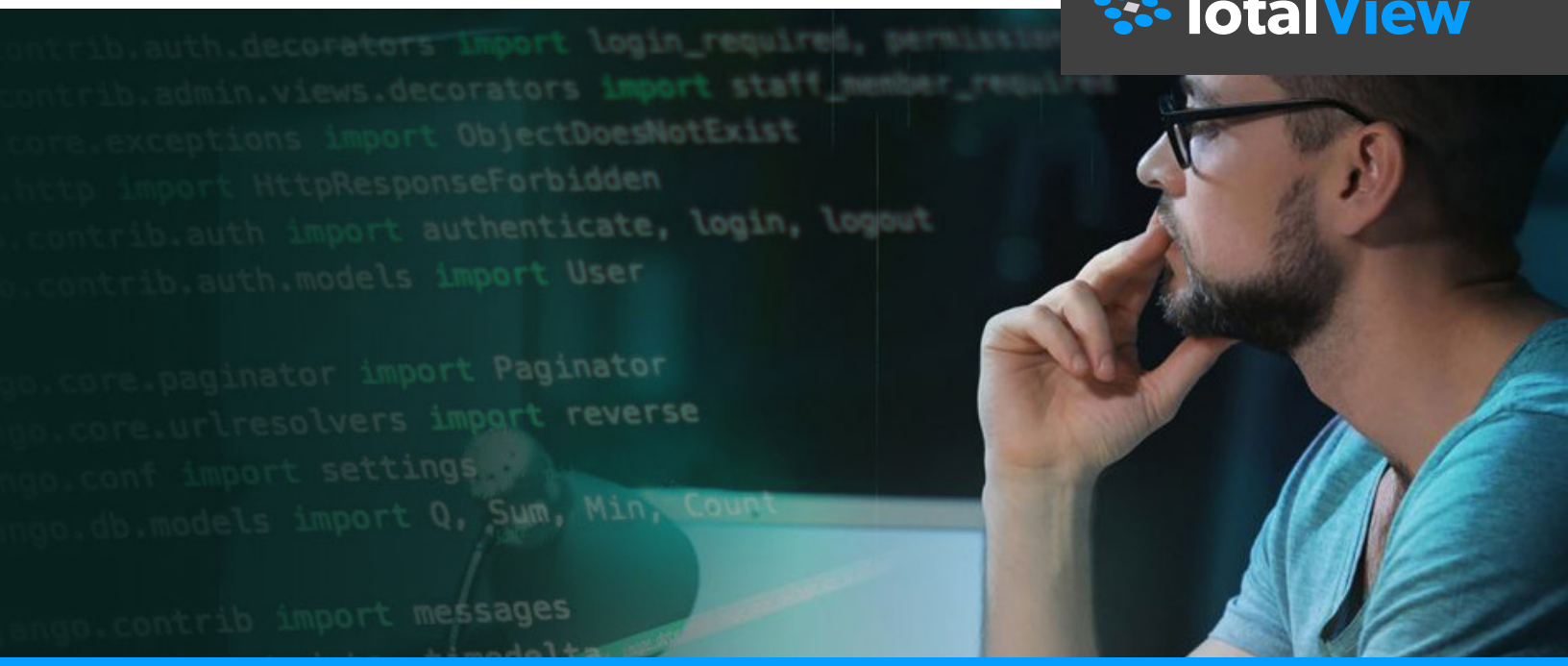
TotalView



WHITE PAPER

Understand and Debug Multi-Language Applications

Introduction

According to the PYPL Index, Python's™ popularity has steadily grown over the last 10+ years. Its popularity across industries like deep learning, artificial intelligence, and stock trading stems from its ease of use, clean syntax, extensive third-party documentation, and large number of extension libraries.

Extensions enable Python applications to access legacy algorithms to leverage hardware and perform specialized calculations. A major benefit to using Python is that it's easily extensible with C and C++ code, enabling developers to tie different parts of their program together and create a mixed-language application. However, understanding and debugging the interdependencies between language barriers in a mixed-language application can be a challenge.

In this paper, we'll explore technologies available for creating mixed Python and C/C++ applications and debugging techniques developers can use to understand how data is exchanged between the layers of their program.

Extending Python With C and C++ Extension Modules

Python developers often need to access existing C or C++ code to use legacy or highly-performant code or specialized hardware resources. Python extensions help developers access C or C++ code. Python extensions are C/C++ routines compiled and linked into a shared library and dynamically loaded at runtime by the Python interpreter.

Natively, Python provides ctypes, a foreign function library, which provides an infrastructure for calling functions in shared libraries and the exchange of C-compatible data types between the language barriers. Ctypes is not the only solution for calling C. A plethora of other technologies exist with their own approaches to making it easy to call between and exchange data between Python and C/C++.

Python C/C++ glue technology	Description
ctypes	A foreign function library for Python.
Cython	A superset of the Python language that additionally supports calling C functions and declaring C types on variables and class attributes.
SWIG	A software development tool that connects programs written in C and C++ with a variety of high-level programming languages including Python.
CFFI	Foreign Function Interface for Python calling C code.
PyQt/PySide and SIP	A tool that makes it easy to create Python bindings for C and C++ libraries.
Boost.Python	A C++ library which enables seamless interoperability between C++ and the Python programming language.

Later, we'll dive into debugging Python and C/C++ applications, but first we'll examine how easy it is to combine Python with C and C++ and prepare it for debugging. A simple SWIG example will be illustrated and used to compute the factorial of a number by calling in to a C routine from Python.

The signature for the C factorial function is defined in header file example.h.

```
int fact(int n);
```

example.h

The implementation is a simple recursive routine.

```
#include “example.h”

int fact(int n) {
  if (n < 0){
    return 0;
  }
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
```

example.c

To prepare for calling into C from Python, SWIG builds a wrapper layer between the languages. It does this by analyzing the C header file. If required, finer grained control can be accomplished by writing your own SWIG interface files.

```
%module example

%{
#define SWIG_FILE_WITH_INIT
#include “example.h”
%}

int fact(int n);
```

example.i

With all the files ready, we need to create the Python module using SWIG. Our example is using C — if C++ is used, add the -c++ option to the SWIG command line arguments.

```
# Generate the SWIG wrapper files
$ swig -python example.i
```

Creating the Python module with SWIG

The SWIG program produces two files from the example.i input file: A C source file named example_wrap.c and a Python source file named example.py. Together, these files produce the glue between the languages. We'll look a bit more at the glue code later as we debug the program.

Up to this point there are no specific changes needed to debug your Python or C/C++ code under a debugger. However, to effectively debug C/C++ code called from Python a debug version of the Python interpreter must be installed. For example, on an Ubuntu-based system issue

```
sudo apt-get install python-dbg python-dev
```

For other Linux distributions, check your package manager for equivalent packages.

There are two approaches to building the Python extension: Use distutils or build it by hand. Distutils is a tool that figures out the necessary flags and other system dependencies to properly build the extension. For brevity, we'll compile the extension by hand with a couple of gcc commands. To prepare the extension for debugging add the -g argument to generate C debug information and specify the include file search path to the debug version of the Python interpreter header files.

```
# Compile all files
$ gcc -g -fPIC -c example.c example_wrap.c \
    -I/usr/include/python2.7_d
# Link the files into shared library _example.so
$ gcc -shared example.o example_wrap.o -o _example.so
```

Creating the Python extension shared library

There are a few things to note about compiling Python extensions:

- The shared library must be named the same as the module defined in the SWIG interface file and prefixed by an underscore.
- The module compiled with the debug version of the Python header will not run under the normal version of the Python interpreter, due to the addition of symbols used for full debug information.

With the Python example extension built we can run the following simple Python program to try it out:

```
def getFact():
   import example
   a = 3
   b = 10
   c = a + b
   return example.fact(a)
if __name__ == '__main__':
   result = getFact()
   print result
```

test.py

Run using the debug version of the Python interpreter:

```
ubuntu:~/swigTest> python-dbg test.py
6
```

Running test.py

The Challenge — Debugging Python and C/C++ Applications

Python extensions make it easy to combine Python and C/C++, but create a more complex application. Developers rely on debuggers and dynamic analysis tools to understand how their program runs. However, understanding the calls across language barriers and the flow of data between them creates a challenging debugging situation. Few debuggers provide a seamless solution for debugging both languages together — but there are a few options that'll work.

Debugging Python and C/C++ With Eclipse

Eclipse, and its variants such as PyDev and LiClipse, provide strong environments for debugging Python or C/C++ when the Python and C/C++ plugins are installed. The steps below describe one way of debugging both languages within one session.

SETTING UP A PYTHON AND C/C++ DEBUGGING SESSIONS WITH ECLIPSE

There may be several ways to set up your Python and C/C++ debugging sessions, but one approach that'll work is to start with a Python debugging session and attach to the Python interpreter with a C/C++ debugging session.

1. Create a Python project for your program and set up a Python Run Debug Configuration. Since the Python extension was compiled for debugging, the Python interpreter used for debug configuration must be configured to use this debug version. For Python 2.7 on an Ubuntu 14.04 system this is python2.7-dbg. Make sure the Python debug project runs without any issues.
2. With the Python code debuggable, create a new C/C++ Attach to Application debug configuration. There's nothing special needed to be set up for the configuration other than creating a new one.
3. To begin the Python debugging session, start by setting a few breakpoints in your Python code near where you'll be making calls into C/C++ and then start the Python Run debugging session.
4. Once a Python breakpoint near where a call into the C/C++ Python extension has been hit, start the C/C++, Attach to Application debug session that was set up in step two. This will present a dialog showing the processes on the system. Select your Python process and click OK.

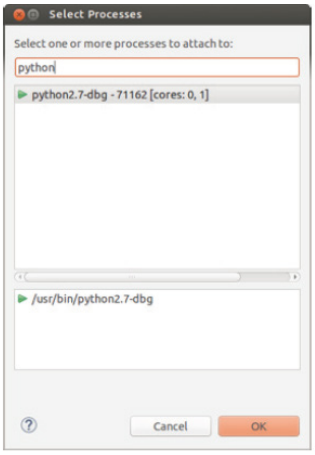


Figure 1: Select Python process to attach to

5. Open the C/C++ files you wish to debug and set breakpoints where you want to stop and then resume the C/C++ debugging session.

BEGIN DEBUGGING PYTHON AND C/C++ WITH ECLIPSE

At this point the Python debugger will be in control, waiting at the breakpoint hit in step three and at the point of the call into the C/C++ extension. Clicking Step will result in the C/C++ extension being called and the C/C++ breakpoint being hit. The Debug tab shows the stack traces for the Python Run debug session and the C/C++ Attach to Application session, but Eclipse won't automatically refocus to the thread that hit the breakpoint, so you must click on the thread.

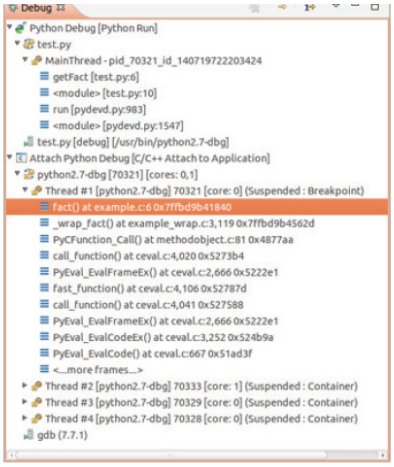


Figure 2: Python and C/C++ call stack — not integrated

At this point debugging in C/C++ can be done. You can examine variables, and compare them to Python variables if focus is changed back to the Python stack frame. To continue debugging the Python code, resume the C/C++ Attach to Application session and focus on the frame in the Python Run session.

Debugging Python and C/C++ With TotalView

TotalView supports debugging Python applications that use C/C++ extensions. The debugger doesn't yet support setting breakpoints and stepping the Python code, but it excels at the ease of setting up your debug session, examining the data exchange between the language barriers, and debugging the C/C++ code. The steps below show how to quickly debug Python and C/C++ code in TotalView:

SETTING UP A PYTHON AND C/C++ DEBUGGING SESSIONS WITH TOTALVIEW

Setting up a Python and C/C++ debugging session with TotalView is very easy, as shown in the steps below:

1. Begin the debugging session by selecting Program Session from the Start Page. Provide the path to the debug Python interpreter and the name of the Python file to run as an argument to the interpreter. Click Load Session.

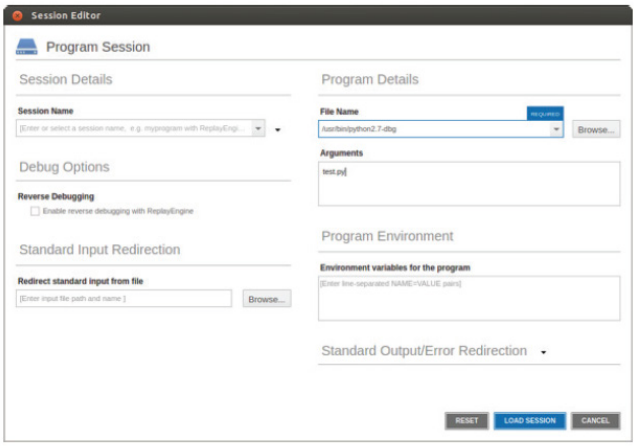


Figure 3: Set up Python debugging session

For even faster startup, provide the information as command line arguments to TotalView:

```
totalview -newui --args /usr/bin/python2.7-dbg test.py
```

2. Set breakpoints in the C/C++ Python extension code by simply using the At Location dialog, available through the Action Points | At Location menu choice. Enter a Python extension file#line location and click Create Breakpoint to create the breakpoint in the specified file and line number.

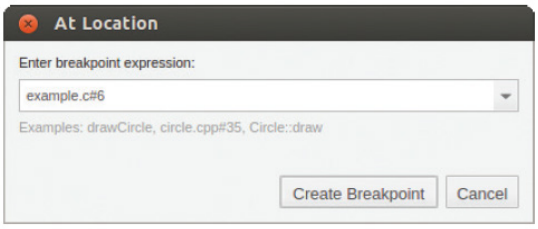


Figure 4: Create a breakpoint on Python extension function

BEGIN DEBUGGING PYTHON AND C/C++ WITH TOTALVIEW

With the Python session set up and a breakpoint set in the Python extension, simply start running the Python interpreter. TotalView will stop when your breakpoint is hit.

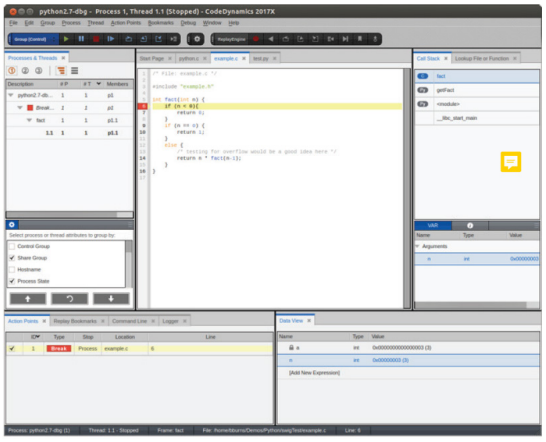


Figure 5: Stopped at Python extension function and clean integrated call stack

One of the nice features with TotalView is that it provides a fully-unified Call Stack of all the Python and C/C++ frames. It also removes all the noisy calls that tie the two languages together, giving a concise, developer-oriented view of the call from Python into C/C++.

Comparing the data from both sides of the language barriers is easily accomplished by clicking on either the C/C++ or Python frame and looking at the values of the local variables in the VAR panel. Variables can be dragged into the Data view to be examined and compared.

Finally, TotalView provides powerful debugging features, such as its reverse debugging engine, ReplayEngine, which allows developers to record the execution of the debugging session and jump back through and debug the execution to understand how the program ran.

Eclipse vs. TotalView

Debugging Python or C/C++ in Eclipse works well with dedicated perspectives, but when combining both into one debugging session, many workflow issues come up. Developers need to spend time setting up a Python and C/C++ debugging session. Once the session is going, Eclipse doesn't provide a unified stack trace across the language barriers, and worse, all the code layers between the languages clouds the program's flow. Developers can examine the Python and C/C++ variables in the different language barriers, but comparing them to ensure data integrity is hard since only the variables for the select language appear when selecting a language stack frame. You can't easily see both at the same time.

The TotalView solution provides a very easy workflow for establishing your Python and C/C++ debugging session. Setting a breakpoint in your C/C++ code is straightforward and when hit, a nicely integrated Python and C/C++ stack trace is presented. The code between the language barriers is stripped away, leaving the call flow the developer expects. It's easy for developers to drag Python and C/C++ variables into the Data View and examine their values, making sure that the data has crossed the language barriers correctly.

Get the Tool Your Developers Need

Python continues to grow in many industry segments, including deep learning and artificial intelligence. Applications such as TensorFlow are examples of complex programs that combine Python with C/C++ extensions, but debugging the mixed language Python and C/C++ programs is a challenge for developers. The tools developers use are beginning to make it easier for developers to debug these mixed language programs. The Eclipse solution enables full Python debugging but its environment is cumbersome to set up and debug within. TotalView provides a fast and convenient way to debug your Python and C/C++ programs, enabling quick understanding of the interchange between the language barriers and debug the data exchange. This, coupled with its advanced C/C++ debugging capabilities such as reverse debugging, make it a solid solution for debugging the execution paths and data exchanges between Python and C/C++. As the industry use of Python continues to grow, more complex applications tools like these will need to advance to provide the debugging capabilities required by developers.

TRY TOTALVIEW FOR FREE

totalview.io/free-trial

About Perforce

Perforce powers innovation at unrivaled scale. With a portfolio of scalable DevOps solutions, we help modern enterprises overcome complex product development challenges by improving productivity, visibility, and security throughout the product lifecycle. Our portfolio includes solutions for Agile planning & ALM, API management, automated mobile & web testing, embeddable analytics, open source support, repository management, static & dynamic code analysis, version control, and more. With over 15,000 customers, Perforce is trusted by the world's leading brands to drive their business critical technology development. For more information, visit www.perforce.com.