

EVTEK University of Applied Sciences  
Institute of Technology  
Degree Programme in Information Technology

**Gergely Erdélyi**

**IDAPython: User Scripting for a Complex Application**

Final Year Project. 20 April 2008

Instructor: Mika Ståhlberg, Program Manager

Supervisor: Hannu Laine, Principal Lecturer

Language advisor: Taru Sotavalta, Senior Lecturer

Author	Gergely Erdélyi
Title	IDAPython: User scripting for a complex application
Number of Pages	45
Date	20 April 2008
Degree Programme	Information Technology
Degree	Bachelor of Engineering
Instructor Supervisor	Mika Ståhlberg, Program Manager Hannu Laine, Principal Lecturer
<p>Developers of today's increasingly complex software packages have to find the delicate balance between the required feature set and implementation time. One way to deal with the issue is to provide end users with means to extend and customize the software to their own needs. Application scripting is the easiest way to involve the users in customisation, with a low entry barrier.</p> <p>The goal of this final year project was to integrate the Python programming language as a user scripting language into the Interactive Disassembler Pro, also known as IDA Pro, the de-facto standard disassembly tool of the computer security industry. The project is called IDAPython, and the software was implemented as plug-in to IDA Pro.</p> <p>The IDAPython project used Python as a scripting language and the Simplified Wrapper Interface Generator (SWIG) for interfacing the interpreter to the host application. This report details the design, implementation and issues related to interfacing Python to a complex Application Programming Interface through SWIG, targeted for three different software platforms.</p> <p>Over the years IDAPython has become a powerful tool, popular among security researchers around the world. IDAPython is available in binary and source code form for free.</p>	
Keywords	application scripting, python, ida pro, swig, embedding

## **Acknowledgements**

As this report is the culmination of many years, I would like to thank a number of people without whom I could not have succeeded. First and foremost, I thank my family for everything I am, and Ilona for her patience when I could not be there.

I would also like to thank Katrin Tocheva for convincing me to go back to school, however crazy that sounded at the time. Without Sami Rautiainen, school work would have been a lot less fun and a lot harder, if not impossible.

Mika Ståhlberg, Hannu Laine and Taru Sotavalta read several versions of this report and provided invaluable comments, I am grateful for that.

The IDAPython project would not have been possible without the help and encouragement from the creator of IDA Pro, Ilfak Guilfanov, and the earliest and most loyal IDAPython users, Ero Carrera, Pedram Amini and Jarkko Turkulainen. Thanks go to Mikko Hyppönen for allowing me to release the IDAPython code to the public.

Last, but not least, I would like to thank all the users of IDAPython who have provided me with much valued feedback over the years.

## Contents

Abstract	2
Acknowledgements	3
1 Introduction	6
2 Application Scripting	7
2.1 Introduction to Application Scripting	7
2.2 Emacs	8
2.3 Blender	8
3 IDA Pro	10
3.1 Architectural Overview of IDA Pro	10
3.2 User Programming of IDA Pro	11
4 Evaluation Criteria	13
4.1 End-user Usability Requirements	13
4.2 Development Requirements	14
4.3 Other Requirements	14
5 Solution	15
5.1 Evaluation of Scripting Languages	16
5.1.1 Custom Language Implementation	16
5.1.2 Perl	16
5.1.3 Python	17
5.1.4 Lua	17
5.1.5 Scheme	18
5.1.6 Summary and Result of Language Evaluation	18
5.2 Evaluation of Wrapping Methods	19
5.2.1 Introduction to Wrapping	19
5.2.2 Manual Wrapping	19

5.2.3 Simplified Wrapper Interface Generator	20
5.2.4 Boost.Python	20
5.2.5 SIP	21
5.2.6 Summary and Result of Wrapping Method Evaluation	21
5.3 Interfacing a C/C++ API to Python with SWIG	22
5.3.1 SWIG interface files	22
5.3.2 SWIG and the IDA Pro API	23
5.3.3 Pointer and String Handling	24
5.3.4 Pointers to Integers and Small Arrays	26
5.3.5 Input/output Parameters	26
5.3.6 Callbacks and Notification Hooks	27
5.3.7 Other Wrapping Issues	31
5.3.8 Preparation of the Header Files for SWIG	31
5.4 Embedding Python into IDA Pro	33
5.5 IDAPython Modules	34
5.6 Building and Distribution	37
5.7 Documentation	38
6 Results	39
6.1 Implementation Results	39
6.2 User Experience	40
6.3 Further Development Directions	40
6.4 Availability	41
7 Conclusion	42
References	43

## 1 Introduction

With time computer software is becoming more and more complex. Developers of any software package have to find the delicate balance between the feature set demanded by the users, and the time it takes to implement all of those features. Another, probably inevitable, fact is that no matter how many features and use cases the developers implement, there will be someone who would need yet another feature or prefer to use the software differently.

One way to deal with complex user requirements is to design the application to be flexible, and provide the end users with means to extend and customize the software according to their own needs. This way the application serves as a base and framework for the customers to solve their own problems in the most suitable way. The amount of extensibility needed depends on the application and, more importantly, the target user base. There are certain users, often called “power users”, whose needs are more demanding and require extensive customisation.

The subject of this final year project is IDAPython, which aimed to integrate the Python programming language as a user scripting language into the Interactive Disassembler Pro, also known as IDA Pro, the de-facto standard disassembly tool of the computer security industry.

The IDAPython project was started in April 2004 and has since been in steady, but sometimes slow, development. IDAPython was first introduced to the public in 2004, in a joint paper with Ero Carrera [1]. Since its inception, the project’s main author and maintainer has been the author of this report, with occasional, and welcome, contributions from users.

This report outlines the motivation, study, implementation and learnings over the years of the IDAPython project. The main focus is on the issues related to the integration of a scripting language into a complex, real-world application that runs on several different platforms.

## 2 Application Scripting

### 2.1 Introduction to Application Scripting

User-extensible applications are not a new phenomenon. Ever since complex user applications appeared, many of them provided some form of extensibility by either a plugin interface or user scripting. A scripting facility can easily take the application to a whole new level, by turning it from a rigid object into a flexible platform that users can build on to solve their problems.

Uses of user scripting range from the automation of mundane tasks with a macro facility, to the addition of new features through a plugin interface. Macro facilities tend to be rather simplistic, practically allowing the recording and subsequent playback of a sequence of user actions. Plugin interfaces allow the user to extend the application with new functionality. The extent of plugin interfaces ranges from very specific, for example file format plugins, to far-reaching that allow plugins to tap into the application and extend it.

Application scripting is a form of a macro facility which can be extensive enough, even to be used as a plugin interface as well. The main distinguishing attribute of application scripting is that it employs a high-level scripting language.

Using a scripting language has several benefits. Scripting languages are often dynamic, very high-level and interactive which allows the user to concentrate on the problem at hand instead of implementation nuances. The other advantage of scripting languages is that they are interpreted which means that program failures will not result in catastrophic crashes and data loss.

On the flipside, interpreted languages tend to be slower, hence not always best-suited for performance-intensive tasks. However, it is worthwhile to note that in the age when a typical personal computer already has more processing power than its user needs, trading execution speed for user productivity is easily justifiable.

## 2.2 Emacs

GNU Emacs is highly customizable and extensible text editor that runs on all modern, mainstream operating systems [2; 3]. The appeal and popularity of Emacs spans over three decades and still lasts today. The distinguishing feature of Emacs is its virtually unlimited extensibility and customisability, which makes it adaptable to even the most recent requirements.

Emacs itself was written in a dialect of the Lisp language, called Emacs Lisp, also known as Elisp. Making Elisp accessible to end users, provides a means for solving problems and adding features that the original authors would have never anticipated. Today there are literally hundreds of packages available for Emacs, ranging from programming language support, to fully featured email clients and even a web browser. In fact, Emacs is so feature-rich, that it is often jokingly referred to as “a great operating system, lacking only a decent editor”.

An interesting aspect of Emacs is that most of the application itself is implemented in the same language and environment as the user additions [4]. This makes scripts written by end users equal to those provided by developers. End user scripts also possess about the same power and reach, allowing easy addition of even the most unexpected features.

## 2.3 Blender

Blender is a free, open source 3D modelling and rendering and compositing, application. Blender is available for all major operating systems under the GNU General Public License. [5]

Blender is a recent example of a user scriptable application. The developers of Blender chose Python as the scripting language. Using the Python scripting Application Programming Interface (API), Blender has been extended with numerous import and export modules as well as complex modeling and animation scripts, written in pure Python. The API provided by the application allows the user to control almost all aspects of the 3D environment, including models, textures, lighting, animation and rendering features.

The addition of Python scripting capability has arguably raised Blender to the level of the best-of-breed 3D content creation suites. Built-in scripting allows an artist, even with limited programming skills, to create complex scenes and animations. Scripting the content creation process makes it possible to experiment with different settings and combinations in a repeatable manner.

It is worth noting that since Blender is an open source application, it is possible to tailor it to specific needs by modifying its source code. However, when it comes to end-user extensibility, the application base code tends to be too low-level and difficult to master for people who are not software developers. Thus, it is beneficial to provide a higher-level environment that hides the internal intricacies of the application.

## 3 IDA Pro

IDA Pro is a multi-purpose disassembler that supports dozens of processor and file formats [6]. It is arguably the most commonly used code analysis tool in the security research industry. Many security researchers use it on a daily basis to reverse engineer binary code for different purposes.

While IDA Pro is still a disassembler at heart, today it features a host of advanced code analysis tools and also a runtime debugger. The power and versatility of IDA Pro has been proven over its history, which spans over a decade and a half.

### 3.1 Architectural Overview of IDA Pro

IDA Pro is a highly modular system with most of the functionality implemented as plugins, as shown in figure 1. The modular nature of the application provides great flexibility and extensibility. Today, IDA Pro comes with over a hundred different plugins for different processors, file formats and other functionality. The extensibility of IDA Pro does not stop there, as a Software Development Kit (SDK) for the plugin API is also available. Users can develop plugins to support new processors, file formats and other useful functionality they might need.

The user interface of IDA Pro is also separated from the rest of the application. There is a separate text-mode and a graphical user interface available, both of which use the same kernel modules to operate. Plugins and all functionality work nearly identically under both user interfaces.

In the heart of the application is the kernel, a complex code analysis engine with several sub-modules. It uses the file format loaders to load the file to be analysed and the processor modules for disassembling the actual code.

The kernel makes use of so-called type libraries and FLIRT signatures. Type libraries are pre-parsed C/C++ header files that can be used to define C function prototypes, structures and constants in the disassembly. FLIRT, which stands for Fast Library Identification and Recognition Technology, provides a database of signatures for the library code that is commonly found in compiled binaries [7]. The identified common library code needs no analysis, which leaves more time to focus on the interesting code.

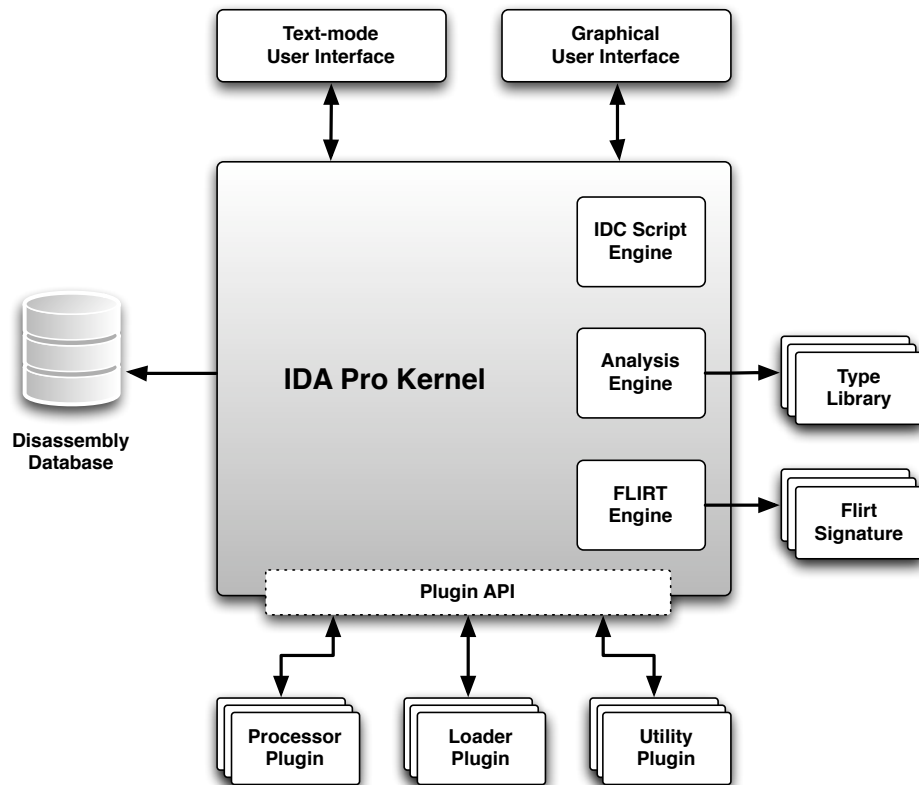


Figure 1: Architecture of IDA Pro

### 3.2 User Programming of IDA Pro

The course of binary code analysis often involves a set of repetitive tasks, ranging from simple cleanups to complex analysis tasks. Typical tasks would be, for example, renaming a number of symbols at once, or comment code sequences based on certain criteria. Anything that helps to automate these can save a significant amount of user time.

IDA Pro provides a built-in scripting language, called IDC, with C-inspired syntax and control structures. IDC has an extensive API, of over four hundred functions, for different disassembly and code analysis tasks. While it provides a familiar environment for C programmers, it lacks the advanced features and expressiveness of modern scripting languages. Furthermore, since IDC is specific to only one application, the number of available extensions and third-party modules is rather limited.

For users whose problems require capabilities beyond IDC, IDA Pro offers the extensive plugin API. Using the API, IDA pro can be extended in new ways, with nearly complete control over the application. IDA Pro has a group of active users, who have created a considerable number of plugins for different purposes [8].

The two standard ways of extending IDA Pro are either IDC scripts, or binary executable plugins through a C/C++ interface. IDAPython provides a third alternative, writing extensions in Python through the plugin API, which is in the focus of this report.

## 4 Evaluation Criteria

### 4.1 End-user Usability Requirements

Before embarking on a project, one of the most important tasks is to lay down the evaluation criteria for both the tools and partial results during the project as well as the final result of the completed effort. The criteria is separately specified for end-user usability and development requirements for the tools and modules to be used. End-user requirements will be the main criteria for the evaluation of the success of the project, while development and other requirements serve as a reference for selecting tools and components during the development process.

The first aspect of specifying end-user requirements is to define the target user base. In the case of IDAPython the users are experts at reverse engineering, often proficient in several programming languages. This makes the selection of the programming language easier, as previous programming experience or even familiarity with the particular language is very probable.

Although the user base consists of experts mostly, the extension language should be easy to learn and remember. At the same time, power users tend to be concerned with the power the language environment has to offer; functionality and expressiveness cannot be compromised.

The language used for end user scripting must come with complete documentation, including quick start guides and tutorials for beginners. Also, the user environment must be tolerant towards user errors, all error situations must be handled gracefully. In such an environment no complete application failure or, worse yet, a crash, can be tolerated. The user expects to get an error message at worst, and continue from where the work was interrupted.

Another important aspect is the availability of ready-to-use code libraries and modules. An extensive software library may cut the development time to a fraction of what it would take to implement everything from scratch.

## 4.2 Development Requirements

An equally important set of requirements is one related to the technical development aspects of the project. It is of paramount to choose tools and components that suit the problem at hand and do not stand in the way. Ease of development is especially important for projects, such as IDAPython, that have a single developer, responsible for all development, testing and administration efforts.

When choosing a scripting language as an extension language, the most important aspect is that it must be easy to integrate into the target application and interface to its functionality. Some languages make integration more difficult, some easier, some are specifically designed to be embedded in applications and provide easy interfacing capabilities.

Other technical aspects must also be evaluated, such as memory needs, and the ability to work in multi-threaded environment and platform support. The tools used must work reliably on the same platforms the application does.

## 4.3 Other Requirements

Beyond end-user and technical requirements, other aspects must be considered. One of them is the licensing of the tools. It is essential that the licensing terms allow distribution and use in the same manner as the main application does. The licensing terms must at be least as permissive as the application itself. The exact details of licensing and legal considerations are beyond the scope of this report. However, most free and open source software comes with reasonably clear and well understood licensing terms.

Another important consideration is the community and support around the tools. Since these small integration projects have no manpower to provide support and maintenance for all the components themselves, it is important that all the used components have established communities around them filling that need. This way the developer can concentrate on the issues specific to the project and rely on the communities for external components.

## 5 Solution

The essential idea behind the design is that the scripting language will be embedded into the host application and at the same time extended with functionality from the host. Figure 2 illustrates the concept of integration of a script language into a host application.

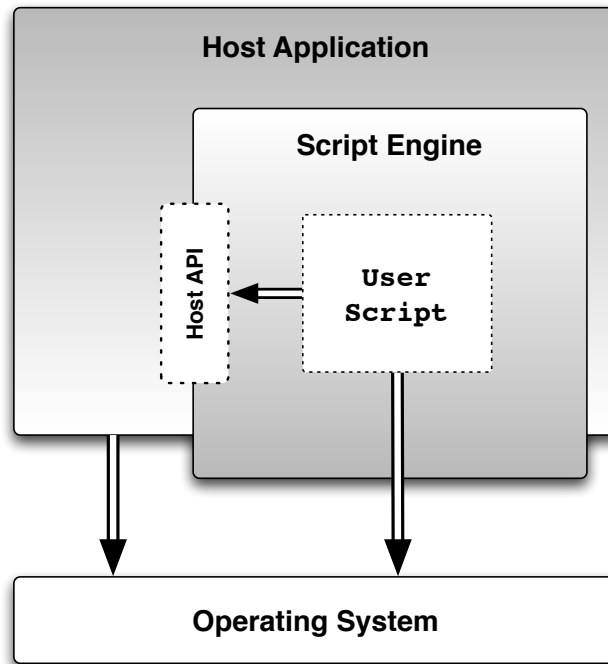


Figure 2: Script Engine Integration

As shown in figure 2, the script engine is running inside the application, with direct access to both the operating system and functionality from the application. For the script engine to be useful, the application must provide the scripts with an interface similar to the rest of the application. The amount of functionality that needs to be exported from the host application depends on the nature and extent of the user scripting to be provided. As more application code is exposed to the scripting engine, the user scripts will also provide more power. However, interfacing more functionality requires more development, and hence balance must be found in-between.

## 5.1 Evaluation of Scripting Languages

The first step of implementing application scripting is the decision on what scripting language will be used. Two fundamental choices are to either design and implement a custom programming language or use one of the many existing ones. The evaluation criteria for the scripting languages has been outlined in chapter 4. The important aspects to consider are maturity, the size of user community, the distribution license and the ease of embedding the language into applications. Many of these are arguably subjective, yet important and should be evaluated with common sense.

### 5.1.1 Custom Language Implementation

One of the obvious choices is to implement a custom scripting language interpreter for the application. The main benefit of the approach is complete control over the language, its features and runtime. Since the code is owned entirely by its developer, there are no licensing issues to consider.

On the flip side, a custom interpreter is often prohibitively expensive to implement and maintain. At the same time, unless the custom language is based on some established one, the lack of prior user experience might make the learning curve too steep.

### 5.1.2 Perl

Perl is a dynamic programming language, created by Larry Wall and first released in 1987. The language itself borrows from C, awk, shell scripts and many other languages. Over the years Perl has been ported to almost all operating systems capable of running it. The environment supports all features expected from modern programming languages, such as unicode, database access, and threading. [9,1-4]

Other distinguishing features of Perl are its strong user community and extensive software library, called Comprehensive Perl Archive Network, that hosts over twelve thousand software packages for Perl [10]. The only apparent disadvantages of Perl are its often cryptic syntax, and the relative difficulty to embed and extend it.

### 5.1.3 Python

Python is a dynamic, object-oriented scripting language, designed by Guido van Rossum in 1991. Its clean syntax and high-level data structures make it an ideal prototyping and rapid application development language [11]. There is one main implementation of Python, often referred to as CPython, and a few alternative versions, for example Jython, written in Java [12], and IronPython, which was written for Microsoft .NET platform [13].

One of Python's mottos is "batteries included". The language comes with modules to solve a wide range of problems, from text processing to network programming. In recent years Python has gained a considerable user base among the reverse-engineering community, which results in even more available software in that domain.

One possible disadvantage of Python is its significant-whitespace syntax, which some people dislike. The syntax is hardcoded and thus not possible to alter. Furthermore, because of its interpreted nature, the performance of Python programs most often cannot match natively compiled code. However, the execution speed of Python code can be enhanced with a just-in-time compiler, such as Psyco, that might yield a several-fold improvement, depending on the algorithm and workload [14].

### 5.1.4 Lua

Lua, which is Portuguese for moon, is a language specifically designed for extending applications. It is a light-weight language, still supporting object-oriented programming and high-level data structures. [15] Lua is written in plain ANSI C and was first released in 1993 under a BSD license, which was later changed to an MIT license, starting from version 5.0 [16; 17].

The main advantages of Lua are the ease of embedding and its modest memory footprint. The disadvantages are the relatively small user community and the lack of readily available software.

### 5.1.5 Scheme

Scheme is one of the two main dialects of Lisp, the other one being Common Lisp. First designed and implemented in the 1970s, Scheme is the grandfather of the other languages discussed in this chapter. The language itself is dynamic, and it supports object-oriented programming and high-level data structures. The features that distinguish Scheme from other languages is its clean, though for many programmers scary, syntax and very flexible macro system. There are dozens of implementations of Scheme, many of them freely available.

On the positive side Scheme is a mature language with several advanced features that many programming languages still lack. On the other hand, the simple but parenthesis-heavy syntax of Scheme scares a many programmers away, which makes the user community relatively small. Consequently, the library of available Scheme software is also somewhat limited.

### 5.1.6 Summary and Result of Language Evaluation

Table 1 summarizes all the important aspects of each scripting language. Some of the attributes are arguably subjective. However one's choice of programming language often is subjective.

Table 1: Comparison of Scripting Languages

	Custom	Perl	Python	Lua	Scheme
Implementations	one	one	several	one	several
Maturity	immature	mature	mature	mature	mature
Implementation effort	major	moderate	low	low	low
Available software library	nothing	very extensive	extensive	fair	fair
Existing user base	none	very large	large	medium	medium
License	own	Artistic/GPLv2	PSFv2	MIT	several

After careful weighting of the properties of the different languages, Python was selected to be the scripting language for the project. The choice fell on Python primarily because of its easy embedding and extension interface, comprehensive software library and its clean syntax. Its interpreted nature makes it a reliable stable environment, where errors result only in error messages, and full application crashes are almost nonexistent. Also, the Python environment is distributed with extensive, well-written documentation, starting from a beginner's tutorial to detailed specifications for the language and all included modules.

## **5.2 Evaluation of Wrapping Methods**

### **5.2.1 Introduction to Wrapping**

Wrapping is the process of making functionality written in a lower-level language, typically C or C++, available to a higher-level, scripting language environment. In case the scripting language is embedded, the external functionality to be wrapped is provided by the host application itself. The wrapper code is responsible for converting the data types and calling conventions between C/C++ and the scripting language.

In the case of a simple application, almost any wrapping method is sufficient, but in case the host application is complex, wrapping all the needed functions and data can be time-consuming. Thus the importance of choosing the wrapping method that best fits the application, is paramount.

Since IDAPython is a plugin for a commercial, closed source, proprietary application, licensing of the used tools must be evaluated as well. The license of the tool must allow integration of the code into such an application, and also commercial use without written permissions or licensing fees.

### **5.2.2 Manual Wrapping**

Manual wrapping means that the glue code that converts code calls and data between the C/C++ code and the scripting environment is created manually. For each piece of data, constant or function to be exposed, a separate conversion code needs to be written and maintained.

Manual wrapping is the most powerful and accurate wrapping method, as it allows complete control over the code and data conversion, down to the smallest detail. Consequently, for the same reason, manual wrapping is also tedious, error prone and takes too long to be practical. Since most of the wrapper functions are very similar, their creation can be automated to a large extent.

In some cases, if the requirements are special or strict, manual wrapping might be the only way. For those cases, the Python documentation includes an extensive guide and a full reference on the topic of extending and embedding the language [18].

### **5.2.3 Simplified Wrapper Interface Generator**

Simplified Wrapper Interface Generator (SWIG), is a tool that automates the tedious process of creating the vast amount wrapper code needed to interface low-level code to scripting environments. SWIG supports more than 15 different script languages, Perl, Python, Ruby, Lua and many others. [19]

The greatest advantage of SWIG is that it is capable of parsing regular C/C++ header files for function and data declarations to be wrapped. In most cases the original API headers can be used with small modifications. In the case of a complex API, such as the IDA Pro API, this greatly simplifies the problem by removing the need to maintain two separate API definitions.

SWIG's license is liberal and allows commercial and closed-source distributions without an explicit, written license agreement.

### **5.2.4 Boost.Python**

Boost.Python is a framework that simplifies the interfacing of C++ code to Python [20]. The main difference from SWIG is that Boost.Python is not capable of using the original header files, and thus it requires custom definition for all the functions and data to be wrapped. Though Boost.python has extensive support for the less frequented corners of C++, it requires manual definition of all code and data, which makes it a suboptimal choice for wrapping a large API.

Boost.Python has a liberal distribution license that allows use in commercial applications.

### 5.2.5 SIP

The SIP wrapping tool was originally designed to help interfacing the Qt GUI toolkit to Python [21]. Since Qt was written in C++, SIP has strong support for advanced C++ features. Unfortunately it suffers from the same drawback as Boost.Python: it requires manual annotation of the headers to be wrapped.

SIP comes with the same distribution license as Python, making it usable in all scenarios where Python itself is.

### 5.2.6 Summary and Result of Wrapping Method Evaluation

Table 2 collects the most important attributes of all wrapping methods examined in section 5.2. After the evaluation of the different methods, the choice fell on SWIG, primarily for its ability to use the original API headers with few modifications.

Table 2: Comparison of Wrapping Methods

	Manual Wrapping	SWIG	Boost.Python	SIP
Ease of use	very difficult	easy	easy	easy
C support	good but tedious	excellent	-	-
C++ support	possible but difficult	fair	excellent	excellent
Use of header files	-	yes, with minor changes	-	yes, with extensive changes
Target languages	any	Python, Perl, Lua, etc.	Python	Python

## 5.3 Interfacing a C/C++ API to Python with SWIG

### 5.3.1 SWIG interface files

SWIG makes wrapping of C/C++ APIs easier by automating the dull task of creating similar but slightly different wrapper functions for all code and data that is exposed to Python. SWIG is fairly versatile and capable of handling code and data found in most APIs today. However there are a number of special cases and details that need to be considered.

Functions and data to be wrapped are described to SWIG in so-called interface files, which most commonly have a `.i` extension. Figure 3 demonstrates a simple interface file. Interface files are essentially C/C++ headers with additional directives for SWIG. Declarations of C/C++ functions and data are almost verbatim copies of those in the original header files. However, SWIG's own parser has its limits, which is discussed in more detail in section 5.3.8. Just the like C/C++ preprocessor, SWIG is case-sensitive for names.

```
/* Define a SWIG module name */
/* This module can be used in Python with 'import test' */
%module test

/* Define a constant */
/* In Python: test.TEST_VALUE */
#define TEST_VALUE 1

/* Declare a function with an int input and output */
/* In Python: test.testfunc() */
int testfunc(int x);
```

Figure 3: Simple SWIG interface file

For its own purposes, SWIG introduces a number of special directives, which start with a percent symbol. The most essential one is the `%module` directive, which tells SWIG how to introduce the extension module to Python. There are other directives; for example, `%include` for code inclusion, `%apply` and `%clear` for precise type definition and many others. The SWIG manual provides a detailed list of all the available directives and their use [22].

The interface files are processed with SWIG, which creates a C or, if requested, C++ source file with the required wrapper code. The wrapper source file can be compiled and linked just like any other source file of the application. It is possible to link the wrappers either dynamically for later loading, or statically straight into the application binary.

### 5.3.2 SWIG and the IDA Pro API

The IDA Pro plugin API carries a development history that extends over a decade. Because of that, the API is a mixture of different programming paradigms and, to some extent, even languages. The API touches a spectrum ranging from a C-like procedural, to object-oriented C++ class interfaces.

While SWIG is capable of handling wrapping most C/C++ constructs, in certain cases it needs help in wrapping code or data that cannot be processed from the header file unambiguously. Those cases include functions that pass memory pointers as input/output argument, complicated macros and function callbacks.

Figure 4 shows a simple function declaration with a scalar input and output value. By following the typedefs `ea_t` is resolved to a basic unsigned long C type. The function expects an unsigned long value and returns one with the same type. SWIG has no problem converting both the input and output values between the C and Python environments.

```
typedef unsigned long  ulong;
typedef ulong ea_t;

inline ea_t idaapi get_item_head(ea_t ea);
```

Figure 4: Function with a simple argument and return value

For all data type conversions that SWIG can not handle on its own, it provides a facility called typemaps. Typemaps are used to specify how a certain data type should be converted when passed in or returned from a C/C++ function. SWIG typemaps can contain arbitrary C/C++ code to facilitate precise conversion of complex data types. SWIG typemaps are a rather complex subject, and a full discussion is beyond the scope of this report. The SWIG manual contains all the necessary information about typemaps [22].

### 5.3.3 Pointer and String Handling

The main problem with wrapping C pointers is that from the function declaration it is not possible to find out how the memory block at the pointer is used by the called function. The data at the pointer might be a single value or an array of values, with its size specified in a separate argument.

SWIG handles the most common pointers gracefully. For example `char *` is automatically converted from and to Python strings. However, Python strings may contain zero bytes, which C allows only as a terminating character. Because of that, proper handling of strings requires careful consideration.

Since C allows only a single return value, functions often return strings and other binary data in user-supplied buffers. A commonly used declaration is shown in figure 5. If the call to `get_manual_insn()` succeeds, the resulting string will be copied to `buf` with the maximum length of `bufsize`. The function returns the address of `buf` on success and `NULL` when it fails. This calling convention does not map directly to Python as it has no notion of pointers.

```
C declaration:
char *get_manual_insn(ea_t ea,
                     char *buf, size_t bufsize);

Python version:
def get_manual_insn(ea):
```

Figure 5: Function returning data in user-supplied buffer

The optimal way of wrapping a function that returns data in a buffer is to remove the buffer parameter altogether, and return either the string on success or `None` on failure. Before the function is called, the buffer is automatically allocated for the output data. When the function returns, the buffer is converted to a Python string, and then automatically deallocated. It is the Python string that is finally returned to the Python script. The typemap in figure 6 implements this method. The current implementation implicitly limits the length of the buffer to `MAXSTR` which is defined to be 1024. In the future that could be changed so that the size parameter would be an optional second parameter to the function.

The `cstring_output_maxstr_none` typemap is a general purpose map. It can be applied to several different functions that work similarly. These types of macros work in a syntactic level. When the SWIG parser encounters a `char *buf, size_t bufsize` combination in a function declaration, it removes them and applies the appropriate typemap. The Python version of the function does not have the output parameter at all, as that is provided by the typemap implicitly.

```

#define %cstring_output_maxstr_none(TYPEMAP, SIZE)
%typemap (default) SIZE {
    $1 = MAXSTR;
}
%typemap(in,numinputs=0) (TYPEMAP, SIZE) {
    /* Allocate buffer for the string */
    $1 = ($1_ltype) new char[MAXSTR+1];
}
%typemap(out) ssize_t {
    /* REMOVING ssize_t return value in $symname */
}
%typemap(argout) (TYPEMAP,SIZE) {
    /* Convert the result to Python string */
    if (result > 0)
    {
        resultobj = PyString_FromString($1);
    }
    else
    {
        /* ...or return None on failure */
        Py_INCREF(Py_None);
        resultobj = Py_None;
    }
    /* Deallocate the buffer */
    delete [] $1;
}
#endif

```

Figure 6: Bounded string typemap

For example a `%cstring_output_maxstr_none(char *buf, size_t bufsize);` statement applies the typemap to all functions with a combination of `char *buf, size_t bufsize` input arguments.

### 5.3.4 Pointers to Integers and Small Arrays

In certain places the IDA Pro API makes use of pointers to single values and small, static sized arrays. SWIG provides a convenient way to handle these cases. After including the 'carrays.i' and 'cpointer.i', pointers and arrays of certain simple types can be created.

```
%include "carrays.i"
#include "cpointer.i"
%array_class(ea_t, eaArray);
%pointer_class(int, int_pointer);

// Usage in Python:
// intp = int_pointer();           # Allocate space for the pointer
// intp.assign(1)                 # Set the value at the pointer
// some_func(intp)                # Pass in the pointer to int
//
// tidarray = tidArray(5)         # Create an array with 5 items
// tidarray[0] = 1                # Set an item in the array
// some_func(tidarray.cast())     # Pass in the pointer to the array
```

Figure 7: Simple pointers and static arrays

Figure 7 shows the definition and use of simple pointers and arrays. Using the automatically generated class wrappers, the content at the pointer, or in the array, can be accessed easily. Such objects can be passed as pointer parameters both for input and output parameters.

### 5.3.5 Input/output Parameters

Since in C/C++ a function can return only a single value, developers use pointer parameters to return several values. The function returns one value regularly and writes the second one to the memory at the pointer argument. To accommodate such a calling convention, SWIG implements a special directive called %apply.

```
// Declare arg2 as output parameter
%apply unsigned char *OUTPUT { uchar *arg2 };

bool func(int arg1, uchar *arg2);

// Python usage:
// res1, res2 = func(1)
```

Figure 8: Input/output parameters

As shown in figure 8, the Python version of the function will return two values as a tuple, which is quite typical in Python. Using the `%apply` directive, arguments can be designated to be input, output or both. If the argument is only for output, it does not need to be specified in the Python version, because it will be allocated automatically and returned as an additional return value in a tuple.

The uses of `%apply` are more diverse than just input/output arguments. The `%apply` directive can also be used to enforce constraints and change types of specific arguments. Detailed information on the use of `%apply` can be found in the Argument Handling chapter of the SWIG User Manual [22].

### 5.3.6 Callbacks and Notification Hooks

Functionality that requires the use of callback functions presents a new set of challenges. SWIG does have limited support for callback functions but only ones that have been implemented in C. Specifying a Python function to be called is not possible. The function could explicitly call a Python function, but that requires the callback to receive a user defined data pointer and also, to convert all parameters from C to Python explicitly.

Figure 9 contains a simplified example of an event notification callback. When the callback is registered, an object pointer the Python function to be called is supplied as `user_data`. This allows the C layer to find the Python function, making it possible to have several Python functions registered at the same time.

```

/* Type of callback function */
typedef bool event_cb_t(void *user_data, int event);

/* Function to register a callback */
bool hook_events(event_cb_t *cb, void *user_data);

/* Callback */
bool my_callback(void *user_data, int event)
{
    /* 1, Pick up Python function object */
    /* 2, Convert event to Python type */
    /* 3, Call Python function */
    /* 4, Convert return value to C++ bool */
    /* 5, Return code */
}

```

Figure 9: Function with callback

In case the callback type does not have a dedicated `user_data` or other unique parameter, there can be only one function registered at any given time. For more functions, there must always be some callback parameter that uniquely identifies the appropriate callback function.

Even though the IDA Pro API does not make frequent use of callbacks, there is certain functionality that needs them. For that reason, proper handling of complex callback mechanisms has to be implemented. In order to find out whether this problem can be solved for all cases, the most difficult problem is tackled first: a notification hook that uses variable parameters according to the event types it receives. Figure 10 contains the type definition of the hook function. What makes this difficult to handle is the `va_list` va argument. In C/C++ `va_list` type arguments signal that the function is capable of accepting a variable number of arguments. It is the responsibility of the hook function to know how many, and what type of arguments it will receive in which case. This makes it impossible for SWIG to wrap such functions properly.

```
typedef int idaapi hook_cb_t(void *user_data,
                             int notification_code,
                             va_list va);
```

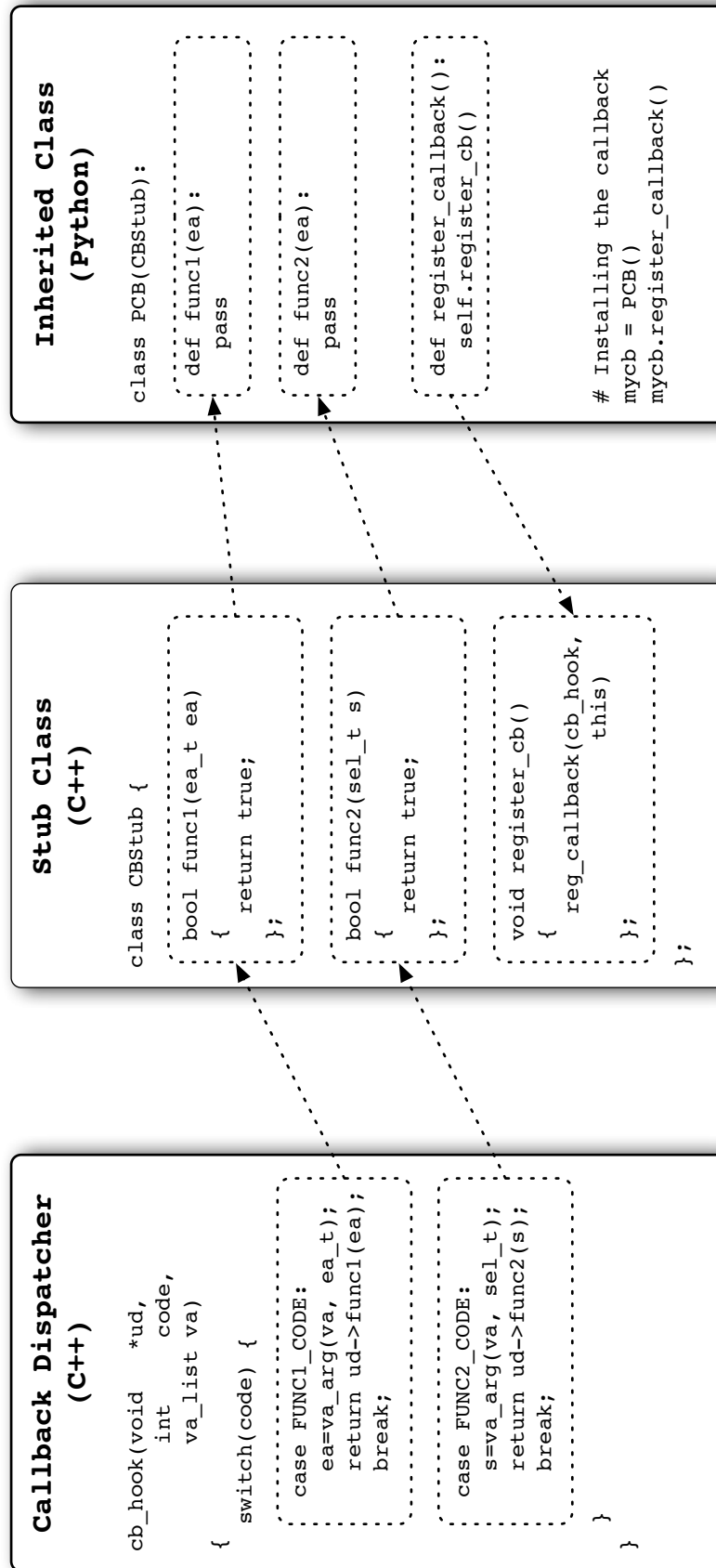
Figure 10: Notification hook with variable arguments

The only way to solve this problem is to write a wrapper function that picks up the proper number and type of arguments for each valid value of `notification_code` and call the appropriate Python function to handle the actual call. By default SWIG easily converts data types when calls are made from Python to C, but not the other way around. If a C wrapper needs to call into Python, it needs to convert the arguments and return value itself.

There is one, relatively new, feature in SWIG, called directors. The idea behind directors is to make it possible to support C++ classes that can be subclassed with Python code. Generally, once a C++ class has been wrapped, it can easily be subclassed with Python, but all the methods that had been overridden will be visible only from Python and not from the original C++ code. For example, if method `foo()` is overridden, its Python version will be called only from Python. If the one of the other C++ class methods attempts to call `foo()`, it will call the original C++ version of `foo()` instead. Further details of the implementation and usage of class directors are explained in the Cross language polymorphism section of the Python chapter in the SWIG Users Manual[22].

To alleviate this problem, newer versions of SWIG implement so-called director classes that check each virtual method if it had been overridden in Python and call that version instead, even from the C++ code. The greatest benefit of this feature is that all the C++ arguments will be automatically converted when doing the C++ to Python call, which was not possible otherwise. This can be used to implement the automatic data conversion required for variable argument callbacks, with a very thin wrapper code.

The ingredients to make this work include a hook class with virtual methods for each event, one simple wrapper that receives the event code and picks up the arguments and last but not least, SWIG directors. Figure 11 shows the complete call chain of both the C++ and Python side of the class with directors, the dispatcher callback and its registration.



**Note:**

The argument ud is a pointer to a CBStub class instance.

**Note:**

Members func1 and func2 are virtual that the director will dispatch to Python if they have been overridden.

Figure 11: Complex callback with class directors

### 5.3.7 Other Wrapping Issues

When wrapping an API with the size and complexity of IDA Pro, it is almost inevitable to encounter certain obstacles along the way. The API contains a number of constructs that make sense in C/C++ but can not be translated to Python one-to-one. For example, unsigned data types that are initialized with negative values. While this makes sense in C, where the integer types wrap around, Python uses signed long integers which can be arbitrarily large. In contrast to C, in Python `-1` is never considered equal to an unsigned integer, `-1` cannot be equal to `0xFFFFFFFF`.

One prominent example in the IDA Pro API is the `BADADDR` constant, an unsigned `ea_t` type, which is defined to be `-1`. While the C++ compiler turns `BADADDR` into an unsigned  $2^{32} - 1$  or  $2^{64} - 1$  value, SWIG turns the constant into `-1` for Python. For example, the C return value `0xFFFFFFFF` is a positive long integer in Python, and will never match the `-1` constant Python programs expect. The solution to this problem is to remove the original declaration and manually define the constants to the proper unsigned value, for example `BADADDR` to `0xFFFFFFFF` for 32-bit mode.

### 5.3.8 Preparation of the Header Files for SWIG

One of the most important benefits of SWIG is the possibility to use the original API header files for wrapping. However, the parser built into SWIG is unable to handle certain valid C++ constructs. For that reason the header files need preparation, in most cases a simple removal of problematic parts that confuse SWIG.

The IDA Pro SDK headers contain a number of internal symbols which are exported to the user plugins. Trying to wrap those symbols results in linking errors which must be excluded. Fortunately the SWIG parser defines several symbols when parsing the headers, which make it possible to conditionally remove parts of the header file. Using simple `#ifdef / #endif` preprocessor conditionals, the problematic code is made invisible to SWIG, yet retained for the C++ parser. As an example, the use of such a construct is depicted in figure 13.

There are certain declarations that work with the C++ preprocessor, but not with SWIG. The main reason is that while the CPP merely performs a textual replacement of a `#define`, SWIG has to apply more complex logic to find constants to wrap as well. For this reason the macro expansion in SWIG is not nearly as tolerant as C++. One such case is a con-

stant definition with `#define` in the middle of another declaration as shown in figure 12. It is easy to solve the problem by moving the `#define` outside, so that SWIG can parse it. This modification has no effect on the behaviour of the C++ parser.

Even though the use of preprocessor conditionals is a powerful and simple way of dealing with SWIG parser issues, the changes to the headers present another problem: maintenance. The IDA Pro SDK is rather large, about 38000 lines of code in 53 files that weight 1.6 MiB. Every time the vendor publishes a new upstream release, the local modifications must be merged, which is a time-consuming and error-prone process.

Fortunately many of the SWIG `#ifdefs` can be supplanted by `%ignore` statements. As the name suggests, SWIG will ignore any subsequent definitions with a specified name. Since `%ignore` statements are located in the SWIG interface files and not in the headers, they are convenient and unobtrusive, in case function or data definitions need to be removed for any reason. However, the preprocessor macros are still needed for cases when SWIG cannot parse a certain construct. Anything that can be placed in the SWIG interface files, means less modifications to the header files, and thus less to merge and maintain. The ultimate goal is to reduce the amount of required header modifications to the bare minimum.

```

/* SWIG can not parse this */
int function(int arg1,
            int arg2,
#define CONST1 1
#define CONST2 2
            int arg3);

/* ...nor this one ... */
enum constants {
    constant1,
    constant2,
#define OTHER1 1
#define OTHER2 2
    constant3
};

```

Figure 12: Problematic constant definitions

Another limitation of SWIG is that it has limited support for nested structures and unions. In case a structure contains an unnamed union member, the union declaration must be removed to make the union members also accessible.

```

class test
{
public:
#ifdef SWIG
    union {
#endif // SWIG
        int member1;
        int member2;
#ifdef SWIG
    };
#endif // SWIG
};

```

Figure 13: A nested union

Figure 13 demonstrates the handling of nested unions. It is quite apparent that this change makes the header more difficult to read, but at least it works transparently with both C++ and SWIG.

#### 5.4 Embedding Python into IDA Pro

Embedding Python into applications is quite simple. In most cases the Python interpreter is dynamically linked and initialised with a single call. Figure 14 shows a minimal example of embedding Python [18]. The code initialises the interpreter, runs two lines of Python code and shuts down the interpreter. IDAPython uses the same scheme, although in a more extended form.

```

#include <Python.h>

int main(int argc, char *argv[])
{
    Py_Initialize();
    PyRun_SimpleString("from time import time,ctime\n"
                      "print 'Today is',ctime(time())\n");
    Py_Finalize();
    return 0;
}

```

Figure 14: Minimal Python Embedding

The way to extend IDA Pro is to interface the new code as plugins to the application. IDAPython is no different in this respect. The whole Python interpreter and all the required wrapping code is compiled into a single plugin.

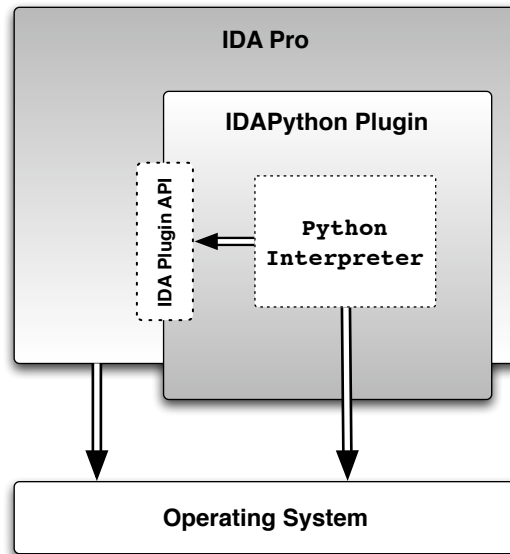


Figure 15: Embedding Python into IDA Pro

As depicted in figure 15, the user's Python scripts run inside the IDA Pro process. The scripts running in the interpreter have full access to all the modules provided by the Python environment, as well as to an extensive list of IDA Pro plugin API functions.

## 5.5 IDAPython Modules

IDAPython has a layered design as shown in figure 16. The modules are separated by the level of abstraction they provide.

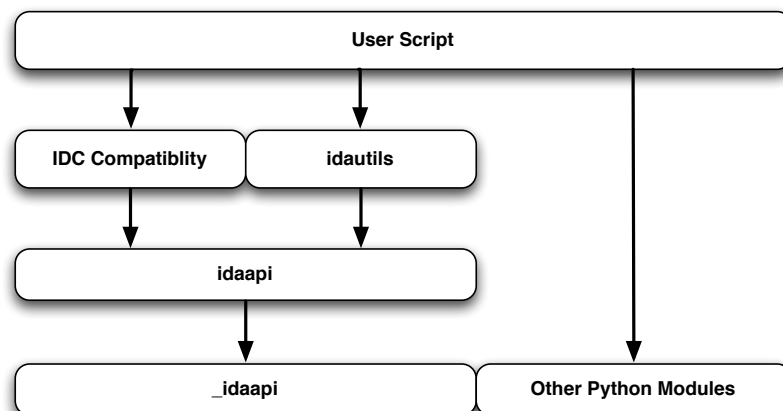


Figure 16: IDAPython modules

The lowest layer is `_idaapi`, which is a direct wrapper over the IDA Pro C++ API. This layer directly mirrors the C++ API, with no convenience features at all. Because of this `_idaapi` should not be directly used. This module is implemented by the plugin binary. There is not external `.py` file to import, and it will be available as soon as the plugin has been loaded successfully.

On the top of `_idaapi` resides another thin layer, called `idaapi`. This module is automatically generated by SWIG, and while it exposes most of the C++ API directly, it also adds certain convenience features, for example simple Python class wrappers around C++ structures and classes.

IDA Pro's own scripting language, IDC has been used by many advanced IDA Pro users for years. To make the transition to IDAPython easier, an IDC compatibility layer was implemented. IDC contains more than 450 functions. Most of them translate directly to low-level plugin API functions, with names and parameters which are easier to remember.

The main requirement for the IDC layer is to follow the names, parameters and return values of the original IDC functions as closely as it is allowed by Python. Since Python is a rich programming environment, there is certain overlap of functionality. In all the cases where an equivalent function is already present in Python, the IDC counterparts have been deprecated. To ease the porting of existing IDC code, all deprecated functions throw a `DeprecatedIDCError` exception, with an informative message on what to use instead. Figure 17 shows an example of a deprecated IDC function.

```
def xtol(str):
    raise DeprecatedIDCError, \
        "xtol() is deprecated. Use python long() instead."
```

Figure 17: Example of a deprecated function

Though `idc` provides an extensive API, it is still a rather low-level, strongly procedural, C-like interface that lacks certain convenience features for commonly performed tasks. To further ease the everyday work of the users, `idautils` was introduced. The `idautils` module contains a list of utility functions, which make certain operations more convenient and provide a high-level 'pythonic' interface.

```

# Get current ea
ea = get_screen_ea()
# Get segment class
seg = getseg(ea)
# Loop from segment start to end
func = get_func(seg.startEA)

while func != None and func.startEA < seg.endEA:
    funcea = func.startEA
    print "Function %s at 0x%x" % (GetFunctionName(funcea), funcea)
    ref = get_first_cref_to(funcea)
    while ref != BADADDR:
        print "    called from %s(0x%x)" % (get_func_name(ref), ref)
        ref = get_next_cref_to(funcea, ref)
    func = get_next_func(funcea)

```

Figure 18: Simple IDAPython program using idaapi

Figure 18 shows a simple IDAPython script that lists code references to all functions in the current section.

```

# Get current ea
ea = ScreenEA()
# Loop from start to end in the current segment
for funcea in Functions(SegStart(ea), SegEnd(ea)):
    print "Function %s at 0x%x" % (GetFunctionName(funcea), funcea)
    # Find all code references to funcea
    for ref in CodeRefsTo(funcea, 1):
        print "    called from %s(0x%x)" % (GetFunctionName(ref), ref)

```

Figure 19: Simple IDAPython program using idutils

In figure 19 the same script is implemented using `idc` and `idutils` functions. It is not only more concise but also built from functions that are easier to remember by their name.

## 5.6 Building and Distribution

IDA Pro runs on Windows, Linux and Mac OS X platforms. Typically, the latest two versions of IDA Pro and Python are widely used, which, multiplied by three platforms, gives 12 different configurations of plugin builds. Comfortable building for the whole matrix of versions presents its own challenge. Originally the project was built with a reasonably simple makefile. Unfortunately, creating a makefile that works seamlessly on all three platforms and allows easy selection of the versions of components to build for, proved to be too difficult.

After learning from the experiences with the unix-style makefile and evaluation of different alternatives, the decision was made that the build process would be performed by a custom Python script. The build script should be self-contained, and not dependent on anything not in the default Python installation. Also it should make it easy to build different combinations of the component versions.

The current incarnation of the build system automatically detects which version of Python is used to run it and uses the appropriate header and library files. Fortunately, the IDA Pro plugin API is developed so that each new version only adds new functionality and does not change the existing set. This makes it possible to build binaries which are forward-compatible with future versions of the application. The stable releases of IDAPython are always built against the version of IDA Pro previous to the last. This ensures that the binary plugin is compatible with at least the last two versions at any given time.

Each stable release includes binary versions for Windows built against the latest two versions of Python, as well as a source code archive. Because of the difficulties associated with building binaries that work across different Linux distributions and Mac OS X versions, users of those platforms will have to build the plugin themselves. Most Linux distributions and Mac OS X come with the compiler installed by default, which makes the plugin easy to build.

The original API headers need modifications in order to be used with SWIG. As the copyright of the IDA Pro Plugin belongs to the vendor, the required SDK modifications are distributed in a patch form. Only the patch is included in the IDAPython source archive and not a full copy of the SDK headers.

## 5.7 Documentation

For any product, software or hardware, documentation is among the most important items to include. For software with the complexity of IDA Pro, and consequently IDAPython, proper documentation is essential. However, producing and maintaining the amount of documentation required for such software is no small task. Documentation must be automated to the largest possible extent.

Since the majority of the documentation needed for IDAPython belongs to the API, that must be tackled first. Python provides a facility, called docstrings, that allows the inclusion of documentation to the source code as comments, in a specific format. Building on the top of docstrings, epydoc is a tool that provides an easy way of building well formatted, fully cross-referenced documentation for Python modules, straight from the source code [23].

To accommodate the needs of epydoc, the original documentation for the `idc` module, provided by the makers of IDA Pro, has been converted into epydoc's custom format. Once the conversion was complete, the reference manual for `idc` can be produced by a simple script.

While the conversion of `idc` documentation was a relatively straightforward task, the same, unfortunately, is not true for `idaapi`. SWIG does provide rudimentary automatic documentation features that cover only function names, argument names and their types, with no information about their purpose or usage. SWIG makes it possible to attach a docstring to any wrapped symbols. However that would mean major modifications to the SDK headers, which should be avoided for maintenance reasons. For the time being, users relying on `idaapi` have to use the documentation comments included in the SDK headers.

Each time IDAPython is released, the documentation will be rebuilt and distributed along with the binary and source archives. Since the documentation weighs more than the rest of the files together, it is distributed in a separate archive file to save bandwidth.

## 6 Results

### 6.1 Implementation Results

IDAPython was first released to the public in 2004. Since then it has been in steady, although sometimes slow, development. The current version of the code provides feature parity with IDC, all IDC operations that apply in Python are supported.

The IDC support is built upon the low-level plugin API which extends beyond what IDC can do. IDAPython wraps a fair amount of functionality from the plugin API, which makes it possible to write code for problems beyond the capabilities of IDC. Table 3 compares the current functionality of IDAPython to the other methods of extending IDA Pro: IDC and binary plugins.

Table 3: Comparison of IDAPython 0.9.55 to other IDA Pro 5.1 extension options

	IDC	Binary Plugins	IDAPython
Development model	Interactive and write-debug	Write-compile-link-debug	Interactive and write-debug
User error tolerance	good: only error message	limited: likely crash on user error	good: only error message
Low-level functions	-	~2000	~1050
High-level functions	466	-	475
Operating system access	limited	extensive	extensive
Available libraries	few	many	many

IDAPython combines the advantage of the high-level coverage, interactivity and error tolerance of IDC, with access to low-level database functionality. In relation to binary plugins, IDAPython offers extensive access to the operating system and an extensive list of available third-party software packages.

The codebase of the plugin is fairly stable and relatively bug-free. Advertised functionality works well. However with something this complex, more testing would be beneficial.

## 6.2 User Experience

Since one of the main objectives of IDAPython was to ease the work of reverse engineers, evaluation from that point of view is also important.

The first aspect of user experience is the ease of installation. IDAPython can be installed by copying the content of the distribution archive to the IDA Pro directory. The only prerequisite is a working Python 2.4 or 2.5 installation on the computer.

User scripts can be run either from a file or directly typed into a dialog in IDA Pro. The former is useful for developing larger scripts that need to be saved for later use. Scripts that are executed once in the work session are kept in a list, called ScriptBox, where they can be quickly recalled later. By using ScriptBox the user does not have to browse to the location of the script file for each run.

The direct entry dialog allows quick experimentation with simple scripts by allowing to see the result of the run immediately. The contents of the script dialog are saved in the disassembly database, so they will not be lost between IDA Pro sessions.

IDAPython has support for simple user-init scripts where the user can place their customisations that will be taken into use each time the plugin is loaded.

## 6.3 Further Development Directions

Now that IDAPython reaches, in some cases even surpasses, the capabilities of IDC, further ideas can be explored. First of all, wrapping of the low-level plugin API is incomplete: about 60-70% percent of the functionality is wrapped properly. The next target is to wrap everything that is possible and useful for plugin development.

One aspect of the Python integration is that, at the moment, Python can be used mainly for automation tasks, replacing IDC. In the future support could be added for the development of processor and loader plugins in Python as well. Much of the required functionality is already wrapped, the proper skeleton plugins need to be developed and tested.

While it is already possible to prototype IDAPython scripts inside IDA Pro, the capabilities are somewhat limited. Another useful addition could be a fully interactive Python console, running inside IDA Pro. Such an interface could make interactive prototyping even easier, though advanced features, such as built-in documentation and symbol completion.

Software with the complexity of IDA Pro and IDAPython greatly benefits from testing. At the moment only limited smoke testing is performed before each IDAPython release. Ideally, there would be a collection of module tests to verify all the functionality. Developing module tests for such an amount of functionality would be very time consuming. However it should be possible with the help of the user community.

#### **6.4 Availability**

IDAPython is freely available from the project website, under a BSD-style license. The distribution consists of pre-built binaries for Windows platform and source code release for Linux and Mac OS X. The development of the plugin is coordinated through a Google Code project, where the latest test builds are released. The IDAPython discussion group is hosted on Google Groups.

**Project homepage:** <http://d-dome.net/idapython/>

**Development page:** <http://code.google.com/p/idapython/>

**Discussion group:** <http://groups.google.com/group/idapython/>

Any IDA Pro user is welcome to download the software, try it and provide much-valued feedback. All questions and comments are welcome in the discussion group. Bug reports can be submitted through the issue tracker on the development page.

## 7 Conclusion

The IDAPython project set out to try and prove that integrating the Python programming language into a complex application, such as Interactive Disassembler Pro was possible. Almost four years later, it can be concluded that it is indeed possible, and practical, even with the rather limited time of one developer.

The main success factors in IDAPython were the choice of a mature but easy to use programming language and the use of an advanced wrapping tool. Python and SWIG turned out to be excellent tools to work with. These tools made it possible to interface a large amount of complex functionality to a scripting language, with relative ease, in a reasonable amount of time.

IDAPython was first introduced to the public in 2004 [1]. Since then the project gathered considerable interest and numerous references in different publications [24; 25; 26].

Over the years, the feedback from the users has been positive. While it is difficult to estimate, as the security research field is often quite secretive, IDAPython is apparently well known, and is actively used by many researchers around the world. While it gives no proper estimation on the size of the user base, the stable 0.9.0 release of IDAPython was downloaded over three thousand times within a year. The test releases of the development tree see a few hundred downloads each.

IDAPython tried to fill a need, to make it easier to automate some of the many daily tasks of security researchers by extending their most commonly used tool with an easy, fully featured programming language. According to their responses from the field, it did just that.

## References

- 1 Carrera E, Erdélyi G. Digital genome mapping - advanced binary malware analysis. Proceedings of the 14th Virus Bulletin Conference. Abingdon, United Kingdom: Virus Bulletin; 2004.  
URL: [http://d-dome.net/papers/Digital\\_Genome\\_Mapping.pdf](http://d-dome.net/papers/Digital_Genome_Mapping.pdf). Accessed 29 March 2008.
- 2 Stallman R. GNU Emacs [online]. Boston, MA: Free Software Foundation; 2008.  
URL: <http://www.gnu.org/software/emacs/>. Accessed 29 March 2008.
- 3 Stallman R. GNU Emacs manual. 16th ed. Boston, MA: Free Software Foundation; 2007.  
URL: <http://www.gnu.org/software/emacs/manual/emacs.pdf>. Accessed 29 March 2008.
- 4 Chassell RJ. An introduction to programming in Emacs Lisp. revised 2nd ed. Boston, MA: Free Software Foundation; 2004.  
URL: <http://www.gnu.org/software/emacs/emacs-lisp-intro/>. Accessed 29 March 2008.
- 5 Roosendaal T. blender.org [online]. Amsterdam, Netherlands: Stichting Blender Foundation; 2008.  
URL: <http://www.blender.org/>. Accessed 29 March 2008.
- 6 Vandervenne P. Executive summary: IDA Pro - at the cornerstone of IT security [online]. Liège, Belgium: DataRescue; 2007.  
URL: <http://www.hex-rays.com/idapro/ida-executive.pdf>. Accessed 29 March 2008.
- 7 Guilfanov I. Fast library identification and recognition technology [online]. Liège, Belgium: DataRescue; 1997.  
URL: <http://www.hex-rays.com/idapro/flirt.htm>. Accessed 29 March 2008.
- 8 Elser D. The IDA Palace [online].  
URL: <http://old.idapalace.net/>. Accessed 29 March 2008.
- 9 Cozens S, Wainwright P. Beginning Perl. West Sussex, England: Wrox Press Inc; 2000.  
URL: <http://www.perl.org/books/beginning-perl/>. Accessed 29 March 2008.

- 10 Hietaniemi J. Comprehensive Perl Archive Network [online]. Helsinki, Finland: CPAN; 6 January 2008.  
URL: <http://www.cpan.org/>. Accessed 29 March 2008.
- 11 van Rossum G, Drake FL. Python tutorial [online]. Hampton, NH: Python Software Foundation; 2006.  
URL: <http://docs.python.org/tut/tut.html>. Accessed 29 March 2008.
- 12 Hugunin J, Warsaw B. The Jython project [online].  
URL: <http://www.jython.org/Project/index.html>. Accessed 29 March 2008.
- 13 Hugunin J. IronPython [online]. Redmond, WA: Microsoft Corporation; 2008.  
URL: [http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython%](http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython%20). Accessed 29 March 2008.
- 14 Rigo A. The ultimate Psyco guide [online]. Aigle, Switzerland: Armin Rigo; 2007.  
URL: <http://psyco.sourceforge.net/psycoguide/index.html>. Accessed 29 March 2008.
- 15 Ierusalimsky R, de Figueiredo LH, Celes W. Lua 5.1 reference manual. Rio de Janeiro, Brazil: Lua.org; 2006.  
URL: <http://www.lua.org/manual/5.1/>. Accessed 29 March 2008.
- 16 Nelson R. The BSD license [online]. Potsdam, NY: Open Source Initiative; 2006.  
URL: <http://opensource.org/licenses/bsd-license.php>. Accessed 29 March 2008.
- 17 Nelson R. The MIT license [online]. Potsdam, NY: Open Source Initiative; 2006.  
URL: <http://www.opensource.org/licenses/mit-license.html>. Accessed 29 March 2008.
- 18 van Rossum G. Extending and embedding the Python interpreter [online]. Hampton, NH: Python Software Foundation; 2006.  
URL: <http://docs.python.org/ext/ext.html>. Accessed 29 March 2008.
- 19 Project S. Simplified wrapper and interface generator [online].  
URL: <http://www.swig.org/>. Accessed 29 March 2008.
- 20 de Guzman J, Abrahams D. Boost.Python tutorial introduction [online]. Onancock, VA: Boost Project; 2003.  
URL: <http://www.boost.org/libs/python/doc/>. Accessed 29 March 2008.
- 21 Limited RC. SIP - overview [online]. Wimborne, United Kingdom: Riverbank Computing Limited; 2007.  
URL: <http://www.riverbankcomputing.co.uk/sip/>. Accessed 29 March 2008.

- 22 Project S. SWIG users manual [online]. Chicago, IL: Swig Project; 2007.  
URL: <http://www.swig.org/Doc1.3/index.html>. Accessed 29 March 2008.
- 23 Loper E. Epydoc: API documentation extraction in Python [online].  
URL: <http://epydoc.sourceforge.net/pycon-epydoc.ps>. Accessed 13 January 2008.
- 24 Carrera E. Introduction to IDAPython [online]. Bochum, Germany: dkbza.org; 2007.  
URL: <http://dkbza.org/data/Introduction%20to%20IDAPython.pdf>. Accessed 29 March 2008.
- 25 Sabanal PV, Yason MV. Reversing C++ [online]. Atlanta, GA: IBM Internet Security Systems X-Force; 2007.  
URL: [http://www.blackhat.com/presentations/bh-dc-07/Sabanal\\_Yason/%Paper/bh-dc-07-Sabanal\\_Yason-WP.pdf](http://www.blackhat.com/presentations/bh-dc-07/Sabanal_Yason/%Paper/bh-dc-07-Sabanal_Yason-WP.pdf). Accessed 29 March 2008.
- 26 Whitehouse O. Analysis of GS protections in Microsoft® Windows Vista™ [online]. Cupertino, CA: Symantec Corporation; 2007.  
URL: [http://www.symantec.com/avcenter/reference/GS\\_Protections\\_in\\_%Vista.pdf](http://www.symantec.com/avcenter/reference/GS_Protections_in_%Vista.pdf). Accessed 29 March 2008.