IronPython Tools for Visual Studio Spec

Bill Chiles and Dino Viehland

This document should be up to date, but some places lack highly detailed descriptions due to the team's common understanding. Details can be added where requested. There are NO GUARANTEES OR PROMISES here.

1 Introduction

IronPython Tools for Visual Studio provides Python programmers with a great development experience inside of Visual Studio and enables Visual Studio developers to automate VS with Python. 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. IronPython Tools also enables scripting VS from IronPython files and an embedded REPL.

IronPython Tools for Visual Studio ships as a Visual Studio extension (VSIX) for developers who already have Visual Studio 2010 installed. You can download the .vsix from ironpython.net. The extension also works with the VS Integrated Shell, which is free. We might also provide a stand-alone download with IronPython Tools for Visual Studio and the Integrated VS shell.

2 Scenarios or Storyboards

2.1 Writing a Script

Jane wants to write a script (single file of code) to execute for a specific task. She is going to copy some code that does a similar task that John wrote for his desktop management scripts. Jane's going to modify the script to suit her purposes. She launches VS and then opens the file of code from John. She sees the code is colorized.

Jane starts editing the file. She enters import statements and gets completion lists of available CPython and IronPython libraries on her path. These completions are based on information from the IronPython engine and Python semantics of search paths. Jane could also set options for the current working directory in which to launch her scripts, and the IronPython features would provide library completions appropriate to that directory. Jane also gets completion while importing .NET assembly namespaces from referenced DLLs. IronPython Tools has default references for projects, and if she adds references to other DLLs, then she gets completion on their namespaces too.

As Jane enters her code, it colorizes as Python code. She enjoys auto-indenting as she hits enter and types code. If the next line's indentation was incorrect (ending a control construct, adding an 'else' instead of another consequent statement, etc), she hits backspace once to go back one indentation level. She gets completion on any identifiers that occur in the buffer by pressing a key that requests completion, and she can select from possible completions if the partial identifier is not a unique initial substring. Also, if she sufficiently qualifies an identifier relative to her imported libraries and .NET namespaces, she gets completion on those identifiers without any popping UI. For example, if Jane imported System, she could type "Sys" and complete "System". She could then type a dot followed by "Env" and complete "Envionment".

If Jane tries to complete a name based on a member access relative to a variable, she gets very good assistance due to our code analysis. The variable's type might be known because it is in a scope (and within a lifetime) where the variable has been set to the result of constructing a type or calling a .NET static method that returns a known static type. Similarly, ignoring that dynamic run time changes to types may invalidate completion information, if a variable were set to the result of constructing a Python type, there might be analysis information for completing the members of that type. For example, the language engine may analyze slot declarations or initialization/constructor functions to determine possible members.

Jane realizes there's part of John's code that does exactly what she wants, and she likes how he's solved the logic and corner cases. She decides to keep those functions just as they are in another library that she'll use from her IronPython script. She cuts all the code she doesn't need from her file and saves it to a directory on her Python path or to the working directory where she'll be running her new script. She adds a new import statement to the top of her IronPython file. As she goes on to write her code she can complete the names from the new library of the functions after adding the import statement.

Of course, while Jane is editing, she has all the basic text permuting, navigation, and searching operations. She can move around in units of lines, characters, and words (constituent characters determined by settings or language engine query). She can delete and select in these textual units as well. She can scroll the script pane up and down over the buffer of text, and she can also scroll from the keyboard so that the caret does not change its location. She uses the latter functionality to bump the pane a line (or several lines) at a time to move the area where she's working to the middle of the pane to see more surrounding context. Because she's used to seeing code in the middle of her pane, she might scroll the pane so that there's several lines of whitespace at the bottom of the pane, even though the end of the buffer is in the middle of the pane where she's typing. Jane can also search through the buffer from the caret's location to the bottom of the buffer (or backwards the beginning), and she can do so optionally replacing find hits with a replacement string.

While writing her code, Jane is mixing totally new code and code fragments lifted from John. She is also commenting her code so that others can readily understand what it does. She pastes in some code from John, and she morphs it easily into what she needs. Jane selects the modified code and uses rigid indenting commands to move the code left or right to fit into her code while keeping the same relative indentation of each line. Jane is happy that John commented his code; however, as Jane moved the code left and right and slightly tweaked the code, she needed to update the comments to be correct and to fit the width to which she formats code. She uses the fill paragraph command (which shortens some lines and lengthens others by inserting and removing newline sequences) so that all lines in comments are as long as they can be without going past the column to which she formats code. Jane uses this command frequently when she updates code or refactors code so that her comments are up to date and formatted to her liking.

Jane is ready to try out some of the code she's written. She knows it is not complete, and she needs to investigate some fragments of code and some objects to see how best to use them. She hits F5 to load her code into a script engine. Jane sees the interaction pane below the script pane echo'ing the reset for her module, as well as any output side-effects of loading her script. The execution engine associated with Jane's program is totally distinct from the one that executes IronPython code against the VS Automation Model. Jane can reset the program's engine state or suffer code crashes, without any effect on VS.

See the next section for an interaction pane scenario or storyboard.

2.2 Dynamically Developing Code in an Interaction Pane

Jane wants to explore some code fragments and objects to finish writing a script she was working on, as well as test some parts of the script. Jane has just loaded her code, and the interaction pane shows a prompt following the load. To load her code Jane could have launched VS, used the command to launch an interaction window, and imported her code at the command line. Alternatively, Jane could have launched VS, opened a file of code, and hit F5.

The only difference in these two approaches is the module context (or scope for name binding) in which the interaction window evaluates. Viewing the REPL window with no file context yields an empty, default global module. Importing Jane's file of code then just creates a global name bound to the module. Hitting F5 in a Python buffer causes the execution engine's to load all the top-level names into the default module. In one case Jane has to qualify global functions in the default module with her module's name (for example, MyScript.Myfunc). In the F5 case, all of Jane's module's global definitions are available without having to qualify them.

Jane wrote a top-level function that is the entry point to her script, and it has the logic to conduct the overall mission of her script. However, she knows she left some of the functions unwritten, and she knows she has some broken code fragments in some of the functions. Jane picks one of the functions to fix, which involves figuring out how to work with a .NET object she's unfamiliar with. Jane needs a result from using this .NET type, which she then passes to a lower level function she already wrote.

Before learning about that .NET type, Jane decides to quickly confirm that her lower-level function works. She types into the interaction window a series of inputs to construct a few objects and link them together (perhaps using a list or using pointers between the objects). When Jane hits enter after a well formed input, the interaction window sends that input to the execution engine for evaluation. Jane invokes the lower-level function, and it works. She changes the objects a few times to make sure the function handles corner cases. Jane both makes calls on the objects to change them and re-evaluates past input that she modifies before hitting enter. Jane can evaluate previous inputs by using the interaction window's history commands, which rotate through previous complete inputs (not a line at a time). She can also move up in the editor buffer in the interaction window and hit c-enter after a line of input. IronPython Tools copies the line or complete input fragment ending on that line to the end of the interaction window so that the copied text is appended to any other input waiting to be confirmed in the interaction window. Jane realizes her helper function is solid. Now Jane is ready to figure out how to get the objects she needs from the .NET object she is unfamiliar with.

Jane navigates (using a key binding) to the Python buffer with her code. She selects part of the code that creates the .NET object. She uses commands to evaluate the selection as though it were input to the interaction window. She doesn't have to evaluate any imports or references to .NET DLLs since she's already loaded her file of code. Jane flips between the script and interaction windows to evaluate code, depending on what she's doing, and the interaction window correctly interleaves input and output so that there is a correct record of the flow of interactions. Each printed result follows the input that produced it. Even if Jane were to paste several complete expressions or statements (with newline sequences after each) into the interaction window, the results for each individual input would immediately follow that input. This is important for Jane to be able to look back at her interactions and make sense of them. It also aids her grabbing several correct lines of input to paste them back into her script code to build a function or to write a unit test.

Let's get back to Jane's playing with the .NET object she just created. First she fetches a list of members on the object (using 'dir' in Python). After skimming the names, she gets help on the type by just entering the type's name (which evaluates to the type) and passing it to a documentation function. The 'help' function provides her with a summary of the type so that she sees short descriptions for the members. Jane uses the help function on specific members to get summary, remarks, example text, etc., that's available in the doc comments or built-in help for the members that may come from Python. Jane tries a few members out, just calling them based on the signatures and help. She gets errors, tries again, and iterates until she thinks she knows how to use the object and get the results she needs. Of course, she's also consulted examples in MSDN or the net in general as needed.

While typing code in the interaction window and viewing printed results, member lists, and help, Jane has all the functionality of the editor behind her in terms of coloring, searching, etc. Past input sections of the buffer might be read-only so that Jane doesn't inadvertently change the input and forget what produced the associated output. She can always edit previous input by copying it to a new editor buffer, the buffer containing her script, or to the end of the interaction buffer.

As a side note, completion works in the interaction window. In the REPL window completion is a combination of code analysis and live object inspection. In the future, IronPython Tools may support live object inspection in editor buffers, especially when the engine is stopped at a break point with appropriate stack frame and local bindings info.

As Jane figures out code fragments she needs and how to fix up all her code, she pastes code into the Python buffer from the interaction window, re-indents the code, adds comments, reformats the comment paragraphs, etc. She selects whole functions she's fixed up and evaluates those back into the script engine behind the interaction window. She sees all that input with associated output properly ordered in the interaction window. She's also been writing little helper functions in the interaction window or in her script's file, as well as creating and re-creating various data structure fragments or objects to help her play with and test her code. Some of this she saves for unit testing or future development helpers.

After fixing up the lower-level functions, Jane is ready to try her code from the top down. She could just call her top-level entry point function, but she thinks her current environment is pretty cluttered and patched together at this point. She decides to get a fresh result and hits F5 in the Python buffer. This resets the module in the execution engine completely, losing global variable bindings, ephemeral definitions of helper functions defined in the interaction window, etc. The interaction buffer still contains all her work for the VS session, or from the last %cls command.

Before hitting F5, Jane also added some code at the end of her script to call the top-level entry point when her script is loaded, as opposed to just defining the function but never calling it in the script. For example, in Python, Jane would add something like the following:

```
If __name__ == "__main__":
    DoMyMainThang(argv)
```

Jane's script has a bug in it, and the interaction window (after some text about resetting the module) displays the prompt for the command line debugger. See the next section for scenario/storyboard information on debugging.

2.3 Debugging In-situ, Recursively in the REPL

These behaviors may be on a tools options. We won't have this support for quite a while, but we should work with pdb.py soon.

Exact debug command names are not spec, just examples.

Jane has loaded a file of IronPython code into the interactive window by hitting F5 in a Python buffer associated with the file. While running the code, she hit a bug. The interactive window displays the debugger's prompt. Jane enters "bt" to get a backtrace printed. She sees a terse printing of each stack frame relating to script code or .NET calls script code made. The frame print representation is essentially the function's name and the arguments passed to it.

Jane moves up the stack a few frames and types "l" to see locals for that frame. This is enough for her to see the problem. She enters a line of code to get a few values out of objects that a couple of locals are bound to, and her hypothesis is correct about what's wrong. Code she enters at a debugger prompt binds variables in her expressions to locals in that stack frame first, then to globals, etc. She enters "q" to quit the debugger and return to the interaction window's top level prompt, cleaning up the call stack on the way.

Jane navigates (using a key binding) to the Python buffer for her code and fixes the bug. She selects the function and uses the Python buffer's command to evaluate the selection, which redefines the function in its module. She often works with the code side by side with the interaction window, so she sees the selected text pop into the interaction window as input to get evaluated.

Jane navigates (using a key binding) back to the interaction window. She calls the initialization function for a global variable and sets the global to the result so that the code will run fine. She then calls her top-level entry point again. This time the code runs farther, but she hits another error. That's fine, she knew she was going to have to shake down the code. This time she realizes a couple of frames up that there was a function called with two arguments passed in the wrong order. She goes to that frame and enters "dbg_return(<expr>)", where <expr> is a call to the function. Jane uses the locals to get the values but places the arguments to the function in the right order.

The execution engine evaluates the argument to dbg_return to get one or more return values. Dbg_return unwinds the stack and causes the return value(s) to be returned from the stack frame where Jane called dbg_return. This let's her keep running her code without having to re-run it from scratch and re-establish global state again. That's a big time savings since there could have been several prompts or steps for her to get to this failure point again. Jane navigates back to the Python buffer to fix the code she worked around in the debugger. She saves the file, but she doesn't bother re-evaluating this function now since it doesn't get called again.

As an aside, if Jane evaluates code from the Python buffer while the execution engine is running some input, IronPython Tools queues the input. We could just report that the interaction window is busy. If there is an evaluation in the queue, and the interaction window's current execution hits an exception, IronPython Tools dumps the queue of input to make sure it doesn't affect the debugging state before Jane can look at it.

After some edit/test cycles, the code works on simple data. Jane decides to go all out and calls her entry point function again on complex data. This time nothing appears to happen. Jane types ^c to interrupt the program, and she lands in the debugger. She enters "bt" to see the stack backtrace, and she looks at the line number for where the stack frame is executing. She goes to her code to look there, and she's in the middle of a pretty simple loop. Jane goes back to the interaction window and pokes around at some locals, drilling into a couple of the objects a few levels. She decides to step through some code from where she stopped. As she suspected, she's infinitely looping, and she's confirmed why.

Jane fixes that problem. She goes on to try various tests and make sure her code is running well. She saves some of the interactions to fold into a few test cases for regression testing her code in the future.

2.4 Customizing Visual Studio with IronPython Code

Still need to think about what we want to do for IronPython Tools. The strike through text is from a long ago prototype.

2.4.1 MEF helpers

2.4.2 Macros/Funs -> cmd/key binding

~~Jane wants to customize Nessie, and she sees from the documentation that she can create an init file for herself that overrides defaults. She uses a command that opens init files, and it can create init files too. This command is different than the normal Open File command because it creates a Python buffer that modifies the script environment that's running Nessie's commands code. Jane can either create the init file in the settings and documents app data directory for Nessie, or she can set the environment variable NESSIE_INIT so that Nessie finds the file wherever she puts it. She copies some settings from the default init file (or an example of a default init file) and changes settings she copied. For now, Jane just changes the default indentation level from four spaces to two spaces, and sets the paragraph fill column to 78 characters.~~

~~After using Nessie a while longer, Jane really misses having delete-horizontal-whitespace and open-line commands. She goes back to the init file and sets the path variable for where Nessie looks for command files. She appends to the path a directory where she keeps tool customizations. Jane then uses a new file command that opens a new Nessie script file for JSX. She also use the Nessie script open command to open one of the files of commands that came with Nessie. These new and open commands open files so that evaluations use an interaction window that affects the Nessie commands script environment. Jane pokes around in the file of commands that came with Nessie to find examples that inform what she's trying to do. She writes the two top-level functions in her new file and saves it.~~

~~Jane doesn't want to restart Nessie to fixup its path, so she uses the Nessie customization interaction window. She evaluates the line that sets the commands search path. Jane does not have to do anything else to pick up the new commands for two reasons. First, setting the path causes Nessie to look for new directories and files since it read commands files last. Second, Nessie notices changed directories and files and asks to reload them, which Jane could turn off if she were making several modifications to an existing file and didn't want Nessie trying to reload the file on every save.~~

~~Jane tries out the commands, and she didn't get them quite right. She goes to her new commands file and uses a command that sets the Nessie customization interaction window's context to her file's language and module. Then she switches to the interaction window and plays with code fragments on the command line as well as evaluating selections in her commands file. She figures out how a couple editor primitives work and how to call them to make her new commands work correctly. She saves the file, which causes Nessie to incorporate the new definitions now that the file's directory is on the commands path. Jane does not have to re-evaluate them in the interaction window as she would when writing a normal program.~~

~~Lastly, for today, Jane decides to add key bindings for her commands, so she decides to do that in her init file rather than the commands file. This way if someone else wants to take her commands file, they can decide on their own key bindings. When Jane saves the init file, Nessie does not reload it because often changes in init files are not functionally independent and can have cumulative effects (for example, appending the same path over and over to the commands path). Jane copies the two lines of code that bind her new commands to keys into the interaction window, and she evaluates them to get key bindings.~~

3 Features

Features are tagged with priority and or staging indicators.

- **"Pycon"** indicates the feature will be working completely or to some extent by 22 FEB 10.
- **"P1"** means the feature will be working before we declare a v1 release. When we can feel solid enough and feature rich enough to declare a v1 depends on other team priorities and our resources.
- **"P2"** means we'll get to the feature as soon as possible, but we don't think we need it to declare a v1.

The primary use of IronPython Tools for Visual Studio is writing Python code, so the early focus needs to be great editing with code modeling or IntelliSense® features. IronPython Tools's editing experience should be similar to other tools in Visual Studio. For example, code is colorized, member completion works, parameter tips pop after typing a function name and an open parenthesis, outlining works, and so on.

3.1 Pycon: IntelliSense® Completion

IronPython Tools for Visual Studio should have the best code modeling of any python tool. This along with a great editor/REPL interaction experience will set IronPython Tools apart from others. Our code analysis will model all possible return values from functions, calls to function that inform understanding return types, assignments, and many other aspects of the code. The analyzer will determine little about return values for highly dynamic code that uses __getattr__ and __getattribute__. For uses of statically typed (.NET library) code we infer most types.

A quick glance at other tools suggests they are not very rich in their code modeling. Many seem to be lacking for basic member completion; some just provide any member name on any and all objects. Some tools don't analyze arguments passed to functions when they can reveal return type information. None seem to do any analysis on core data types such as lists and dictionaries, though IDLE provide parameter tips for built-ins. We aim to do better than all of these existing tool experiences.

3.1.1 P1: Completing Members with Multiple Types Possible

There is a design choice for completion of members on parameters, function returns, or variables assigned multiple types. If you view completion as a poor-man's in-situ browser or discovery tool, you want to see all members for all possible types that are known to flow to this point of the code. If you view completion as paternalistic guidance, you want to see only those members for which you know you can call without error.

By default, IronPython Tools only shows the members it knows you can call. If it has no type info at the completion site, there will be no members shown. If there are multiple types it knows can flow to this point, IronPython Tools only shows the intersection of members with the same names on all types.

pycon: There is a tools option to cause IronPython Tools to show all possible members on all possible types that are known to flow into a completion site.

A couple of ideas for providing choice to the user is to use the tabbed capability of the completion pop-up or to use coloring of the items in the list if that's possible.

3.1.2 P2: Optional Explicit Type Declarations

At some point, we may consider some tactics for support optionally declaring types. The mechanism could be stylized comments, no-op assertion calls, etc. If there's any other work around python in this vein that starts to gather interest (PEP 3107), we'll just use that. We could also consider clr.Accepts and clr.Returns decorators. In any event, this is not a top priority for now.

3.1.3 P2: Completion Works in Comments and Strings

This is not implemented yet.

3.1.4 P2: Filter Completion Lists from Type In

Filter the completion list based on what the user has typed. There are several options here, but basically we could mimic C# (initial substring, contains string, pascal casing abbreviations, etc.).

3.2 Pycon: Find All References

Based upon the analysis provided for intellisense it is easy to provide a find all references feature – although the feature may sometimes not find all references.

All navigation in IronPython Tools centers the target line or selected span of text in two cases. The first if part of the span is out of the view of the current window. The second is if the location is on the first or last line of the window. This behavior allows you to see more context around the location without needlessly jerking the display.

3.3 Pycon: Goto Definition

Also based upon the analysis provided for completion, it is easy to provide a goto-definition feature. This feature will sometimes not be able to goto the definition requested because the defining location cannot be determined. In that case there needs to be some user notification that the feature is unavailable. Alternately a mechanism for notifying the user that the feature is available needs to be provided. Definitions in Exec string arguments are never found.

If there are multiple possible defining locations, or the tool cannot disambiguate, it lists all possible locations from which you can choose.

All navigation in IronPython Tools centers the target line or selected span of text in two cases. The first if part of the span is out of the view of the current window. The second is if the location is on the first or last line of the window. This behavior allows you to see more context around the location without needlessly jerking the display.

The tool goes to assignment locations for globals, locals, and class members. For example, if you set class attributes in the __init__ method, we'll go to those locations as the definitions. The tool shows all assignment locations in a disambiguation list from which you can choose where to go.

3.4 Pycon: Editor Navigation Dropdowns

Similar to other languages the left dropdown contains all class definitions in the current document, including nested definitions. The right dropdown contains all function definitions for the currently selected class in the left dropdown. Because IronPython supports top-level functions, the left dropdown also contains all top-level functions. Module scoped variables that are simply assigned to (that is, not defined as a class or function) are in the left dropdown. If there are multiple assignments to the same variable at top level, the dropdown shows them all.

All navigation in IronPython Tools centers the target line or selected span of text in two cases. The first if part of the span is out of the view of the current window. The second is if the location is on the first or last line of the window. This behavior allows you to see more context around the location without needlessly jerking the display.

3.5 P2: NavigateTo Command Support

NavigateTo is a feature that finds identifiers and files based on substring matching and matching just the camel-cased characters in the name. It is a handy, quick navigation feature introduced in VS 10, but it requires special support rather than just using the APIs supplied for completion, object browser, or class view.

3.6 Refactoring

IronPython Tools for Visual Studio initially includes only a couple of the most used refactorings. We will add others based on demand and other priorities.

3.6.1 P1: Extract Method

Extract method is the safest to perform. Lexically we know which references are to locals, globals, or closure variables. Python supports multiple return values, so passing values in and out of the code are easy source transforms.

3.6.2 P1: Rename Local

This feature works within strings and comments since when you're renaming something, you'd like any error reporting strings or comments to reflect the new name. Other languages in VS get this right now too.

This feature is only reliable for locals and parameters. The feature cannot be reliably provided for class members or module globals. There are a few examples. In this code, we may not catch all patterns for this sort of highly dynamic reference:

```
import sys
sys.modules["foo"].SomeGlobal
```

It might be something as simple as importing from a module and redefining something you don't use in that module, which just adds complexity to the code modeling:

```
from Blah import *    #Blah has a def foo ().
...
def foo ...
```

In this code, if you can't know whether to rename SomeMember if starting from the type, or which type to change if starting from the reference site:

```
def f(x):
   if x.__class__ == str:
      return OneThing()
   else:
      return Another()
f(42).SomeMember()
```

We have a few options:

- Provide a version that may rename incorrect call sites.
- Provide a query replace version with UI to preview locations and individually opt them in or out of the renaming. This is a refinement to the first option.

3.7 Pycon: Object Browser and Class View

IronPython Tools for Visual Studio provides type information to the Object Browser and Class View for the domain of its code analysis. See the Project section for how IronPython Tools determined what code to analyze.

3.8 Pycon: Project System

IronPython Tools for Visual Studio does not require an explicit project to get started, as with classic VS overhead. Nor does it require a class and a Main to execute code. You can use c-o to open a file and immediately start getting smart editor features or run the file. When used this way IronPython Tools creates an implicit project to work well in the VS shell model.

You can create an explicit project. This is a very basic and lightweight project system for working with groups of files. There are a few features supported by the explicit project described below.

Consider whether to list all explicitly opened projects side by side in the solution, or whether to next them with folders that mimic the actual disk structures.

3.8.1 Pycon: Implicit Project for Opened File

The implicit project assumes all files in the same directory as the file you opened are part of the project for code modeling purposes. It also includes all files in sub directories. ~~If the directory containing the opened file contains an __init__.py file, then the implicit project also includes files from parent directories until it encounters a parent directory with no __init__.py file.~~

Consider an option for only including sub directories if they contain an __init__.py file.

As to chasing upwards for a root dir with no __init__.py, you can open a higher level directory, and since code modeling all merges together for now, you'll get completion and navigation over everything under the highest directory.

Consider option for not showing the implicit project in the sln expl. Users can just hide the sln explr if seeing it is the issue and no real user model problems.

You can see the implicit project in the Solution Explorer for the file that you opened with the name "ImplicitProject<containing-dir-name>". It lists the folders and files discovered for the code modeling domain. If you already have any kind of project open (and therefore, a solution), IronPython Tools does NOT show the implicit project to avoid dirtying the solution. The code modeling is the same. The tool just does not show the project, and the files appear in the Misc Files.

Consider option to show implicit projects vs. always showing them.

You can explicitly add or remove files from the implicit project. The project in the Solution Explorer immediately reflects these changes. Code modeling adds or removes information based on directory changes too.

When you invoke "Execute File in IronPython Interactive" (alt-s-F5) with an implicit project, IronPython Tools executes the current editor buffer in the REPL after clearing the execution context of the REPL.

You can select an implicit project item to be the start item, like the explicit project has, and use "Execute Start Item in IronPython Interactive" (c-a-s-F5). This command also clears the execution context in the REPL before running the start item of the project. The difference is that IronPython Tools does not persist this choice. You can unselect the start item in an implicit project by right-clicking on the project item or going to the project properties page.

The functionality to choose a start item is not available in the pycon CTP release.

3.8.2 Pycon: Explicit Project

The explicit project is directory-based. The project file does not explicitly list the filenames included in the project. To remain lightweight IronPython Tools projects automatically include all .py files in the project directory. You can explicitly add or remove files from the project. The project in the Solution Explorer immediately reflects these changes. Code modeling adds or removes information based on directory changes too.

Explicit projects do not include parent directories when directories have an __init__.py file. With the explicit project, the model is that there is an explicit root. If you have a directory structure with distinct python package directories, each may have its own project file and show up as parallel projects rather than sub folders.

Consider an option for only including sub directories if they contain an __init__.py file.

Revisit model that each sub directory with a .pyproj file is a parallel project in the solution, and those sub directories are not included in parent projects as sub folders

When you create an explicit project, IronPython Tools writes a .pyproj file in the directory. You can incorporate these projects into solutions, which may include VB, C#, or kinds of projects.

Explicit projects have a start item. This is the file executed when you invoke "Execute Start Item in IronPython Interactive" (c-a-s-F5). As with the implicit project, the REPL's execution state is re-initialized on c-a-s-F5. You can change the start item, and you can right-click to unselect the start item (or going to the project properties page).

Explicit projects also persists various standard options, such as default command line switches for launching ipy.exe. The options persist within multiple build configurations so that, for example, Release runs can support the –O option, and Debug runs can specify –D.

The project file also persists:

- the ipy.exe path for executing code
- the current working directory option

3.8.3 Pycon: UI and Solution Explorer

Projects should have distinct icons for packages versus normal folders.

Should the solution explorer include non .py files in the project items?

Should we include non .py files and support a right-click exclude functionality?

Should miscellaneous open files (that is, outside the project) be shown in Miscellaneous files, parallel projects, virtual folder, etc?

Should not show files marked as hidden.

3.8.4 pycon: CWD for Executing Code

When you invoke "Execute File in IronPython Interactive" (alt-s-F5) on a file in an implicit project, the current working directory (CWD) of the ipy.exe process is the root directory of the project. The root directory of an implicit project is the one containing the file you opened initially, which created the implicit project. Since you are likely hitting alt-s-F5 in the file you opened originally, the CWD is likely the directory containing the file that was active when you hit alt-s-F5.

When you invoke "Execute File in IronPython Interactive" with an explicit project, the current working directory of the ipy.exe process is the root project directory (where the .pyproj file lives). The project property for the CWD, if it is explicitly set, overrides the project's root directory. If there is more than one project, the current working directory is the root directory of the active project.

If you just launch the REPL without using "Execute File in IronPython Interactive" or "Execute Start Item in IronPython Interactive", and you have a project, then the CWD is the project's root directory. If there's more than one project, it is the active project's root directory. If you do not have a project, then the REPL's CWD is the default from the tools options page, see section 3.9.7. Launching the REPL does not mean just showing the window; it must be the first showing when the executing process gets created, or resetting the execution context (%reset).

Projects have an option you can set to control the current working directory. You might do this to run the code you're developing from the location you know you'll be launching the application for real when you are done with it. This option overrides the project's root directory in the description above.

3.8.5 P1: Project Properties

The project properties has a General tab. It has an Application group with three properties:

- Start Item -- defaults to Program.py for our templates.
- Working directory -- defaults to "." indicating the project's root directory where the .pyroj file is.
- Check box for Windows Application -- checked means do NOT launch program in a cmd.exe window.

Debug group with three properties:

- Search paths -- defaults to nothing
- Command Line Arguments - defaults to nothing
- Interpreter Path -- defaults to the path of ipy.exe installed with IronPython Tools

3.8.6 pycon: Templates

IronPython Tools releases with the following templates (all under "IronPython" as a top-level node in the left navigation pane of the New Project Dialog):

- Windows Command Line -- this is just a collection of .py files from a directory. There needs to be a start item to invoke "Execute File in IronPython Interactive" (alt-s-F5) and launch the program.
- Silverlight Page -- has a Silverlight web page using the <script>…</script> support to enable python code. Launches Chiron.exe and then launches the users default web browser on the web page, which Chiron serves. The project launches the browser under the debugger when launching w/ debugging.
- WPF -- contains a Program.py and Program.xaml file. Program.py loads the XAML and runs it as an app. The designer is supported for the xaml file.
- WinForms -- contains a "hello world" WinForms app. There is no designer support.
- Package -- this is a directory template with an __init__.py file

3.8.7 P1: TFS Binding Support

For now, this is just a place holder section for future feature thinking.

3.8.8 P2: Building Projects

The typical experience for developing apps is to just edit code and execute the current buffer, selected expressions from editor buffers, entered expression in the REPL, or the start item for an explicit project. However IronPython Tools supports compiling projects into EXEs. This support uses the existing IronPython compilation support via pyc/clr.CompileModules. It does not support be any form of "static" compilation such as producing .NET DLLs that C# and VB can link and build against.

P3: We could consider a command for compiling to a stand-alone EXE. We could compile IronPython.dll/IronPython.Modules/etc… into .NET modules instead of compiling them as normal DLLs. Then the user code could also be compiled to a module. These could all be link.exe'ed into one .EXE, giving the user a single EXE to deploy.

3.8.9 P2: Miscellaneous Files and Multiple Domains of Code Analysis

Regardless of whether you have an explicit project or an implicit one, there are situations when you open .py files that are not in the current project. Code modeling for that file should pick up a distinct set of files for smart editor features.

If you open a file that is not considered to be in the implicit or explicit project, IronPython Tools do model the code in these file. However, currently there is a single domain of code modeling. It is as if these miscellaneous files are part of the project once they are opened.

The miscellaneous files may be listed in the misc files project or in their own distinct IronPython Tools project. This is TBD depending on VS.

3.9 pycon: Executing Code and the REPL

There are several ways to execute code in IronPython Tools:

- invoke "Execute File in IronPython Interactive" (alt-s-F5) to run the current buffer in an implicit project
- invoke "Execute Start Item in IronPython Interactive" (alt-s-F5) to run the start item of an explicit project
- invoke "Send to IronPython Interactive" to evaluate a selected expression from an editor buffer in the REPL
- invoke "Send to IronPython Module" to evaluate a selected expression from an editor buffer in the REPL after ensuring that the REPL execution context is the same module as represented by the file in the editor buffer
- invoke "Send to IronPython Defining Context" to evaluate a selected method from an editor buffer in the REPL after ensuring that the REPL execution context is the same module represented by the file in the editor buffer. This command also patches the containing class to have the new method definition.
- entering and evaluating expressions in the REPL directly
- launching ipy.exe on a file in classic VS debug mode (doesn't run inside REPL)

The REPL is the focus typically for executing code and iteratively developing and exploring code. IronPython Tools resets the execution environment on alt-s-F5 commands. The module name of the REPL after an alt-s-F5 command is "__main__". You can change the module in which the REPL evaluates input with the %module command. This allows you to executing code more easily in the context of a file that was imported by the main script.

When you execute a selection from an editor buffer, IronPython Tools first issues a %module command in the REPL to set the module correctly for the evaluation. If a file has not executed, but some other files have executed, then attempting to evaluate a selection in that file's buffer results in an error dialog that IronPython Tools cannot execute the evaluation. You first need to import or execute the file so that its module exists in this case. If no files have executed at all, and you evaluate a selection from an editor buffer, the selection evaluates in the default module, __main__.

3.9.1 pycon: REPL History, Input Editing, and Interrupting Execution

The primary REPL focus is basic editing, intellisense, and input evaluation. For interactive development and API exploration the REPL makes it simple and fun to write code (for example, auto-indenting multi-line input). You can easily access libraries and experiment while developing your application. After verifying an expression or function you've iteratively developed, IronPython Tools makes it easy to copy that to an editor buffer for saving in your program, fixing up whitespace and whatnot as needed.

Here are some common commands and bindings used in the REPL:

- **Interrupt Execution** -- bound to c-break. Users can bind it to c-c in Tools Options. On button in tool window toolbar.
- **Previous and Next History** -- bound to uparrow and downarrow, but the caret must be at the end of the current input or EOB.
- **Previous and Next Line** -- bound to with uparrow and downarrow everywhere except the end of the current input or EOB. To edit a multi-line history item or typed input, you first need to use leftarrow or the mouse to navigate around in the input text.
- **Maybe Execute Input** -- bound to enter at the EOB or current input. Executes the input if it is complete (doesn't end with an operator, in parens, in a construct like 'if' or after 'else:", etc.).
- **New Line and Indent** -- bound to enter. See Execute Input for behavior at the end of the current input.
- **Execute Input** -- bound to c-enter inside the current input. Executes the current input regardless of caret position or completeness of the expression.
- **Force New Line and Indent** -- bound to s-enter and always inserts a newline and indents, regardless of where the caret is or whether the previous text in the current input is a complete syntactic form suitable for execution.
- **Copy Input for Execution** -- bound to c-enter when caret is inside or immediately after a previous input region. This copies the input to the end of the buffer and leaves the caret at the end of the input. History is unaffected as if the input were pasted, except the new input does get added as the most recent history item.
- **Cancel Input** -- bound to escape. This cuts the current input region, leaving the caret after the prompt.

If you type in read-only region, the REPL scrolls to the current input region and appends the typed input there.

Users can rebind the history commands to something like alt-p and alt-n to rotate the history ring regardless of caret position (perhaps rebinding up/downarrow to normal nav commands).

3.9.2 pycon: Tool Window Toolbar

The REPL window has a tool bar with buttons for discovering command and easily accessing them with the mouse:

- **help** -- show a list of REPL meta commands
- **cls** -- clears the contents of the REPL editor window without changing execution state or input history
- **echo** -- suppress or unsuppress output to the buffer
- **load** -- executes code from a file as if the code were entered as input and support meta commands (% REPL commands)
- **reset** -- reset to an empty execution engine, keeping buffer contens and input history
- **module** drop down for changing the module in which code executes
- **Interrupt** execution button

3.9.3 P1: Saving Code from the REPL to an Editor Buffer

After exploring code and getting a function or expression to work in the REPL, you want to save that in an editor buffer or project item where you are building you program.

3.9.4 P1: Live Object Completion

This is just a place holder for now.

3.9.5 P1: Command Placements

There is a View -> Other Windows -> IronPython Interactive command to go to the REPL, creating it if necessary.

There is a Tools -> IronPython sub menu. It starts out with "IronPython Interactive". If a .py file is open and has focus, it also has "Execute File in IronPython Interactive". If there is a selection in the editor, it also has "Send to IronPython Interactive" and "Send to IronPython Module". If there is a project with a start item, then there is an "Execute Start Item in IronPython Interactive".

There is a Debug -> Execute File in IronPython Interactive bound to alt-s-F5. If there is an explicit project active, then there is also an "Execute Start Item in IronPython Interactive". If the current file is not a Python file, and there is no active project, the menu item is grey.

There's a tool bar on the REPL tool window with common REPL buttons/commands.

3.9.6 P1: Meta Commands

The following commands can be entered in the REPL to control the REPL itself or for other functionality that is not executing code:

- pycon: %help -- lists % meta commands.
- pycon: %module [module] -- switches the REPL's execution context to the specified module. The module must already have been created. If module is unsupplied, this switches to __main__.
- pycon: %cls -- deletes the REPL buffer's contents and shows a fresh prompt. This does NOT reset the execution state.
- pycon: %reset -- resets the execution context in the REPL.
- %thread {gui|repl} -- switches the thread code the REPL executes code on so that you can dynamically affect WPF and WinForm programs you launch with REPL control.

3.9.7 P2: Tools Options Page for REPL Defaults

There is a tools options page for setting defaults for when the user launches the REPL with no implicit or explicit project open. For example, there may be a default current working directory the user tends to hack in (which would default to the user's home directory).

This is a place holder section for now, haven't put much thought into this.

3.10 Debugging

While the typical usage will be executing code in the IronPython RePL and debugging there, you can launch your code for debugging in a traditional VS style. You can use the Debug -> Execute IronPython Start Item command to launch ipy.exe as a classic VS debugee process. Only applications launched with the –D flag support VS style debugging. In addition to the basic support for stepping, moving the IP, etc., that is available today, IronPython Tools adds good visualization features for Python objects. Visualizations include user-defined types, instances of those types, and core data types (Python's dictionaries, lists, and sets).

P2: IronPython Tools supports in-situ, recursive style debugging in the REPL (such as with sys.settrace). In-situ recursive debugging requires significant support while the normal Visual Studio debugging experience is low hanging fruit given what IronPython already has always had this. The in-situ, recursive style debugging in the REPL needs custom debugging support.

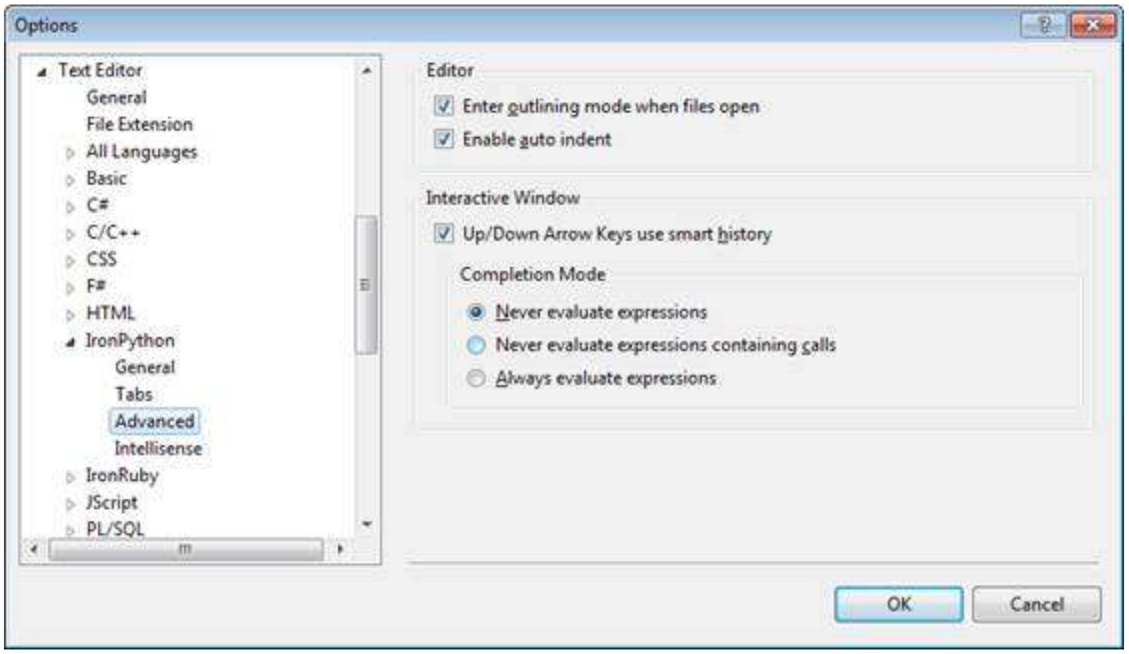
P3: IronPython Tools supports debugging CPython scripts as well. After we support in-situ, recursive style debugging in the REPL for IronPython, then debugging cpython will be easier due to some shared infrastructure.

3.11 pycon: Text Editor Tools Options Pages

There's a tools options page under Text Editor called IronPython. It has the General and Tabs common options that can be specialized per language. It also has an Advanced page with the following options:

- Editor
  - Enter outlining mode when files open -- on by default

    - Enable auto indent -- on by default
- Interactive Window
    - Up/Down arrow keys use smart history -- on by default, means that if you must be at the very end of the input region for up/down arrow to do history, otherwise they navigate in the text.
    - Completion Mode -- Never evaluate partial expressions is the default, but you can select to never evaluation partial expressions with calls or to always evaluate partial expressions to get live objects for completion.



Editor
- [✓] Enter outlining mode when files open
- [✓] Enable auto indent

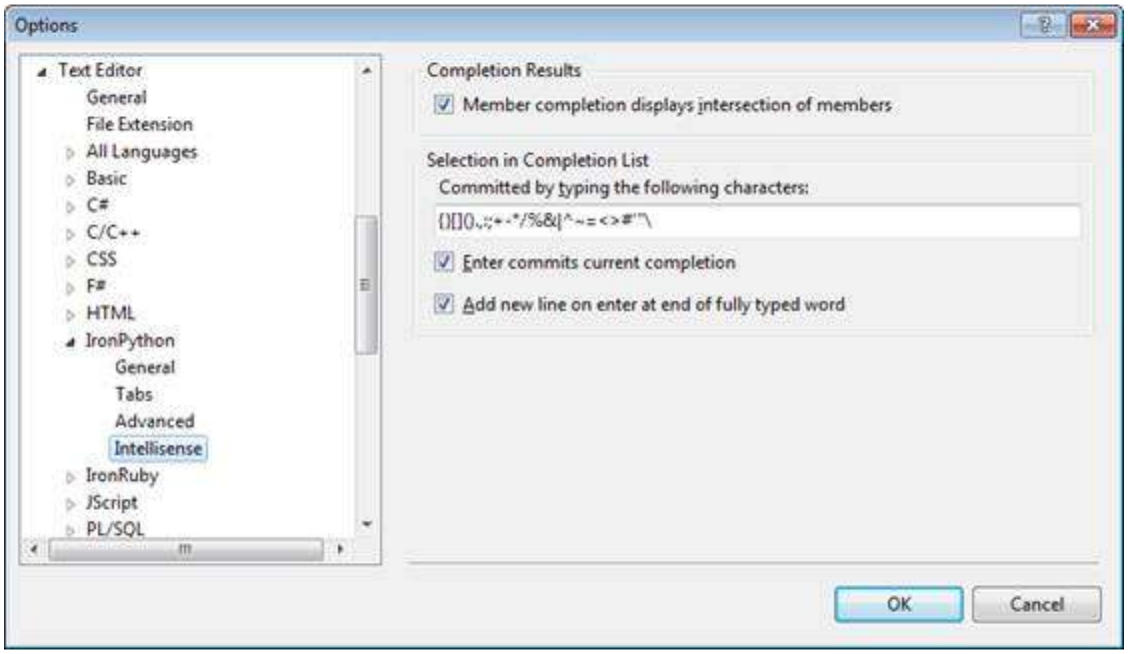
Interactive Window
- [✓] Up/Down Arrow Keys use smart history

Completion Mode
- [✓ Never evaluate expressions]
- Never evaluate expressions containing calls
- Always evaluate expressions

There is an Intellisense page with :

- Member completion displays intersection of members -- on by default, turning it off shows union of members when multiple types can flow through a site
- List of characters whose input signals choosing a selected item in the completion list -- '(', ')', ',', ':', '.' are the defaults.
- Enter commits current completion -- off by default
- Add new line to buffer when using enter after fully typing member -- on by default

Some users think the default on showing an intersection vs. a union of members is backwards, waiting for more feedback to change it.



Completion Results

☑ Member completion displays intersection of members

Selection in Completion List

Committed by typing the following characters:

{}[]().,:;+-*/%&|^~=<>#'"\

☑ Enter commits current completion

☑ Add new line on enter at end of fully typed word

3.12 P1: Scripting Visual Studio with IronPython

IronPython Tools also supports using IronPython as a scripting and extension language for VS. There is a distinct REPL that is specific to the VS process instead of to the application you are building with IronPython. The VS process REPL starts up with an implicit import of DTE, the root of the VS object model. You can write functions and then add them as commands or bind them to keys. This REPL has distinct markings or coloring as well as a different tool window name to distinguish it from the REPL used to develop your IronPython program.

Through the DTE object, you can enumerate projects and files in solutions, access items for reading or modifying, get to the editor's object model, tool windows, etc. We will provide some small convenience functions making it easy to accomplish certain scenarios. For example, DTE.GetObject("IronPythonDteHelpers").BindCommand might take a string representation of a key sequence and an IronPython function, setting up the function as the command handler for the key binding.

What other support is there such as a way to write a MEF extension that we're the dispatcher for in terms of loading IronPython files when the extension is loaded?

3.13 Miscellaneous Editing Commands

This section is a raw list of command features that still need spec'ing.

- pycon: FillCommentParagraph (need option for fill column)
- DONE? P1: auto indent in editor and REPL
- DONE? pycon: rigid indent (falls out of editor plug in)
- P1: comment/uncomment region
- P1: goto matching delimiter (c-])

3.14 P2: Mixed Solutions with Static Languages

For now, this is just a place holder section for future feature thinking.

When you use IronPython in a solution with C# and VB, you are likely doing one of two things. You might be implementing some static .NET helper classes or types that you'll consume from IronPython. You might be building an application in C# or VB for which you want to use IronPython as your scripting or extensibility story. In the latter case, the IronPython project is likely seed/sample code or real features implemented in on the application's object model.

3.14.1 Simple C#/VB Access to IronPython Code for Scripting

We're not sure what the tool support should be yet for setting up an application for scripting. It could be a C# or VB item template that has a static class with a static entry point. This method would create a new ScriptRuntime, set up the IronPython engine, and get ready to execute files or snippets. The support could be a project template for IronPython with some dummy start code that accesses a host's object model and puts up a "Hello World" dialog. The support might be just a text editor snippet.

3.14.2 Implementing a Library for Use from IronPython

In this scenario the user is putting some code for speed purposes or general re-use with other .NET languages into a static .NET assembly. To consume this code, the use likely wants to add a reference to the static language project to consume the DLL produced when building.

Need more details …

For now we think adding a reference to another project sets up the IronPython project to get the DLL from the other project's output directory and copy it to a sub directory of the IronPython project's root. Other DLLs may go there as well when the project is zipped up for distribution or whatnot. Adding the reference also updates the code to insert "clr.AddReference(…)" calls (could be to the start item, a specially named project item, etc.). Using any of the build commands or executing the project in the REPL would cause the dependencies to build, get the DLLs, and then run the IronPython start item.

There could be options for the sub directory's name that holds referenced DLLs. There could be options on where to add AddReference() calls, or the name of the project item. Code may need to be added, or options updated to frob the search path to include the referenced DLLs directory.

3.15 P2: XAML Integration

IronPython Tools for Visual Studio will leverage VS's support for creating stand alone XAML files. We need to provide a standard way for loading XAML files which will work well with both files on-disk and a set of files compiled into their own assembly. If compilation support is provided in IronPython Tools, then this mechanism also needs to support compiling XAML into the resulting assembly and loading it.

3.16 P2: TDD support for VS Features

This is a place holder section to remember to spec something here.