

## Using FINCAD Developer with scripting languages

### Overview

Scripting languages, such as Python and Perl are becoming popular for automating calculations, running regression tests and also for enhancing build processes, especially on UNIX and LINUX platforms. For example, some systems use a scripting language to build an interest rate curve on a daily basis to use it for valuations or for regression testing. It can be very useful to be able to call FINCAD Developer functions from the scripting language, so that you can use FINCAD analytics for building the curves, for enhancing market data or for performing other calculations that provide analytical results which support your daily activities. This article will be of interest to FINCAD Developer users who already use a scripting language for this purpose or who are considering doing this in the future.

The article describes how to use a generic tool called SWIG to create an interface by processing the information in an interface definition file which references a slightly modified version of the FINCAD fc.h header file.

### SWIG tool

The Simplified Wrapper and Interface Generator (SWIG) is a software development tool that can be used to connect libraries written in C or C++ to a variety of languages including both high-level languages and scripting languages. For the purpose of simplicity, this article will focus on calling the FINCAD Developer library from Python using an interface built using SWIG. If you are interested in calling FINCAD functions from another language that is compatible with SWIG, such as C#, Perl or PHP then you may also find some of the examples in this article helpful since the process for building the interface will be similar in some ways.

### Requirements

The article assumes that you are using FINCAD Developer v10 which includes some additional typedef statements which make it easier for SWIG to process the fc.h header file. The additional statements from the fc.h file are listed below:

```
typedef LPXLOPER Double;  
typedef LPXLOPER DoubleMatrix;  
typedef LPXLOPER DoubleVector;  
typedef LPXLOPER OptionalDouble;  
typedef LPXLOPER SwitchArray;
```

The typedef statements let you know whether the expected value being passed or returned using the XLOPER data structure is a single value, a matrix, a vector, a variant or a vector of switches (i.e. input statistic list). Effectively, in other environments these may be implemented using different data types rather than using a generic data structure such as the XLOPER and the additional information allows for the selection of the correct data type in the target language.

### Modify fc.h so it can be used by SWIG

You should make a copy of fc.h and put it into a working directory to allow it to be customized without impacting other projects which may use the file. You will need to add compiler directives to the header file, which allow sections of the header file to be ignored by SWIG. The sections affected are not needed for generating the Python interface using SWIG.

With your copy of fc.h add the statement `#if !defined(SWIG)` immediately before the statements which define the internal names for FINCAD functions or in other words, immediately prior to the line below in fc.h:

```
#define aaErrorHandlingEnable      aaErrorHandlingEnable_dll
```

The corresponding `#endif` statement should be added prior to the list of function prototypes in the next section of the header file. The end of the header file contains a structure called `name_table`, its definition and initialization should also be ignored by preceding it with `#if !defined(SWIG)` and then by including `#endif` at the end of the structure definition after the structure has been initialized ( i.e. after `};` ). This is necessary because the SWIG tool is not able to process the text from these sections of the header file.

**Simple example**

Now the modified header file is ready for use later by SWIG when it runs against our interface definition file. Next, we will look at a simple example of how to use SWIG to create an interface and then later show how to do the equivalent for all FINCAD functions.

Before, we get started on describing how to create an interface definition file and use SWIG to create the python interface to FINCAD Developer, let us look at a simpler example. For example, if we wanted to create an interface for the standard c library routine `pow(x,y)` in Python, we would create an interface file that looks like the one below:

```
// example.i : A simple SWIG interface definition file
%module example
%{
#include <math.h>
}%

// ANSI C/C++ prototypes
double pow(double x, double y);
```

Instead of writing wrapper code, with SWIG you only need to include the header file(s) which define the functions and the function prototypes for the ones that you wish to use from Python in an interface definition file and also add data type mappings if you are working with more complex data types, such as structures. The SWIG tool then uses this interface definition file to create a Python wrapper for you.

The `%module` line provides the name that will be used when the extension module (i.e. function wrappers) is imported into Python. SWIG will produce a file called `example.py` when run against the sample interface definition file above and also a source file for the wrappers called `example.cpp`. The `%{...%}` section is used to include references that need to be added to the wrapper code that is output by SWIG. You will need to compile the wrappers into a DLL called `_example.dll` before they can be used in Python.

You can create `example.py` and `example.cpp` using the command below:

```
swig.exe -c++ -python -o example.cpp example.i
```

**Creating the wrapper library using SWIG**

Now let us look at a more complex example which is based on the `fc.h` header file (The c interface) in FINCAD Developer. The interface definition file is called `fincad_python.i` and the module name listed in the file is 'fincad', so running SWIG against this file will produce the following output files, `fincad.py` and `fincad_python.cpp` assuming that it is done using the command below:

```
swig.exe -c++ -python -o fincad_python.cpp fincad_python.i
```

`fincad.py` contains symbolic information that your Python script will need to have so that it knows how to call the wrapper functions in `_fincad.dll`. It includes the names for the wrapper functions included in the `_fincad.dll` wrapper library as well as the Python definitions for some classes needed for data type conversion and class initialization. It also includes an import statement that causes additional information to be loaded from `_fincad.dll` at runtime. This file is generated by SWIG and you change the interface definition file and rerun SWIG to correct any problems with this file.

`fincad_python.cpp` is the source code that is used to create `_fincad.dll`. You do not need to use this file directly (only compile it) but you may choose to look at it to understand how the generated interface works. An excerpt from the file is included below to show you what some of the generated wrapper code looks like.

`_fincad.dll` is the compiled wrapper code which allows you to call the FINCAD functions from Python. You compile `fincad_python.cpp` to create `_fincad.dll`. You will need to create a Visual Studio project for doing this and information about how to do that will be provided later in this article. The underscore preceding the DLL name relates to a naming convention used by SWIG. The generated Python file `fincad.py` imports this as a Python module using the statement: "import fincad.py".

### Let's take a quick look at the SWIG generated wrappers

Here is an excerpt from the `fincad_python.cpp` file for the function `aaAccrual_days`. The code is generated by the `SWIG.exe` tool.

```

SWIGINTERN PyObject *_wrap_aaAccrual_days(PyObject *SWIGUNUSEDPARM(self), PyObject *args) {
    PyObject *resultobj = 0;
    double arg1 ;
    double arg2 ;
    int arg3 ;
    Double result;
    double val1 ;
    int ecode1 = 0 ;
    double val2 ;
    int ecode2 = 0 ;
    int val3 ;
    int ecode3 = 0 ;
    PyObject * obj0 = 0 ;
    PyObject * obj1 = 0 ;
    PyObject * obj2 = 0 ;

    if (!PyArg_ParseTuple(args,(char *)"OOO:aaAccrual_days",&obj0,&obj1,&obj2)) SWIG_fail;
    ecode1 = SWIG_AsVal_double(obj0, &val1);
    if (!SWIG_IsOK(ecode1)) {
        SWIG_exception_fail(SWIG_ArgError(ecode1), "in method ""aaAccrual_days"" , argument ""1"" of type ""
double""");
    }
    arg1 = static_cast< double >(val1);
    ecode2 = SWIG_AsVal_double(obj1, &val2);
    if (!SWIG_IsOK(ecode2)) {
        SWIG_exception_fail(SWIG_ArgError(ecode2), "in method ""aaAccrual_days"" , argument ""2"" of type ""
double""");
    }
    arg2 = static_cast< double >(val2);
    ecode3 = SWIG_AsVal_int(obj2, &val3);
    if (!SWIG_IsOK(ecode3)) {
        SWIG_exception_fail(SWIG_ArgError(ecode3), "in method ""aaAccrual_days"" , argument ""3"" of type ""
int""");
    }
    arg3 = static_cast< int >(val3);
    result = (Double)aaAccrual_days(arg1,arg2,arg3);
    {
        CHECK_ERROR(result,STRINGIFY(aaAccrual_days));
        resultobj = Double_to_PyObject(result);
    }
    return resultobj;
fail:
    return NULL;
}

```

This article will not go into detail about the generated wrapper code above since this is not necessary for running FINCAD functions in Python, but will provide a high-level overview of what it is doing so that you better understand the code. Python has an argument list object that it uses for passing all inputs when calling a c function. The `PyArg_ParseTuple` function is used to break up the list into the individual arguments that are needed for the function call. Routines, such as `SWIG_AsVal_double` are used to convert the Python data types into the ones that are understood by SWIG. The SWIG data types are then typecast to the appropriate type needed for the c function call. The `Double_to_PyObject` is used to convert the result of the function call to an appropriate format for returning the data to Python.

**Overview of the interface definition file**

Now, let us look at the `fincad_python.i` interface definition file in detail. We will first go through a high level overview of what the data type conversion functions do and then look at the entire file. Most of the complexity in this file is related to converting arrays from Python into the XLOPER data structure. This is where the typedef statements mentioned earlier come into play. Since the XLOPER data structure can hold different types of data, this information is used to allow for the correct data type conversion. There is also a routine called `aaGlobalFreeAllA()` which is needed for memory management in the c interface. We need to ensure that the SWIG wrappers call this function after converting the result to a Python object.

The data type conversion utility functions included in the interface definition file are:

`Double_to_PyObject` ->Convert a FINCAD Double to a Python double  
`DoubleMatrix_to_PyObject` ->Convert a FINCAD DoubleMatrix to a Python array of arrays  
`DoubleVector_to_PyObject` ->Convert a FINCAD DoubleVector to a Python array of doubles

The XLOPER array inputs used in the c interface needed to be created and destroyed. The functions below support this:

`new_xloper_array` ->Allocate a new xloper array with a default value  
`free_xloper_array` ->Free an array input when its no longer needed

A macro is needed for checking the result and throwing an exception in the case where the result is an error. The macro is required as a means of creating compatibility with the way that SWIG handles exceptions. The code below implements this `CHECK_ERROR` routine.

```
// Implementation detail: Check the result from a FINCAD function (fname) and throw an exception
// if the result was an error. This is target language independent
#define CHECK_ERROR(result,fname) \
if(fincad::XL_IS_ERROR((result)) \
{ \
    std::string buffer("An error occurred calling " fname);\
    buffer.resize(2096); \
    GetErrorString(result->val.err,2096,const_cast<char *>(buffer.c_str())); \
    SWIG_exception(SWIG_RuntimeError,buffer.c_str()); \
}
```

The data type conversion utility functions listed above are used in `typemap` routines. These routines do the basic data type conversions that are needed in the wrapper routines. The list contains `typemap` functions for converting inputs and also those for converting outputs.

`%typemap(out) Double` ->return a Double  
`%typemap(out) DoubleMatrix` ->return a DoubleMatrix  
`%typemap(out) DoubleVector` ->return a DoubleVector  
`%typemap(in) OptionalDouble` ->map to OptionalDouble as input  
`%typemap(in) DoubleMatrix` ->Convert Python list of lists to a DoubleMatrix  
`%typemap(in) DoubleVector` ->Convert Python list of floats to a DoubleVector  
`%typemap(in) SwitchArray` ->Convert Python list of integers to a SwitchArray

There is also a `%typemap(freearg)` used to handle the freeing of input tables for the c interface once the wrapper is finished with them. Information about them is below:

`%typemap(freearg) OptionalDouble` -> frees an optional input table  
`%typemap(freearg) LPXLOPER` ->free any other type of LPXLOPER input

Note: LPXLOPER -> refers to a long pointer to the XLOPER data structure.

**Source for the interface definition file**

Now that you understand every routine in the interface definition file at a high-level, let us now look at the source for the file:

```

-----
%module fincad
%{
// fincad_python.i: An interface definition for SWIG to generate code // for Python (target language)
#include <cassert>
#include <fc.h>

// On certain compilers (vc++ w/ /Zi flag), need to use this to
// properly stringify __LINE__
#define STRINGIFY2(x) #x
#define STRINGIFY(x) STRINGIFY2(x)

// -----
// XLOPER->Python Object conversion functions
// These conversions are language specific
// -----

// Convert a FINCAD Double to a Python double
inline
PyObject * Double_to_PyObject(LPXLOPER value)
{
    assert(fincad::XL_IS_TYPE(value,FCtypeNum));
    return PyFloat_FromDouble(value->val.num);
}

// Convert a FINCAD DoubleMatrix to a Python array of arrays
inline
PyObject * DoubleMatrix_to_PyObject(LPXLOPER value)
{
    assert(fincad::XL_IS_TYPE(value,FCtypeMulti));
    int rows = value->val.array.rows,
        columns = value->val.array.columns;
    PyObject * ret = PyList_New(rows);
    for(int i = 0; i < rows; ++i)
    {
        PyObject * col = PyList_New(columns);
        for(int j = 0; j < columns; ++j)
        {
            PyList_SET_ITEM(col,j,PyFloat_FromDouble(value->val.array.lparray[i*columns+j].val.num));
        }
        PyList_SET_ITEM(ret,i,col);
    }
    return ret;
}

// Convert a FINCAD DoubleVector to a Python array of doubles
// inline
PyObject * DoubleVector_to_PyObject(LPXLOPER value)
{
    assert(fincad::XL_IS_TYPE(value,FCtypeMulti));
    int rows = value->val.array.rows;
    PyObject * ret = PyList_New(rows);
    for(int i = 0; i < rows; ++i)
    {
        PyList_SET_ITEM(ret,i,PyFloat_FromDouble(value->val.array.lparray[i].val.num));
    }
}

```

```

}
return ret;
}

// Implementation detail: Allocate a new xloper array with a default
// value
inline
xloper * new_xloper_array(int rows, int columns, double default_value=0.0)
{
    xloper * ret(new xloper[rows*columns+1]);
    ret->xltype=FCtypeMulti;
    ret->val.array.lpparray = ret+1;
    ret->val.array.rows=rows;
    ret->val.array.columns=columns;
    xloper * ptr(ret+1);
    for(int i = 0 ; i < rows*columns; ++i,++ptr)
    {
        ptr->xltype=FCtypeNum;
        ptr->val.num=default_value;
    }
    return ret;
}

// free temporary input tables used by our wrappers
inline
void free_xloper_array(xloper * thearray)
{
    delete[] thearray;
}

// Implementation detail: Check the result from a FINCAD function
// (fname) and throw an exception
// if the result was an error. This is target language independent
#define CHECK_ERROR(result,fname) \
if(fincad::XL_IS_ERROR((result)) \
{ \
    std::string buffer("An error occurred calling " fname);\
    buffer.resize(2096); \
    GetErrorString(result->val.err,2096,const_cast(buffer.c_str())); \
    SWIG_exception(SWIG_RuntimeError,buffer.c_str()); \
} \

%}

// Tell SWIG to ignore these symbols
//
// Unfortunately, SWIG can't seem to parse the NAME_TAB constant
// in v10 so fc.h needs to be modified to:
//
// #if !defined(SWIG)
// ... NAME_TABLE ...
// };
// #endif
//
// or taken out altogether
%ignore name_table;
%ignore NAME_TAB;
%ignore aaGlobalFreeAllOld;

```

```

#include <exception.i>

// SWIG uses the notion of typemaps to map from the target language
// types to the underlying C-library's type. SWIG doesn't reduce
// typedefs to the underlying type and therefore it is able to
// differentiate between Double, DoubleMatrix, etc (see fc.h) in
// order to provide more useful typemaps
//
// See http://www.swig.org/Doc1.3/Typemaps.html#Typemaps for more
// information

// This typemap will be used for FINCAD functions
// that return a Double
%typemap(out) Double {
    CHECK_ERROR($1,STRINGIFY($name));
    resultobj = Double_to_PyObject($1);
    // free memory used for result of c function call
    aaGlobalFreeAllA();
}

// This typemap will be used for FINCAD functions
// that return a DoubleMatrix
%typemap(out) DoubleMatrix {
    CHECK_ERROR($1,STRINGIFY($name));
    resultobj = DoubleMatrix_to_PyObject($1);
    // free memory used for result of c function call
    aaGlobalFreeAllA();
}

// This typemap will be used for FINCAD functions
// that return a DoubleVector
%typemap(out) DoubleVector {
    CHECK_ERROR($1,STRINGIFY($name));
    resultobj = DoubleVector_to_PyObject($1);
    // free memory used for result of c function call
    aaGlobalFreeAllA();
}

// This typemap will be used for FINCAD functions
// that accept an OptionalDouble as input
%typemap(in) OptionalDouble {
    // Note that here we only allocate a *number*
    $1=new xloper;
    xloper & value=*( $1);
    // if the input is null
    if($input==0)
    {
        // then the input should be 0
        value.val.num = 0.;
    }
    else
    {
        // then convert the input
        value.val.num = PyFloat_AsDouble($input);
    }
    value.xltype=FCtypeNum;
}
    
```

```

// Since we allocate memory for the in typemap
// of OptionalDouble, we must free it here
// Since we allocate using new, we must
// delete it using delete
%typemap(freearg) OptionalDouble {
    delete $1;
}

// Convert Python list of lists to a DoubleMatrix
// Note that this typemap *requires* a list of lists
%typemap(in) DoubleMatrix {
    // ensure that the python object is a list
    if(PyList_Check($input))
    {
        int rows = PyList_Size($input),
            columns = 0;
        if(rows != 0)
        {
            PyObject * item = PyList_GET_ITEM($input,0);
            // and that each element in the list is also a list
            if(!PyList_Check(item))
            {
                SWIG_exception(SWIG_RuntimeError,"Expected list of lists, got list of something else!");
            }
            columns = PyList_Size(item);
        }
        // allocate memory for the xloper array to pass into
        // the FINCAD function
        $1 = new_xloper_array(rows,columns);

        // populate the array
        for(int i = 0; i < rows; ++i)
        {
            PyObject * row = PyList_GET_ITEM($input,i);
            for(int j = 0; j < columns; ++j)
            {
                PyObject * item = PyList_GetItem(row,j);
                $1->val.array.lpparray[i*columns+j].val.num = PyFloat_AsDouble(item);
            }
        }
    }
    else
    {
        // The object is not even a list!
        SWIG_exception(SWIG_RuntimeError,"Expected a list of lists, got something else!");
    }
}

// Convert Python list of floats to a DoubleVector
// Similar to the above typemap but for arrays of floating points only
%typemap(in) DoubleVector
{
    if(PyList_Check($input))
    {
        int rows = PyList_Size($input);
        $1 = new_xloper_array(rows,1);
        for(int i = 0; i < rows; ++i)
        {
    
```

```

PyObject * item = PyList_GetItem($input,i);
$1->val.array.lparray[i].val.num = PyFloat_AsDouble(item);
}
}
else
{
    SWIG_exception(SWIG_RuntimeError,"Expected an array of doubles");
}
}

// Convert Python list of integers to a SwitchArray
// Similar to the above typemap but for integers only
%typemap(in) SwitchArray
{
    if(PyList_Check($input))
    {
        int rows = PyList_Size($input);
        $1 = new_xloper_array(rows,1);
        for(int i = 0; i < rows; ++i)
        {
            PyObject * item = PyList_GetItem($input,i);
            $1->val.array.lparray[i].val.num = double(PyInt_AsLong(item));
        }
    }
    else
    {
        SWIG_exception(SWIG_RuntimeError,"Expected SwitchArray as input");
    }
}

// Since all the other typemaps allocate
// using new_xloper_array, we deallocate
// the rest of them using free_xloper_array
%typemap(freearg) LPXLOPER
{
    free_xloper_array($1);
}

// Now actually include the FINCAD functions to apply the above code!
#include <fc.h>

```

### Setting up to run the tutorial

Since you now understand what is in the interface definition files and how the various data type mappings work, we will look next at how to install SWIG, Python and then run the build process to create the wrapper DLL from the source file.

### Installing SWIG

The SWIG tool can be downloaded from • HYPERLINK "<http://www.swig.org/>" ••<http://www.swig.org/>•. SWIG may ask you to complete a two question survey when you go to the download page so that they know what you are using SWIG for. Simply download the file and unzip it to get access to the SWIG tool.

### Installing Python

Python is distributed under an open source license that allows it to be used for free, even in commercial applications. The latest version can be downloaded from • HYPERLINK "<http://www.python.org/download/>" ••<http://www.python.org/download/>•. Download and unzip the version that you wish to use.

It is recommended that you use Python 2.4 (or an older version) since the latest version of SWIG may not yet have been tested against Python 2.5. If you get a warning which mentions `_CRT_SECURE_NO_DEPRECATED` when compiling the wrappers in `fincad_python.cpp` then you are likely using a version of Python that is not yet supported by the version of SWIG you have and switching to the previous version will resolve the issue.

### Use SWIG to create the source for the function wrappers

Copy the modified version of `fc.h` and `fincad_python.i` to the root directory for your SWIG installation. This will be something like `C:\myInstallPath\swigwin\`

Open the command prompt (Start->All Programs->Accessories->Command Prompt) and change the current directory to the root directory for SWIG. Now use the command below to create the source for the Python wrapper.

```
swig.exe -c++ -python -o fincad_python.cpp fincad_python.i
```

This will create two new files in the directory, `fincad.py` and `fincad_python.cpp`. Now we need to look at how to compile the source code from `fincad_python.cpp` to create the compiled wrapper library, `_fincad.dll`. Create a build directory for compiling the DLL and copy `fincad.py`, `fincad_python.cpp` and `fc.h` to the folder.

### Setting up our project to compile the wrapper library

Your Visual Studio project will need to reference some files from your Python installation, `Python.h` and `pythonxy.lib` (where `xy` is the Python version number). Assuming that you installed Python 2.4 in the folder `C:\Python24` then you will find these files in the locations below:

```
C:\Python24\include\Python.h  
C:\Python24\libs\python24.lib
```

Now let us look at how to build the Visual Studio project which compiles the wrapper.

Open Visual Studio 2005 and choose File->New->Project from the menu and create an empty C++ project in the build directory.

To ensure that your project is able to find `Python.h` and `python24.lib` use the Tools->Options menu items and select the Projects and Solutions list item and then use the VC++ Directories sub-item to add the paths to the include and lib directories for Python.C. Use the "Show directories for:" combobox to add the paths to the Include and Library Files search paths. You should also add a path to your build directory to the Include Files path. In older versions of Visual Studio, there is a Directories tab that provides a similar function.

Now we should also set the correct configuration for the project. Select Project->Properties from the menu and select the General sub-item under Configuration Properties in the list view. The Configuration Type should be changed to Dynamic Library (.dll), also choose the C/C++ item and select the Code Generation sub-item to

change the Runtime Library setting to Multi-Threaded DLL and finally choose the Linker item and then the General sub-item and change the Output File to: \$(OutDir)\\_fincad.dll

Choose Project->Add existing item to add the Python wrapper code that we just generated with SWIG (fincad\_python.cpp) to the project. We also want to link to the fca.lib (import library) when building the DLL. One way to do this is to Select the Project->Properties menu item and under Configuration Properties select the Linker sub-item and then the Command Line item. The additional options box allows you to link additional libraries, such as fca.lib when building the project.

**Building the \_fincad.dll wrapper library**

You can now choose Build->Rebuild Solution to compile a debug version of \_fincad.dll or use the configuration manager to change to building a release version of \_fincad.dll (the Python wrapper).

**Running a Python program that calls a FINCAD function**

Now that we have compiled our Python wrapper, we can look at how to call a simple Python program that calls a FINCAD function. As a starter, it is useful to add the path to the Python folder to the path environmental variable using a statement like the one below:

```
set path=%path%;C:\python24
```

The statement above can be run from the Command Prompt. To get more information about the inputs for the Python interpreter you can use this command:

```
python.exe -h
```

To run an example from the Command Prompt you will need to have several files in your Python application folder

- fincad.py -> Python header generated by SWIG
- \_fincad.dll -> Compiled wrapper interface
- aaOption\_LV\_smile\_p.py -> Python code which calls a FINCAD function

**Setting up the environmental variables**

You should also setup a couple of environmental variables in the Command Prompt window to make running the example easier. Use the statements below:

```
SET PATH=%CD%\;%PATH%
SET PYTHONPATH=%CD%\;%PYTHONPATH%
```

The statements above will cause the current directory to be included in the search path which simplifies running the example.

**Running the Python sample program which calls a FINCAD function**

You should now be able to execute the example Python script using the statement below:

```
Python aaOption_LV_smile_p.py
```

Let's look at the content of our sample file now. Below is the source for the sample Python script which calls the FINCAD function aaOption\_LV\_smile\_p

**Source for the sample file: aaOption\_LV\_smile\_p.py**

```
# load the wrapper interface which provides the ability to call FINCAD functions
import fincad;
```

```
# setup the inputs for our function call
price_u = 50;
payoff_type = 2;

strike_tbl = [[50]];

option_style = 1;
d_exp = 39264;
d_v = 39083;
d_mkt = 39083;
smile_type = 1;

smile_tbl = [
  [39264,30,0.25],
  [39264,40,0.23],
  [39264,50,0.2],
  [39264,60,0.18]];

interp_smile = 3;
extrap_smile = 1;

smoothing_param = [[0.001]];

# This is the simplest format for inputting a discount factor curve
# The function builds the curve internally using a flat rate of 5%
df_crv_std = [[0.05]];
df_crv_hld = [[0.05]];

intrap = 1;
S_steps = 200;
t_steps = 200;

scheme = 1;

# The built-in Python function range() creates a list containing an arithmetic progression
stat = range(1,9);

# Call the FINCAD function
output =
fincad.aaOption_LV_smile_p(price_u,payoff_type,strike_tbl,option_style,d_exp,d_v,d_mkt,smile_type,smile_tbl,interp_smile,extrap_smile,smoothing_param,df_crv_std,df_crv_hld,intrap,S_steps,t_steps,scheme,stat);

# The function call returns a Python vector into the variable output
# Vectors are implemented and indexed as a collection in Python
# The two lines below print the elements in the Vector one by one
for out in output:
  print out;
```

---

## Conclusion

Using FINCAD Developer from a scripting language is helpful when automating repetitive tasks and SWIG is a good reusable tool that makes creating the interface easier.

The skills that you learn the first time that you use SWIG to create an interface may come in handy later when you want to interface to another language or to another library using the same language. This article, describes the process of using SWIG to create the interface, the process for compiling the interface code into a wrapper DLL and it also provides a simple example of calling a FINCAD function using the wrapper library. You may find this useful as you work to automate tasks or extend the functionality in the processes that you have already automated. Good luck with your future projects that use scripting.

**References**

Foetsch, Michael Python Extensions in C++ Using SWIG (on-line article)  
Beazley, David Tcl and SWIG as a C/C++ Development Tool (on-line article)

**Disclaimer**

Your use of the information in this article is at your own risk. The information in this article is provided on an "as is" basis and without any representation, obligation, or warranty from FINCAD of any kind, whether express or implied. We hope that such information will assist you, but it should not be used or relied upon as a substitute for your own independent research.

2007 © FinancialCAD Corporation. All rights reserved. FinancialCAD® and FINCAD® are registered trademarks of FinancialCAD Corporation. Other trademarks are the property of their respective holders. This email is for informational purposes only. FinancialCAD Corporation MAKES NO WARRANTIES, EXPRESSED OR IMPLIED, IN THIS SUMMARY.