Community Experience Distilled

Python Tools for Visual Studio

Leverage the power of the Visual Studio IDE to develop better and more efficient Python projects

Martino Sabia
Cathy Wang

[PACKT] PUBLISHING open source* community experience distilled

Python Tools for Visual Studio

Leverage the power of the Visual Studio IDE to develop better and more efficient Python projects

Martino Sabia

Cathy Wang

[PACKT] open source
community experience distilled
PUBLISHING
BIRMINGHAM - MUMBAI

Python Tools for Visual Studio

Copyright © 2014 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: April 2014

Production Reference: 1140414

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-868-7

www.packtpub.com

Cover Image by Cathy Wang (ms.cathywang@gmail.com)

Credits

Authors
Martino Sabia
Cathy Wang

Reviewers
Steve Dower
Fabio Lonegro
Chris Marinic

Commissioning Editor
Anthony Albuquerque

Acquisition Editor
Harsha Bharwani

Content Development Editor
Sriram Neelakantan

Technical Editor
Shashank Desai

Copy Editors
Roshni Banerjee
Gladson Monteiro

Project Coordinator
Melita Lobo

Proofreader
Paul Hindle

Indexers
Monica Ajmera Mehta
Priya Subramani

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

About the Authors

Martino Sabia is a curious-minded developer with close to 30 years of coding experience. Throughout his years of working with different platforms and languages, he has always kept his mind fresh while finding creative ways of using different technologies. Based in Italy, Martino has spent his career in various start-up companies, working in numerous roles from junior developer to software architect. Now he is the Project Lead for Deltatre; he works on consumer-facing, heavy-traffic websites and media-streaming platforms in the sports industry.

Cathy Wang is an experienced designer who specializes in service design and experience strategy. She has worked on many cross-channel projects and served as a design lead for enterprise services around the globe in fields ranging from Telecom to public sectors. Cathy has worked for world-class design agencies to help bring visions to life. In her free time, she builds web projects and apps. She is infinitely curious about new technologies and the experiences they can bring.

About the Reviewers

Steve Dower works at Microsoft and is a developer of Python Tools for the Visual Studio team.

Fabio Lonegro has spent many years doing research in theoretical physics (String and Gauge theory) and collaborating with many divulgating projects, including the translation of Peter Woit's book *Not Even Wrong*. He was always passionate about web development and has spent the last 15 years working on web projects related to e-learning and data visualization. He is now a developer at Deltatre spa, where his work is focused on many fields, from the integration of complex data with multimedia streams for both mobile and desktop experiences to custom solutions for web content indexing and the development of Node.js. Currently, he uses Python for a variety of applications that involve data which comes from Arduino and Raspberry Pi shields. He is also a capoeira teacher, a passionate cyclist, and above all, a caring father.

Chris Marinic is an autodidact with decades of engineering experience. Growing up, he excelled at computer science, often mentoring his fellow students. He designed, developed, launched, and sold his own start-up while working full-time as the Director of Engineering at Sabre Hospitality Solutions.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface **1**

Chapter 1: Introduction to PTVS **7**

Step-by-step installation and configuration **7**
PTVS tools overview **12**
The Python Environments window 13
Python Interactive 14
Visual Studio panels with PTVS **14**
Summary **16**

Chapter 2: Python Tools in Visual Studio **17**

Mastering IntelliSense with Python **17**
Using REPL in Visual Studio **21**
Navigating code with ease **24**
Object Browser **28**
Summary **31**

Chapter 3: Day-to-day Coding Tools **33**

Project handling **33**
Solution 33
Project 34
Specifying Python environments 37
Defining Search Paths 41
Refactoring **42**
Debugging **46**
Using breakpoints 47
Utilizing watch entries 48
Summary **49**

Chapter 4: Django in PTVS **51**
Django project template and tools **52**
Installing a Python package 53
Running the application 55
IntelliSense in Django templates 57
Setting up and managing a database for a Django project **58**
Setting up the admin interface **61**
Creating a new Django application **63**
Deploying a Django project on Microsoft Azure **65**
Summary **71**
Chapter 5: Advanced Django in PTVS **73**
Library management **73**
The Fabric library – the deployment and development task manager **75**
South – the database deployment library **79**
Why use South with Django 80
Installing South 80
Schema migration with South 83
Summary **87**
Chapter 6: IPython and IronPython in PTVS **89**
IPython in PTVS **89**
IronPython **95**
Using .NET classes in Python code with IronPython 95
Using the Python code in .NET with IronPython 100
Summary **105**
Index **107**

Preface

Like many other developers, Python developers have always had to find ways to manage the development workflow between different tools. Most of the time, this happens without using a comprehensive guide that is available in a complete IDE which is specifically designed for Python development.

The rare, exceptional IDEs that offer complete guides are often expensive and don't provide hands-on steps to help speed up the development process.

Visual Studio, as a matured and well-developed tool over the last few decades, has dominated the market of compiled languages and languages that are strictly oriented toward Windows and .NET. Packed with handy tools and functionalities to speed up and facilitate the workflow of developers, it helps users to render repetitive tasks, manage projects, and provide a detailed outlook into the structure of a project. However, most importantly, it helps users gain a clear view into the inner structure of the code.

In the last few years, Microsoft has started exploring how to integrate new languages into Visual Studio; as a result, Python Tools for Visual Studio (PTVS) was developed. It's a well-developed tool that is already on its second release and is commonly used by professional developers as their new IDE of choice for Python projects.

PTVS has everything that a Python developer can dream of: consistent project files management, interactive debugging and code completion features with the rock solid Microsoft IntelliSense technology, project templates, a first-class Django integration package, virtual environment management right in the IDE for REPL, and a native code-based IDE that loads and reacts fast.

This book will focus more on the integration of Python in Visual Studio than the language itself. It will try to delve into the power offered by the tool and venture into the feasibility of its day-to-day usage for a developer. We will show real examples of how to use PTVS with Django and how to deal with occasional difficulties when it comes to integrating well-known libraries into a Python project on Microsoft Windows.

What this book covers

Chapter 1, *Introduction to PTVS*, provides a high-level overview of PTVS and the interaction between Visual Studio and a Python interpreter.

Chapter 2, *Python Tools in Visual Studio*, provides an in-depth analysis of the tools, type checking, inner functionalities, and automatisms (IntelliSense and REPL) of PTVS.

Chapter 3, *Day-to-day Coding Tools*, talks about browsing through the code and the flexible setting of Python environments. It also talks about refactoring and the debugging process.

Chapter 4, *Django in PTVS*, shows how to harness the powerful Visual Studio IDE and tooling to speed up Django development.

Chapter 5, *Advanced Django in PTVS*, provides an in-depth look at remote task management and schema migrations using the third-party Python libraries Fabric and South.

Chapter 6, *IPython and IronPython*, provides an overview of the IPython library and how it's integrated in Visual Studio. It also provides an introduction to IronPython and its integration with the .NET framework.

What you need for this book

You will need a basic understanding of Python, a computer with Windows installed, and an Internet connection. To follow through the exercises and examples, we would suggest that you have Visual Studio as well.

Who this book is for

This book is intended for developers who are aiming to enhance their productivity in Python projects with automation tools that Visual Studio provides for the .NET community. Some basic knowledge of Python programming is essential.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
class foo:
    """
    Documentation of the class.
    It can be multiline and contain any amount of text
    """
    @classmethod
    def bar(self, first=0, second=0):
        """This is the documentation for the method"""
        return first + second

print(foo.bar())
```

Any command-line input or output is written as follows:

```
python manage.py schemamigration south2ptvs --initial
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Clicking on the **Next** button moves you to the next screen."

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: `https://www.packtpub.com/sites/default/files/downloads/8687OS_ColoredImages.pdf`

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

Special thanks from the authors

Thanks to Packt Publishing for giving us the opportunity to publish this book for the developer community, and the help they have provided during the entire process: from the injection of the idea to the whole process of giving birth to it. It has been a journey filled with surprises and discoveries.

We'd also like to appreciate our reviewers, Fabio Lonegro and Chris Marinic, who have provided us with clear and unbiased feedback along the way, giving us great insights on untangling the details of the book.

Last but not least, we would like to thank the Microsoft PTVS team, specifically Steve Dower, who has contributed to the book personally and through providing technical support on every detail. Thanks to Shahrokh Mortazavi for reaching out to us through a tweet (`https://twitter.com/cathycracks/status/421336498748006400`). Steve and the rest of the team have given us lots of help, insights, and suggestions on how to overcome some complex but very important parts of the book. They even invited us to visit them in person to gain a greater insight into their work. We truly feel that PTVS is developed by a group of passionate people who care for the community and are eager to develop PTVS to be an even better and useful tool. The Microsoft PTVS team has done a great job with the tool so far in our opinion, and we look forward to what's yet to come.

We have enjoyed this journey so far, and we are very happy to be doing this together to bring this book to life. It has been an intimate and difficult process filled with love and with some very deep and long discussions into late nights. We hope that you enjoy and gain knowledge from this book as much as we have learned from it.

We hope that you will find this book interesting and that it will help you discover the inner power of PTVS, as Scott Hanselman described PTVS in a post on his blog, One of Microsoft's Best-Kept Secrets - Python Tools for Visual Studio (PTVS), created on July 2, 2013 and found at `http://www.hanselman.com/blog/OneOfMicrosoftsBestKeptSecretsPythonToolsForVisualStudioPTVS.aspx`.

1
Introduction to PTVS

Python Tools in Visual Studio (PTVS) is an extremely powerful tool because of the following reasons:

- It gives Python developers a powerful IDE with many helpful coding features and integrations in one unique environment.
- PTVS provides developers on the Windows platform the opportunity to use their favorite IDE—Visual Studio—to explore, learn, and manage one of the most commonly used scripting languages.

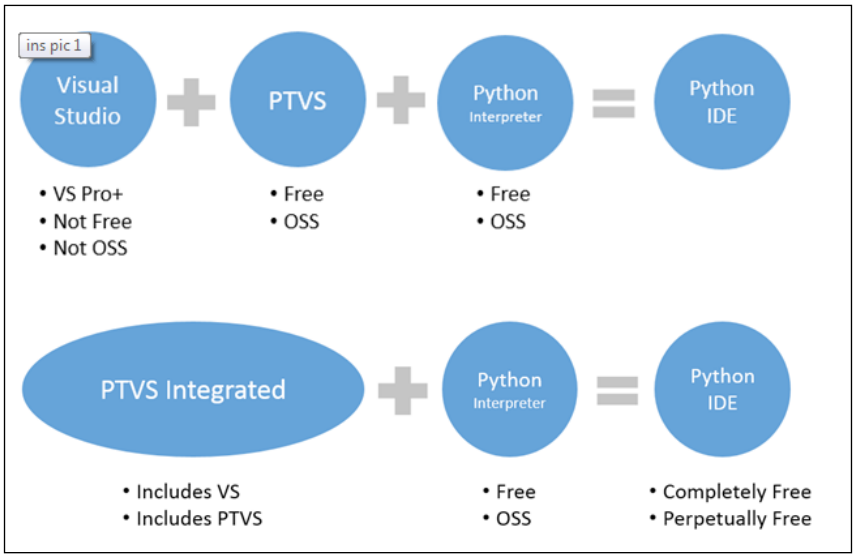
In this chapter, we will have a high-level overview of PTVS, starting with a step-by-step tutorial for installing and configuring it correctly followed by a quick overview of the principle tools of Visual Studio to control the Python environment and configuration. Understanding the Visual Studio windows will greatly benefit your ability to explore and manage workflows of the source code and the structure of your Python project.

Step-by-step installation and configuration

There are various formats of PTVS available for installation depending on your preexisting installed version of Visual Studio. PTVS is available for Visual Studio 2010, 2012, and 2013 (Pro edition or above).

If the previously mentioned versions of Visual Studio are not installed on your computer, it's possible to install a standalone version of PTVS. Visual Studio permits side-by-side installation, meaning it provides the ability to install multiple versions on one system. The only prerequisite is that the older version must be installed before the newer one.

The different types of installations possible for PTVS are described on its **CodePlex** website, `http://go.microsoft.com/fwlink/?LinkID=390659`.



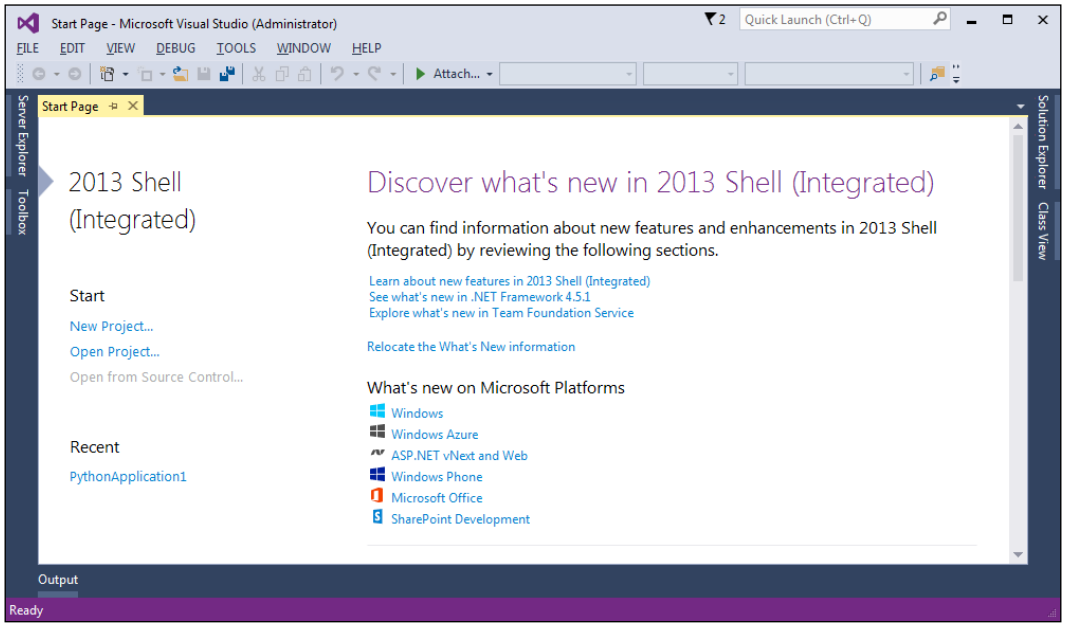
The preceding figure is taken from `http://go.microsoft.com/fwlink/?LinkID=390659`.

The most important prerequisite for Visual Studio 2013 is to have Windows 7 (32 or 64 bit) or above running as your operating system.

Once you have sorted out the prerequisites and installed the PTVS package of your choice, you will need to decide on the type of Python interpreter. Choosing the appropriate Python interpreter depends on your need for your project. Refer to the PTVS CodePlex page at `http://go.microsoft.com/fwlink/?LinkID=299429` to help your decision-making process. You can choose between CPython and IronPython (32 or 64 bit). If you chose CPython, then you can choose between Python Version 2.7 and 3.3. It is recommended to use CPython 3.3 32 bit for most cases. For web development, the recommendation would be CPython 2.7 32 bit.

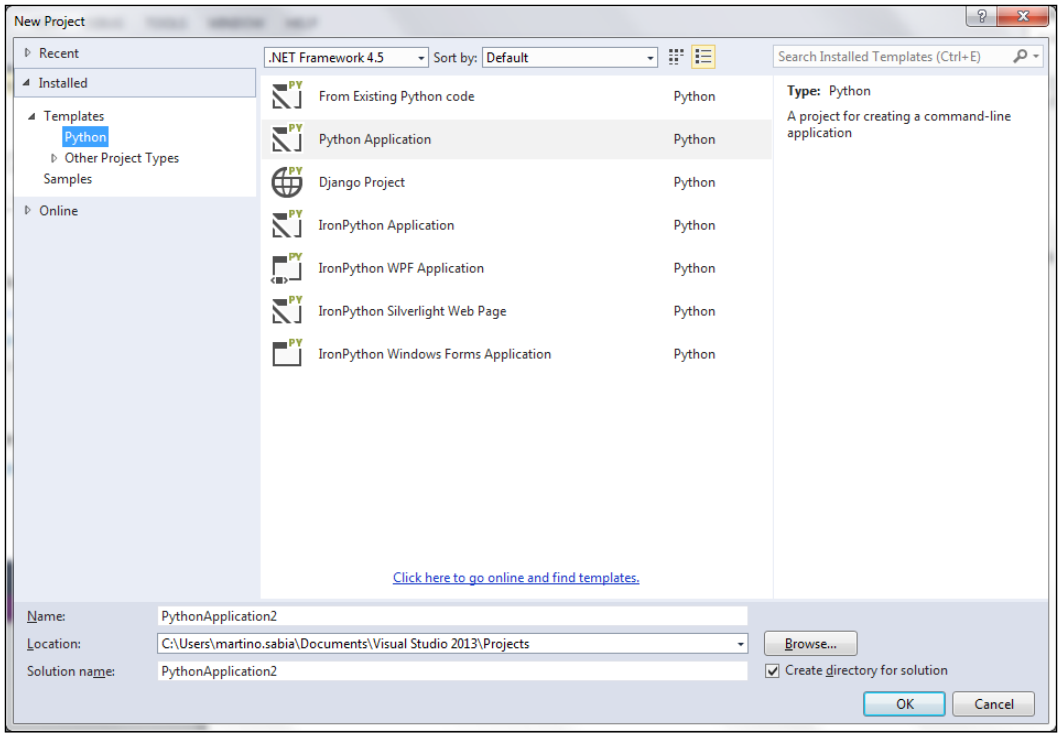
Make your choice based on what you intend to do and the framework that you will be using with Python. For the scope of this book, we suggest to install the 32-bit CPython Version 2.7. For the latest complete list of downloadable Python interpreters, please refer to the PTVS CodePlex page at `http://go.microsoft.com/fwlink/?LinkID=390659`.

Once the interpreter is installed, you can fire up PTVS by opening the Visual Studio 2013 application from the **Start** menu. If everything works, this is what you are going to see on your screen:

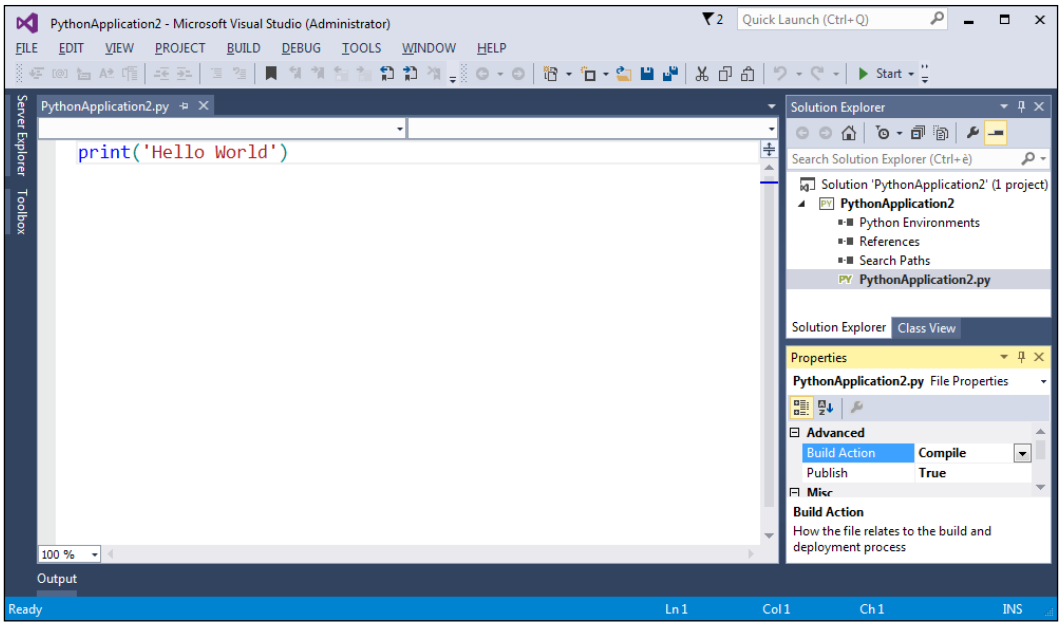


Let's check whether the whole system works properly. Create a new project and see if it runs as follows:

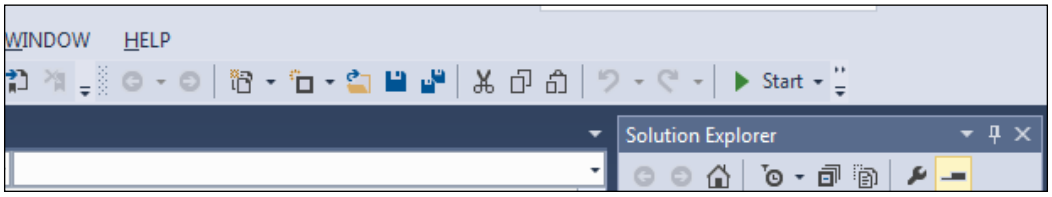
1. To create a new project, navigate to the **New Project** menu under **File** to launch the **New Project** dialog box.



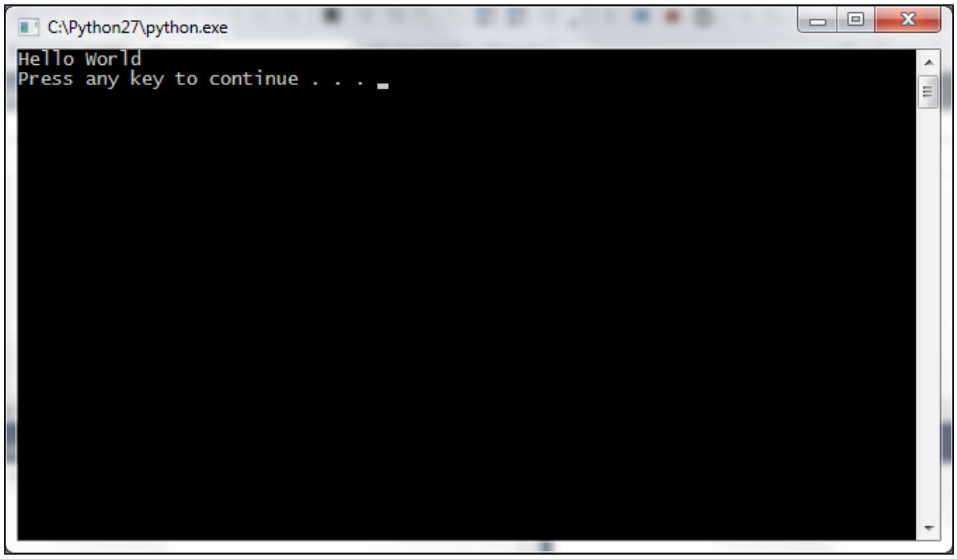
2. Select **Python Application** and click on **OK**. This will create a new project and a basic `Hello World` Python application file.



3. Start the app by clicking on the **Start** button in the toolbar, or just hit *F5*.

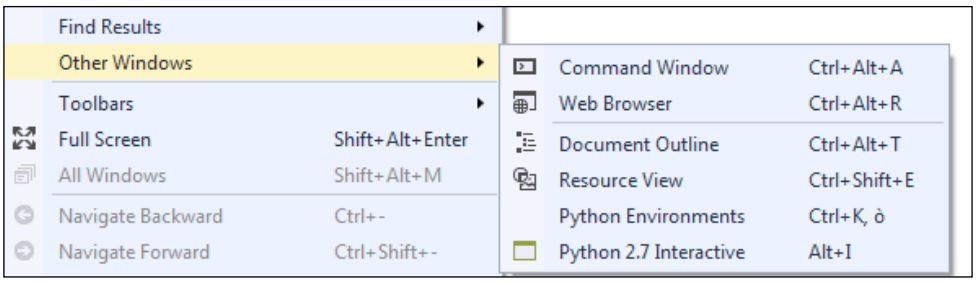


4. If you have any previous installations of Python on your system, you should see the application response window with the **Hello World** message, as shown in the following screenshot:



PTVS tools overview

Now that you have PTVS up and running, let's take a closer look at the various tools provided by Visual Studio that empower the Python development cycle. Let's start with the windows, which are accessible through the **View** menu:



From the **View** menu, you can choose two windows that are more important for Python:

- **Python Environments**
- **Python Interactive**

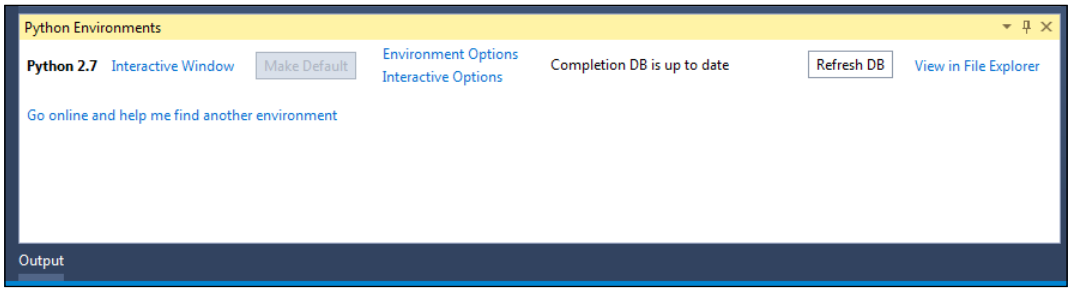
The Python Environments window

The Python Environments window shows all the Python interpreters' versions (environments) installed on the system. For each of them, an interactive window called **read–eval–print loop** (**REPL**) can be accessed, and it's possible to see the status of the package analysis made on all the packages installed. This is used by Visual Studio to carry out syntax and type analyses of all the classes and methods available for a given Python environment.

Since the analysis of Python code is complex, it's possible that you will not see progress in the **Completion DB** when you open it the first time. Even if Visual Studio performs background analysis to not interfere with the user experience of the IDE, the first analysis can take from one minute to an hour. This depends on different factors such as the number of installed libraries in the Python environment and the system resources available. Once the analysis of all the Python packages in Visual Studio is complete, the message **Completion DB is up to date** will be shown on the row of a given Python environment.

The **Completion DB** is automatically updated every time we open a new project in PTVS or install a new Python library; in such cases, Visual Studio reruns the background analysis on the new reachable code.

Sometimes, the automated background analysis process could be disabled or blocked, and the lists of installed libraries are not shown automatically. If the newly installed libraries and packages are not shown, we can manually trigger the analysis process with the **Refresh DB** button. By clicking on the button, we re-enable it, forcing a background analysis.

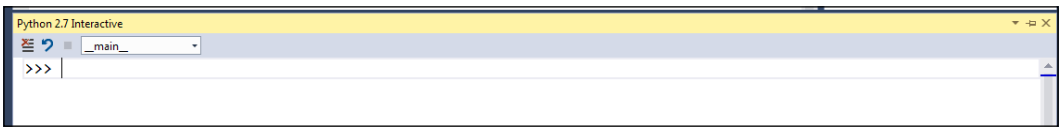


The Python Environments tool window with the list of installed Python environments and the tool buttons to access most used functions

Clicking on the **View in File Explorer** link in the **Python Environments** window will provide you with direct access to the Python installation folder.

Python Interactive

The Python Interactive window gives you access to the standard REPL tool for Python directly in the IDE along with the ability to access the modules that you are developing. This is a great and quick way to debug and test some code snippets.

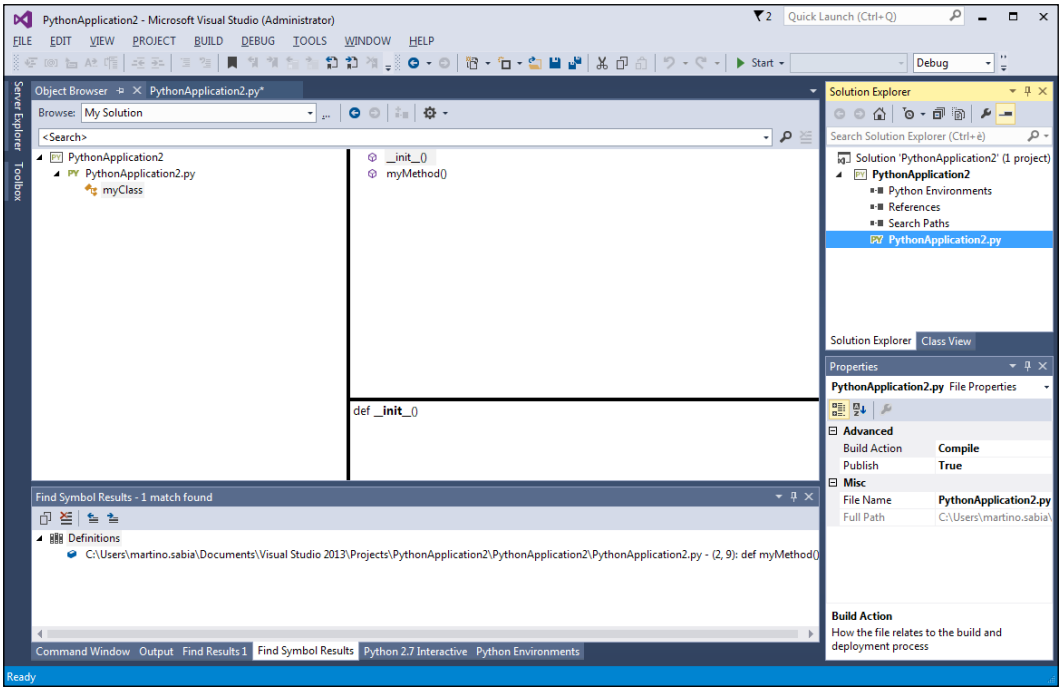


The Python Interactive tool window from where you can access the Python standard REPL tool

Besides the normal Python commands available in the standard Python REPL, Microsoft has further added some commands that are reachable by the *$* (dollar) symbol. The list of available commands is available through `$help`.

Visual Studio panels with PTVS

Visual Studio offers lots of standard tool windows to control the structure and workflow of your application. The main tool windows are **Solution Explorer**, **Properties**, **Find Symbol Results**, and **Object Browser**, as shown in the following screenshot:

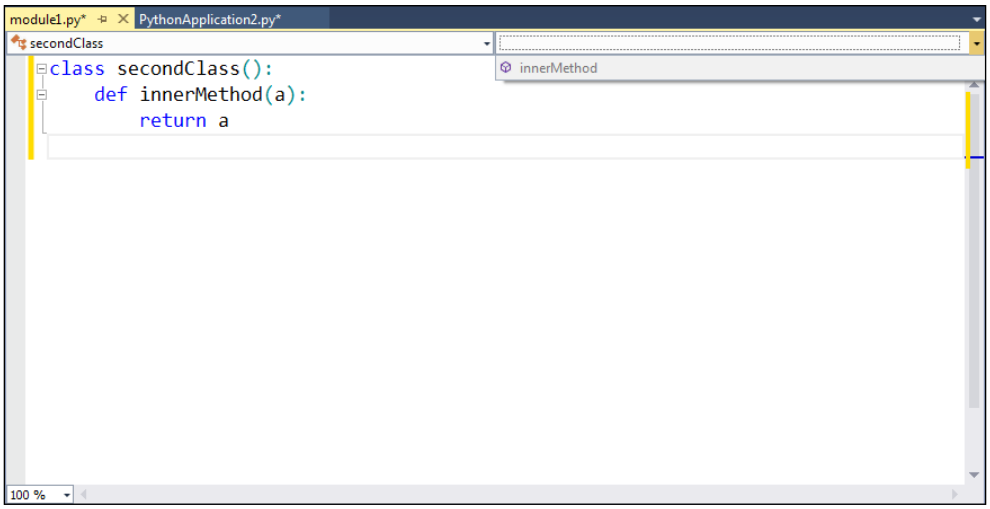


To the right, we have the **Solution Explorer** window. It provides a glimpse of the structure of the current solution. In Visual Studio, a solution is a bundle of projects. In the **Solution Explorer** window, not only can you manage the different source files of the projects, but also configure the Python environment and the packages used in it (i.e. references and dependencies).

Besides the file structure of the project, the **Solution Explorer** window also provides a class view, which shows an overview of all the classes and structures (i.e. fields, properties, and methods). This is a quick view of the more complex window, **Object Browser**, which is visible in the middle of the screenshot. This window is accessible through the **Object Browser** menu item under **View** (or using the *Ctrl* + *Alt* + *J* shortcut). The two tools together provide a manner to browse and navigate the object structure of your code.

Under the **Solution Explorer** window, we can find the **Properties** window that shows the properties related to various objects of your projects such as the single source code files in it. It also shows more detailed information, for example, the path, and how it has to be managed in the built system of the files.

The most important and powerful window we have in PTVS is the source code window, which is where any programmer spends most of his/her time. It provides multitab source code navigation; every pane is a single source code file:



In each pane there are two comboboxes. The left one provides the function to navigate between classes in the file; the right one provides the function to navigate between methods of the selected class. In the source code window, Visual Studio unleashes much more powerful tools such as refactoring, IntelliSense, and code traversing, which we will explore in depth in the next chapters.

There are other windows that will become clearer during our exploration of PTVS in the coming chapters, such as the **Find Symbol Results** window at the bottom of the screenshot. That window shows the result of a search command or the list of references of a given code element, like a method, class, or property.

Summary

In this chapter, we introduced a quick high-level overview of PTVS and the basis of it. Now that you have PTVS up and running and have familiarized yourself with the two windows, you are ready to dive into PTVS with more detailed project knowledge in the following chapters.

In the next chapter, we'll go in to more detail and start to analyze the coding tools that Visual Studio provides in PTVS that can tremendously help during the coding process and also manage Python projects.

2
Python Tools in Visual Studio

Now that we have our tools up and running, we can start to take a deeper look into one of the most important features of PTVS: the intelligent code completion feature or IntelliSense of Visual Studio.

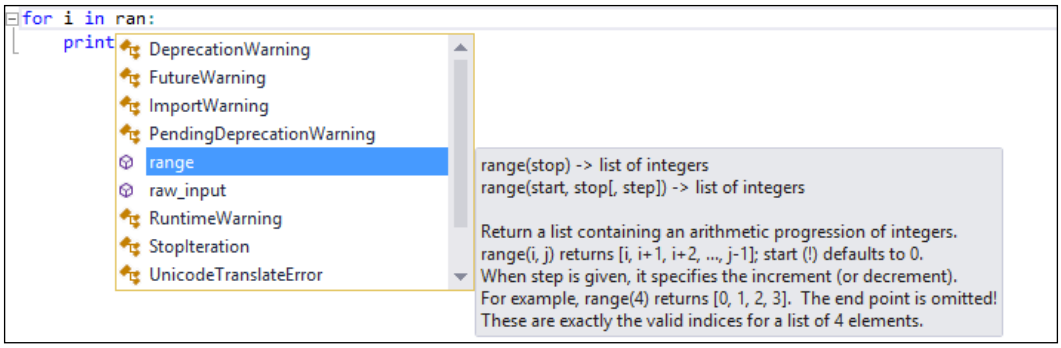
In this chapter, we will dig deeper into the automatic syntax and hierarchic analysis tools of Visual Studio that we can use with Python. Essentially, these are IntelliSense and navigation tools, which are really helpful for a Python developer during the coding process.

Furthermore, we will see how to maximize the capabilities of Visual Studio in conjunction with the inner REPL tool for quick and useful code testing and debugging.

Mastering IntelliSense with Python

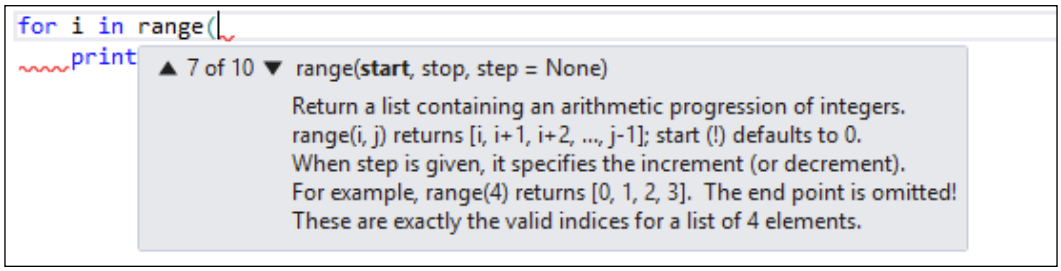
As shown in the **Python Environments** tab, Visual Studio analyzes the Python code that is available in the current solution and the installed libraries to populate the internal database. By doing so, we are able to gain a better understanding of the available classes, methods, and field descriptions. This is done in a way that can help the developer speed up the coding process.

The IntelliSense context-aware code completion feature can be recalled in line using the *Ctrl* + Space bar or *Ctrl* + *J* shortcuts. The *Ctrl* + *J* shortcut displays the list even when there's only one possibility. This is what happens when you call it in the middle of a command:



IntelliSense shows all the available methods, classes, and fields available in the current code, ordered in alphabetical order. Besides showing them from the list of available commands, it also provides you with a quick documentation. In our example, for the `range` method, IntelliSense shows the list of its overloaded methods and its signature.

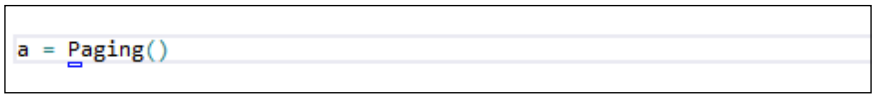
If you are aware of the method that you are searching for but want a little help with the list of parameters available, just recall the IntelliSense window either through the parameter parenthesis when you open it or with the *Ctrl* + *Shift* + Space bar shortcut.



In this case, IntelliSense will show possible combinations of the parameters and a quick documentation for each one, given by the position of the parameter in the list as shown in preceding screenshot.

The automatic importing of modules is another interesting functionality that helps in speeding up development. It also provides the functionality of automatic inclusion when you use a class from another Python file.

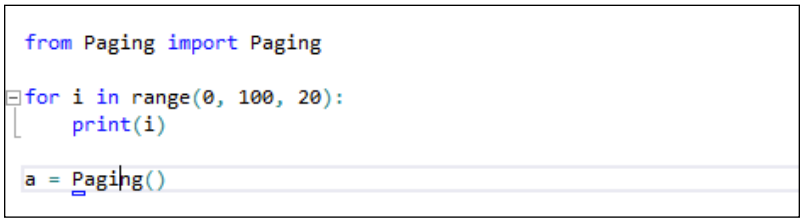
As shown in the preceding screenshot, if you have a `Paging` class in one of your Python files in your project, IntelliSense can recognize it as a class that is reachable from your project; also, it will suggest further operations by showing a smart tag under the first character of the classname:



IntelliSense in action on a classname for importing. The little blue rectangle under the classname means there are functions available from Visual Studio

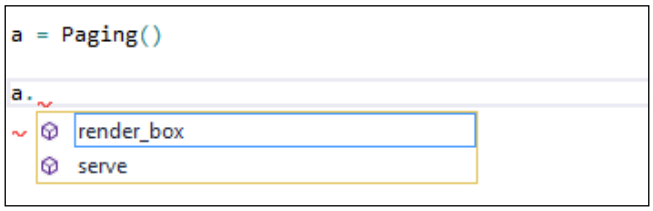
When it's clicked, an action pop up is shown. You can choose the operation to execute along with its ability to reference the source/originated file into the current code:

If you click on one of the two options, an inclusion statement will be added as the header of your code file:



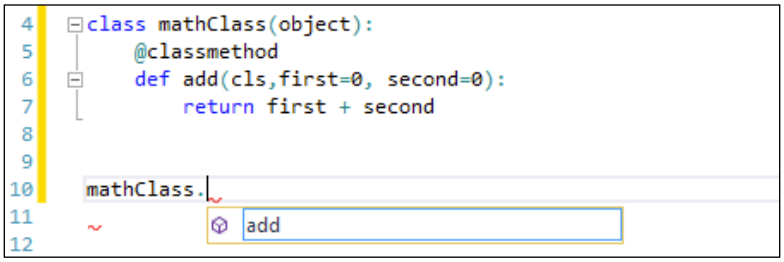
After selecting the "from Paging import Paging" option, the inclusion statement is generated at the top of the code file

Once the class is visible in your code, Visual Studio is able to inspect the referenced class. You will start seeing the class in the IntelliSense window when it's called, as shown in the following screenshot:

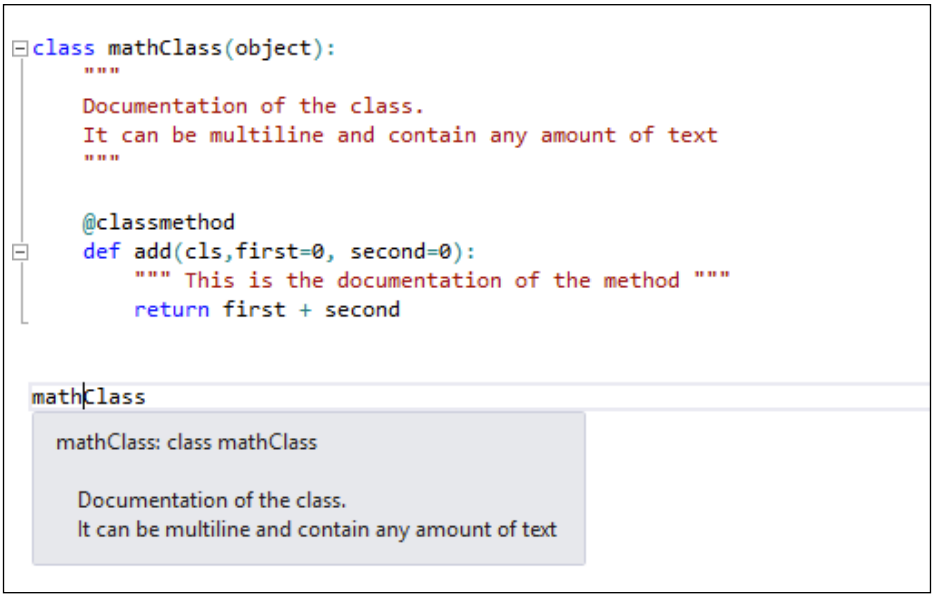


IntelliSense showing the methods available in the Paging class

IntelliSense can be extended even further. In the following example, when the `foo` class is defined with the `bar` method, IntelliSense will fetch the class structure to display the helper to be used in the code:



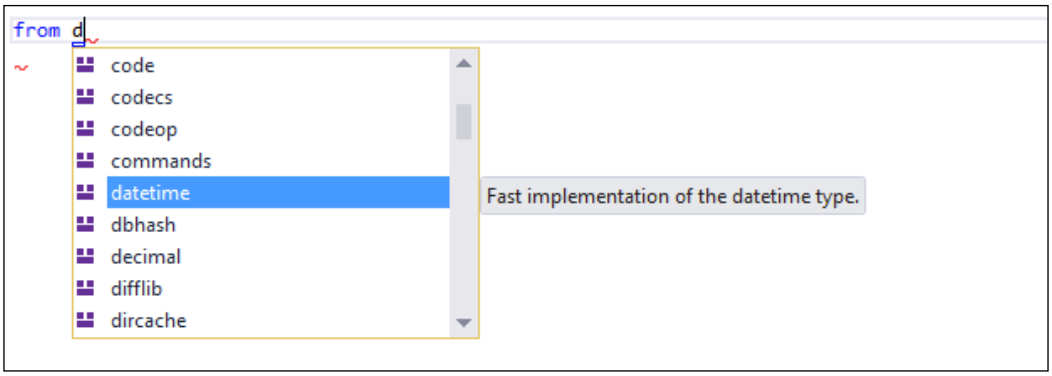
IntelliSense is able to provide us with an insight into the structure of the class and the available elements of it, but without any documentation. To have the documentation shown in the code, we can simply add it to the code of the class as follows:



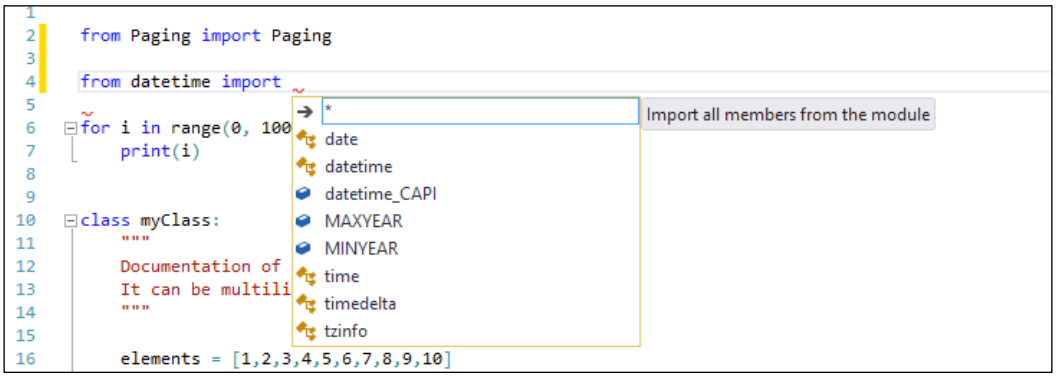
IntelliSense showing the class or method documentation

It's really straightforward and simple. Any element in the Python project is automatically analyzed and fed into the reference database of Visual Studio without having to rework the code or execute external tools during the coding session.

IntelliSense can also help when the code is referenced, giving us an overview of all the modules that are available, all the PythonPath-referenced modules, the modules you are going to reference in your solution, and the modules that are part of your project:



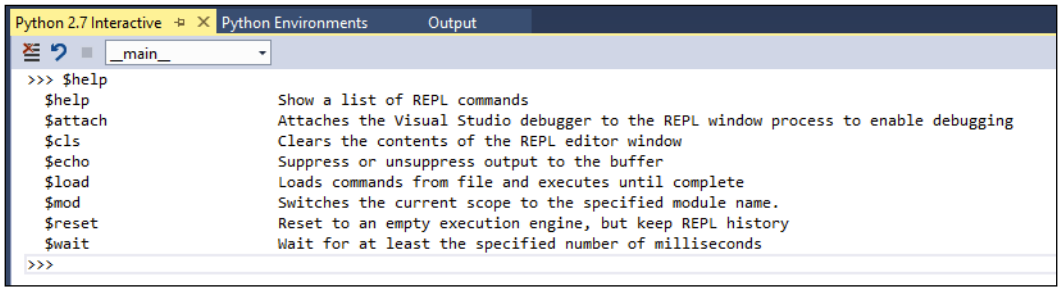
Furthermore, IntelliSense is useful not only to select the module, but also to select the import part:



Using REPL in Visual Studio

In this section, we will explore the usage of the standard **read-eval-print loop** (**REPL**) tool for Python inside Visual Studio. As mentioned in the introduction, PTVS has an enhanced version of REPL. Besides the standard Python commands in the REPL version of PTVS—the Interactive Python window—there are some added commands and functionalities that can help speed up the debugging process and also enable simple testing of your code.

The enhanced commands are shown by typing `$help` in REPL, as shown in the following screenshot:



We'll go into detail on the most used and interesting commands. The `$cls` command cleans up the command line, while the `$reset` command cleans up the engine in a way that you can restart with a clean REPL environment.

The most interesting functions are `$load` and `$mod`. The `$load` command permits you to execute the content of a given Python file inside REPL:

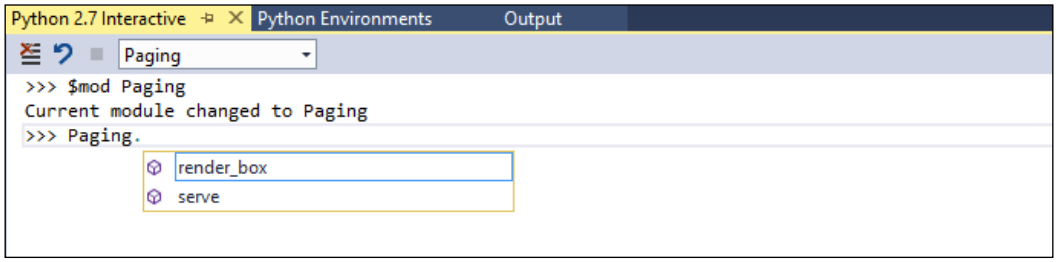
```
>>> $load C:\Users\ezu\Documents\Visual Studio 2013\Projects\PythonApplication1\PythonApplication1
\PythonApplication1.py
>>> from Paging import Paging
>>> for i in range(0, 100, 20):
...     print(i)
...
...
0
20
40
60
80
>>> class foo:
...     """
...     Documentation of the class.
...     It can be multiline and contain any amount of text
...     """
...     @classmethod
...     def bar(self, first=0, second=0):
...         """This is the documentation for the method"""
...         return first + second
...
>>> print(foo.bar())
0
>>>
```

An example of the $load function in the REPL tool

These functions are useful as they provide an on-the-fly view of the execution cycle of your code.

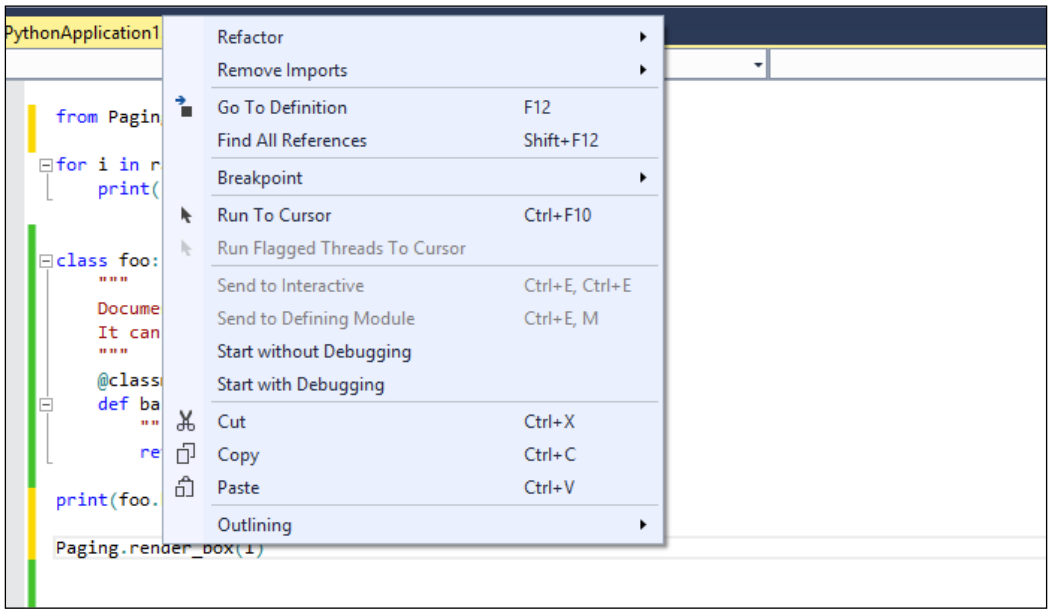
The `$mod` command gives you the opportunity to change the scope on which the REPL tool is operating.

When REPL starts, it's automatically set on the main module of the opened project, `(__main__)`. Typing `$mod` followed by the name of the module allows you to switch to another module of the same project, giving access to the module and its content.



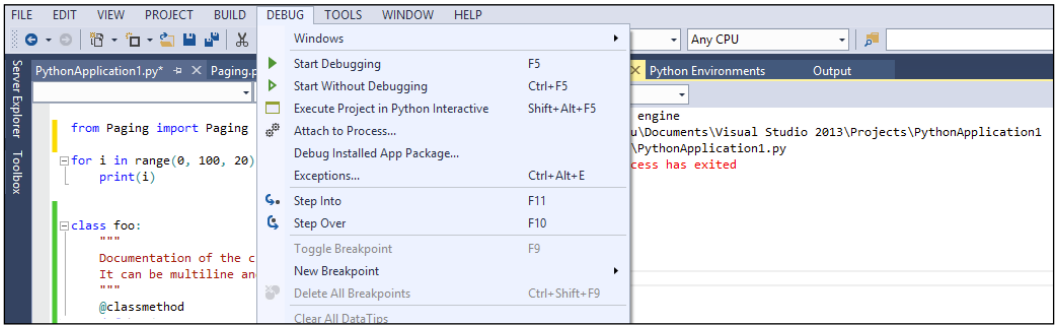
As we can see in the preceding screenshot, the user interface of the REPL window has a combobox at the top, which allows us to interactively switch the scope. IntelliSense is also managed in REPL.

We can execute every piece of code in our project using the **Send to Interactive** command in the context menu under the coding panel (or, use the *Ctrl* + *E* shortcut):



We can also instruct the debugger to use REPL as the output of the application by selecting the **Execute Project in Python Interactive** option from the **DEBUG** menu (or by using the *Shift* + *Alt* + *F5* shortcut).

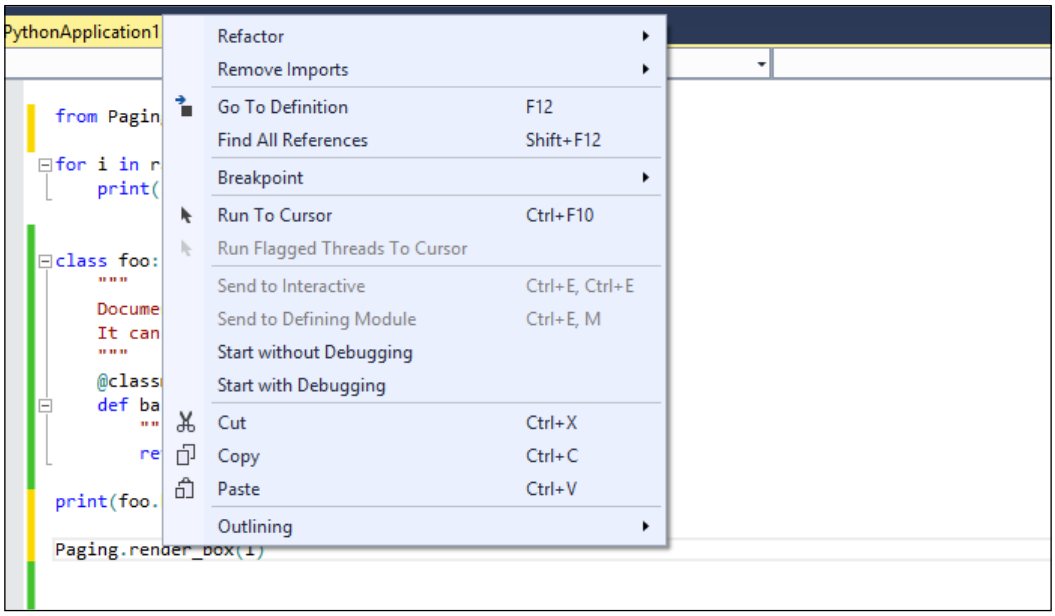
It's also possible to tell the debugger to use REPL as the output of the application instead of the standard console output; to do this, just click on the **Execute Project in Python Interactive** entry in the **DEBUG** menu or use the *Shift* + *Alt* + *F5* shortcut:



This is particularly useful when developing back-office modules in a web application for which the debugging and testing of the code is particularly difficult if you wish to do this directly in the browser. Using the REPL tool, you can achieve a much more productive and quicker process.

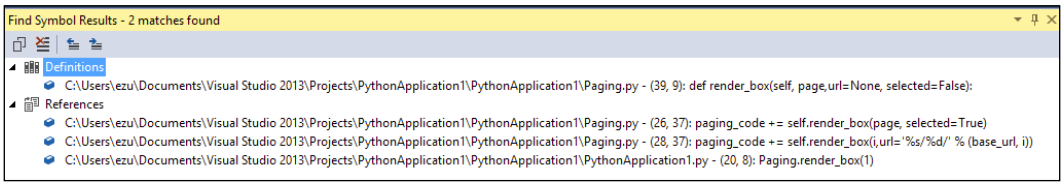
Navigating code with ease

Visual Studio provides lots of features to speed up the process of code navigation; at the same time, these features allow Visual Studio to have a streamlined coding process with a more holistic view of the project. This is particularly useful when you need to switch to a module to see the actual implementation or to update some parts of it, even when it is located somewhere else in the project. Some of the most important features for code navigation are reachable from the context menu in the coding panel:



As shown in the preceding screenshot, when the cursor is over a method and the contextual menu is opened, the command **Go To Definition** can be found. The **Go To Definition** command moves the view to the implemented code of the method inside the referenced module.

Another useful debugging function is the **Find All References** command (the *Shift* + *F12* shortcut). It shows all the points inside the project where the method is used:

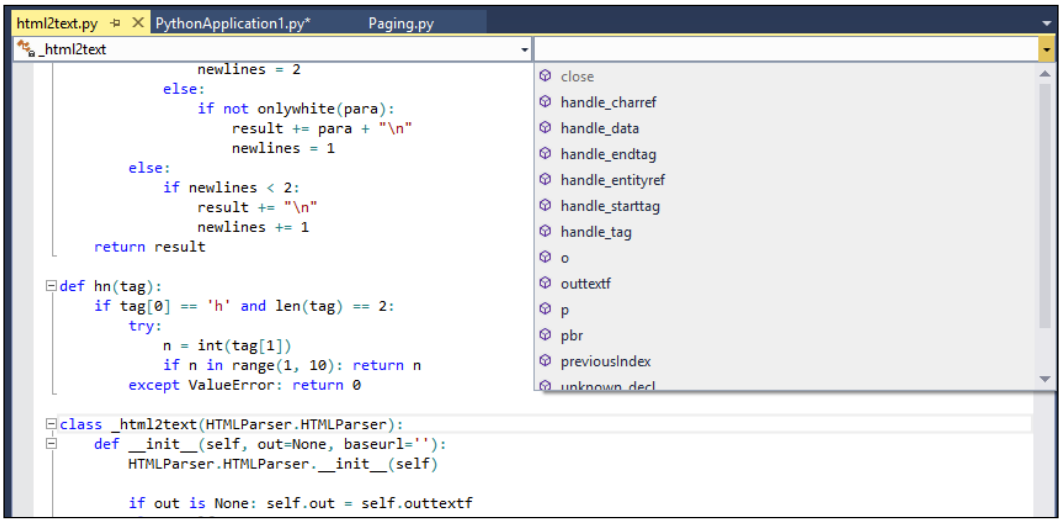


The result of the execution of the **Find All References** command will be shown in a new tool window, **Find Symbol Results**. It shows both the definition of the method and the actual references inside the project in which the method is actually used.

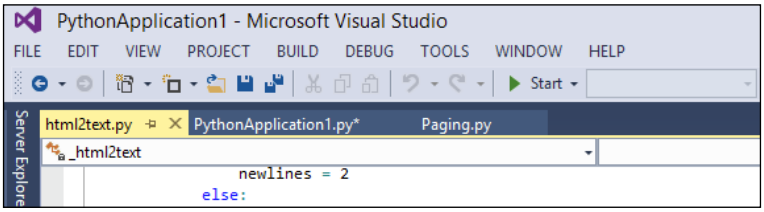
Furthermore, the code panel provides tools to quickly navigate to any given point of the code. Because of the fact that Visual Studio dynamically analyzes the code when a Python file is opened in the code editor panel, the IDE automatically creates a hierarchal index of it. To navigate the index, use the two comboboxes with the navigation bar that are located at the top of the window.

The first one shows all the global reachable elements in the file, while the second one shows all the inner elements of the item selected in the first combobox. The selection of an element in comboboxes controls the view of the code editor and jumps to the referenced code.

In the following screenshot, we can see how navigation of the comboboxes works, showing the hierarchal view by displaying all the inner components of the `html2text` class in the second combobox:

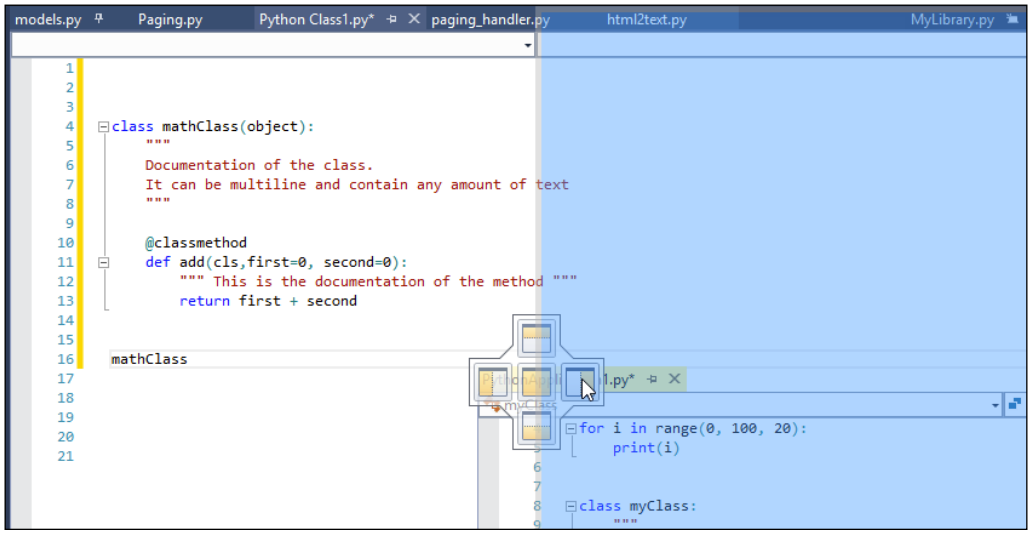


To easily go back and forward while navigating the code, especially when jumping between modules, there are two useful buttons in the toolbar that allow you to jump back to the starting point. Look for the two arrow-shaped buttons at the very left-hand side of the toolbar. The arrow pointing to the left is to move backward, which is accessible through the *Ctrl* + - shortcut; the arrow pointing to the right is used to move forward, which is accessible through the *Shift* + *Ctrl* + - shortcut.



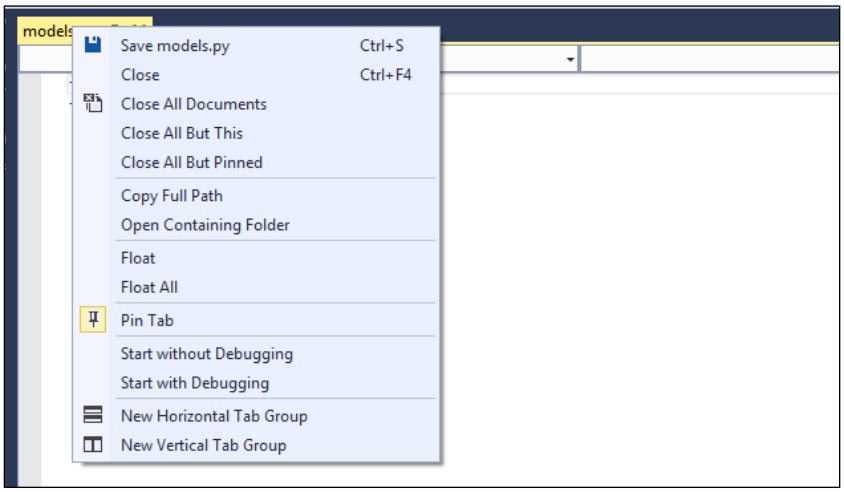
The navigation buttons in blue on the left side of the toolbar, are a good way to quickly navigate through the code

The editor window provides a way to navigate the files that are already opened through the file tabs located at the top of the window. As of the time of writing this book, Visual Studio has refined the features available for this. So now, even for PTVS, there are many possibilities to do so. You can close the tabs and move them around. A single tab can be shown in an independent window or side-by-side with another one.



Visual Studio offers a powerful window management, providing a full set of alignment option on every window in the IDE by simply dragging them from the tab

Tabs can be pinned to the leftmost position of the tab bar so they can be easily reached when many files are opened in the code editor. The tab itself has a contextual menu with more commands for the file:

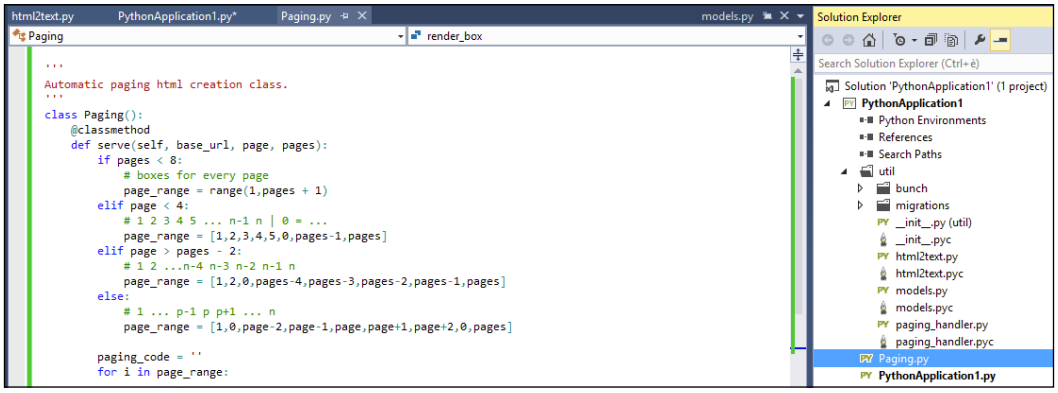


From the tab contextual menu, you are able to manage the tabs or directly execute the code inside the tabs. The **Copy Full Path** and **Open Containing Folder** options are very handy during the coding process.

It is also possible to navigate through the files of the project through the **Solution Explorer** window. Clicking on the file of interest will open the file in the code editor window. A single click opens the file in a temporary state, which means that the file will be opened in a tab at the right end of the tab bar. This is really useful when going through various files without working with them.

Unlike a permanent tab, the temporary tab will remain open until you navigate to another file through the **Solution Explorer** window. This trims down the amount of tabs open in the code editor. As we all know, it will become difficult to manage and navigate a huge number of open tabs.

Double-clicking on a file in the **Solution Explorer** window opens it in a permanent tab in the code editor, which is indicated with a pin icon:



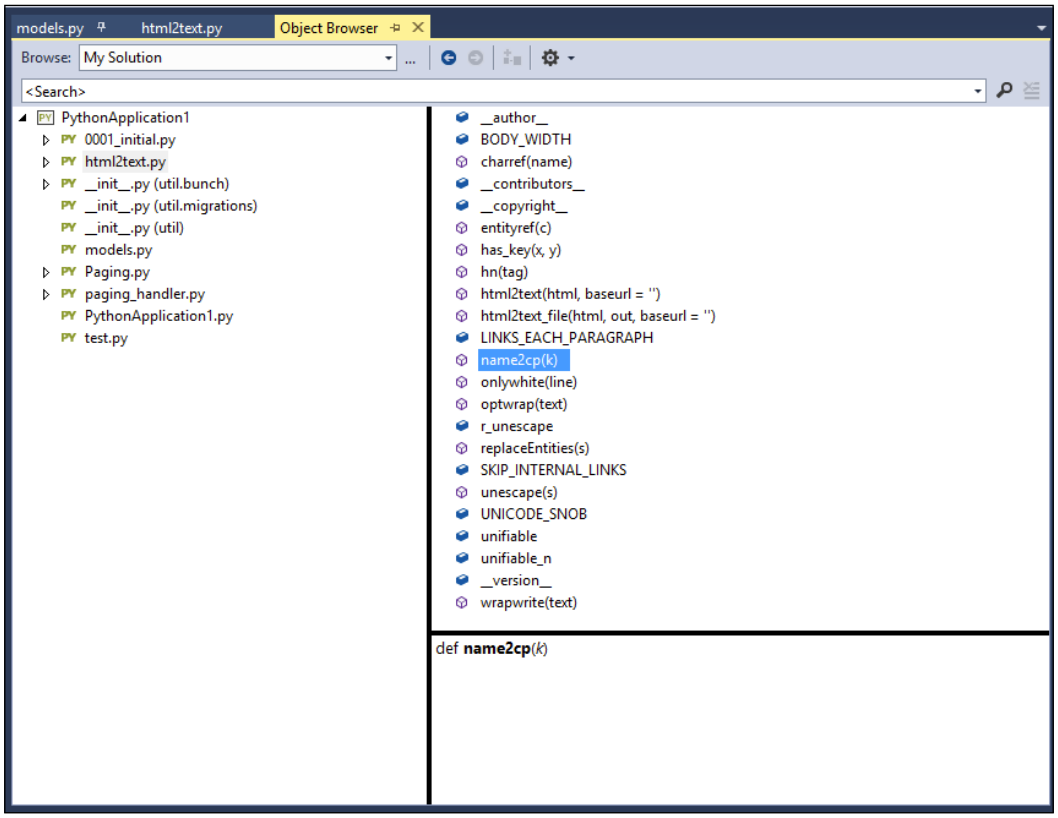
The "Solution Explorer" window, on the right, is a powerful tool to navigate through the files of the projects. Open a file in temporary tab by clicking on it. Double-click opens a permanent tab which is indicated with a pin icon

Object Browser

Another way to have a high-level clear view of the project and the elements that compose it is to use the **Object Browser** tool. This tool gives you a more hierarchal view of the entire project. There are two different ways to access the view; let's take a look at both of them.

The first way is to open the full version of the **Object Browser** tool, which is accessible from the **VIEW** menu through the main toolbar, or by using the *Alt* + *Ctrl* + *J* shortcut.

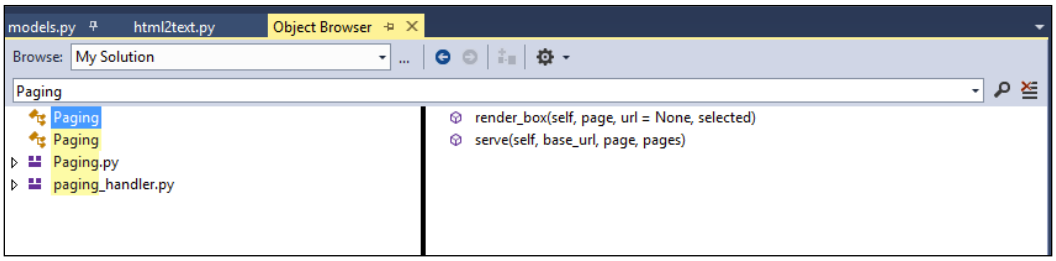
Using the shortcut will open a new tab in the code editor window, which will present us with the following view:



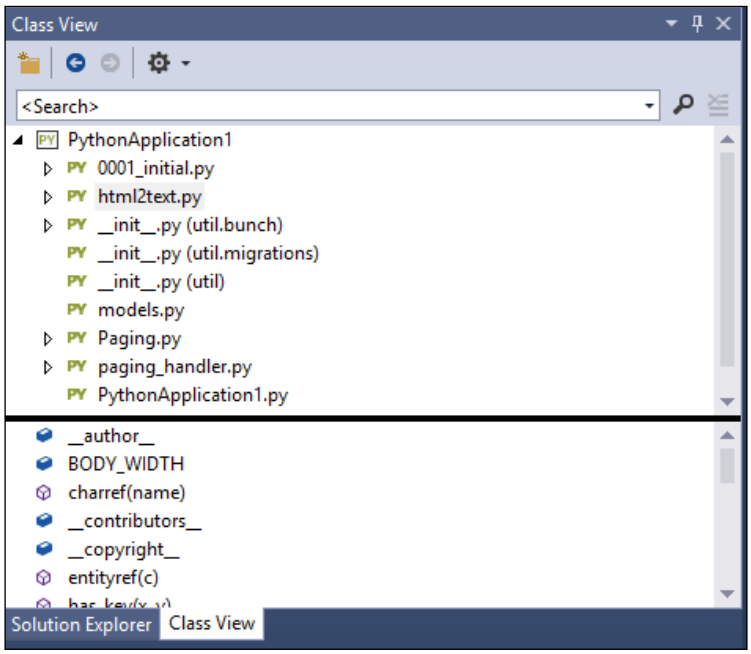
The **Object Browser** tool is a three-pane window. The top-left pane shows a list of all the Python files that compose the project. The top-right pane shows a list of elements that compose the selected file (methods, fields, and so on). The bottom pane shows the element footprint and related documentation (if available) of the selected element.

Any of the elements in the **Object Browser** tool are clickable. A double-click will open the code file and jump to the code that implements the clicked element.

It is also possible to filter the kind of elements to be shown using the **Object Browser** settings icon in the toolbar at the top of the window. It is also possible to perform an actual search using the top search bar, which will show all the references of the searched text in the underlying panels:



Besides using the full **Object Browser** tool, we can traverse the project hierarchy through the **Class View** window, which is normally located in the same window as that of **Solution Explorer**. Switching to the **Class View** tab shows the reduced **Object Browser** tool, which is practically the same but without the documentation panel. The **Class View** tab is shown as follows:



This tool gives the exact same functionalities as the full-fledged **Object Browser** tool, including the search and filter capabilities. Double-clicking on the element will jump to the code at the exact point of the implementation as well.

Summary

In this chapter, we introduced a wide range of usage of IntelliSense with Python, including working with classes between project files with simple importing and referencing capabilities. We also learned how to use REPL for a more streamlined debugging and testing process.

Now you are familiarized with the different code navigation functionalities that can help you find code references using filtering as well as tabs. Combined with the **Object Browser** tool, you are able to view your Python project with a high-level view of all methods, classes, fields, elements, related documents, and so on.

In the next chapter, we will dive into the day-to-day coding tools to guide you through the whole programming lifecycle.

3
Day-to-day Coding Tools

In this chapter, we will go through the coding tools that are essential during a normal day of work for a Python programmer in Visual Studio.

First we will analyze how to handle projects and solutions in Visual Studio, and then we will go through the refactoring of functionalities. Finally, we will go through the debugger functions that are available.

Project handling

One of the most important and useful features of Visual Studio is the solution and project handling. Since the whole workflow is integrated into the IDE, the developer does not have the burden of dealing with files, working paths, and libraries. All of these can be managed directly in the IDE with the powerful Visual Studio user interface.

Before we dig into the tools in detail, we will first take a look at the Visual Studio lingo relating to project handling. The two main concepts used in this chapter are the solution and the project.

Solution

A solution is essentially a container of projects that are bundled together to cover a unique scope. The projects can be referenced to each other and they can be of different types. For example, in a solution, you can mix a Python project with a C++ project while referencing the output of the project in the Python solution to use it as an external library. A solution also provides a way to group the whole code base of work in a single file/folder structure. You can then insert and manage the grouping in the versioning tool of your choice to share it quickly. A Visual Studio solution is also capable of maintaining shared configurations for the inner projects, while handing different commands and operations during different events (i.e. during the build of the solution).

Project

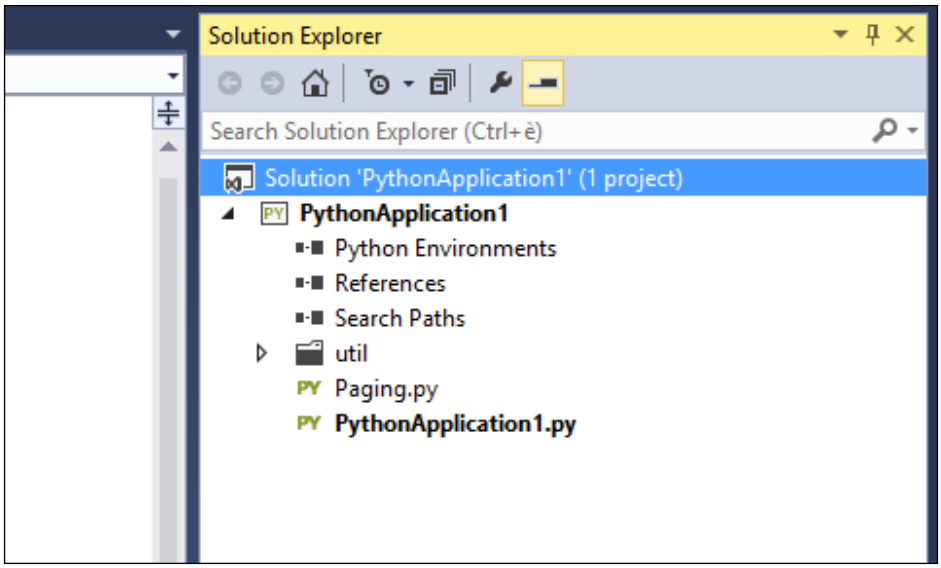
A project is the classical definition of a bunch of files written in the same language and which covers a single scope. The types of projects can range from a website to a library or a console application.

Visual Studio projects for Python contain the environment definition: where to target the code, the references to external libraries, and the search paths that the compiler has to search in the libraries. The last one is particularly important since PTVS does not use the computer's PythonPath environment variable.

The deliberate and useful feature of ignoring system-wide settings allows you to reference different libraries in different projects for different Python versions. Furthermore, the dependency list in the code brings the added bonus of an easier debugging process and also provides an easy setup of a new development environment on other computers.

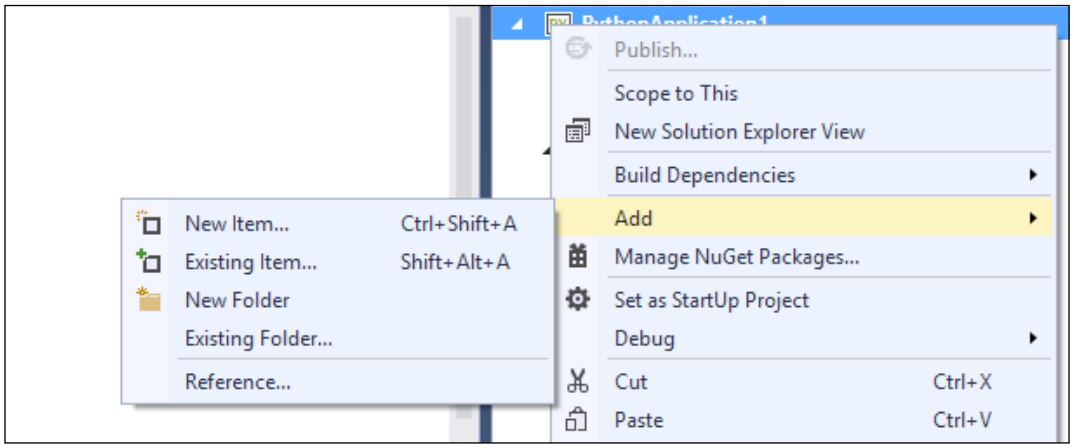
PTVS offers preconfigured Python projects called project templates, as we have seen in *Chapter 1*, *Introduction to PTVS*, which take care of creating the right project structure so that the developer can focus on the code.

Let's take a look at the actual tools that will handle solutions and projects. The most important and powerful one is the **Solution Explorer** window tool. This tool gives a complete view of the solution composition and the files and configurations available in each project.

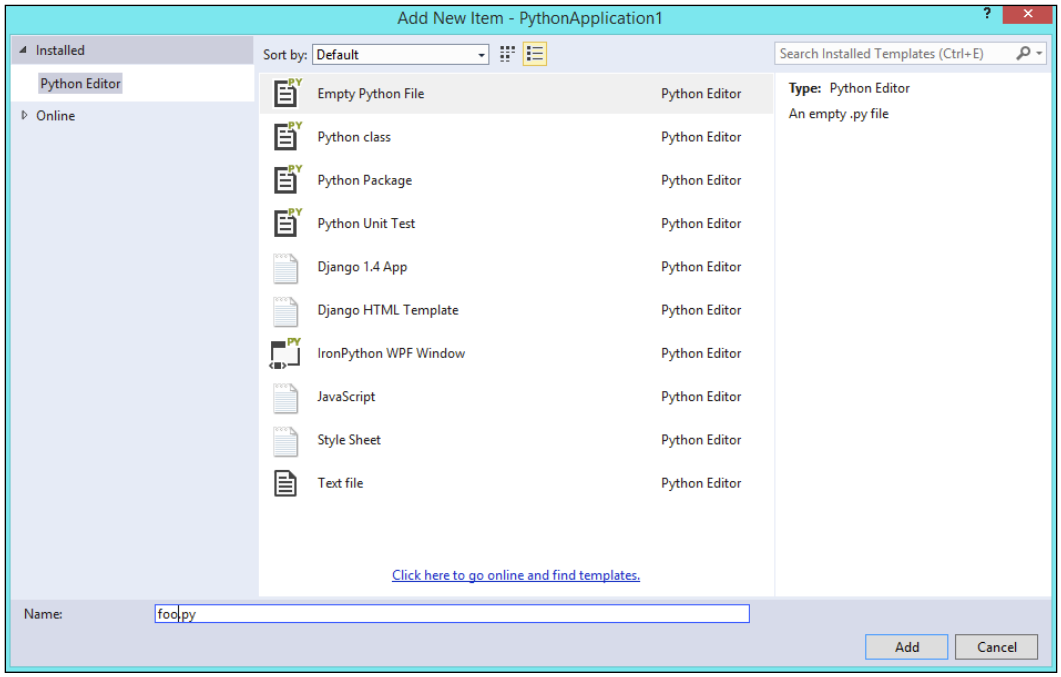


To add something or to perform actions on a Solution or a Project, select the Solution or Project node in the tree-view

By now, you should already know how to navigate through the code using this tool, as we learned in *Chapter 2*, *Python Tools in Visual Studio*. The **Solution Explorer** window tool also provides file handling capabilities, giving us the ability to add or remove files directly in the project structure. Just select a folder item in the project structure to insert a new file. To add something in the project root, select the project item in **Solution Explorer**, right-click to open the contextual menu, and go into the **Add** submenu as shown in the following screenshot:



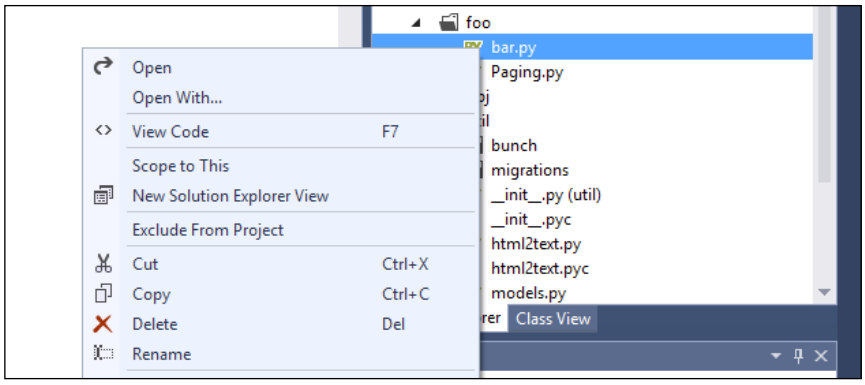
Here, you can choose to add either **New Item**, **Existing Item**, or **New Folder**. If you choose to add **New Item**, the **Add New Item** window will show up as follows:



From here, you can create a new item from the various types that are available in the project.

If instead you want to add an existing file to the selected folder, select the **Add existing item** option from the contextual add menu. This will open a standard Windows browse file window; from here, you can navigate through the filesystem and select the file that you want to add to the selected folder. A copy of that file will be added to the folder.

To delete a file, just select the item in the project list and select the **Delete** menu item from the contextual menu.

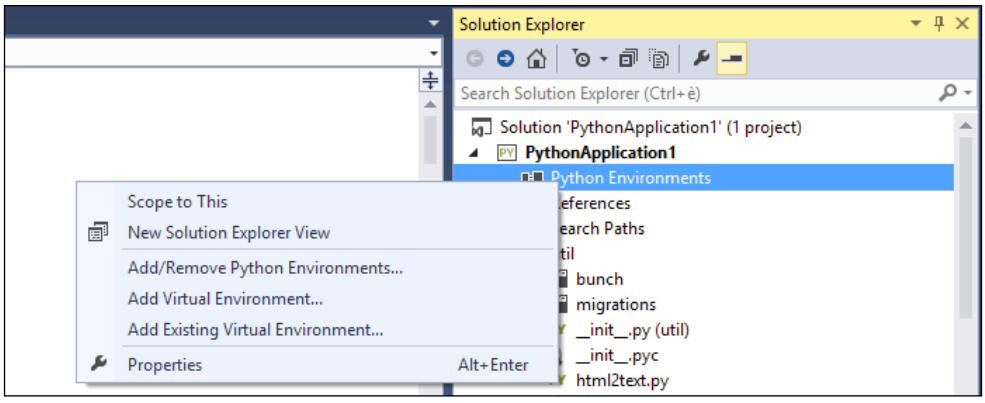


The file contextual menu in the Solution Explorer window offers lots of functions on file, like "Delete", "Rename" and others

Now that we have more confidence in **Solution Explorer**, let's dig a little deeper into the Python-specific options that **Solution Explorer** offers. In the following sections, you will learn more about the configuration of a Python project: the environment, the references, and the search path.

Specifying Python environments

It's possible to specifically define a Python environment version for a project instead of using the default Python version installed on the machine. This is particularly important for projects that we work on with other developers. By default, Visual Studio uses the default Python version installed on the machine when starting a new project. To link a project to a given Python version (environment), right-click on the **Python Environment** item in **Solution Explorer** to see the contextual menu. This is shown in the following screenshot:

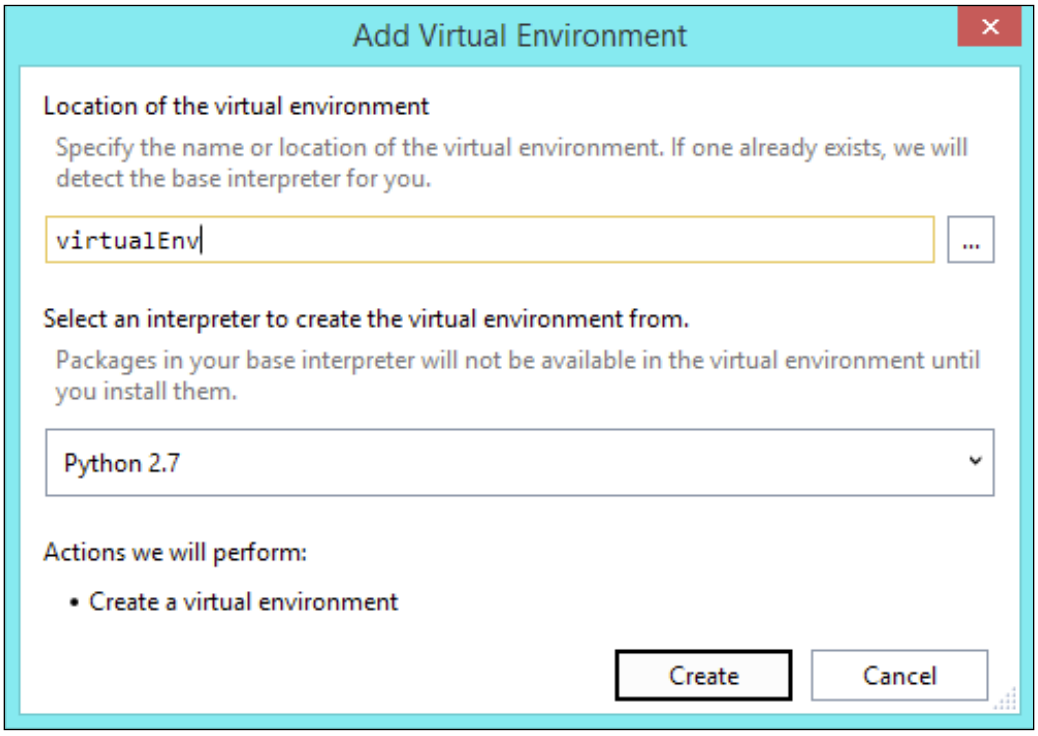


The Python Environments contextual menu in the Solution Explorer window

This contextual menu provides various functionalities such as adding or removing a Python environment and linking the project to either a virtual Python environment or an existing one. The last two options are very useful when you need to have a project running in a completely isolated environment space on the machine. The project can then be run with all of its dependencies and libraries in an isolated place, without interfering with the existing Python installations and Python path configurations on the machine.

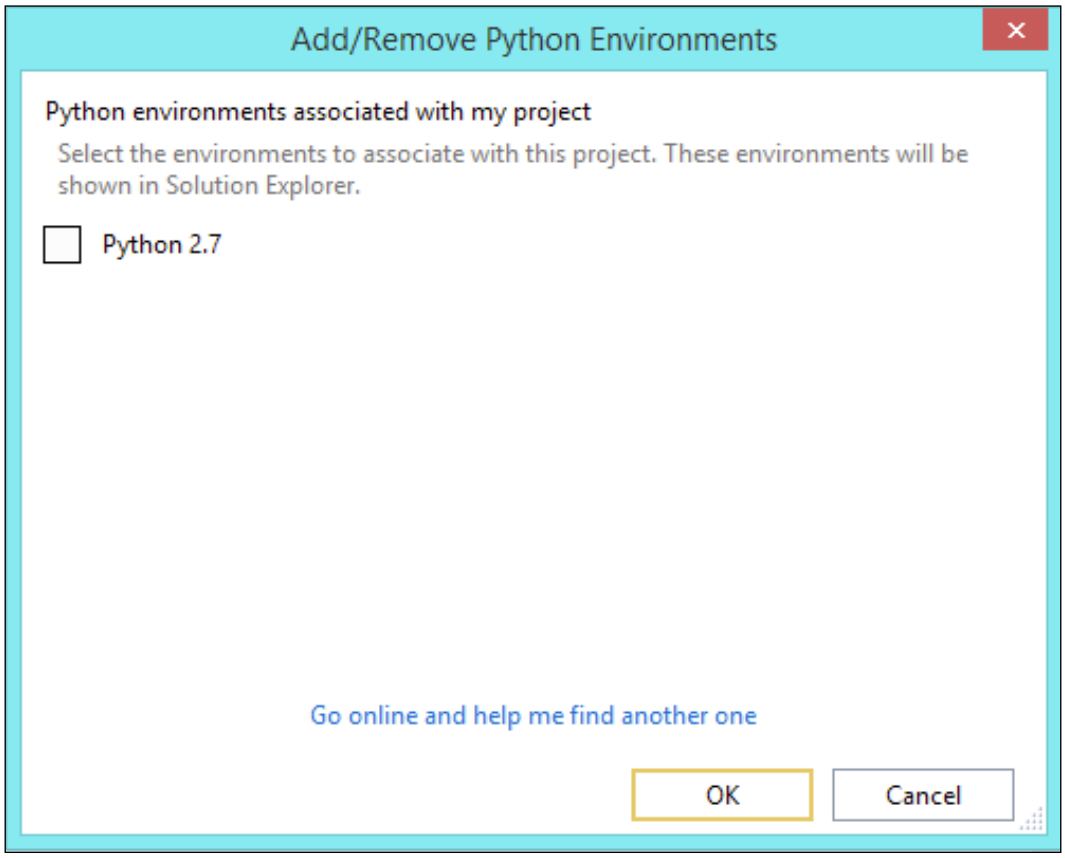
Creating a virtual environment in Visual Studio is straightforward. Click on either **Add Virtual Environment** or **Add Existing Virtual Environment** and follow a few steps to complete the setup.

As an example, we will create a virtual environment for our project. Clicking on **Add Virtual Environment** will show the following modal window:

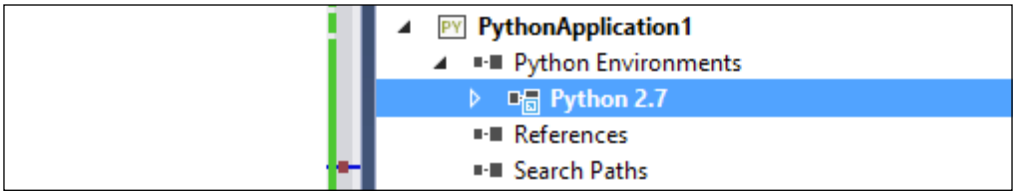


You can define the name of the virtual environment and the targeted Python version in this window. Once the **Create** button is pressed, Visual Studio will create the virtual environment. As a nifty bonus, if the necessary Python libraries are not installed on the machine—essentially `pip`, `setuptools`, and `virtualEnv`—PTVS will take care of this by downloading and installing them. Like other generic Python packages, they will be installed in the system-defined `site-packages` folder.

To link a project to a given Python version, just click on the **Add Python Environment** option in the contextual menu and the following helper box will show up:

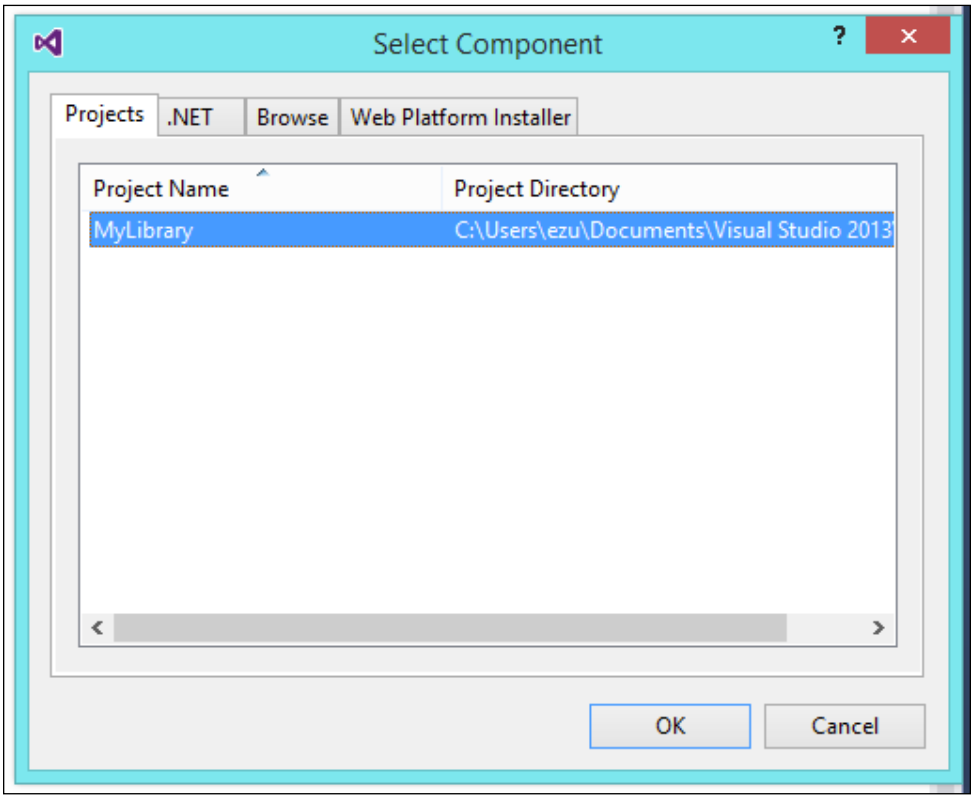


This helper box shows a list of the Python versions installed on the machine and which are available for you to choose. Once a Python version is selected, the reference will show up in **Solution Explorer**.

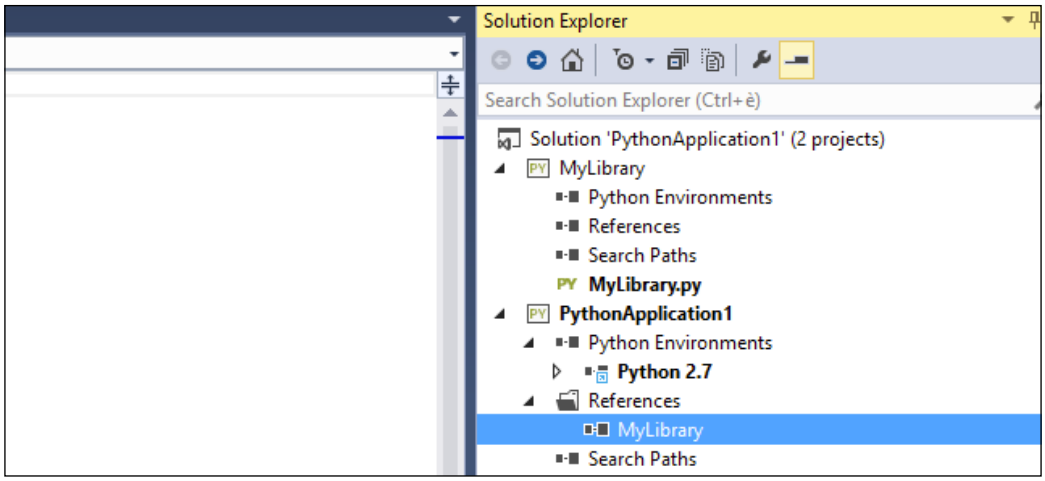


The **References** item elements in a project provide the ability to tightly link a library in your project or reference to packages compiled in the .pyd files

Right-click on the **Reference** option and then click on **Add Reference** to bring up the following helper window:

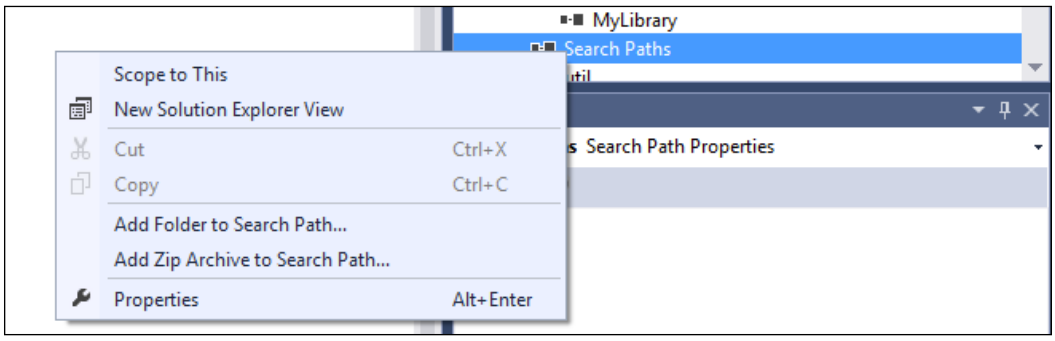


We will focus on the **Projects** tab, which shows all the other projects in your solution. If you wish to use another library project to handle a subscope of the application, select the desired project and click on **OK**. This creates a reference in your project. You can find the list of references under the reference option in the **Solution Explorer** window:



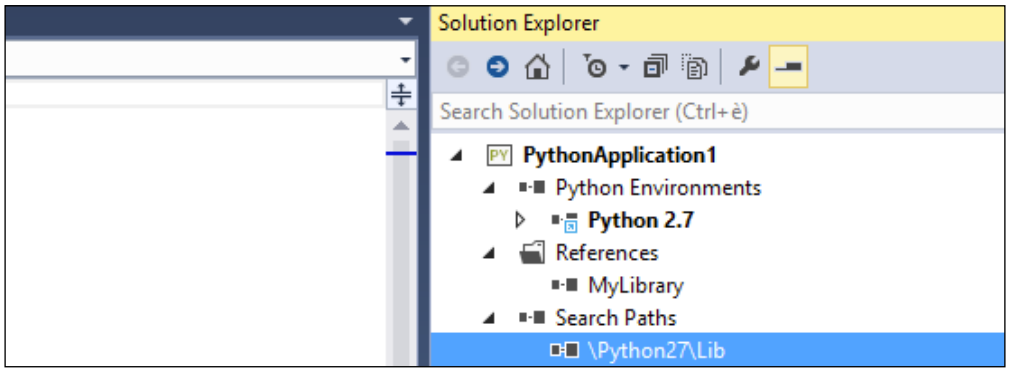
Defining Search Paths

The **Search Paths** functionality basically tells Visual Studio where to search for additional libraries that will be used in the project. You can reference a folder in the system or a `.zip` file that contains the libraries:



The Search Path contextual menu, with the "Add Folder" and "Add Zip Archive" options

Once the folder of the `.zip` file is selected, you can find the libraries in the window:



Refactoring

Refactoring is one of the biggest advancements in modern IDEs. It significantly cuts down on time and reduces the margin of error by the way in which it handles changes in the code and automated operations. Visual Studio comes with great out-of-the-box refactoring functionalities such as renaming and the creating method from a selected piece of code.

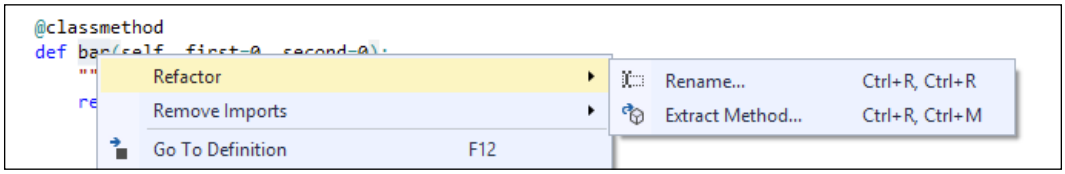
The renaming functionality can really help with potential errors in code, such as when changing the name of an element. There might be instances in the codebase where the old name is still used. Let's have a look at the following code:

```
class foo:
    """
    Documentation of the class.
    It can be multiline and contain any amount of text
    """
    @classmethod
    def bar(self, first=0, second=0):
        """This is the documentation for the method"""
        return first + second

print(foo.bar())
```

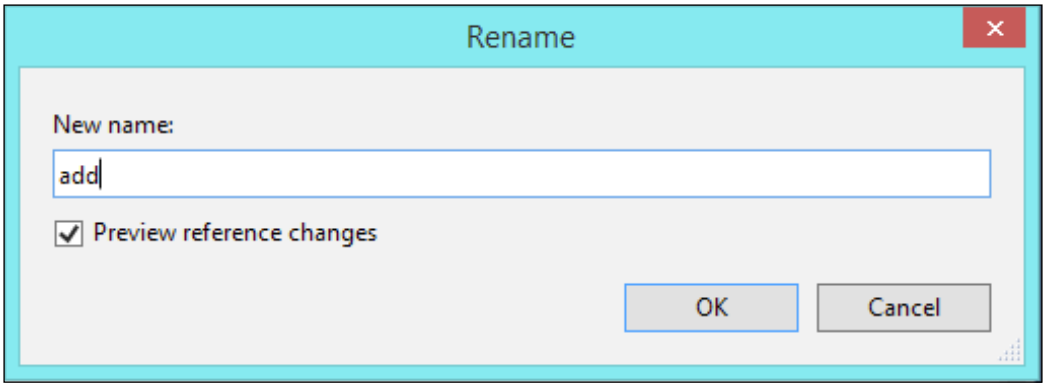
In this code, there's a class, `foo`, that has a method called `bar`. If `bar` is renamed, it will create an error by referencing to a nonexistent method.

Visual Studio's refactoring functionality helps the renaming process by taking all the references of the element into account. Select the element that you wish to demand and then access the refactoring function in the code contextual menu by right-clicking on it:

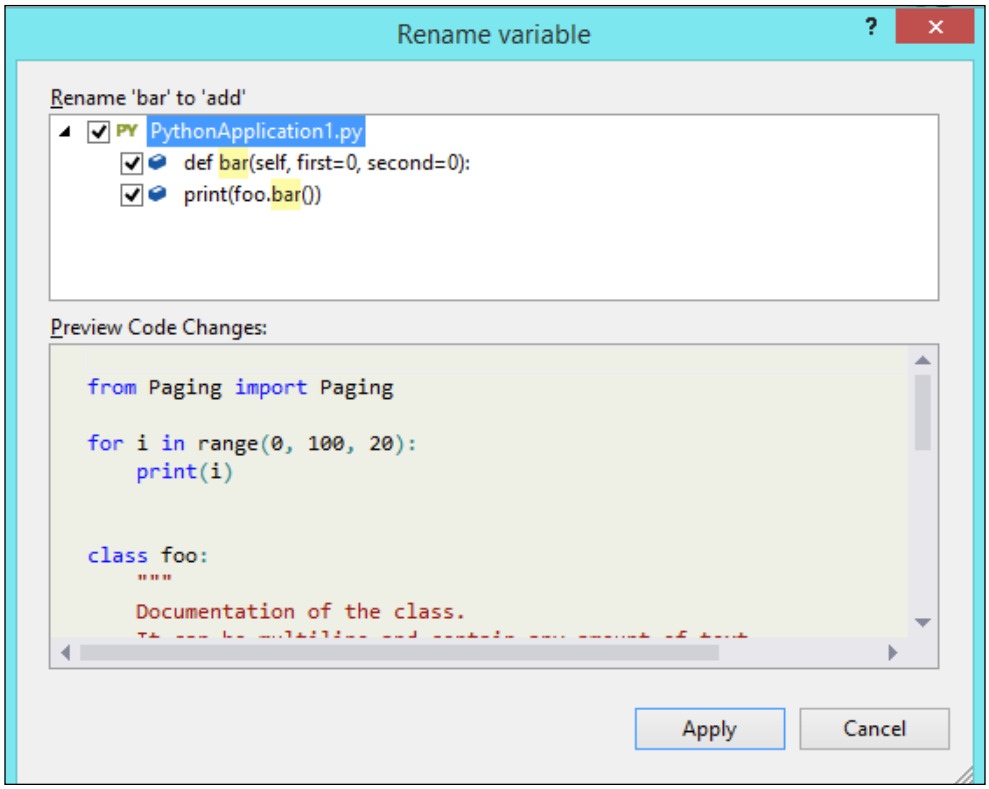


Access refactoring function by right clicking on the element

Select the **Rename** function to start the two-step process:



You are prompted to enter a new name for the element. There's also a checkbox that permits you to preview the references of the element to be renamed. When the checkbox is unchecked, all the references that are found will be renamed; you will not be able to preview which references are going to be renamed.

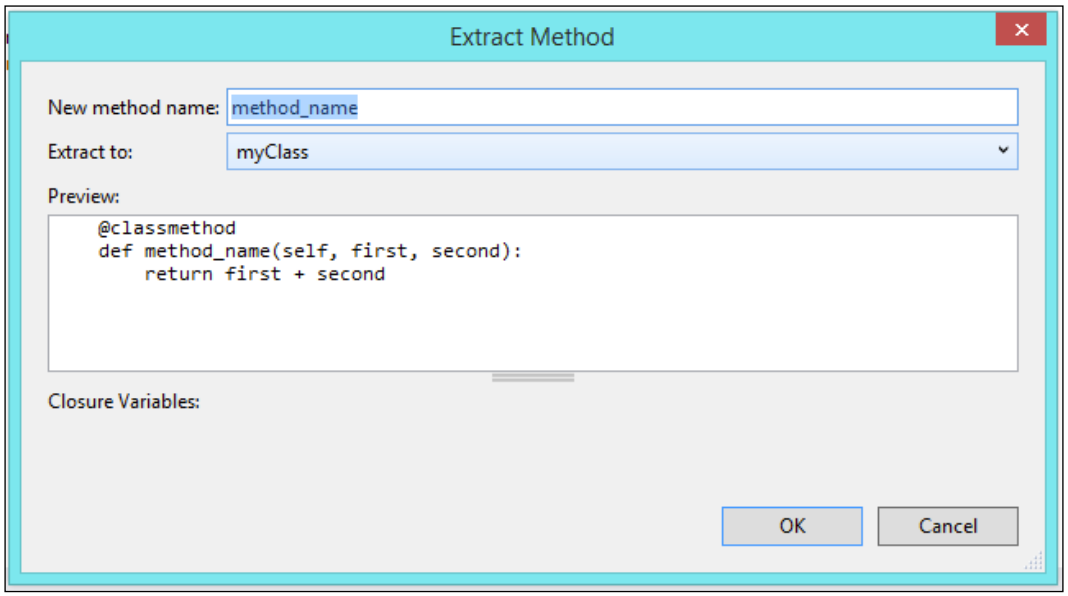


In a situation where you wish to preview the reference change, a preview window will be shown. As shown in the preceding screenshot, in this window all references of the old element name can be found at the top panel and the code preview will be at the bottom. All the files of the project will be analyzed to refactor the element correctly. A checkbox near each reference provides the option to activate the refactoring of that reference. Clicking on the **Apply** button will rename the element and all the selected references.

The other refactoring function is **Extract Method**. This comes in handy when you wish to reuse a piece of code somewhere else as a function or a method. Visual Studio can generate it as a function/method. As an example, refer to the following screenshot:

```
@classmethod
def doSomething(self, first=0, second=0):
    """This is the documentation for the method"""
    return first + second
```

In the code, the highlighted code aims to create a generic method that calculates the sum of two elements. Select the code and then select **Extract Method** from the **Refactor** submenu in the contextual menu. This brings up the **Extract Method** dialog box, which is shown in the following screenshot:

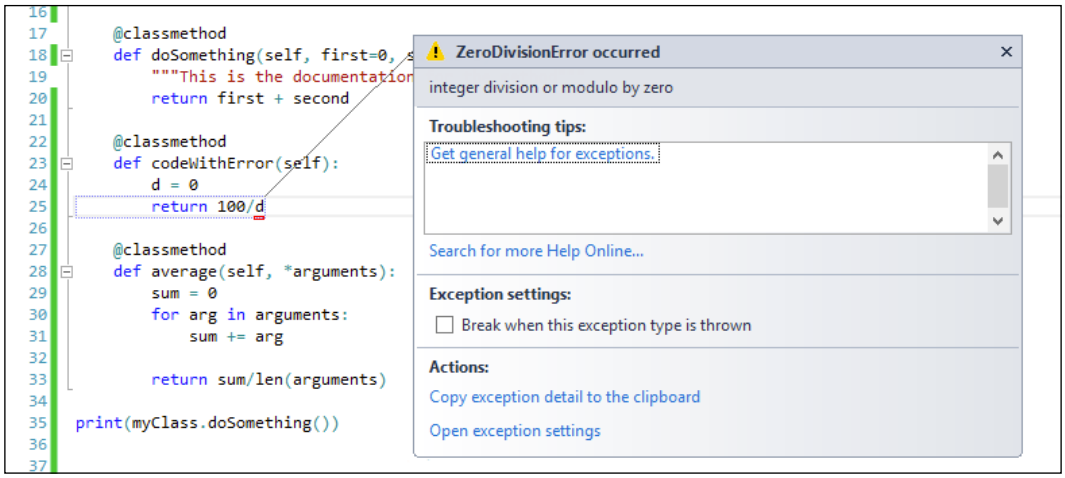


In the **Extract Method** window, you can define the name of the method in the **New method name** field and provide a path for the method to be created in the **Extract to** field. A **Preview** panel also shows the generated code. Click on **OK** to create the new method based on the selected code.

Debugging

Visual Studio offers a large set of debugging tools; PTVS inherits a lot of them, which helps Python developers to debug code by using step-by-step execution, runtime variable watch capabilities, breakpoints, and the ability to see where the code fails during a debugging session.

The ability to see where the code breaks can significantly speed up the debugging session. In the following screenshot, we can see an example of an untracked exception:



An example of error dialog box during debugging a Python application

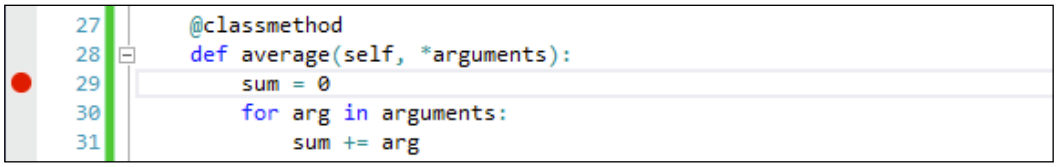
When you run the code, Visual Studio will stop the execution because it detects the raised exception of a problematic code. It highlights the exact point where the error occurred while also suggesting ways to fix it—even if right now Visual Studio may not suggest useful solutions for the problem.

The debugging process is not only about understanding where exceptions are raised, but also to understand what happens in the code when it is not behaving as expected. This is where step-by-step execution and breakpoints come in handy.

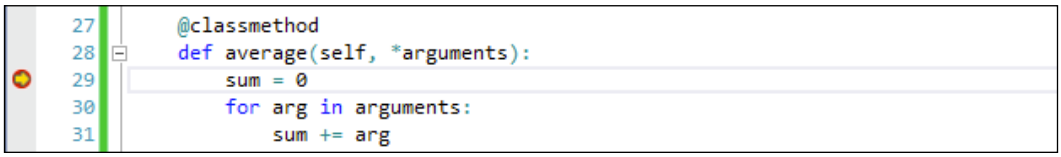
Using breakpoints

A breakpoint is a point that you can define in the code to stop the execution. Visual Studio has made it very simple to set a breakpoint. It allows better visibility of the content of variables and it follows the flow of the code. A breakpoint can be set by clicking in the gutter of the code window, which will bring up a red circle. Select **Insert Breakpoint** in the **Breakpoint** submenu in the code contextual menu.

Once the breakpoint is set, you can see it in the code window as shown in the following screenshot:



Now that the breakpoint is set, if you run the application, Visual Studio will stop its execution precisely at the breakpoint while following the flow of the code:

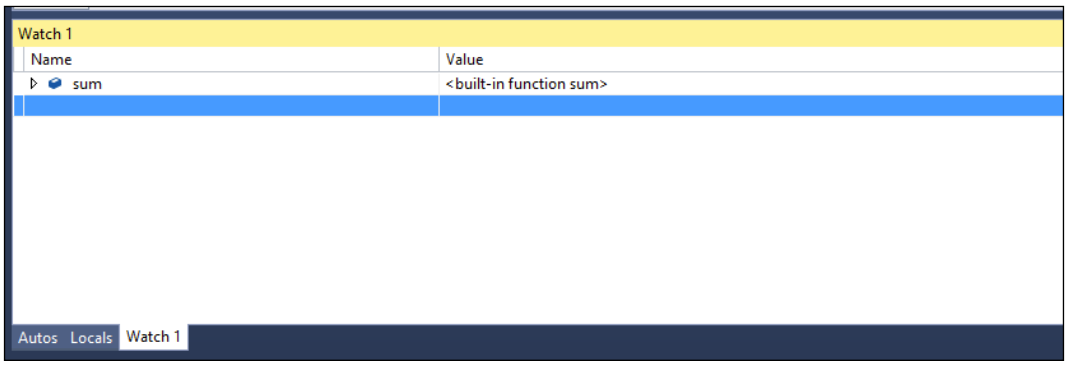


Indicated that line 29 as the breakpoint

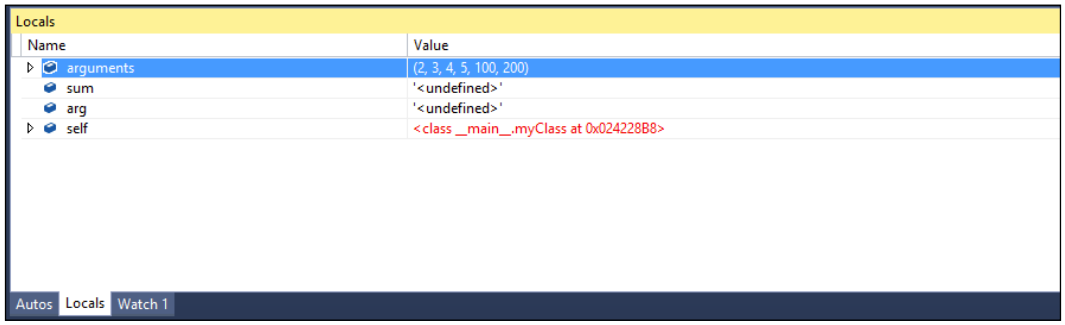
The IDE puts the caret on the first column of the line of code in which we set the breakpoint. When hovering around the variables in that context, we can see the current value of the variable.

Utilizing watch entries

We can also create a watch entry on a variable in order to see how the value of a variable changes during the program flow. To watch a variable, right-click on it during the debugging process and click on **Add Watch** in the contextual menu. The variable will be added into the **Watch** window as shown in the following screenshot:



Besides the `watch` variable, it is also possible to see all the variables in the current scope from the **Locals** tab:



Once a breakpoint has been hit, it's possible to use one of the following three functions to move on in the program flow: `Step Into`, `Step Over`, and `Step Out`. These functions are accessible through the **Debug** menu or the buttons available in the toolbar. Alternatively, you can also use `Run to Cursor` (*Ctrl* + *F10*) to run through the program until you reach where the cursor is:

- `Step Into`: This executes the next statement and stops. If the next statement is a call to a function, the debugger will stop at the first line of the function being called entering the function.
- `Step Over`: This executes the next statement. However, if the next statement is a function, calling it will not go into it. It's useful when you are not willing to follow the entire program flow of the function.
- `Step Out`: This executes the code until the end of the current function. It's useful when you do not wish to go through the entire program flow of the current function.

If you wish to just continue the execution of the program flow without going into a single line of code at the time, just press the **Continue** button in the toolbar or *F5*. If there are other breakpoints in your code, the execution will continue through all of them until the last one.

Summary

In this chapter, we introduced the tools for day-to-day coding. You are now familiar with browsing through the code with **Solution Explorer** and the flexible setting of Python environments. You also learned about the more efficient refactoring and debugging process and that setting up breakpoints and watching entries helps you trace exactly where the code breaks.

In the next chapter, we will explore how to harness the powerful Visual Studio IDE and the tools available to speed up Django development.

4
Django in PTVS

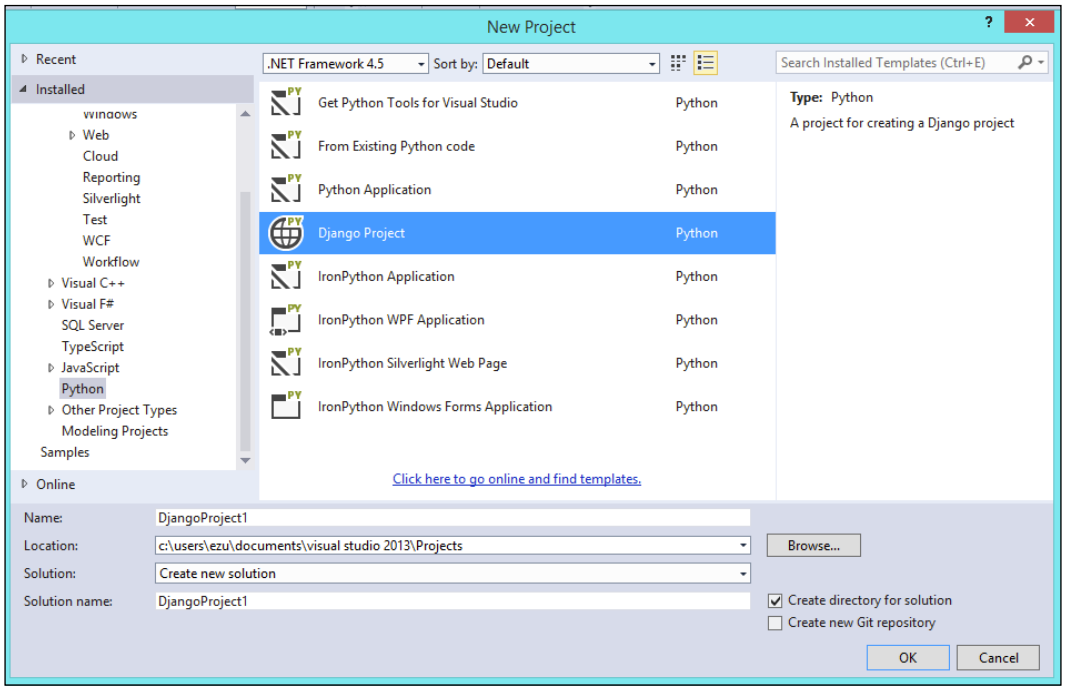
Django is a high-level Python Web framework based on the **Model View Controller** (**MVC**) pattern; it provides a series of tools and helpers to create a rapid development environment for the Web. There are plenty of successful websites that are based on Django, such as Instagram, Pinterest, Disqus, and some parts of Dropbox. It has been in development since 2006, making it a rock-solid choice for web projects, especially when using Python as the language of choice. For more information about Django, refer to its official project website at `https://www.djangoproject.com/`.

In this chapter, we will go deeper into Django framework integration in Visual Studio. We will see how to start a Django project, taking advantage of Visual Studio tools and setting up the development environment for it.

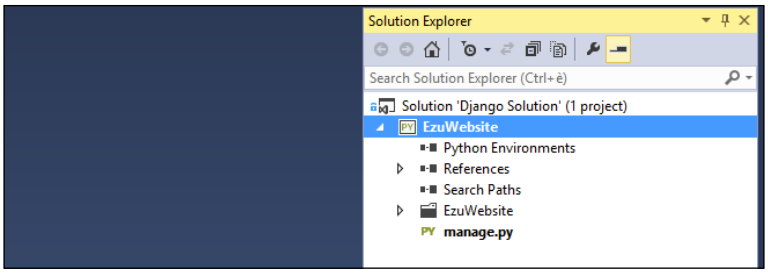
Once we have a basic working website, we will learn how to create a connection to the database and set up the admin interface to create a new Django application within Visual Studio.

Django project template and tools

Let's take a look at Django-related tools and templates available in PTVS, starting with the Django project template. Project templates in Visual Studio are boilerplate helpers that create a project's outline based on the specific type of the project. To start a new Django project, open the **New Project** window under the **File** menu. Once it's open, select **Python** from the list on the left-hand side. This displays the installed project types available in the system. Here is what the window looks like:



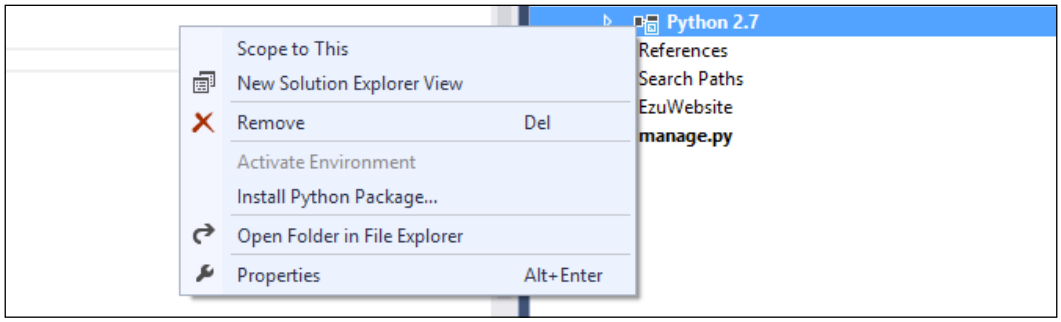
Click on the **Django Project** option in the right-hand panel to define the project name in the bottom of the window. We perform this action while specifying where the file should be saved. You can also define the solution name for the Visual Studio solution for the project. Once these properties are defined, clicking on **OK** creates the Django project.



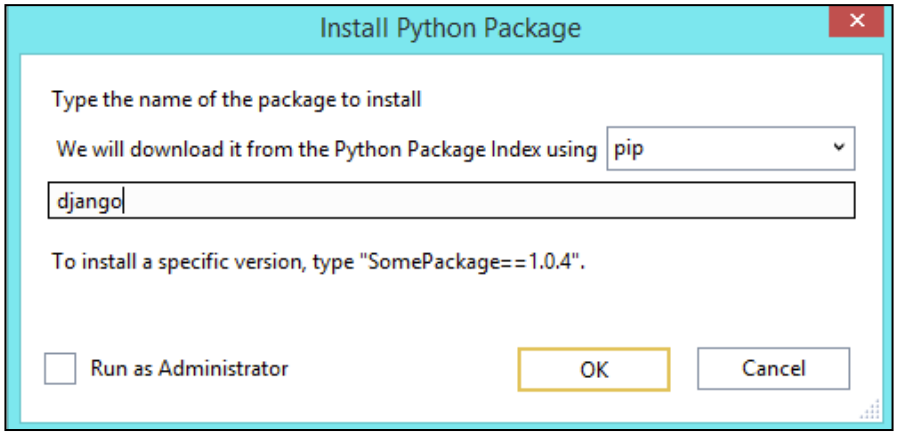
Example default Django project structure shown in Solution Explorer

Installing a Python package

The basic Django project file structure is shown in the **Solution Explorer** window. With the project structure in place, the next step is to install the Python package required. We have to instruct our solution about either the Django framework's whereabouts or the location where we need to install it. To begin, add a Python interpreter to the project—we are going to use Python 2.7—then install the Django packages by right-clicking on the installed **Python** entry in the **Python Environments** node of the **Solution Explorer** window, and finally select the **Install Python Package** entry:

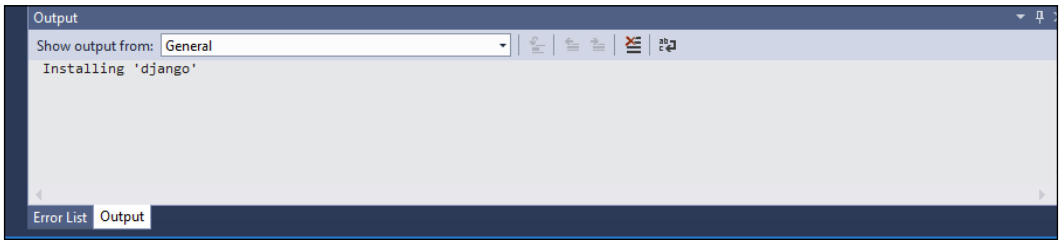


The **Install Python Package** window will open; here, insert the name of the `pip` Python package that is required to start referencing the framework in our environment, which is `django`. You can even choose a specific version of the framework by naming the version of the package in the `django==x.x.x` format. The `x.x.x` indicates the complete version label. If no version number is supplied, the latest available version of the package will be installed.

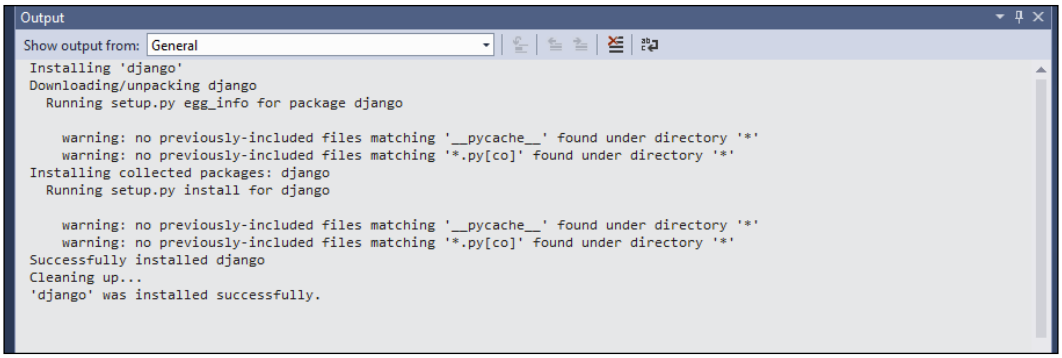


Click on **OK** to begin the Django package installation by downloading it from the `pip` repository. Make sure to check your system settings on the system agreement to give Visual Studio the privileges of a system administrator.

The **Run as Administrator** checkbox should only be checked when the previous installation has failed. It ensures that `pip` has the necessary privileges on the system to install the package.



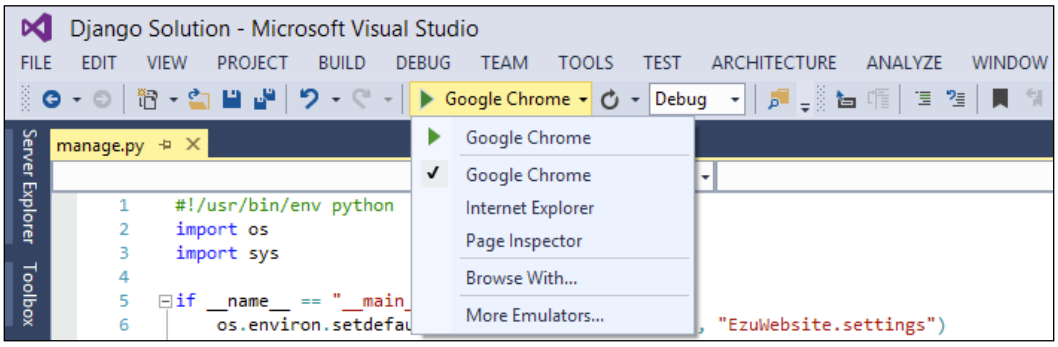
The installation process can be viewed in the output window as shown in preceding screenshot. If you are using **Run as Administrator**, the real-time download progress is not available. In this scenario, **Installing 'django'** is displayed in the output window for a period of time, as this indicates that `pip` is still downloading.



Once the downloading process finishes, a more detailed view is available as shown in preceding screenshot. It'll keep you informed about the installation process and whether it is successful. If everything happens without a hitch, a **'django' was installed successfully** message is shown at the end of the `pip` log in the **Output** window.

Running the application

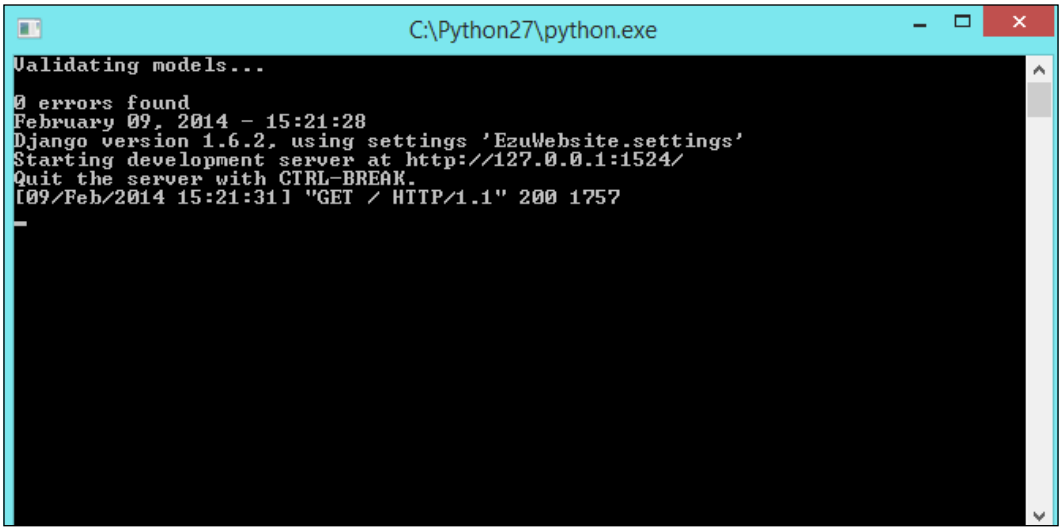
Now that the Django framework is successfully installed in the system and is referenced in the project, let's make sure it works and all the workflows are running correctly. Run the application by pressing *F5* or by clicking on the **Run** icon in the toolbar. Since Visual Studio understands that the current project is a website, it will run the output in a browser instance. You can see the **Run** button followed by the name of the default system browser that will be started. It's possible to select the browser in which we'll run the project by clicking on the drop-down button that shows all possible choices:



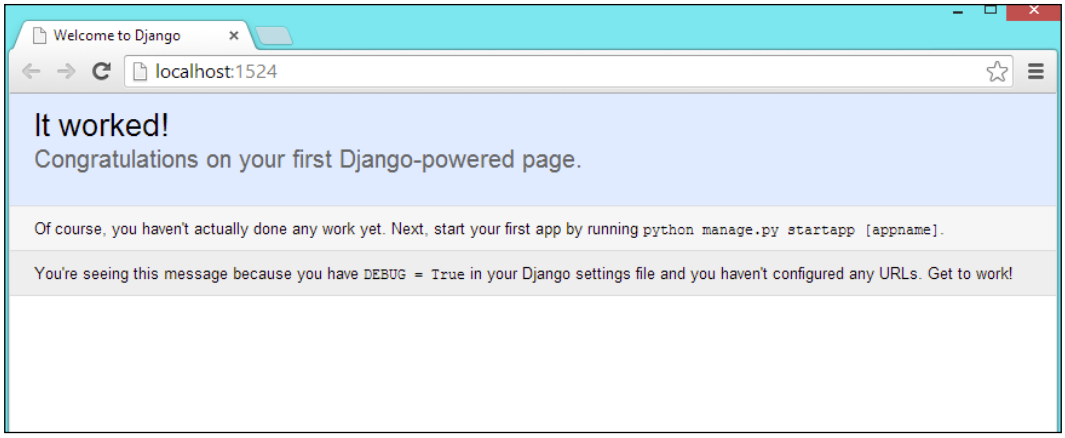
The following two things happen when you run the application:

- PTVS runs the Django `manage.py` command
- A browser instance with the output of the website is shown

The first one is shown in a command-prompt window that informs us about the result of the execution. It indicates whether there was any problem in the settings. It is the equivalent of launching the `python manage.py runserver` command from the command line.



At the same time, a new instance of the browser is created with our website's home page. In a new classical Django installation, you will see the following **It worked!** page:

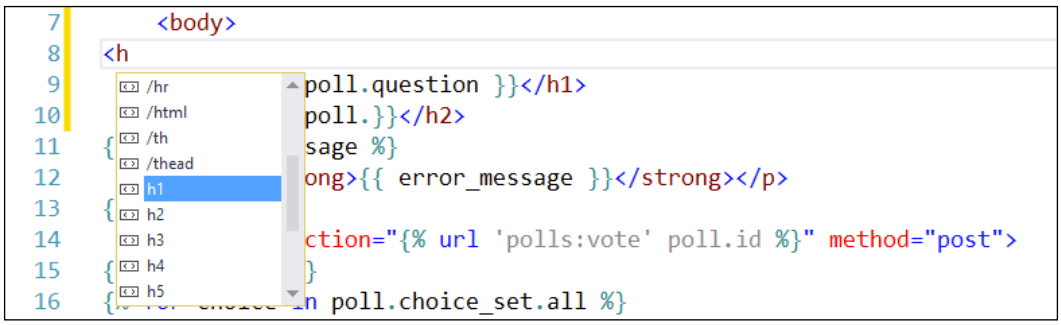


If you're able to see the preceding screen, it means that the setup of the Django project has been successfully completed. Now we can continue with the development of the website.

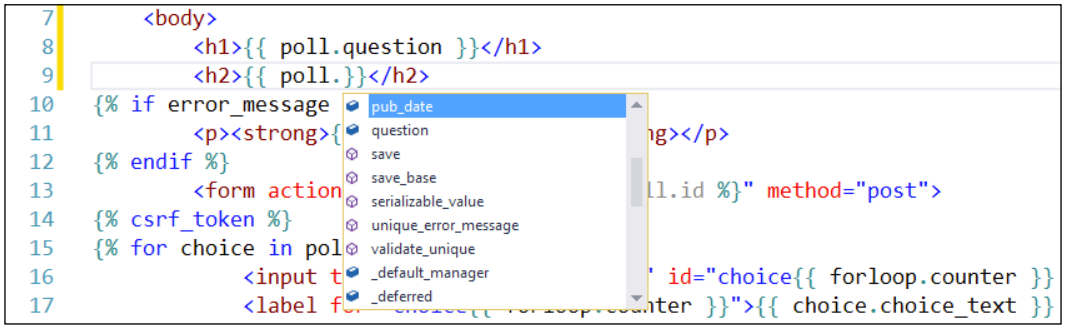
IntelliSense in Django templates

We have already talked about the power of IntelliSense in Visual Studio in Python code. IntelliSense supports even the Django template editor, providing it with access to all the template tags of Django; it also provides access to the context defined by the calling view.

IntelliSense offers help with the HTML part as follows:



Additionally, when in a Django block tag, `("{% … %}")`, it shows Django-specific tags and view context objects. IntelliSense also helps with filters:



Setting up and managing a database for a Django project

Once a working Django project is set up, we will need to attach a database in order to have a place to store the website data and the overall configuration of the Django admin console. For the purpose of this book, we are going to use a SQLite database, which is easy to manage, to connect to the project. It's a file-based database and can be easily managed by Django. For more information on SQLite, refer to its website at `http://www.sqlite.org/`.

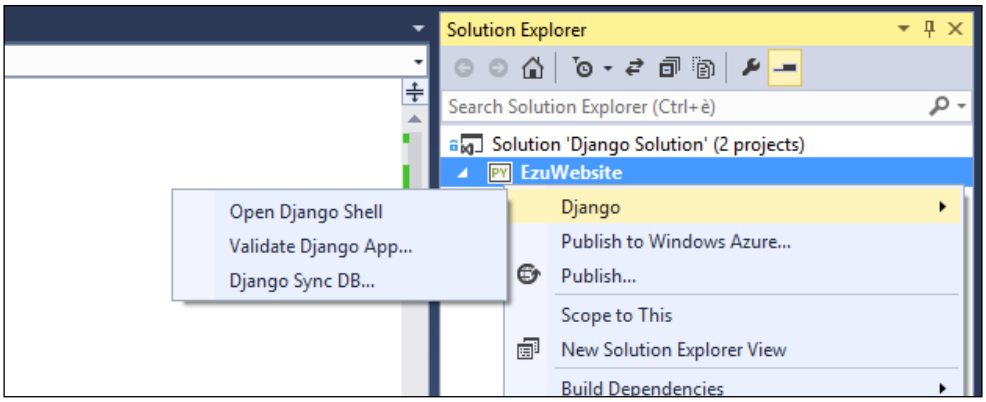
Attaching a database to Django is really easy. You just have to tell Django which database to use and how to connect to it. This has to be done in the Django `settings.py` file. To connect to and create a SQLite database, the database section of code should look like the following:

```
17
18  # Database definition section
19  DATABASES = {
20      'default': {
21          'ENGINE': 'django.db.backends.sqlite3', # Add 'postgresql_psycopg2', 'mysql', 'sqlite3' or 'oracle'.
22          'NAME':  path.join(PROJECT_ROOT, 'db.sqlite3'),   # Or path to database file if using sqlite3.
23          'USER': '',                      # Not used with sqlite3.
24          'PASSWORD': '',                  # Not used with sqlite3.
25          'HOST': '',                      # Set to empty string for localhost. Not used with sqlite3.
26          'PORT': '',                      # Set to empty string for default. Not used with sqlite3.
27      }
28  }
```

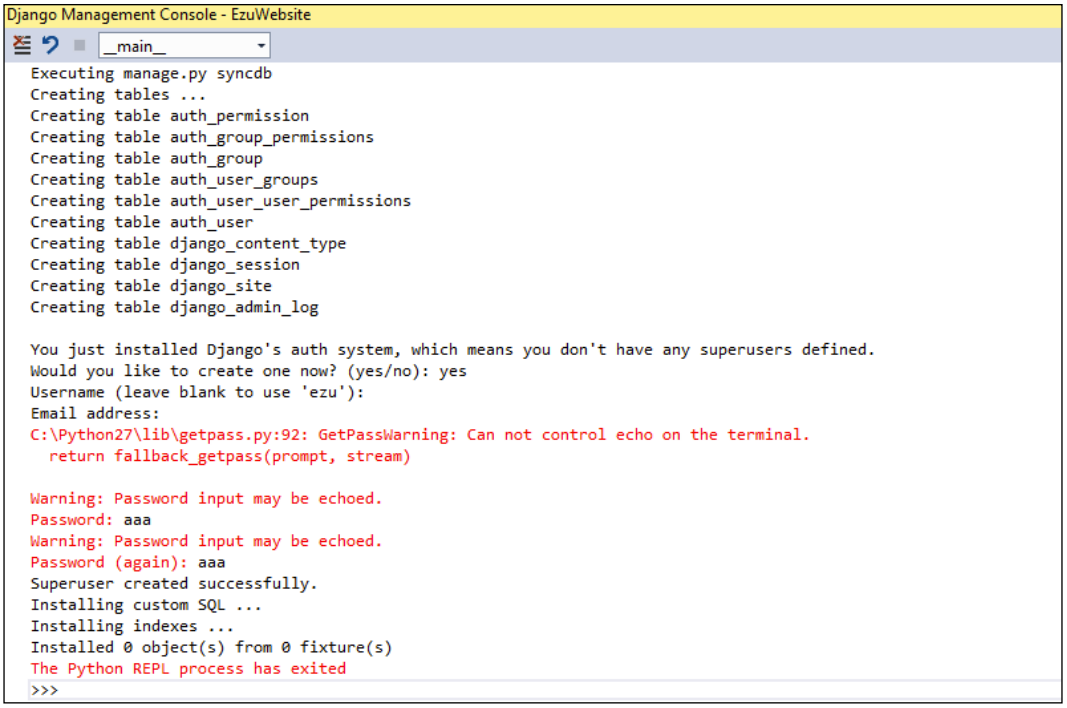
Since SQLite is a file-based database, the `Name` property should be the path of the database file. We are using a constant that contains the project's root path, `Project_Root`, which has to be defined first with the following two lines of code:

```
12  # Adding path exploring library
13  from os import path
14
15  # Getting the project root path
16  PROJECT_ROOT = path.dirname(path.abspath(path.dirname(__file__)))
```

The database is now connected to the Django project. To create the configuration tables in the database, use the Django sync DB command. Visual Studio provides a command, which starts this process from inside the IDE. To access the `sync` command, right-click on the project node of the **Solution Explorer** window and select the **Django Sync DB** entry to start the process:

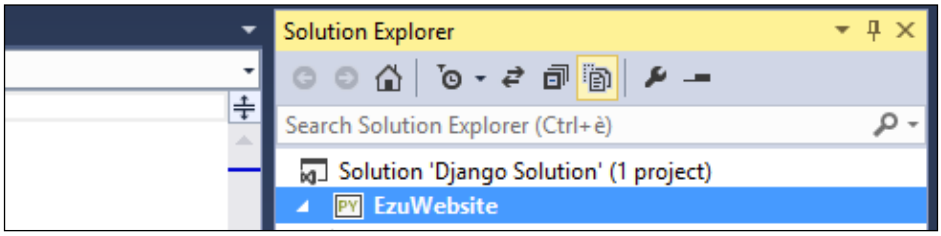


Once you click on **Django Sync DB**, the synchronizing starts. The **Django Management Console** option displays a detailed log of activities as follows:



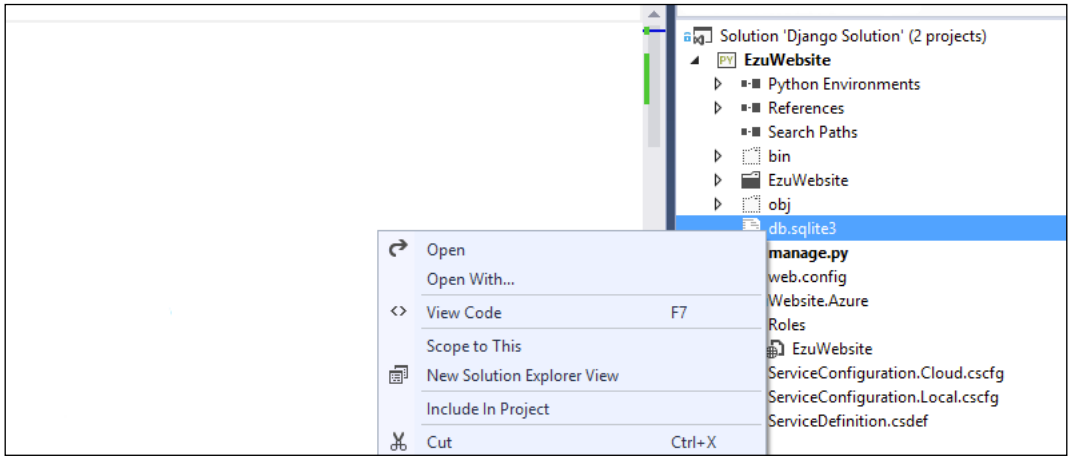
When the command is first executed, Django creates the actual database and also the authentication entries to be used for the Django administration console. The preferred username and password are asked to be entered. Once the information is provided, the process ends with the database created.

We can also see the result of the process in the **Solution Explorer** window since we have instructed Django to use SQLite and indicated the location to create the database file. To view this information, enable the **View all files** option in the solution folder by clicking on the **View all files** command in the **Solution Explorer** toolbar.



Find the "View all files" option in the toolbar

Once the view is active, the file `db.sqlite3` can be found in the solution folder. It can be included in the solution files by right-clicking on it and selecting the **Include in Project** command. The SQLite file is then included in the list of files that are managed by Visual Studio.



Setting up the admin interface

Now that a database is attached to the project, we can activate the administration interface of Django in it. The process is really simple at this point; you just need to uncomment a couple of lines of code in the project settings file and the main URL manager.

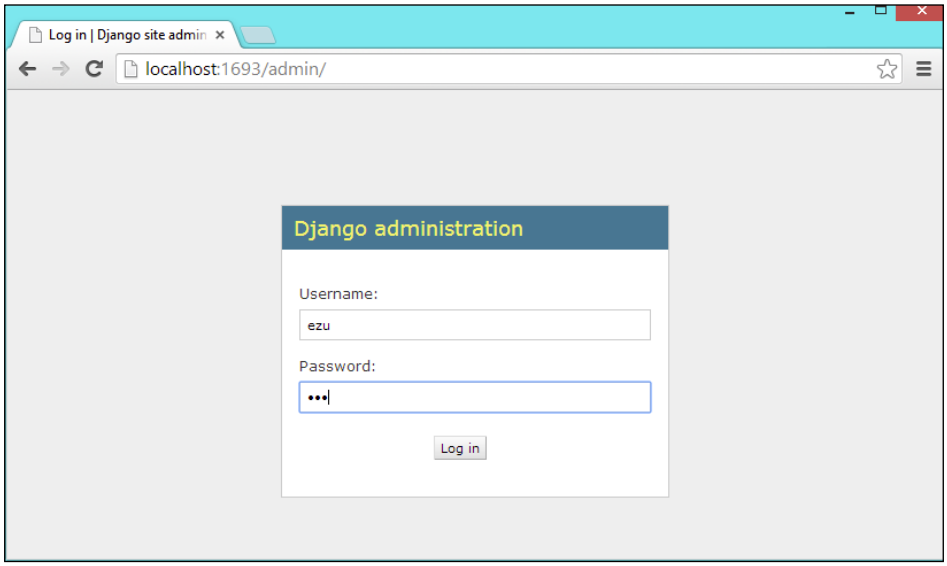
First, activate the Django application `django.contrib.admin` in the `Installed_Apps` section of the settings file:

```
120
121     INSTALLED_APPS = (
122         'django.contrib.auth',
123         'django.contrib.contenttypes',
124         'django.contrib.sessions',
125         'django.contrib.sites',
126         'django.contrib.messages',
127         'django.contrib.staticfiles',
128         # Uncomment the next line to enable the admin:
129         'django.contrib.admin',
130         # Uncomment the next line to enable admin documentation:
131         # 'django.contrib.admindocs',
132     )
133
```

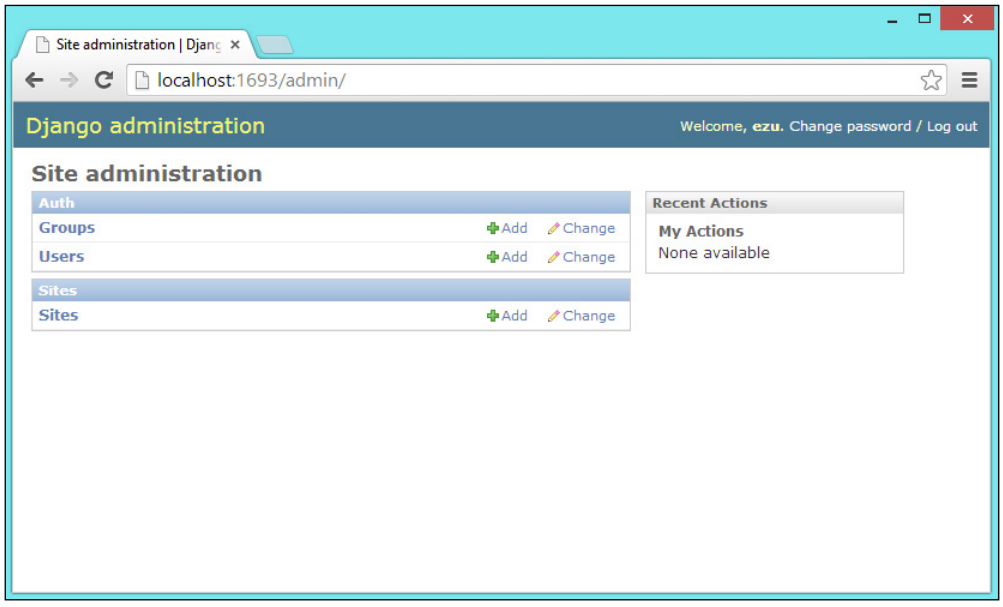
Second, go to the `urls.py` file and uncomment the section that imports the `admin` class and that enables the admin's discovery of the models inside the project. Besides this, it is also necessary that we uncomment the last line of the URL pattern that manages the `/admin/` path:

```
3   # Uncomment the next two lines to enable the admin:
4   from django.contrib import admin
5   admin.autodiscover()
6
7   urlpatterns = patterns('',
8       # Examples:
9       # url(r'^$', 'EzuWebsite.views.home', name='home'),
10      # url(r'^EzuWebsite/', include('EzuWebsite.EzuWebsite.urls')),
11
12      # Uncomment the admin/doc line below to enable admin documentation:
13      # url(r'^admin/doc/', include('django.contrib.admindocs.urls')),
14
15      # Uncomment the next line to enable the admin:
16      url(r'^admin/', include(admin.site.urls)),
17  )
18
```

Now the Django administration application is activated in the project. If the operation is successful, you will be presented with the admin login page. After launching the application, navigate to the `/admin/` page:



Insert the credentials created during the database setup to access the administration section:



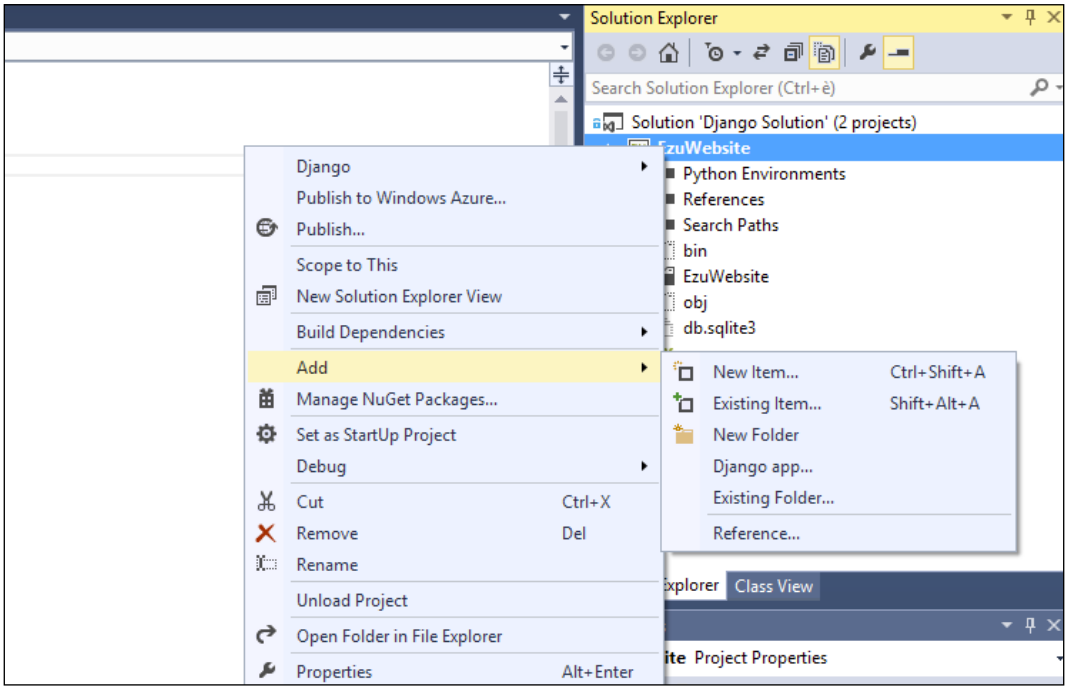
Now we have a fully set up and working Django environment. Let's go ahead and create a new Django application in Visual Studio.

Creating a new Django application

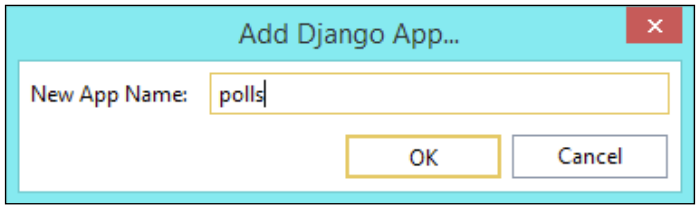
A Django application is a submodule of the project which is self-contained and not intertwined with other applications. In theory, you could copy it and put it in another project without much, or any, change.

Typically, to start a Django app, you have to run the `manage.py startapp` command in the command line, which will create a new folder in your project where you can find a view, a model, an admin, and a test Python file.

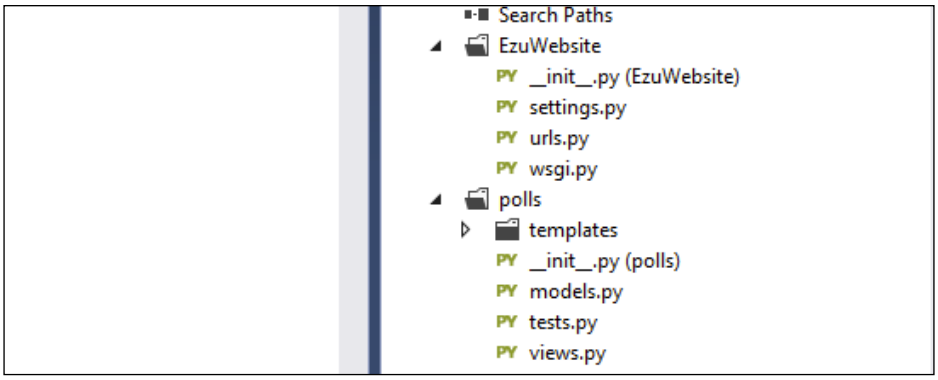
Visual Studio provides an easy way of creating a new Django app right in the IDE, automating the whole process; furthermore, the command is also in the process of creating a template folder. To create a new Django application, just right-click on the project node in the **Solution Explorer** window and select the **Django app** command in the **Add** menu:



This opens the **Add Django App** window in which you can assign the name of the Django app to be created:



By clicking on **OK**, the Django app is created in the project, and a list of added files can be found in the **Solution Explorer** window:



Now you can create the code for the app as we usually do for Django. Take into account that even if Visual Studio automates the creation of the Django application, it doesn't necessarily create the entry in the settings file. This has to be done manually as usual, adding the reference into the `Installed_Apps` section; to do this, just open the main `settings.py` file and add the newly created application in the `Installed_Apps` section:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # Uncomment the next line to enable the admin:
    'django.contrib.admin',
    # Uncomment the next line to enable admin documentation:
    # 'django.contrib.admindocs',
    'EzuWebsite.polls',
)
```

Deploying a Django project on Microsoft Azure

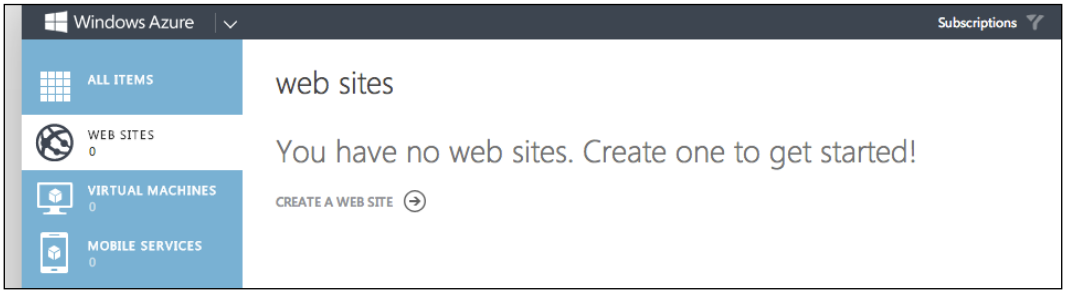
Cloud platforms are complex systems that provide ways to scale web applications across multiple server instances based on the traffic the application is receiving. They also offer an abstraction of the server environment in a way that developers do not need to deal with the hardware and software architecture of the system, but only with the resources and services. This is an advantage because with this in place, the developer does not actually need to configure and manage the server; however, at the same time, it also proves to be a disadvantage, since not all software components are able to work in such systems.

Azure is a cloud-hosting platform by Microsoft that enables developers to quickly build, deploy, and manage applications across a global network of Microsoft-managed data centers. It is tightly integrated in Visual Studio; also, the Python language is covered by this service.

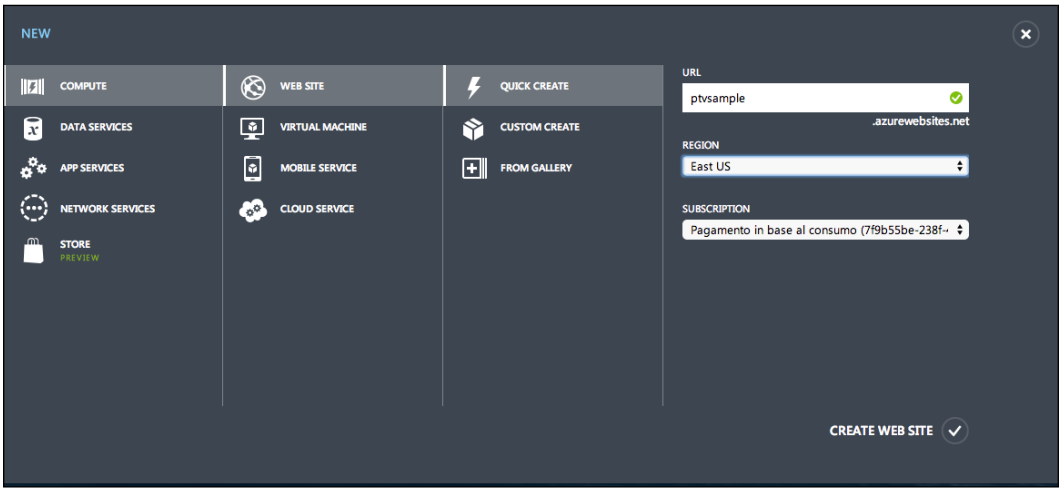
We are going to see how to deploy a Django project on Azure by using the tools that Visual Studio provides.

To use Microsoft Windows Azure, first we need to create an Azure account via `http://www.windowsazure.com/`.

After the account is created, you can access the **Portal** section that brings you to your Azure services portal in which you can find all the services available on the left-hand side menu. Now we'll create a website:

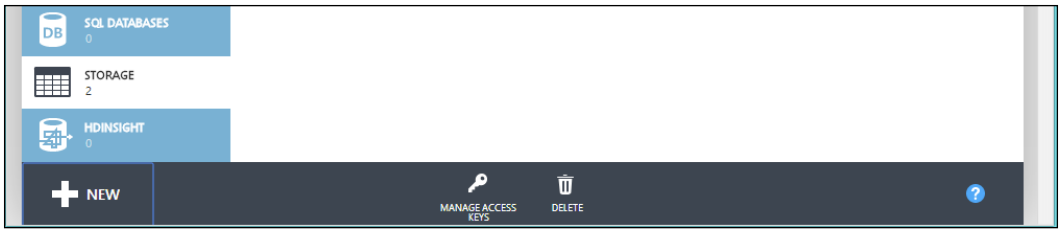


To create a new website, click on the **CREATE A WEB SITE** link; this will open the **NEW** tab at the bottom of the page:

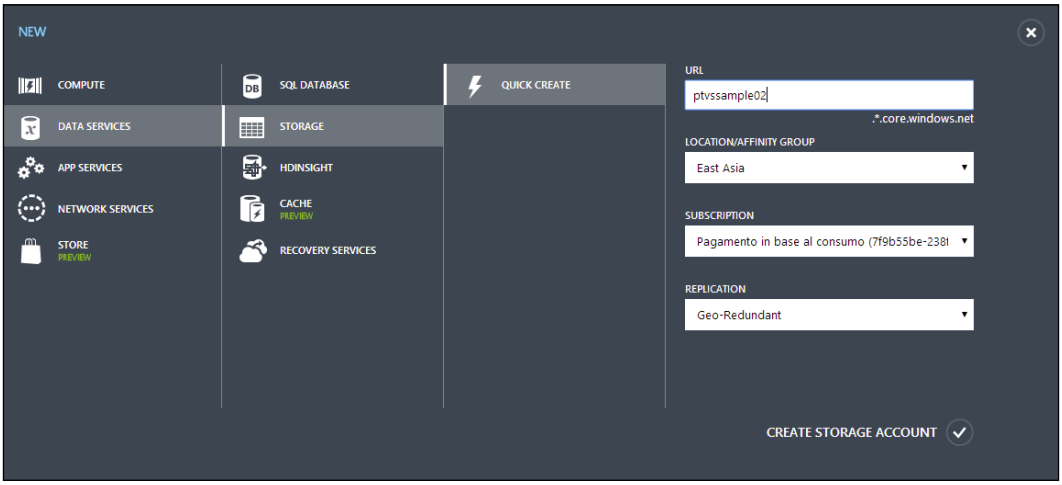


Just insert the name of the website that you want to create in the **URL** textbox; if the name is available, click on the **CREATE WEB SITE** button to create the website. This will initiate the website generation process, at the end of which you will see the created website:

Besides the website service, we also need to create a storage entry where the website files can be uploaded to. To create a storage entry, click on the **Storage** item on the left-hand side menu and click on **NEW**:



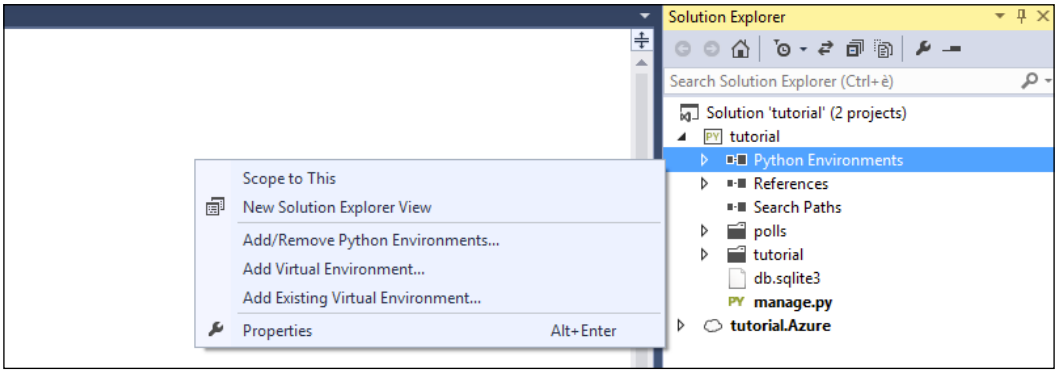
This will open the storage creation area at the bottom of the page:



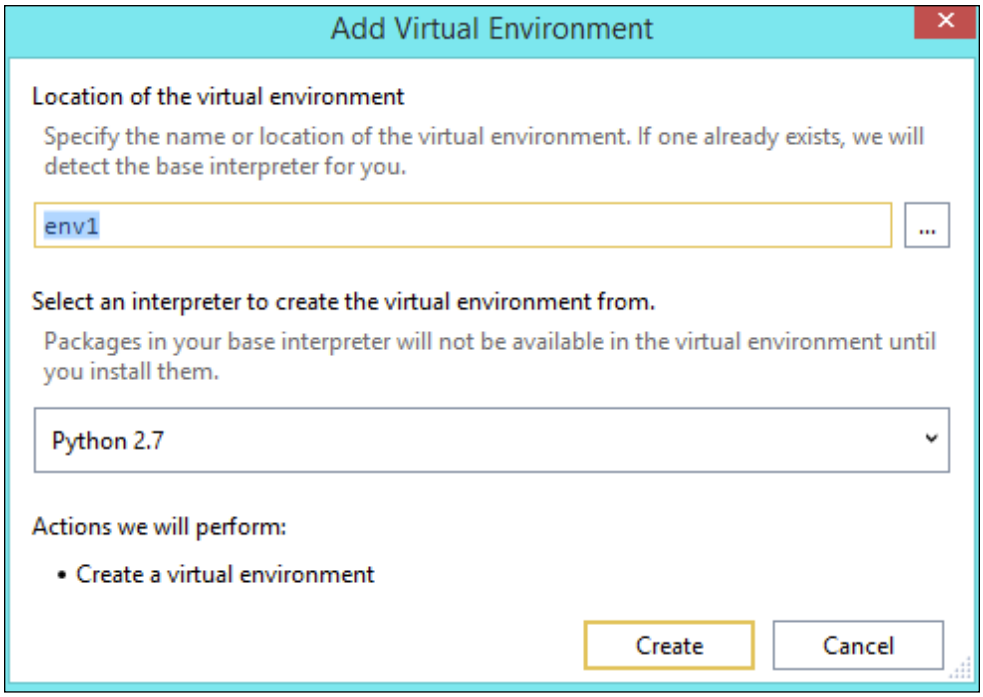
As for the website service, we just need the name of the storage service in the **URL** textbox; once you have this, click on the **CREATE STORAGE ACCOUNT** button to create the storage service instance.

Now we have all the elements needed to deploy our Django project to Azure. However, before going ahead, we need to ensure that our project is ready for it. Azure needs all the files related to the project that reside in the project folder, since it needs to ensure that all the Python libraries are present. To ensure that these requirements are met, the Django project needs to be created in a virtual environment. This will keep all the files to be included in the project folder; otherwise, the libraries just reside in the site-packages folder of the system.

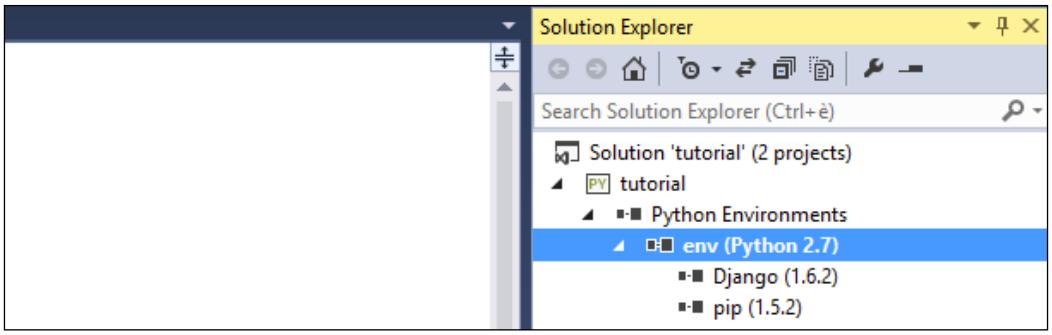
A virtual environment is an isolated working copy of the Python environment, which allows you to work on a specific project without worrying about affecting other projects. At the same time, it keeps together all the dependencies in the same folder structure. Creating a virtual environment is easy in PTVS; you can do this by selecting the **Add Virtual Environment** command from the **Python Environments** contextual menu, as shown in the following screenshot:



This will open the **Add Virtual Environment** window:

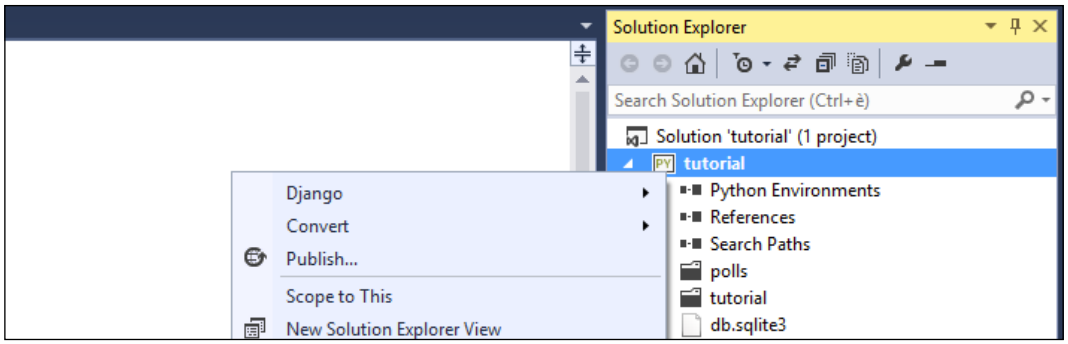


Once you enter the name of the virtual environment and click on the **Create** button, the virtual environment is created in the project. Now, all the packages that you include in the project will be copied into it, so all the dependencies will be available inside the project folder, which doesn't rely on the system's site-packages folder:

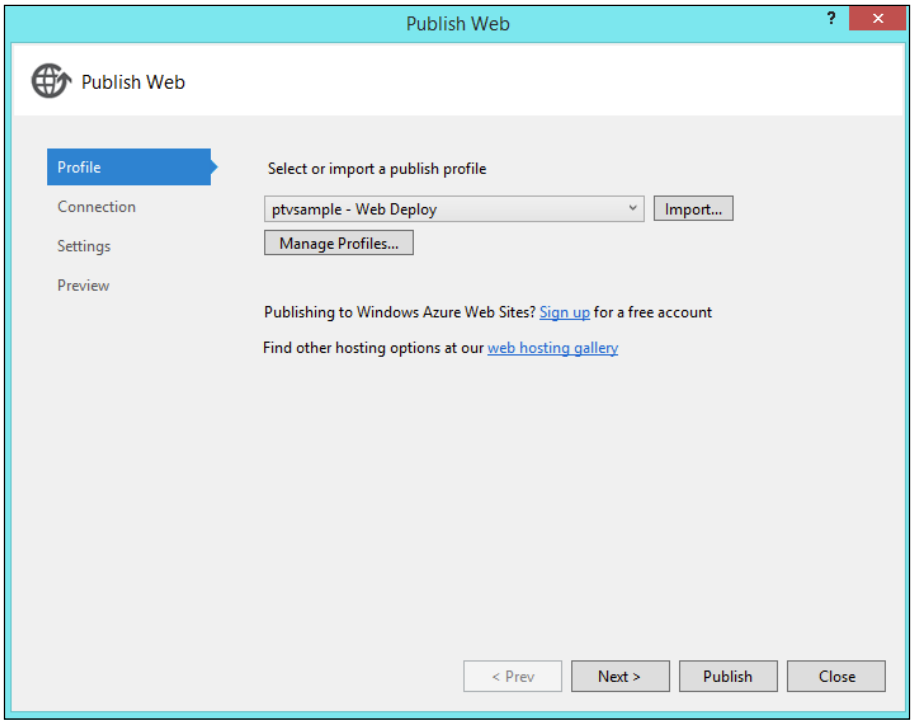


Once we have ensured that our project is contained in a virtual environment, we can go ahead and configure Visual Studio in a way to be ready to deploy our Django project to Azure.

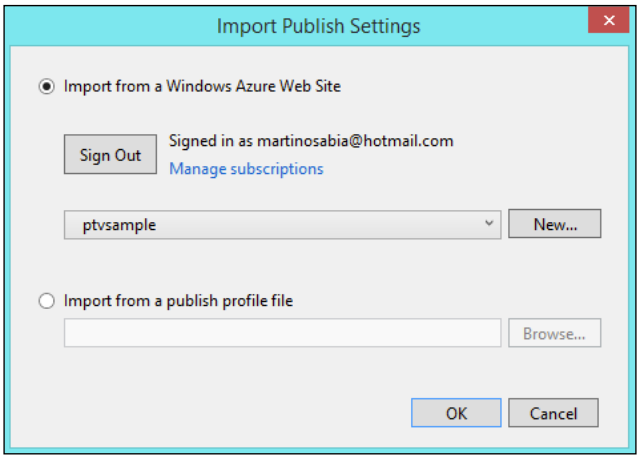
The deployment procedure starts by invoking the **Publish** command from the contextual menu of the Django website node in the **Solution Explorer** tree view:



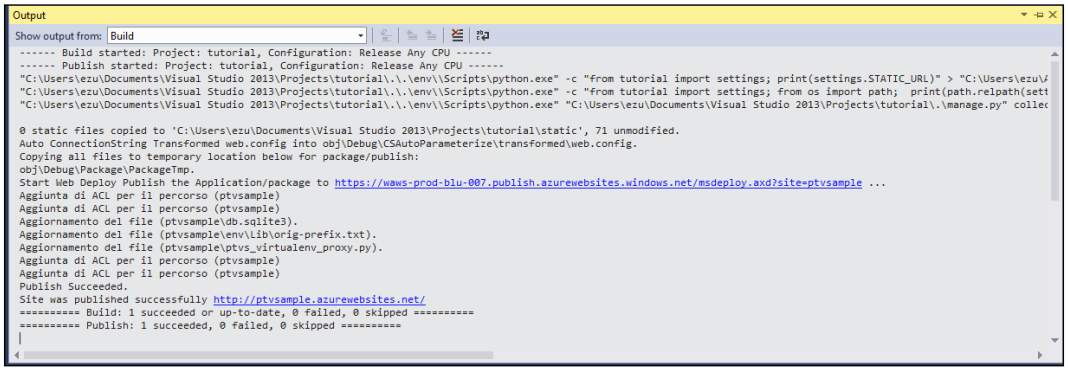
This will open the **Publish Web** wizard window, which will guide you through the deployment process of the Django project:



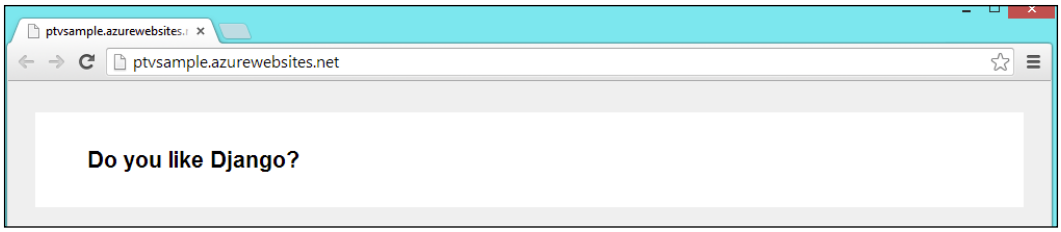
To link Visual Studio to the Azure website we created earlier, click on the **Import** button; once you are certain about being able to log in with your Azure account, select the name of the website you want the Django application to be deployed to:



At this point, follow the other wizard entries and click on **Publish**. With this, Visual Studio will start the process of deploying all the files to the Azure platform; you can follow the deployment process from the **Output** window:



Once the deployment process finishes, Visual Studio will open in the default browser of the deployed website on Azure:



For further documentation about Python on Windows Azure, refer to the Windows Azure website at `http://www.windowsazure.com/en-us/develop/python/`.

Summary

In this chapter, we learned how to create a Django project in Visual Studio and set up the environment. We also learned about deploying a Django project on Microsoft Azure.

In the next chapter, we are going to learn how to manage Python libraries inside Visual Studio. Also, we'll learn how to use some of the most powerful Python libraries for the Django framework to manage deployments, the synchronization of databases between environments, and a library that helps us use and manage images successfully and dynamically in our website.

5
Advanced Django in PTVS

Once we look at how a Django development environment in Visual Studio with PTVS is set up, we can start analyzing some powerful libraries for Django and how to use them in Visual Studio. Over the years, lots of developers have created powerful libraries and tools for Django that speed up various aspects of the development cycle. We are going to take a closer look at some of them here to see how they integrate in Visual Studio and PTVS.

In this chapter, we will analyze two libraries that are useful for a Django developer in two different aspects: automatizing tasks using the Fabric library, and managing model migrations on Django with South.

Library management

We have already learned how to install new packages in PTVS using the GUI tools that it provides. Now, we will learn more about the criteria for choosing one package index over another; in other words, when to choose `easy_install` over `pip`.

Generally speaking, using `pip` is much better than using `easy_install`, and there are major reasons for this. As Ian Bicking, the creator of `pip`, wrote in his own introduction to `pip`, the advantages are as follows:

- All packages are downloaded before installation. As a result, partially completed installation doesn't occur.
- Care is taken when presenting useful output on the console.
- The reasons for actions are being tracked. For instance, if a package is being installed, `pip` keeps track of why that package was required.
- Error messages should be useful.

- The code is relatively concise and cohesive, making it easier to use programmatically.
- Packages don't have to be installed as egg archives; they can be installed flat (while keeping the egg metadata).
- Native support is available for other version control systems (`Git`, `Mercurial`, and `Bazaar`).
- Uninstallation of packages is easy.
- It is simple to define, has fixed sets of requirements, and reliably reproduces a set of packages.

It may seem that there are no reasons to choose `easy_install` over `pip`. However, this is where careful consideration is needed.

There is a caveat that makes the choice really hard for Windows environments: some libraries or dependencies are written in Python C, which is a way for Python to call libraries written in C/C++. To compile these libraries on your Windows machine, you have to install the exact same version of the original compiler that has been used to compile to the original interpreter. For example, you will need the C++ compiler of Visual Studio 2008 if you use Python 2.7 or Visual Studio 2010 for Python 3.

This is due to a long tradition where Python extension modules must be built with the same compiler version (more specifically, a CRT version) as Python itself, which is mentioned at `https://mail.python.org/pipermail/python-list/2010-April/573606.html`.

Using the `easy_install` package installer, the precompiled packages are downloaded and installed into the system's `site-packages` folder.

A rule of thumb: if the library you are trying to install on your Windows machine has Python C extensions in it, it's far better to choose `easy_install`. For all other cases, `pip` is way better.

If you don't know what kind of library you are importing, you should go with `pip`. If it encounters a problem during the compilation process of the installation, you can uninstall the library and reinstall it using `easy_install`.

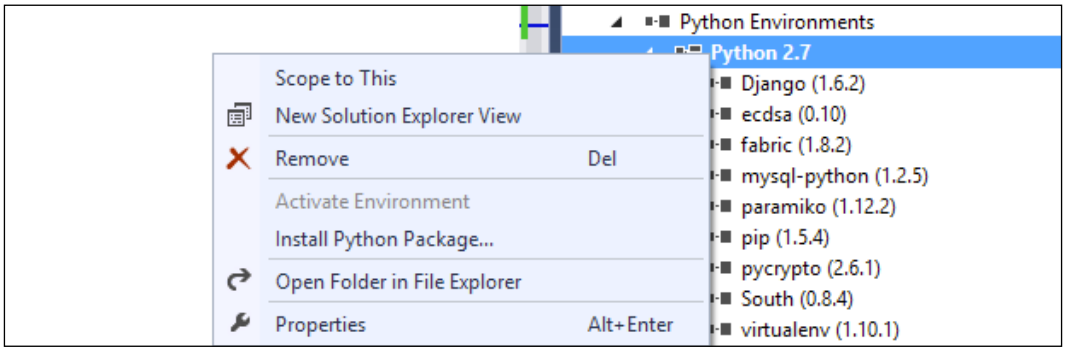
Generally, most libraries that have low-level capabilities (for example, cryptography, graphics, and mathematical functions) and interaction with other software (for example, drivers) use Python C extensions.

The Fabric library – the deployment and development task manager

Fabric is a Python library and a command-line tool that allows execution in application deployment and administration tasks. Essentially, Fabric is a tool that allows the developer to execute arbitrary Python functions via the command line and also a set of functions in order to execute shell commands on remote servers via SSH. Combining these two things together offers developers a powerful way to administrate the application workflow without having to remember the series of commands that need to be executed on the command line.

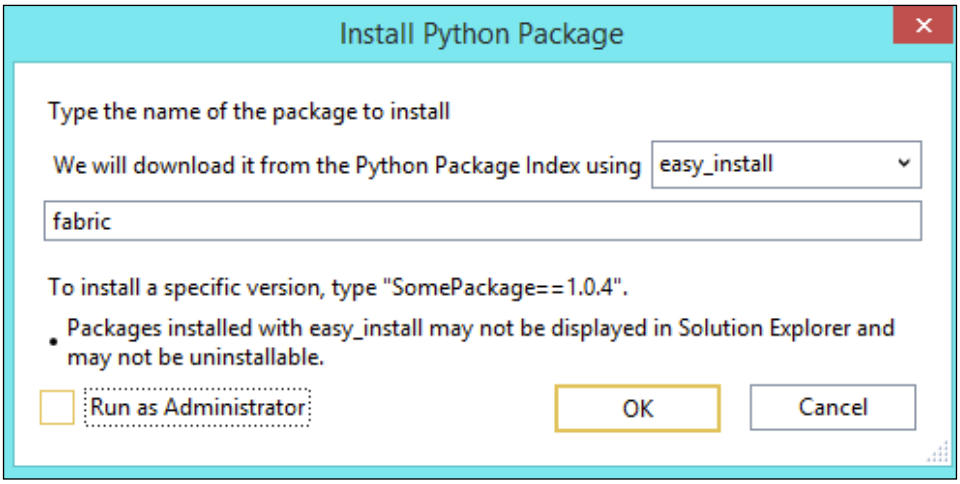
The library documentation can be found at `http://fabric.readthedocs.org/`.

Installing the library in PTVS is straightforward. Like all other libraries, to insert this library into a Django project, right-click on the **Python 2.7** node in **Python Environments** of the **Solution Explorer** window. Then, select the **Install Python Package** entry.



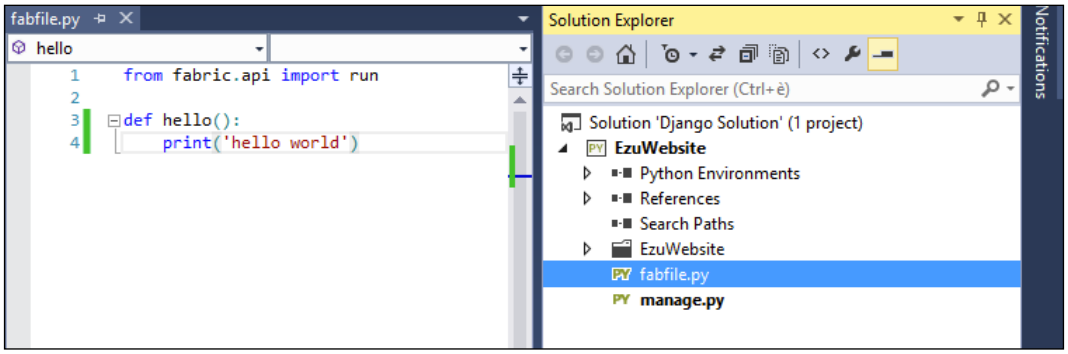
The Python environment contextual menu

Clicking on it brings up the **Install Python Package** modal window as shown in the following screenshot:

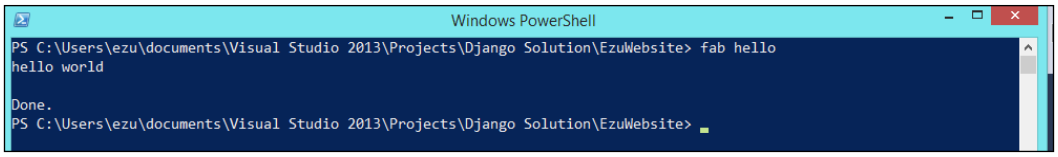


It's important to use `easy_install` to download from the Python package index. This will bring the precompiled versions of the library into the system instead of the plain Python C libraries that have to be compiled on the system.

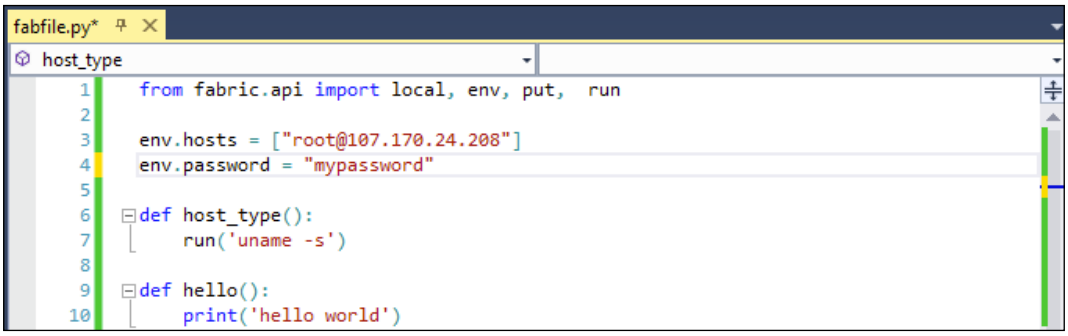
Once the package is installed in the system, you can start creating tasks that can be executed outside your application from the command line. First, create a configuration file, `fabfile.py`, for Fabric. This file contains the tasks that Fabric will execute.



The previous screenshot shows a really simple task: it prints out the string `hello world` once it's executed. You can execute it from the command prompt by using the Fabric command `fab`, as shown in the following screenshot:



Now that you know that the system is working fine, you can move on to the juicy part where you can make some tasks that interact with a remote server through `ssh`. Create a task that connects to a remote machine and find out the type of OS that runs on it.

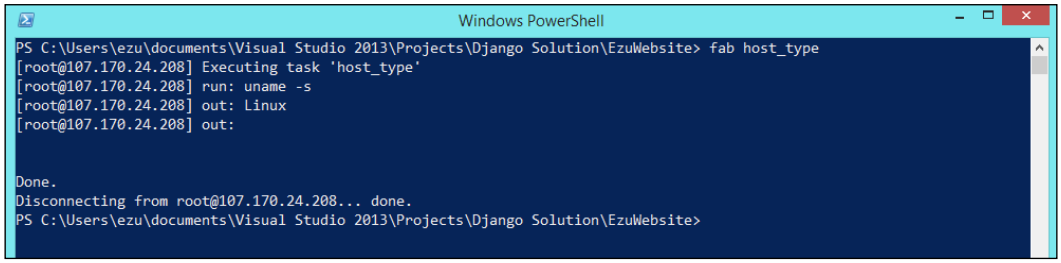


The env object provides a way to add credentials to Fabric in a programmatic way

We have defined a Python function, `host_type`, that runs a POSIX command, `uname -s`, on the remote. We also set up a couple of variables to tell Fabric which is the remote machine we are connecting to, i.e. `env.hosts`, and the password that has to be used to access that machine, i.e. `env.password`.

It's never a good idea to put plain passwords into the source code, as is shown in the preceding screenshot example.

Now, we can execute the `host_type` task in the command line as follows:

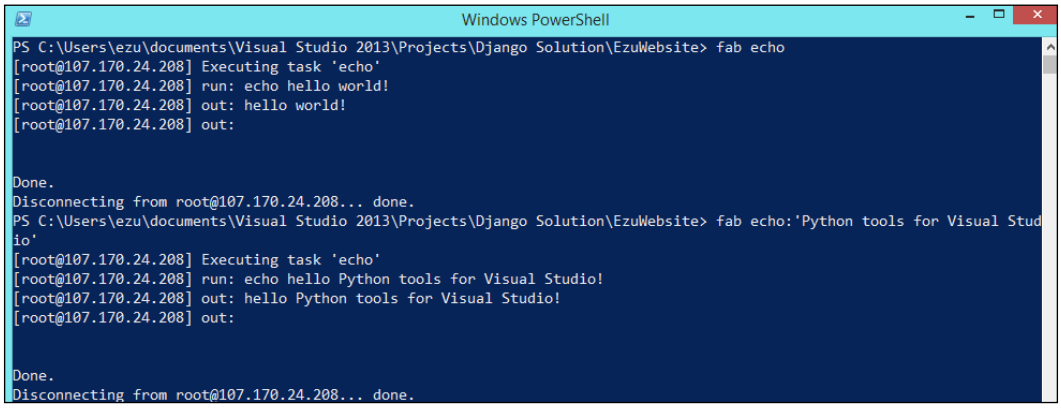


The Fabric library connects to the remote machine with the information provided and executes the command on the server. Then, it brings back the result of the command itself in the output part of the response.

We can also create tasks that accept parameters from the command line. Create a task that echoes a message on the remote machine, starting with a parameter as shown in the following screenshot:

```
11
12  def echo(who='world'):
13      run('echo hello %s!' % who)
```

The following are two examples of how the task can be executed:



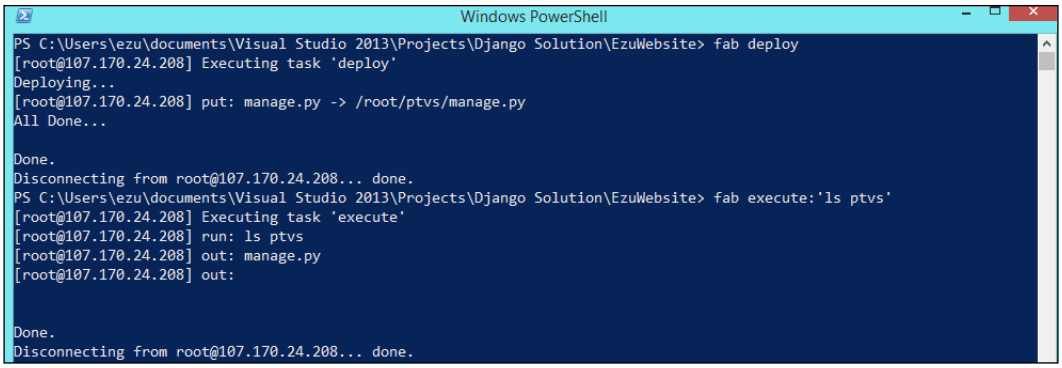
We can also create a helper function that executes an arbitrary command on the remote machine as follows:

```
def execute(cmd):
    run(cmd)
```

We are also able to upload a file into the remote server by using `put`:

```
18  def deploy():
19      print "Deploying..."
20      put("manage.py", "~/ptvs/manage.py")
21      print "All Done..."
22
```

The first argument of `put` is the local file you want to upload and the second one is the destination folder's filename. Let's see what happens:



Deploying process with Fabric

The possibilities of using Fabric are really endless, since the tasks can be written in plain Python language. This provides the opportunity to automate many operations and focus more on the development instead of focusing on how to deploy your code to servers to maintain them.

South – the database deployment library

Developed by the Python community, South is a Django library that brings schema migration to Django applications. The South library's main objective is to provide a simple, stable, and database-independent migration layer to prevent all the hassles of schema changes.

The key features of South are as follows:

- **Automatic migration creation**: South can detect what's changed in your application model by analyzing your `model.py` files and automatically creating the migration code—basically the SQL commands for the database you are using—that matches the changes in the models.
- **Database independence**: South is database agnostic, supporting different database backends. Currently, South supports PostgreSQL, MySQL, SQLite, Microsoft SQL Server, Oracle, and Firebird (beta support).
- **App-savvy**: South knows and works with the concept of Django applications, allowing developers to use migrations on only some of the applications and not on the whole project.
- **VCS-proof**: South will notice when someone else commits migrations to the same application and can check if there are conflicts.

Why use South with Django

One of the most interesting parts of Django is its **Object-relational mapping** (**ORM**), which creates a consistent abstraction of the database structure. This is a very powerful tool that allows programmers to focus on the Python code. Django takes good care of the database structure management only for new models (for example, when creating them). It doesn't have an out-of-the-box solution to manage updates in the models that can be applied to existing database schemas.

It's usually a painful operation to change the model during the application lifecycle. Technically, when changing the schema of the model or when migrating the schema, whether you are modifying a field or adding another one, the database structure needs to be recreated. This means that all the data of that model is lost, or a manual migration needs to be done to move the data from the old version of the tables to the new one on the database. This is especially time consuming if you have to align that database from a development server environment to a production server environment.

Installing South

Let's see how to bring South into PTVS. Like other Python libraries, we can install it from the **Solution Explorer** window by right-clicking on the environment of your choice (Python 2.7) and selecting **Install Python Package** to bring up the following installation dialog box:

As stated in the South documentation, you have to use the `easy_install` **Python Package Index**; be sure to select it.

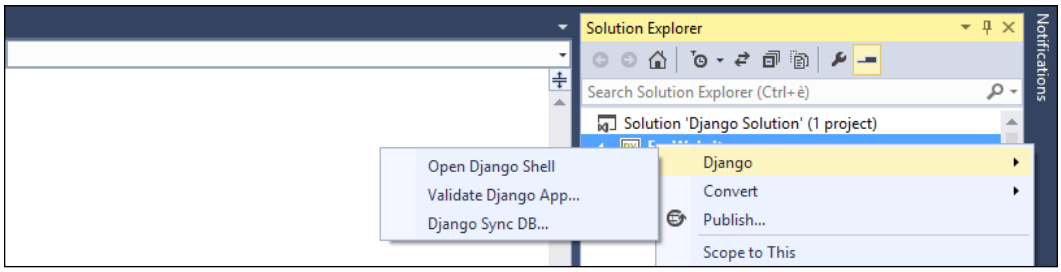
Once the package is installed, it's important to make sure that it's activated in the settings file. To do so, add `south` at the end of the code for `Installed_Apps`.

```
121  INSTALLED_APPS = (
122      'django.contrib.auth',
123      'django.contrib.contenttypes',
124      'django.contrib.sessions',
125      'django.contrib.sites',
126      'django.contrib.messages',
127      'django.contrib.staticfiles',
128      # Uncomment the next line to enable the admin:
129      'django.contrib.admin',
130      # Uncomment the next line to enable admin documentation:
131      # 'django.contrib.admindocs',
132      'EzuWebsite.polls',
133
134      # Be sure South is the last library in the section
135      'south'
136  )
```

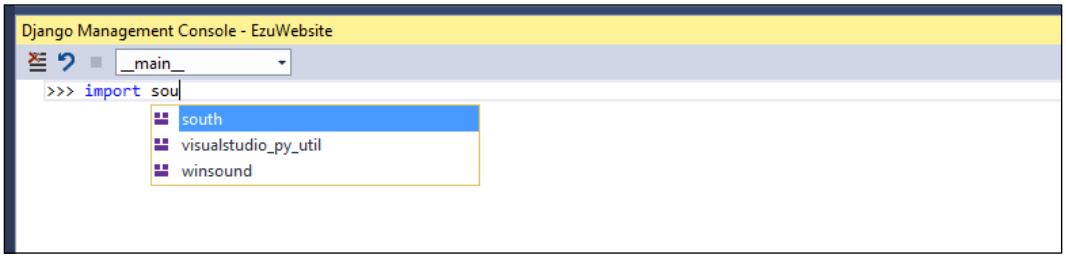
Be sure that the "south" library is the last in the "Installed_Apps" section in Settings.py

South needs to be the last package of the list due to the fact that when Django executes the library, all the models of the Django project are already created and are discoverable by South.

To test if everything is working, navigate to the Django shell and try to import the library. Ordinary Python developers will go to the command line and run the `manage.py` shell, but in PTVS, there's a panel for this. To open it, quickly right-click on the **Django** project entry in the **Solution Explorer** window and select the **Open Django Shell** option in **Django**:

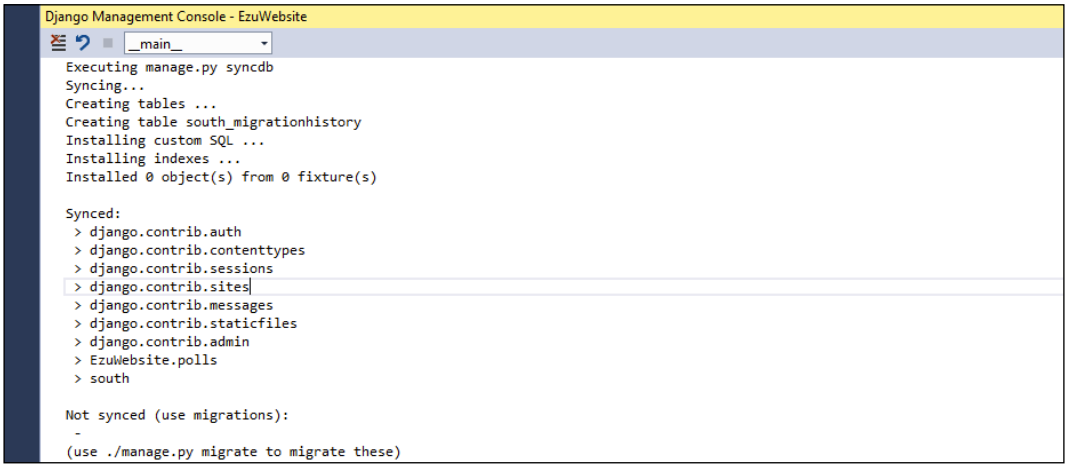


This opens a new **Django Management Console** panel, which is basically a REPL but with Django integration. From here, it's possible to see if the South library is working correctly by trying to import the library:



IntelliSense is active in the Django shell, so if you see **south** appear, then everything is working fine.

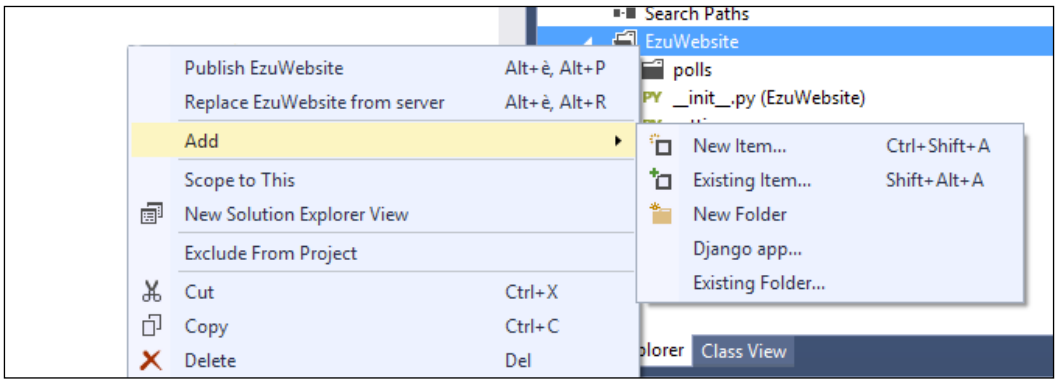
To finish the initialization process, run `sync_db` for South to create the migration-tracking tables. This can also be done from the Django contextual menu as seen earlier: just select the **Django Sync DB** command in the **Django** menu.



As shown in the preceding screenshot, this starts the synchronization process of the current models in your application on South.

Schema migration with South

Now that we have `south` installed and working in our solution, let's try to create something to test the migration. Create a new application in your project and call it `south2ptvs` by using the **Django app** command in the **Add** menu, as shown in the following screenshot:



Don't forget to register the new application in the `settings.py` file, and make sure that `south` remains as the last entry of the `Installed_Apps` section:

```
121  INSTALLED_APPS = (
122      'django.contrib.auth',
123      'django.contrib.contenttypes',
124      'django.contrib.sessions',
125      'django.contrib.sites',
126      'django.contrib.messages',
127      'django.contrib.staticfiles',
128      # Uncomment the next line to enable the admin:
129      'django.contrib.admin',
130      # Uncomment the next line to enable admin documentation:
131      # 'django.contrib.admindocs',
132      'EzuWebsite.polls',
133      'south2pvts',
134      # Be sure South is the last library in the section
135      'south'
136  )
```

Then, open the `models.py` file of the newly created application in which we are going to define our testing model:

```
models.py
1  from django.db import models
2
3  class Knight(models.Model):
4      name = models.CharField(max_length=100)
5      of_the_round_table = models.BooleanField()
```

Instead of using the standard Django `sync_db` command to create the schema of the model in the database, let's set up a migration for the model `Knight`. This operation will be the entry point for the entire migration history of the model.

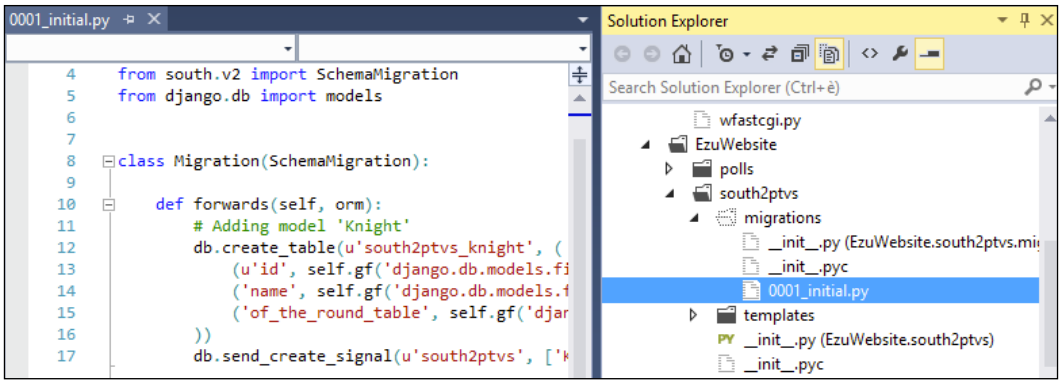
Navigate to the command line and execute the initialization migration by executing the following command:

```
python manage.py schemamigration south2ptvs --initial
```

This will execute South's `schemamigration` command on the `south2pvts` application for the initialization process. Here is what is going to happen:

```
Windows PowerShell
PS C:\Users\ezu\documents\Visual Studio 2013\Projects\Django Solution\EzuWebsite> python manage.py schemamigration south
2ptvs --initial
Creating migrations directory at 'C:\Users\ezu\documents\Visual Studio 2013\Projects\Django Solution\EzuWebsite\EzuWebsi
te\south2ptvs\migrations'...
Creating __init__.py in 'C:\Users\ezu\documents\Visual Studio 2013\Projects\Django Solution\EzuWebsite\EzuWebsite\south2
ptvs\migrations'...
 + Added model south2ptvs.Knight
Created 0001_initial.py. You can now apply this migration with: ./manage.py migrate south2ptvs
PS C:\Users\ezu\documents\Visual Studio 2013\Projects\Django Solution\EzuWebsite>
```

We have successfully created the migration file but haven't applied it to `db`. Since **South** works on one application at a time, the migration file in which the information of the migration is stored is created inside the `south2ptvs` folder.

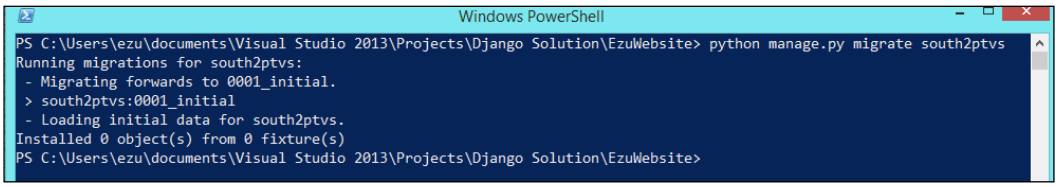


The content of the migrations folder in the Django app

The migration files are plain files written in Python. They can be edited, but you should do it with caution and only do so when necessary.

The only thing left to do is to apply the migration to the database by calling the South library's `migrate` command on the app with the following command:

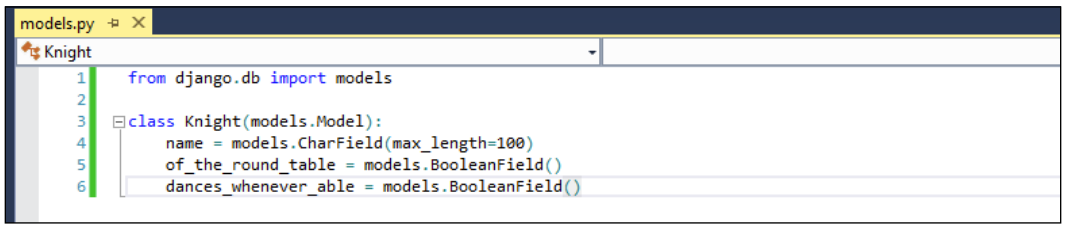
```
python manage.py migrate south2ptvs
```



Execution of South's migration command

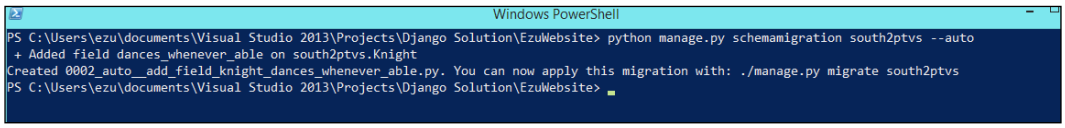
This will finalize the migration; now our model is ready to be modified. Future updates to the model can be easily applied to the database by South using migrations.

Update the model by adding a new field as follows:



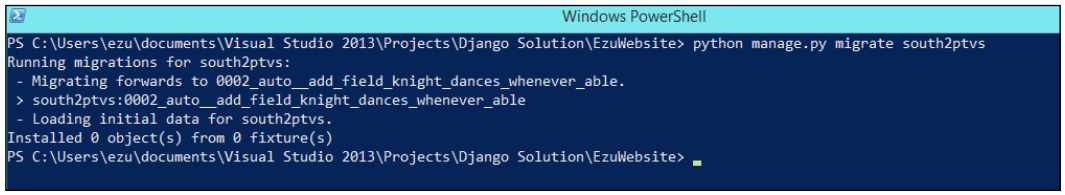
So, now we have to create a new migration file and then apply it to the database. To create the migration file, use the `schemamigration` command again as shown in the following command. However, instead of the `--initial` parameter, use `--auto`, since a migration is already defined in the model.

```
python manage.py schemamigration south2ptvs --auto
```



To obtain the result of the migration file creation state to apply the migration to `db`, just run the following `migrate` command:

```
python manage.py migrate south2ptvs
```



If we register the model in the admin interface, we can go to the admin section of our website and see if the new field appears in the model:

Since the migrations are stored on the files, you can apply the changes on the remote server by just copying the migration files to the server and then applying the migration on the remote system (maybe with Fabric).

For more details and insights into this powerful tool, visit the official documentation website for South at `http://south.readthedocs.org/`.

Summary

In this chapter, we took a more in-depth look into how to deal with third-party Python libraries in PTVS, the main differences between the `pip` and `easy_install` package indexes, and how they deal with precompiled libraries written with Python C extensions.

We also looked into two popular and powerful Django open source libraries, Fabric and South, which add remote task management and schema migrations to your Django projects.

In the next chapter, we will introduce IPython and its graphic power in Visual Studio in order to cover the topic of IronPython and its integration with the .NET framework.

6
IPython and IronPython in PTVS

In this chapter, we will see how PTVS interacts with two particularly useful Python extensions: IPython and IronPython.

Despite their names, they are very different from each other. IPython is more oriented toward extending the REPL interface in a way that can help you have a more interactive approach to the code, providing you with features such as on-the-fly graph plotting. IronPython provides .NET class access to your Python code and integrates Python in .NET applications.

IPython in PTVS

IPython is a command shell for interactive computing for Python (also available for other language integrations) that offers enhanced type introspection—the possibility to examine the type or properties of an object at runtime—rich media, and REPL extensions.

As an interactive shell tool used for data analysis and math graph plotting, IPython comes from an academic-scientific computing background, but appeals to data scientists through the power of graphing integration.

An interesting feature of IPython is its ability to plot mathematical graphs of expressions in an interactive way, much like MATLAB.

PTVS supports IPython libraries and provides the ability to integrate the graph that is generated inside REPL.

We need to install IPython and its supporting dependencies such as matplotlib from a distribution that has all the code already compiled for Windows OS. The distribution of this package can be downloaded from `http://www.lfd.uci.edu/~gohlke/pythonlibs/#scipy-stack`. This web page, from the University of California in Irivine, contains an unofficial repository of Windows binaries for a large number of Python packages. There are different packages that are available, which depend on the version of the Python interpreter and the operating system you are using. For our proposal, we are going to install Python 2.7 for Windows 32-bit.

As an alternative, you can use the Python (x, y) distribution that contains a whole range of Python libraries for scientific and engineering software. The installer can be downloaded from `https://code.google.com/p/pythonxy/`.

Running the installer gives you the ability to choose the libraries it offers; for our scope, ensure that you include the SciPy libraries and IPython.

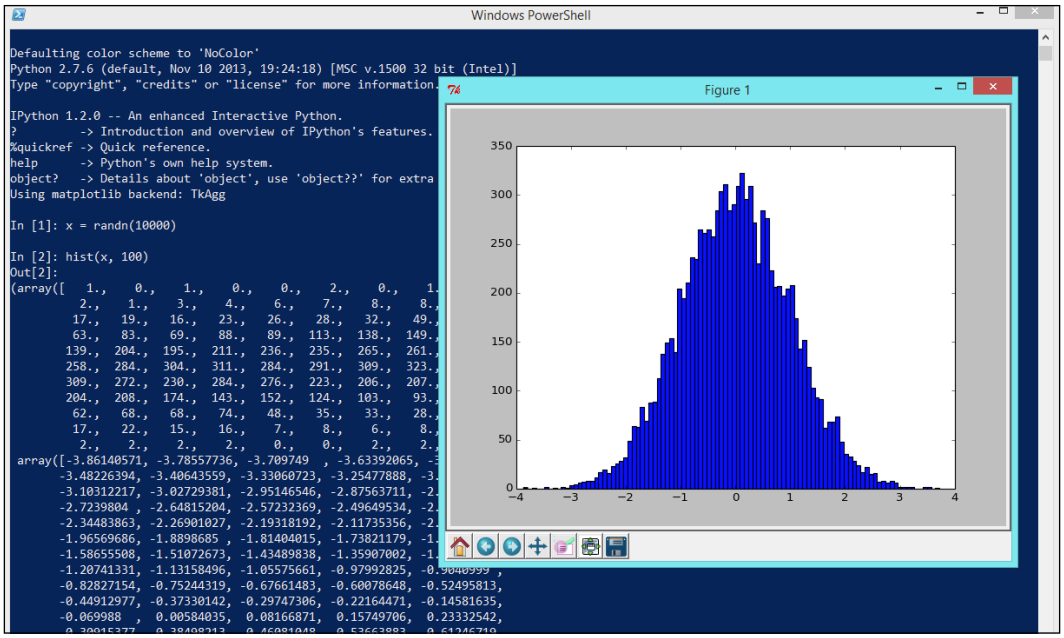
Once you have the libraries from the source of your choice, we can test the whole installation by executing the following command from the command prompt:

```
ipython --pylab
```

This command executes the IPython environment with the `pylab` extension. Also, it permits you to use the matplotlib library to plot graphs; this is a Python-plotting library that can be used with Python to plot graphs using mathematical functions. As an example, let's try to plot a histogram out of 10,000 random numbers that are clustered in 100 samples:

```
x = randn(10000)
hist(x, 100)
```

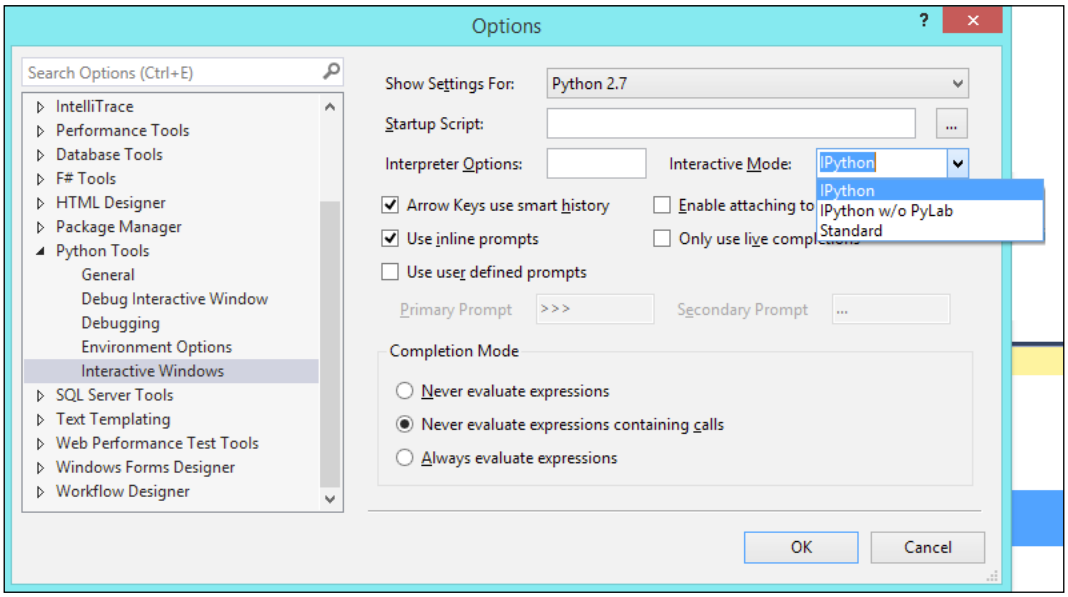
Typing the preceding code into the IPython shell will display the following result:



For more details on what the matplotlib library can do, refer to the library website at `http://matplotlib.org/`.

Now that IPython is up and working, let's instruct PTVS in a way that will allow REPL to talk to IPython to extend it along with its plotting capabilities. First, we need to find the Python interpreter's REPL options. You can quickly locate this from the **Python Environments** window by clicking on the **Interactive Options** label in the Python environment that is being used.

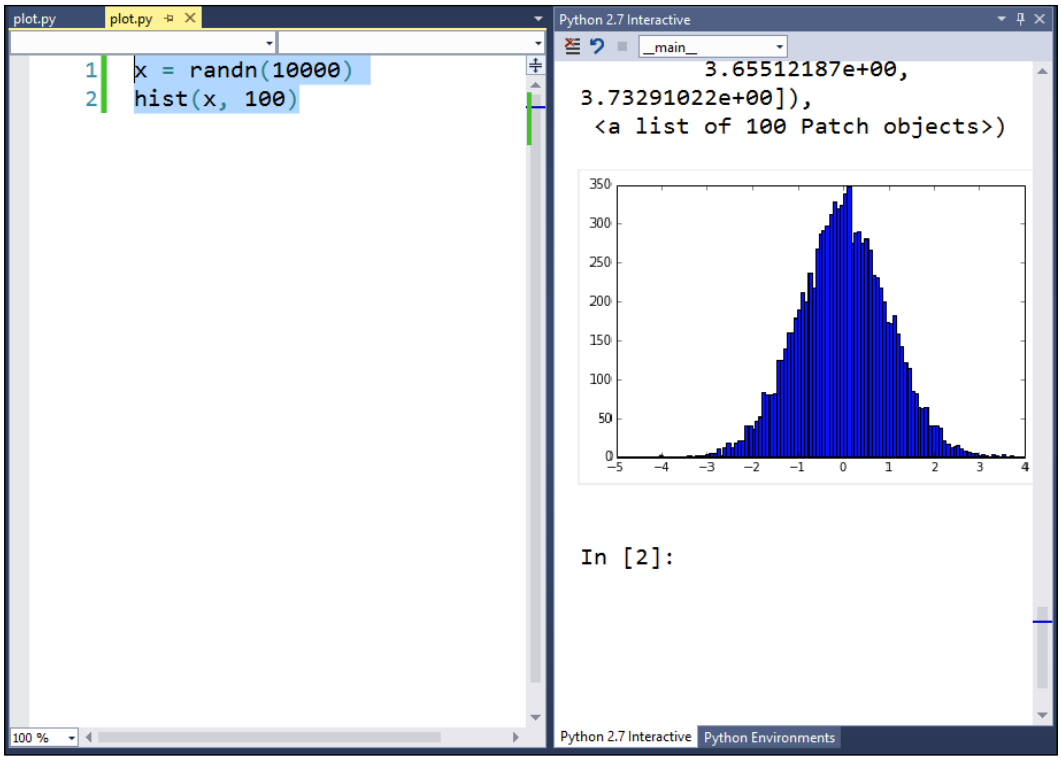
This will bring up the **Options** dialog box, as shown in the following screenshot:



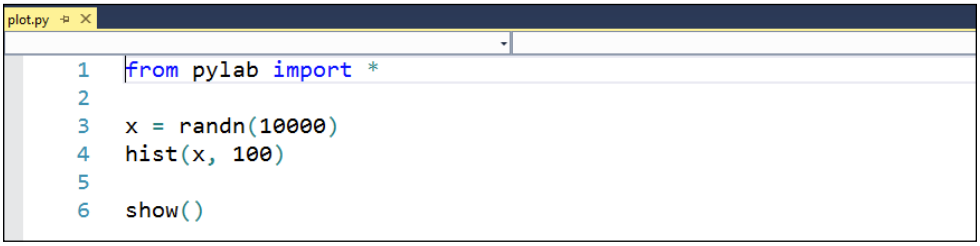
In the **Interactive Mode** drop-down menu, the different modes of the PTVS REPL tool are listed as follows:

- **Standard**: This mode offers the default REPL interactive window in which we can execute the Python code
- **IPython**: This mode permits us to see the graphs directly inside REPL when REPL interacts with the PyLab library directly
- **IPython w/o PyLab**: This mode permits us to see the graphs that are rendered in an independent window

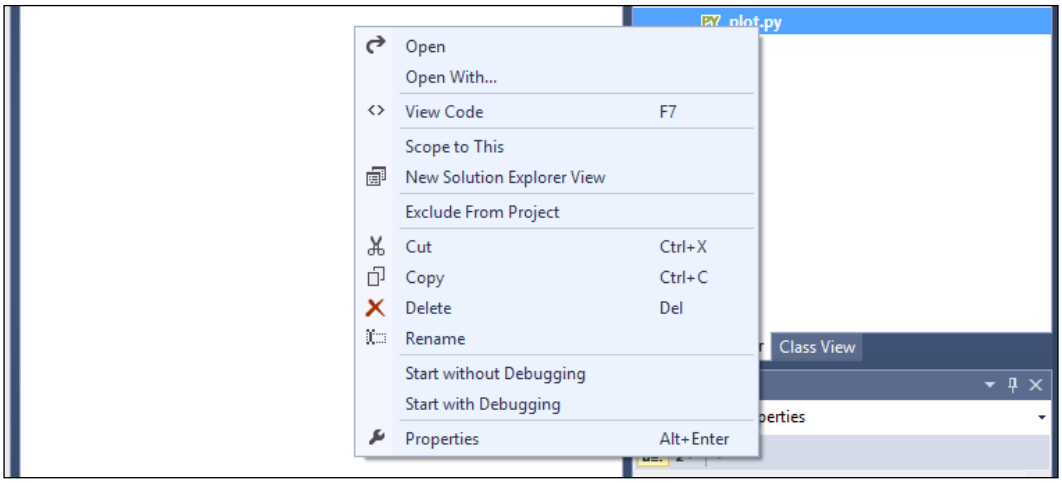
For our example, we are choosing the IPython mode. Now that we have set up **Interactive Window**, let's see how Visual Studio acts. Instead of writing the code of our previous example inside **Interactive Window**, you can write it in the code editor and then execute it in REPL through the **Send to Interactive Window** option in the contextual menu. As we have seen in *Chapter 2*, *Python Tools in Visual Studio*, the following is the result to be expected:



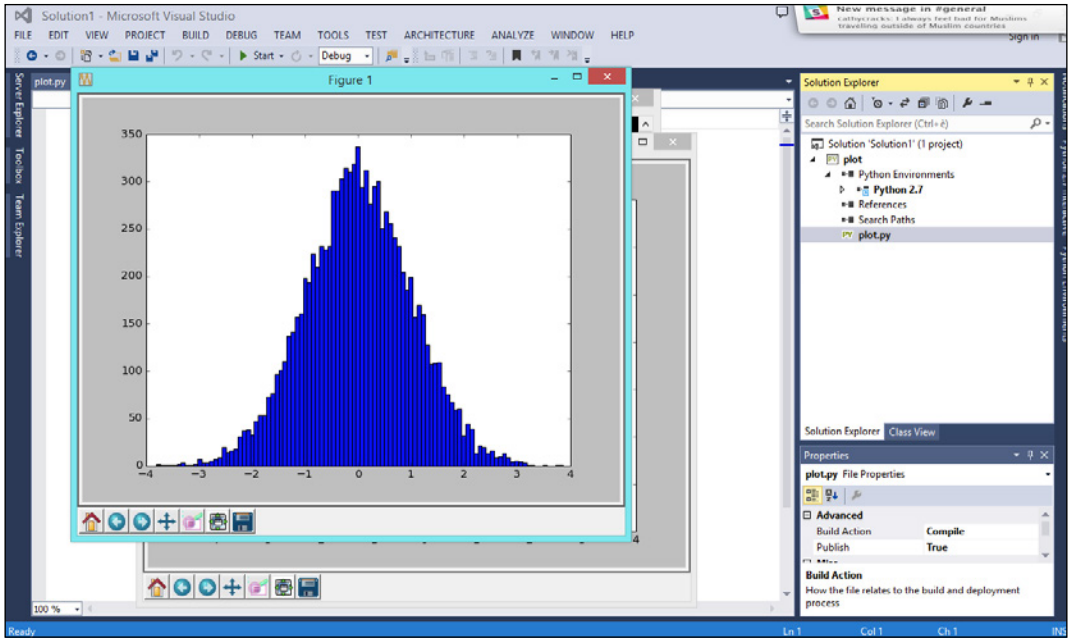
We can also execute the code in the file in a way through which we'll only see the resulting graph in an external window. However, before we can do this, we need to add some other code as follows:



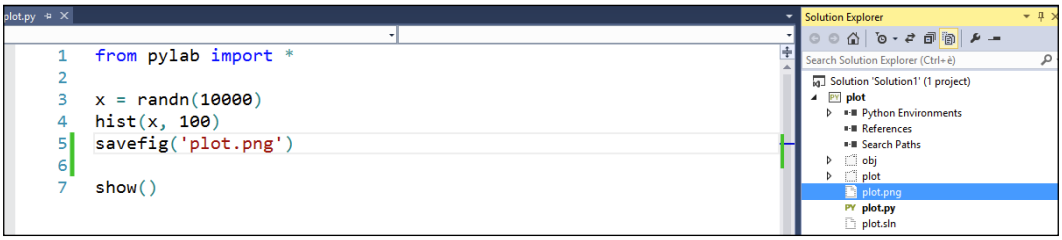
The first line in the preceding screenshot will reference the `pylab` libraries, and the last line will render the plot. To run the code in an external window, right-click on the file node in the **Solution Explorer** window and select the **Start without debugging** command as shown in the following screenshot:



This will execute the code in the console, and the resulting rendering window will appear at the end of the execution:



The matplotlib library also offers the possibility to save the resulting plot into a file with just a line of code by adding the `savefig` command as follows:



In this example, the resulting graph will be saved as a `plot.png` file in the root of the project folder.

For more in-depth functionalities and to dig deeper into the plotting functionalities offered by the IPython integration, please refer to the IPython website at `http://ipython.org/`. You can also refer to the matplotlib website at `http://matplotlib.org/`, which contains great documentation on this subject that are correlated with examples.

The only limitation of IPython is the fact that it cannot be used with the other library, IronPython, that we are going to explore right now. IronPython currently does not support IPython, despite the fact that you can select it on the **Interactive Options** form.

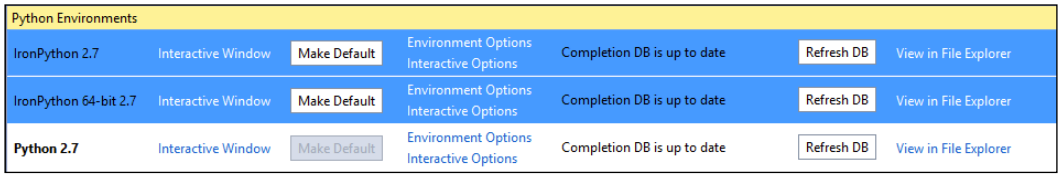
IronPython

IronPython is an open source implementation of the Python language which is tightly integrated with the Microsoft .NET framework. This means that you can use the .NET libraries through IronPython in your Python applications or use Python scripts inside .NET languages.

Using .NET classes in Python code with IronPython

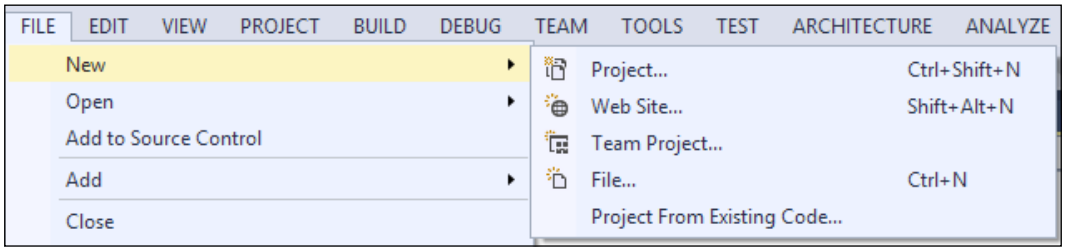
To set up IronPython in PTVS, first we need to download the IronPython installer from the official website, `http://ironpython.net/`.

Once you download the version of the package for your operating system, install the package by double-clicking on it. After the installation is complete, you will see that a new interpreter is available in the **Python Environments** window as follows:

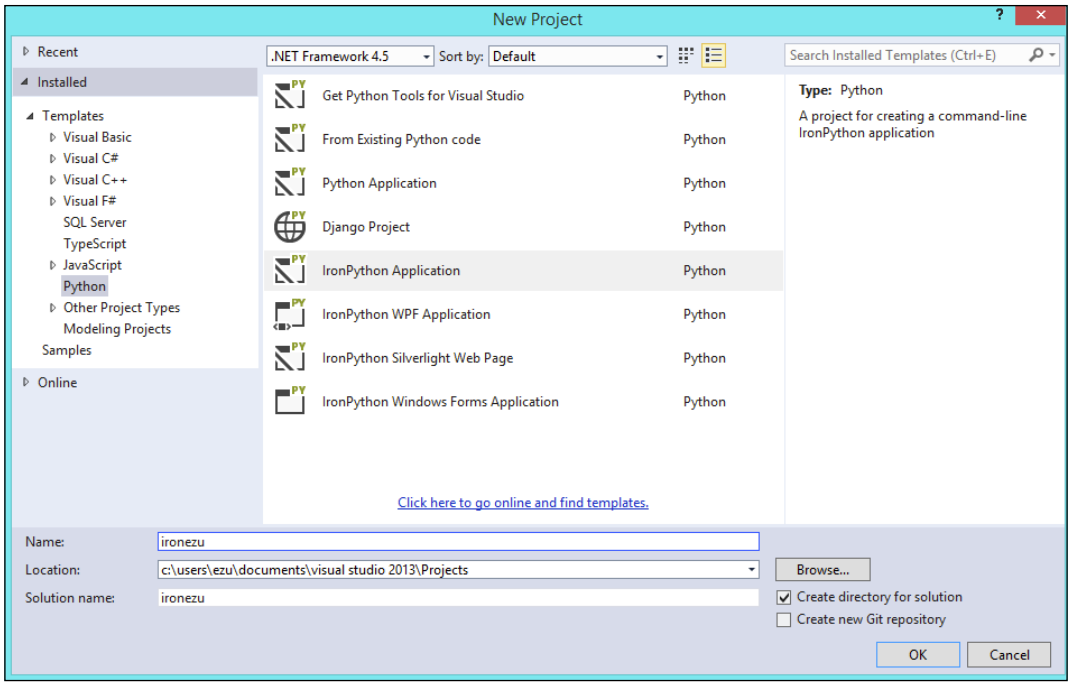


In the example shown in the preceding screenshot, there are actually two interpreters. This is because the 64-bit version of IronPython is installed, which results in the installation of both the 32-bit and 64-bit versions on the machine.

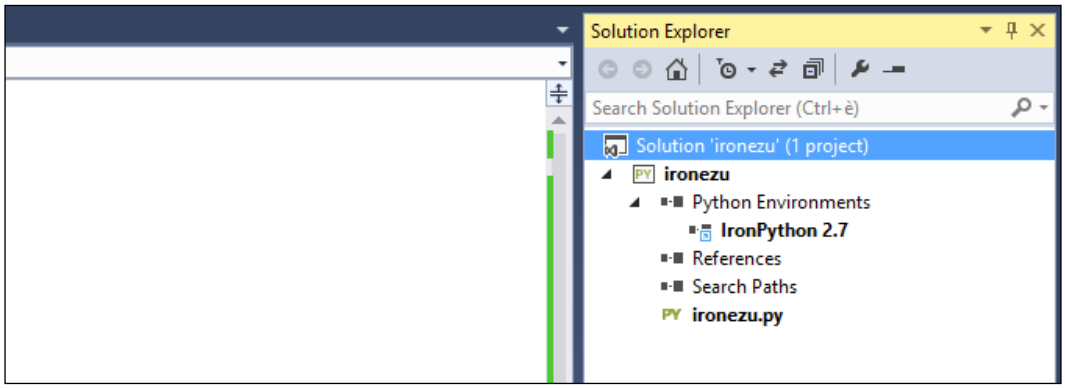
Let's try to create a new IronPython project to learn how to interact with the .NET libraries from Python. Navigate to **File** | **New** | **Project** to create a new project:



This will open the **New Project** dialog box. Select the **IronPython Application** template and assign a project name to it:



This will create a normal Python project, except that the environment for the project will be IronPython instead of Python.



You can find IronPython indicated as the environment in Solution Explorer

Now you can access .NET libraries from inside the Python applications. The system's .NET namespace is referenced by default, so we can start using the elements inside it to see how to interact with the base classes in Python.

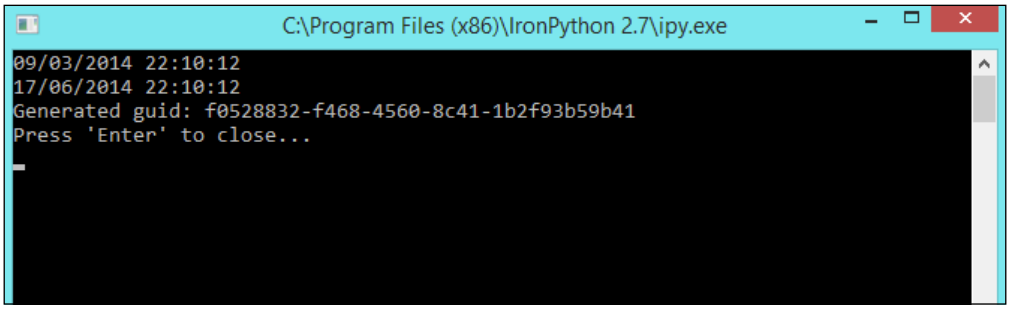
As an example, let's see how to create a **Globally Unique Identifier** (**GUID**), play with the `date` function, and then print them out to the console; we're doing all of this using Python by accessing the .NET classes.

```
1  from System import Console, Guid, DateTime
2
3  a = Guid.NewGuid()
4  b = DateTime.UtcNow
5  c = b.AddDays(100)
6
7  Console.WriteLine(b)
8  Console.WriteLine(c)
9  Console.WriteLine("Generated guid: {0}",a)
10
11 Console.WriteLine("Press 'Enter' to close...")
12 Console.ReadLine()
13
```

Example of using .NET classes inside Python

As shown, we imported the `Console`, `Guid`, and `DateTime` .NET objects and used them in the code to create a new GUID object (line **3**), get the current UTC date and time (line **4**), and add 100 days to it (line **5**). After this, we used the .NET console object to print out the result (line **7** to **11**) and waited for the user to press the *Enter* key to close the application. Obviously, we could have used the normal Python `print` command instead of the `Console` object to print out the result. However, since there is no distinction between Python and the .NET code with IronPython, we used the `Console` object for the sake of seeing different object integrations in action.

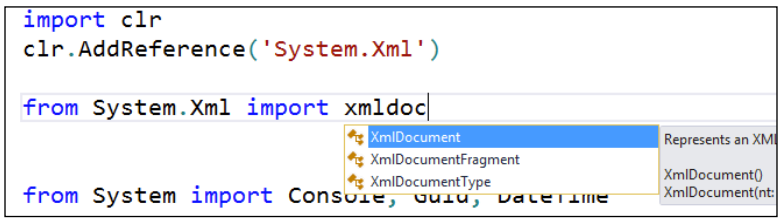
The execution of the code will provide us with the following result:



We can also take advantage of other .NET namespaces outside of the Core System assembly. For example, if we want to use the `System.Xml` assembly, which is a .NET core library that is installed in the **Global Assembly Cache** (**GAC**) of the system, all we need to do is to load it in our code using the load functionality of the `clr` module as follows:

```
import clr
clr.AddReference('System.Xml')
```

Now it can be referenced in the code, and the IntelliSense functionalities become available:



.NET types are exposed as Python classes, and you can do many of the same operations on .NET types as with Python classes. In either case, you create an instance by calling the type. Even for complex types, `XmlDocument` for example, you don't need to instantiate it as you do in .NET; it will be done by the IronPython runtime under the hood.

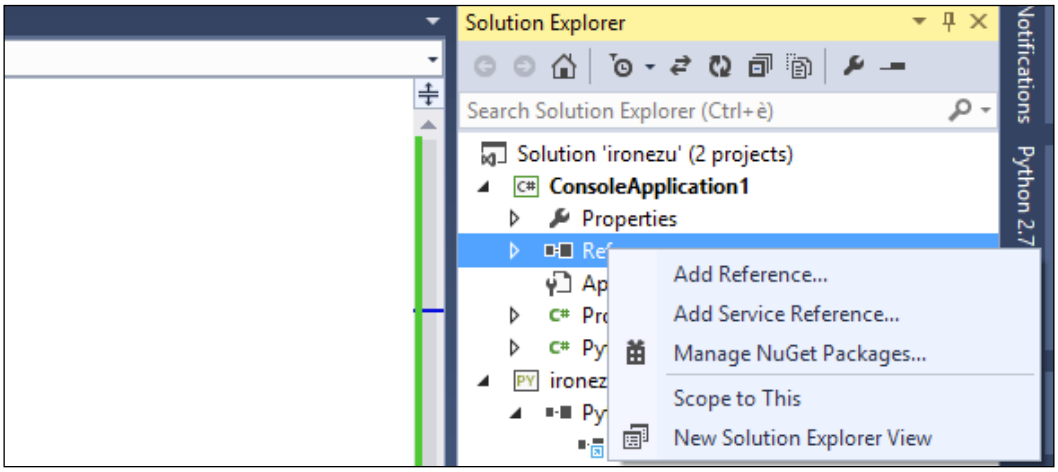
```
1  import clr
2  clr.AddReference('System.Xml')
3
4  from System.Xml import XmlDocument
5  from System import Console
6
7  xml = '''
8  <doc>
9     <node>
10     <subNode>content</subNode>
11    </node>
12 </doc>
13 '''
14 doc = XmlDocument()
15 doc.LoadXml(xml)
16
17 Console.WriteLine(doc.FirstChild.OuterXml)
18 Console.WriteLine("Press 'Enter' to close...")
19 Console.ReadLine()
20
```

An example of using the XmlDocument class in Python

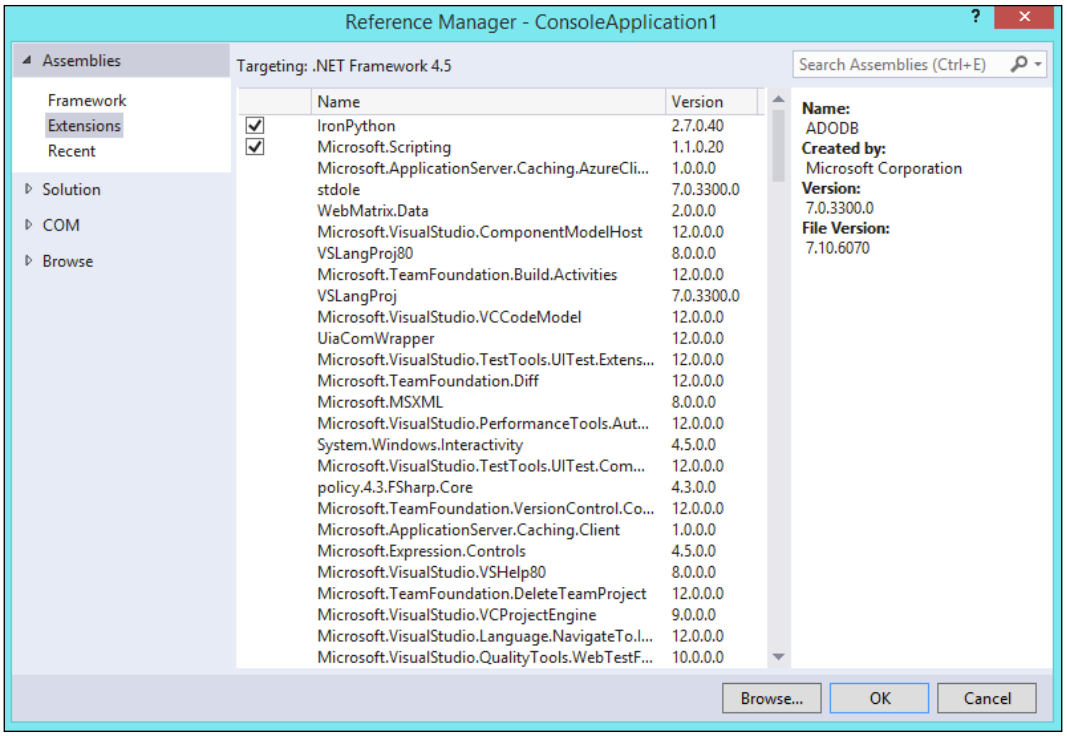
Using the Python code in .NET with IronPython

So far, we have learned how we can interact with .NET classes from the Python code; now let's take a look at how to use Python inside our .NET code.

To start, let's create a new C# console application. To be able to run the Python code from your .NET code, you need to reference two assemblies that are necessary to add the integration functionality for our .NET application: `IronPython` and `Microsoft.Scripting`. To add a reference to an assembly in a .NET application, right-click on the **Reference** node of the **Solution Explorer** window and select the **Add Reference** menu item:

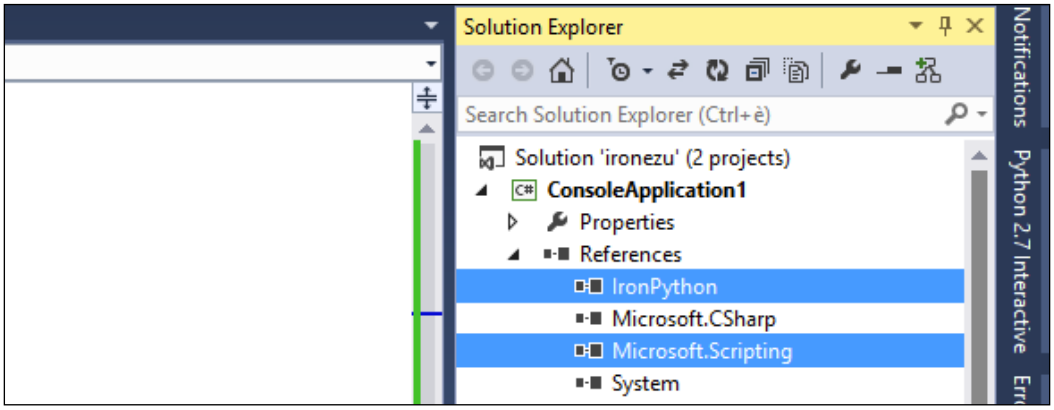


This will open the **Reference Manager** dialog window. The two assemblies that we need are located in the **Extensions** list, which can be activated by clicking on the tree view on the left-hand side:



Once the two assemblies are selected from the list by placing a tick in the checkboxes next to them, click on **OK**. The references to these assemblies are made in the project.

You will see them listed in the **Reference** list in the **Solution Explorer** window as shown in the following screenshot:



Now let's create a new class in our project that contains the code for our Python integration:

```
1  using IronPython.Hosting;
2  using Microsoft.Scripting.Hosting;
3
   1 reference
4  class PythonExecute
5  {
       1 reference
6      public static void Execute()
7      {
8          // Instantiate the Python scripting engine
9          ScriptEngine engine = Python.CreateEngine();
10
11         // Python script to execute
12         string theScript =
13 @"
14 def PrintMessage():
15     print 'This is a message!'
16
17 PrintMessage()
18 ";
19
20         // Execute the script
21         engine.Execute(theScript);
22     }
23 }
```

This code will create `scripting engine` for Python (line **8**), define the string that contains the Python code to be executed (lines **12-18**), and then execute the Python script. Pay special attention to the string that contains the Python code. It has to be indented correctly; otherwise, the interpreter will return an indentation error.

To run the code and see the result in the console, add the following code into the `Program.cs` file:

```
1  using System;
2
3  namespace ConsoleApplication1
4  {
       0 references
5      class Program
6      {
           0 references
7          static void Main(string[] args)
8          {
9              PythonExecute.Execute();
10
11             Console.ReadLine();
12         }
13     }
14 }
15
```

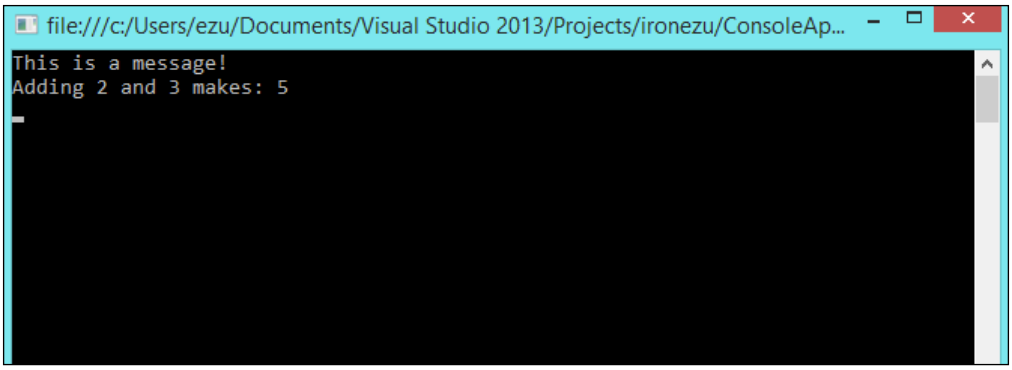
This will execute our function defined earlier and expect the user to press *Enter*. Run the application to see the following result:

You can also call variables and functions defined in .NET applications and use them inside the Python code. To do this, we need to define a scope and pass it as an argument to the `Execute` method in a way that the interpreter can pass those elements to the Python code.

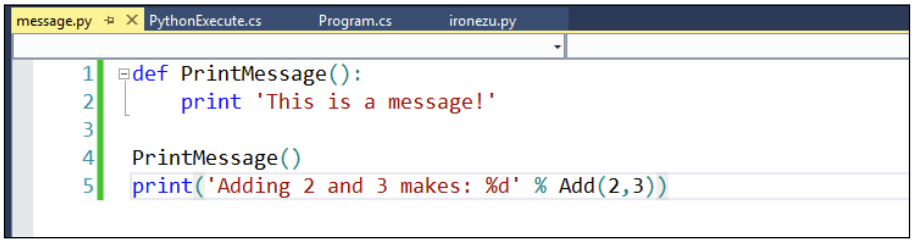
Extend our previous `Execute` method by adding a scope that contains an `Add` function:

```
7       public static void Execute()
8       {
9           // Instantiate the Python scripting engine
10          ScriptEngine engine = Python.CreateEngine();
11
12          // Adds a scope to the engine and adds a new function to it (Add)
13          dynamic scope = engine.CreateScope();
14          scope.Add = new Func<int, int, int>((x, y) => x + y);
15
16          // Python script to execute
17          string theScript =
18  @"
19  def PrintMessage():
20      print 'This is a message!'
21
22  PrintMessage()
23  print('Adding 2 and 3 makes: %d' % Add(2,3))
24  ";
25
26          // Execute the script
27          engine.Execute(theScript,scope);
28      }
29  }
```

We created a scope and the `Add` function with a lambda function (lines **13** and **14**); then, we added a new Python command (line **23**) that invokes this function. Finally, we executed the Python code and passed the scope variable to the script (line **27**). Executing the program will display the following result:



In our last example, we will see how to execute a code that comes from an external file. Let's say that we have a Python file that contains the following code, which is actually the code we had as a string in our last example:



This is how we can execute the file from inside our .NET application:

```
1 reference
30  public static void ExecuteFromFile()
31  {
32      // Instantiate the Python scripting engine
33      ScriptEngine engine = Python.CreateEngine();
34
35      // Adds a scope to the engine and adds a new function to it (Add)
36      dynamic scope = engine.CreateScope();
37      scope.Add = new Func<int, int, int>((x, y) => x + y);
38
39      // Execute and store variables in scope
40      engine.ExecuteFile(@"C:\message.py", scope);
41  }
42 }
```

In the example, we define the scripting engine and the scope. Instead of defining and executing the Python code from inside the .NET code, we are loading it from an external file, passing the scope to the interpreter, and executing it (line **40**).

The possibilities offered by integrating Python code into .NET applications are really endless. Sharing the scope variables with the interpreter opens up a possibility to use existent Python libraries from inside the .NET applications or use Python as a scripting language inside our application.

Summary

In this chapter, we looked at two ways to extend PTVS and Python in Visual Studio in general along with two powerful tools: IPython and IronPython. IPython is more related to plain Python language and IronPython is more integrated with the Microsoft .NET framework.

Both tools show new ways to use and interact with Python, providing new frontiers to explore with this powerful language; all made possible from inside Visual Studio and PTVS.

With this chapter, our voyage to explore the Python tools in Visual Studio ends. We tried to show Python developers the power of Visual Studio and the amount of automatism and help that the Microsoft IDE offers; we also explored and learned the possibility of using Python as a language to create new powerful applications.

Besides the tools themselves, we also went through the possible problems and workarounds of using Python libraries on the Microsoft Windows operating system. We also looked at the topic of exploring Django in Visual Studio and also some of the powerful libraries it offers to accelerate and manage the application's life cycle.

We have only scratched the surface, but we hope that this book has provided you with a deep insight into PTVS and has sparked the curiosity for you to go deeper and explore more.

Happy coding!

Index

Symbols

$cls command 22
$load command 22
$mod command 22
$reset command 22
--initial parameter 86
.NET
  Python code used, with IronPython 100-105
.NET classes, using
  in Python code, with IronPython 95-99

A

Add function 103
Add Virtual Environment command 68
admin interface
  setting up 61-63

B

bar method 20
breakpoints 47

C

code
  navigating 24-28
CodePlex
  URL 8
coding tools 33
Create button 69
CREATE STORAGE ACCOUNT button 67
CREATE WEB SITE button 66

D

database
  managing, for Django project 58-60
  setting up, for Django project 58-60
date function 98
debugging
  about 46
  breakpoints using 47
  watch entries, utilizing 48, 49
Django
  South, using with 80
Django app command 63, 83
Django project
  database, managing for 58-60
  database, setting up for 58-60
  deploying, on Microsoft Azure 65-71
  URL 51
Django project template
  about 52
  application, running 55-57
  IntelliSense, using 57
  Python package, installing 53-55

E

Execute method 103
Extract Method 45

F

Fabric library
  about 75-78
  URL 75
Find All References command 25

G

Global Assembly Cache (GAC) 99
Globally Unique Identifier (GUID) 98
Go To Definition command 25

I

Import button 70
Include in Project command 60
installer
URL 90
IntelliSense
mastering, with Python 17-21
IPython
about 89
URL 95
used, in PTVS 89-95
IronPython
about 95
.NET classes used, in Python code with 95-99
Python code used, in .NET with 100-105
IronPython installer
URL 95

M

manage.py command 56
matplotlib library
URL 91
Microsoft Azure
Django project, deploying on 65-71
Microsoft Windows Azure
URL 65
migrate command 85, 86

N

Name property 58
new Django application
creating 63, 64

O

Object Browser tool 28-30
Object-relational mapping (ORM) 80

P

package distribution
URL 90
pip
advantages 73, 74
print command 98
project handling
about 33-37
Python environments, specifying 37-41
Search Paths, defining 41, 42
solution 33
project templates 34
PTVS
about 34, 39
configuring 7-11
installing 7-11
IPython, using 89-95
Visual Studio panels, using with 14, 15
PTVS CodePlex
URL 8
PTVS tools
Python Environments window 13
Python Interactive window 14
Publish command 69
Python
IntelliSense, mastering with 17-21
Python code
.NET classes, using with IronPython 95-99
Python code, using
in .NET, with IronPython 100-105
Python documentation
on Windows Azure, URL 71
Python environments 37-41
Python Environments window 13
Python Interactive window 14
python manage.py runserver command 56
Python Tools in Visual Studio. *See* **PTVS**

R

range method 18
read-eval-print loop. *See* **REPL**
refactoring 42-45
Refresh DB button 13

REPL
about 13
used, in Visual Studio 21-24
Run button 55

S

savefig command 95
schema migration
used, with South 83-86
schemamigration command 84
Search Paths
defining 41, 42
Send to Interactive command 23
Solution Explorer window tool 35
South
installing 80-82
key features 79
schema migration, using with 83-86
URL 86
using, with Django 80
SQLite
URL 58
Start button 11
Step Into 49
Step Out 49
Step Over 49
sync command 58
sync_db command 84

V

View all files command 60
Visual Studio
debugging tools 46
project handling 33
REPL, using 21-24
Visual Studio panels
used, with PTVS 14, 15

W

watch entries
utilizing 48, 49

Thank you for buying

Python Tools for Visual Studio

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

About Packt Open Source

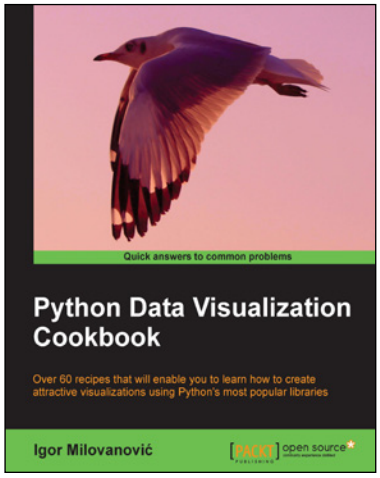
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

[PACKT] open source
community experience distilled
PUBLISHING

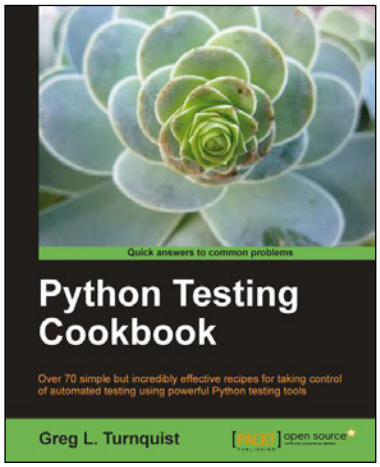


Python Data Visualization Cookbook

ISBN: 978-1-78216-336-7 Paperback: 280 pages

Over 60 recipes that will enable you to learn how to create attractive visualizations using Python's most popular libraries

1. Learn how to set up an optimal Python environment for data visualization.
2. Understand the topics such as importing data for visualization and formatting data for visualization.
3. Understand the underlying data and how to use the right visualizations.



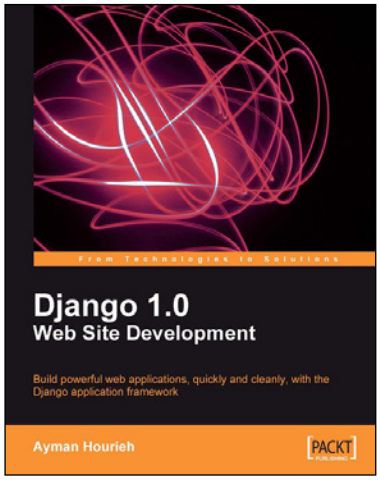
Python Testing Cookbook

ISBN: 978-1-84951-466-8 Paperback: 364 pages

Over 70 simple but incredibly effective recipes for taking control of automated testing using powerful Python testing tools

1. Learn to write tests at every level using a variety of Python testing tools.
2. The first book to include detailed screenshots and recipes for using Jenkins continuous integration server (formerly known as Hudson).
3. Explore innovative ways to introduce automated testing to legacy systems.

[PACKT] open source
community experience distilled
PUBLISHING

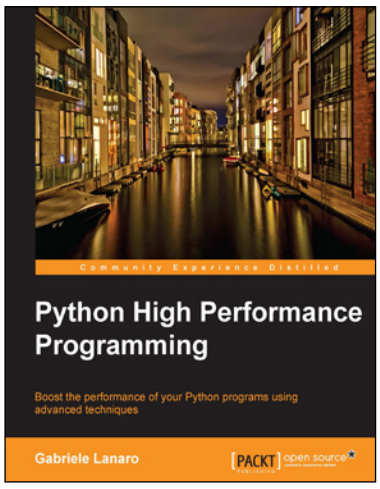


Django 1.0 Web Site Development

ISBN: 978-1-84719-678-1 Paperback: 272 pages

Build powerful web applications, quickly and cleanly, with the Django application framework

1. Teaches everything you need to create a complete Web 2.0-style web application with Django 1.0.
2. Learn rapid development and clean, pragmatic design.
3. No knowledge of Django required.
4. Packed with examples and screenshots for better understanding.



Python High Performance Programming

ISBN: 978-1-78328-845-8 Paperback: 108 pages

Boost the performance of your Python programs using advanced techniques

1. Identify the bottlenecks in your applications and solve them using the best profiling techniques.
2. Write efficient numerical code in NumPy and Cython.
3. Adapt your programs to run on multiple processors with parallel programming.