IronPython Tools for Visual Studio Walkthrough

IronPython Tools for Visual Studio provides Python programmers with a great development experience inside of Visual Studio. IronPython Tools for Visual Studio (IronPython Tools for short) is a set of components that extends Visual Studio. It supports editing, debugging, creating Python projects, and participating in VS solutions.

At this time, IronPython Tools is just a prototype. We are handing it out to a few customers to get feedback. This walkthrough takes you on a tour of features to familiarize you with the IronPython Tools Prototype. Feedback about IronPython Tools can be reported publicly to the IronPython Mailing List or privately to ironpy@microsoft.com.

Installation

IronPython Tools for Visual Studio installs as a Visual Studio extension (VSIX). To use IronPython Tools for Visual Studio you'll first need to install Visual Studio 2010. IronPython Tools work with either the free integrated shell or with the Professional, Premium, or Ultimate editions. If you don't already have Visual Studio 2010 installed you can download the integrated shell or a free trial edition.

Once Visual Studio is installed you can double click IronPythonTools.vsix to install the extension into Visual Studio. You're now ready to start using IronPython Tools for Visual Studio – just start up Visual Studio and start editing some Python code!

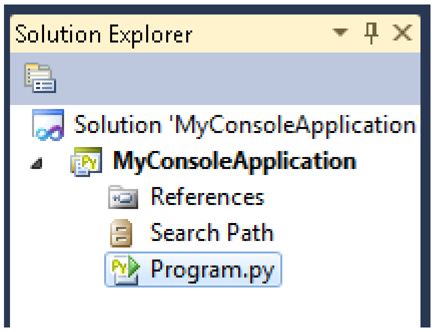
Getting Started

Upon starting Visual Studio you can either proceed with or without a project file, while still getting code analysis over related code. To start working with a project you can choose New Project from the start page or File->New Project. Select the appropriate project type and select a location to create the project. For more information about the project types see the Projects section below.

To start working without a project you can just open any .py files using File->Open. IronPython Tools provides completion, parameter tips, go to definition, etc., on all the code in the directory containing the file you opened, as well as sub directories.

Projects

IronPython Tools supports the Visual Studio project system. Project files for IronPython are .pyproj files. Like all Visual Studio languages, Python project files are referenced from a solution which can contain one or more project files from varying languages. There is a primary difference between the IronPython project system and the many Visual Studio project systems. IronPython project items are determined by the files that are on disk in the directory containing the project file, or in sub directories.



Projects include a start-up file which is the file that runs when you start with debugging or execute your project in the interactive window. This file is indicated by the green play button on the file icon. You can change this file either by right clicking and selecting "Set as Startup File" or by setting the startup file in the project's properties.

IronPython Tools includes 4 project types:

Console Application

This is a simple application or script which is launched as "ipy.exe Program.py". The program will have a console window for input and output. If you use Execute in IronPython Interactive, input and output occur in the interactive as if it were a console window. NOTE: in the prototype release, there is a bug, and input does not work.

IronPython Silverlight Web Page

This is an IronPython application which will run in the web browser using Silverlight. Your script code is written in a .py file that a web page includes using a <script type="text/python" src="…"> tag. A boilerplate script tag pulls down some JavaScript code which initializes IronPython running inside of Silverlight. From there your Python code can interact with the DOM.

WPF Application

This is an application which is launched as "ipyw.exe Program.py". This program will not have a console window and includes a template which creates a Windows Presentation Foundation application. The code and design are separated with the markup being stored in a .XAML file. There is a WYSIWIG designer which enables drag and drop editing of the design. Python code can be used to wire up event handling.

WinForms Application

This is an application which is launched as "ipyw.exe Program.py". The program will not have a console window. A typical usage of this is to start a WinForms application. IronPython Tools currently does not provide any WYSIWIG support for developing WinForms applications. Instead you'll need to write the logic to create the UI in Python (see the IronPython tutorial for examples).

Lightweight Usage Project-free

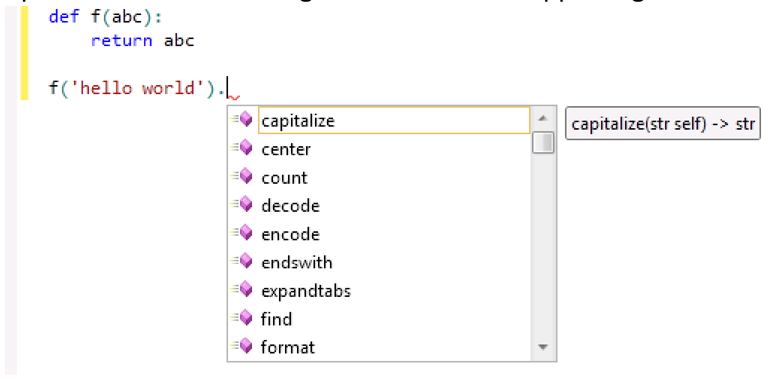
IronPython Tools also supports editing your code without a project system. You can open any files on disk and start editing them. IronPython Tools will automatically include other files and packages in the same directory in its analysis, so you'll get member completion and help without being required to create a project file.

The Editor

One of the primary experiences when working with IronPython Tools is of course editing the source code. IronPython Tools includes syntax color highlighting, outlining of classes and functions, and drop down navigation. There is an analysis engine that understands your classes and functions using type inference. This enables you to receive member completion and signature help, receive tool tips when hovering over source code, and quickly navigate and find references within your source code.

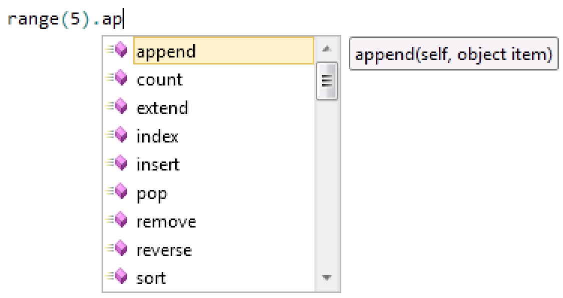
Intellisense

Some of the primary features of the editor are support for rich member completion, signature help, and quick info when hovering over members. Supporting this is a new analysis engine which infers types from their usage to provide useful information to the developer. The type inference is control flow independent and works across function, class, and module boundaries and is updated in real time while you're developing your application.



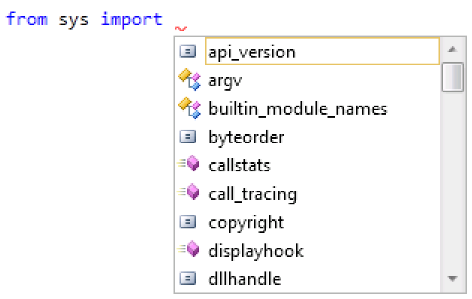
```
def f(abc):
    return abc

f('hello world').
```

IronPython Tools also is able to provide significant intellisense against standard Python built-in functions and .NET classes by relying upon the strong typing available for .NET members. To IronPython Tools these are both identical problems – so the same great intellisense you'll get against normal .NET classes you'll also get against our implementations of various Python core built-in members.



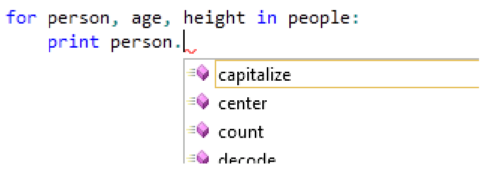
```
range(5).ap
```

You'll also find you get completion when you're dotting through members and also when you're importing modules, or members from inside of a module. In this example you can also see that the intellisense engine is aware of the members and also of the types of members, so you get a quick visual clue for the member type that you're looking for.



```
from sys import 
```

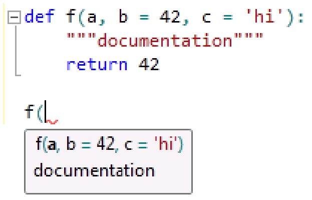
These are all fairly simple examples so far, but the analysis engine is capable of some sophisticated tricks. For example, it's common to use tuples or lists to store homogeneous data and



```
for person, age, height in people:
    print person.
```

then either iterate over those lists or index into them using global constants. The analysis engine is capable of tracking the types of indices and providing intellisense on a per-index basis.

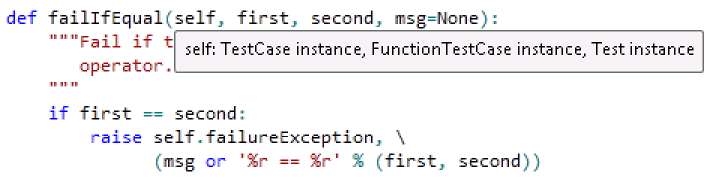
So far we've just been talking about member completion which is extremely useful, but the editor can offer more information than just that. One common form of additional information is signature help.



```
def f(a, b = 42, c = 'hi'):
    """documentation"""
    return 42

f(
```

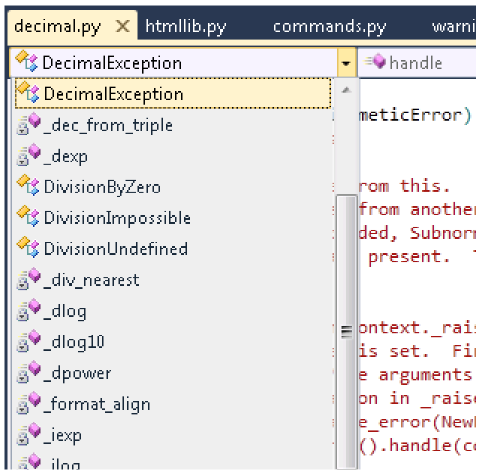
Here IronPython Tools captures any relevant doc strings from the function you're calling and provides information about default parameters as well. When calling a .NET function that has multiple overloads, you'll also be able to iterate through the available overloads using the arrow keys. Of course, you don't need to use the arrow keys – we'll try to automatically select the correct function based upon the number of parameters.

Other times you're just browsing source code, and you're curious what some variable could be holding as a value. IronPython Tools provides rich information about what an expression can possibly represent when you hover over it.



```
def failIfEqual(self, first, second, msg=None):
    """Fail if t
       operator.
    """
    if first == second:
        raise self.failureException, \
              (msg or '%r == %r' % (first, second))
```

self: TestCase instance, FunctionTestCase instance, Test instance

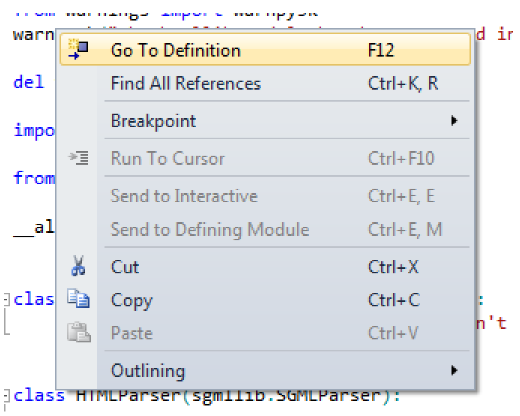


Navigation Bar

When working with a large file or class it's often useful to quickly jump to a class or function without needing to search through the entire document. Visual Studio has traditionally provided a navigation bar in the editor's view. It enables you to quickly jump around the document. The left hand drop down allows you to select top-level classes and function definitions while the right hand drop down allows selecting methods or nested classes within the current class. It's as simple as clicking on the drop down and selecting where you want to go.

Go To Definition

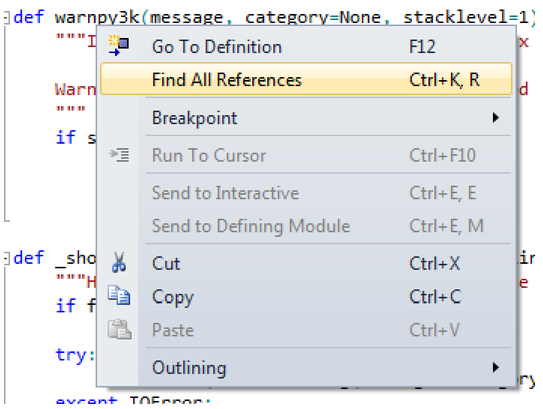
One very handy tool when looking at source code is to be able to quickly navigate to a function definition from a caller to understand what it's going to do. For any expression you can right click and select "Go To Definition" or press F12. IronPython Tools will analyze what the current expression is, and

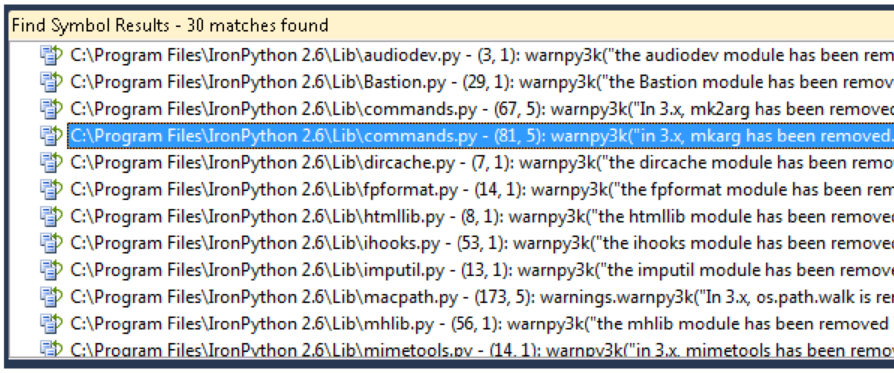


if the expression resolves to a single function, IronPython Tools immediately navigates to that function. If there are multiple definitions, the tool displays choices in the Find Symbol Results window, where you can double click on any definition to navigate to it.

Find All References

Another useful tool when refactoring or working with a large code base is to find all the references to a function or a class. IronPython Tools provides this functionality through the context menu's Find All References or via the Shift-F12 hot key. Based upon the analysis of the code, all of the known references will be displayed in the Find Symbol Results along with the line of code where the reference appears. You can then double click on each line to immediately navigate to the nearest reference.

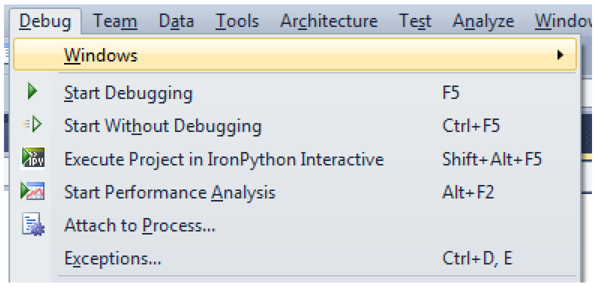


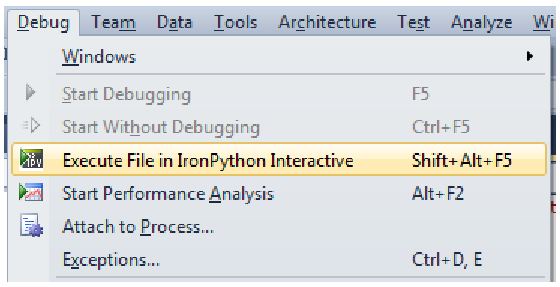


```
Find Symbol Results - 30 matches found
C:\Program Files\IronPython 2.6\Lib\audiodev.py - (3, 1): warnpy3k("the audiodev module has been rem
C:\Program Files\IronPython 2.6\Lib\Bastion.py - (29, 1): warnpy3k("the Bastion module has been remov
C:\Program Files\IronPython 2.6\Lib\commands.py - (67, 5): warnpy3k("In 3.x, mk2arg has been removed
C:\Program Files\IronPython 2.6\Lib\commands.py - (81, 5): warnpy3k("in 3.x, mkarg has been removed.
C:\Program Files\IronPython 2.6\Lib\dircache.py - (7, 1): warnpy3k("the dircache module has been remo
C:\Program Files\IronPython 2.6\Lib\fpformat.py - (14, 1): warnpy3k("the fpformat module has been rem
C:\Program Files\IronPython 2.6\Lib\htmllib.py - (8, 1): warnpy3k("the htmllib module has been removed
C:\Program Files\IronPython 2.6\Lib\ihooks.py - (53, 1): warnpy3k("the ihooks module has been removed
C:\Program Files\IronPython 2.6\Lib\imputil.py - (13, 1): warnpy3k("the imputil module has been remov
C:\Program Files\IronPython 2.6\Lib\macpath.py - (173, 5): warnings.warnpy3k("In 3.x, os.path.walk is re
C:\Program Files\IronPython 2.6\Lib\mhlib.py - (56, 1): warnpy3k("the mhlib module has been removed
C:\Program Files\IronPython 2.6\Lib\mimetools.py - (14, 1): warnpy3k("in 3.x, mimetools has been remo
```

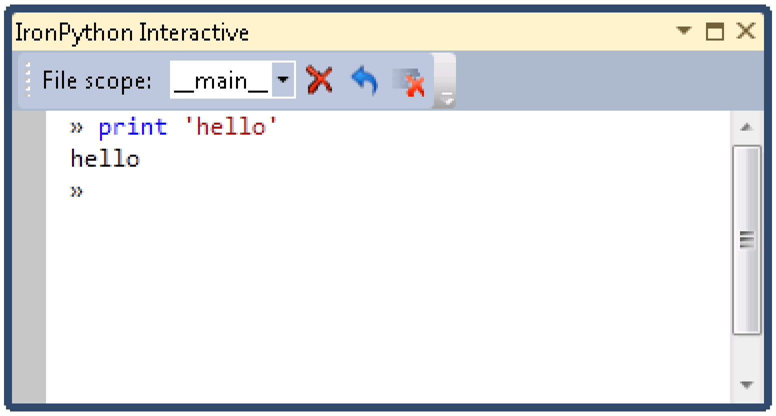
Working with the Interactive Window

IronPython Tools for Visual Studio includes an interactive window for developing your code – commonly called a REPL (Read/Evaluate/Print/Loop). The interactive window supports executing files or the start file of your project. The text of the menu item will change depending on if you have a project opened or not. When in a project the command will be called "Execute Project in IronPython Interactive". When working with a loose set of files the command will be "Execute File in IronPython Interactive":


Debug Team Data Tools Architecture Test Analyze Window
Windows
Start Debugging F5
Start Without Debugging Ctrl+F5
Execute Project in IronPython Interactive Shift+Alt+F5
Start Performance Analysis Alt+F2
Attach to Process...
Exceptions... Ctrl+D, E


Debug Team Data Tools Architecture Test Analyze Wi
Windows
Start Debugging F5
Start Without Debugging Ctrl+F5
Execute File in IronPython Interactive Shift+Alt+F5
Start Performance Analysis Alt+F2
Attach to Process...
Exceptions... Ctrl+D, E

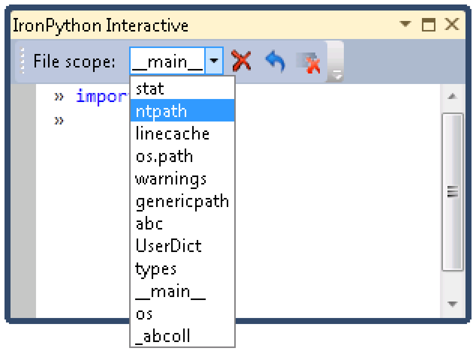
You can also bring up the interactive window without executing any code with the Tools->Other Windows->IronPython Interactive menu item (or Alt-I). This opens the interactive window if it is not already open and sets focus on it. If you've previously used the interactive window for executing a script or for other interactive development, the contents will still be available for your perusal:



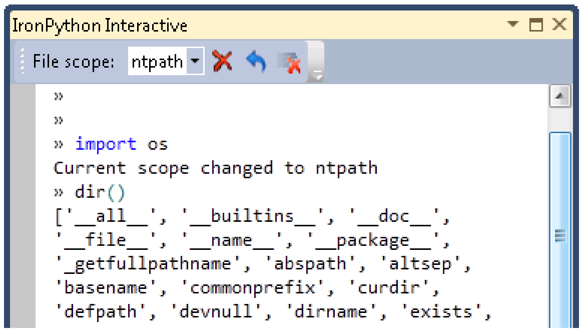
```
» print 'hello'
hello
»
```

Switching Scopes

The interactive window when initially started is in the scope for the __main__ module. If you bring up the interactive window without starting a file, the module is empty. If you start a file or your project, the module contents is the starting file, as if you ran it from the command prompt. You can also view other scopes which contain user code and switch amongst them to execute code within those modules. For example, after typing "import os" a number of modules execute, and you can now switch between them using the File Scope drop down.

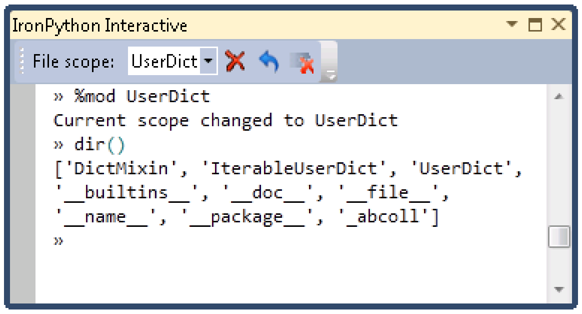


Selecting a module changes the current scope, and input now executes in that module. Whenever you perform any actions such as switching scopes, a log message is written in the output window, so you can keep track of how you got to a certain state during your development.



```
»
»
» import os
Current scope changed to ntpath
» dir()
['__all__', '__builtins__', '__doc__',
'__file__', '__name__', '__package__',
'_getfullpathname', 'abspath', 'altsep',
'basename', 'commonprefix', 'curdir',
'defpath', 'devnull', 'dirname', 'exists',
```

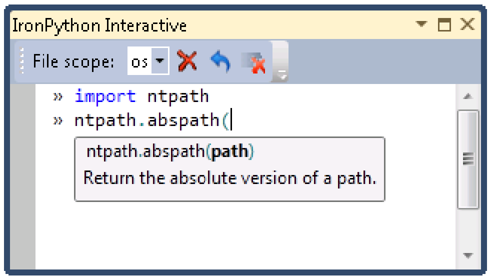
The interactive window also supports several meta commands. All meta commands start with a %, and you can type % or %help to get a list of the meta commands. The most useful meta command is probably the command that switches between modules without using the drop down box. This command is the %mod command and is followed by the module name you'd like to switch to. Other commands include clearing the screen, running a file



```
» %mod UserDict
Current scope changed to UserDict
» dir()
['DictMixin', 'IterableUserDict', 'UserDict',
'__builtins__', '__doc__', '__file__',
'__name__', '__package__', '_abcoll']
»
```

which can include % commands as well, or resetting the interactive process. The commands are also extensible via MEF (the Managed Extensibility Framework for .NET).

The interactive window includes intellisense based on the live objects rather than by analyzing the source code. This differs from the editor buffers of .py files in that analysis of the code in the REPL window is not necessary. The benefits are that you always get good completion. . The drawback here is that functions which have side effects may impact your development experience. To control this behavior there are few different options in the Tools->Options->Editor->IronPython->Advanced menu. Here you can choose to never evaluate expressions, never evaluate expressions containing calls, or always evaluate expressions. In the never evaluate expressions mode the normal intellisense engine will be used for discovering possible completions. In the never evaluate expressions containing calls simple expressions such as “a.b” will be evaluated but expressions involving calls, such as “a.b()”, will use the normal intellisense engine. The idea here is that simple member accesses are unlikely to have side effects. Finally always evaluate expressions will execute the expression to get completion information regardless of the expression

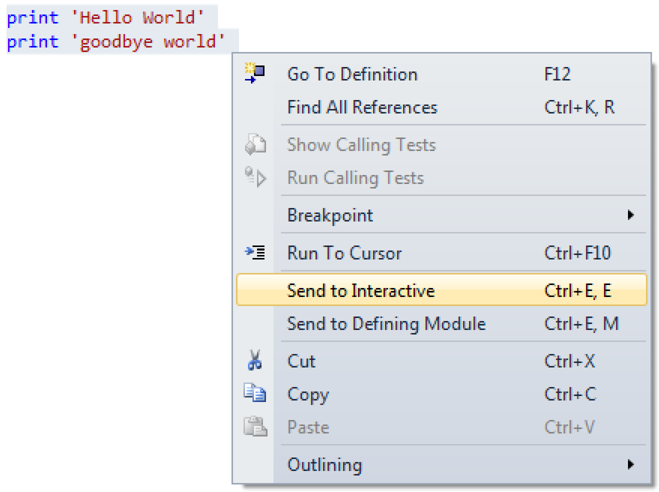


```
» import ntpath
» ntpath.abspath(
```

ntpath.abspath(**path**)
Return the absolute version of a path.

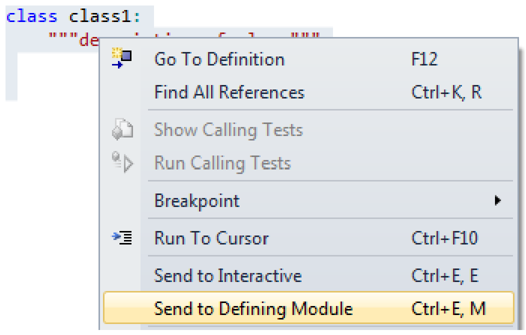
Sending Code to Interactive

In addition to working within the interactive window there are commands available that send selected code from the editor to the interactive window. This lets you work within the interactive window, update code in the editor, and then quickly send the updated code to the interactive window.



```
print 'Hello World'
print 'goodbye world'
```

In addition to simply sending the code to the current scope in the interactive window there is also a separate command which sends the code to the defining module. This command will search the interactive process to find the module that matches the current file being edited. If the command finds the correct module, then it uses the %module command to switch to that module. The %module command becomes part of the history of the interactive window so that the interactive buffer remains as a comprehensive history of

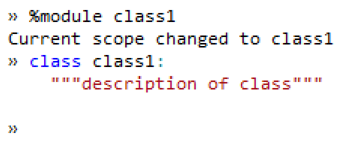


```
class class1:
    """de
```

Go To Definition F12
Find All References Ctrl+K, R
Show Calling Tests
Run Calling Tests
Breakpoint
Run To Cursor Ctrl+F10
Send to Interactive Ctrl+E, E
Send to Defining Module Ctrl+E, M

everything you've done. Then the command pastes the selection into the interactive window for evaluation.

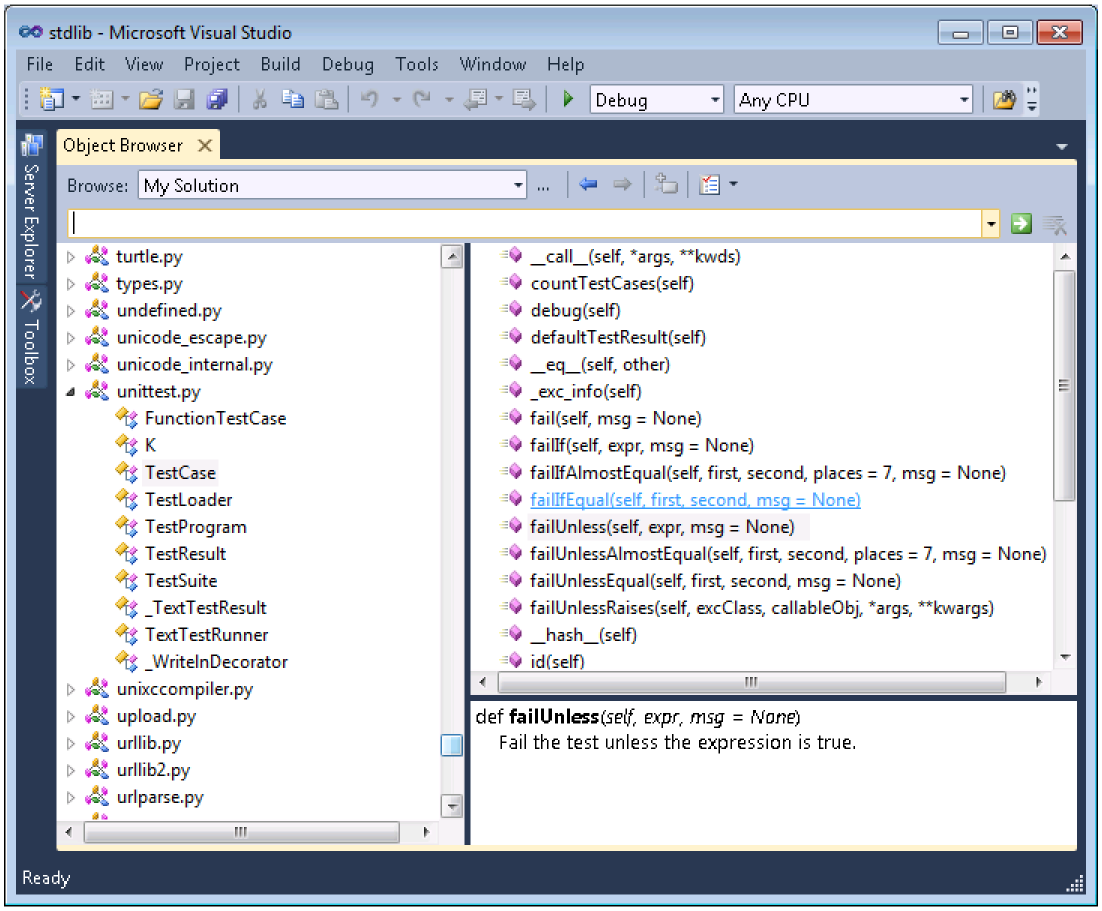
The end result of this in the interactive window is indistinguishable from if you had manually switched to the correct scope and pasted the text yourself:



```
» %module class1
Current scope changed to class1
» class class1:
      """description of class"""

»
```

The Object Browser

When a project is loaded, IronPython Tools supports using the Object Browser for viewing classes defined in each module and the functions defined in those classes. The left hand pane of the Object Browser enables you to view all of the modules defined and drill down into them. As you do so, the right hand side updates to reflect the current class you are browsing, and the bottom right hand pane will show you any information about the method currently selected on the right hand side.



stdlib - Microsoft Visual Studio
File Edit View Project Build Debug Tools Window Help
Debug
Any CPU
Server Explorer
Toolbox
Object Browser
Browse: My Solution
Ready

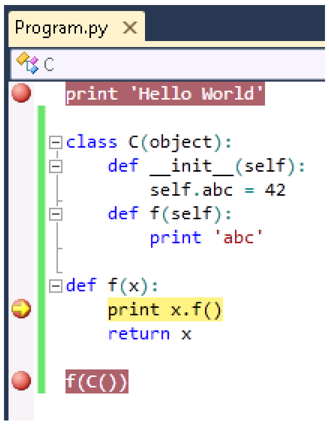
```
▷ turtle.py
▷ types.py
▷ undefined.py
▷ unicode_escape.py
▷ unicode_internal.py
◢ unittest.py
      FunctionTestCase
      K
      TestCase
      TestLoader
      TestProgram
      TestResult
      TestSuite
      _TextTestResult
      TextTestRunner
      _WritelnDecorator
▷ unixccompiler.py
▷ upload.py
▷ urllib.py
▷ urllib2.py
▷ urlparse.py
```

```
__call__(self, *args, **kwds)
countTestCases(self)
debug(self)
defaultTestResult(self)
__eq__(self, other)
_exc_info(self)
fail(self, msg = None)
failIf(self, expr, msg = None)
failIfAlmostEqual(self, first, second, places = 7, msg = None)
failIfEqual(self, first, second, msg = None)
failUnless(self, expr, msg = None)
failUnlessAlmostEqual(self, first, second, places = 7, msg = None)
failUnlessEqual(self, first, second, msg = None)
failUnlessRaises(self, excClass, callableObj, *args, **kwargs)
__hash__(self)
id(self)
```

def **failUnless**(*self, expr, msg = None*)
  Fail the test unless the expression is true.

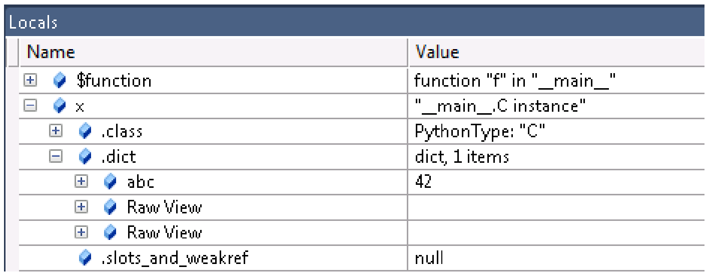
Debugging

IronPython Tools currently supports debugging using the .NET debugger and normal Visual Studio debugging support. This enables you to start your program under the debugger, set break points, step through functions, change the current statement, inspect local variables, and perform other operations that you are used to while debugging.



```
print 'Hello World'

class C(object):
    def __init__(self):
        self.abc = 42
    def f(self):
        print 'abc'

def f(x):
    print x.f()
    return x

f(C())
```

Currently debugging in IronPython Tools is primarily handled by the C# expression evaluator, so you will not get an extremely Pythonic experience. However, you will be able to accomplish basic debugging tasks. For example when broken into a Python function you will be able to inspect local variables, including those pointing to instances of Python classes. When drilling into these objects, you'll see some pseudo-fields such as ".class" and ".dict" which can be expanded to look into the class and instance variables. This is currently a fairly primitive experience but is better than needing to explicitly dig into the IronPython data structures themselves to inspect classes. In future releases of IronPython Tools we will improve this experience so it reflects what the Python programmer would expect to see.



Name	Value
$function	function "f" in "__main__"
x	"__main__.C instance"
.class	PythonType: "C"
.dict	dict, 1 items
abc	42
Raw View	
Raw View	
.slots_and_weakref	null