.

To use it, first install and configure the `wfastcgi` package as described on pypi.org/project/wfastcgi/.

Next, modify your app's `web.config` file to include the full paths to `python.exe` and `wfastcgi.py` in the `PythonHandler` key. The steps below assume that Python is installed in `c:\python36-32` and that your app code is in `c:\home\site\wwwroot`; adjust for your paths accordingly:

1. Modify the `PythonHandler` entry in `web.config` so that the path matches the Python install location (see [IIS Configuration Reference \(iis.net\)](#) for exact details).

```

<system.webServer>
  <handlers>
    <add name="PythonHandler" path="*" verb="*" modules="FastCgiModule"
      scriptProcessor="c:\python36-32\python.exe|c:\python36-32\wfastcgi.py"
      resourceType="Unspecified" requireAccess="Script"/>
  </handlers>
</system.webServer>

```

2. Within the `<appSettings>` section of `web.config`, add keys for `WSGI_HANDLER`, `WSGI_LOG` (optional), and `PYTHONPATH`:

```

<appSettings>
  <add key="PYTHONPATH" value="c:\home\site\wwwroot"/>
  <!-- The handler here is specific to Bottle; see the next section. -->
  <add key="WSGI_HANDLER" value="app.wsgi_app()"/>
  <add key="WSGI_LOG" value="c:\home\LogFiles\wfastcgi.log"/>
</appSettings>

```

These `<appSettings>` values are available to your app as environment variables:

- The value for `PYTHONPATH` may be freely extended but must include the root of your app.
- `WSGI_HANDLER` must point to a WSGI app importable from your app.

- `WSGI_LOG` is optional but recommended for debugging your app.
3. Set the `WSGI_HANDLER` entry in `web.config` as appropriate for the framework you're using:

- **Bottle:** make sure that you have parentheses after `app.wsgi_app` as shown below. This is necessary because that object is a function (see `app.py`) rather than a variable:

```
<!-- Bottle apps only -->
<add key="WSGI_HANDLER" value="app.wsgi_app()"/>
```

- **Flask:** Change the `WSGI_HANDLER` value to `<project_name>.app` where `<project_name>` matches the name of your project. You can find the exact identifier by looking at the `from <project_name> import app` statement in the `runserver.py`. For example, if the project is named "FlaskAzurePublishExample", the entry would appear as follows:

```
<!-- Flask apps only: change the project name to match your app -->
<add key="WSGI_HANDLER" value="flask_iis_example.app"/>
```

- **Django:** Two changes are needed to `web.config` for Django projects. First, change the `WSGI_HANDLER` value to `django.core.wsgi.get_wsgi_application()` (the object is in the `wsgi.py` file):

```
<!-- Django apps only -->
<add key="WSGI_HANDLER" value="django.core.wsgi.get_wsgi_application()"/>
```

Second, add the following entry below the one for `WSGI_HANDLER`, replacing `DjangoAzurePublishExample` with the name of your project:

```
<add key="DJANGO_SETTINGS_MODULE" value="django_iis_example.settings" />
```

4. **Django apps only:** In the Django project's `settings.py` file, add your site URL domain or IP address to `ALLOWED_HOSTS` as shown below, replacing '1.2.3.4' with your URL or IP address, of course:

```
# Change the URL or IP address to your specific site
ALLOWED_HOSTS = ['1.2.3.4']
```

Failure to add your URL to the array results in the error **DisallowedHost at / Invalid HTTP_HOST header: '<site URL>'. You may need to add '<site URL>' to ALLOWED_HOSTS.**

Note that when the array is empty, Django automatically allows 'localhost' and '127.0.0.1', but adding your production URL removes those capabilities. For this reason you might want to maintain separate development and production copies of `settings.py`, or use environment variables to control the run time values.

Deploy to IIS or a Windows VM

With the correct `web.config` file in your project, you can publish to the computer running IIS by using the **Publish** command on the project's context menu in **Solution Explorer**, and selecting the option, **IIS, FTP, etc..** In this case, Visual Studio simply copies the project files to the server; you're responsible for all server-side configuration.

Edit Python code

4/9/2019 • 10 minutes to read • [Edit Online](#)

Because you spend much of your development time in the code editor, [Python support in Visual Studio](#) provides functionality to help you be more productive. Features include IntelliSense syntax highlighting, auto-completion, signature help, method overrides, search, and navigation.

The editor is also integrated with the **Interactive** window in Visual Studio, making it easy to exchange code between the two. See [Tutorial Step 3: Use the Interactive REPL window](#) and [Use the Interactive window - Send to Interactive command](#) for details.

For general documentation on editing code in Visual Studio, see [Features of the code editor](#). Also see [Outlining](#), which helps you stay focused on particular sections of your code.

You can also use the Visual Studio **Object Browser** ([View > Other Windows > Object Browser](#) or **Ctrl+W > J**) for inspecting Python classes defined in each module and the functions defined in those classes.

IntelliSense

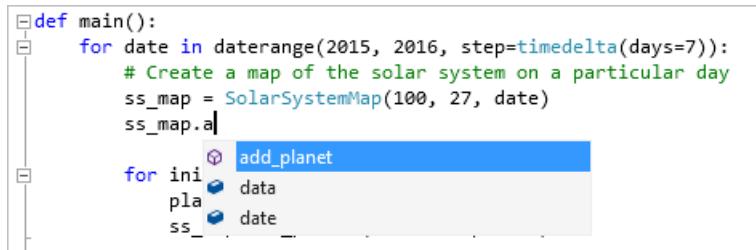
IntelliSense provides [completions](#), [signature help](#), [quick info](#), and [code coloring](#). Visual Studio 2017 versions 15.7 and later also support [type hints](#).

To improve performance, IntelliSense in Visual Studio 2017 version 15.5 and earlier depends on a completion database that's generated for each Python environment in your project. Databases may need refreshing if you add, remove, or update packages. Database status is shown in the **Python Environments** window (a sibling of **Solution Explorer**) on the **IntelliSense** tab (see [Environments window reference](#)).

Visual Studio 2017 version 15.6 and later uses a different means to provide IntelliSense completions that are not dependent on the database.

Completions

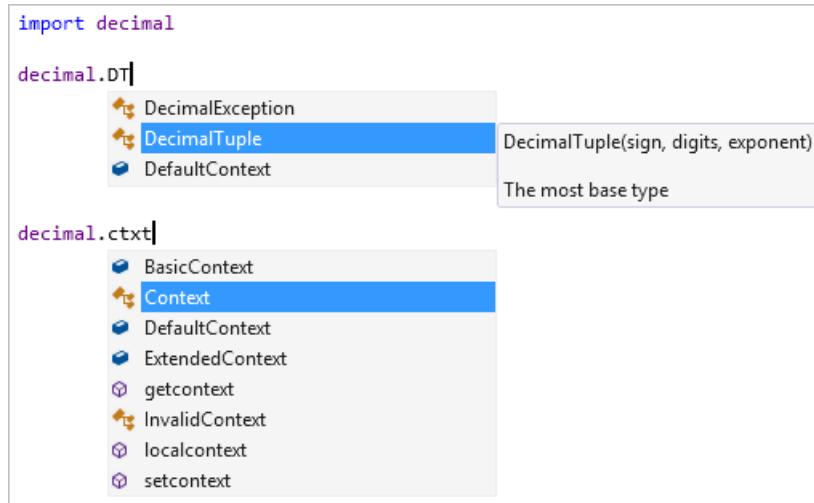
Completions appear as statements, identifiers, and other words that may be appropriately entered at the current location in the editor. What's shown in the list is based on context and is filtered to omit incorrect or distracting options. Completions are often triggered by typing different statements (such as `import`) and operators (including a period), but you can have them appear at anytime by typing **Ctrl+J > Space**.



When a completion list is open, you can search for the completion you want using the arrow keys, the mouse, or by continuing to type. As you type more letters, the list is further filtered to show likely completions. You can also use shortcuts such as:

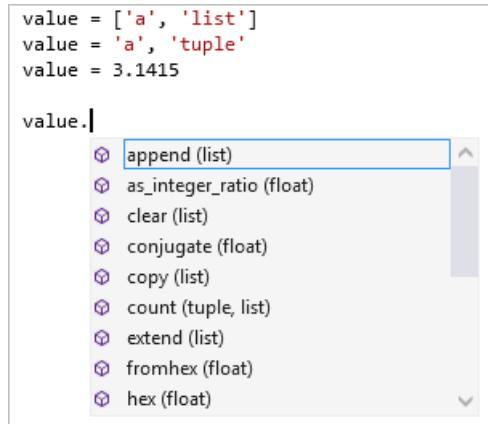
- Typing letters that are not at the start of the name, such as 'parse' to find 'argparse'
- Typing only letters that are at the start of words, such as 'abc' to find 'AbstractBaseClass' or 'air' to find 'as_integer_ratio'
- Skipping letters, such as 'b64' to find 'base64'

Some examples:



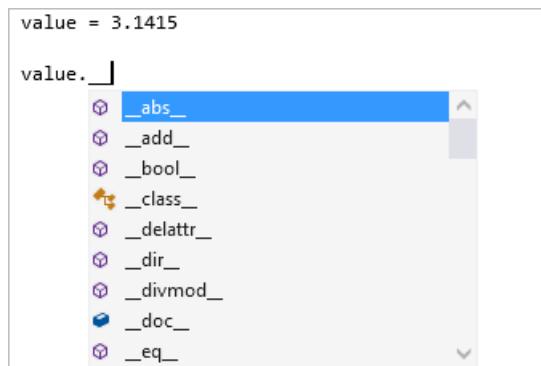
The screenshot shows two code completion dropdowns. The first, for `decimal.DT`, lists `DecimalException`, `DecimalTuple` (which is highlighted), and `DefaultContext`. A tooltip for `DecimalTuple` says "DecimalTuple(sign, digits, exponent)" and "The most base type". The second, for `decimal.ctx`, lists `BaseContext`, `Context` (highlighted), `DefaultContext`, `ExtendedContext`, `getcontext`, `InvalidContext`, `localcontext`, and `setcontext`.

Member completions appear automatically when you type a period after a variable or value, along with the methods and attributes of the potential types. If a variable could be more than one type, the list includes all possibilities from all types, with extra information to indicate which types support each completion. Where a completion is supported by all possible types, it is shown without annotation.



The screenshot shows a code completion dropdown for `value.`. It lists various methods and attributes, with `append (list)` highlighted. Other items include `as_integer_ratio (float)`, `clear (list)`, `conjugate (float)`, `copy (list)`, `count (tuple, list)`, `extend (list)`, `fromhex (float)`, and `hex (float)`.

By default, "dunder" members (members beginning and ending with a double underscore) are not shown. In general, such members should not be accessed directly. If you need one, however, typing the leading double underscore adds these completions to the list:



The screenshot shows a code completion dropdown for `value._`. It lists dunder methods: `_abs_`, `_add_`, `_bool_`, `_class_` (highlighted), `_delattr_`, `_dir_`, `_divmod_`, `_doc_`, and `_eq_`.

The `import` and `from ... import` statements display a list of modules that can be imported. With `from ... import`, the list includes members that can be imported from the specified module.

The screenshot shows two code snippets in a Python editor. The first snippet is `import decimal`, and the second is `from decimal import Decimal`. A completion dropdown is open over the `decimal` import, listing `_decimal`, `_codecs_iso2022`, `_decimal`, and `_pydecimal`. When the cursor moves to the second snippet, the completion dropdown changes to list `Decimal`, `DecimalException`, and `DecimalTuple`.

The `raise` and `except` statements display lists of classes likely to be error types. The list may not include all user-defined exceptions, but helps you find suitable built-in exceptions quickly:

Typing `@` starts a decorator and shows potential decorators. Many of these items aren't usable as decorators; check the library documentation to determine which to use.

The screenshot shows a code block starting with `class MyClass:`. A completion dropdown is open over the `@` symbol, listing `classmethod`, `compile`, `complex`, and `ConnectionAbortedError`. A tooltip for `classmethod` provides the definition: `classmethod(function) -> method` and the note: `Convert a function to be a class method.` Another tooltip notes: `A class method receives the class as implicit first argument,`.

TIP

You can configure the behavior of completions through **Tools > Options > Text Editor > Python > Advanced**. Among these, **Filter list based on search string** applies filtering of completion suggestions as you type (default is checked), and **Member completion displays intersection of members** shows only completions that are supported by all possible types (default is unchecked). See [Options - completion results](#).

Type hints

Visual Studio 2017 version 15.7 and later.

"Type hints" in Python 3.5+ ([PEP 484](#) ([python.org](#)) is an annotation syntax for functions and classes that indicate the types of arguments, return values, and class attributes. IntelliSense displays type hints when you hover over functions calls, arguments, and variables that have those annotations.

In the example below, the `Vector` class is declared as `List[float]`, and the `scale` function contains type hints for both its arguments and return value. Hovering over a call to that function shows the type hints:

```

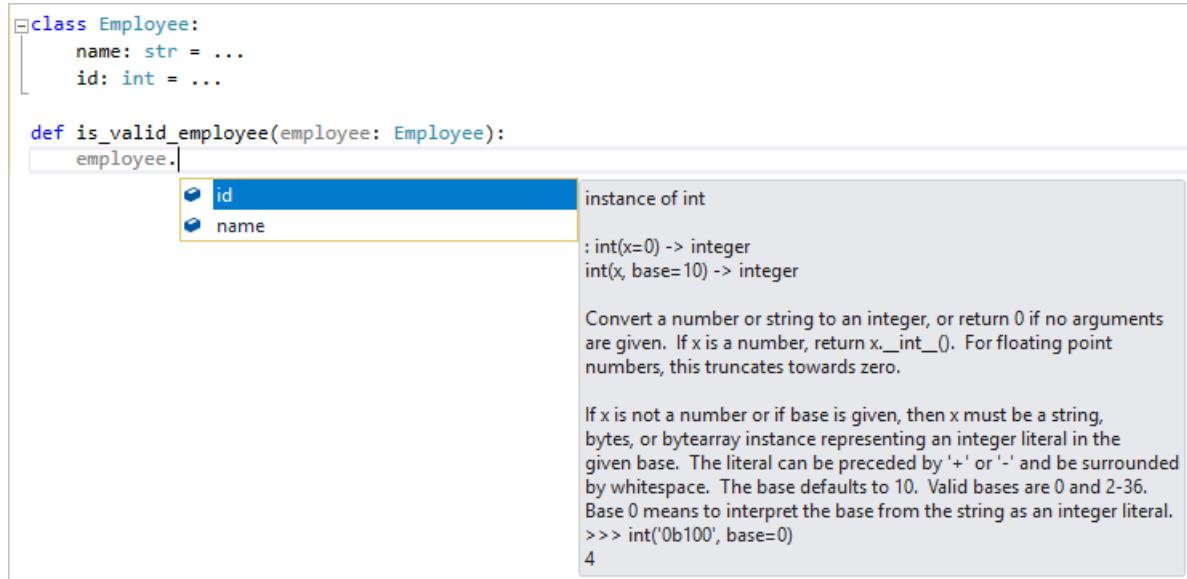
from typing import List
Vector = List[float]

def scale(scalar: float, vector: Vector) -> List:
    return [scalar * num for num in vector]

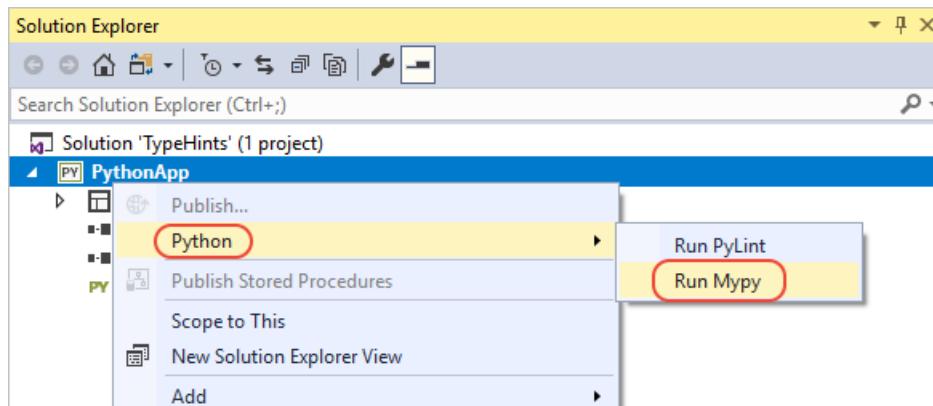
new_vector = scale(2.0, [1.0, -4.2, 5.4])
    scale: def app.scale(scalar : float, vector : list, list[float]) -> list

```

In the following example, you can see how the annotated attributes of the `Employee` class appear in the IntelliSense completion popup for an attribute:



It's also helpful to validate type hints throughout your project, because errors won't normally appear until run time. For this purpose, Visual Studio integrates the industry standard MyPy tool through the context menu command **Python > Run Mypy in Solution Explorer**:



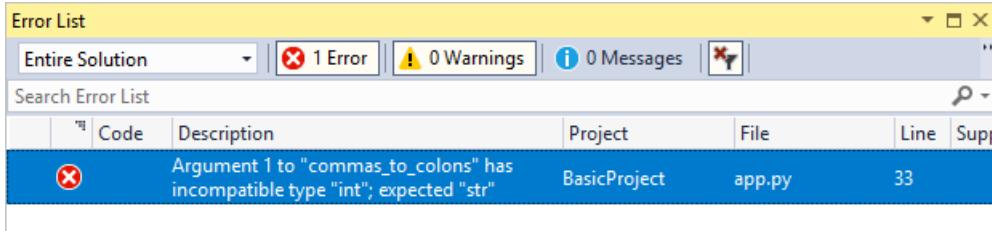
Running the command prompts you to install the `mypy` package, if needed. Visual Studio then runs `mypy` to validate type hints in every Python file in the project. Errors appear in the Visual Studio **Error List** window. Selecting an item in the window navigates to the appropriate line in your code.

As a simple example, the following function definition contains a type hint to indicate that the `input` argument is type `str`, whereas the call to that function attempts to pass an integer:

```
def commas_to_colons(input: str):
    items = input.split(',')
    items = [x.strip() for x in items]
    return ':' .join(items)

commas_to_colons(1)
```

Using the **Run Mypy** command on this code generates the following error:



TIP

For versions of Python before 3.5, Visual Studio also displays type hints that you supply through *Typeshed stub files (.pyi)*. You can use stub files whenever you don't want to include type hints directly in your code, or when you want to create type hints for a library that doesn't use them directly. For more information, see [Create stubs for Python modules](#) in the mypy project wiki.

At present, Visual Studio doesn't support type hints in comments.

TIP

For versions of Python before 3.5, Visual Studio also displays type hints that you supply through *Typeshed stub files (.pyi)*. You can use stub files whenever you don't want to include type hints directly in your code, or when you want to create type hints for a library that doesn't use them directly. For more information, see [Create stubs for Python modules](#) in the mypy project wiki.

Visual Studio includes a bundled set of Typeshed files for Python 2 and 3, so additional downloads aren't necessary. However, if you want to use a different set of files, you can specify the path in the **Tools > Options > Python > Language Server** options. See [Options - Language Server](#).

At present, Visual Studio doesn't support type hints in comments.

Signature help

When writing code that calls a function, signature help appears when you type the opening `(` and displays available documentation and parameter information. You can also make it appear with **Ctrl+Shift+Space** inside a function call. The information displayed depends on the documentation strings in the function's source code, but includes any default values.

```
import decimal

decimal.Decimal()
Decimal(value: str = '0', context: Context = None)
Create a decimal point instance.
...
```

TIP

To disable signature help, go to **Tools > Options > Text Editor > Python > General** and clear **Statement completion > Parameter information**.

Quick info

Hovering the mouse pointer over an identifier displays a Quick Info tooltip. Depending on the identifier, Quick Info may display the potential values or types, any available documentation, return types, and definition locations:

```
from decimal import *
d = Decimal(123)
d: Decimal instance
d.conjugate
d.conjugate: method conjugate of Decimal objects -> Decimal instance
```

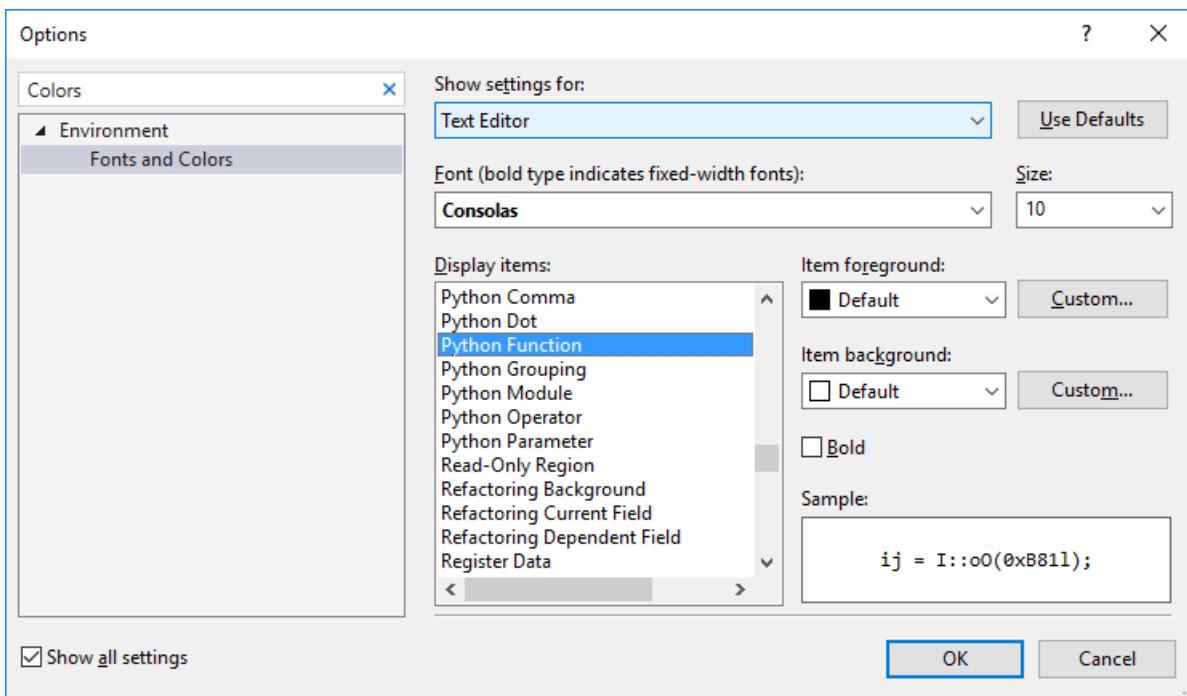
Code coloring

Code coloring uses information from code analysis to color variables, statements, and other parts of your code. For example, variables that refer to modules or classes may be shown in a different color than functions or other values, and parameter names appear in a different color than local or global variables. (By default, functions are not shown in bold):

```
Module name
from xml.etree.cElementTree import ElementTree, ParseError
Class name
def parse_xml_from_file(filename):
Function name
    try:
        Parameter name
        data = ElementTree().parse(filename)
    except ParseError:
        print('Failed to parse file {}'.format(filename))

filename = r'C:\Data.xml'
parse_xml_from_file(filename)
```

To customize the colors, go to **Tools > Options > Environment > Fonts and Colors** and modify the **Python** entries in the **Display items** list:



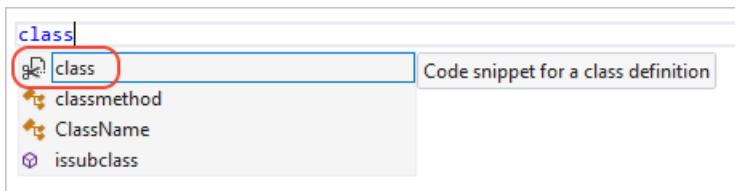
TIP

To disable code coloring, go to **Tools > Options > Text Editor > Python > Advanced** and clear **Miscellaneous Options > Color names based on type**. See [Options - Miscellaneous options](#).

Code snippets

Code snippets are fragments of code that can be inserted into your files by typing a shortcut and pressing **Tab**, or using the **Edit > IntelliSense > Insert Code Snippet** and **Surround With** commands, selecting **Python**, then selecting the desired snippet.

For example, `class` is a shortcut for a code snippet that inserts a class definition. You see the snippet appear in the auto-completion list when you type `class`:

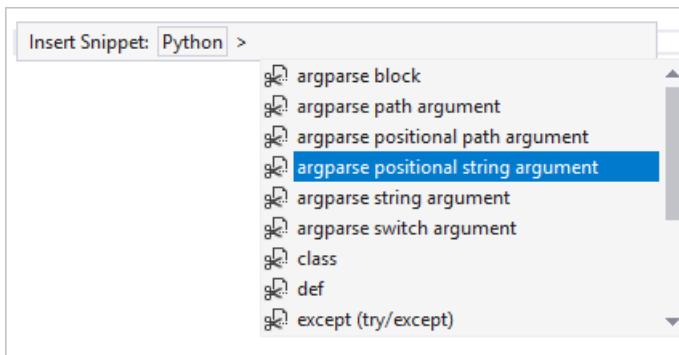


Pressing **Tab** generates the rest of the class. You can then type over the name and bases list, moving between the highlighted fields with **Tab**, then press **Enter** to begin typing the body.

```
class ClassName(object):
    pass
```

Menu commands

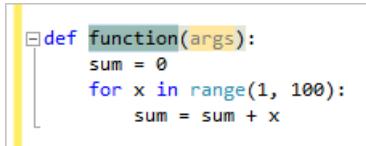
When you use the **Edit > IntelliSense > Insert Code Snippet** menu command, you first select **Python**, then select a snippet:



The **Edit > IntelliSense > Surround With** command, similarly, places the current selection in the text editor inside a chosen structural element. For example, suppose you had a bit of code like the following:

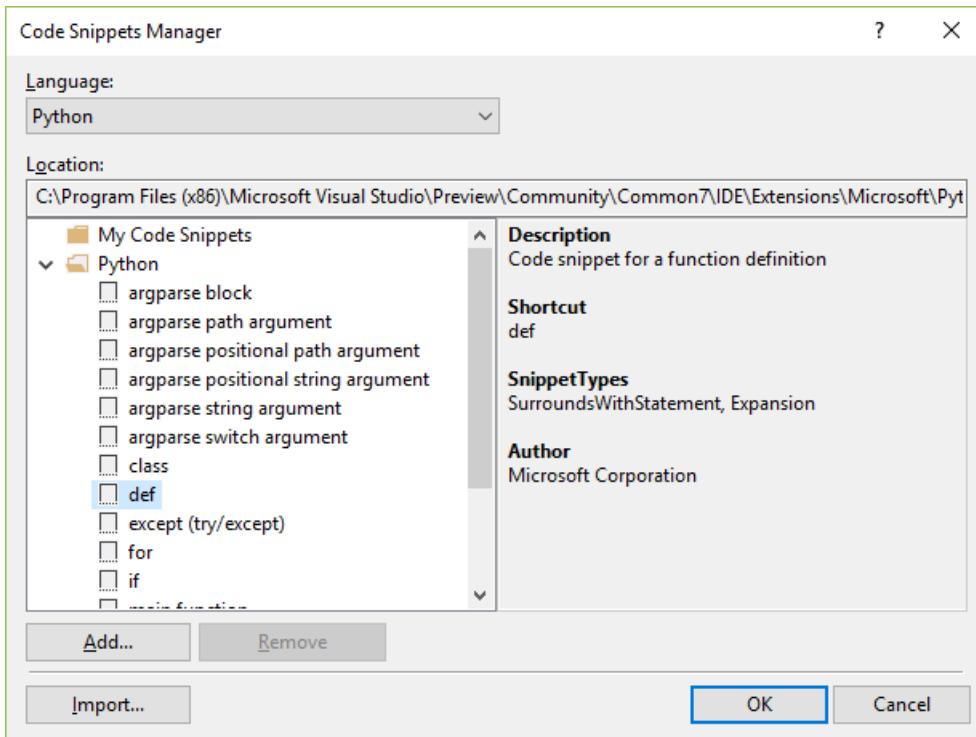
```
sum = 0
for x in range(1, 100):
    sum = sum + x
```

Selecting this code and choosing the **Surround With** command displays a list of available snippets. Choosing **def** from the list places the selected code within a function definition, and you can use the **Tab** key to navigate between the highlighted function name and arguments:



Examine available snippets

You can see the available code snippets in the **Code Snippets Manager**, opened by using **Tools > Code Snippets Manager** menu command and selecting **Python** as the language:



To create your own snippets, see [Walkthrough: Create a code snippet](#).

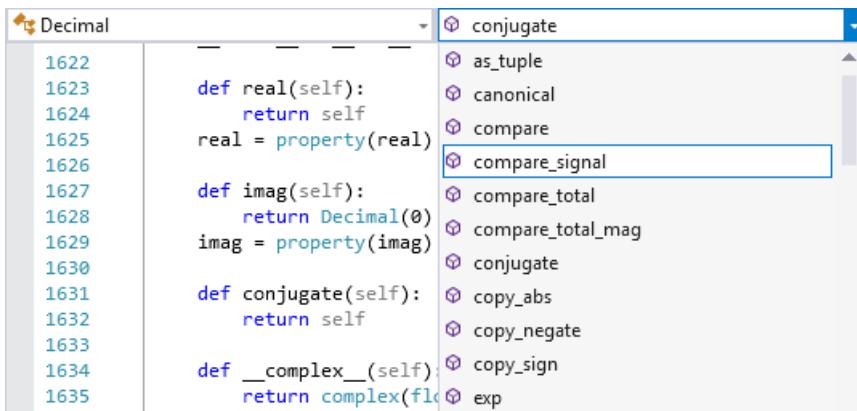
If you write a great code snippet that you'd like to share, feel free to post it in a gist and [let us know](#). We may be able to include it in a future release of Visual Studio.

Navigate your code

Python support in Visual Studio provides several means to quickly navigate within your code, including libraries for which source code is available: the [navigation bar](#), [Go To Definition](#), [Navigate To](#), and [Find All References](#). You can also use the Visual Studio [Object Browser](#).

Navigation bar

The navigation bar is displayed at the top of each editor window and includes a two-level list of definitions. The left drop-down contains top-level class and function definitions in the current file; the right drop-down displays a list of definitions within the scope shown in the left. As you move around in the editor, the lists update to show your current context, and you can also select an entry from these lists to jump directly to.

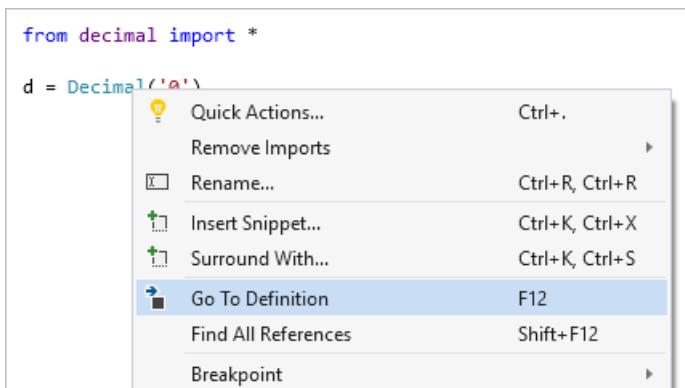


TIP

To hide the navigation bar, go to **Tools > Options > Text Editor > Python > General** and clear **Settings > Navigation bar**.

Go To Definition

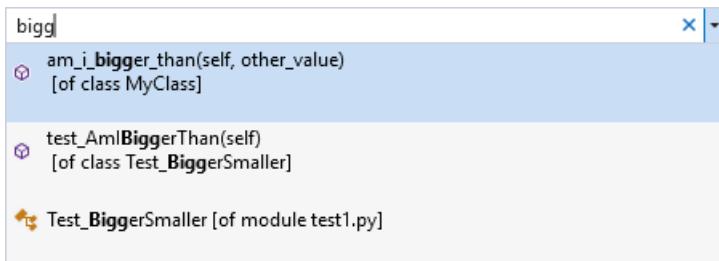
Go To Definition quickly jumps from the use of an identifier (such as a function name, class, or variable), to the source code where it's defined. You invoke it by right-clicking an identifier and selecting **Go To Definition** or, by placing the caret in the identifier and pressing **F12**. It works across your code and external libraries provided that source code is available. If library source code is not available, **Go To Definition** jumps to the relevant `import` statement for a module reference, or displays an error.



Navigate To

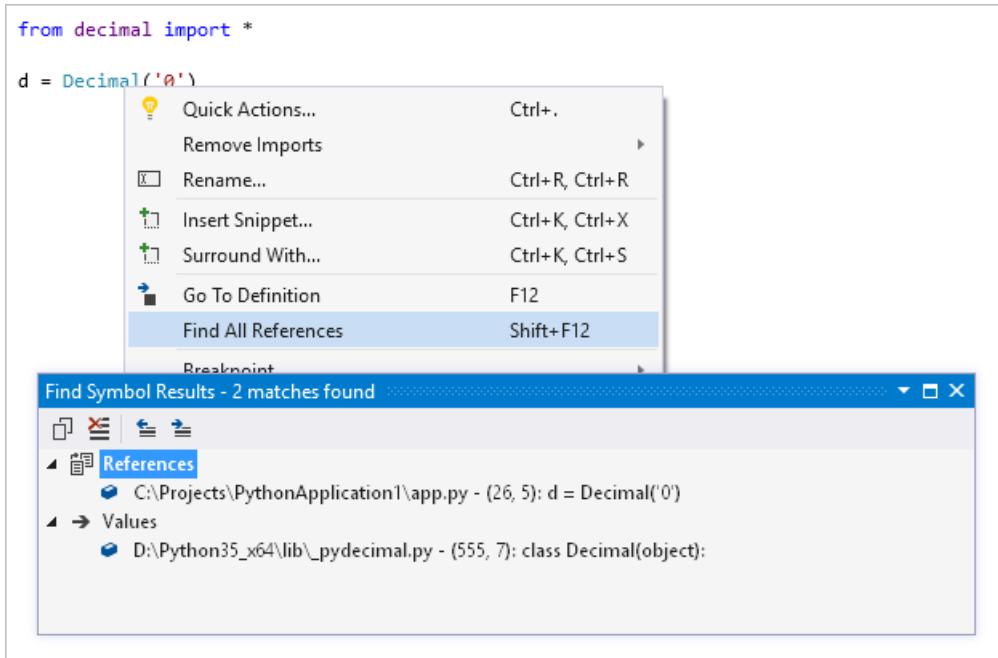
The **Edit > Navigate To** command (**Ctrl+Shift+F**) displays a search box in the editor where you can type any string and see possible matches in your code that defines a function, class, or variable containing that string. This feature provides a similar capability as **Go To Definition** but without having to locate a use of an identifier.

Double-clicking any name, or selecting with arrow keys and **Enter**, navigates to the definition of that identifier.



Find All References

Find All References is a helpful way of discovering where any given identifier is both defined and used, including imports and assignments. You invoke it by right-clicking an identifier and selecting **Find All References**, or by placing the caret in the identifier and pressing **Shift+F12**. Double-clicking an item in the list navigates to its location.



See also

- [Formatting](#)
- [Refactoring](#)
- [Use a linter](#)

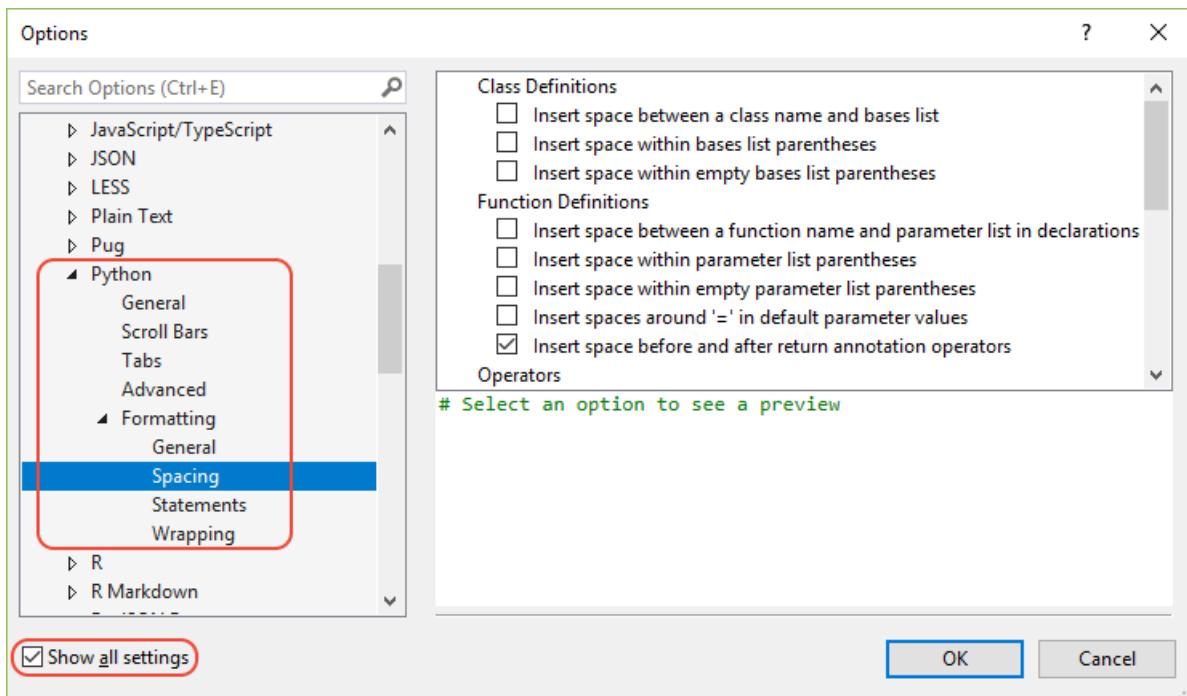
Format Python code

4/9/2019 • 3 minutes to read • [Edit Online](#)

Visual Studio lets you quickly reformat code to match pre-configured formatting options.

- To format a selection: select **Edit > Advanced > Format Selection** or press **Ctrl+E > F**.
- To format the whole file: select **Edit > Advanced > Format Document** or press **Ctrl+E > D**.

Options are set through **Tools > Options > Text Editor > Python > Formatting** and its nested tabs. You need to select **Show all settings** for these options to appear:



Formatting options by default are set to match a superset of the [PEP 8 style guide](#). The **General** tab determines when formatting is applied; settings for the other three tabs are described in this article.

[Python support in Visual Studio](#) also adds the useful **Fill Comment Paragraph** command to the **Edit > Advanced** menu as described in a later section.

Spacing

Spacing controls where spaces are inserted or removed around various language constructs. Each option has three possible values:

- Checked: ensures the spacing is applied.
- Cleared: removes any spacing.
- Indeterminate: leaves original formatting in place.

Examples for the various options are provided in the following tables:

CLASS DEFINITIONS OPTION	CHECKED	CLEARED
Insert space between a class declaration's name and bases list	<code>class X (object): pass</code>	<code>class X(object): pass</code>

CLASS DEFINITIONS OPTION	CHECKED	CLEARED
Insert space within bases list parentheses	<code>class X(object): pass</code>	<code>class X(object): pass</code>
Insert space within empty bases list parentheses	<code>class X(): pass</code>	<code>class X(): pass</code>

FUNCTION DEFINITIONS OPTION	CHECKED	CLEARED
Insert space between a function declaration's name and parameter list	<code>def X (): pass</code>	<code>def X(): pass</code>
Insert space within parameter list parentheses	<code>def X(a, b): pass</code>	<code>def X(a, b): pass</code>
Insert space within empty parameter list parentheses	<code>def X(): pass</code>	<code>def X(): pass</code>
Insert spaces around '=' in default parameter values	<code>includes X(a = 42): pass</code>	<code>includes X(a=42): pass</code>
Insert space before and after return annotation operators	<code>includes X() -> 42: pass</code>	<code>includes X()->42: pass</code>

OPERATORS OPTION	CHECKED	CLEARED
Insert spaces around binary operators	<code>a + b</code>	<code>a+b</code>
Insert spaces around assignments	<code>a = b</code>	<code>a=b</code>

EXPRESSION SPACING OPTION	CHECKED	CLEARED
Insert space between a function call's name and argument list	<code>X ()</code>	<code>X()</code>
Insert space within empty argument list parentheses	<code>X()</code>	<code>X()</code>
Insert space within argument list parentheses	<code>X(a, b)</code>	<code>X(a, b)</code>
Insert space within parentheses of expression	<code>(a)</code>	<code>(a)</code>
Insert space within empty tuple parentheses	<code>()</code>	<code>()</code>

EXPRESSION SPACING OPTION	CHECKED	CLEARED
Insert space within tuple parentheses	(a, b)	(a, b)
Insert space within empty square brackets	[]	[]
Insert spaces within square brackets of lists	[a, b]	[a, b]
Insert space before open square bracket	x [i]	x[i]
Insert space within square brackets	x[i]	x[i]

Statements

The **Statements** options control automatic rewriting of various statements into more Pythonic forms.

OPTION	BEFORE FORMATTING	AFTER FORMATTING
Place imported modules on new line	import sys, pickle	import sys import pickle
Remove unnecessary semicolons	x = 42;	x = 42
Place multiple statements on new lines	x = 42; y = 100	x = 42 y = 100

Wrapping

Wrapping lets you set the **Maximum comment width** (default is 80). If the **Wrap comments that are too wide** option is set, Visual Studio reformats comments to not exceed that maximum width.

```
# Wrapped to 40 columns
# There should be one-- and preferably
# only one --obvious way to do it.
```

```
# Not-wrapped:
# There should be one-- and preferably only one --obvious way to do it.
```

Fill Comment Paragraph command

Edit > Advanced > Fill Comment Paragraph (Ctrl+E > P) reflows and formats comment text, combining short lines together and breaking up long ones.

For example:

```
# foo  
# bar  
# baz
```

changes to:

```
# foo bar baz
```

```
# This is a very long  
long comment
```

changes to:

```
# This is a very long  
# long long long long long long long comment
```

Refactor Python code

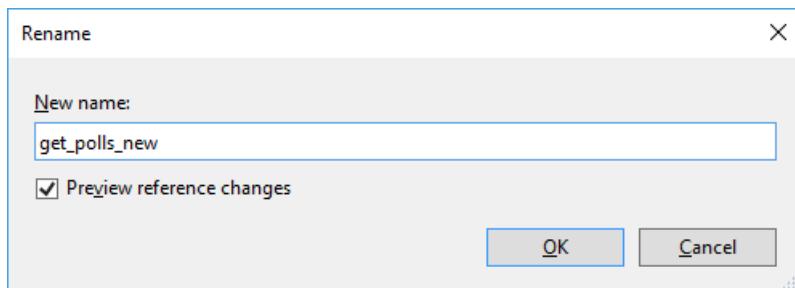
4/9/2019 • 3 minutes to read • [Edit Online](#)

Visual Studio provides several commands for automatically transforming and cleaning up your Python source code:

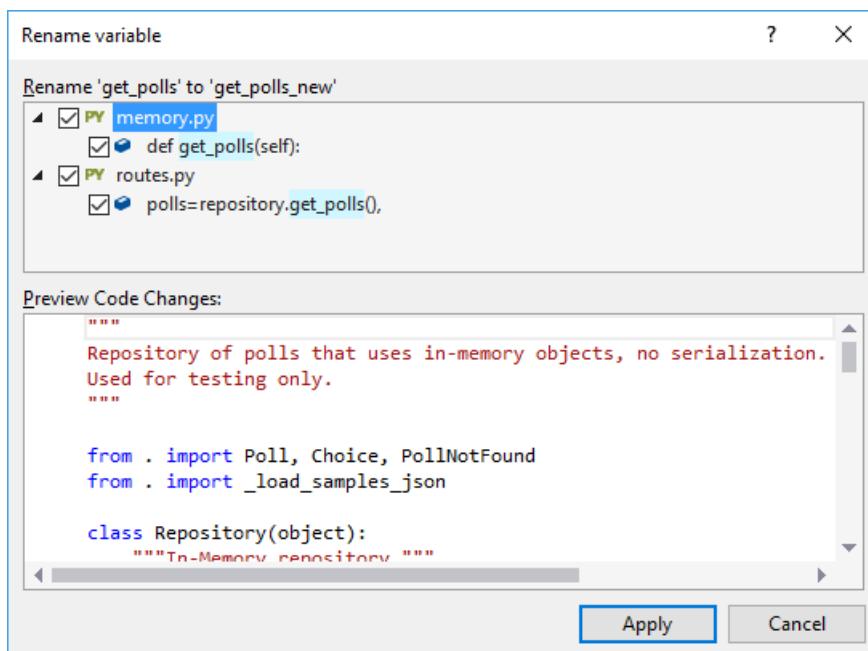
- [Rename](#) renames a selected class, method, or variable name
- [Extract method](#) creates a new method from the selected code
- [Add import](#) provides a smart tag to add a missing import
- [Remove unused imports](#) removes unused imports

Rename

1. Right-click the identifier you wish to rename and select **Rename**, or place the caret in that identifier and select the **Edit > Refactor > Rename** menu command (**F2**).
2. In the **Rename** dialog that appears, enter the new name for the identifier and select **OK**:



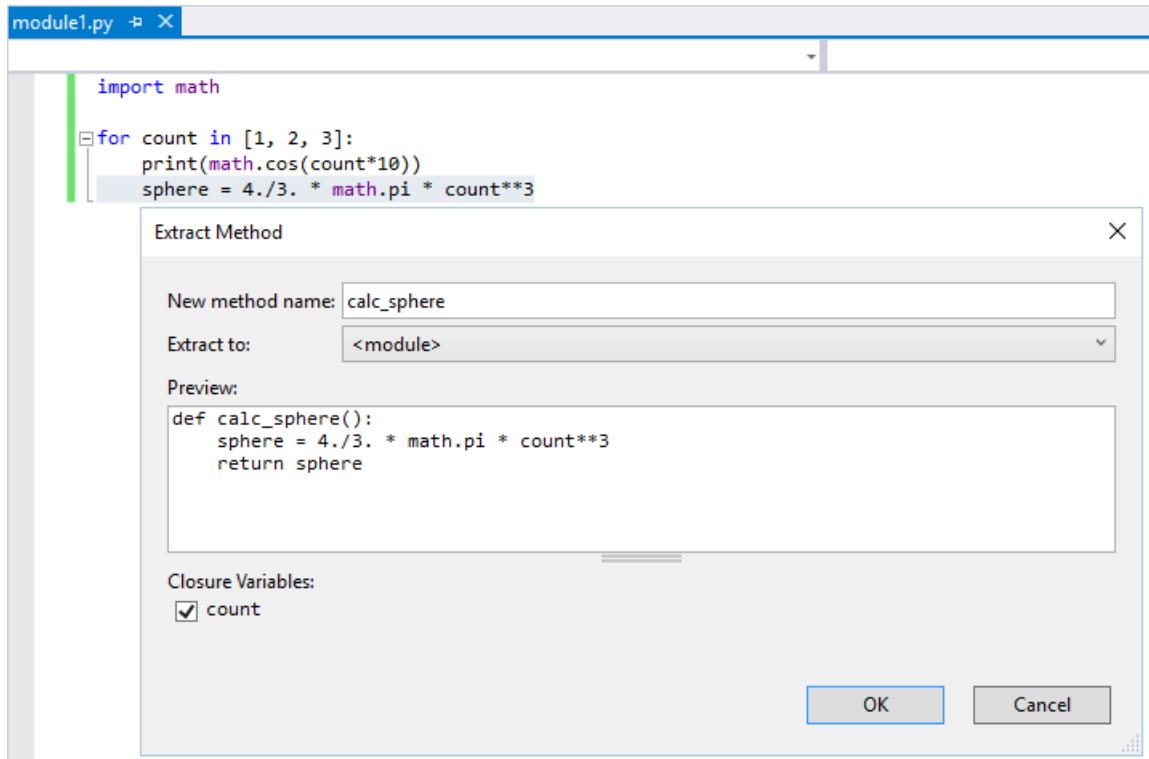
3. In the next dialog, select the files and instances in your code to which to apply the renaming; select any individual instance to preview the specific change:



4. Select **Apply** to make the changes to your source code files. (This action can be undone.)

Extract method

1. Select the lines of code or the expression to extract into a separate method.
2. Select the **Edit > Refactor > Extract method** menu command or type **Ctrl+R > M**.
3. In the dialog that appears, enter a new method name, indicate where to extract it to, and select any closure variables. Variables not selected for closure are turned into method arguments:



4. Select **OK** and the code is modified accordingly:

```
module1.py*  # X
import math

def calc_sphere(count):
    sphere = 4./3. * math.pi * count**3
    return sphere

for count in [1, 2, 3]:
    print(math.cos(count*10))
    sphere = calc_sphere(count)
```

Add import

When you place the caret on an identifier that lacks type information, Visual Studio provides a smart tag (the lightbulb icon to the left of the code) whose commands add the necessary `import` or `from ... import` statement:

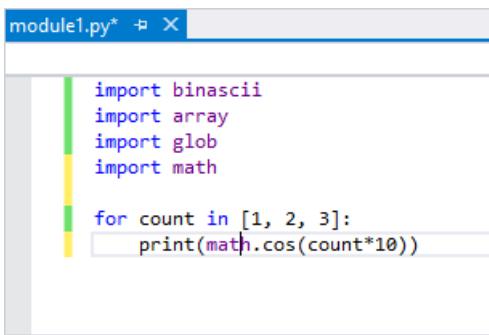
```
module1.py*  # X
import binascii
import array
import glob

for count in [1, 2, 3]:
    print(math.cos(count*10))

from module1 import math
import math
```

Visual Studio offers `import` completions for top-level packages and modules in the current project and the standard library. Visual Studio also offers `from ... import` completions for submodules and subpackages as well

as module members. Completions include functions, classes, or exported data. Selecting either option adds the statement to at the top of the file after other imports, or into an existing `from ... import` statement if the same module is already imported.



A screenshot of the Visual Studio code editor showing a Python file named "module1.py". The code contains imports for `binascii`, `array`, `glob`, and `math`. A cursor is positioned inside a `for` loop, specifically on the `print` statement. A completion dropdown is open, showing options like `math.cos` and `math.sin`.

Visual Studio attempts to filter out members that aren't actually defined in a module, such as modules that are imported into another but aren't children of the module doing the importing. For example, many modules use `import sys` rather than `from xyz import sys`, so you don't see a completion for importing `sys` from other modules even if the modules are missing an `__all__` member that excludes `sys`.

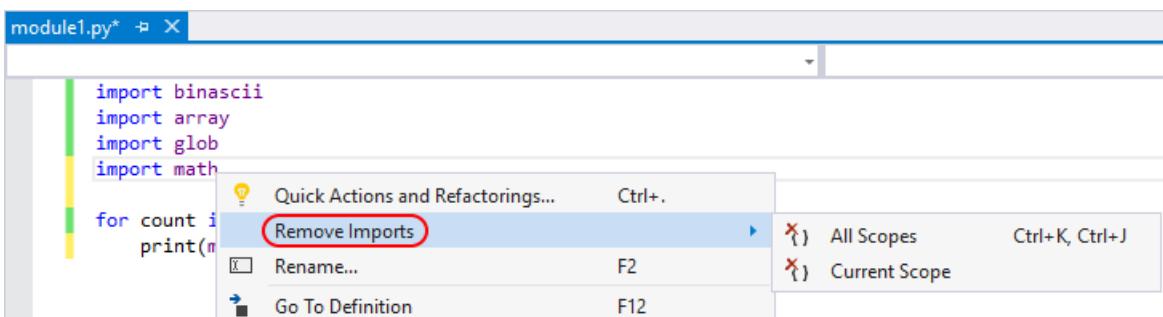
Similarly, Visual Studio filters functions that are imported from other modules or from the built-in namespace. For example if a module imports the `settrace` function from the `sys` module, then in theory you could import it from that module. But it's best to use `import settrace from sys` directly, and so Visual Studio offers that statement specifically.

Finally, if something would normally be excluded but has other values that would be included (because the name was assigned a value in the module, for example), Visual Studio still excludes the import. This behavior assumes that the value shouldn't be exported because it is defined in another module, and thus the additional assignment is likely to be a dummy value that is also not exported.

Remove unused imports

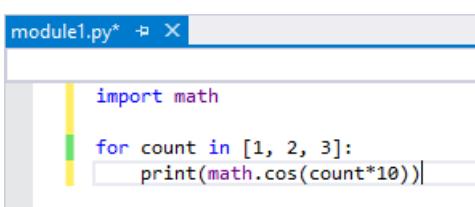
When writing code, it's easy to end up with `import` statements for modules that aren't being used at all. Because Visual Studio analyzes your code, it can automatically determine whether an `import` statement is needed by looking at whether the imported name is used within the scope below where the statement occurs.

Right-click anywhere in the editor and select **Remove Imports**, which gives you options to remove from **All Scopes** or just the **Current Scope**:



A screenshot of the Visual Studio context menu for a Python file named "module1.py". The menu includes options like "Quick Actions and Refactorings...", "Remove Imports" (which is highlighted with a red circle), "Rename...", and "Go To Definition". Below the menu, there are two buttons: "All Scopes" (Ctrl+K, Ctrl+J) and "Current Scope".

Visual Studio then makes the appropriate changes to the code:



A screenshot of the Visual Studio code editor showing the simplified Python code. The `import binascii`, `import array`, and `import glob` statements have been removed, leaving only the `import math` statement and the original code block.

Note that Visual Studio does not account for control flow; using a name before an `import` statement is treated as

if the name was in fact used. Visual Studio also ignores all `from __future__` imports, imports that are performed inside of a class definition, as well from `from ... import *` statements.

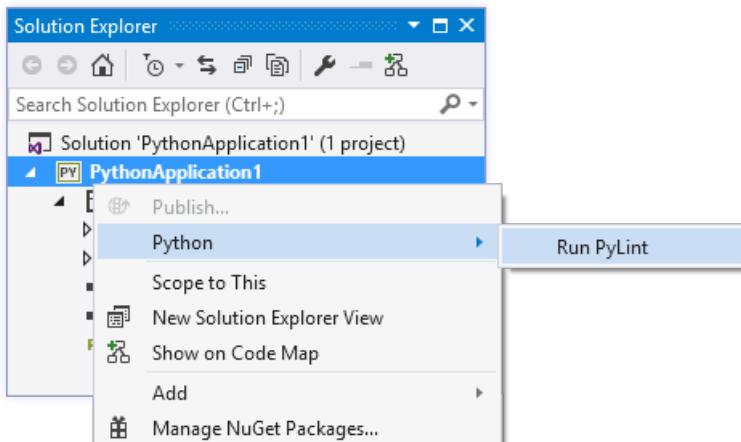
Use PyLint to check Python code

4/9/2019 • 2 minutes to read • [Edit Online](#)

PyLint, a widely used tool that checks for errors in Python code and encourages good Python coding patterns, is integrated into Visual Studio for Python projects.

Run PyLint

Just right-click a Python project in **Solution Explorer** and select **Python > Run PyLint**:



Using this command prompts you to install PyLint into your active environment if it's not already present.

PyLint warnings and errors appear in the **Error List** window:

A screenshot of Visual Studio showing a Python file 'hello.py' and the Error List window. The code in 'hello.py' is as follows:

```

from math import sin, cos, radians
import sys

def make_dot_string(x):
    return ' ' * int(10 * cos(radians(x))) + 'o'

def main():
    for i in range(360 * 5):
        s = make_dot_string(i)
        print(s)

main()

```

The Error List window shows the following results:

Code	Description	Project	File	Line
!	Unused sin imported from math [W:unused-import]	PythonApplication1	hello.py	1
!	Missing module docstring [C:missing-docstring]	PythonApplication1	hello.py	1
!	Unused import sys [W:unused-import]	PythonApplication1	hello.py	2
!	Invalid argument name "x" [C:invalid-name]	PythonApplication1	hello.py	4
!	Missing function docstring [C:missing-docstring]	PythonApplication1	hello.py	4
!	Missing function docstring [C:missing-docstring]	PythonApplication1	hello.py	7
!	Invalid variable name "s" [C:invalid-name]	PythonApplication1	hello.py	9

Double-clicking an error takes you directly to the source code that generated the issue.

TIP

See the [PyLint features reference](#) for a detailed list of all the PyLint output messages.

Set PyLint command-line options

The [command-line options](#) section of the PyLint documentation describes how to control PyLint's behavior through a `.pylintrc` configuration file. Such a file can be placed in the root of a Python project in Visual Studio or elsewhere depending on how widely you want those settings applied (see the [command-line options](#) for details).

For example, to suppress the "missing docstring" warnings shown in the previous image with a `.pylintrc` file in a project, do the steps:

1. On the command line, navigate to your project root (which has your `.pyproj` file) and run the following command to generate a commented configuration file:

```
pylint --generate-rcfile > .pylintrc
```

2. In Visual Studio Solution Explorer, right-click your project, select **Add** > **Existing Item**, navigate to the new `.pylintrc` file, select it, and select **Add**.
3. Open the file for editing, which has several settings you can work with. To disable a warning, locate the `[MESSAGES CONTROL]` section, then locate the `disable` setting in that section. There's a long string of specific messages, to which you can append whichever warnings you want. In the example here, append `,missing-docstring` (including the delineating comma).
4. Save the `.pylintrc` file and run PyLint again to see that the warnings are now suppressed.

TIP

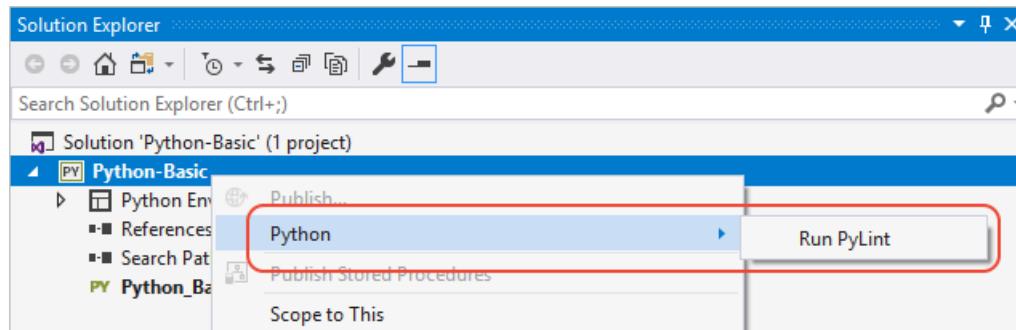
To use a `.pylintrc` file from a network share, create an environment variable named `PYLINTRC` with the value of the filename on the network share using a UNC path or a mapped drive letter. For example, `PYLINTRC=\\myshare\\python\\.pylintrc`.

Define custom commands for Python projects

4/9/2019 • 13 minutes to read • [Edit Online](#)

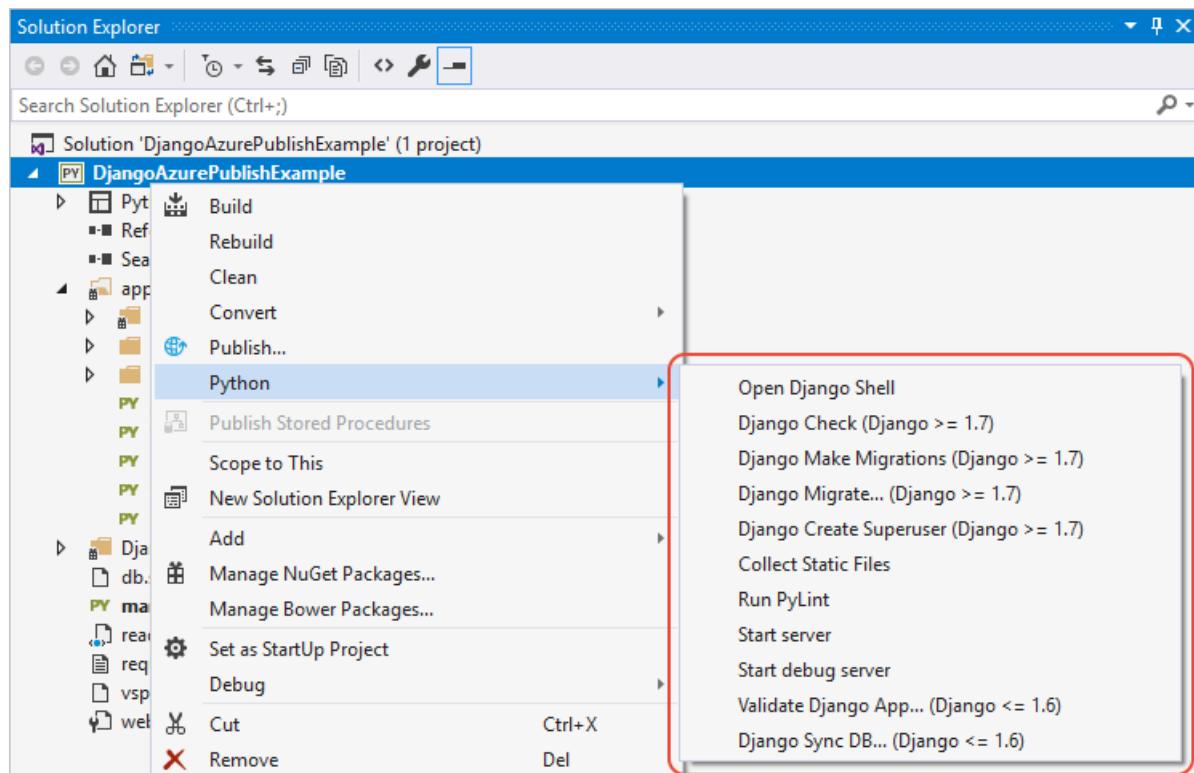
In the process of working with your Python projects, you may find yourself switching to a command window to run specific scripts or modules, run pip commands, or run some other arbitrary tool. To improve your workflow, you can add custom commands to the **Python** submenu in the Python project context menu. Those commands can run in a console window or in the Visual Studio **Output** window. You can also use regular expressions to instruct Visual Studio how to parse errors and warnings from the command's output.

By default, that menu contains only the single **Run PyLint** command:



Custom commands appear in this same context menu. Custom commands are added to a project file directly, where they apply to that individual project. You can also define custom commands in a *.targets* file that can easily be imported into multiple project files.

Certain Python project templates in Visual Studio already add custom commands of their own using their *.targets* file. For example, the Bottle Web Project and Flask Web Project templates both add two commands, **Start server** and **Start debug server**. The Django Web Project template adds these same commands plus quite a few more:



Each custom command can refer to a Python file, a Python module, inline Python code, an arbitrary executable, or a pip command. You can also specify how and where the command runs.

TIP

Whenever you make changes to a project file in a text editor, it's necessary to reload the project in Visual Studio to apply those changes. For example, you must reload a project after adding custom command definitions for those commands to appear on the project's context menu.

As you may know, Visual Studio provides a means to edit the project file directly. You first right-click the project file and select **Unload project**, then right-click again and select **Edit <project-name>** to open the project in the Visual Studio editor. You then make and save edits, right-click the project once more, and select **Reload project**, which also prompts you to confirm closing the project file in the editor.

When developing a custom command, however, all these clicks can become tedious. For a more efficient workflow, load the project in Visual Studio and also open the *.pyproj* file in a separate editor altogether (such as another instance of Visual Studio, Visual Studio Code, Notepad, etc.). When you save changes in the editor and switch to Visual Studio, Visual Studio detects changes and asks whether to reload the project (**The project <name> has been modified outside the environment.**). Select **Reload** and your changes are immediately applied in just one step.

Walkthrough: Add a command to a project file

To familiarize yourself with custom commands, this section walks through a simple example that runs a project's startup file directly using *python.exe*. (Such a command is effectively the same as using **Debug > Start without Debugging**.)

1. Create a new project named "Python-CustomCommands" using the **Python Application** template. (See [Quickstart: Create a Python project from a template](#) for instructions if you're not already familiar with the process.)
2. In *Python_CustomCommands.py*, add the code `print("Hello custom commands")`.
3. Right-click the project in **Solution Explorer**, select **Python**, and notice that the only command that appears on the submenu is **Run PyLint**. Your custom commands appear on this same submenu.
4. As suggested in the introduction, open *Python-CustomCommands.pyproj* in a separate text editor. Then add the following lines at the end of the file just inside the closing `</Project>` and save the file.

```
<PropertyGroup>
  <PythonCommands>
    $(PythonCommands);
  </PythonCommands>
</PropertyGroup>
```

5. Switch back to Visual Studio and select **Reload** when it prompts you about the file change. Then check the **Python** menu again to see that **Run PyLint** is still the only item shown there because the lines you added only replicate the default `<PythonCommands>` property group containing the PyLint command.
6. Switch to the editor with the project file and add the following `<Target>` definition after the `<PropertyGroup>`. As explained later in this article, this `<Target>` element defines a custom command to run the startup file (identified by the "StartupFile" property) using *python.exe* in a console window. The attribute `ExecuteIn="consolepause"` uses a console that waits for you to press a key before closing.

```

<Target Name="Example_RunStartupScriptFile" Label="Run startup file" Returns="@.Commands">
  <CreatePythonCommandItem
    TargetType="script"
    Target="$(StartupFile)"
    Arguments=""
    WorkingDirectory="$(MSBuildProjectDirectory)"
    ExecuteIn="consolepause">
    <Output TaskParameter="Command" ItemName="Commands" />
  </CreatePythonCommandItem>
</Target>

```

- Add the value of the Target's `Name` attribute to the `<PythonCommands>` property group added earlier, so that the element looks like the code below. Adding the name of the target to this list is what adds it to the **Python** menu.

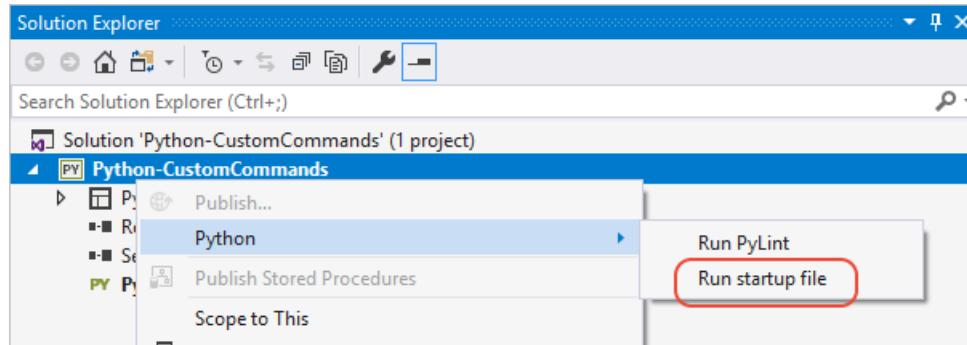
```

<PythonCommands>
$(PythonCommands);
Example_RunStartupScriptFile
</PythonCommands>

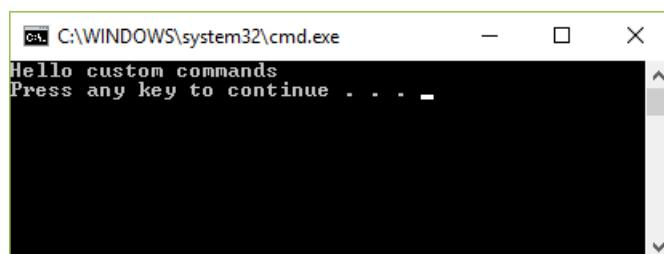
```

If you want your command to appear before those defined in `$(PythonCommands)`, place them before that token.

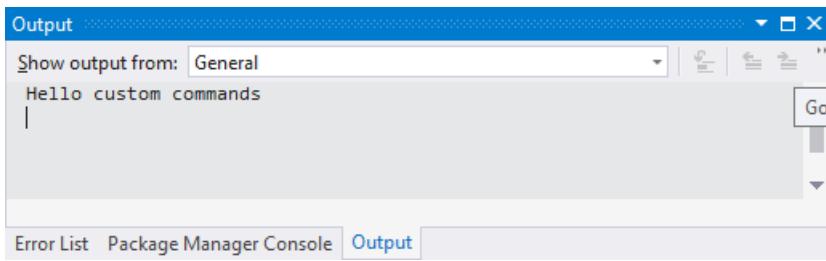
- Save the project file, switch to Visual Studio, and reload the project when prompted. Then right-click the **Python-CustomCommands** project and select **Python**. You should see a **Run startup file** item on the menu. If you don't see the menu item, check that you added the name to the `<PythonCommands>` element. Also see [Troubleshooting](#) later in this article.



- Select the **Run startup file** command and you should see a command window appear with the text **Hello custom commands** followed by **Press any key to continue**. Press a key to close the window.



- Return to the editor with the project file and change the value of the `ExecuteIn` attribute to `output`. Save the file, switch to Visual Studio, reload the project, and invoke the command again. This time you see the program's output appear in Visual Studio's **Output** window:



11. To add more commands, define a suitable `<Target>` element for each command, add the name of the target into the `<PythonCommands>` property group, and reload the project in Visual Studio.

TIP

If you invoke a command that uses project properties, such as `($StartupFile)`, and the command fails because the token is undefined, Visual Studio disables the command until you reload the project. Making changes to the project that would define the property, however, does not refresh the state of these commands, so you still need to reload the project in such cases.

Command target structure

The general form of the `<Target>` element is shown in the following pseudo-code:

```
<Target Name="Name1" Label="Display Name" Returns="@.Commands">
  <CreatePythonCommandItem Target="filename, module name, or code"
    TargetType="executable/script/module/code/pip"
    Arguments="..."
    ExecuteIn="console/console/pause/output/repl[:Display name]/none"
    WorkingDirectory="..."
    ErrorRegex="..."
    WarningRegex="..."
    RequiredPackages="...;..."
    Environment="...">

  <!-- Output always appears in this form, with these exact attributes -->
  <Output TaskParameter="Command" ItemName="Commands" />
</CreatePythonCommandItem>
</Target>
```

To refer to project properties or environment variables in attribute values, use the name within a `$()` token, such as `$(StartupFile)` and `$(MSBuildProjectDirectory)`. For more information, see [MSBuild properties](#).

Target attributes

ATTRIBUTE	REQUIRED	DESCRIPTION
Name	Yes	The identifier for the command within the Visual Studio project. This name must be added to the <code><PythonCommands></code> property group for the command to appear on the Python submenu.
Label	Yes	The UI display name that appears in the Python sub-menu.
Returns	Yes	Must contain <code>@.Commands</code> , which identifies the target as a command.

CreatePythonCommandItem attributes

All attribute values are case-insensitive.

ATTRIBUTE	REQUIRED	DESCRIPTION
TargetType	Yes	<p>Specifies what the Target attribute contains and how it's used along with the Arguments attribute:</p> <ul style="list-style-type: none">• executable: Run the executable named in Target, appending the value in Arguments, as if entered directly on the command line. The value must contain only a program name without arguments.• script: Run <code>python.exe</code> with the filename in Target, followed with the value in Arguments.• module: Run <code>python -m</code> followed by the module name in Target, followed with the value in Arguments.• code: Run the inline code contained in Target. The Arguments value is ignored.• pip: Run <code>pip</code> with the command in Target, followed by Arguments; if Executeln is set to "output", however, pip assumes the <code>install</code> command and uses Target as the package name.
Target	Yes	The filename, module name, code, or pip command to use, depending on the TargetType.
Arguments	Optional	Specifies a string of arguments (if any) to give to the target. Note that when TargetType is <code>script</code> , the arguments are given to the Python program, not <code>python.exe</code> . Ignored for the <code>code</code> TargetType.

ATTRIBUTE	REQUIRED	DESCRIPTION
Executeln	Yes	<p>Specifies the environment in which to run the command:</p> <ul style="list-style-type: none"> • console: (Default) Runs Target and the arguments as if they are entered directly on the command line. A command window appears while the Target is running, then is closed automatically. • consolepause: Same as console, but waits for a keypress before closing the window. • output: Runs Target and displays its results in the Output window in Visual Studio. If TargetType is "pip", Visual Studio uses Target as the package name and appends Arguments. • repl: Runs Target in the Python Interactive window; the optional display name is used for the title of the window. • none: behaves the same as console.
WorkingDirectory	Optional	The folder in which to run the command.
ErrorRegex WarningRegEx	Optional	Used only when Executeln is <code>output</code> . Both values specify a regular expression with which Visual Studio parses command output to show errors and warnings in its Error List window. If not specified, the command does not affect the Error List window. For more information on what Visual Studio expects, see Named capture groups for regular expressions .
RequiredPackages	Optional	A list of package requirements for the command using the same format as requirements.txt (pip.readthedocs.io). The Run PyLint command, for example specifies <code>pylint>=1.0.0</code> . Before running the command, Visual Studio checks that all packages in the list are installed. Visual Studio uses pip to install any missing packages.
Environment	Optional	A string of environment variables to define before running the command. Each variable uses the form <NAME>=<VALUE> with multiple variables separated by semicolons. A variable with multiple values must be contained in single or double quotes, as in 'NAME=VALUE1;VALUE2'.

Named capture groups for regular expressions

When parsing error and warnings from a command's output, Visual Studio expects that the regular expressions in the `ErrorRegex` and `WarningRegex` values use the following named groups:

- `(?<message>....)` : Text of the error
- `(?<code>....)` : Error code
- `(?<filename>....)` : Name of the file for which the error is reported
- `(?<line>....)` : Line number of the location in the file for which the error reported.
- `(?<column>....)` : Column number of the location in the file for which the error reported.

For example, PyLint produces warnings of the following form:

```
***** Module hello
C: 1, 0: Missing module docstring (missing-docstring)
```

To allow Visual Studio to extract the right information from such warnings and show them in the **Error List** window, the `WarningRegex` value for the **Run Pylint** command is as follows:

```
^(?<filename>.+?)\((?<line>\d+),(?<column>\d+)\): warning (?<msg_id>.+?): (?<message>.+?)$]]
```

(Note that `msg_id` in the value should actually be `code`, see [Issue 3680](#).)

Create a .targets file with custom commands

Defining custom commands in a project file makes them available to only that project file. To use commands in multiple project files, you instead define the `<PythonCommands>` property group and all your `<Target>` elements in a `.targets` file. You then import that file into individual project files.

The `.targets` file is formatted as follows:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <PythonCommands>
      $(PythonCommands);
      <!-- Additional command names -->
    </PythonCommands>
  </PropertyGroup>

  <Target Name="..." Label="..." Returns="@((Commands))">
    <!-- CreatePythonCommandItem and Output elements... -->
  </Target>

  <!-- Any number of additional Target elements-->
</Project>
```

To load a `.targets` file into a project, place a `<Import Project="(path)">` element anywhere within the `<Project>` element. For example, if you have a file named `CustomCommands.targets` in a `targets` subfolder in your project, use the following code:

```
<Import Project="targets/CustomCommands.targets"/>
```

NOTE

Whenever you change the `.targets` file, you need to reload the `solution` that contains a project, not just the project itself.

Example commands

Run PyLint (module target)

The following code appears in the *Microsoft.PythonTools.targets* file:

```
<PropertyGroup>
  <PythonCommands>$({PythonCommands});PythonRunPyLintCommand</PythonCommands>
  <PyLintWarningRegex>
    <![CDATA[^(?<filename>.+)\((?<line>\d+),(?<column>\d+)\): warning (?<msg_id>.+?): (?<message>.+?)$]]>
  </PyLintWarningRegex>
</PropertyGroup>

<Target Name="PythonRunPyLintCommand"
      Label="resource:Microsoft.PythonTools.Common;Microsoft.PythonTools.Common.Strings;RunPyLintLabel"
      Returns="@.Commands">
  <CreatePythonCommandItem Target="pylint.lint"
    TargetType="module"
    Arguments="--msg-template={abspath}({line},{column}): warning {msg_id}: {msg}
    [{C}:{symbol}]"
    WorkingDirectory="$(MSBuildProjectDirectory)"
    ExecuteIn="output"
    RequiredPackages="pylint>=1.0.0"
    WarningRegex="$(PyLintWarningRegex)">
    <Output TaskParameter="Command" ItemName="Commands" />
  </CreatePythonCommandItem>
</Target>
```

Run pip install with a specific package (pip target)

The following command runs `pip install my-package` in the **Output** window. You might use such a command when developing a package and testing its installation. Note that Target contains the package name rather than the `install` command, which is assumed when using `ExecuteIn="output"`.

```
<PropertyGroup>
  <PythonCommands>$({PythonCommands});InstallMyPackage</PythonCommands>
</PropertyGroup>

<Target Name="InstallMyPackage" Label="pip install my-package" Returns="@.Commands">
  <CreatePythonCommandItem Target="my-package" TargetType="pip" Arguments=""
    WorkingDirectory="$(MSBuildProjectDirectory)" ExecuteIn="output">
    <Output TaskParameter="Command" ItemName="Commands" />
  </CreatePythonCommandItem>
</Target>
```

Show outdated pip packages (pip target)

```
<PropertyGroup>
  <PythonCommands>$({PythonCommands});ShowOutdatedPackages</PythonCommands>
</PropertyGroup>

<Target Name="ShowOutdatedPackages" Label="Show outdated pip packages" Returns="@.Commands">
  <CreatePythonCommandItem Target="list" TargetType="pip" Arguments="-o --format columns"
    WorkingDirectory="$(MSBuildProjectDirectory)" ExecuteIn="consolepause">
    <Output TaskParameter="Command" ItemName="Commands" />
  </CreatePythonCommandItem>
</Target>
```

Run an executable with consolepause

The following command simply runs `where` to show Python files starting in the project folder:

```

<PropertyGroup>
  <PythonCommands>$({PythonCommands});ShowAllPythonFilesInProject</PythonCommands>
</PropertyGroup>

<Target Name="ShowAllPythonFilesInProject" Label="Show Python files in project" Returns="@({Commands})">
  <CreatePythonCommandItem Target="where" TargetType="executable" Arguments="/r . *.py"
    WorkingDirectory="$(MSBuildProjectDirectory)" ExecuteIn="output">
    <Output TaskParameter="Command" ItemName="Commands" />
  </CreatePythonCommandItem>
</Target>

```

Run server and run debug server commands

To explore how the **Start server** and **Start debug server** commands for web projects are defined, examine the [Microsoft.PythonTools.Web.targets](#) (GitHub).

Install package for development

```

<PropertyGroup>
  <PythonCommands>PipInstallDevCommand;${PythonCommands};</PythonCommands>
</PropertyGroup>

<Target Name="PipInstallDevCommand" Label="Install package for development" Returns="@({Commands})">
  <CreatePythonCommandItem Target="pip" TargetType="module" Arguments="install --editable ${ProjectDir}"
    WorkingDirectory="$(WorkingDirectory)" ExecuteIn="Repl:Install package for development">
    <Output TaskParameter="Command" ItemName="Commands" />
  </CreatePythonCommandItem>
</Target>

```

From [fxthomas/Example.pyproj.xml](#) (GitHub), used with permission.

Generate Windows installer

```

<PropertyGroup>
  <PythonCommands>$({PythonCommands});BdistWinInstCommand;</PythonCommands>
</PropertyGroup>

<Target Name="BdistWinInstCommand" Label="Generate Windows Installer" Returns="@({Commands})">
  <CreatePythonCommandItem Target="$(ProjectDir)setup.py" TargetType="script"
    Arguments="bdist_wininst --user-access-control=force --title "${InstallerTitle}" --dist-dir="${DistributionOutputDir}"">
    WorkingDirectory="$(WorkingDirectory)" RequiredPackages="setuptools"
    ExecuteIn="Repl:Generate Windows Installer">
    <Output TaskParameter="Command" ItemName="Commands" />
  </CreatePythonCommandItem>
</Target>

```

From [fxthomas/Example.pyproj.xml](#) (GitHub), used with permission.

Generate wheel package

```

<PropertyGroup>
  <PythonCommands>$({PythonCommands});BdistWheelCommand;</PythonCommands>
</PropertyGroup>

<Target Name="BdistWheelCommand" Label="Generate Wheel Package" Returns="@({Commands})">

  <CreatePythonCommandItem Target="$(ProjectDir)setup.py" TargetType="script"
    Arguments="bdist_wheel --dist-dir=""$(DistributionOutputDir)""
    WorkingDirectory="$(WorkingDirectory)" RequiredPackages="wheel;setuptools"
    ExecuteIn="Repl:Generate Wheel Package">
    <Output TaskParameter="Command" ItemName="Commands" />
  </CreatePythonCommandItem>
</Target>

```

From [fxthomas/Example.pyproj.xml](#) (GitHub), used with permission.

Troubleshooting

Message: "The project file could not be loaded"

Indicates that you have syntax errors in the project file. The message includes the specific error with a line number and character position.

Console window closes immediately after command is run

Use `ExecuteIn="consolepause"` instead of `ExecuteIn="console"`.

Command does not appear on the menu

Check that the command is included in the `<PythonCommands>` property group, and that the name in the command list matches the name specified in the `<Target>` element.

For example, in the following elements, the "Example" name in the property group does not match the name "ExampleCommand" in the target. Visual Studio does not find a command named "Example", so no command appears. Either use "ExampleCommand" in the command list, or change the name of the target to "Example" only.

```

<PropertyGroup>
  <PythonCommands>$({PythonCommands});Example;</PythonCommands>
</PropertyGroup>
<Target Name="ExampleCommand" Label="Example Command" Returns="@({Commands})">
  <!-- ... -->
</Target>

```

Message: "An error occurred while running <command name>. Failed to get command <target-name> from project."

Indicates that the contents of the `<Target>` or `<CreatePythonCommandItem>` elements are incorrect. Possible reasons include:

- The required `Target` attribute is empty.
- The required `TargetType` attribute is empty or contains an unrecognized value.
- The required `ExecuteIn` attribute is empty or contains an unrecognized value.
- `ErrorRegex` or `WarningRegex` is specified without setting `ExecuteIn="output"`.
- Unrecognized attributes exist in the element. For example, you may have used `Argumnets` (misspelled) instead of `Arguments`.

Attribute values can be empty if you refer to a property that's not defined. For example, if you use the token `$(StartupFile)` but no startup file has been defined in the project, then the token resolves to an empty string. In such cases, you may want to define a default value. For example, the **Run server** and **Run debug server**

commands defined in the Bottle, Flask, and Django project templates default to `manage.py` if you haven't otherwise specified a server startup file in the project properties.

Visual Studio hangs and crashes when running the command

You're likely attempting to run a console command with `ExecuteIn="output"`, in which case Visual Studio may crash trying to parse the output. Use `ExecuteIn="console"` instead. (See [Issue 3682](#).)

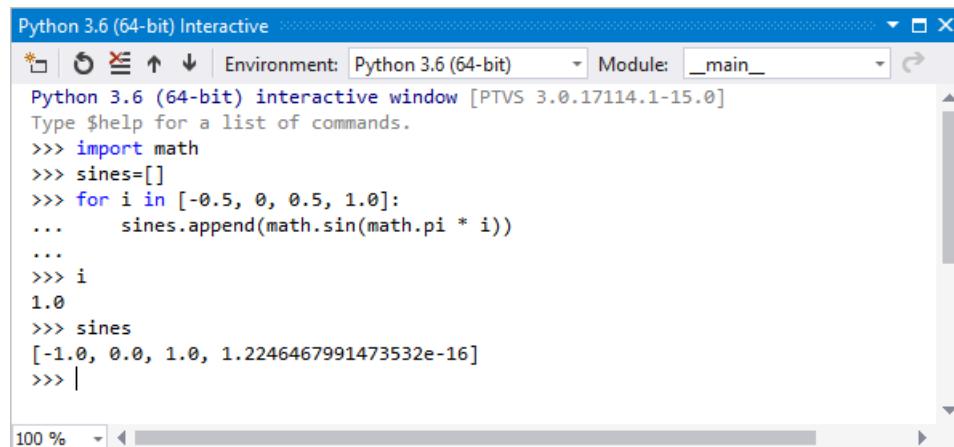
Executable command "is not recognized as an internal or external command, operable program or batch file"

When using `TargetType="executable"`, the value in `Target` must be *only* the program name without any arguments, such as `python` or `python.exe` only. Move any arguments to the `Arguments` attribute.

Work with the Python Interactive window

4/9/2019 • 6 minutes to read • [Edit Online](#)

Visual Studio provides an interactive read-evaluate-print loop (REPL) window for each of your Python environments, which improves upon the REPL you get with `python.exe` on the command line. The **Interactive** window (opened with the **View > Other Windows > <environment> Interactive** menu commands) lets you enter arbitrary Python code and see immediate results. This way of coding helps you learn and experiment with APIs and libraries, and to interactively develop working code to include in your projects.



The screenshot shows the Python 3.6 (64-bit) Interactive window. The title bar says "Python 3.6 (64-bit) Interactive". The environment dropdown is set to "Python 3.6 (64-bit)". The module dropdown is set to "_main_". The window displays Python code and its output:

```
Python 3.6 (64-bit) interactive window [PTVS 3.0.17114.1-15.0]
Type $help for a list of commands.
>>> import math
>>> sines=[]
>>> for i in [-0.5, 0, 0.5, 1.0]:
...     sines.append(math.sin(math.pi * i))
...
>>> i
1.0
>>> sines
[-1.0, 0.0, 1.0, 1.2246467991473532e-16]
>>> |
```

Visual Studio has a number of Python REPL modes to choose from:

REPL	DESCRIPTION	EDITING	DEBUGGING	IMAGES
Standard	Default REPL, talks to Python directly	Standard editing (multiline, etc.).	Yes, via <code>\$attach</code>	No
Debug	Default REPL, talks to debugged Python process	Standard editing	Only debugging	No
IPython	REPL talks to IPython backend	IPython commands, Pylab conveniences	No	Yes, inline in REPL
IPython w/o Pylab	REPL talks to IPython backend	Standard IPython	No	Yes, separate window

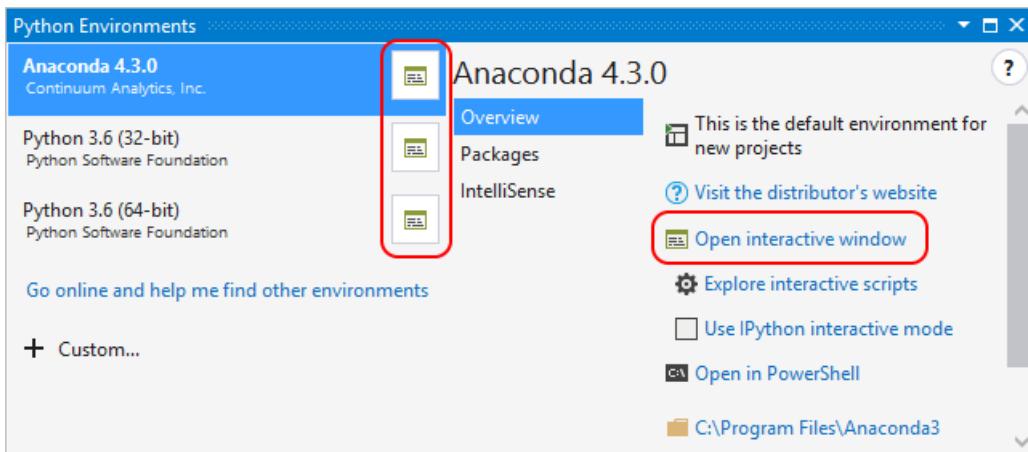
This article describes the **Standard** and **Debug** REPL modes. For details on IPython modes, see [Use the IPython REPL](#).

For a detailed walkthrough with examples, including the interactions with the editor such as **Ctrl+Enter**, see [Tutorial Step 3: Use the Interactive REPL window](#).

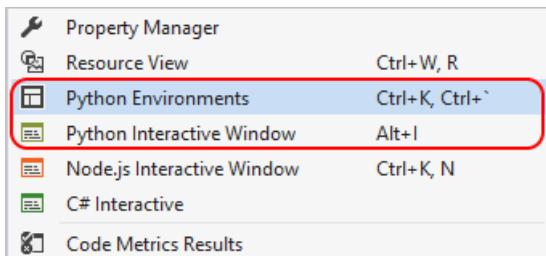
Open an Interactive window

There are several ways to open the **Interactive** window for an environment.

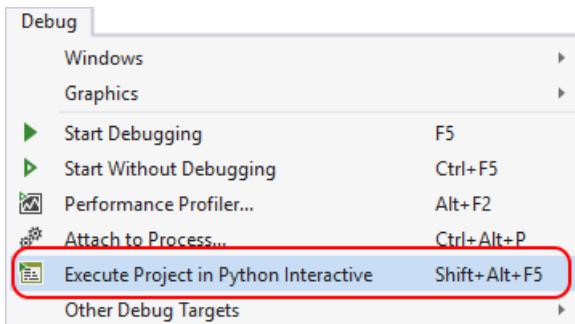
First, switch to the Python Environments window (**View > Other Windows > Python Environments** or **Ctrl+K > Ctrl+`**) and select the **Open Interactive Window** command or button for a chosen environment.



Second, near the bottom of the **View > Other Windows** menu, there's a **Python Interactive Window** command for your default environment, as well as a command to switch to the **Environments** window:



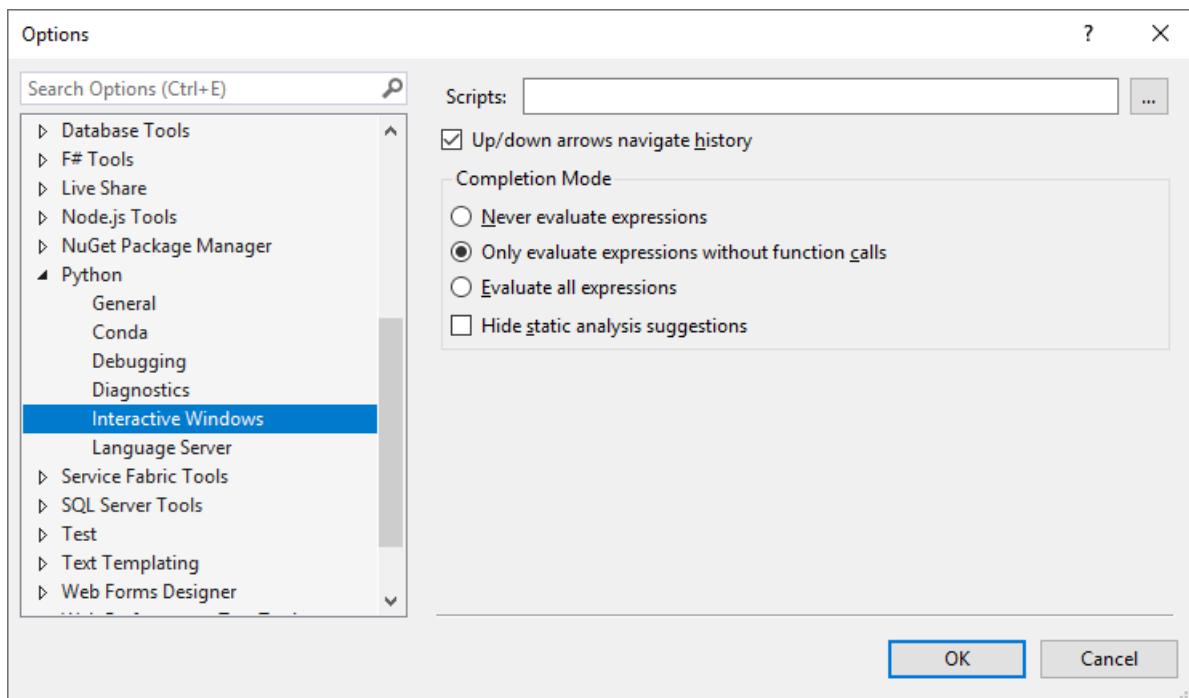
Third, you can open an **Interactive** window on the startup file in your project, or for a stand-alone file, by selecting the **Debug > Execute <Project | File> in Python Interactive** menu command (**Shift+Alt+F5**):



Finally, you can select code in file and use the **Send to Interactive** command described below.

Interactive window options

You can control various aspects of the **Interactive** window through **Tools > Options > Python > Interactive Windows** (see [Options](#)):



Use the Interactive window

Once the **Interactive** window is open, you can start entering code line-by-line at the `>>>` prompt. The **Interactive** window executes each line as you enter it, which includes importing modules, defining variables, and so on:

```
Python 3.6 (64-bit) Interactive
Python 3.6 (64-bit) interactive window [PTVS 3.0.17114.1-15.0]
Type $help for a list of commands.
>>> import math
>>> sines=[]
>>> for i in [-0.5, 0, 0.5, 1.0]:
...     sines.append(math.sin(math.pi * i))
...
>>> i
1.0
>>> sines
[-1.0, 0.0, 1.0, 1.2246467991473532e-16]
>>> |
```

The exception is when additional lines of code are needed to make a complete statement, such as when a `for` statement ends in a colon as shown above. In these cases, the line prompt changes to `...` indicating that you need to enter additional lines for the block, as shown on the fourth and fifth lines in the graphic above. When you press **Enter** on a blank line, the **Interactive** window closes the block and runs it in the interpreter.

TIP

The **Interactive** window improves upon the usual Python command-line REPL experience by automatically indenting statements that belong to a surrounding scope. Its history (recalled with the up arrow) also provides multiline items, whereas the command-line REPL provides only single lines.

The **Interactive** window also supports several meta-commands. All meta-commands start with `$`, and you can type `$help` to get a list of the meta-commands and `$help <command>` to get usage details for a specific command.

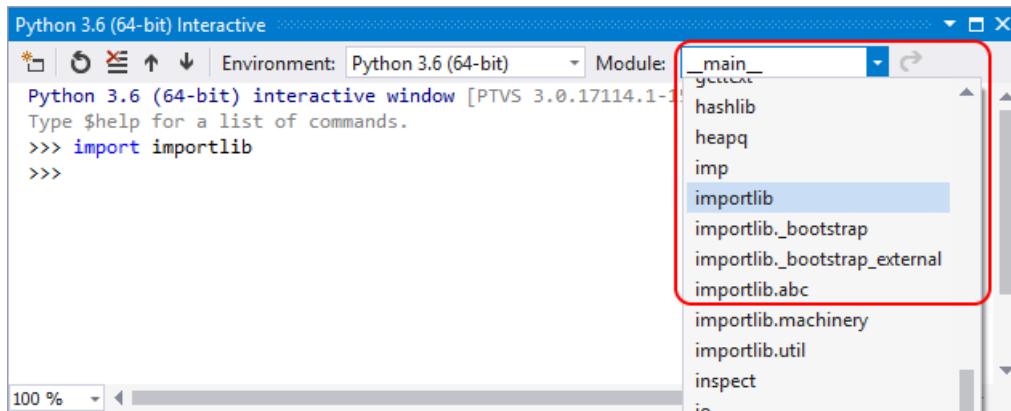
META-COMMAND	DESCRIPTION
<code>\$\$</code>	Inserts a comment, which is helpful to comment code throughout your session.
<code>\$attach</code>	Attaches the Visual Studio debugger to the REPL window process to enable debugging.
<code>\$cls</code> , <code>\$clear</code>	Clears the contents of the editor window, leaving history and execution context intact.
<code>\$help</code>	Display a list of commands, or help on a specific command.
<code>\$load</code>	Loads commands from file and executes until complete.
<code>\$mod</code>	Switches the current scope to the specified module name.
<code>\$reset</code>	Resets the execution environment to the initial state, but keeps history.
<code>\$wait</code>	Waits for at least the specified number of milliseconds.

Commands are also extensible by Visual Studio extensions by implementing and exporting

`IInteractiveWindowCommand` ([example](#)).

Switch scopes

By default, the **Interactive** window for a project is scoped to the project's startup file as if you ran it from the command prompt. For a stand-alone file, it scopes to that file. At any time during your REPL session, however, the drop-down menu along the top of the **Interactive** window lets you change scope:



Once you import a module, such as typing `import importlib`, options appear in the drop-down to switch into any scope in that module. A message in the **Interactive** window also indicates the new scope, so you can track how you got to a certain state during your session.

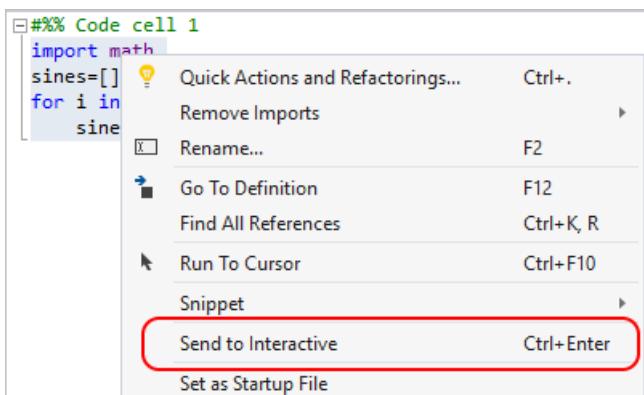
Entering `dir()` in a scope displays valid identifiers in that scope, including function names, classes, and variables. For example, using `import importlib` followed by `dir()` shows the following:

The screenshot shows the Python 3.6 (64-bit) Interactive window. At the top, it says "Python 3.6 (64-bit) interactive window [PTVS 3.0.17114.1-15.0]". Below that, the command line shows:

```
>>> import importlib
Current module changed to importlib
>>> dir()
['_RELOADING', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__im...']
```

Send to Interactive command

In addition to working within the **Interactive** window directly, you can select code in the editor, right-click, and choose **Send to Interactive** or press **Ctrl+Enter**.



This command is useful for iterative or evolutionary code development, including testing your code as you develop it. For example, once you've sent a piece of code to the **Interactive** window and seen its output, you can press the up arrow to show the code again, modify it, and test it quickly by pressing **Ctrl+Enter**. (Pressing **Enter** at the end of input executes it, but pressing **Enter** in the middle of input inserts a newline.) Once you have the code you want, you can easily copy it back into your project file.

TIP

By default, Visual Studio removes `>>>` and `...` REPL prompts when pasting code from the **Interactive** window into the editor. You can change this behavior on the **Tools > Options > Text Editor > Python > Advanced** tab using the **Paste removes REPL prompts** option. See [Options - Miscellaneous options](#).

When using a code file as a scratchpad, you often have a small block of code you want to send all at once. To group code together, mark the code as a *code cell* by adding a comment starting with `#%%` to the beginning of the cell, which ends the previous one. Code cells can be collapsed and expanded, and using **Ctrl+Enter** inside a code cell sends the entire cell to the **Interactive** window and moves to the next one.

Visual Studio also detects code cells starting with comments like `# In[1]:`, which is the format you get when exporting a Jupyter notebook as a Python file. This detection makes it easy to run a notebook from [Azure Notebooks](#) by downloading as a Python file, opening in Visual Studio, and using **Ctrl+Enter** to run each cell.

The screenshot shows a Python code editor and an interactive window side-by-side. The code editor window at the top contains two code cells:

```
#%% Code cell 1
import math
tau = math.tau
pi = math.pi

#%% Code cell 2

if math.isclose(tau, pi * 2):
    print("Indeed, τ = 2π")
else:
    print("Oh no, my circles are broken")
```

The interactive window below it shows the execution of these cells:

```
>>> #%% Code cell 1
... import math
...
>>> tau = math.tau
>>> pi = math.pi
>>> #%% Code cell 2
...
... if math.isclose(tau, pi * 2):
...     print("Indeed, τ = 2π")
... else:
...     print("Oh no, my circles are broken")
...
Indeed, τ = 2π
>>>
```

IntelliSense behavior

The **Interactive** window includes IntelliSense based on the live objects, unlike the code editor in which IntelliSense is based on source code analysis only. These suggestions are more correct in the **Interactive** window, especially with dynamically generated code. The drawback is that functions with side-effects (such as logging messages) may impact your development experience.

If this behavior is a problem, change the settings under **Tools > Options > Python > Interactive Windows** in the **Completion Mode** group, as described on [Options - Interactive windows options](#).

Use IPython in the Interactive window

4/9/2019 • 2 minutes to read • [Edit Online](#)

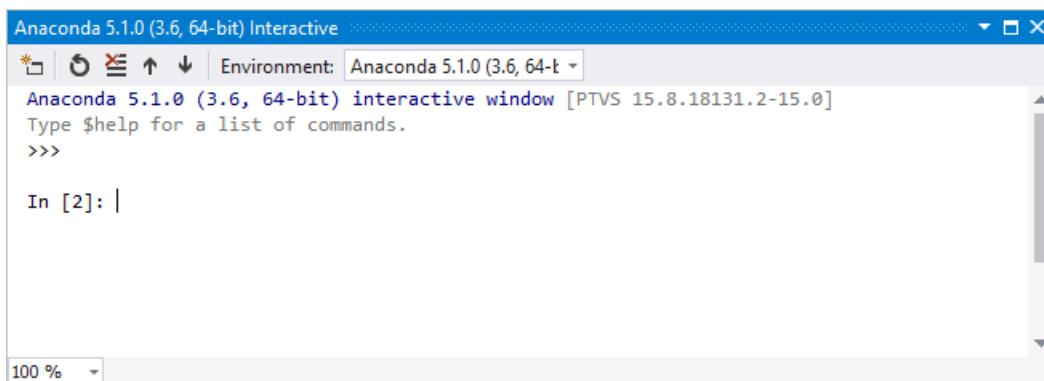
The Visual Studio **Interactive** window in IPython mode is an advanced yet user-friendly interactive development environment that has Interactive Parallel Computing features. This article walks through using IPython in the Visual Studio **Interactive** window, in which all of the regular **Interactive window** features are also available.

For this walkthrough you should have the [Anaconda](#) environment installed, which includes IPython and the necessary libraries.

NOTE

IronPython does not support IPython, despite the fact that you can select it on the **Interactive Options** form. For more information see the [feature request](#).

1. Open Visual Studio, switch to the **Python Environments** window (**View > Other Windows > Python Environments**), and select an Anaconda environment.
2. Examine the **Packages (Conda)** tab (which may appear as **pip** or **Packages**) for that environment to make sure that `ipython` and `matplotlib` are listed. If not, install them here. (See [Python Environments windows - Packages tab](#).)
3. Select the **Overview** tab and select **Use IPython interactive mode**. (In Visual Studio 2015, select **Configure interactive options** to open the **Options** dialog, then set **Interactive Mode** to **IPython**, and select **OK**).
4. Select **Open interactive window** to bring up the **Interactive** window in IPython mode. You may need to reset the window if you have just changed the interactive mode; you might also need to press **Enter** if only a `>>>` prompt appears, so that you get a prompt like `In [2]`.

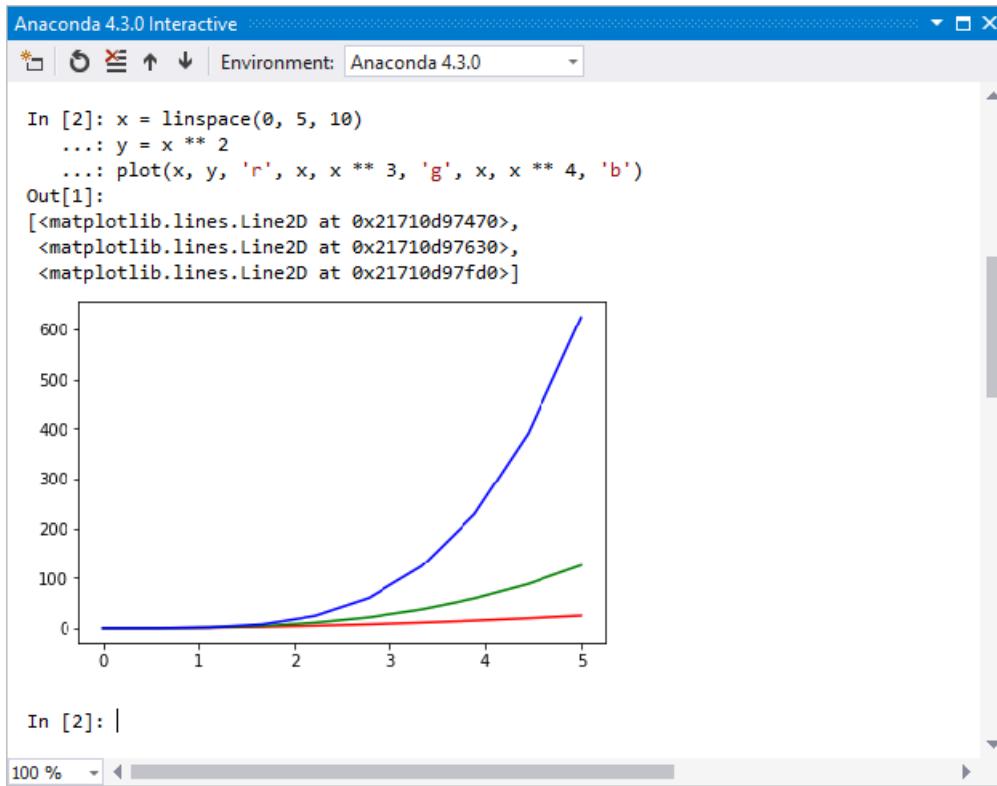


5. Enter the following code:

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

x = np.linspace(0, 5, 10)
y = x ** 2
plt.plot(x, y, 'r', x, x ** 3, 'g', x, x ** 4, 'b')
```

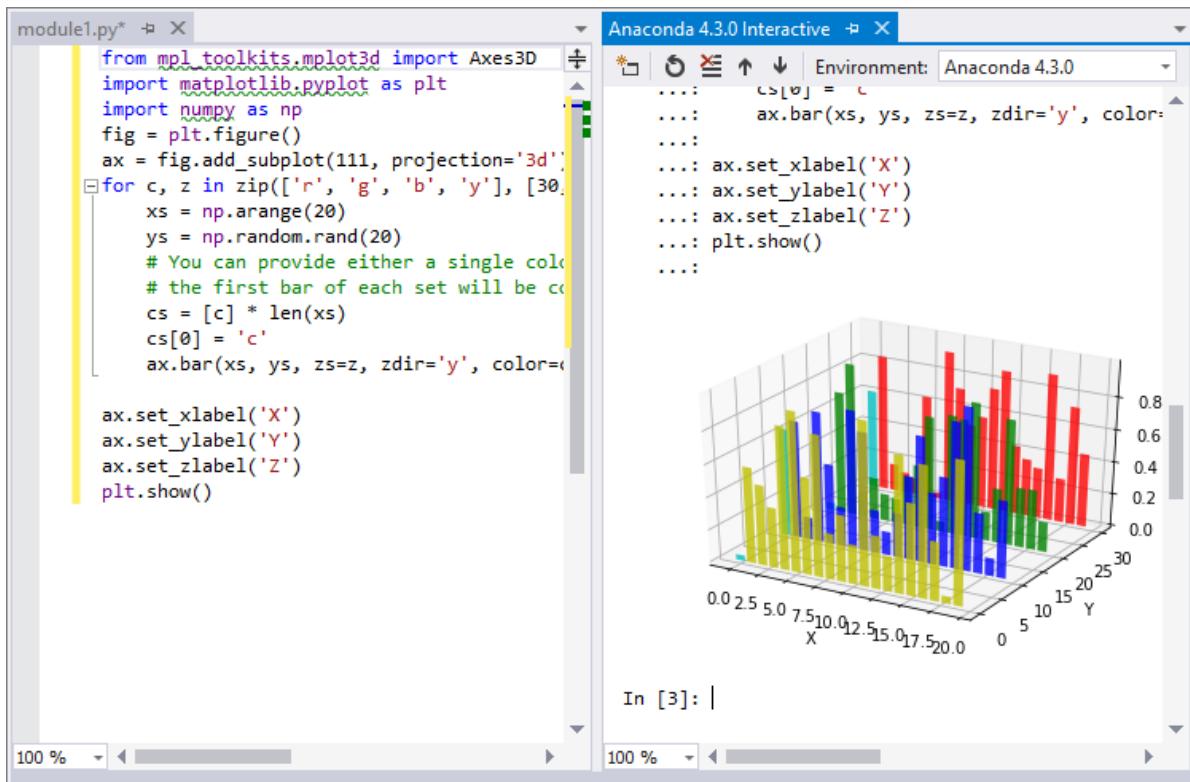
6. After entering the last line, you should see an inline graph (which you can resize by dragging on the lower right-hand corner if desired).



- Instead of typing in the REPL, you can instead write code in the editor, select it, right-click, and select the **Send to Interactive** command (or press **Ctrl+Enter**). Try pasting the code below into a new file in the editor, selecting it with **Ctrl+A**, then sending to the **Interactive** window. (Visual Studio sends the code as one unit to avoid giving you intermediate or partial graphs. And if you don't have a Python project open with a different environment selected, Visual Studio opens an **Interactive** window for whatever environment is selected as your default in the **Python Environments** window.)

```
from mpl_toolkits.mplot3d import Axes3D
import matplotlib.pyplot as plt
import numpy as np
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for c, z in zip(['r', 'g', 'b', 'y'], [30, 20, 10, 0]):
    xs = np.arange(20)
    ys = np.random.rand(20)
    # You can provide either a single color or an array. To demonstrate this,
    # the first bar of each set is colored cyan.
    cs = [c] * len(xs)
    cs[0] = 'c'
    ax.bar(xs, ys, zs=z, zdir='y', color=cs, alpha=0.8)

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.show()
```



- To see the graphs outside of the **Interactive** window, run the code instead using the **Debug > Start without Debugging** command.

IPython has many other useful features such as escaping to the system shell, variable substitution, capturing output, etc. Refer to the [IPython documentation](#) for more.

See also

- To use Jupyter easily and without installation, try the free [Azure Notebooks hosted service](#) that lets you keep and share your notebooks with others.
- The [Azure Data Science Virtual Machine](#) is also pre-configured to run Jupyter notebooks along with a wide range of other data science tools.

Debug your Python code

4/9/2019 • 12 minutes to read • [Edit Online](#)

Visual Studio provides a comprehensive debugging experience for Python, including attaching to running processes, evaluating expressions in the **Watch** and **Immediate** windows, inspecting local variables, breakpoints, step in/out/over statements, **Set Next Statement**, and more.

Also see the following scenario-specific debugging articles:

- [Linux remote debugging](#)
- [Mixed-mode Python/C++ debugging](#)
- [Symbols for mixed-mode debugging](#)

TIP

Python in Visual Studio supports debugging without a project. With a stand-alone Python file open, right-click in the editor, select **Start with Debugging**, and Visual Studio launches the script with the global default environment (see [Python environments](#)) and no arguments. But from then on, you have full debugging support.

To control the environment and arguments, create a project for the code, which is easily done with the [From existing Python code project template](#).

Basic debugging

The basic debugging workflow involves settings breakpoints, stepping through code, inspecting values, and handling exceptions as described in the following sections.

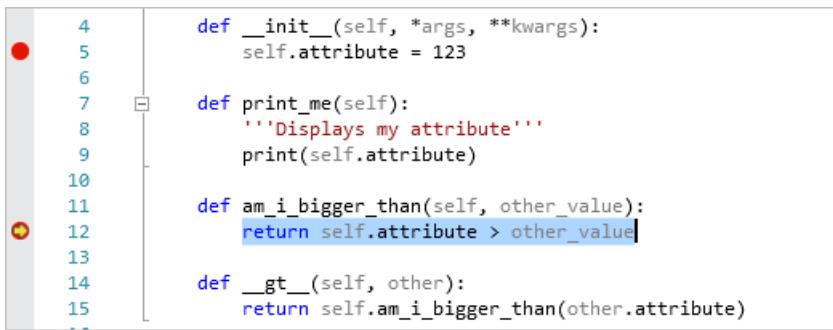
A debugging session starts with the **Debug > Start Debugging** command, the **Start** button on the toolbar, or the **F5** key. These actions launch your project's startup file (shown in bold in **Solution Explorer**) with the project's active environment and any command-line arguments or search paths that have been specified in **Project Properties** (see [Project debugging options](#)). Visual Studio 2017 version 15.6 and later alerts you if you don't have a startup file set; earlier versions may open an output window with the Python interpreter running, or the output window briefly appears and disappears. In any case, right-click the appropriate file and select **Set as Startup File**.

NOTE

The debugger always starts with the active Python environment for the project. To change the environment, make a different one active as described on [Select a Python environment for a project](#).

Breakpoints

Breakpoints stop execution of code at a marked point so you can inspect the program state. Set breakpoints by clicking in the left margin of the code editor or by right-clicking a line of code and selecting **Breakpoint > Insert Breakpoint**. A red dot appears on each line with a breakpoint.



```

4     def __init__(self, *args, **kwargs):
5         self.attribute = 123
6
7     def print_me(self):
8         '''Displays my attribute'''
9         print(self.attribute)
10
11    def am_i_bigger_than(self, other_value):
12        return self.attribute > other_value
13
14    def __gt__(self, other):
15        return self.am_i_bigger_than(other.attribute)

```

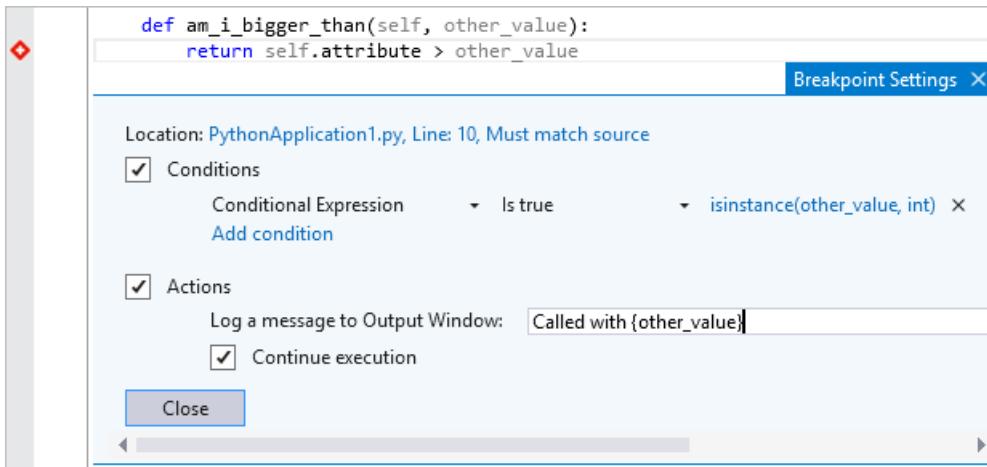
Clicking the red dot or right-clicking the line of code and selecting **Breakpoint** > **Delete Breakpoint** removes the breakpoint. You can also disable it without removing it using the **Breakpoint** > **Disable Breakpoint** command.

NOTE

Some breakpoints in Python can be surprising for developers who have worked with other programming languages. In Python, the entire file is executable code, so Python runs the file when it's loaded to process any top-level class or function definitions. If a breakpoint has been set, you may find the debugger breaking part-way through a class declaration. This behavior is correct, even though it's sometimes surprising.

You can customize the conditions under which a breakpoint is triggered, such as breaking only when a variable is set to a certain value or value range. To set conditions, right-click the breakpoint's red dot, select **Condition**, then create expressions using Python code. For full details on this feature in Visual Studio, see [Breakpoint conditions](#).

When setting conditions, you can also set **Action** and create a message to log to the output window, optionally continuing execution automatically. Logging a message creates what is called a *tracepoint* without adding logging code to your application directly:



Step through code

Once stopped at a breakpoint, you have various ways to step through code or run blocks of code before breaking again. These commands are available in a number of places, including the top debug toolbar, the **Debug** menu, on the right-click context menu in the code editor, and through keyboard shortcuts (though not all commands are in all places):

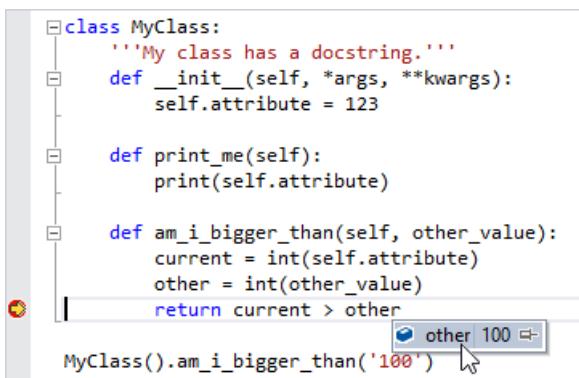
FEATURE	KEYSTROKE	DESCRIPTION
Continue	F5	Runs code until the next breakpoint is reached.

FEATURE	KEYSTROKE	DESCRIPTION
Step Into	F11	Runs the next statement and stops. If the next statement is a call to a function, the debugger stops at the first line of the function being called.
Step Over	F10	Runs the next statement, including making a call to a function (running all its code) and applying any return value. Stepping over allows you to easily skip functions that you do not need to debug.
Step Out	Shift+F11	Runs code until the end of the current function, then steps to the calling statement. This command is useful when you don't need to debug the remainder of the current function.
Run to Cursor	Ctrl+F10	Runs code up to the location of the caret in the editor. This command allows you to easily skip over a segment of code that you don't need to debug.
Set Next Statement	Ctrl+Shift+F10	Changes the current run point in the code to the location of the caret. This command allows you to omit a segment of code from being run at all, such as when you know the code is faulty or produces an unwanted side-effect.
Show Next Statement	Alt+Num *	Returns you to the next statement to run. This command is helpful if you've been looking around in your code and don't remember where the debugger is stopped.

Inspect and modify values

When stopped in the debugger, you can inspect and modify the values of variables. You can also use the **Watch** window to monitor individual variables as well as custom expressions. (See [Inspect variables](#) for general details.)

To view a value using **DataTips**, simply hover the mouse over any variable in the editor. You can click on the value to change it:



The **Autos** window (**Debug > Windows > Autos**) contains variables and expressions that are close to the current statement. You can double-click in the value column or select and press **F2** to edit the value:

The Autos window displays a table of variables:

Name	Value	Type
<input checked="" type="checkbox"/> int(other.attribute)	3	int
<input checked="" type="checkbox"/> other.attribute	3.14159	float
<input checked="" type="checkbox"/> self.attribute	2	int
<input checked="" type="checkbox"/> value_1	2	int
<input checked="" type="checkbox"/> value_2	3	int

Locals Autos Watch 1

The **Locals** window (**Debug > Windows > Locals**) displays all variables that are in the current scope, which can again be edited:

The Locals window displays a table of variables across different scopes:

Name	Value	Type
<input checked="" type="checkbox"/> value_1	2	int
<input checked="" type="checkbox"/> value_2	3	int
<input checked="" type="checkbox"/> self	<MyClass: 2>	MyClass
<input checked="" type="checkbox"/> attribute	2	int
<input checked="" type="checkbox"/> other	<MyClass: 3.14159>	MyClass
<input checked="" type="checkbox"/> attribute	3.14159	float

Locals Autos Watch 1

For more on using **Autos** and **Locals**, see [Inspect variables in the Autos and Locals windows](#).

The **Watch** windows (**Debug > Windows > Watch > Watch 1-4**) allow you to enter arbitrary Python expressions and view the results. Expressions are reevaluated for each step:

The Watch 1 window displays a table of evaluated expressions:

Name	Value	Type
<input checked="" type="checkbox"/> type(other.attribute)	<class 'float'>	type
<input checked="" type="checkbox"/> value_2 - value_1	1	int
<input checked="" type="checkbox"/> str(other)	'<MyClass: 3.14159>'	str

Locals Autos Watch 1

For more on using **Watch**, see [Set a watch on variables using the Watch and QuickWatch windows](#).

When inspecting a string value (`str`, `unicode`, `bytes`, and `bytearray` are all considered strings for this purpose), a magnifying glass icon appears on the right side of the value. Clicking the icon displays the unquoted string value in a popup dialog, with wrapping and scrolling, which is useful for long strings. In addition, selecting the drop-down arrow on the icon allows you to select plain text, HTML, XML, and JSON visualizations:

The Autos window shows a context menu for a string value:

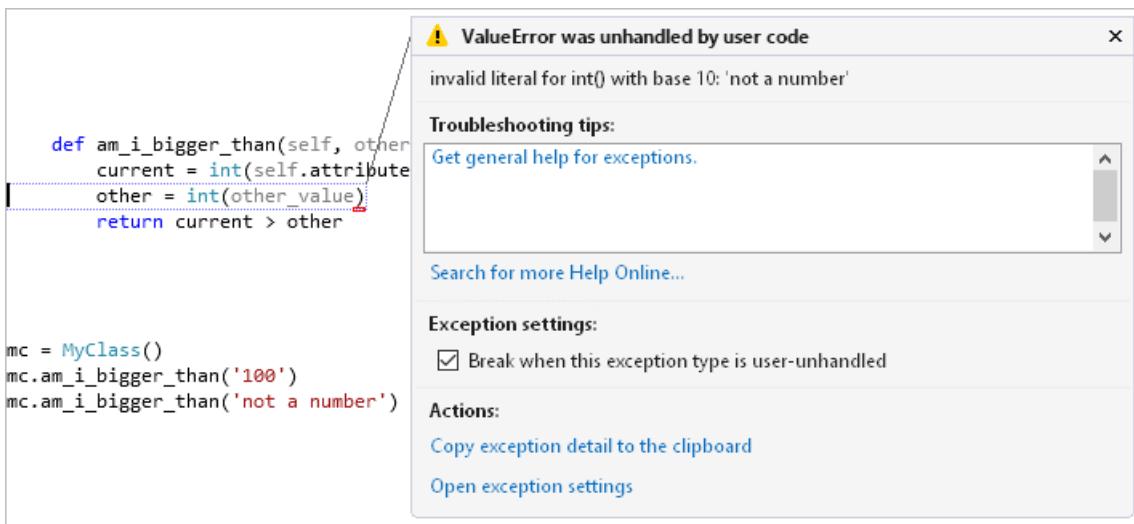
- Text Visualizer
- XML Visualizer
- HTML Visualizer
- JSON Visualizer

Locals Autos Watch 1

HTML, XML, and JSON visualizations appear in separate popup windows with syntax highlighting and tree views.

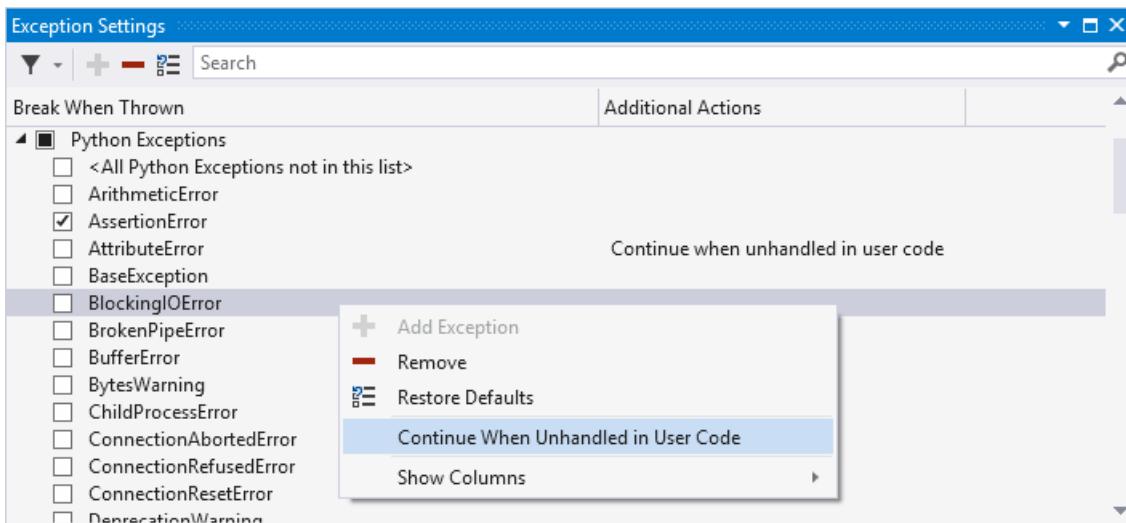
Exceptions

If an error occurs in your program during debugging, but you don't have an exception handler for it, the debugger breaks at the point of the exception:



At this point you can inspect the program state, including the call stack. However, if you attempt to step through the code, the exception continues being thrown until it is either handled or your program exits.

The **Debug > Windows > Exception Settings** menu command brings up a window in which you can expand **Python Exceptions**:



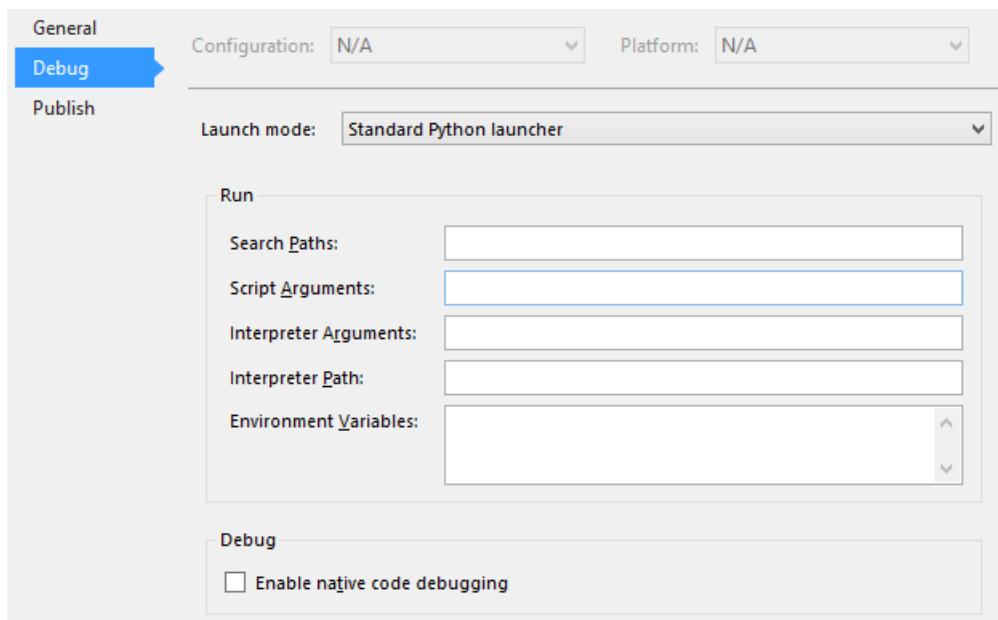
The checkbox for each exception controls whether the debugger *always* breaks when it is raised. Check this box when you want to break more often for a particular exception.

By default, most exceptions break when an exception handler cannot be found in the source code. To change this behavior, right-click any exception and modify the **Continue When Unhandled in User Code** option. Clear this box when you want to break less often for an exception.

To configure an exception that does not appear in this list, click the **Add** button to add it. The name must match the full name of the exception.

Project debugging options

By default, the debugger starts your program with the standard Python launcher, no command-line arguments, and no other special paths or conditions. Startup options are changed through the project's debug properties accessed by right-clicking your project in **Solution Explorer**, selecting **Properties**, and selecting the **Debug** tab.



Launch mode options

OPTION	DESCRIPTION
Standard Python launcher	Uses debugging code written in portable Python that is compatible with CPython, IronPython, and variants such as Stackless Python. It provides the best experience for debugging pure Python code. When you attach to a running <code>python.exe</code> process, this launcher is used. This launcher also provides mixed-mode debugging for CPython, allowing you to step seamlessly between C/C++ code and Python code.
Web launcher	Starts your default browser on launch and enables debugging of templates. See the Web template debugging section for more information.
Django Web launcher	Identical to the Web launcher and shown only for backwards compatibility.
IronPython (.NET) launcher	Uses the .NET debugger, which only works with IronPython but allows for stepping between any .NET language project, including C# and VB. This launcher is used if you attach to a running .NET process that is hosting IronPython.

Run options (search paths, startup arguments, and environment variables)

OPTION	DESCRIPTION
Search Paths	These values match what's shown in the project's Search Paths node in Solution Explorer . You can modify this value here, but it's easier to use Solution Explorer that lets you browse folders and automatically converts paths to relative form.
Script Arguments	These arguments are added to the command used to launch your script, appearing after your script's filename. The first item here is available to your script as <code>sys.argv[1]</code> , the second as <code>sys.argv[2]</code> , and so on.

OPTION	DESCRIPTION
Interpreter Arguments	<p>These arguments are added to the launcher command line before the name of your script. Common arguments here are <code>-w ...</code> to control warnings, <code>-o</code> to slightly optimize your program, and <code>-u</code> to use unbuffered IO. IronPython users are likely to use this field to pass <code>-X</code> options, such as <code>-X:Frames</code> or <code>-X:MTA</code>.</p>
Interpreter Path	<p>Overrides the path associated with the current environment. The value may be useful for launching your script with a non-standard interpreter.</p>
Environment Variables	<p>In this multi-line text box, add entries of the form <code><NAME>=<VALUE></code>. Because this setting is applied last, on top of any existing global environment variables, and after <code>PYTHONPATH</code> is set according to the Search Paths setting, it can be used to manually override any of those other variables.</p>

Immediate and Interactive windows

There are two interactive windows you can use during a debugging session: the standard Visual Studio **Immediate** window, and the **Python Debug Interactive** window.

The **Immediate** window (**Debug > Windows > Immediate**) is used for quick evaluation of Python expressions and inspection or assignment of variables within the running program. See the general [Immediate window](#) article for details.

The **Python Debug Interactive** window (**Debug > Windows > Python Debug Interactive**) is richer as it makes the full [Interactive REPL](#) experience available while debugging, including writing and running code. It automatically connects to any process started in the debugger using the Standard Python launcher (including processes attached through **Debug > Attach to Process**). It's not, however, available when using mixed-mode C/C++ debugging.

```

Program.py ✘ Utility.py
volume
5 def worker():
6     msg = 'ping'
7     while True:
8         time.sleep(interval)
9         Utility.LogText(msg)
10
11 t = threading.Thread(target=worker)
12 t.daemon = True
13 t.start()
14
15 def volume(width, height, depth):
16     vol = width * height * depth
17     return vol
18     vol| 117.6
19 def is_boxAuthorized(box_width, box_height, box_depth):
20     '''check if the box is small enough'''
21     max = 5000
22     box_vol = volume(box_width, box_height, box_depth)
23     return box_vol < max
24
25 authorized = is_boxAuthorized(5, 2, 12)

```

100 %

Python Debug Interactive

```

<CurrentFrame>
>>> $where
=> Frame id=0, function=volume
  Frame id=1, function=is_boxAuthorized
    Frame id=2, function=<module>
>>> vol
120
>>> def adjust(volume, factor):
...     return volume * factor
...
>>> vol = adjust(vol, 0.98)
>>> vol
117.6
>>>

```

Python Debug Interactive | Autos | Locals | Watch 1

The **Debug Interactive** window supports special meta-commands in addition to the standard REPL commands:

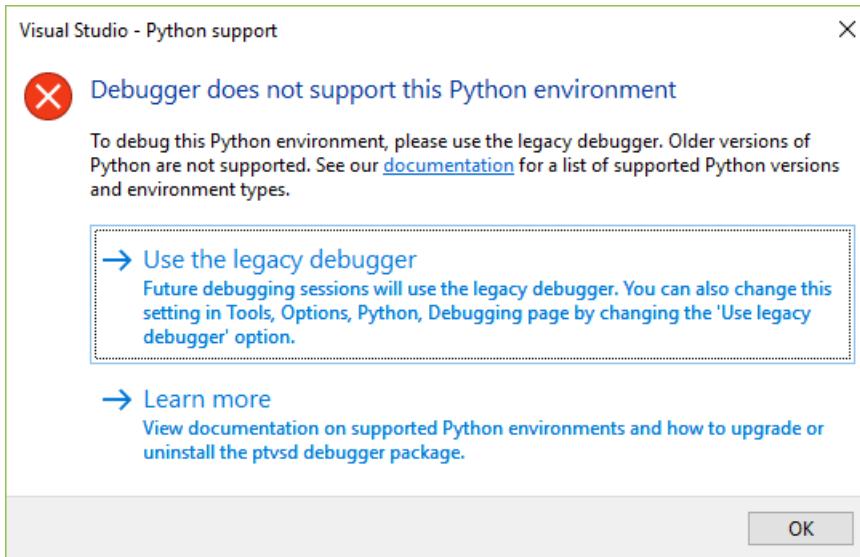
COMMAND	ARGUMENTS	DESCRIPTION
\$continue , \$cont , \$c		Starts running the program from the current statement.
\$down , \$d		Move the current frame one level down in the stack trace.
\$frame		Displays the current frame ID.
\$frame	frame ID	Switches the current frame to the specified frame ID.
\$load		Loads commands from file and executes until complete
\$proc		Displays the current process ID.
\$proc	process ID	Switches the current process to the specified process ID.
\$procs		Lists the processes currently being debugged.

COMMAND	ARGUMENTS	DESCRIPTION
<code>\$stepin</code> , <code>\$step</code> , <code>\$s</code>	Steps into the next function call, if possible.	
<code>\$stepout</code> , <code>\$return</code> , <code>\$r</code>	Steps out of the current function.	
<code>\$stepover</code> , <code>\$until</code> , <code>\$unt</code>	Steps over the next function call.	
<code>\$thread</code>		Displays the current thread ID.
<code>\$thread</code>	thread ID	Switches the current thread to the specified thread ID.
<code>\$threads</code>		Lists the threads currently being debugged.
<code>\$up</code> , <code>\$u</code>		Move the current frame one level up in the stack trace.
<code>\$where</code> , <code>\$w</code> , <code>\$bt</code>	Lists the frames for the current thread.	

Note that the standard debugger windows such as **Processes**, **Threads**, and **Call Stack** are not synchronized with the **Debug Interactive** window. Changing the active process, thread, or frame in the **Debug Interactive** window does not affect the other debugger windows. Similarly, changing the active process, thread, or frame in the other debugger windows does not affect the **Debug Interactive** window.

Use the legacy debugger

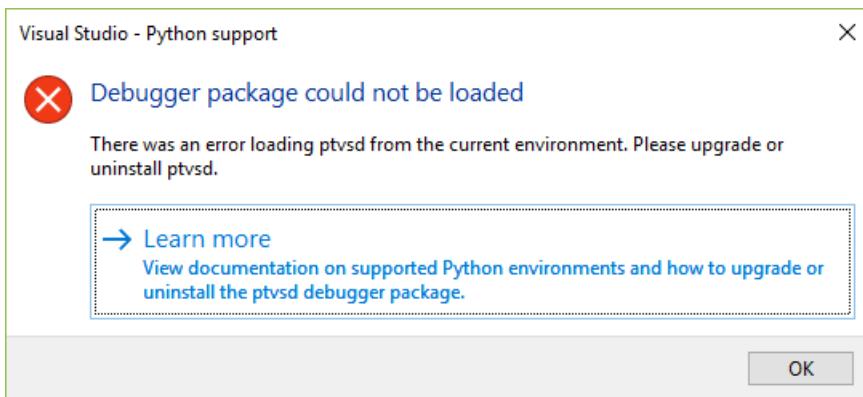
Visual Studio 2017 versions 15.8 and later use a debugger based on ptvsd version 4.1+. This version of ptvsd is compatible with Python 2.7 and Python 3.5+. If you're using Python 2.6, 3.1 to 3.4, or IronPython, Visual Studio shows the error, **Debugger does not support this Python environment**:



In these cases you must use the older debugger (which is the default in Visual Studio 2017 versions 15.7 and earlier). Select the **Tools** > **Options** menu command, navigate to **Python** > **Debugging**, and select the **Use legacy debugger** option.

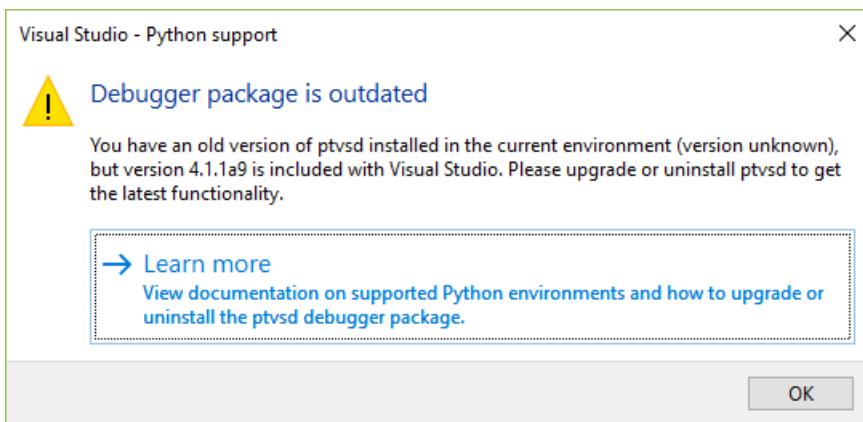
If you've installed an older version of ptvsd in the current environment (such as an earlier 4.0.x version, or a 3.x version required for remote debugging), Visual Studio may show an error or warning.

The error, **Debugger package could not be loaded**, appears when you've installed ptvsd 3.x:



In this case, select **Use the legacy debugger** to set the **Use legacy debugger** option, and restart the debugger.

The warning, **Debugger package is outdated**, appears when you've installed an earlier 4.x version of ptvsd:

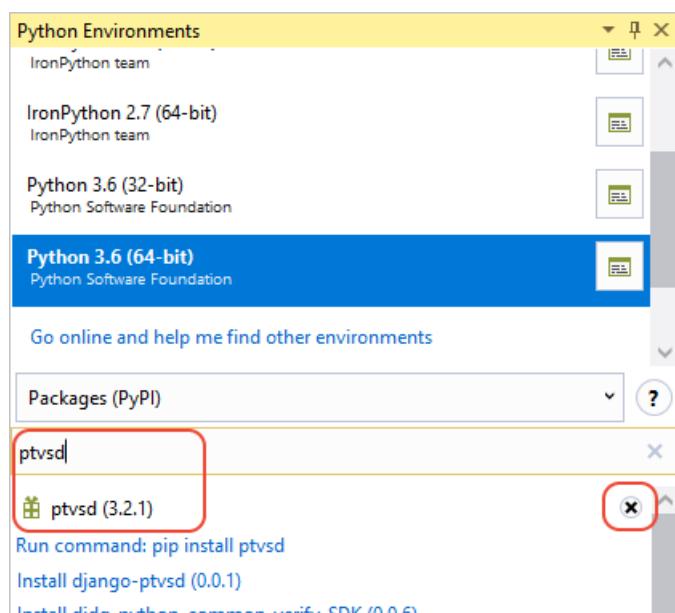


IMPORTANT

Although you may choose to ignore the warning for some versions of ptvsd, Visual Studio may not work correctly.

To manage your ptvsd installation:

1. Navigate to the **Packages** tab in the **Python Environments** window.
2. Enter "ptvsd" in the search box and examine the installed version of ptvsd:

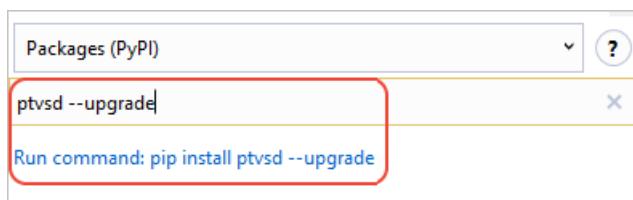


- If the version is lower than 4.1.1a9 (the version bundled with Visual Studio), select the **X** to the right of the package to uninstall the older version. Visual Studio then uses its bundled version. (You can also uninstall from PowerShell using `pip uninstall ptvsd`)
- Alternately, you can update the ptvsd package to its newest version by following the instructions in the [Troubleshooting](#) section.

Troubleshooting

If you have issues with the debugger, first upgrade your version of ptvsd as follows:

- Navigate to the **Packages** tab in the **Python Environments** window.
- Enter `ptvsd --upgrade` in the search box, then select **Run command: pip install ptvsd --upgrade**. (You can also use the same command from PowerShell.)



If issues persist, please file an issue on the [PTVS GitHub repository](#).

Enable debugger logging

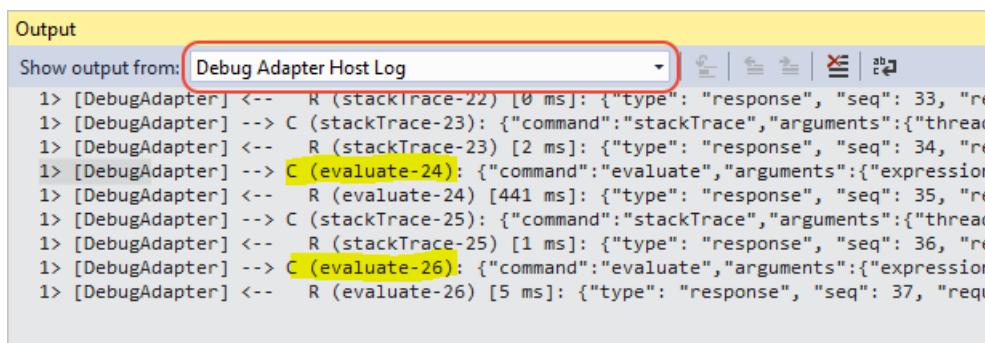
In the course of investigating a debugger issue, Microsoft may ask you to enable and collect debugger logs that help in diagnosis.

The following steps enable debugging in the current Visual Studio session:

- Open a command window in Visual Studio using the **View > Other Windows > Command Window** menu command.
- Enter the following command:

```
DebugAdapterHost.Logging /On /OutputWindow
```

- Start debugging and go through whatever steps are necessary to reproduce your issue. During this time, debug logs appear in the **Output** window under **Debug Adapter Host Log**. You can then copy the logs from that window and paste into a GitHub issue, email, etc.



- If Visual Studio hangs or you are otherwise not able to access the **Output** window, restart Visual Studio, open a command window, and enter the following command:

```
DebugAdapterHost.Logging /On
```

5. Start debugging and reproduce your issue again. The debugger logs can then be found in

`%temp%\DebugAdapterHostLog.txt`.

See also

For complete details on the Visual Studio debugger, see [Debugging in Visual Studio](#).

Remotely debug Python code on Linux

4/9/2019 • 7 minutes to read • [Edit Online](#)

Visual Studio can launch and debug Python applications locally and remotely on a Windows computer (see [Remote debugging](#)). It can also debug remotely on a different operating system, device, or Python implementation other than CPython using the [ptvsd library](#).

When using ptvsd, the Python code being debugged hosts the debug server to which Visual Studio can attach. This hosting requires a small modification to your code to import and enable the server, and may require network or firewall configurations on the remote computer to allow TCP connections.



For an introduction to remote debugging, see [Deep Dive: Cross-platform remote debugging](#) (youtube.com, 6m22s), which is applicable to both Visual Studio 2015 and 2017.

Set up a Linux computer

The following items are needed to follow this walkthrough:

- A remote computer running Python on an operating system like Mac OSX or Linux.
- Port 5678 (inbound) opened on that computer's firewall, which is the default for remote debugging.

You can easily create a [Linux virtual machine on Azure](#) and [access it using Remote Desktop](#) from Windows.

Ubuntu for the VM is convenient because Python is installed by default; otherwise, see the list on [Install a Python interpreter of your choice](#) for additional Python download locations.

For details on creating a firewall rule for an Azure VM, see [Open ports to a VM in Azure using the Azure portal](#).

Prepare the script for debugging

1. On the remote computer, create a Python file called *guessing-game.py* with the following code:

```
import random

guesses_made = 0
name = input('Hello! What is your name?\n')
number = random.randint(1, 20)
print('Well, {0}, I am thinking of a number between 1 and 20.'.format(name))

while guesses_made < 6:
    guess = int(input('Take a guess: '))
    guesses_made += 1
    if guess < number:
        print('Your guess is too low.')
    if guess > number:
        print('Your guess is too high.')
    if guess == number:
        break
if guess == number:
    print('Good job, {0}! You guessed my number in {1} guesses!'.format(name, guesses_made))
else:
    print('Nope. The number I was thinking of was {0}'.format(number))
```

2. Install the `ptvsd` package into your environment using `pip3 install ptvsd`.

NOTE

It's a good idea to record the version of `ptvsd` that's installed in case you need it for troubleshooting; the [ptvsd listing](#) also shows available versions.

3. Enable remote debugging by adding the code below at the earliest possible point in `guessing-game.py`, before other code. (Though not a strict requirement, it's impossible to debug any background threads spawned before the `enable_attach` function is called.)

```
import ptvsd  
ptvsd.enable_attach()
```

4. Save the file and run `python3 guessing-game.py`. The call to `enable_attach` runs in the background and waits for incoming connections as you otherwise interact with the program. If desired, the `wait_for_attach` function can be called after `enable_attach` to block the program until the debugger attaches.

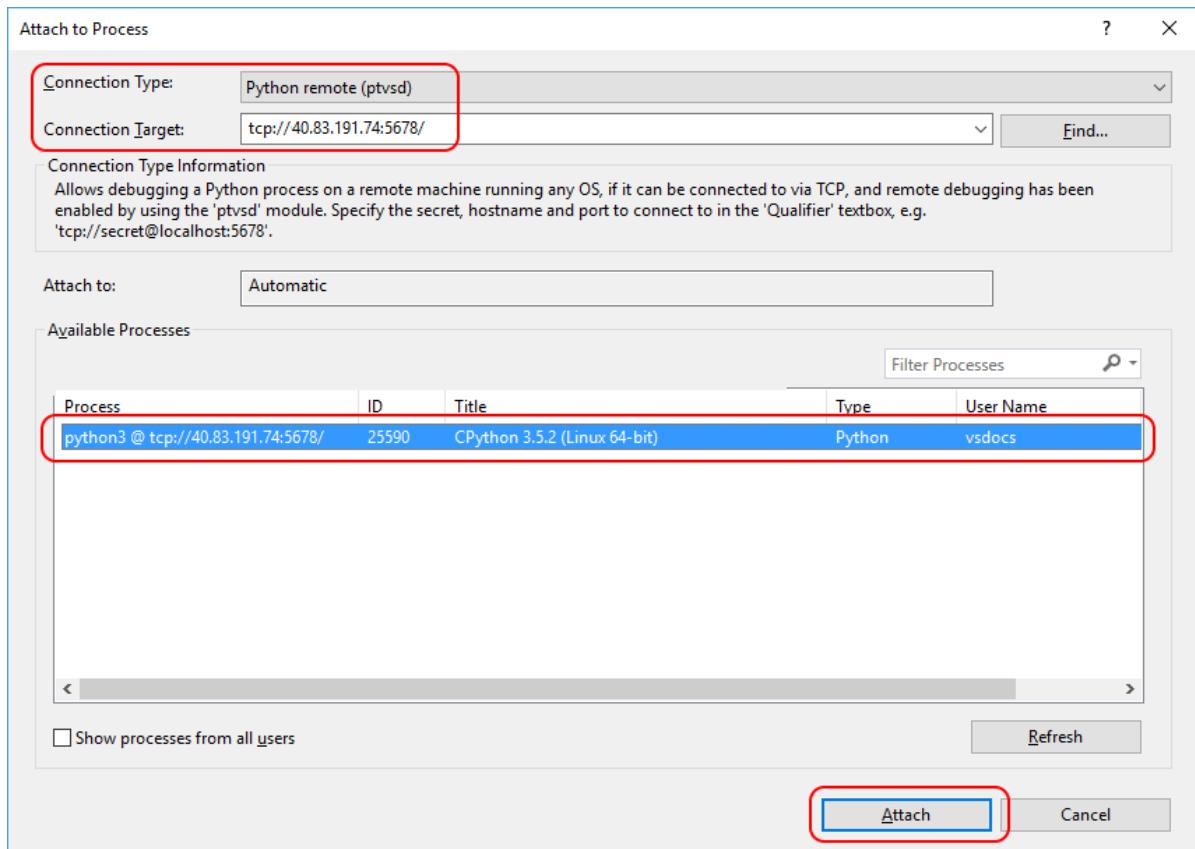
TIP

In addition to `enable_attach` and `wait_for_attach`, `ptvsd` also provides a helper function `break_into_debugger`, which serves as a programmatic breakpoint if the debugger is attached. There is also an `is_attached` function that returns `True` if the debugger is attached (note that there is no need to check this result before calling any other `ptvsd` functions).

Attach remotely from Python Tools

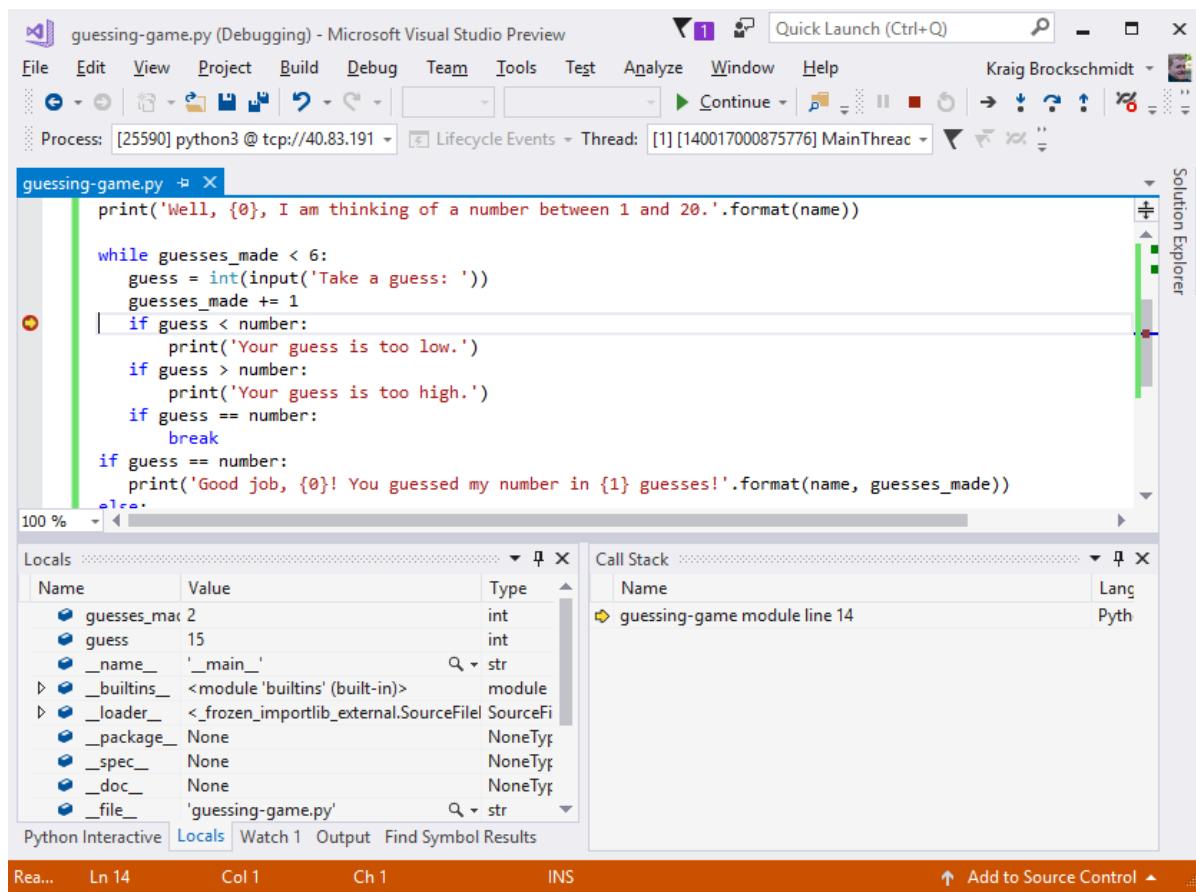
In these steps, we set a simple breakpoint to stop the remote process.

1. Create a copy of the remote file on the local computer and open it in Visual Studio. It doesn't matter where the file is located, but its name should match the name of the script on the remote computer.
2. (Optional) To have IntelliSense for `ptvsd` on your local computer, install the `ptvsd` package into your Python environment.
3. Select **Debug > Attach to Process**.
4. In the **Attach to Process** dialog that appears, set **Connection Type** to **Python remote (ptvsd)**. (On older versions of Visual Studio these commands are named **Transport** and **Python remote debugging**.)
5. In the **Connection Target** field (**Qualifier** on older versions), enter `tcp://<ip_address>:5678` where `<ip_address>` is that of the remote computer (which can be either an explicit address or a name like `myvm.cloudapp.net`), and `:5678` is the remote debugging port number.
6. Press **Enter** to populate the list of available `ptvsd` processes on that computer:



If you happen to start another program on the remote computer after populating this list, select the **Refresh** button.

7. Select the process to debug and then **Attach**, or double-click the process.
8. Visual Studio then switches into debugging mode while the script continues to run on the remote computer, providing all the usual [debugging](#) capabilities. For example, set a breakpoint on the `if guess < number:` line, then switch over to the remote computer and enter another guess. After you do so, Visual Studio on your local computer stops at that breakpoint, shows local variables, and so on:



9. When you stop debugging, Visual Studio detaches from the program, which continues to run on the remote computer. ptvsd also continues listening for attaching debuggers, so you can reattach to the process again at any time.

Connection troubleshooting

1. Make sure that you've selected **Python remote (ptvsd)** for the **Connection Type (Python remote debugging for Transport)** with older versions.)
2. Check that the secret in the **Connection Target (or Qualifier)** exactly matches the secret in the remote code.
3. Check that the IP address in the **Connection Target (or Qualifier)** matches that of the remote computer.
4. Check that you're opened the remote debugging port on the remote computer, and that you've included the port suffix in the connection target, such as `:5678`.
 - If you need to use a different port, you can specify it in the `enable_attach` call using the `address` argument, as in `ptvsd.enable_attach(address = ('0.0.0.0', 8080))`. In this case, open that specific port in the firewall.
5. Check that the version of ptvsd installed on the remote computer as returned by `pip3 list` matches that used by the version of the Python tools you're using in Visual Studio in the table below. If necessary, update ptvsd on the remote computer.

VISUAL STUDIO VERSION	PYTHON TOOLS/PTVSD VERSION
2017 15.8	4.1.1a9 (legacy debugger: 3.2.1.0)
2017 15.7	4.1.1a1 (legacy debugger: 3.2.1.0)
2017 15.4, 15.5, 15.6	3.2.1.0

VISUAL STUDIO VERSION	PYTHON TOOLS/PTVSD VERSION
2017 15.3	3.2.0
2017 15.2	3.1.0
2017 15.0, 15.1	3.0.0
2015	2.2.6
2013	2.2.2
2012, 2010	2.1

Using ptvsd 3.x

The following information applies only to remote debugging with ptvsd 3.x, which contains certain features that were removed in ptvsd 4.x.

1. With ptvsd 3.x, the `enable_attach` function required that you pass a "secret" as the first argument that restricts access to the running script. You enter this secret when attaching the remote debugger. Though not recommended, you can allow anyone to connect, use `enable_attach(secret=None)`.
2. The connection target URL is `tcp://<secret>@<ip_address>:5678` where `<secret>` is the string passed `enable_attach` in the Python code.

By default, the connection to the ptvsd 3.x remote debug server is secured only by the secret and all data is passed in plain text. For a more secure connection, ptvsd 3.x supports SSL using the `tcsp` protocol, which you set up as follows:

1. On the remote computer, generate separate self-signed certificate and key files using openssl:

```
openssl req -new -x509 -days 365 -nodes -out cert.cer -keyout cert.key
```

When prompted, use the hostname or IP address (whichever you use to connect) for the **Common Name** when prompted by openssl.

(See [Self-signed certificates](#) in the Python `ssl` module docs for additional details. Note that the command in those docs generates only a single combined file.)

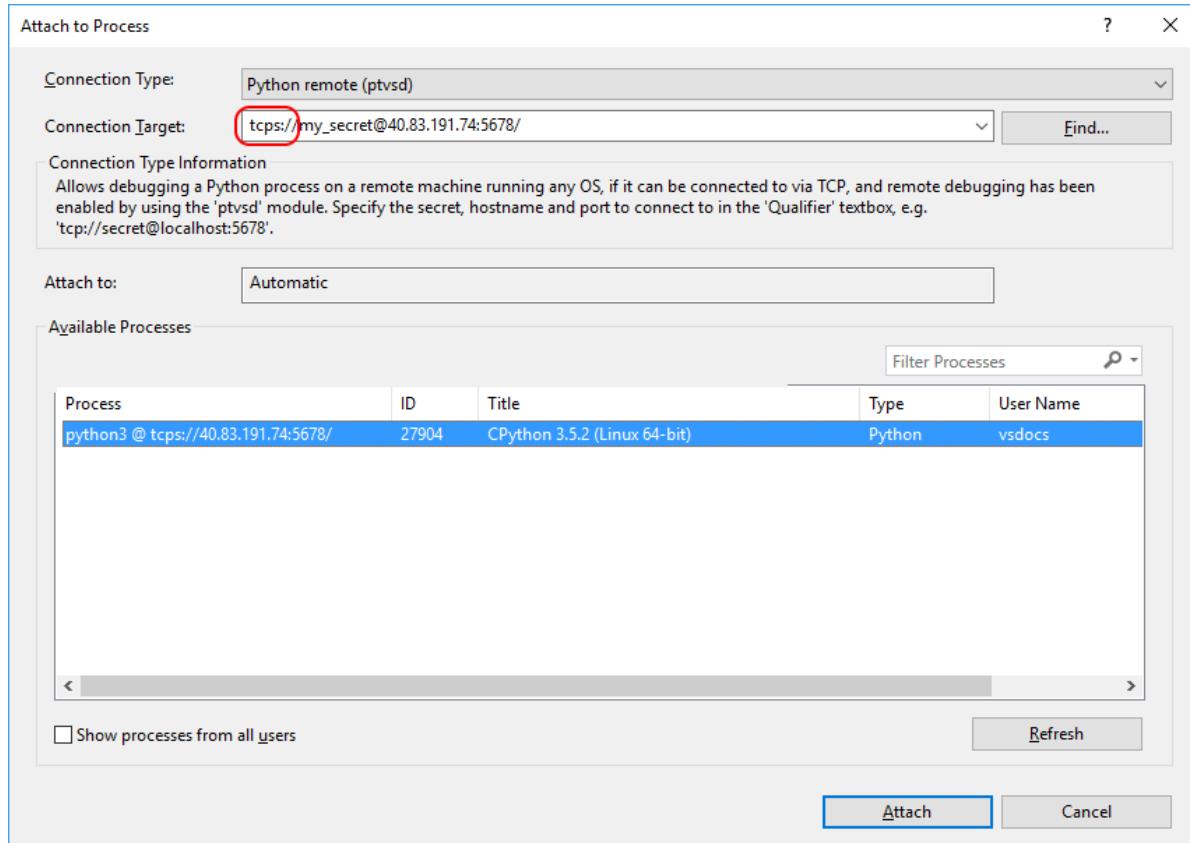
2. In the code, modify the call to `enable_attach` to include `certfile` and `keyfile` arguments using the filenames as the values (these arguments have the same meaning as for the standard `ssl.wrap_socket` Python function):

```
ptvsd.enable_attach(secret='my_secret', certfile='cert.cer', keyfile='cert.key')
```

You can also make the same change in the code file on the local computer, but because this code isn't actually run, it isn't strictly necessary.

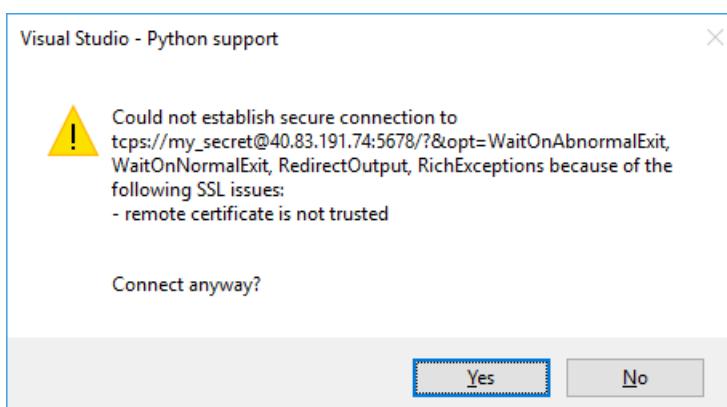
3. Restart the Python program on the remote computer, making it ready for debugging.
4. Secure the channel by adding the certificate to Trusted Root CA on the Windows computer with Visual Studio:

- a. Copy the certificate file from the remote computer to the local computer.
 - b. Open **Control Panel** and navigate to **Administrative Tools > Manage computer certificates**.
 - c. In the window that appears, expand **Trusted Root Certification Authorities** on the left side, right-click **Certificates**, and select **All Tasks > Import**.
 - d. Navigate to and select the .cer file copied from the remote computer, then click through the dialogs to complete the import.
5. Repeat the attach process in Visual Studio as described earlier, now using `tcp://` as the protocol for the **Connection Target** (or Qualifier).



6. Visual Studio prompts you about potential certificate issues when connecting over SSL. You may ignore the warnings and proceed, but although the channel is still encrypted against eavesdropping it can be open to man-in-the-middle attacks.

- a. If you see the **remote certificate is not trusted** warning below, it means you did not properly add the certificate to the Trusted Root CA. Check those steps and try again.



- b. If you see the **remote certificate name does not match hostname** warning below, it means you did not use the proper hostname or IP address as the **Common Name** when creating the certificate.

Visual Studio - Python support



Could not establish secure connection to
tcp://my_secret@40.83.191.74:5678/?&opt=WaitOnAbnormalExit,
WaitOnNormalExit, RedirectOutput, RichExceptions because of the
following SSL issues:
- remote certificate name does not match hostname

Connect anyway?

Publish to Azure App Service

4/9/2019 • 2 minutes to read • [Edit Online](#)

At present, Python is supported on Azure App Service for Linux, and you can publish apps using [Git deploy](#) and [containers](#), as described in this article.

NOTE

Python support on Azure App Service for Windows is officially deprecated. As a result, the **Publish** command in Visual Studio is officially supported only for an [IIS target](#), and remote debugging on Azure App Service is no longer officially supported.

However, [Publishing to App Service on Windows](#) features still work for the time being, as the Python extensions for App Service on Windows remain available but will not be serviced or updated.

Publish to App Service on Linux using Git deploy

Git deploy connects an App Service on Linux to a specific branch of a Git repository. Committing code to that branch automatically deploys to the App Service, and App Service automatically installs any dependencies listed in *requirements.txt*. In this case, App Service on Linux runs your code in a pre-configured container image that uses the Gunicorn web server. At present, this service is in Preview and not supported for production use.

For more information, see the following articles in the Azure documentation:

- [Quickstart: Create a Python web app in App Service](#) provides a short walkthrough of the Git deploy process using a simple Flask app and deployment from a local Git repository.
- [How to configure Python](#) describes the characteristics of the App Service on Linux container and how to customize the Gunicorn startup command for your app.

Publish to App Service on Linux using containers

Instead of relying on the pre-built container with App Service on Linux, you can provide your own container. This option allows you to choose which web servers you use and to customize the container's behavior.

There are two options to build, manage, and push containers:

- Use Visual Studio Code and the Docker extension, as described on [Deploy Python using Docker Containers](#). Even if you don't use Visual Studio Code, the article provides helpful details on building container images for Flask and Django apps using the production-ready uwsgi and nginx web servers. You can then deploy those same container using the Azure CLI.
- Use the command line and Azure CLI, as described on [Use a custom Docker image](#) in the Azure documentation. This guide is generic, however, and not specific to Python.

Publish to IIS

From Visual Studio, you can publish to a Windows virtual machine or other IIS-capable computer with the **Publish** command. When using IIS, be sure to create or modify a *web.config* file in the app that tells IIS where to find the Python interpreter. For more information, see [Configure web apps for IIS](#).

Create a C++ extension for Python

4/9/2019 • 18 minutes to read • [Edit Online](#)

Modules written in C++ (or C) are commonly used to extend the capabilities of a Python interpreter as well as to enable access to low-level operating system capabilities. There are three primary types of modules:

- Accelerator modules: because Python is an interpreted language, certain pieces of code can be written in C++ for higher performance.
- Wrapper modules: expose existing C/C++ interfaces to Python code or expose a more "Pythonic" API that's easy to use from Python.
- Low-level system access modules: created to access lower-level features of the CPython runtime, the operating system, or the underlying hardware.

This article walks through building a C++ extension module for CPython that computes a hyperbolic tangent and calls it from Python code. The routine is implemented first in Python to demonstrate the relative performance gain of implementing the same routine in C++.

This article also demonstrates two ways to make the C++ available to Python:

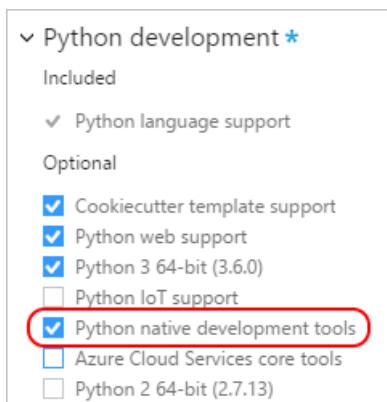
- The standard CPython extensions as described in the [Python documentation](#)
- [PyBind11](#), which is recommended for C++ 11 because of its simplicity.

A comparison between these and other means is found under [alternative approaches](#) at the end of this article.

The completed sample from this walkthrough can be found on [python-samples-vs-cpp-extension](#) (GitHub).

Prerequisites

- Visual Studio 2017 or later with both the **Desktop Development with C++** and **Python Development** workloads installed with default options.
- In the **Python Development** workload, also select the box on the right for **Python native development tools**. This option sets up most of the configuration described in this article. (This option also includes the C++ workload automatically.)



TIP

Installing the **Data science and analytical applications** workload also includes Python and the **Python native development tools** option by default.

For more information, see [Install Python support for Visual Studio](#), including using other versions of Visual Studio. If you install Python separately, be sure to select **Download debugging symbols** and **Download debug binaries** under **Advanced Options** in the installer. This option ensures that you have the necessary debug libraries available if you choose to do a debug build.

Create the Python application

1. Create a new Python project in Visual Studio by selecting **File > New > Project**. Search for "Python", select the **Python Application** template, give it a suitable name and location, and select **OK**.
2. Working with C++ requires that you use a 32-bit Python interpreter (Python 3.6 or above recommended). In the **Solution Explorer** window of Visual Studio, expand the project node, then expand the **Python Environments** node. If you don't see a 32-bit environment as the default (either in bold, or labeled with **global default**), then follow the instructions on [Select a Python environment for a project](#). If you don't have a 32-bit interpreter installed, see [Install Python interpreters](#).
3. In the project's .py file, paste the following code that benchmarks the computation of a hyperbolic tangent (implemented without using the math library for easier comparison). Feel free to enter the code manually to experience some of the [Python editing features](#).

```
from itertools import islice
from random import random
from time import perf_counter

COUNT = 500000 # Change this value depending on the speed of your computer
DATA = list(islice(iter(lambda: (random() - 0.5) * 3.0, None), COUNT))

e = 2.7182818284590452353602874713527

def sinh(x):
    return (1 - (e ** (-2 * x))) / (2 * (e ** -x))

def cosh(x):
    return (1 + (e ** (-2 * x))) / (2 * (e ** -x))

def tanh(x):
    tanh_x = sinh(x) / cosh(x)
    return tanh_x

def test(fn, name):
    start = perf_counter()
    result = fn(DATA)
    duration = perf_counter() - start
    print('{} took {:.3f} seconds\n\n'.format(name, duration))

    for d in result:
        assert -1 <= d <= 1, "incorrect values"

if __name__ == "__main__":
    print('Running benchmarks with COUNT = {}'.format(COUNT))

    test(lambda d: [tanh(x) for x in d], '[tanh(x) for x in d] (Python implementation)')
```

4. Run the program using **Debug > Start without Debugging (Ctrl+F5)** to see the results. You can adjust the `COUNT` variable to change how long the benchmark takes to run. For the purposes of this walkthrough, set the count so that the benchmark take around two seconds.

TIP

When running benchmarks, always use **Debug > Start without Debugging** to avoid the overhead incurred when running code within the Visual Studio debugger.

Create the core C++ projects

Follow the instructions in this section to create two identical C++ projects named "superfastcode" and "superfastcode2". Later you'll use different means in each project to expose the C++ code to Python.

1. Make sure the `PYTHONHOME` environment variable is set to the Python interpreter you want to use. The C++ projects in Visual Studio rely on this variable to locate files such as `python.h`, which are used when creating a Python extension.
2. Right-click the solution in **Solution Explorer** and select **Add > New Project**. A Visual Studio solution can contain both Python and C++ projects together (which is one of the advantages of using Visual Studio for Python).
3. Search on "C++", select **Empty project**, specify the name "superfastcode" ("superfastcode2" for the second project), and select **OK**.

TIP

With the **Python native development tools** installed in Visual Studio, you can start with the **Python Extension Module** template instead, which has much of what's described below already in place. For this walkthrough, though, starting with an empty project demonstrates building the extension module step by step. Once you understand the process, the template saves you time when writing your own extensions.

4. Create a C++ file in the new project by right-clicking the **Source Files** node, then select **Add > New Item**, select **C++ File**, name it `module.cpp`, and select **OK**.

IMPORTANT

A file with the `.cpp` extension is necessary to turn on the C++ property pages in the steps that follow.

5. Right-click the C++ project in **Solution Explorer**, select **Properties**.
6. At the top of the **Property Pages** dialog that appears, set **Configuration** to **All Configurations** and **Platform** to **Win32**.
7. Set the specific properties as described in the following table, then select **OK**.

TAB	PROPERTY	VALUE
General	General > Target Name	Specify the name of the module as you want to refer to it from Python in <code>from...import</code> statements. You use this same name in the C++ when defining the module for Python. If you want to use the name of the project as the module name, leave the default value of \$(ProjectName) .
	General > Target Extension	<code>.pyd</code>

TAB	PROPERTY	VALUE
	Project Defaults > Configuration Type	Dynamic Library (.dll)
C/C++ > General	Additional Include Directories	Add the Python <i>include</i> folder as appropriate for your installation, for example, <code>c:\Python36\include</code> .
C/C++ > Preprocessor	Preprocessor Definitions	CPython only: add <code>Py_LIMITED_API;</code> to the beginning of the string (including the semicolon). This definition restricts some of the functions you can call from Python and makes the code more portable between different versions of Python. If you're working with PyBind11, don't add this definition, otherwise you'll see build errors.
C/C++ > Code Generation	Runtime Library	Multi-threaded DLL (/MD) (see Warning below)
Linker > General	Additional Library Directories	Add the Python <i>libs</i> folder containing <i>.lib</i> files as appropriate for your installation, for example, <code>c:\Python36\libs</code> . (Be sure to point to the <i>libs</i> folder that contains <i>.lib</i> files, and <i>not</i> the <i>Lib</i> folder that contains <i>.py</i> files.)

TIP

If you don't see the C/C++ tab in the project properties, it's because the project doesn't contain any files that it identifies as C/C++ source files. This condition can occur if you create a source file without a `.c` or `.cpp` extension. For example, if you accidentally entered `module.coo` instead of `module.cpp` in the new item dialog earlier, then Visual Studio creates the file but doesn't set the file type to "C/C+ Code," which is what activates the C/C++ properties tab. Such misidentification remains the case even if you rename the file with `.cpp`. To set the file type properly, right-click the file in **Solution Explorer**, select **Properties**, then set **File Type** to **C/C++ Code**.

WARNING

Always set the **C/C++ > Code Generation > Runtime Library** option to **Multi-threaded DLL (/MD)**, even for a debug configuration, because this setting is what the non-debug Python binaries are built with. With CPython, if you happen to set the **Multi-threaded Debug DLL (/MDd)** option, building a **Debug** configuration produces error **C1189: Py_LIMITED_API is incompatible with Py_DEBUG, Py_TRACE_REFS, and Py_REF_DEBUG**. Furthermore, if you remove `Py_LIMITED_API` (which is required with CPython, but not PyBind11) to avoid the build error, Python crashes when attempting to import the module. (The crash happens within the DLL's call to `PyModule_Create` as described later, with the output message of **Fatal Python error: PyThreadState_Get: no current thread.**)

The `/MDd` option is used to build the Python debug binaries (such as `python_d.exe`), but selecting it for an extension DLL still causes the build error with `Py_LIMITED_API`.

8. Right-click the C++ project and select **Build** to test your configurations (both **Debug** and **Release**). The **.pyd** files are located in the **solution** folder under **Debug** and **Release**, not the C++ project folder itself.

9. Add the following code to the C++ project's *module.cpp* file:

```
#include <Windows.h>
#include <cmath>

const double e = 2.7182818284590452353602874713527;

double sinh_impl(double x) {
    return (1 - pow(e, (-2 * x))) / (2 * pow(e, -x));
}

double cosh_impl(double x) {
    return (1 + pow(e, (-2 * x))) / (2 * pow(e, -x));
}

double tanh_impl(double x) {
    return sinh_impl(x) / cosh_impl(x);
}
```

10. Build the C++ project again to confirm that your code is correct.

11. If you haven't already done so, repeat the steps above to create a second project named "superfastcode2" with identical contents.

Convert the C++ projects to extensions for Python

To make the C++ DLL into an extension for Python, you first modify the exported methods to interact with Python types. You then add a function that exports the module, along with definitions of the module's methods.

The sections that follow explain how to perform these steps using both the CPython extensions and PyBind11.

C_{Py}thon extensions

For background on what's shown in this section for Python 3.x, refer to the [Python/C API Reference Manual](#) and especially [Module Objects](#) on python.org (remember to select your version of Python from the drop-down control on the upper right to view the correct documentation).

If you're working with Python 2.7, refer instead to [Extending Python 2.7 with C or C++](#) and [Porting Extension Modules to Python 3](#) (python.org).

1. At the top of *module.cpp*, include *Python.h*:

```
#include <Python.h>
```

2. Modify the `tanh_impl` method to accept and return Python types (a `PyObject*`, that is):

```
PyObject* tanh_impl(PyObject *, PyObject* o) {
    double x = PyFloat_AsDouble(o);
    double tanh_x = sinh_impl(x) / cosh_impl(x);
    return PyFloat_FromDouble(tanh_x);
}
```

3. Add a structure that defines how the C++ `tanh_impl` function is presented to Python:

```

static PyMethodDef superfastcode_methods[] = {
    // The first property is the name exposed to Python, fast_tanh, the second is the C++
    // function name that contains the implementation.
    { "fast_tanh", (PyCFunction)tanh_impl, METH_O, nullptr },

    // Terminate the array with an object containing nulls.
    { nullptr, nullptr, 0, nullptr }
};


```

4. Add a structure that defines the module as you want to refer to it in your Python code, specifically when using the `from...import` statement. (Make this match the value in the project properties under **Configuration Properties > General > Target Name**.) In the following example, the "superfastcode" module name means you can use `from superfastcode import fast_tanh` in Python, because `fast_tanh` is defined within `superfastcode_methods`. (Filenames internal to the C++ project, like `module.cpp`, are inconsequential.)

```

static PyModuleDef superfastcode_module = {
    PyModuleDef_HEAD_INIT,
    "superfastcode",                  // Module name to use with Python import statements
    "Provides some functions, but faster", // Module description
    0,
    superfastcode_methods            // Structure that defines the methods of the module
};


```

5. Add a method that Python calls when it loads the module, which must be named `PyInit_<module-name>`, where `<module-name>` exactly matches the C++ project's **General > Target Name** property (that is, it matches the filename of the `.pyd` built by the project).

```

PyMODINIT_FUNC PyInit_superfastcode() {
    return PyModule_Create(&superfastcode_module);
}


```

6. Set the target configuration to **Release** and build the C++ project again to verify your code. If you encounter errors, see the [Troubleshooting](#) section below.

PyBind11

If you completed the steps in the previous section, you certainly noticed that you used lots of boilerplate code to create the necessary module structures for the C++ code. PyBind11 simplifies the process through macros in a C++ header file that accomplish the same result with much less code. For background on what's shown in this section, see [PyBind11 basics](#) (github.com).

1. Install PyBind11 using pip: `pip install pybind11` or `py -m pip install pybind11`.
2. At the top of `module.cpp`, include `pybind11.h`:

```
#include <pybind11/pybind11.h>
```

3. At the bottom of `module.cpp`, use the `PYBIND11_MODULE` macro to define the entrypoint to the C++ function:

```

namespace py = pybind11;

PYBIND11_MODULE(superfastcode2, m) {
    m.def("fast_tanh2", &tanh_impl, R"pbdoc(
        Compute a hyperbolic tangent of a single argument expressed in radians.
    )pbdoc");

    #ifdef VERSION_INFO
        m.attr("__version__") = VERSION_INFO;
    #else
        m.attr("__version__") = "dev";
    #endif
}

```

- Set the target configuration to **Release** and build the C++ project to verify your code. If you encounter errors, see the next section on troubleshooting.

Troubleshooting

The C++ module may fail to compile for the following reasons:

- Unable to locate *Python.h* (**E1696: cannot open source file "Python.h"** and/or **C1083: Cannot open include file: "Python.h": No such file or directory**): verify that the path in **C/C++ > General > Additional Include Directories** in the project properties points to your Python installation's *include* folder. See step 6 under [Create the core C++ project](#).
- Unable to locate Python libraries: verify that the path in **Linker > General > Additional Library Directories** in the project properties points to your Python installation's *libs* folder. See step 6 under [Create the core C++ project](#).
- Linker errors related to target architecture: change the C++ target's project architecture to match that of your Python installation. For example, if you're targeting x64 with the C++ project but your Python installation is x86, change the C++ project to target x86.

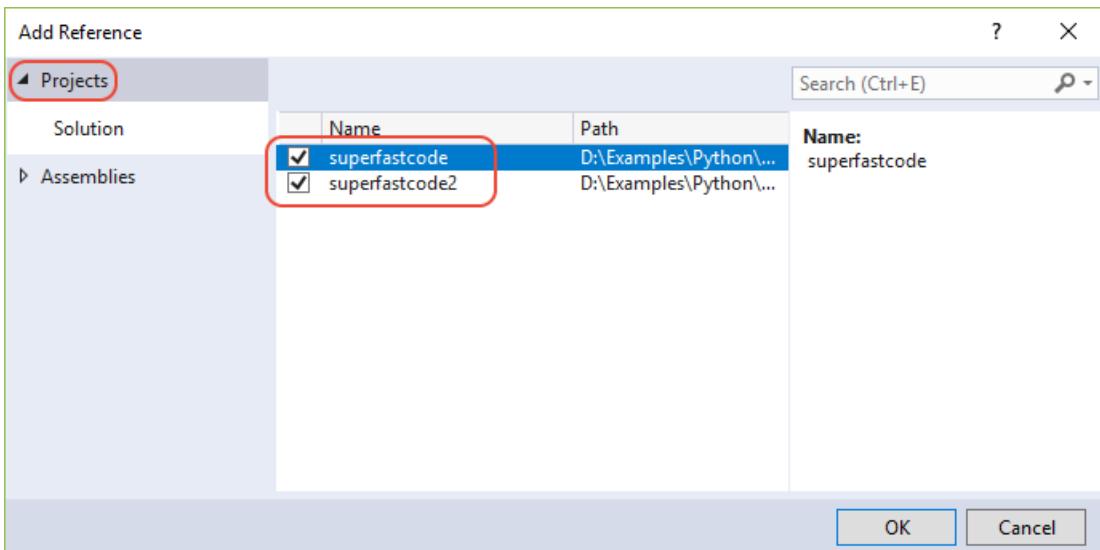
Test the code and compare the results

Now that you have the DLLs structured as Python extensions, you can refer to them from the Python project, import the modules, and use their methods.

Make the DLL available to Python

There are two ways to make the DLL available to Python.

The first method works if the Python project and the C++ project are in the same solution. Go to **Solution Explorer**, right-click the **References** node in your Python project, and then select **Add Reference**. In the dialog that appears, select the **Projects** tab, select both the **superfastcode** and **superfastcode2** projects, and then select **OK**.



The alternate method, described in the following steps, installs the module in the global Python environment, making it available to other Python projects as well. (Doing so typically requires that you refresh the IntelliSense completion database for that environment in Visual Studio 2017 version 15.5 and earlier. Refreshing is also necessary when removing the module from the environment.)

1. If you're using Visual Studio 2017 or later, run the Visual Studio installer, select **Modify**, select **Individual Components > Compilers, build tools, and runtimes > Visual C++ 2015.3 v140 toolset**. This step is necessary because Python (for Windows) is itself built with Visual Studio 2015 (version 14.0) and expects that those tools are available when building an extension through the method described here. (Note that you may need to install a 32-bit version of Python and target the DLL to Win32 and not x64.)
2. Create a file named `setup.py` in the C++ project by right-clicking the project and selecting **Add > New Item**. Then select **C++ File (.cpp)**, name the file `setup.py`, and select **OK** (naming the file with the `.py` extension makes Visual Studio recognize it as Python despite using the C++ file template). When the file appears in the editor, paste the following code into it as appropriate to the extension method:

CPython extensions (superfastcode project):

```
from distutils.core import setup, Extension, DEBUG

sfc_module = Extension('superfastcode', sources = ['module.cpp'])

setup(name = 'superfastcode', version = '1.0',
      description = 'Python Package with superfastcode C++ extension',
      ext_modules = [sfc_module]
)
```

See [Building C and C++ extensions](#) (`python.org`) for documentation on this script.

PyBind11 (superfastcode2 project):

```

import os, sys

from distutils.core import setup, Extension
from distutils import sysconfig

cpp_args = ['-std=c++11', '-stdlib=libc++', '-mmacosx-version-min=10.7']

sfc_module = Extension(
    'superfastcode2', sources = ['module.cpp'],
    include_dirs=['pybind11/include'],
    language='c++',
    extra_compile_args = cpp_args,
)

setup(
    name = 'superfastcode2',
    version = '1.0',
    description = 'Python package with superfastcode2 C++ extension (PyBind11)',
    ext_modules = [sfc_module],
)

```

- The *setup.py* code instructs Python to build the extension using the Visual Studio 2015 C++ toolset when used from the command line. Open an elevated command prompt, navigate to the folder containing the C++ project (that is, the folder that contains *setup.py*), and enter the following command:

```
pip install .
```

or:

```
py -m pip install .
```

Call the DLL from Python

After you've made the DLL available to Python as described in the previous section, you can now call the `superfastcode.fast_tanh` and `superfastcode2.fast_tanh2` functions from Python code and compare their performance to the Python implementation:

- Add the following lines in your *.py* file to call methods exported from the DLLs and display their outputs:

```

from superfastcode import fast_tanh
test(lambda d: [fast_tanh(x) for x in d], '[fast_tanh(x) for x in d] (CPython C++ extension)')

from superfastcode2 import fast_tanh2
test(lambda d: [fast_tanh2(x) for x in d], '[fast_tanh2(x) for x in d] (PyBind11 C++ extension)')

```

- Run the Python program (**Debug > Start without Debugging** or **Ctrl+F5**) and observe that the C++ routines run approximately five to twenty times faster than the Python implementation. Typical output appears as follows:

```

Running benchmarks with COUNT = 500000
[tanh(x) for x in d] (Python implementation) took 0.758 seconds
[fast_tanh(x) for x in d] (CPython C++ extension) took 0.076 seconds
[fast_tanh2(x) for x in d] (PyBind11 C++ extension) took 0.204 seconds

```

If the **Start Without Debugging** command is disabled, right-click the Python project in **Solution**

Explorer and select **Set as Startup Project**.

3. Try increasing the `COUNT` variable so that the differences are more pronounced. A **Debug** build of the C++ module also runs slower than a **Release** build because the **Debug** build is less optimized and contains various error checks. Feel free to switch between those configurations for comparison.

NOTE

In the output, you can see that the PyBind11 extension isn't as fast as the CPython extension, though it's still significantly faster than the straight Python implementation. The difference is due to a small amount of per-call overhead that PyBind11 introduces in order to make its C++ interface dramatically simpler. This per-call difference is actually quite negligible: because the test code calls the extension functions 500,000 times, the results you see here greatly amplify that overhead! Typically, a C++ function does much more work than the trivial `fast_tanh[2]` methods used here, in which case the overhead is unimportant. However, if you're implementing methods that might be called thousands of times per second, using the CPython approach can result in better performance than PyBind11.

Debug the C++ code

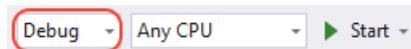
Visual Studio supports debugging Python and C++ code together. This section walks through the process using the **superfastcode** project; the steps are the same for the **superfastcode2** project.

1. Right-click the Python project in **Solution Explorer**, select **Properties**, select the **Debug** tab, and then select the **Debug > Enable native code debugging** option.

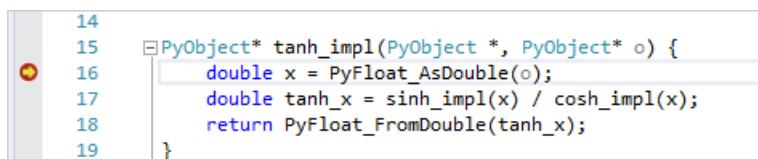
TIP

When you enable native code debugging, the Python output window may disappear immediately when the program has completed without giving you the usual **Press any key to continue** pause. To force a pause, add the `-i` option to the **Run > Interpreter Arguments** field on the **Debug** tab when you enable native code debugging. This argument puts the Python interpreter into interactive mode after the code finishes, at which point it waits for you to press **Ctrl+Z > Enter** to exit. (Alternately, if you don't mind modifying your Python code, you can add `import os` and `os.system("pause")` statements at the end of your program. This code duplicates the original pause prompt.)

2. Select **File > Save** to save the property changes.
3. Set the build configuration to **Debug** in the Visual Studio toolbar.



4. Because code generally takes longer to run in the debugger, you may want to change the `COUNT` variable in your `.py` file to a value that's about five times smaller (for example, change it from `500000` to `100000`).
5. In your C++ code, set a breakpoint on the first line of the `tanh_impl` method, then start the debugger (**F5** or **Debug > Start Debugging**). The debugger stops when that code is called. If the breakpoint is not hit, check that the configuration is set to **Debug** and that you've saved the project (which does not happen automatically when starting the debugger).



6. At this point you can step through the C++ code, examine variables, and so on. These features are detailed in [Debug C++ and Python together](#).

Alternative approaches

There are a variety of means to create Python extensions as described in the following table. The first two entries for CPython and PyBind11 are what has been discussed in this article already.

APPROACH	VINTAGE	REPRESENTATIVE USER(S)	PRO(S)	CON(S)
C/C++ extension modules for CPython	1991	Standard Library	Extensive documentation and tutorials . Total control.	Compilation, portability, reference management. High C knowledge.
PyBind11 (Recommended for C++)	2015		Lightweight, header-only library for creating Python bindings of existing C++ code. Few dependencies. PyPy compatibility.	Newer, less mature. Heavy use of C++11 features. Short list of supported compilers (Visual Studio is included).
Cython (Recommended for C)	2007	gevent , kivy	Python-like. Highly mature. High performance.	Compilation, new syntax, new toolchain.
Boost.Python	2002		Works with just about every C++ compiler.	Large and complex suite of libraries; contains many workarounds for old compilers.
ctypes	2003	oscrypto	No compilation, wide availability.	Accessing and mutating C structures cumbersome and error prone.
SWIG	1996	crfsuite	Generate bindings for many languages at once.	Excessive overhead if Python is the only target.
cffi	2013	cryptography , pypy	Ease of integration, PyPy compatibility.	Newer, less mature.
cppyy	2017		Similar to cffi using C++.	Newer, may have some issues with VS 2017.

See also

The completed sample from this walkthrough can be found on [python-samples-vs-cpp-extension](#) (GitHub).

Debug Python and C++ together

4/9/2019 • 11 minutes to read • [Edit Online](#)

Most regular Python debuggers support debugging of only Python code. In practice, however, Python is used in conjunction with C or C++ in scenarios requiring high performance or the ability to directly invoke platform APIs. (See [Create a C++ extension for Python](#) for a walkthrough.)

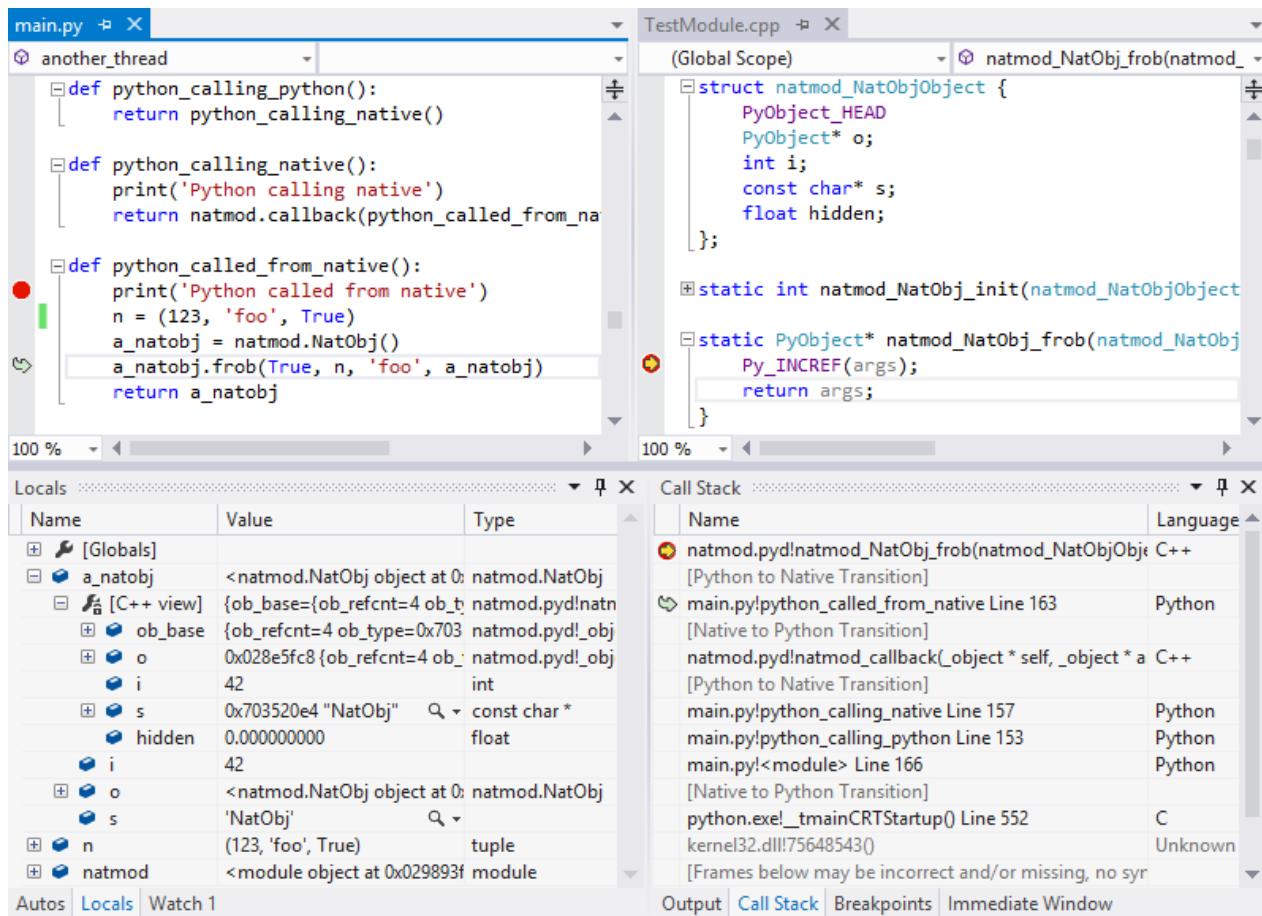
Visual Studio provides integrated, simultaneous mixed-mode debugging for Python and native C/C++, provided that you select the **Python native development tools** option for the **Python Development** workload in the Visual Studio installer.

NOTE

Mixed-mode debugging is not available with Python Tools for Visual Studio 1.x in Visual Studio 2015 and earlier.

Mixed-mode debugging features include the following, as explained in this article:

- Combined call stacks
- Step between Python and native code
- Breakpoints in both types of code
- See Python representations of objects in native frames and vice versa
- Debugging within the context of the Python project or the C++ project

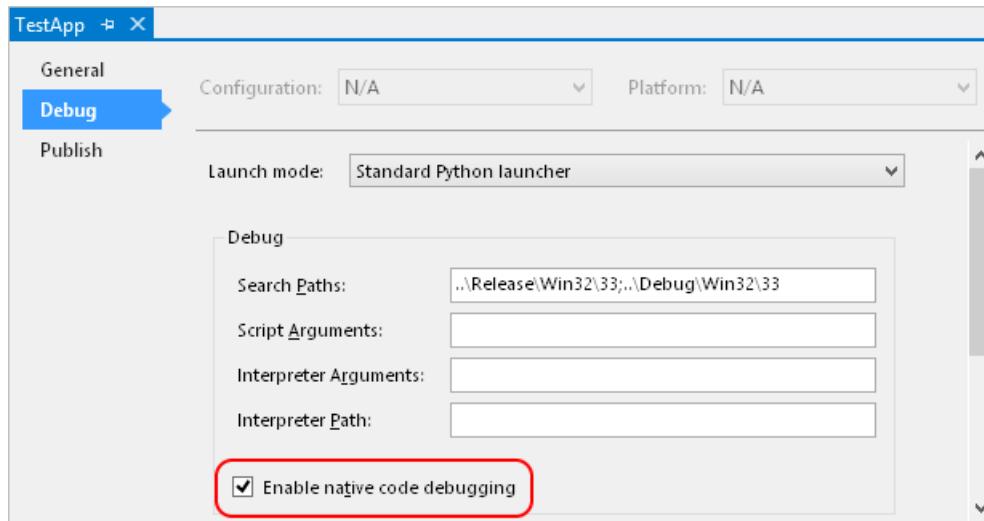




For an introduction to building, testing, and debugging native C modules with Visual Studio, see [Deep Dive: Create Native Modules](#) (youtube.com, 9m 09s). The video is applicable to both Visual Studio 2015 and 2017.

Enable mixed-mode debugging in a Python project

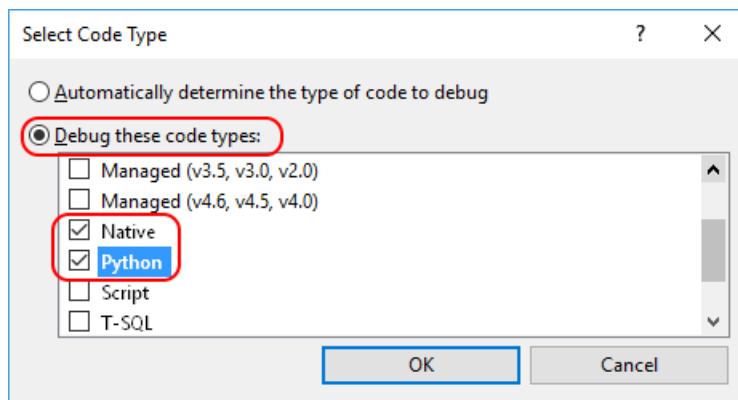
1. Right-click the Python project in **Solution Explorer**, select **Properties**, select the **Debug** tab, and then select **Enable native code debugging**. This option enables mixed-mode for all debugging sessions.



TIP

When you enable native code debugging, the Python output window may disappear immediately when the program has completed without giving you the usual **Press any key to continue** pause. To force a pause, add the `-i` option to the **Run > Interpreter Arguments** field on the **Debug** tab when you enable native code debugging. This argument puts the Python interpreter into interactive mode after the code finishes, at which point it waits for you to press **Ctrl+Z > Enter** to exit.

2. When attaching the mixed-mode debugger to an existing process (**Debug > Attach to Process**), use the **Select** button to open the **Select Code Type** dialog. Then set the **Debug these code types** option and select both **Native** and **Python** in the list:



The code type settings are persistent, so if you want to disable mixed-mode debugging when attaching to a different process later, clear the **Python** code type.

It's possible to select other code types in addition to, or instead of, **Native**. For example, if a managed application hosts CPython, which in turn uses native extension modules, and you want to debug all three,

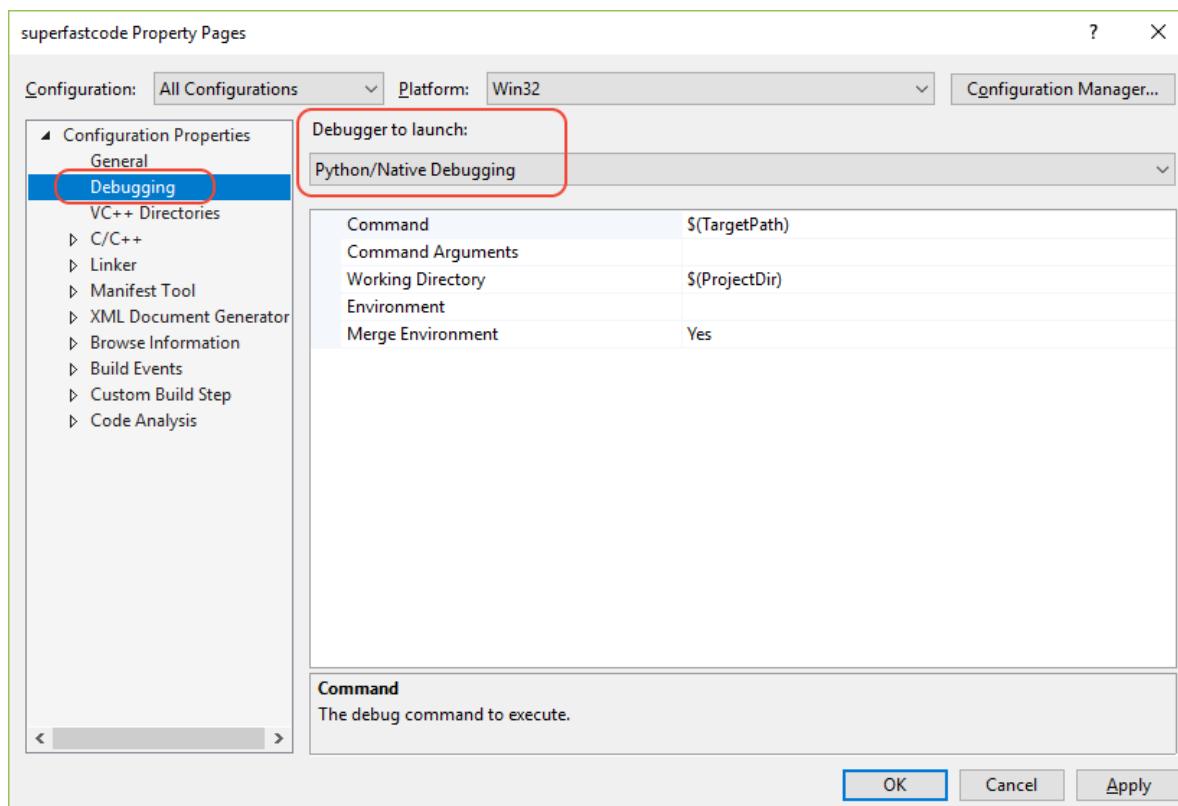
you can check **Python**, **Native**, and **Managed** together for a unified debugging experience including combined call stacks and stepping between all three runtimes.

3. When you start debugging in mixed mode for the first time, you may see a **Python Symbols Required** dialog (see [Symbols for mixed-mode debugging](#)). You need to install symbols only once for any given Python environment. Symbols are automatically included if you install Python support through the Visual Studio installer (Visual Studio 2017 and later).
4. To make the source code for standard Python itself available when debugging, visit <https://www.python.org/downloads/source/>, download the archive appropriate for your version, and extract it to a folder. You then point Visual Studio to specific files in that folder at whatever point it prompts you.

Enable mixed-mode debugging in a C/C++ project

Visual Studio (2017 version 15.5 and later) supports mixed-mode debugging from a C/C++ project (for example, when [embedding Python in another application as described on python.org](#)). To enable mixed-mode debugging, configure the C/C++ project to launch **Python/Native Debugging**:

1. Right-click the C/C++ project in **Solution Explorer** and select **Properties**.
2. Select the **Debugging** tab, select **Python/Native Debugging** from the **Debugger to launch**, and select **OK**.

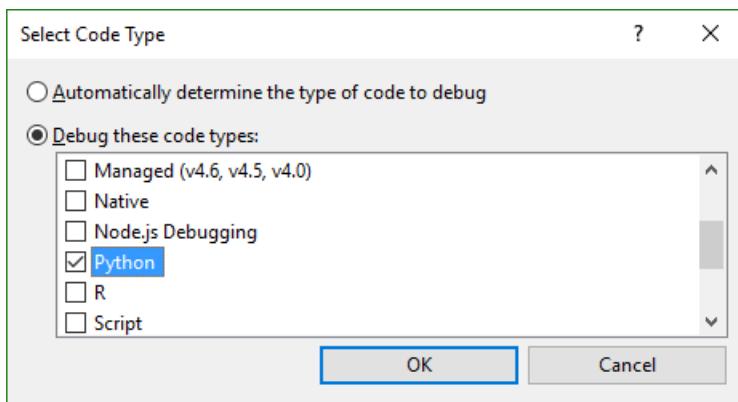


Using this method, be aware that you can't debug the `py.exe` launcher itself, because it spawns a child `python.exe` process that the debugger won't be attached to. If you want to launch `python.exe` directly with arguments, change the **Command** option in the **Python/Native Debugging** properties (shown in the previous image) to specify the full path to `python.exe`, then specify arguments in **Command Arguments**.

Attaching the mixed-mode debugger

For all previous versions of Visual Studio, direct mixed-mode debugging is enabled only when launching a Python project in Visual Studio because C/C++ projects use only the native debugger. You can, however, attach the debugger separately:

1. Start the C++ project without debugging (**Debug > Start without Debugging** or **Ctrl+F5**).
2. Select **Debug > Attach to Process**. In the dialog that appears, select the appropriate process, then use the **Select** button to open the **Select Code Type** dialog in which you can select **Python**:



3. Select **OK** to close that dialog, then **Attach** to start the debugger.
4. You may need to introduce a suitable pause or delay in the C++ app to ensure that it doesn't call the Python code you want to debug before you have a chance to attach the debugger.

Mixed-mode specific features

- Combined call stack
- Step between Python and native code
- PyObject values view in native code
- Native values view in Python code

Combined call stack

The **Call Stack** window shows both native and Python stack frames interleaved, with transitions marked between the two:

Name	Language
natmod.pyd!natmod_NatObj_frob(natmod_NatObjObject * self, _object * args)	C++
[Python to Native Transition]	
main.py!python_called_from_native Line 162	Python
[Native to Python Transition]	
natmod.pyd!natmod_callback(_object * self, _object * arg)	C++
[Python to Native Transition]	
main.py!python_calling_native Line 157	Python
main.py!python_calling_python Line 153	Python
main.py!<module> Line 164	Python
[Native to Python Transition]	
python.exe!_tmainCRTStartup() Line 552	C

Transitions appear as **[External Code]**, without specifying the direction of transition, if the **Tools > Options > Debugging > General > Enable Just My Code** option is set.

Double-clicking any call frame makes it active and opens the appropriate source code, if possible. If source code is not available, the frame is still made active and local variables can be inspected.

Step between Python and native code

When using the **Step Into (F11)** or **Step Out (Shift+F11)** commands, the mixed-mode debugger correctly handles changes between code types. For example, when Python calls a method of a type that is implemented in C, stepping in on a call to that method stops at the beginning of the native function implementing the method. Similarly, when native code calls some Python API function that results in Python code being invoked. For example, stepping into a `PyObject_CallObject` on a function value that was originally defined in Python stops at the beginning of the Python function. Stepping in from Python to native is also supported for native functions

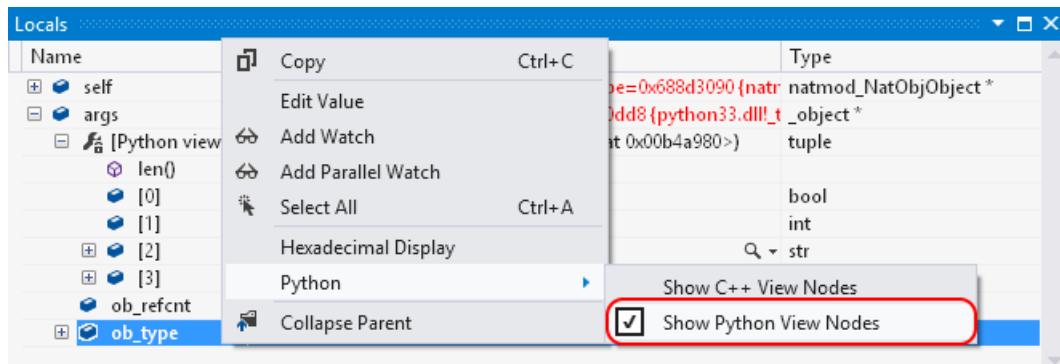
invoked from Python via `ctypes`.

PyObject values view in native code

When a native (C or C++) frame is active, its local variables show up in the debugger **Locals** window. In native Python extension modules, many of these variables are of type `PyObject` (which is a typedef for `_object`), or a few other fundamental Python types (see list below). In mixed-mode debugging, these values present an additional child node labeled **[Python view]**. When expanded, this node shows the variable's Python representation, identical to what you'd see if a local variable referencing the same object was present in a Python frame. The children of this node are editable.

Name	Value	Type
self	0x00b4a980 {ob_base=0x688d3090 {natr natmod_NatObjObject *}	
args	0x00bef4b0 {ob_refcnt=1 ob_type=0x1e240dd8 {python33.dll!_t _object *}	
[Python view]	(True, 123, 'foo', <natmod.NatObj object at 0x00b4a980>)	tuple
len()	4	
[0]	True	bool
[1]	123	int
[2]	'foo'	str
[3]	<natmod.NatObj object at 0x00b4a980>	natmod.NatObj
ob_refcnt	1	int
ob_type	0x1e240dd8 {python33.dll!_typeobject PyTuple_Type} {ob_base _typeobject *}	

To disable this feature, right-click anywhere in the **Locals** window and toggle the **Python > Show Python View Nodes** menu option:



C types that show **[Python view]** nodes (if enabled):

- `PyObject`
- `PyVarObject`
- `PyTypeObject`
- `PyByteArrayObject`
- `PyBytesObject`
- `PyTupleObject`
- `PyListObject`
- `PyDictObject`
- `PySetObject`
- `PyIntObject`
- `PyLongObject`
- `PyFloatObject`
- `PyStringObject`
- `PyUnicodeObject`

[Python view] does not automatically appear for types you author yourself. When authoring extensions for Python 3.x, this lack is usually not an issue because any object ultimately has an `ob_base` field of one of the types

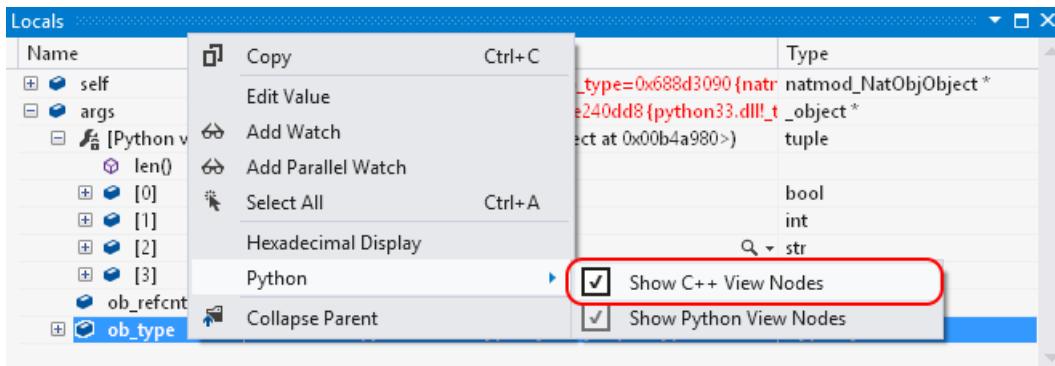
above, which causes **[Python view]** to appear.

For Python 2.x, however, each object type typically declares its header as a collection of inline fields, and there is no association between custom authored types and `PyObject` at the type system level in C/C++ code. To enable **[Python view]** nodes for such custom types, edit the `PythonDkm.natvis` file in the [Python tools install directory](#), and add another element in the XML for your C struct or C++ class.

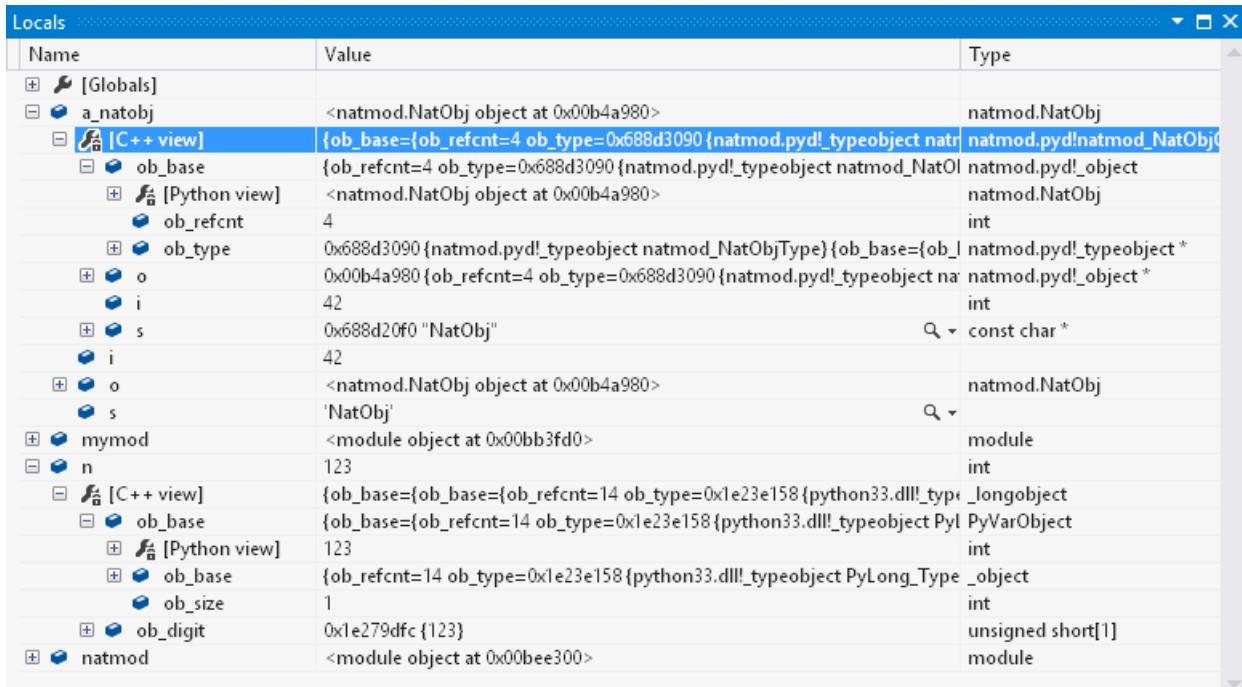
An alternate (and better) option is to follow [PEP 3123](#) and use an explicit `PyObject ob_base;` field rather than `PyObject_HEAD`, though that may not always be possible for backwards-compatibility reasons.

Native values view in Python code

Similar to the previous section, you can enable a **[C++ view]** for native values in the **Locals** window when a Python frame is active. This feature is not enabled by default, so you turn it on by right-clicking in the **Locals** window and toggling the **Python > Show C++ View Nodes** menu option.



The **[C++ view]** node provides a representation of the underlying C/C++ structure for a value, identical to what you'd see in a native frame. For example, it shows an instance of `_longobject` (for which `PyLongObject` is a typedef) for a Python long integer, and it tries to infer types for native classes that you have authored yourself. The children of this node are editable.



If a child field of an object is of type `PyObject`, or one of the other supported types, then it has a **[Python view]** representation node (if those representations are enabled), making it possible to navigate object graphs where links are not directly exposed to Python.

Unlike **[Python view]** nodes, which use Python object metadata to determine the type of the object, there's no similarly reliable mechanism for **[C++ view]**. Generally speaking, given a Python value (that is, a `Pyobject`

reference) it's not possible to reliably determine which C/C++ structure is backing it. The mixed-mode debugger tries to guess that type by looking at various fields of the object's type (such as the `PyTypeObject` referenced by its `ob_type` field) that have function pointer types. If one of those function pointers references a function that can be resolved, and that function has a `self` parameter with type more specific than `PyObject*`, then that type is assumed to be the backing type. For example, if `ob_type->tp_init` of a given object points at the following function:

```
static int FobObject_init(FobObject* self, PyObject* args, PyObject* kwds) {
    return 0;
}
```

then the debugger can correctly deduce that the C type of the object is `FobObject`. If it's unable to determine a more precise type from `tp_init`, it moves on to other fields. If it's unable to deduce the type from any of those fields, the **[C++ view]** node presents the object as a `PyObject` instance.

To always get a useful representation for custom authored types, it's best to register at least one special function when registering the type, and use a strongly-typed `self` parameter. Most types fulfill that requirement naturally; if that's not the case, then `tp_init` is usually the most convenient entry to use for this purpose. A dummy implementation of `tp_init` for a type that is present solely to enable debugger type inference can just return zero immediately, as in the code sample above.

Differences from standard Python debugging

The mixed-mode debugger is distinct from the [standard Python debugger](#) in that it introduces some additional features but lacks some Python-related capabilities:

- Unsupported features: conditional breakpoints, **Debug Interactive** window, and cross-platform remote debugging.
- **Immediate** window: is available but with a limited subset of its functionality, including all the limitations listed here.
- Supported Python versions: CPython 2.7 and 3.3+ only.
- Visual Studio Shell: When using Python with Visual Studio Shell (for example, if you installed it using the integrated installer), Visual Studio is unable to open C++ projects, and the editing experience for C++ files is that of a basic text editor only. However, C/C++ debugging and mixed-mode debugging are fully supported in Shell with source code, stepping into native code, and C++ expression evaluation in debugger windows.
- Viewing and expanding objects: When viewing Python objects in the **Locals** and **Watch** debugger tool windows, the mixed-mode debugger shows only the structure of the objects. It does not automatically evaluate properties, or show computed attributes. For collections, it shows only elements for built-in collection types (`tuple`, `list`, `dict`, `set`). Custom collection types are not visualized as collections, unless they are inherited from some built-in collection type.
- Expression evaluation: see below.

Expression evaluation

The standard Python debugger allows evaluation of arbitrary Python expressions in **Watch** and **Immediate** windows when the debugged process is paused at any point in the code, so long as it is not blocked in an I/O operation or other similar system call. In mixed-mode debugging, arbitrary expressions can be evaluated only when stopped in Python code, after a breakpoint, or when stepping into the code. Expressions can be evaluated only on the thread on which the breakpoint or the stepping operation occurred.

When stopped in native code, or in Python code where the conditions above do not apply (for example, after a step-out operation, or on a different thread), expression evaluation is limited to accessing local and global variables in scope of the currently selected frame, accessing their fields, and indexing built-in collection types with literals. For example, the following expression can be evaluated in any context (provided that all identifiers refer to

existing variables and fields of appropriate types):

```
foo.bar[0].baz['key']
```

The mixed-mode debugger also resolves such expressions differently. All member-access operations look up only fields that are directly part of the object (such as an entry in its `__dict__` or `__slots__`, or a field of a native struct that is exposed to Python via `tp_members`), and ignore any `__getattr__`, `__getattribute__` or descriptor logic. Similarly, all indexing operations ignore `__getitem__`, and access the inner data structures of collections directly.

For the sake of consistency, this name resolution scheme is used for all expressions that match the constraints for limited expression evaluation, regardless of whether arbitrary expressions are allowed at the current stop point. To force proper Python semantics when a full-featured evaluator is available, enclose the expression in parentheses:

```
(foo.bar[0].baz['key'])
```

Install debugging symbols for Python interpreters

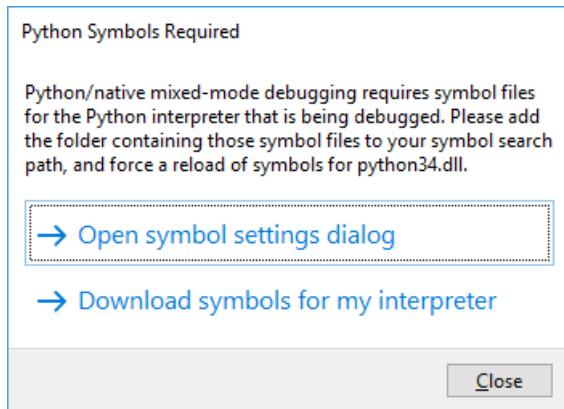
4/9/2019 • 4 minutes to read • [Edit Online](#)

To provide a full debugging experience, the [mixed-mode Python debugger](#) in Visual Studio needs debug symbols for the Python interpreter being used to parse numerous internal data structures. For `python27.dll`, for example, the corresponding symbol file is `python27.pdb`; for `python36.dll`, the symbol file is `python36.pdb`. Each version of the interpreter also supplies symbol files for a variety of modules.

With Visual Studio 2017 and later, the Python 3 and Anaconda 3 interpreters automatically install their respective symbols and Visual Studio finds those symbols automatically. For Visual Studio 2015 and earlier, or when using other interpreters, you need to download symbols separately and then point Visual Studio to them through the **Tools > Options** dialog in the **Debugging > Symbols** tab. These steps are detailed in the following sections.

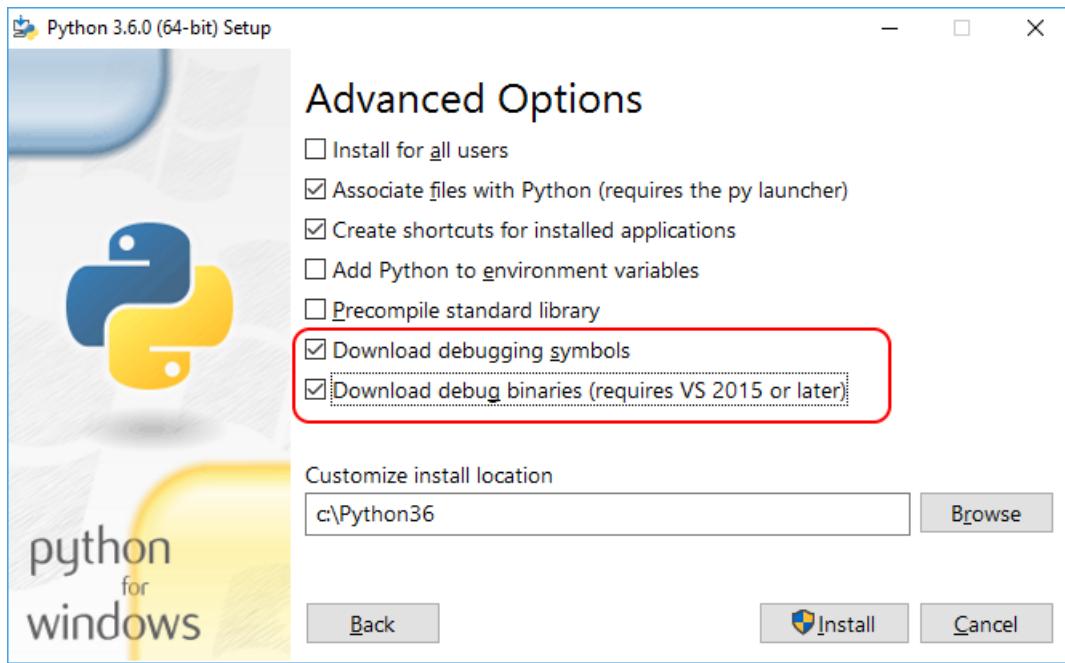
Visual Studio may prompt you when it needs symbols, typically when starting a mixed-mode debugging session. In this case, it displays a dialog with two choices:

- **Open symbol settings dialog** opens the **Options** dialog to the **Debugging > Symbols** tab.
- **Download symbols for my interpreter** opens this present documentation page, in which case, select **Tools > Options** and navigate to the **Debugging > Symbols** tab to continue.



Download symbols

- Python 3.5 and later: acquire debug symbols through the Python installer. Select **Custom installation**, select **Next** to get to **Advanced Options**, then select the boxes for **Download debugging symbols** and **Download debug binaries**:



The symbol files (.pdb) are then found in the root installation folder (symbol files for individual modules are in the *DLLs* folder as well). Because of this, Visual Studio finds them automatically, and no further steps are needed.

- Python 3.4.x and earlier: symbols are available as downloadable .zip files from the [official distributions](#) or [Enthought Canopy](#). After downloading, extract files to a local folder to continue, such as a *Symbols* folder within the Python folder.

IMPORTANT

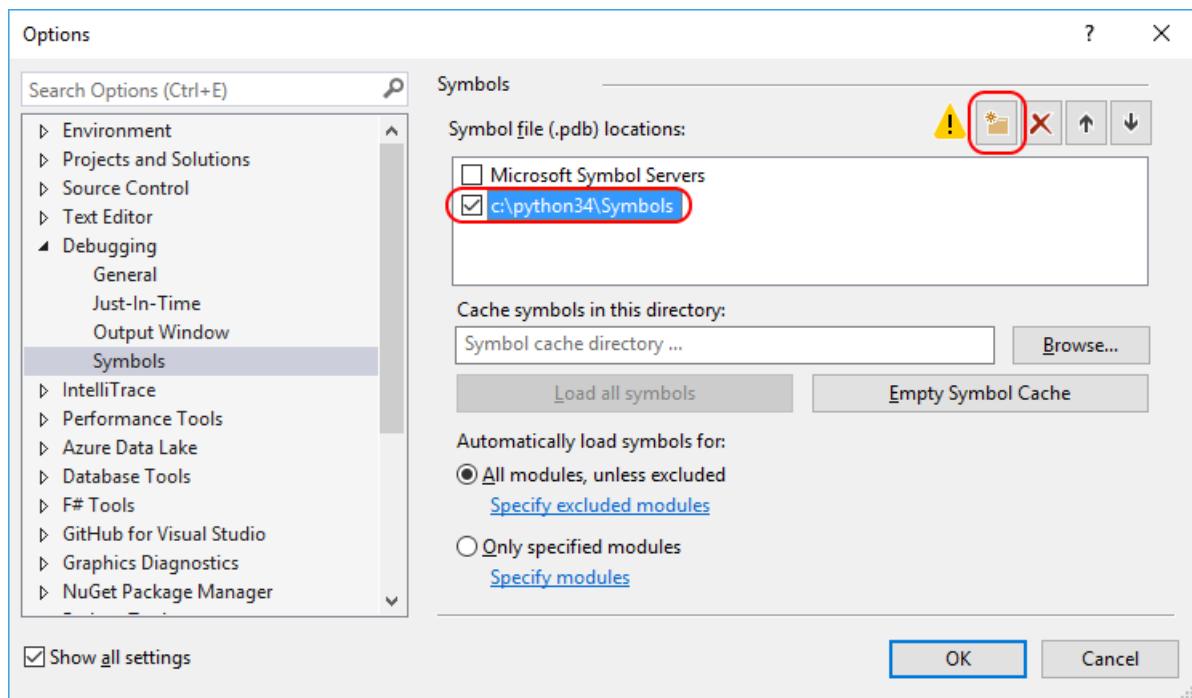
Symbols differ between minor builds of Python, and between 32-bit and 64-bit builds, so you want to exactly match the versions. To check the interpreter being used, expand the **Python Environments** node under your project in **Solution Explorer** and note the environment name. Then switch to the **Python Environments** window and note the install location. Then open a command window in that location and start *python.exe*, which displays the exact version and whether it's 32-bit or 64-bit.

- For any other third-party Python distribution such as ActiveState Python: contact the authors of that distribution and request them to provide you with symbols. WinPython, for its part, incorporates the standard Python interpreter without changes, so use symbols from the official distribution for the corresponding version number.

Point Visual Studio to the symbols

If you downloaded symbols separately, follow the steps below to make Visual Studio aware of them. If you installed symbols through the Python 3.5 or later installer, Visual Studio finds them automatically.

1. Select the **Tools > Options** menu and navigate to **Debugging > Symbols**.
2. Select the **Add** button on the toolbar (outlined below), enter the folder where you expanded the downloaded symbols (which is where *python.pdb* is located, such as *c:\python34\Symbols*, shown below), and select **OK**.



3. During a debugging session, Visual Studio might also prompt you for the location of a source file for the Python interpreter. If you've downloaded source files (from python.org/downloads/, for example), then you of course can point to them as well.

NOTE

The symbol caching features shown in the dialog are used to create a local cache of symbols obtained from an online source. These features aren't needed with the Python interpreter symbols as symbols are already present locally. In any case, refer to [Specify symbols and source files in the Visual Studio debugger](#) for details.

Official distributions

PYTHON VERSION	DOWNLOADS
3.5 and later	Install symbols through the Python installer.
3.4.4	32-bit - 64-bit
3.4.3	32-bit - 64-bit
3.4.2	32-bit - 64-bit
3.4.1	32-bit - 64-bit
3.4.0	32-bit - 64-bit
3.3.5	32-bit - 64-bit
3.3.4	32-bit - 64-bit
3.3.3	32-bit - 64-bit
3.3.2	32-bit - 64-bit

PYTHON VERSION	DOWNLOADS
3.3.1	32-bit - 64-bit
3.3.0	32-bit - 64-bit
2.7.15	32-bit - 64-bit
2.7.14	32-bit - 64-bit
2.7.13	32-bit - 64-bit
2.7.12	32-bit - 64-bit
2.7.11	32-bit - 64-bit
2.7.10	32-bit - 64-bit
2.7.9	32-bit - 64-bit
2.7.8	32-bit - 64-bit
2.7.7	32-bit - 64-bit
2.7.6	32-bit - 64-bit
2.7.5	32-bit - 64-bit
2.7.4	32-bit - 64-bit
2.7.3	32-bit - 64-bit
2.7.2	32-bit - 64-bit
2.7.1	32-bit - 64-bit

Enthought Canopy

Enthought Canopy provides symbols for its binaries starting from version 1.2. They are automatically installed alongside with the distribution, but you still need to manually add the folder containing them to symbol path as described earlier. For a typical per-user installation of Canopy, the symbols are located in `%UserProfile%\AppData\Local\Enthought\Canopy\User\Scripts` for the 64-bit version and `%UserProfile%\AppData\Local\Enthought\Canopy32\User\Scripts` for the 32-bit version.

Enthought Canopy 1.1 and earlier, as well as Enthought Python Distribution (EPD), do not provide interpreter symbols, and are therefore not compatible with mixed-mode debugging.

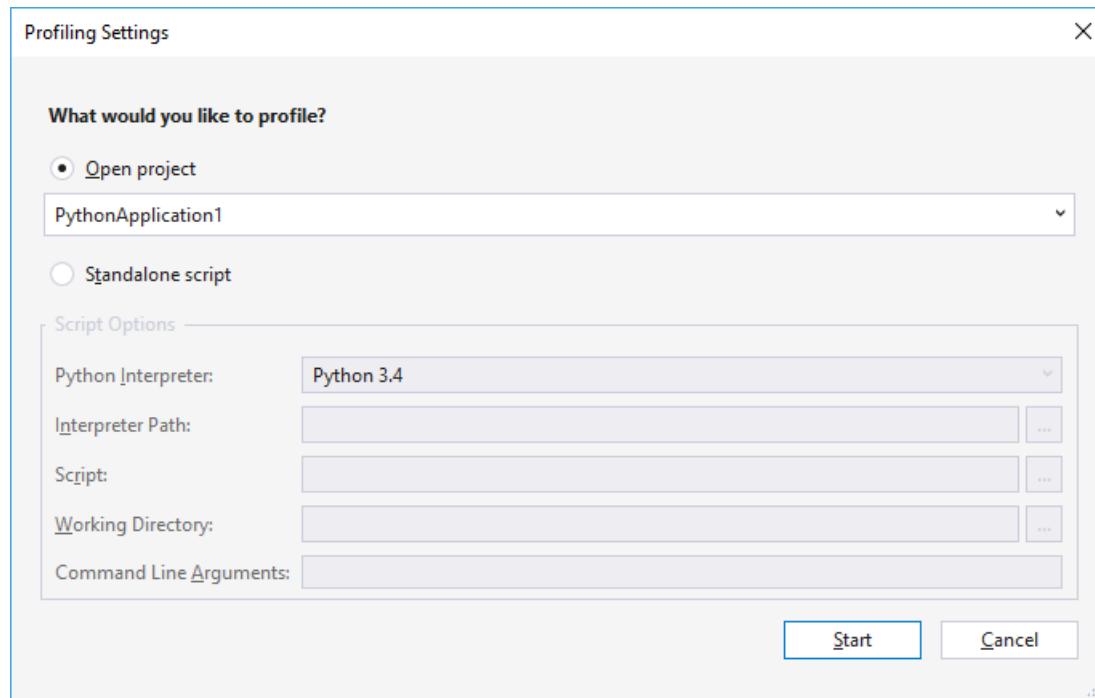
Profile Python code

4/9/2019 • 2 minutes to read • [Edit Online](#)

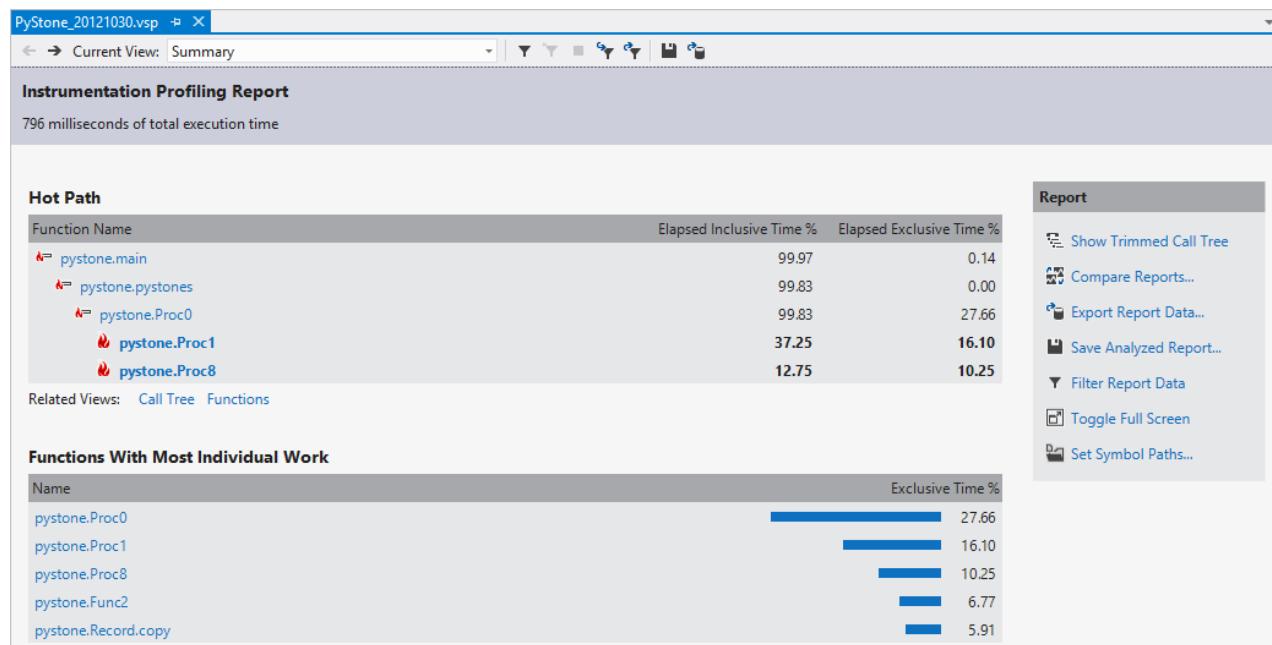
You can profile a Python application when using CPython-based interpreters. (See [Features matrix - profiling](#) for the availability of this feature for different versions of Visual Studio.)

Profiling for CPython-based interpreters

Profiling is started through the **Analyze > Launch Python Profiling** menu command, which opens a configuration dialog:



When you select **OK**, the profiler runs and opens a performance report through which you can explore how time is spent in the application:



NOTE

At present, Visual Studio supports only this level of full-application profiling, but we certainly want to hear your feedback on future capabilities. Use the **Product feedback** button at the bottom of this page.

Profiling for IronPython

Because IronPython is not a CPython-based interpreter, the profiling feature above does not work.

Instead, use the Visual Studio .NET profiler by launching *ipy.exe* directly as the target application, using the appropriate arguments to launch your startup script. Include `-x:Debug` on the command line to ensure that all of your Python code can be debugged and profiled. This argument generates a performance report including time spent both in the IronPython runtime and your code. Your code is identified using mangled names.

Alternately, IronPython has some of its own built-in profiling but there's currently no good visualizer for it. See [An IronPython Profiler](#) (MSDN blogs) for what's available.

Set up unit testing for Python code

4/9/2019 • 2 minutes to read • [Edit Online](#)

Unit tests are pieces of code that test other code units in an application, typically isolated functions, classes, and so on. When an application passes all its unit tests, you can at least trust that its low-level functionality is correct.

Python uses unit tests extensively to validate scenarios while designing a program. Python support in Visual Studio includes discovering, executing, and debugging unit tests within the context of your development process, without needing to run tests separately.

This article provides a brief outline of unit testing capabilities in Visual Studio with Python. For more on unit testing in general, see [Unit test your code](#).

Discover and view tests

By convention, Visual Studio identifies tests as methods whose names start with `test`. To see this behavior, do the following:

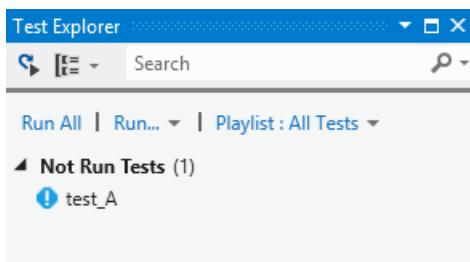
1. Open a [Python project](#) loaded in Visual Studio, right-click your project, select **Add > New Item**, then select **Python Unit Test** followed by **Add**.
2. This action creates a `test1.py` file with code that imports the standard `unittest` module, derives a test class from `unittest.TestCase`, and invokes `unittest.main()` if you run the script directly:

```
import unittest

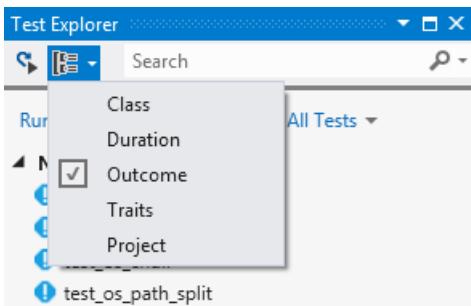
class Test_test1(unittest.TestCase):
    def test_A(self):
        self.fail("Not implemented")

if __name__ == '__main__':
    unittest.main()
```

3. Save the file if necessary, then open **Test Explorer** with the **Test > Windows > Test Explorer** menu command.
4. **Test Explorer** searches your project for tests and displays them as shown below. Double-clicking a test opens its source file.



5. As you add more tests to your project, you can organize the view in **Test Explorer** using the **Group by** menu on the toolbar:



6. You can also enter text in the **Search** field to filter tests by name.

For more information on the `unittest` module and writing tests, see the [Python 2.7 documentation](#) or the [Python 3.4 documentation](#) ([python.org](#)).

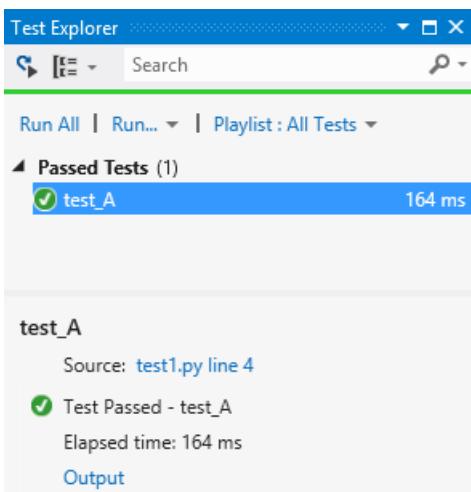
Run tests

In **Test Explorer** you can run tests in a variety of ways:

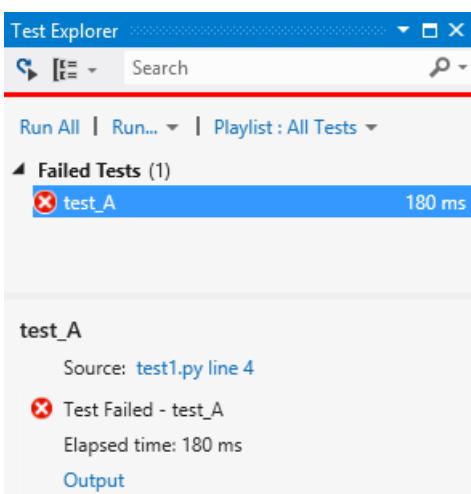
- **Run All** clearly runs all shown tests (subject to filters).
- The **Run** menu gives you commands to run failed, passed, or not run tests as a group.
- You can select one or more tests, right-click, and select **Run Selected Tests**.

Tests run in the background and **Test Explorer** updates each test's status as it completes:

- Passing tests show a green tick and the time taken to run the test:



- Failed tests show a red cross with an **Output** link that shows console output and `unittest` output from the test run:



Test Output-test_A-3-1 ✘ X

Test Name: test_A

Test Outcome: Failed

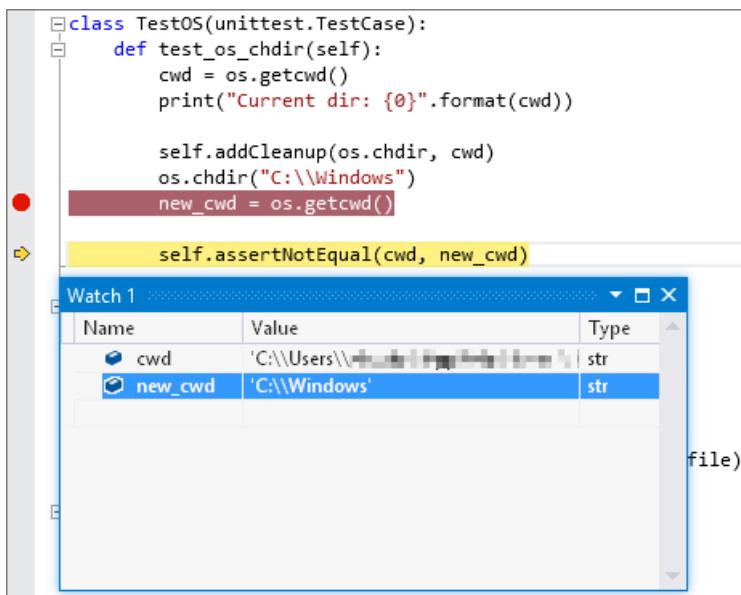
Standard Error

```
=====
FAIL: test_A (test1.Test_test1)
-----
Traceback (most recent call last):
  File "C:\Users\stevdo\AppData\Local\Temp\PythonApplication2\PythonApplication2\test1.py", line 6, in test_A
    self.fail("Not implemented")
AssertionError: Not implemented
-----
Ran 1 test in 0.037s
FAILED (failures=1)
```

Debug tests

Because unit tests are pieces of code, they are subject to bugs just like any other code and occasionally need to be run in a debugger. In the debugger you can set breakpoints, examine variables, and step through code. Visual Studio also provides diagnostic tools for unit tests.

To start debugging, set an initial breakpoint in your code, then right-click the test (or a selection) in **Test Explorer** and select **Debug Selected Tests**. Visual Studio starts the Python debugger as it would for application code.



You can also use the **Analyze Code Coverage for Selected Tests** and **Profile Test** commands.

Known issues

- When starting debugging, Visual Studio appears to start and stop debugging, and then start again. This behavior is expected.
- When debugging multiple tests, each one is run independently, which interrupts the debugging session.
- Visual Studio intermittently fails to start a test when debugging. Normally, attempting to debug the test again succeeds.
- When debugging, it is possible to step out of a test into the `unittest` implementation. Normally, the next step runs to the end of the program and stops debugging.

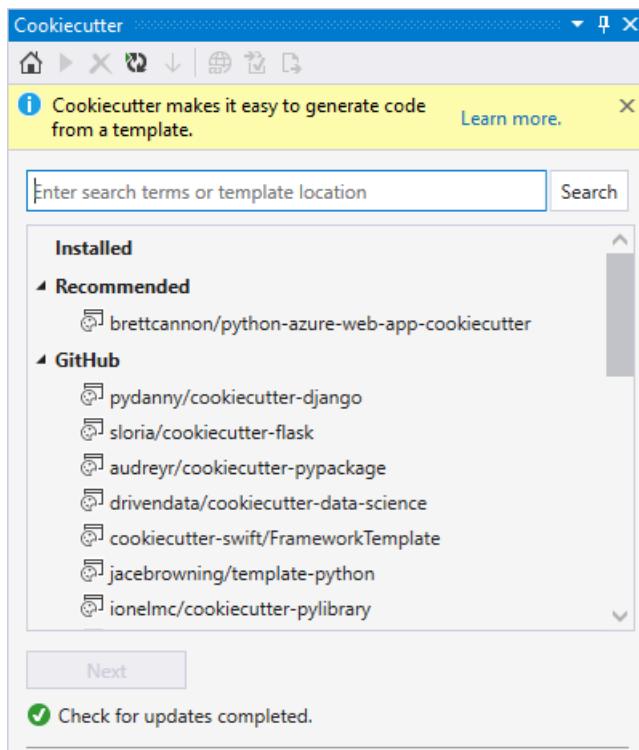
Use the Cookiecutter extension

4/9/2019 • 8 minutes to read • [Edit Online](#)

Cookiecutter provides a graphical user interface to discover templates, input template options, and create projects and files. It's included with Visual Studio 2017 and later and can be installed separately in earlier versions of Visual Studio.

Cookiecutter requires Python 3.3 or later (32-bit or 64-bit) or Anaconda 3 4.2 or later (32-bit or 64-bit). If a suitable Python interpreter isn't available, Visual Studio displays a warning. If you install a Python interpreter while Visual Studio is running, click the **Home** button on the Cookiecutter toolbar to detect the newly installed interpreter. (See [Python environments](#) for more about environments in general.)

Once installed, select **View > Cookiecutter Explorer** to open its window:



Cookiecutter workflow

Working with Cookiecutter is a process of browsing and selecting a template, cloning it to your local computer, setting options, then creating code from that template, as described in the sections that follow.

Browse templates

The Cookiecutter home page displays a list of templates to choose from, organized into the following groups:

GROUP	DESCRIPTION
Installed	Templates that have been installed to your local computer. When an online template is used, its repository is automatically cloned to a subfolder of <code>~/.cookiecutters</code> . You can delete a selected installed template by pressing Delete .

GROUP	DESCRIPTION
Recommended	Templates loaded from the recommended feed. The default feed is curated by Microsoft. See Cookiecutter options below for details on customizing the feed.
GitHub	GitHub search results for the cookiecutter keyword. Results from GitHub come back paginated, if more results are available, Load More appears at the end of the list.
Custom	When a custom location is entered in the search box, it appears in this group. You can either type in a full path to the GitHub repository, or the full path to a folder on your local disk.

Cloning

When you select a template followed by **Next**, Cookiecutter makes a local copy to work from.

If you select a template from the **Recommended** or **GitHub** groups, or enter a custom URL into the search box and select that template, it's cloned and installed on your local computer. If that template was installed in a previous session of Visual Studio, it's automatically deleted and the latest version is cloned.

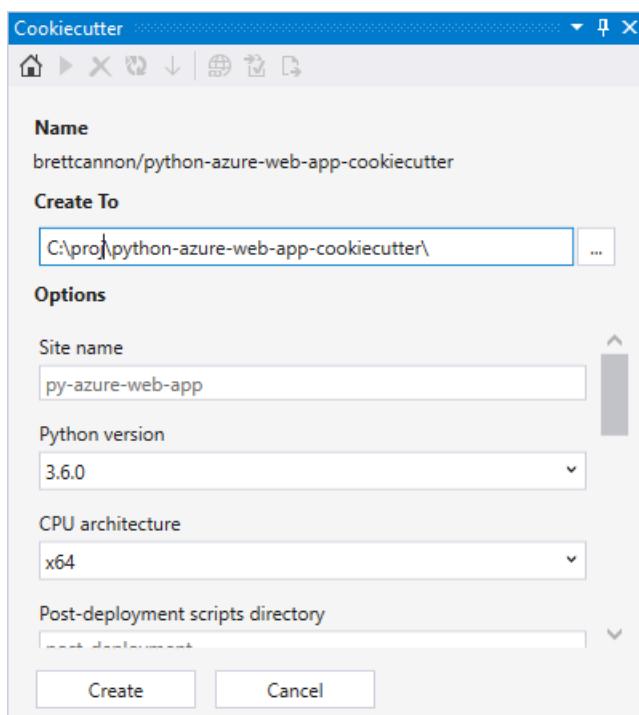
If you select a template from the **Installed** group, or enter a custom folder path into the search box and select that template, Visual Studio loads that template without cloning.

IMPORTANT

Cookiecutter templates are cloned under a single folder `~/.cookiecutters`. Each subfolder is named after the git repository name, which does not include the GitHub user name. Conflicts can arise if you clone different templates with the same name that come from different authors. In this case, Cookiecutter prevents you from overwriting the existing template with a different template of the same name. To install the other template, you must first delete the existing one.

Set template options

After the template is installed locally, Cookiecutter displays an options page where you can specify where you want Cookiecutter to generate files along with other options:



Each Cookiecutter template defines its own set of options, and specifies a default value for each one (displayed as the suggested text in each entry field). A default value may be a code snippet, often when it's a dynamic value that uses other options.

It's possible to customize default values for specific options with a user configuration file. When the Cookiecutter extension detects a user configuration file, it overwrites the template's default values with the user config's default values. This behavior is discussed in the [User Config](#) section of the Cookiecutter documentation.

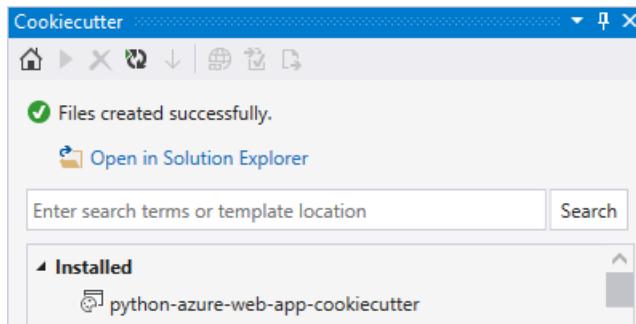
If the template specifies specific Visual Studio tasks to run after code generation, an additional **Run additional tasks on completion** option appears that allows you to opt out of those tasks. The most common use of tasks is to open a web browser, open files in the editor, install dependencies, and so on.

Create

Once you've set your options, select **Create** to generate code (a warning appears if the output folder isn't empty). If you're familiar with the template's output and don't mind overwriting files, you can dismiss the warning.

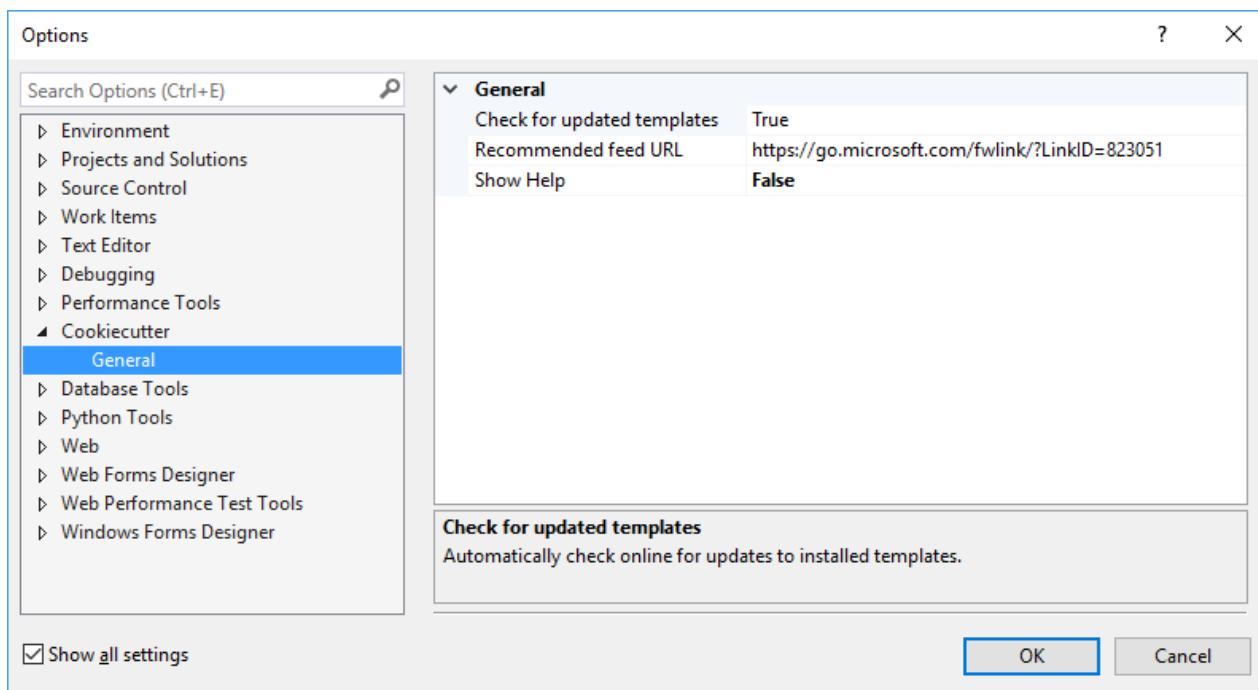
Otherwise, select **Cancel**, specify an empty folder, and then manually copy the created files to your non-empty output folder.

After the files are created successfully, Cookiecutter provides an option to open the files in **Solution Explorer**:



Cookiecutter options

Cookiecutter options are available through **Tools > Options > Cookiecutter**:



OPTION	DESCRIPTION
Recommended feed URL	The location of the recommended templates feed. It can be a URL or a path to a local file. Leave the URL empty to use the default Microsoft curated feed. The feed provides a simple list of template locations, separated by newlines. To request changes to the curated feed, make a pull request against the source on GitHub .
Show Help	Controls the visibility of the help information bar at the top of the Cookiecutter window.

Optimize Cookiecutter templates for Visual Studio

For the basics of authoring a Cookiecutter template, see the [Cookiecutter documentation](#). The Cookiecutter extension for Visual Studio supports templates created for Cookiecutter v1.4.

The default rendering of template variables depends on the type of data (string or list):

- String: Label for variable name, text box for entering value, and a watermark showing the default value. Tooltip on the text box shows the default value.
- List: Label for variable name, combo box for selecting a value. Tooltip on the combo box shows the default value.

It's possible to improve on this rendering by specifying additional metadata in your `cookiecutter.json` file that's specific to Visual Studio (and ignored by the Cookiecutter CLI). All properties are optional:

PROPERTY	DESCRIPTION
Label	Specifies what appears above the editor for the variable, instead of the name of the variable.
Description	Specifies the tooltip that appears on the edit control, instead of the default value for that variable.
URL	Changes the label into a hyperlink, with a tooltip that shows the URL. Clicking on the hyperlink opens the user's default browser to that URL.
Selector	Allows customization of the editor for a variable. The following selectors are currently supported: <ul style="list-style-type: none"> • <code>string</code> : Standard text box, default for strings. • <code>list</code> : Standard combo box, default for lists. • <code>yesno</code> : Combo box to choose between <code>y</code> and <code>n</code>, for strings. • <code>odbcConnection</code> : Text box with a ... button that brings up a database connection dialog.

Example:

```
{
    "site_name": "web-app",
    "python_version": ["3.5.2", "2.7.12"],
    "use_azure": "y",

    "_visual_studio": {
        "site_name": {
            "label": "Site name",
            "description": "E.g. <site-name>.azurewebsites.net (can only contain alphanumeric characters and '-' )"
        },
        "python_version": {
            "label": "Python version",
            "description": "The version of Python to run the site on"
        },
        "use_azure" : {
            "label": "Use Azure",
            "description": "Include Azure deployment files",
            "selector": "yesno",
            "url": "https://azure.microsoft.com"
        }
    }
}
```

Run Visual Studio tasks

Cookiecutter has a feature called *Post-Generate Hooks* that allows for running arbitrary Python code after the files are generated. Although flexible, it doesn't allow easy access to Visual Studio.

For example, you may want to open a file in the Visual Studio editor, or in its web browser, or trigger the Visual Studio UI that prompts the user to create a virtual environment and install package requirements.

To allow these scenarios, Visual Studio looks for extended metadata in *cookiecutter.json* that describes the commands to run after the user opens the generated files in **Solution Explorer** or after the files are added to an existing project. (Again, the user can opt out of running the tasks by clearing **Run additional tasks on completion** in the template options.)

Example:

```
"_visual_studio_post_cmds": [
    {
        "name": "File.OpenFile",
        "args": "{{cookiecutter._output_folder_path}}\\readme.txt"
    },
    {
        "name": "Cookiecutter.ExternalWebBrowser",
        "args": "https://docs.microsoft.com"
    },
    {
        "name": "Python.InstallProjectRequirements",
        "args": "{{cookiecutter._output_folder_path}}\\dev-requirements.txt"
    }
]
```

Commands are specified by name, and should use the non-localized (English) name to work on localized installs of Visual Studio. You can test and discover command names in the Visual Studio **Command** window.

If you want to pass a single argument, specify it as a string like in the previous example.

If you don't need to pass an argument, leave it an empty string or omit it from the JSON:

```
_visual_studio_post_cmds": [
  {
    "name": "View.WebBrowser"
  }
]
```

Use an array for multiple arguments. For switches, split the switch and its value into separate arguments and use proper quoting. For example:

```
_visual_studio_post_cmds": [
  {
    "name": "File.OpenFile",
    "args": [
      "{{cookiecutter._output_folder_path}}\\read me.txt",
      "/e:",
      "Source Code (text) Editor"
    ]
  }
]
```

Arguments can refer to other Cookiecutter variables. In the examples above, the internal `_output_folder_path` variable is used to form an absolute path to generated files.

Note that the `Python.InstallProjectRequirements` command works only when adding files to an existing project. This limitation exists because the command is processed by the Python project in **Solution Explorer**, and there's no project to receive the message while in **Solution Explorer - Folder View**. We hope to remove the limitation in a future release (and provide better **Folder View** support in general).

Troubleshooting

Error loading template

Some templates may be using invalid data types, such as boolean, in their `cookiecutter.json`. Report such instances to the template author by selecting the **Issues** link in the template information pane.

Hook script failed

Some templates may use post-generation scripts that are not compatible with the Cookiecutter UI. For example, scripts that query the user for input fails due to not having a terminal console.

Hook script not supported on Windows

If the post script is `.sh`, then it may not be associated with an application on your Windows computer. You may see a Windows dialog asking you to find a compatible application in the Windows store.

Templates with known issues

Clone failures:

- **wildfish/cookiecutter-django-crud** (invalid character `|` in subfolder name)
- **cookiecutter-pyvanguard** (invalid character `|` in subfolder name)

Load failures:

- **chrisdev/wagtail-cookiecutter-foundation** (uses a boolean type in `cookiecutter.json`)
- **quintoandar/cookiecutter-android** (no template folder)

Run failures:

- **iknite/cookiecutter-ansible-role** (post hook script requires console input)

- **benregn/cookiecutter-django-ansible** (Jinja error)

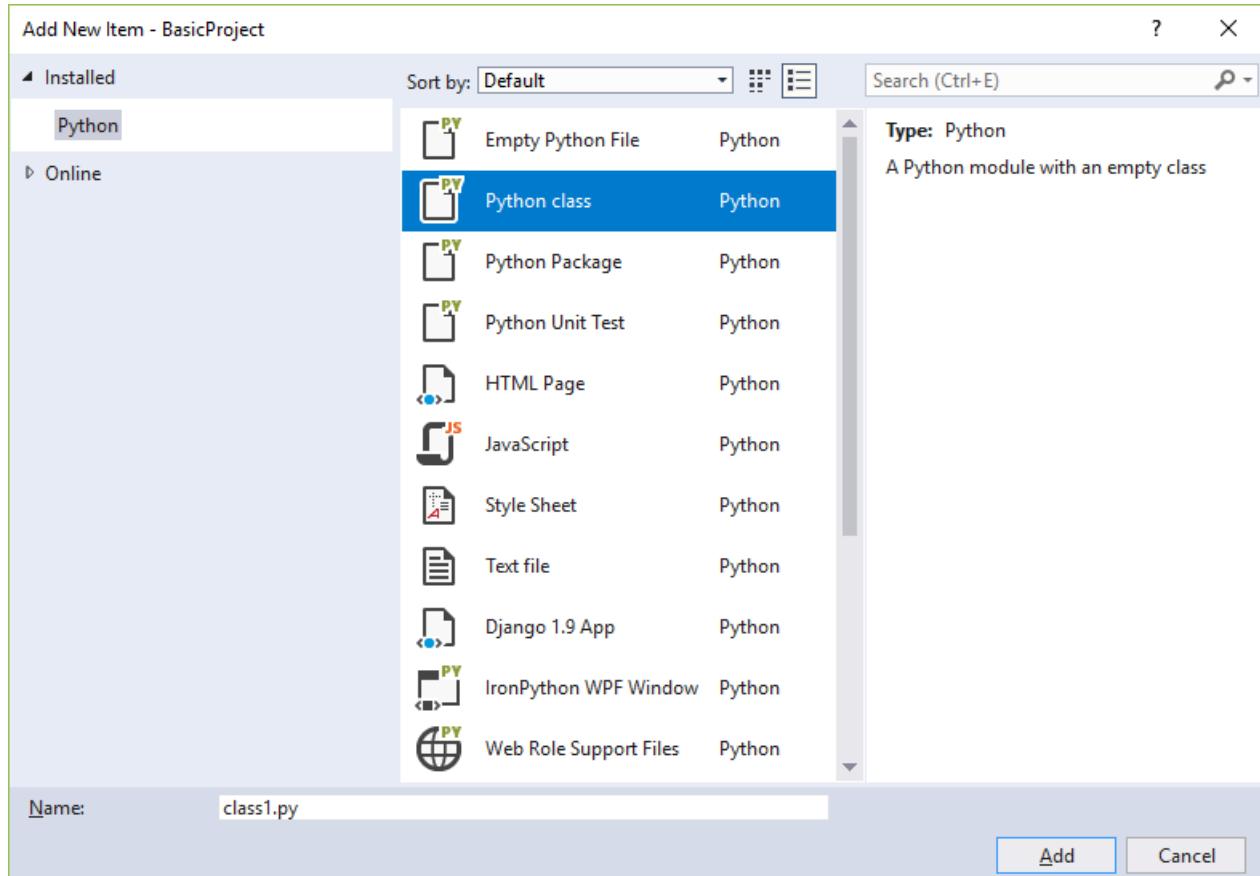
Uses bash (not fatal):

- **openstack-dev/cookiecutter**

Python item templates

4/9/2019 • 2 minutes to read • [Edit Online](#)

The item templates are available in Python projects through the **Project > Add New Item** menu command, or the **Add > New Item** command on the context menu in **Solution Explorer**.



Using the name you provide for the item, a template typically creates one or more files and folders within the currently selected folder in the project (right-clicking a folder to bring up the context menu automatically selects that folder). Adding an item includes it in the Visual Studio project, and the item appears in **Solution Explorer**.

The following table briefly explains the effect of each item template within a Python project:

TEMPLATE	WHAT THE TEMPLATE CREATES
Empty Python File	An empty file with the <code>.py</code> extension.
Python class	A <code>.py</code> file containing a single empty Python class definition.
Python Package	A folder that contains an <code>__init__.py</code> file.
Python Unit Test	A <code>.py</code> file with a single unit test based on the <code>unittest</code> framework, along with a call to <code>unittest.main()</code> to run the tests in the file.
HTML Page	An <code>.html</code> file with a simple page structure consisting of a <code><head></code> and <code><body></code> element.

TEMPLATE	WHAT THE TEMPLATE CREATES
JavaScript	An empty <code>.js</code> file.
Style Sheet	A <code>.css</code> file containing an empty style for <code>body</code> .
Text file	An empty <code>.txt</code> file.
Django 1.9 App Django 1.4 App	A folder with the name of the app, which contains the core files for a Django app as explained in Learn Django in Visual Studio, Step 2-2 for Django 1.9. For Django 1.4, the <i>migrations</i> folder, the <i>admin.py</i> file, and the <i>apps.py</i> file are not included.
IronPython WPF Window	A WPF Window consisting of two side-by-side files: a <code>.xaml</code> file that defines a <code><Window></code> with an empty <code><Grid></code> element, and an associated <code>.py</code> file that loads the XAML file using the <code>wpf</code> library. Typically used within a project created using one of the IronPython project templates. See Manage Python projects - Project templates .
Web Role Support Files	A <code>bin</code> folder in the project root (regardless of the selected folder in the project). The folder contains a default deployment script and a <code>web.config</code> file for Azure Cloud Service web roles. The template also includes a <code>readme.html</code> file that explains the details.
Worker Role Support Files	A <code>bin</code> folder in the project root (regardless of the selected folder in the project). The folder contains default deployment and launch script, along with a <code>web.config</code> file, for Azure Cloud Service worker roles. The template also includes a <code>readme.html</code> file that explains the details.
Azure web.config (FastCGI)	A <code>web.config</code> file that contains entries for apps using a <code>WSGI</code> object to handle incoming connections. This file is typically deployed to the root of a web server running IIS. For more information, see Configure an app for IIS .
Azure web.config (HttpPlatformHandler)	A <code>web.config</code> file that contains entries for apps that listen on a socket for incoming connections. This file is typically deployed to the root of a web server running IIS, such as Azure App Service. For more information, see Configure an app for IIS .
Azure static files web.config	A <code>web.config</code> file typically added to a <code>static</code> folder (or other folder containing static items) to disable Python handling for that folder. This config file works in conjunction with one of the FastCGI or HttpPlatformHandler config files above. For more information, see Configure an app for IIS .
Azure Remote debugging web.config	Deprecated (was used for remote debugging on Azure App Service for Windows, which is no longer supported).

See also

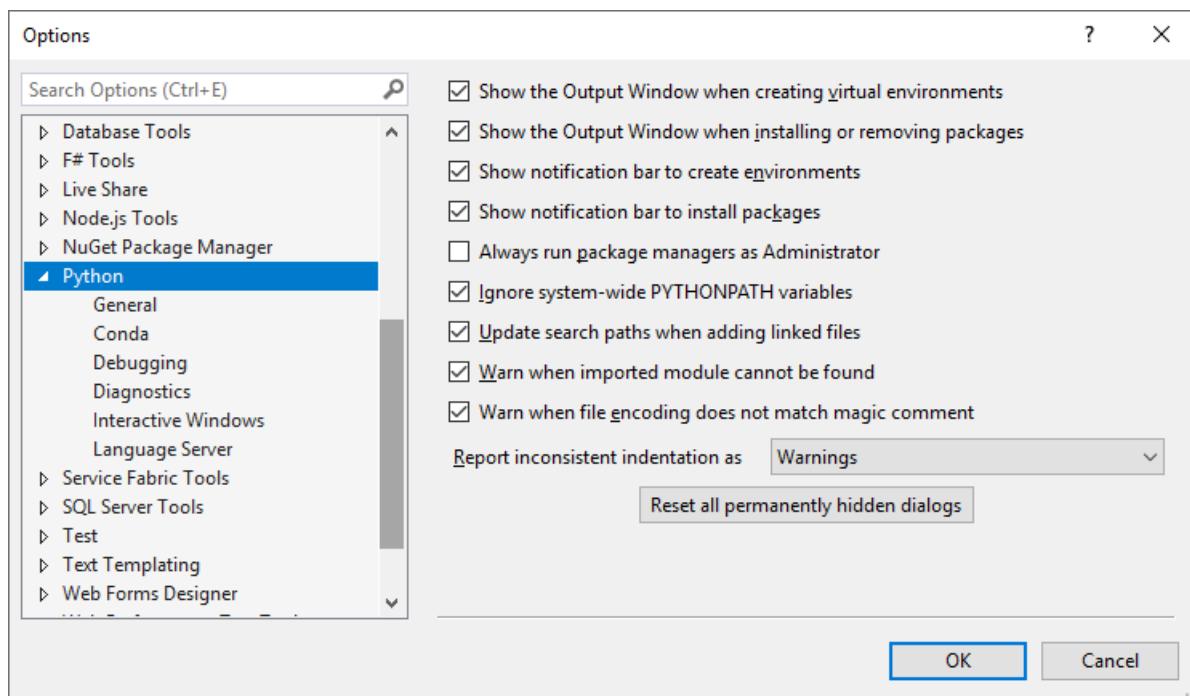
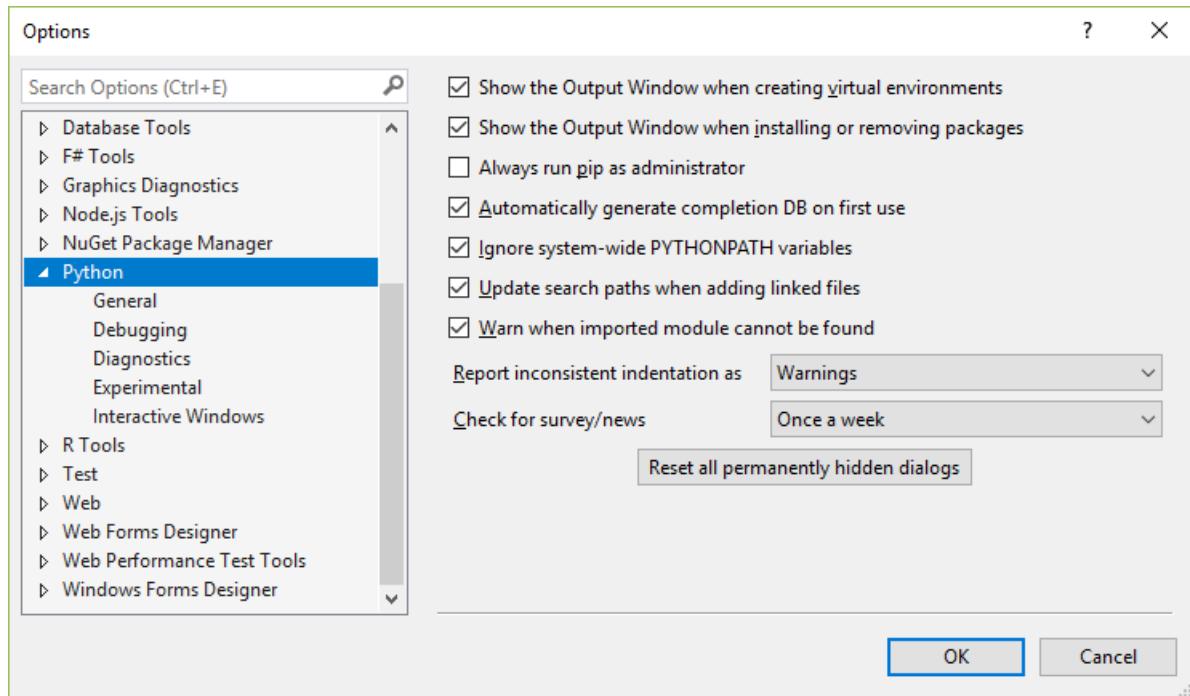
- [Manage Python projects - Project templates](#)
- [Python web project templates](#)

- Publish to Azure App Service

Options for Python in Visual Studio

4/9/2019 • 8 minutes to read • [Edit Online](#)

To view Python options, use the **Tools > Options** menu command, make sure **Show all settings** is selected, and then navigate to **Python**:



There are also additional Python-specific options on the **Text Editor > Python > Advanced** tab, and on the **Environment > Fonts and Colors** tab within the **Text Editor** group.

NOTE

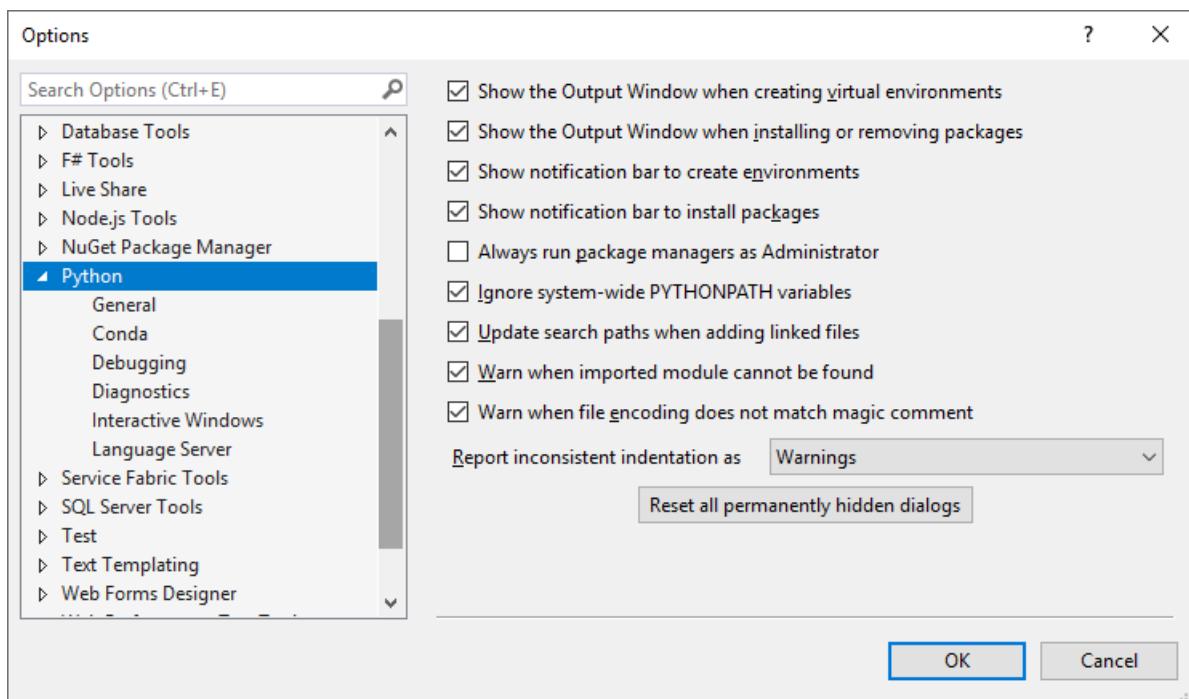
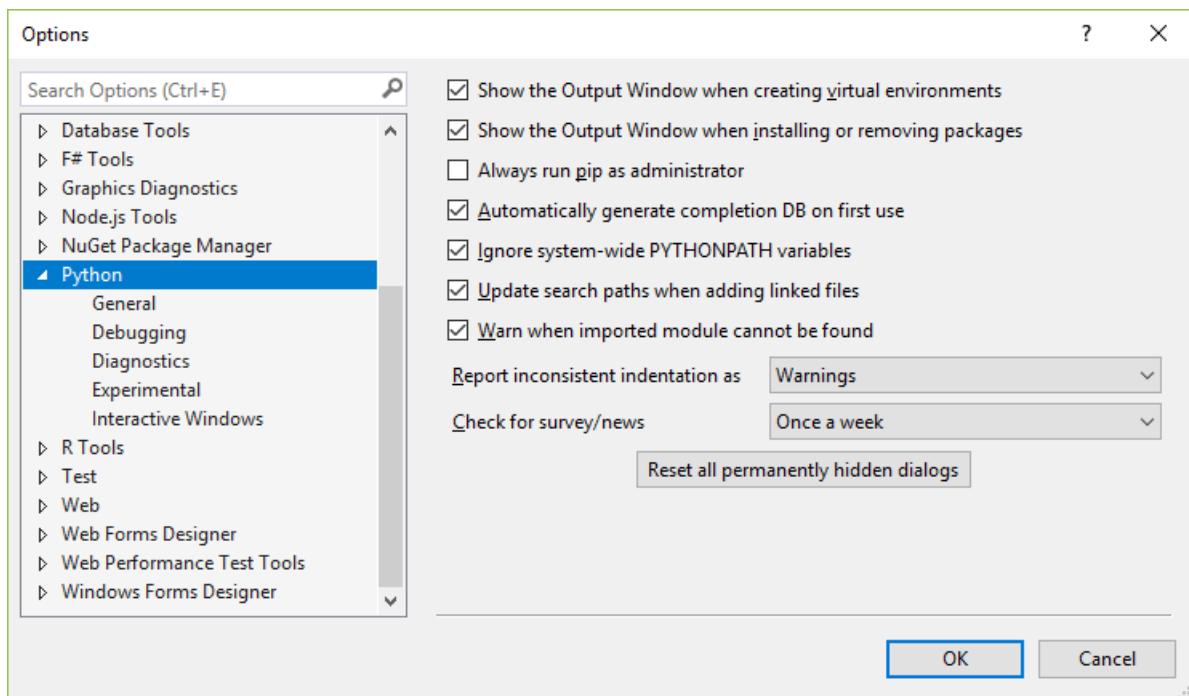
The **Experimental** group contains options for features that are still under development and are not documented here. They are often discussed in posts on the [Python engineering at Microsoft blog](#).

General options

(**Tools > Options > Python tab.**)

OPTION	DEFAULT	DESCRIPTION
Show the Output Window when creating virtual environments	On	Clear to prevent the Output window from appearing.
Show the Output Window when installing or removing packages	On	Clear to prevent the Output window from appearing.
Show notifications bar to create environments	On	<i>Visual Studio 2019 only.</i> When this option is set and the user opens a project that contains a <i>requirements.txt</i> or <i>environment.yml</i> file, Visual Studio displays an information bar with suggestions to create a virtual environment or conda environment, respectively, instead of using the default global environment.
Show notifications bar to install packages	On	<i>Visual Studio 2019 only.</i> When this option is set and the user opens a project that contains a <i>requirements.txt</i> file (and is not using the default global environment) Visual Studio compares those requirements with packages installed in the current environment. If any packages are missing, Visual Studio displays a prompt to install those dependencies.
Always run package managers as administrator	Off	Always elevates <code>pip install</code> and similar package manager operations for all environments. When installing packages, Visual Studio prompts for administrator privileges if the environment is located in a protected area of the file system such as <i>c:\Program Files</i> . In that prompt you can choose to always elevate the install command for just that one environment. See Packages tab .
Automatically generate completion DB on first use	On	<i>Applies to Visual Studio 2017 version 15.5 and earlier and to later versions when using an IntelliSense database.</i> Prioritizes completion of the database for a library when you write code that uses it. For more information, see Intellisense tab .

OPTION	DEFAULT	DESCRIPTION
Ignore system-wide PYTHONPATH variables	On	PYTHONPATH is ignored by default because Visual Studio provides a more direct means to specify search paths in environments and projects. See Search paths for details.
Update search paths when adding linked files	On	When set, adding a linked file to a project updates Search paths so that IntelliSense can include the contents of the linked file's folder in its completion database. Clear this option to exclude such content from the completion database.
Warn when imported module cannot be found	On	Clear this option to suppress warnings when you know an imported module isn't presently available but doesn't otherwise affect code operation.
Report inconsistent indentation as	Warnings	Because the Python interpreter depends heavily on proper indentation to determine scope, Visual Studio by default issues warnings when it detects inconsistent indentations that might indicate coding errors. Set to Errors to be even more strict, which causes the program to exit in such cases. To disable this behavior altogether, select Don't .
Check for survey/news	Once a week	<i>Visual Studio 2017 and earlier.</i> Sets the frequency at which you allow Visual Studio to open a window containing a web page with Python-related surveys and news items, if available. Options are Never , Once a day , Once a week , and Once a month .
Reset all permanently hidden dialogs button	n/a	Different dialog boxes provide options such as Don't show me this again . Use this button to clear those options and cause the dialogs to reappear.

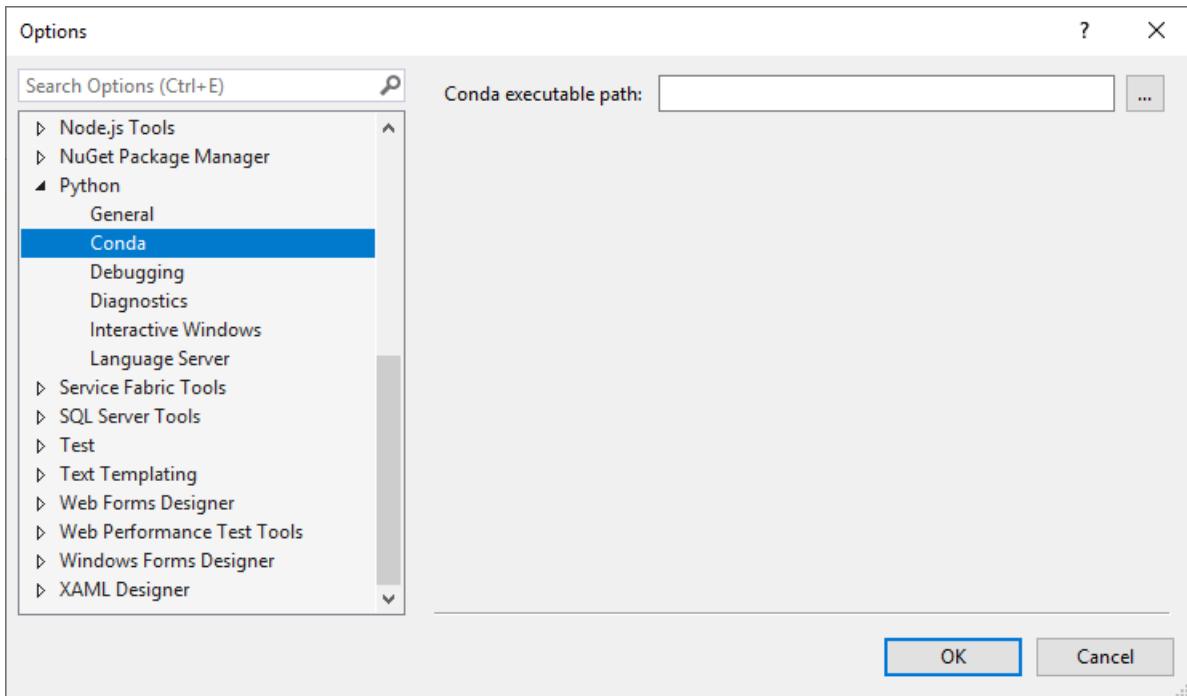


Conda options

(Tools > Options > Python > Conda tab.)

OPTION	DEFAULT	DESCRIPTION

OPTION	DEFAULT	DESCRIPTION
Conda executable path	(blank)	Specifies an exact path to the <i>conda.exe</i> executable rather than relying on the default Miniconda installation that's included with the Python workload. If another path is given here, it takes precedence over the default installation and any other <i>conda.exe</i> executables specified in the registry. You might change this setting if you manually install a newer version of Anaconda or Miniconda, or want to use a 32-bit distro rather than the default 64-bit distro.

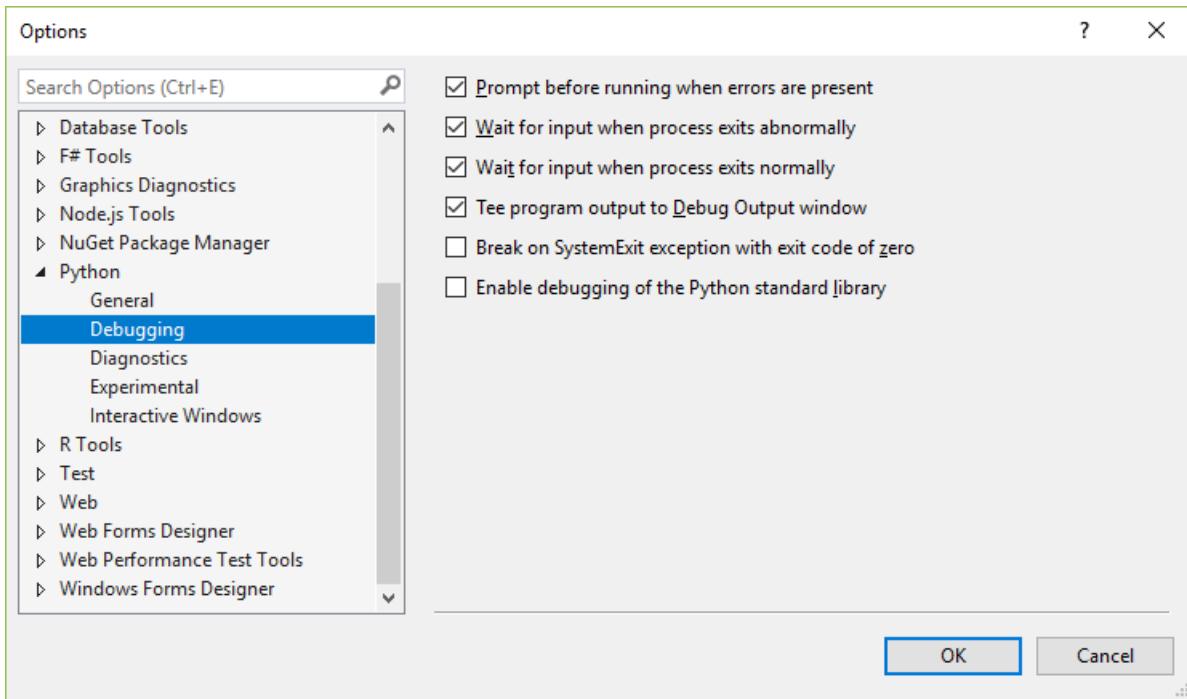


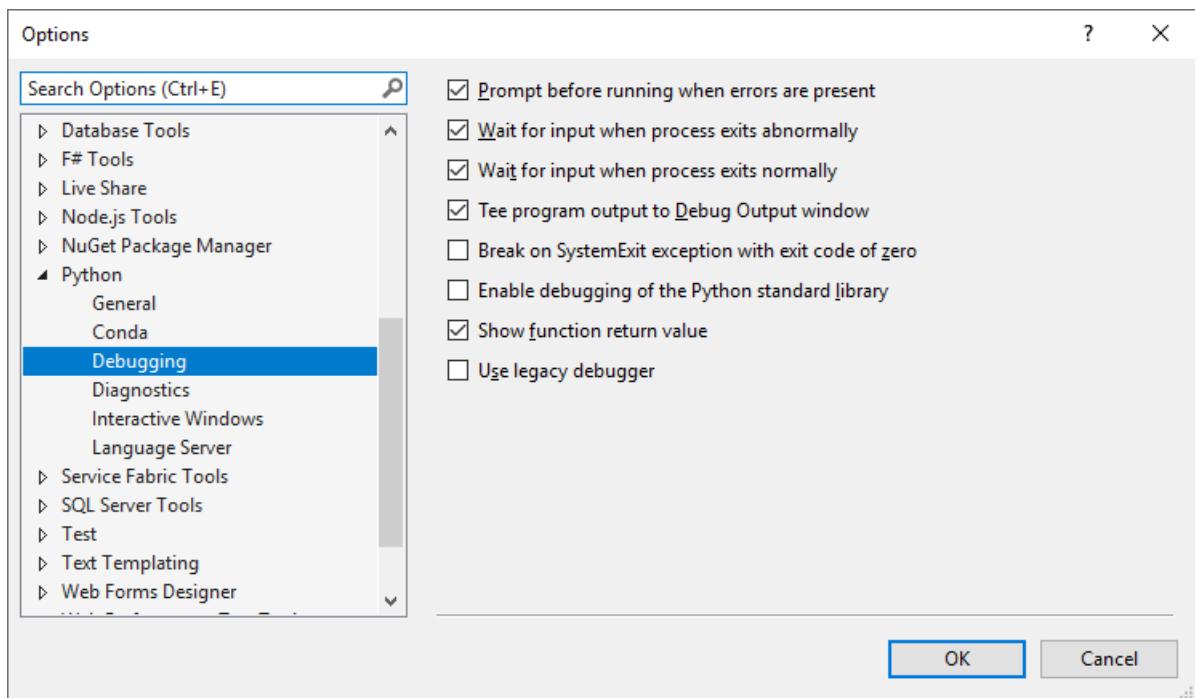
Debugging options

(Tools > Options > Python > Debugging tab.)

OPTION	DEFAULT	DESCRIPTION
Prompt before running when errors are present	On	When set, prompts you to confirm that you want to run code that contains errors. Clear this option to disable the warning.
Wait for input when process exits abnormally	On (for both)	A Python program started from Visual Studio runs in its own console window. By default, the window waits for you to press a key before closing it regardless of how the program exits. To remove that prompt and close the window automatically, clear either or both of these options.
Wait for input when process exits normally		

OPTION	DEFAULT	DESCRIPTION
Tee program output to Debug Output window	On	Displays program output in both a separate console window and the Visual Studio Output window. Clear this option to show output only in the separate console window.
Break on SystemExit exception with exit code of zero	Off	If set, stops the debugger on this exception. When clear, the debugger exits without breaking.
Enable debugging of the Python standard library	Off	Makes it possible to step into the standard library source code while debugging, but increases the time it takes for the debugger to start.
Show function return value	On	<i>Visual Studio 2019 only.</i> Displays function return values in the Locals window then stepping over a function call in the debugger (F10)
Use legacy debugger	Off	<i>Visual Studio 2019 only.</i> Instructs Visual Studio to use the legacy debugger by default. For more information, see Debugging - Use the legacy debugger .

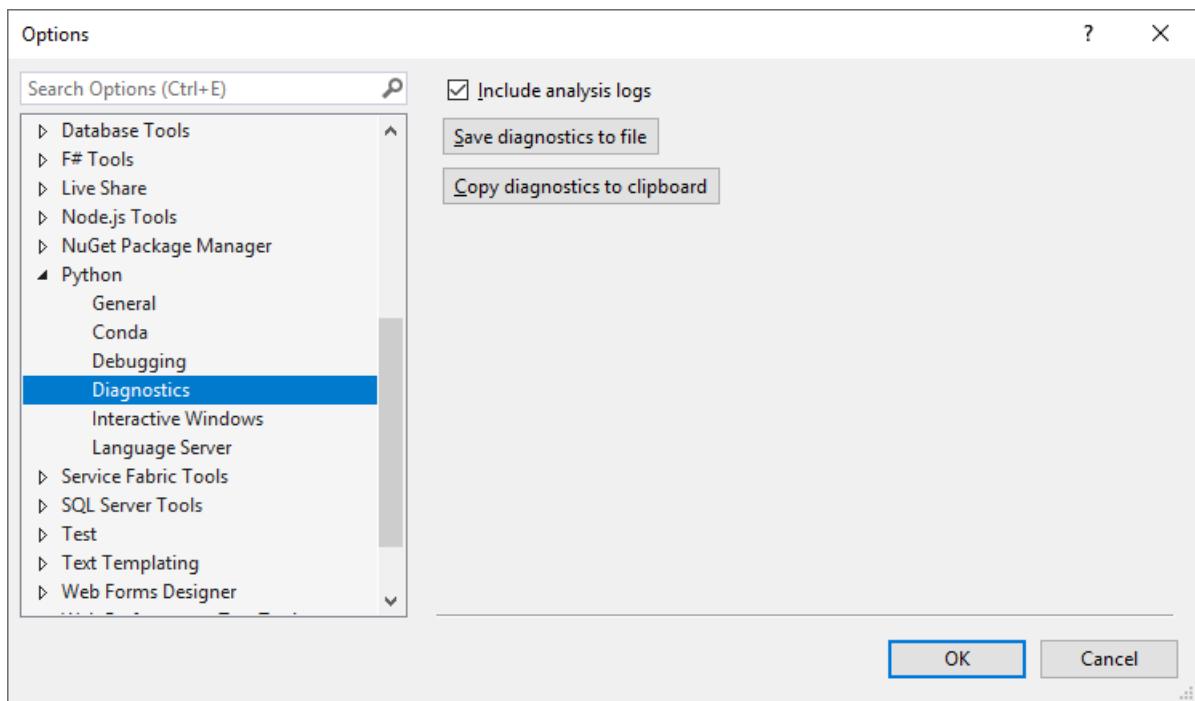




Diagnostics options

(Tools > Options > Python > Diagnostics tab.)

OPTION	DEFAULT	DESCRIPTION
Include analysis logs	On	Includes detailed logs relating to analysis of installed Python environments when saving diagnostics to a file or copying them to the clipboard using the buttons. This option may significantly increase the size of the generated file, but is often required to diagnose IntelliSense issues.
Save diagnostics to file button	n/a	Prompts for a filename, then saves the log to a text file.
Copy diagnostics to clipboard button	n/a	Places the entirety of the log on the clipboard; this operation may take some time depending on the size of the log.

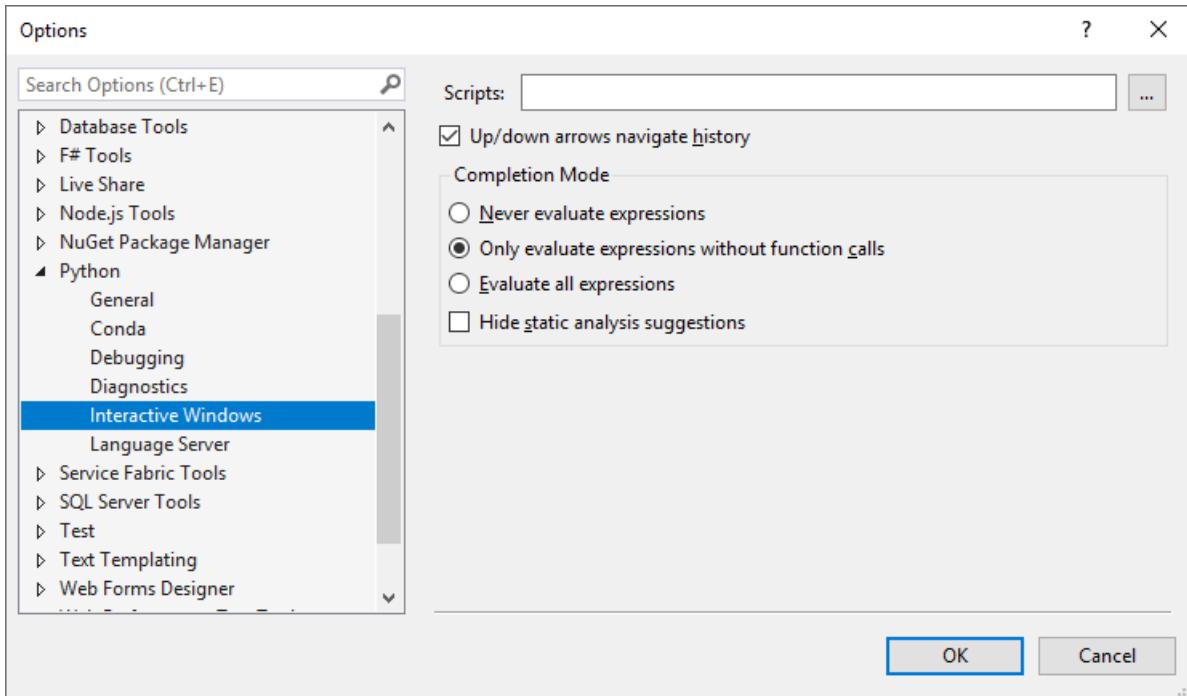


Interactive Windows options

(Tools > Options > Python > Interactive Windows tab.)

OPTION	DEFAULT	DESCRIPTION
Scripts	n/a	Specifies a general folder for startup scripts to apply to Interactive windows for all environments. See Startup scripts . Note, however, that this feature does not currently work.
Up/down arrows navigate history	On	Uses the arrow keys to navigate through history in the Interactive window. Clear this setting to use the arrow keys to navigate within the Interactive window's output instead.
Completion mode	Only evaluate expressions without function calls	The process of determining the available members on an expression in the Interactive window may require evaluating the current unfinished expression, which can result in side-effects or functions being called multiple times. The default setting, Only evaluate expressions without function calls excludes expressions that appear to call a function, but evaluates other expressions. For example, it evaluates <code>a.b</code> but not <code>a().b</code> . Never evaluate expressions prevents all side-effects, using only the normal IntelliSense engine for suggestions. Evaluate all expressions evaluates the complete expression to obtain suggestions, regardless of side effects.

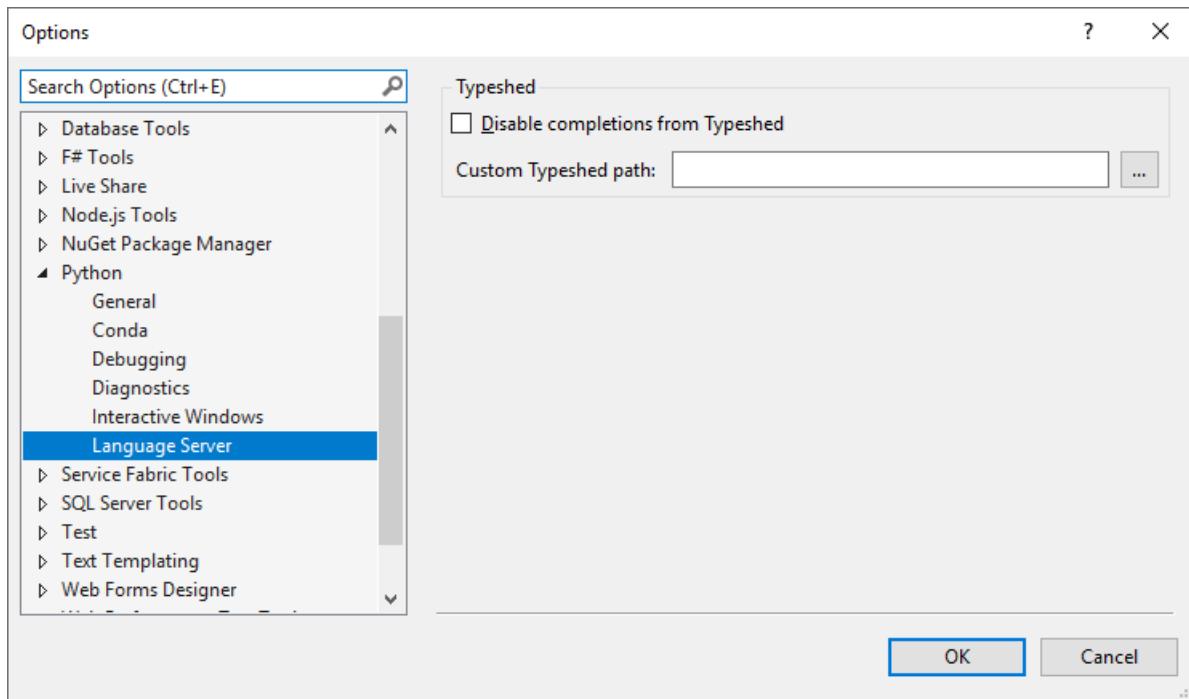
OPTION	DEFAULT	DESCRIPTION
Hide static analysis suggestions	Off	When set, displays only suggestions that are obtained by evaluating the expression. If combined with the Completion mode value Never evaluate expressions , no useful completions appear in the Interactive window.



Language server options

(Tools > Options > Python > Language server tab.)

OPTION	DEFAULT	DESCRIPTION
Disable completions from Typeshed	Off	Visual Studio IntelliSense normally uses a bundled version of Typeshed (a set of .pyi files) to find type hints for standard library and third-party libraries for both Python 2 and Python 3. Setting this option disables the bundled TypeShed behavior.
Custom Typeshed path	(blank)	If set, Visual Studio uses the Typeshed files at this path instead of its bundled version. Ignore if Disable completions from Typeshed is set.



Advanced Python editor options

(Tools > Options > Text Editor > Python > Advanced tab.)

Completion Results

OPTION	DEFAULT	DESCRIPTION
Member completion displays intersection of members	Off	When set, shows only completions that are supported by all possible types.
Filter list based on search string	On	Applies filtering of completion suggestions as you type (default is checked).
Automatically show completions for all identifiers	On	Clear this option to disable completions in both the editor and Interactive windows.

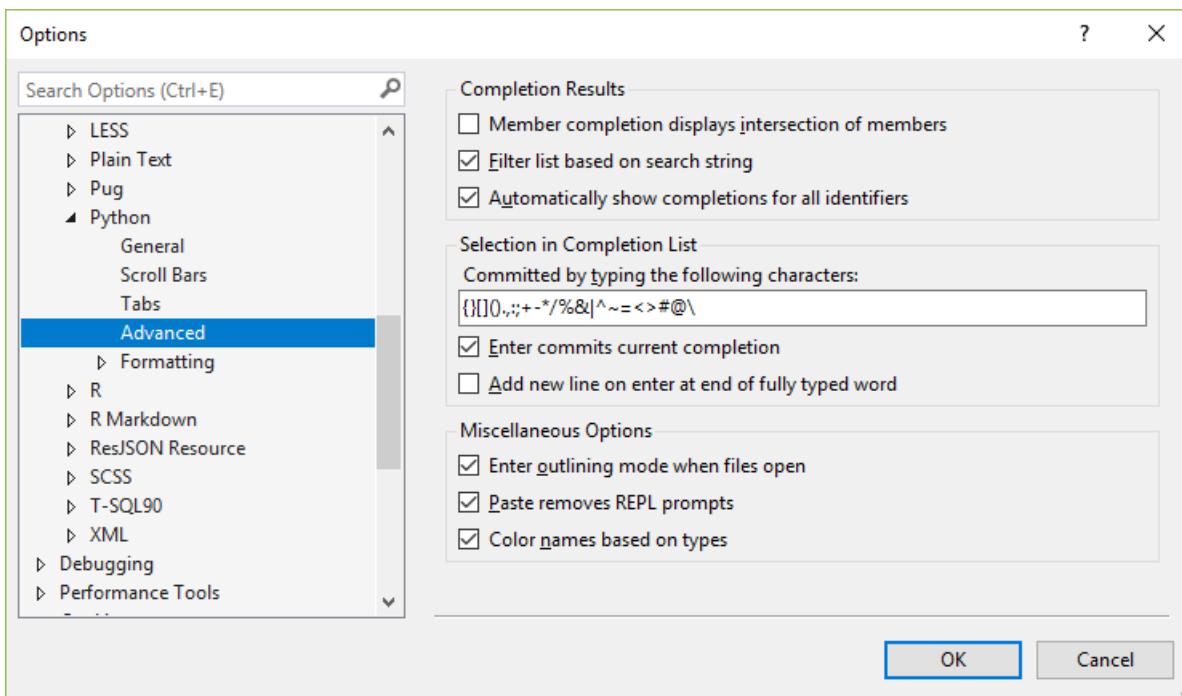
Selection in Completion List

OPTION	DEFAULT	DESCRIPTION
Committed by typing the following characters	{[]().,:+/*%& ^~=<>#@\`	These characters typically follow an identifier that one might select from a completion list, so it's convenient to commit the completion simply by typing a character. You can remove or add specific characters to the list as desired.
Enter commits current completion	On	When set, the Enter key chooses and applies the currently selected completion as with the characters above (but of course, there isn't a character for Enter so it couldn't go into that list directly!).

OPTION	DEFAULT	DESCRIPTION
Add new line on enter at end of fully typed word	Off	By default, if you type the entire word that appears in the completion popup and press Enter , you commit that completion. By setting this option, you effectively commit completions when you finish typing the identifier, such that Enter inserts a new line.

Miscellaneous Options

OPTION	DEFAULT	DESCRIPTION
Enter outlining mode when files open	On	Automatically turn on Visual Studio's outlining feature in the editor when opening a Python code file.
Paste removed REPL prompts	On	Removes >>> and ... from pasted text, allowing easy transfer of code from the Interactive window to the editor. Clear this option if you need to retain those characters when pasting from other sources.
Color names based on types	On	Enables syntax coloring in Python code.



Fonts and Colors options

(Environment > Fonts and Colors tab within the **Text Editor** group.)

The names of the Python options are all prefixed with **Python** and are self-explanatory. The default font for all Visual Studio color themes is 10pt Consolas regular (not bold). The default colors vary by theme. Typically, you change a font or color if you find it difficult to read text with the default settings.

