PTVS Test Strategy Document

Contents

[PTVS Test Strategy Document 1](#_Toc313615014)

[Product Overview – Team and Features 2](#_Toc313615015)

[Product Test Strategy Overview 2](#_Toc313615016)

[Limited Test Resources 2](#_Toc313615017)

[Customer Experience is Our Focus 3](#_Toc313615018)

[Find Bugs Early 3](#_Toc313615019)

[Confidence in Quality 4](#_Toc313615020)

[Quality Risks (new in 1.1) 4](#_Toc313615021)

[Summary of Work 5](#_Toc313615022)

[Test Priorities 5](#_Toc313615023)

[Product Test Strategy by Product Area 6](#_Toc313615024)

[Current IDE Features 6](#_Toc313615025)

[Release Management 6](#_Toc313615026)

[Installation 7](#_Toc313615027)

[Python Version Support Matrix 7](#_Toc313615028)

[Test matrix for Python Tools “full test pass” 8](#_Toc313615029)

[Compatibility with VS Plugins and Core VS Languages 10](#_Toc313615030)

[PTVS Configuration Options 10](#_Toc313615031)

[Code Analysis 10](#_Toc313615032)

[Debugging 10](#_Toc313615033)

[Editor 11](#_Toc313615034)

[Intellisense 11](#_Toc313615035)

[Refactoring 12](#_Toc313615036)

[Profiling 13](#_Toc313615037)

[Solution Management 13](#_Toc313615038)

[Code Browsing 13](#_Toc313615039)

[REPL 14](#_Toc313615040)

[Sho & IPython support 14](#_Toc313615041)

[Libraries 15](#_Toc313615042)

[ditertools 15](#_Toc313615043)

[PyKinect 16](#_Toc313615044)

[Pyvot 16](#_Toc313615045)

[Win8 Support 17](#_Toc313615046)

[Other Test Activities 17](#_Toc313615047)

[Test Automation 17](#_Toc313615048)

[Code Coverage 17](#_Toc313615049)

[Scenarios and Feature Design 17](#_Toc313615050)

[Documentation 18](#_Toc313615051)

[Sample Code and Demos 18](#_Toc313615052)

[Test 18](#_Toc313615053)

[Development 19](#_Toc313615054)

[Community Outreach 19](#_Toc313615055)

[Community involvement in testing PTVS 19](#_Toc313615056)

# Product Overview – Team and Features

Python Tools for Visual Studio is an open source effort within Developer Division that aims to improve the experience of developing programs in Python on Windows. There are two main thrusts of the project:

* Provide a world-class IDE for Python via the PTVS Visual Studio Extension
  + We plan to provide all the language services that you expect from a Visual Studio branded product.
  + We currently support: debugging, remote debugging, and MPI debugging, editing, refactoring, intellisense, profiling, code browsing, and solution management; as well as Python-specific features such as an in-IDE REPL window (including Sho and IPython support), and debugging of embedded Python interpreters
  + Features that are under development include: advanced solution and project features, advanced refactorings, code coverage, and unit test support; as well as Python-specific features such as a debug REPL window.
  + Features that are under consideration include: Django support (including syntax highlighting for Django templates) and Cython project support (including syntax highlighting for Cython and build chain support for Cython projects).
* Provide developers with a Pythonic interface to Microsoft technologies via Python libraries.
  + Ditertools: a Pythonic way to connect to the LinqToHPC runtime (and possibly in the future Hadoop)
  + PyKinect: Python bindings for the Windows Kinect SDK
  + Pyvot: a Python module for interop with Excel (basically, a prettified COM interface), focusing on moving data in and out of Excel, taking advantage of Excel’s data visualization features and Python’s data analysis libraries
  + Under consideration: Windows 8 support, including Python bindings to WinRT

The PTVS team is small, consisting of 2 full time SDEs, 1 Partner PM, and 1 SDET. Additionally, we get code contributions from other teams within Microsoft and we accept feature implementations and bug fixes from the community.

# Product Test Strategy Overview

## Limited Test Resources

The test team for PTVS is very small, and the codebase for PTVS is quite large. PTVS is an outgrowth of over 6 years of work within Microsoft on development environments for Python (and other scripting language), most recently IronPython Tools for Visual Studio, and thus comes with a lot of potential test debt.

Because of the imbalance between number of SDEs and SDETs, all members of the team participate in test work to a greater degree than normal. Luckily, DinoV started life as a tester and has done a lot of work making sure of good unit test coverage across the product. As we move forward with new features, we will rely on dev unit testing as a first line of defense for catching bugs and regressions.

We also rely on customer feedback for bug reports, and will aggressively respond to user bug reports; more details can be found in [Community Outreach](#_Community_Outreach) below.

Because the amount of work to do is large relative to the test resources we can bring to bear, we are going to have to accept more tradeoffs than usual in our test work. On the positive side, as an agile, small, free open source project, our customers are willing to tolerate a slightly lower quality bar in return for more features, and if it’s clear that we take their feedback seriously and are quick to respond to their issues. These tradeoffs will be guided by two key strategies, which I have titled *Customer Experience Is Our Focus* and *Find Bugs Early.*

## Customer Experience is Our Focus

This strategy is best summed up as always keeping in mind that one bad initial customer experience will lose us a customer, and lots of them will damage our reputation irreparably. Thus, we want the common-case customer experience for all our features to be one of delight, and of “it just works!”

In each of the product areas below, we try to identify the key customer experience and organize our testing around that experience. In some cases, such as MPI Debugging, this may mean manual or semi-manual testing of end-to-end scenarios. In cases such as Refactoring this may mean heavy automated testing through the Visual Studio DTE interface. In cases such as ditertools it may mean a focus on API testing and perf and scale testing. Ideally we would have all these modalities for every feature area, of course, but when we have to make hard cuts, we will ensure that we are not cutting the tests that are critical to our success. Similarly, we’ll end up make hard choices in bug triage, but we will be guided by prioritizing bugs which are key to providing a good customer experience.

## Find Bugs Early

A bug caught at design time is the cheapest bug. It is an order of magnitude more expensive to catch the bug in code review, more expensive yet to catch it in unit testing, and again more expensive to catch it in integration or end-to-end testing, and most expensive to have to triage a bug report from a user in the field.

With this in mind, we will organize our test effort to catch bugs as early in the product cycle as possible. This puts a burden on test to make sure scenarios are sharply defined, that designs are well thought through and reviewed, that product code is testable, that the test team understands the product code and has a good idea what areas need extra testing, and that end-to-end tests dig a little deeper into those key areas. Test will spend more time than usual on design review and code review. There will be a lot of white-box testing with an eye toward exercising parts of the code which look fragile or complicated. And we’ll put a lot of value on developing end-to-end and integration tests that make sure all parts of the system are exercised.

## Confidence in Quality

To build confidence that our product quality is at a level where we can ship, we propose the following quality gates that define a work cadence for the test team relative to the cadence of the development team:

**Alpha:** For any release labeled “alpha”, the test team will focus on ensuring that bug fixes and feature work have not regressed the existing product features. Customers should be able to use a PTVS alpha with confidence that their existing scenarios are not broken. The dev team should be writing unit tests as they do feature work to ensure that new scenarios are at least minimally functional.

**Beta:** A “beta” should be presumed feature-complete; in cases where we have multiple betas, early betas may not be feature complete but the final beta will have all features finished, and the team will be aggressive about postponing DCRs. For a beta to ship, the test team must have completed work on end-to-end scenario testing for all the new features.

**RC:** For a build labeled “RC” to ship, all P0 test work must be finished and all unfinished P1 test work should be triaged by the feature team.

**RTM:** For “RC” and “RTM”, quality will be determined by the feature team triage meetings as set out in the “Bug Bar” document.

## Quality Risks (new in 1.1)

Python Tools has to start to cope with the fact that Visual Studio is a fragile environment.

 We had 4 (or maybe even 5) bugs in the 1.1 timeframe that weren't strictly speaking our bug, but which broke us, and which we needed to work around:

* The Roslyn CTP didn't change the GUID on the REPL component, causing PTVS to break.
* DebuggerCanvas was broken in a way which caused our code to be the immediate cause of Visual Studio crashing on C# projects.
* F# didn't correctly register their file types, which caused what should have been a minor bug in our file registration to completely break F# syntax highlighting
* The "New File from Existing Code" wizard derived from a VS class which had a broken implementation. This broke PTVS -- but not consistently, because the bug is only exposed when VS loads its extensions in a certain order.
* Some other aspect of the "New File from Existing Code" bug mentioned above broke the TraceLog extension, which caused C# debugging to stop working. (Do we even understand what happened here yet?)

This seems like a really hard problem, and I'm not sure how to suggest going about addressing it. Expanding the test matrix to include a bunch of random components is hopeless, because who knows what the next fragile piece of code will be that impacts the product?  On the other hand, as you move forward you'll have more and more opportunities to depend on code that other people wrote whose quality might be questionable.  Ditertools will rely on Isotope for quality.  Django support is going to implicate Azure components.  Even something as small as "integrate PTVS into Solution Navigator" is starting to look more risky to me than it did 6 months ago. Additionally, the stochastic nature of some of these bugs (VS extension load order!) seems to indicate that finding them is going to be much less systematic than you might like.

Perhaps one way to start looking at this is to have a “kitchen sink” configuration, which includes every Python version, VS Ultimate, and a lot of popular VS plugins. This will randomize the VS environment a little, hopefully giving us a chance to expose product-load issues and other compat issues.

Another thing that will help is to make a small test matrix that involves checking the key VS languages: C++, C#, VB, and F#. We would need to verify basic functionality doesn’t break as a result of installing PTVS.

This work is all TBD, and not necessarily in scope for 1.1 RTM.

# Summary of Work

## Test Priorities

Effort has been made to ensure that the document below reflects priorities which are actually achievable in the course of a release cycle. There are only four priority levels specified: P0, P1, P2, and “Will NOT do”, which correspond roughly to the following estimates of achievability:

* P0 work must be done. This work generally falls into the following categories:
  + Design work: review dev and PM specs, and write test specs. Participate in code reviews. Review documentation.
  + Installation and release management testing.
  + Testing across the matrix of supported Python interpreters and VS SKUs.
  + End-to-end scenario tests for core features (must be complete for Beta).
  + Automation of frequently-run test cases.
  + API testing of our library components (currently only ditertools, but also in the future Win8), including API correctness, boundary case testing, and failure path testing.
  + Responding to customer issues on the Pytools website.
  + Test hardening of features based on issues reported by users.
  + Writing product documentation, demos, scenario docs for upcoming features, blog posts, and other community interaction with high-visibility communities (Slashdot, Microsoft-sponsored blogs).
  + Planning and coordination of community work on product quality, in conjunction with the rest of the team
* P1 work ought to be done, and is on a best-effort basis. This work generally consists of:
  + Code coverage and test case expansion based on code coverage results
  + End-to-end scenario tests of non-core features, or of features which get extensive testing by partner teams such as Sho.
  + Compatibility testing with popular plugins.
  + Functionality verification of other VS languages for compat issues.
  + Automation of legacy tests for code which we expect to remain frozen.
  + Stress testing of components with significant possibilities for race conditions (e.g., ditertools, the debug engine).
  + Performance testing for performance-critical components (e.g., ditertools, analysis generation).
  + Performance testing of components where our users might notice a perf degradation even though we don’t control the component.
* P2 work would be nice to have, but is essentially out of scope given current test staffing.
  + In-depth microbenchmarking of performance critical components.
  + In-depth testing of sample code.
  + GUI automation of non-core GUI components where manual testing is quick.
  + Automation of non-core scenarios where automation is a lot of work relative to the gain in efficiency.
  + In-depth testing of scenarios which will be completely rewritten for v2.0.
  + Community interaction with lesser-visibility communities where PM and Dev are already engaged (Twitter, Facebook, mailing lists).
* “Will NOT do” is work which is not essential to our quality goals and will be explicitly ignored.
  + In-depth testing of non-core Python interpreters. (Jython, PyPy, Stackless, …)
  + Testing of Dev11 integration in the 1.x timeframe. (We do have a Dev11 build, but we don’t ship it – users have to build it themselves.)
  + Most stress testing.
  + Fuzz testing – API and network-stack related.
  + Security analysis including threat modeling and the like (this may be reconsidered when we move to doing more work with Azure).
  + Integration testing with other Microsoft-supplied VS extensions, or 3rd-party extensions, even though these have been the source of bugs (Roslyn, Debugger Canvas). We will triage these bugs when they are reported, though.

I currently estimate that all the P0 and about half the P1 work items are achievable by the release of v1.1

# Product Test Strategy by Product Area

## Current IDE Features

### Release Management

Part of our product release cycle involves pushing code from our internal repositories to Codeplex. Before this happens, we need to ensure that all the files pushed to Codeplex have the appropriate licenses attached, and that they pass Policheck. Since Codeplex pushes happen whenever bugs are fixed or features are implemented, automating this process is P0.

### Installation

There are two key SKUs we need to ensure work with PTVS:

1. Visual Studio Ultimate: only Ultimate enables all our features (Profiling is Ultimate-only)
2. Visual Studio Integrated Shell: only VSIS is free for users to download.

For those SKUs, we need to ensure that the following scenarios work:

1. PTVS installs in a completely clean environment (nothing other than VS SP1 installed).
2. All PTVS features are installed when all the prerequisites are installed (HPC Pack & MPI4Py for MPI debugging; IPython for IPython REPL; IronPython for the IronPython projects).
3. Installation failure modes should be tested. For example, we should fail if a previous version is installed; and we should fail if IronPython Tools is installed.
4. Uninstallation and Change/Repair Installation should work.

These are P0.

We do NOT need to test with Dev11 – Dev11 support will be a 2.0 feature, and require extra work to make sure the IronPython designer works with the new designer framework.

We do NOT need to do perf or stress testing for installation.

### Python Version Support Matrix

#### CPython

PTVS supports all Python versions from 2.5 to 3.2, in both 32- and 64-bit versions. Additionally, we do extra work to support extended features available in partner distributions such as those by Enthought and ActiveState. Standard CPython 2.5 – 3.2, as well as the current Enthought EPD and the current ActiveState distribution are P0 for our test matrix. It is also P0 to ensure that 2.4 and lower are not detected and acted upon.

#### IronPython

We support only IronPython version 2.7; 2.6 and lower are NOT supported. We will only test on the current download, which is: 2.7.1. We will ensure that Sho works with our tested download, see the section on [Sho](#_Sho_&_IPython) below.

#### Jython, Pypy, Stackless, etc.

Some of our features, such as debugging, should work with any other Python distribution that is compliant with CPython. Some, such as Attach To Process, will only work with CPython. We will not commit to in-depth testing of any distributions other than CPython and IronPython, but as a sanity check we will insure that PyPy can be installed as a custom interpreter and that basic functionality works.

### Test matrix for Python Tools “full test pass”

#### Goals

* The full test pass test matrix should have enough configurations in it that we cover the key customer scenarios with good confidence
* The test matrix should be small enough that it is feasible for one tester to complete testing in less than 20 hours of work (with automation wherever possible)

#### Test Matrix configuration considerations by feature

##### Intellisense:

* Intellisense analysis works on all supported Python configurations
* Intellisense analysis happens automatically on automatically-detected Python installations
* Intellisense analysis works on user-configured Python installations when the user generates a completion database.
* Intellisense responds correctly and does not get confused when multiple Python installations are present:
  + “Default Python” versus project-configured python
  + REPL and editor

##### Debugging:

* Debugging works on all supported Python configurations
* Debugging works correctly on user-configured Python installations (e.g., the correct debugger is launched, Watches and Locals are correct)
* MPI debugging only needs to work with CPython (i.e., explicitly not IronPython)

##### Profiling:

* Profiling only works on CPython and VS Ultimate

##### REPL Window:

* REPL window needs to respect current project configurations irregardless of default configuration
* “send to” REPL window needs to work with ASCII and Unicode source files all configurations
* IPython and IPython cluster integration should be tested in the context of a pre-packaged distribution such as Enthought.

##### Windows:

* We will restrict our test matrix to Win7 x64. As test resources and additional features are added, we may expand the test matrix to Win8 or Win7 x86.

##### Visual Studio:

* Ultimate is the only SKU which supports all our features (including profiling and source code build chain). We will restrict testing to SP1 (or the most recent service pack)
* Basic features should be tested on VS Integrated Shell

##### HPC:

* V3SP3 has the required Dryad bits and most comprehensive Azure support, plus some new features in the MPI stack
* Conclusion: We will target V3SP3 as our supported HPC platform. Moving forward as we start to develop Azure features, we will use the most current Azure SDK, which will contain the HPC Scheduler In Azure bits.

##### Build chain/source code:

* Our users need to be able to download the source from CodePlex and build it on a correctly configured VS2010 Ultimate.
* Users should be able to run our automated unit tests from within VS on a correctly configured VS2010 Ultimate
* Testing should be done on a non-domain-joined non-Microsoft PC.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Basic x86 | Basic 2.7 Int. Shell | Basic 3.2 Int. Shell | Basic IPy Int. Shell | Basic x64 | Kitchen Sink |
| VS2010 Ultimate SP1 | ■ |  |  |  | ■ | ■ |
| VS2010 Integrated Shell |  | ■ | ■ | ■ |  |  |
| Python 2.5 | ■ |  |  |  |  | ■ |
| Python 2.5 x64 |  |  |  |  | ■ |  |
| Python 2.6 | ■ |  |  |  |  | ■ |
| Python 2.6 x64 |  |  |  |  | ■ |  |
| Enthought Python 2.7 | □ | □ | ■ | ■ |  | ■ |
| Enthought Python 2.7 x64 |  |  |  |  | □ |  |
| Python 3.0 | ■ |  |  |  |  | ■ |
| Python 3.0 x64 |  |  |  |  | ■ |  |
| Python 3.1 | ■ |  |  |  |  | ■ |
| Python 3.1 x64 |  |  |  |  | ■ |  |
| ActiveState Python 3.2 | □ | ■ | □ | ■ |  | ■ |
| ActiveState Python 3.2 x64 |  |  |  |  | □ |  |
| IronPython 2.7 | □ | ■ | ■ | □ | □ | ■ |
| PyPy | ■ |  |  |  |  | ■ |
| Visual Studio plugins? |  |  |  |  |  | ■ |
| Lots of different SDKs? |  |  |  |  |  | ■ |
| Other things that can cause interop problems …? |  |  |  |  |  | ■ |

### Compatibility with VS Plugins and Core VS Languages

TBD after 1.1 (see “Quality Risks” section).

### PTVS Configuration Options

PTVS has a number of configuration options which control how the IDE behaves; all options are centralized under the Tools->Options->Python submenu.

These options should be tested in the context of their respective features. Testing configuration options can be automated through the DTE. This is P0.

### Code Analysis

The Code Analysis engine is the basis for all the features of PTVS which rely on a syntactic or semantic understanding of the user’s code. It is not directly available to customers who are using PTVS in the core scenario as a Python IDE. It is accessible programmatically to developers who want to extend or enhance PTVS.

The analysis engine has a large body of unit tests around it, and DinoV is extremely fastidious about extending those unit tests as new features are added.

Therefore we are NOT going to do any extra test work around the API for the Code Analysis engine. Features of the Analysis engine which are visible to the user of the IDE will be tested in the context of those IDE features. Refactoring, Intellisense, Editor, Code Browsing, and Solution Management all test the Analysis engine indirectly.

### Debugging

Debugging is a core IDE feature and one which we have some stand-out features in relative to the competition. All debugger scenarios are automatable through the DTE, and there is a large body of unit tests surrounding the debugger.

The debugger communicates with a python process being debugged using Python’s support for debugging and tracing. On process start or process attach, we inject code into the debuggee’s sys.settrace; our code then communicates back to Visual Studio using sockets. Each Python thread has its own sys.settrace, but due to Python’s Global Interpreter Lock, there are tricky synchronization issues especially during process attach. Also, because only one Python thread can execute with the GIL at a time, there are synchronization issues when talking back to Visual Studio.

Test priorities for the debugger are:

1. Automate end-to-end scenario and stress tests for the debugger. The goal is to make sure that even under multithreaded scenarios, all breakpoints are hit and variables in each thread are kept synchronized between the VS interface and the Python process. (P0)
2. Automate end-to-end scenario and stress tests for process attach. This ensures that process attach continues to work even in the face of large numbers of threads in the Python process. (P0) This testing needs to be done across all supported Python versions (P0)
3. We are capable of attaching to a Python interpreter that is embedded in a separate process. Manual testing of this scenario is P0. Automated testing is P1.
4. We support attaching to embedded debug Python interpreters. Manual testing of this scenario for 2.7 and 3.2 is P1. Automated testing is P2.
5. Automating the Debug/Attach GUI is P2.

#### MPI Debugging & Remote Debugging

Test priorities for MPI debugging are:

1. Ensure thorough end-to-end testing of the core scenario: “build, deploy to cluster, schedule job on cluster, break on breakpoints on cluster”. Semi-automation (given a manually deployed cluster headnode) is P0.
2. Stress testing of MPI debugging using the stress tests for local debugging is a P0, to uncover race conditions and synchronization issues across multiple machines.
3. MPI Debugging is a frozen feature (barring future unscheduled Azure work). Ensuring future work on the debug engine does not cause regressions is P0.
4. Automated testing of local MPI debugging is P1.

Although Remote Debugging is a separate feature from MPI Debugging, we don’t test it separately because MPI Debugging relies on Remote Debugging to work.

### Editor

The editor window provides lots of complex functionality, some of which (such as Intellisense and Refactoring) is important enough to have a separate section in this document. Outside of those features, all the core functionality of the editor window is automatable using the DTE and the accessibility interfaces. Furthermore, this area is thoroughly unit-tested and tested in use by the product team daily. Therefore, the test priorities for the editor are as follows:

1. Ensure that test cases are hardened around user-reported bugs in the editor window. (P0)
2. Ensure that automated tests are expanded around user-requested features so as to maintain high test coverage. (P0)
3. Scale/perf testing of the editor with large buffer sizes and long line lengths is an important P1.

### Intellisense

Intellisense is a key feature in an IDE, and our Intellisense implementation is more featureful than any of our competitors’. The analysis engine and the Intellisense engine are both accessible programmatically; additionally, the Intellisense GUI can be automated through the accessibility interfaces.

Intellisense for the Python standard library is provided through a static database. Intellisense for individual Python distributions is generated and cached when PTVS is first run. Intellisense for user code is based on static code analysis of the users’ project. Intellisense in the REPL is provided using both static analysis and live inspection of the Python runtime.

Our test priorities for Intellisense are as follows:

1. The lag time between when PTVS is first run and when the Intellisense database is completely generated is often long enough that is noticeable by the user. Therefore, perf testing of the Intellisense database generation is P0, in order to establish benchmarks and make sure there are no major regressions that will impact the user experience.
2. Once the Intellisense database is available, completions should be available and correct in the editor. Automated testing of this is P0.
3. Complex project setups should give the user correct intellisense across all user code. Automated testing of this is P0.
4. Live completions in the REPL should be correct, especially for IronPython and dynamically loaded DLLs. Manual testing of this is P0, automating these tests is P1. (This functionality is exercised by the Sho team, so it has good partner coverage.)
5. Intellisense completions should not cause delays for the user; developing perf tests for the GUI is P2.

### Refactoring

Refactorings are take advantage of our code analysis engine to allow the IDE to automatically perform common code transformations for the user. Supported refactorings include “extract method”, “rename symbol”, and the like.

A typical refactoring workflow has the customer selecting some code, invoking the refactoring, providing some additional information and previewing the proposed changes in a dialog box, and committing the refactoring. The complete code transformation workflow is available programmatically, but there is still a GUI component to the customer experience.

The primary customer requirement in refactoring is that the code transformations be correct – refactorings should not produce invalid code. This leads to the following test priorities for the refactoring engine:

1. Automated functionality testing of the refactoring engine through the DTE. The test space here is large; to name some test case examples: comments should be preserved in extracted methods; renames should not rename inside comments; renames and method extractions need to respect Python scope rules; and so forth.
2. Automated testing of the end-to-end scenario of refactoring working correctly with the project system, for instance: Is refactoring undoable, in the case of user error of code bugs? Does refactoring correctly work across Python modules and across VS projects?
3. Testing of the GUI – manually, and automatically using the accessibility layer. Do the previews in the dialog box accurately reflect the code changes? Does the dialog box correctly pass parameters to the refactoring engine and correctly display errors (such as “invalid symbol”)?
4. Performance and stress testing of refactoring is not important

Dev unit tests of the refactoring engine are quite comprehensive, so test resources will be focused on points 2 and 3 above. Point 2 is a P0 for test, and point 3 is a P0 for manual tests, with automation being a P2 “nice to have”.

### Profiling

PTVS hooks into the Profiling feature available in Visual Studio Ultimate. We instrument user code using Python’s debugging and profiling hooks, and use the data from the instrumentation to provide users with drill-down summaries of where time is spent in their code. Our python profiler is written in C++ and works only on CPython. Integration into the VS interface is done with C#.

1. Profiling has not had feature work done on it since 1.0, and no significant bugs have been found. Therefore our primary focus is to make sure that the codebase does not regress. (P0)
2. Profiling is accessible via the DTE, and via the accessibility interfaces, so there are automated tests in the unit tests which test all the basic functionality. Running these tests automatically across the CPython version support matrix is a P0.
3. Further focus in testing profiling will be end-to-end scenario testing. This testing will be done on sample code that uses the ditertools and PyKinect libraries, to make sure that the new code we are developing works well with our developer productivity features. (P1)
4. Because profiling via instrumented code is expected to cause performance of that code to suffer, perf testing of the profiler is P2. We will not do any stress testing.

### Solution Management

Solution Management includes all interactions with the Visual Studio solution system, most notably through the Solution Explorer. (PTVS does not yet integrate to Solution Navigator.) The goal of our Solution Management implementation is to achieve feature-parity with C#. The solution system is automatable through the DTE and through the accessibility interfaces, and there is some unit testing around the feature. Our test priorities for the Solution system are:

1. Automated end-to-end scenario testing of complex solution workflows, which is all new feature work: multiple solutions in a project, cross references between projects, linked files in projects, projects which contain C++ and Python code, projects which contain C# and IronPython code, and the like. (P0)
2. Working with Python files outside of a solution or project (in the “implicit” project) (P0)
3. Scale testing and perf benchmarking of very large projects (> 800 files) (P1)
4. Scenario testing of the IronPython WPF design system (P2 – due for a complete rewrite in 2.0)

### Code Browsing

Code browsing includes three main features: Find All References, Go To Definition, and the VS Object Browser.

The machinery underlying Find All References and Go To Definition is the code analysis engine, and it is automatable via the DTE and via the accessibility interfaces. Priorities for testing in this feature area are:

1. Automated end-to-end scenario tests of complex solutions (projects with nested submodules, solutions with multiple projects, projects with external libraries). (P0)
2. Automated end-to-end scenario tests of Find All References/Go To Definition on live code (code which is being debugged), including going to definitions inside the standard libraries or in other library code. (P0)
3. Scale testing of Find All References on very large projects (P1)

The Object Browser is automatable through the VS accessibility interfaces. Priority for testing the Object Browser is:

1. Manual testing of the Object Browser in the course of other end-to-end scenario tests. (P0)
2. Automated testing of the object browser in end-to-end scenarios (P2).

### REPL

The REPL allows developers to test their code and explore the behavior of library code in real-time. Basic REPL functionality is automatable with a mock REPL window class; advanced functionality is accessible via the DTE and the accessibility automation, and there is a large body of unit tests around the REPL covering evaluation of code and basic REPL behavior in response to user typing. Therefore the test priorities for the REPL fall out as follows:

1. Automation of advanced end-to-end scenarios such as “Send to REPL”, “Paste in REPL”, “Execute project in REPL” is P0.
2. REPL scale testing examining REPL performance under extreme usage scenarios is P0
   1. with lots of output in the window
   2. lots of commands in REPL history
   3. large pastes
   4. large individual-line outputs
   5. many inline graphs
3. Regression and test hardening of corner cases reported by partners and customers is an important P1. (We’ve had numerous bugs where the REPL didn’t behave the way a Python console does, causing customer pain.)

At some point, maintenance of the REPL window code will pass entirely to the Roslyn team, at which time we will either switch over to their REPL completely or shim their REPL with a thin class that implements our REPL functionality on top of it. At this point, we won’t do any further test development.

### Sho & IPython support

The test challenge here is to balance the following competing interests:

1. Support for our partners is important to us and we would like our customers to have the best experience possible using our partners’ products within a PTVS context.
2. We can’t possibly be a test team for Sho and IPython on top of testing PTVS.

#### Sho

Sho is a project released by Microsoft Research. Dubbed a “Playground for Data”, it is a set of libraries which attempts to make IronPython competitive with numpy and Matlab for scientific computing and data analysis.

PTVS has added features, mostly in the REPL, to enable some scenarios for Sho. These REPL features are not Sho-specific and are tested in REPL testing.

1. Our highest priority when working with Sho is to make sure that their setup instructions work for users who want to turn their IronPython REPL into a Sho REPL (replacing the Sho console). This is P0.
2. We also need to test the Sho scenario of developing C# libraries and IronPython modules in the same Visual Studio solution. This is P1.
3. For all other aspects of Sho integration, we rely on the Sho development team to test and report bugs.

#### IPython

IPython is an alternative CPython shell. It provides a lot of utility functions for working with Python code, provides enhanced tracebacks for debugging, provides syntax highlighting and advanced completion in the REPL, and can integrate with external editors and debuggers. Additionally it has added support for inline graphics when working with matplotlib.

PTVS support for IPython consists of a IPython REPL mode that supports their advanced coloring, completion, and inline graphics.

1. Our highest priority for testing IPython support is to ensure that supported IPython builds are correctly detected and IPython mode is enabled for them. This is P0.
2. Users have historically had a lot of problems getting IPython mode to work, so ensuring that our documentation is clear and correct and that our error messages are actionable is also a P0. Test has committed to making a video walkthrough of this scenario.
3. The final P0 is to test end-to-end scenarios with IPython in a standard distribution, such as the Enthought EPD.

## Libraries

### ditertools

ditertools is a library for operating on distributed data. Its interface is modeled after the standard Python itertools library; the current backend is implemented on top of the LinqToHPC stack being developed by the HPC Server team.

Customers of ditertools are expected to value performance, scale, and robustness, as their scenarios involve operations on big, distributed datasets.

Our test priorities for ditertools are as follows:

1. Ensure dev unit tests cover basic functionality for all ditertools operators and that they are runnable using the unittest framework. (P0)
2. Automate dev unit tests so they are runnable inside the HPC team’s WTT automation. (P1)
3. Ensure that all of the LinqToHPC scenario tests from MSR are ported to Python and that all the scenarios work (unless explicitly unsupported) (P0)
4. Write scale and performance tests and benchmark ditertools performance on the LinqToHPC CTQs, which are Histogram and TeraSort. (P0) Automate these tests so they can be run by the HPC team in daily perf tests (P0)
5. Write micro-performance tests for ditertools, exercising our serialization code and comparing the “fast path” serialization to the generic Python serialization (P1)
6. Write samples and documentation (P1)
7. Write scenarios in consultation with the L2H team (P2)

### PyKinect

PyKinect is a Pythonic set of wrappers around the Kinect SDK for Windows. It provides access to all the data coming off a Kinect device, including the image stream, the point cloud stream, the skeletal tracking data, and the audio stream (including an interface to the voice recognition / voice command APIs).

The challenge of writing a test suite for PyKinect is test automation: we don’t have access to any way of recording and playing back data off the Kinect device. I’ve talked to people on the Kinect team and it’s not clear if there is any formal record/playback library; if that library exists, it’s internal-only and not something we could ship with PyKinect.

* The library is getting a lot of usage because DinoV is writing a comprehensive sample app as part of a presentation at PyCon.
* I will write at least one E2E sample code as part of my ditertools demo for Supercomputing 2011.
* If further testing is desired it will have to start with building an automation infrastructure, which will require a solution for datastream record/playback (either at the level of a mock Kinect device, or through a shim Kinect API dll). This would have benefits for the community at large, as such a test infrastructure doesn’t exist currently. But it is far out of scope in terms of day-to-day activities for the PTVS team.

Therefore, testing PyKinect is Pri2. It is currently considered “sample code”.

### Pyvot

Pyvot is a library developed by Mike Stall, formerly of the Miletus team. The goal of the feature is to facilitate using Python to work with data that is originally located within Excel. It implements Python wrapper classes around ctypes calls to the Excel COM interop libraries.

The ship vehicle for Pyvot is as an optional install within the PTVS package.

Pyvot has no formal PM or developer spec. It has developer unit tests and sample code.

Testing Pyvot is P2. It is currently considered “sample code”. If test resources become available, the testing priorities will be, in this order:

1. Development of customer scenarios.
2. End-to-end testing of customer scenarios (manual/exploratory testing, including writing sample code).
3. Scale and perf testing focused on moving big data from Excel to Python and back.
4. API functionality testing.

Pyvot is not being developed further. The only further work that is potentially open for Pyvot is to integrate into the Pypi package library as an alternative distribution method. If we go forward with this, additional test work will involve writing a unit test library in Python for the API to include with the package (promoting point 4 above to 1st place).

## Win8 Support

Win8 support has been postponed because of concerns from the Windows team. Expect this section to be huge. But it also won’t be fleshed out until the 2.0 planning timeframe (or afterwards).

# Other Test Activities

## Test Automation

TBD – Describe automation issues not covered above, including how we automate the supported versions matrix, ditertools, MPI debugging, and strategies for future automation for cloudy features.

## Code Coverage

Code coverage metrics based on the tests automated through the VS test interface can be generated within Visual Studio. Since white box testing is a high priority for the test team, code coverage information has the potential to be extremely valuable when implementing tests. Code coverage will be run and analyzed at the end of the Beta release cycles, and will help drive test development during the preparation of RC.

Often, tests will be available but not automated through Visual Studio which will exercise areas that are lacking coverage. In these cases, code coverage will drive test case automation efforts.

Analysis of code coverage is an explicit release gate that must be completed before entering “Ask” mode, per the bug bar document.

This is P0.

More frequent code coverage runs are nice-to-have, and are P1.

## Scenarios and Feature Design

Test will participate in-depth in feature design work. This is P0 and includes:

1. Involvement in dev spec development early in the process with frequent reviews.
2. Ensuring that features are designed for testability.
3. Work with the dev on the scope of unit testing (and implementation details if necessary).
4. Planning for automatability, especially in projects like ditertools where we would like to integrate into other teams’ test suites.

Test will also participate in the development phase of features by participating in code reviews both of bug fixes and new features. It is test’s job to understand the codebase nearly as well as the developer, and to provide timely feedback on code as work progresses. **This work is P0** and key to our strategy of Finding Bugs Early.

To support the PM team, and as part of the test team’s focus on customer experience, the test team will review scenario specifications and also opportunistically write scenarios for features before development. These scenarios will take into account customer feature requests and team use cases for features, and in the case of features like ditertools and Pyvot they will leverage the work done by other teams in their scenario development. This scenario work is important to ensure that we are building the right feature, and to ensure that feature testing is concentrated on the main use cases. This work is P1.

## Documentation

Our customers clearly have an interest in our documentation being complete and correct. Before any major release, test will review all the documentation on the codeplex website adopting the following strategies. These tests are P0 and required for release:

1. Review documentation for PAST features to ensure it has not gone stale
   1. Have we fixed bugs so we no longer require workarounds?
   2. Have we implemented features that make workflows easier?
   3. Have our partners (IronPython, Sho, IPython) changed their products in a way which obsoletes our documentation?
2. Walk through every step of all documentation for NEW features on a clean PTVS install on both a Visual Studio Ultimate VM and a Visual Studio Integrated Shell VM.
   1. Are the instructions clear and correct for both environments? (Unless it’s an Ultimate-only feature.)
   2. Do all examples work as described in the documentation?

Furthermore, because the PM team can sometimes be overstretched, test will write website documentation for features, and review and elaborate on documentation which may be in a state of rough draft. This is a P1.

Finally, as time allows, test will create video walkthroughs and write blog posts on topics which may be of interest to the community. These are P1.

## Sample Code and Demos

### Test

Customers expect that sample code will install and run correctly on a correctly configured system. In fact, sample code can be considered an extension of documentation and will be tested as such, with a walkthrough of all samples on a clean Visual Studio Ultimate VM to ensure the code runs correctly in Python 2.7 (and 3.2 if the feature is designed for 3.2).

### Development

As part of end-to-end scenario testing for our library products, test will write at least one high-quality, relatively featureful sample for each. (For libraries such as PyKinect and Pyvot this and sample code testing may be the only test work committed to.) The priority of this work is determined by the library in question, but is assumed to be P1 or P0.

Furthermore, in support of PTVS team appearances at conferences such as Supercomputing (and in the future, Mix or Build), the test team will develop and run demos that highlight PTVS capabilities. The priority of this work is determined by the visibility of the conference, but is probably by default P1.

## Community Outreach

Test will actively maintain the relationship with our customers on Codeplex. We will promptly

* respond to user questions
* regress bugs
* verify bug fixes
* ensure test coverage around user-reported bugs.

This work is P0.

Additionally we will participate in user discussions on other forums such as Slashdot, Reddit, StackOverflow, and the team blog. This work is P1 or P2 depending on the visibility of the forum.

## Community involvement in testing PTVS

The PTVS team and the C++ AMP team are actively seeking ways to engage the customer in testing our product. As an ideal first step, it would be great if we could offload much of our manual testing (documentation, end-to-end scenarios) onto community members.

* Community members are not PTVS specialists; they will see bugs which we overlook because we’re overly-familiar with the product.
* Community members have diverse environments, so involving them in testing gives us scattershot coverage of a lot of random test scenarios.

Here is a list of some ideas we could implement for community testing. Highlighted rows are highly desirable for 1.1.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test Item | User Effort | Effort to Enable | User Type | Comments | Benefits |
| Customer advisory board | Medium | Low | Advanced | Have a “virtual” triage team for users who are most involved with the code to provide feedback on quality  Focus on the users who are most engaged (the ones who download and build from source) and have a good idea of our current progress. | Another opportunity for red flags on product quality to be raised  Customers involved in a CAB might be more willing to do detailed test work (like running our tests or walking through scenarios). |
| Community contributions must come with integrated tests | Medium | Low | Advanced | This is actually standard with other open source projects. To some extent, this is the “pay to play” for contributing – make sure your contributions work. PTVS team may have to do a little extra handholding to help people get their tests integrated. | PTVS team doesn’t have to write these tests! |
| Scenario walk-throughs | High | Low | Any | We tried this once and didn’t get a lot of participation | Ad-hoc exploratory testing of key scenarios in a more randomized way than we can get from just one test team member |
| Provide unsigned “daily builds” for customers on the bleeding edge | Low | Low\* | Any | Legal and/or management needs to be involved in this as the biggest barrier to providing these is the release sign-off process. | More customer eyes on the code; quick feedback on new features. |
| Run existing unit tests | Medium | Low | Advanced | User must be able to build PTVS Test runs are lengthy but automated  Need a procedure for test results to be reported back to us | We get test runs on a larger number of configurations than we can support in the lab  Feedback on test stability and reliability  More eyes looking at bugs |
| Survey of user configurations | Low | Low | Any | Copy / paste from Tools / Python Tools / Diagnostic Info …  PII worries? | Detailed glimpse into what users’ configurations look like  Helps focus our testing to make sure we’re covering the right configurations |
| Integration into PerfWatson | Medium | ? | Any | Maybe we already are integrated into PerfWatson, in which case we just need to get the data from the reporting database? | Comprehensive perf testing is a low priority test goal, so PerfWatson reports would be extremely helpful in pinpointing areas that are problems. |
| Customer feedback button | Medium | Medium | Any | Similar to send-a-smile in Office / VS Betas. Could we integrate to CodePlex somehow? Maybe only for non-RTM builds? | Make it easier for customers to give us feedback and report bugs.  Insta-feedback from users as to product quality |
| Automatic reporting of user configs | None | Medium | All | Report user data back (from SQM?)  What SKU of VS, what versions of Python are installed? | High-leve glimpse into users’ configurations |
| Community feedback on documentation | High | Low | Any | Users right now report documentation problems if there are obvious bugs. Can we get them to provide even more detailed feedback / participation in rewriting somehow? | Improved documentation in the absence of technical writers. |
| Post PyCon community sprint | High | Medium | Advanced | All sorts of serious open source projects do stuff like this. | Outreach to customers; get the smart people who go to PyCon involved with the project; encourages people to contribute |

Still TBD is to come up with some incentives that will get the community excited about being engaged with testing.