DEVELOPING AND EXTENDING SEDEEN WITH PYTHON 2.7

Abstract

The Application Programmer's Interface to Python makes it possible for programmers to access the Python interpreter. There are two different reasons for using the Python/C API. The first and most common reason is to write C/C++ modules that extend the Python interpreter. The second reason is to use Python as a component in a C++ application, which is referred as embedding Python in an application. While Python extension is well understood, Python embedding is not straightforward. In this tutorial, I will describe the principle of the Python embedding into a C++ application.

Azadeh Yazdanpanah

azadehya@sri.utoronto.ca

CONTENTS

PYTHON IN SEDEEN SDK

Nowadays, it is becoming more common that applications provide integrated scripting to their users. Microsoft Office is a good example of this kind. Embedding Python as a scripting language into Sedeen SDK has several benefits. Scripting can speed up the prototyping stage and provides an easy access for both novice and expert users. Users with no knowledge of C++ can also extend the application easily. It also makes it easier to write macros and do batch processing. Furthermore, scripting can be used for automated testing.

This tutorial addresses the following areas:

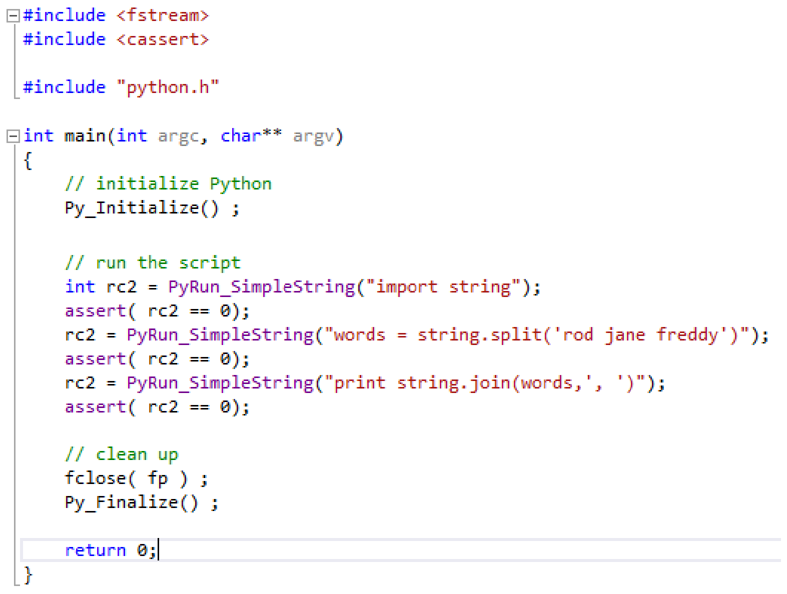
- Principles of Python embedding
- Building embedded applications
- Getting embedded instance information
- Translating between C++ and Python data types
- Exception Handling
- Writing a C++ wrapper library
- PythonQt
- Writing Python Script

PRINCIPLES OF PYTHON EMBEDDING

You can execute Python code from a C++ application in three steps:

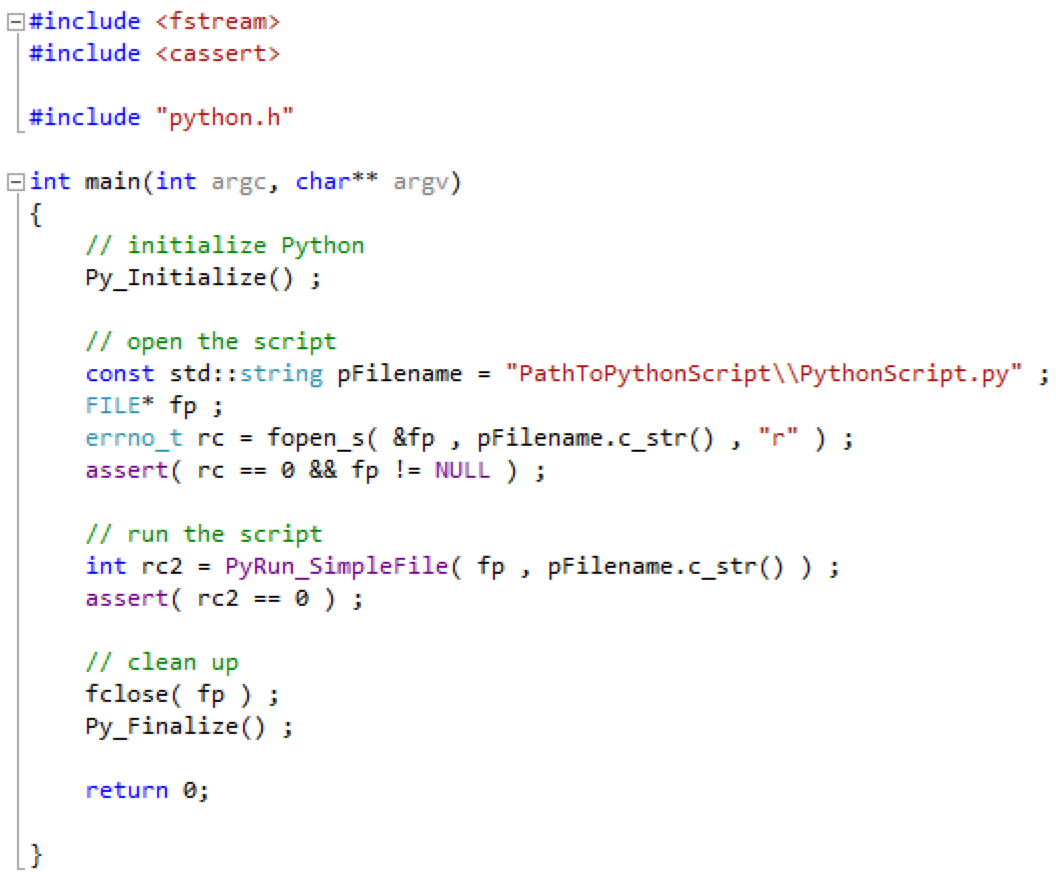
- Initialize an instance of the Python interpreter.
- Execute your Python code.
- Close (finalize) the Python interpreter.

The simplest method of embedding Python code is to use *PyRun_SimpleString()* function. It is similar to typing the individual lines in an interactive Python session. For example, the following code runs a simple Python program to split and reassemble a string:



```
#include <fstream>
#include <cassert>

#include "python.h"

int main(int argc, char** argv)
{
    // initialize Python
    Py_Initialize() ;

    // run the script
    int rc2 = PyRun_SimpleString("import string");
    assert( rc2 == 0);
    rc2 = PyRun_SimpleString("words = string.split('rod jane freddy')");
    assert( rc2 == 0);
    rc2 = PyRun_SimpleString("print string.join(words,', ')");
    assert( rc2 == 0);

    // clean up
    fclose( fp ) ;
    Py_Finalize() ;

    return 0;
}
```

Here is another example using *PyRun_SimpleFile* () function. First, you need to open the python script and then pass the file handle into *PyRun_SimpleFile*, where Python will read and execute the file contents. If you run this program the results would be the same as previous example.



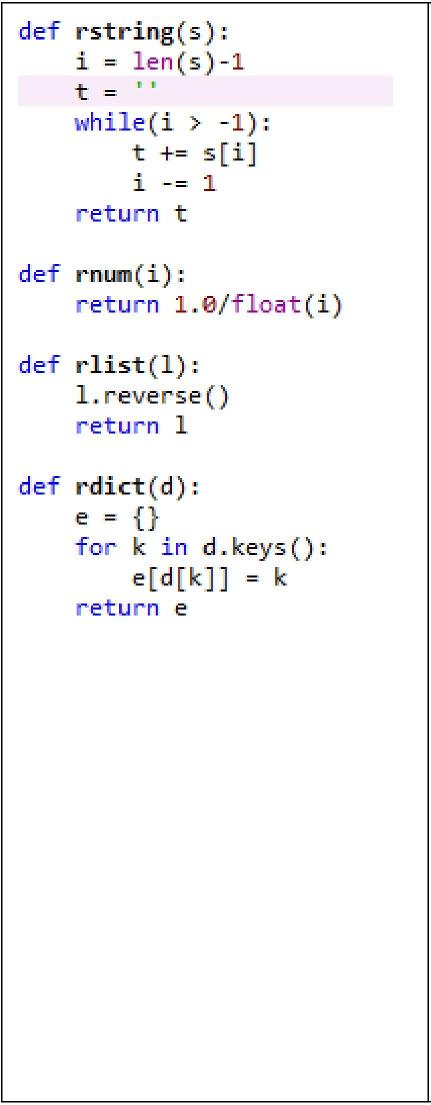
```cpp
#include <fstream>
#include <cassert>

#include "python.h"

int main(int argc, char** argv)
{
    // initialize Python
    Py_Initialize() ;

    // open the script
    const std::string pFilename = "PathToPythonScript\\PythonScript.py" ;
    FILE* fp ;
    errno_t rc = fopen_s( &fp , pFilename.c_str() , "r" ) ;
    assert( rc == 0 && fp != NULL ) ;

    // run the script
    int rc2 = PyRun_SimpleFile( fp , pFilename.c_str() ) ;
    assert( rc2 == 0 ) ;

    // clean up
    fclose( fp ) ;
    Py_Finalize() ;

    return 0;

}
```

Py_Initialize() initializes the table of loaded modules, and creates the fundamental modules builtins, __main__, and sys. It also initializes the module search path (sys.path). Py_Initialize() does not set the “script argument list” (sys.argv). If this variable is needed by Python code that will be executed later, it must be set explicitly with a call to PySys_SetArgvEx(argc, argv, updatepath) after the call to Py_Initialize(). In an application embedding Python, this should be called before using any other Python/C API functions. There is no return value; it is a fatal error if the initialization fails. (python.org)

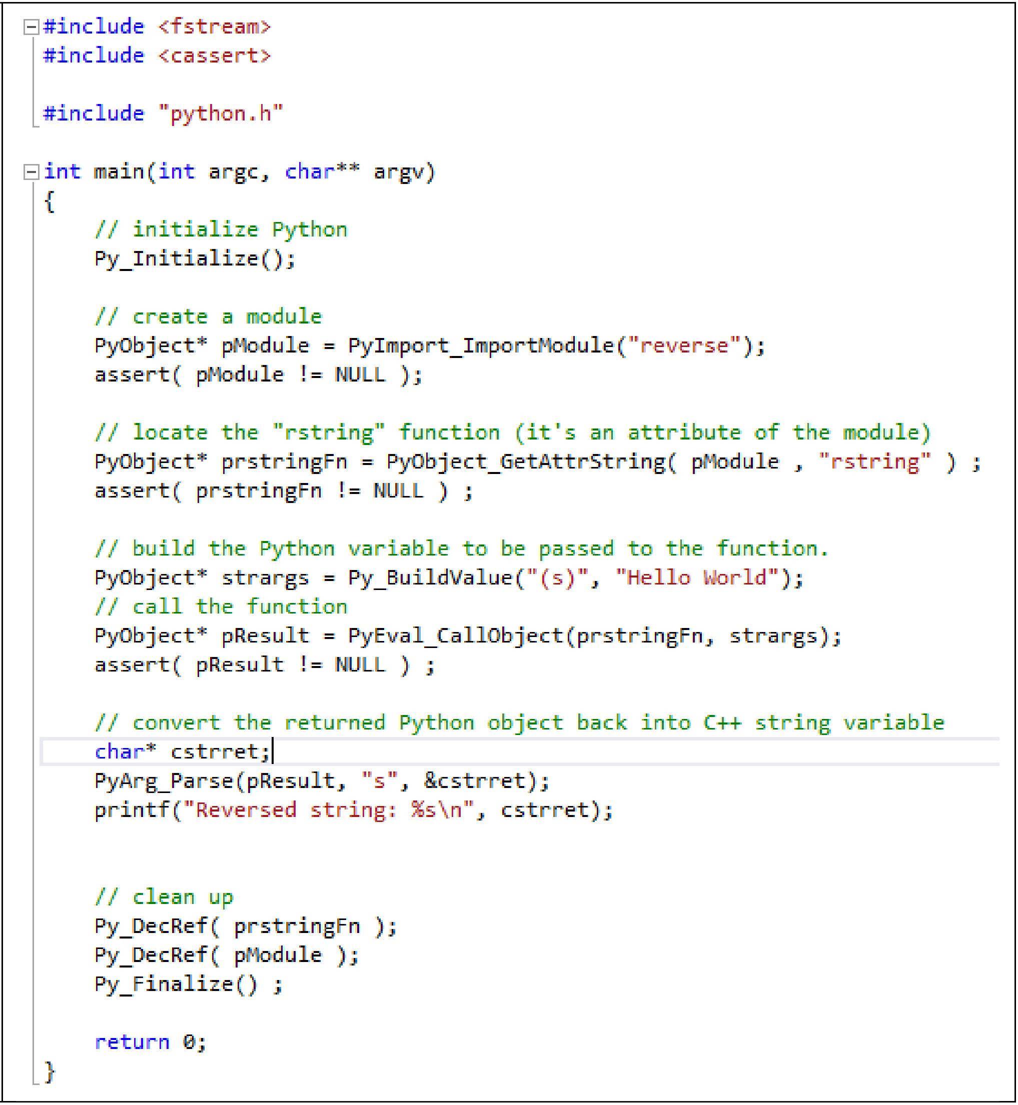
Py_Finalize() undoes all initializations made by *Py_Initialize()* and subsequent use of Python/C API functions, and destroys all sub-interpreters that were created and not yet destroyed since the last call to *Py_Initialize()*. Ideally, this frees all memory allocated by the Python interpreter. (python.org)

Another method of embedding Python code is to call a Python object or to integrate with a Python module. This way, we can pass information to and from the Python interpreter. Let's start with an example that calls a function with some parameters, within a Python module. Here is the source code:

PYTHON SCRIPT



```python
def rstring(s):
    i = len(s)-1
    t = ''
    while(i > -1):
        t += s[i]
        i -= 1
    return t

def rnum(i):
    return 1.0/float(i)

def rlist(l):
    l.reverse()
    return l

def rdict(d):
    e = {}
    for k in d.keys():
        e[d[k]] = k
    return e
```



```cpp
#include <fstream>
#include <cassert>

#include "python.h"

int main(int argc, char** argv)
{
    // initialize Python
    Py_Initialize();

    // create a module
    PyObject* pModule = PyImport_ImportModule("reverse");
    assert( pModule != NULL );

    // locate the "rstring" function (it's an attribute of the module)
    PyObject* prstringFn = PyObject_GetAttrString( pModule , "rstring" ) ;
    assert( prstringFn != NULL ) ;

    // build the Python variable to be passed to the function.
    PyObject* strargs = Py_BuildValue("(s)", "Hello World");
    // call the function
    PyObject* pResult = PyEval_CallObject(prstringFn, strargs);
    assert( pResult != NULL ) ;

    // convert the returned Python object back into C++ string variable
    char* cstrret;
    PyArg_Parse(pResult, "s", &cstrret);
    printf("Reversed string: %s\n", cstrret);


    // clean up
    Py_DecRef( prstringFn );
    Py_DecRef( pModule );
    Py_Finalize() ;

    return 0;
}
```

First, we load the module by using the *PyImport_ImportModule* function. This function accepts a module name and returns a Python object (in this case, a module object). Then, we use *PyObject_GetAttrString* to get the reference to the function. The return object can be used to call the function from within the C++ code. Now, we need to build the argument list; we can call *Py_BuildValue()* to convert a string to a Python Object. For more information about *Py_BuildValue()* function, one can refer to their official website (python.org). The call to the function uses *PyEval_CallObject()*, which accepts a suitable object (the object we extracted through attributes earlier in this section) and the arguments (which we've just built). Finally, the *rstring()* function returns the string in its reversed format. We need to convert it back to a C++ variable. We do this by using *PyArg_Parse()* function.

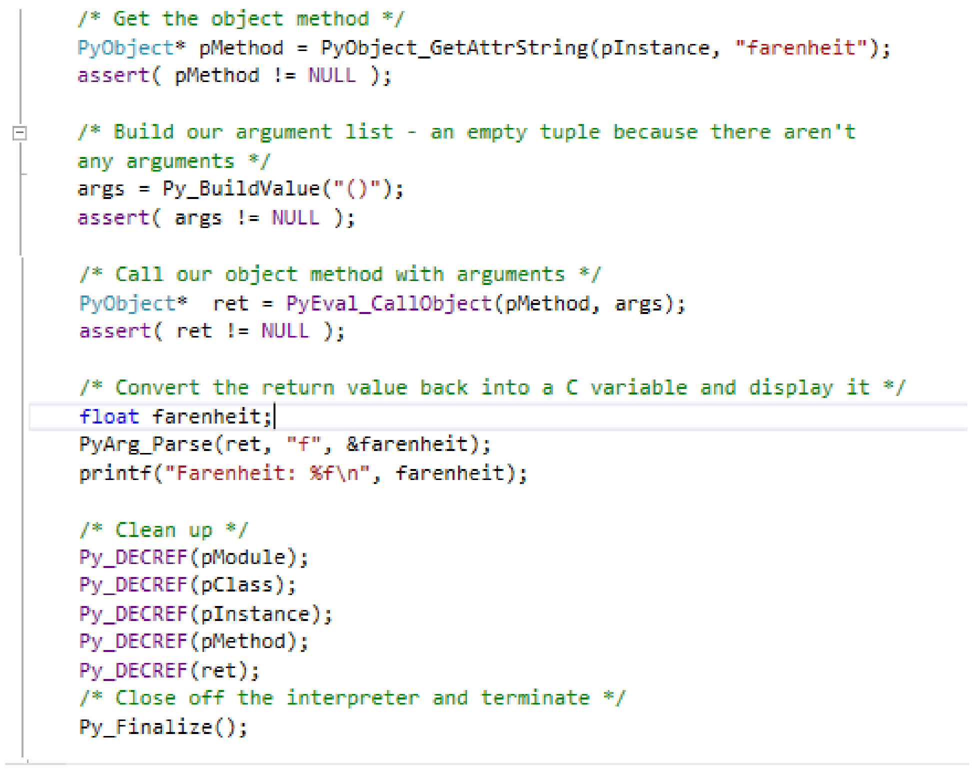
Reference counting is another important aspect of Python embedding. Please refer to the Python documentation at https://docs.python.org/2/c-api/intro.html#objects-types-and-reference-counts for complete documentation on this topic.

The final method for integrating with Python from within C++ is through the class/object system. Because Python is an object-oriented language, most of the interaction with Python will often be with Python classes and objects. To use a class within an embedded application, perform the following steps: load the module, create a suitable object, use *PyObject_GetAttrString* to get the reference to the object or its methods, then execute the method as before. Here is an example that uses a simple module for converting a given Celsius value into Fahrenheit:

PYTHON SCRIPT

```python
class celsius:

    def __init__(self, degrees):
        self.degrees = degrees

    def farenheit(self):
        return ((self.degrees*9.0)/5.0)+32.0
```

```cpp
int main(int argc, char** argv)
{
    // initialize Python
    Py_Initialize();

    /* Load the module */
    PyObject* pModule = PyImport_ImportModule("celsius");
    assert( pModule != NULL );

    /* Find the class */
    PyObject* pClass = PyObject_GetAttrString(pModule, "celsius");
    assert( pClass != NULL );

    /* Build the argument call to our class - these are the arguments
    that will be supplied when the object is created */
    PyObject* args = Py_BuildValue("(f)", 100.0);
    assert( args != NULL );

    /* Create a new instance of our class by calling the class
    with our argument list */
    PyObject* pInstance = PyEval_CallObject(pClass, args);
    assert( pInstance != NULL );

    /* Decrement the argument counter as we'll be using this again */
    Py_DECREF(args);
```



```
/* Get the object method */
PyObject* pMethod = PyObject_GetAttrString(pInstance, "farenheit");
assert( pMethod != NULL );

/* Build our argument list - an empty tuple because there aren't
any arguments */
args = Py_BuildValue("()");
assert( args != NULL );

/* Call our object method with arguments */
PyObject*  ret = PyEval_CallObject(pMethod, args);
assert( ret != NULL );

/* Convert the return value back into a C variable and display it */
float farenheit;
PyArg_Parse(ret, "f", &farenheit);
printf("Farenheit: %f\n", farenheit);

/* Clean up */
Py_DECREF(pModule);
Py_DECREF(pClass);
Py_DECREF(pInstance);
Py_DECREF(pMethod);
Py_DECREF(ret);
/* Close off the interpreter and terminate */
Py_Finalize();
```

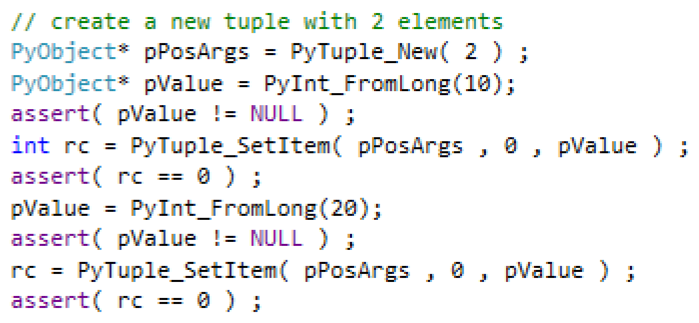
GETTING EMBEDDED INSTANCE INFORMATION

Here is the list of the main functions to get information about the Python interpreter. Using these function, you can get information about the installation, including version numbers, module paths, and compiler, as well as information the Python library used when building the application (i.e. build information).

FUNCTION	DESCRIPTION
Py_GetPrefix()	Returns the prefix for the platform-independent files
char* PyGetExecPrefix()	Returns the execution prefix for the installed Python files
char* Py_GetPath()	Returns the execution prefix for the installed Python files
char *Py_GetProgramFullPath()	Returns the full default path to the Python interpreter
const char* Py_GetVersion()	Returns a string for the current version of the Python library
const char* Py_GetPlatform()	Returns the platform identifier for the current platform
const char* Py_GetCopyright()	Returns the copyright statement
const char* Py_GetCompiler()	Returns the compiler string (name and version of the compiler) used to build the Python library
const char* Py_GetBuilderInfo()	Returns the build information (version and date) of the interpreter

CONVERTING BETWEEN C++ AND PYTHON DATA TYPES

Python data structures are returned from or passed to the Python interpreter in the form of PyObjects. Let's look at some examples of converting C++ data type to Python data type and vice-versa. For instance, let's see how you can pass two integers to Python function. First you need to construct an empty tuple with *PyTuple_New()*, which takes the length of the tuple and returns a PyObject pointer to a new tuple. Then, use *PyTuple_SetItem* to set the values of the tuple items, passing each value as a PyObject pointer. In this example, we also used *PyInt_FromLong()* to convert a C++ integer value to a Python object.



```
// create a new tuple with 2 elements
PyObject* pPosArgs = PyTuple_New( 2 ) ;
PyObject* pValue = PyInt_FromLong(10);
assert( pValue != NULL ) ;
int rc = PyTuple_SetItem( pPosArgs , 0 , pValue ) ;
assert( rc == 0 ) ;
pValue = PyInt_FromLong(20);
assert( pValue != NULL ) ;
rc = PyTuple_SetItem( pPosArgs , 0 , pValue ) ;
assert( rc == 0 ) ;
```

It is also important to be able to extract C++ objects from Python objects. To get a specific type, you need to cast PyObject to the correct type. For instance, if you want to convert a PyObject into a long int, call *PyInt_AsLong()*. *PyInt_AsLong* is a safe function that performs a checked casting to *PyIntObject* before extracting the long int value. If you receive an object from a Python program and you are not sure if it has the right type, you must perform a type check first; for example, to check that an object is a float or a subtype of a float, use *PyFloat_Check()*. If you want to know exact object type, not a subclass, use *PyFloat_CheckExact()*.

See the Python documentation at https://docs.python.org/2/c-api/concrete.html for complete documentation on this topic. Working with Numpy in embedded python is another topic that you can consider. Images can be represented as a Numpy array when they passed between Python script and C++ application.

Almost all Python objects live on the heap: you never declare an automatic or static variable of type *PyObject*, only pointer variables of type *PyObject** can be declared. The only exception are the type objects; since these must never be deallocated, they are typically static *PyTypeObject* objects.

Building embedded applications

To build the application, you need to include Python header file (*"Python.h"*) and link against the Python libraries. "Python.h" exposes the Python/C API into your C++ application. You can link again Python libraries by either changing the CMake file or modifying the project properties in Visual Studio. Also, you need to put a copy of Python ".dll" to the same directory as your plugin ".dll".

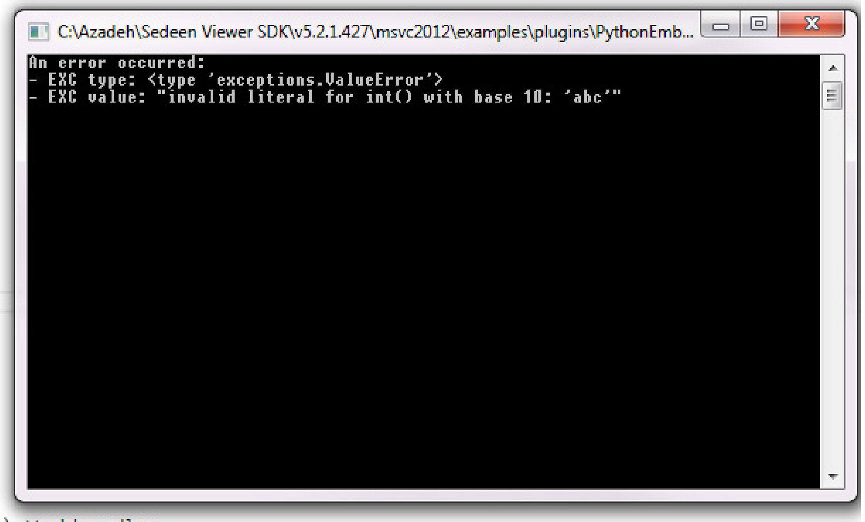
Please note that you need to compile the Python yourself using the same compiler (VS 2012 -32bits), used to build Sedeen plugins. To do so, you can download the source code from python.org website. Unzip the package and in Visual Studio open "pcbuild.sln" solution in the PCBuild directory. Then, you need to build Python in both debug and release version for debug/release version of your plugin. Once the build has finished, Python can be linked into the application.

It would be also a good idea to build Anaconda instead of Python since Python does not sheep with all useful packages but Anaconda does. Instead of attempting to use the whole Anaconda packages, you can use Miniconda, a reduced version of Anaconda, and install a pre-defined set of packages and provide a way to easily install new ones.

EXCEPTION HANDLING

Let's look at how to check for errors when calling Python script from inside the C++ application. Python will return an error value in the event of an error. You can retrieve the details of error by calling *PyErr_Fetch()* function, which returns the normal Python error variables: *sys.exc_type, sys.exc_value and sys.exc_traceback*. If the error indicator is not set, set all three variables to NULL. If it is set, it will be cleared and you own a reference to each object retrieved. The value and traceback object may be NULL even when the type object is not. Here is an example indicating the error handling in embedded Python:

```cpp
#include <fstream>
#include <cassert>
#include <iostream>

#include "python.h"

int main(int argc, char** argv)
{
    // initialize Python
    Py_Initialize();

    // try to parse a bad string as an int
    PyObject* pVal = PyInt_FromString( "abc" , NULL , 10 ) ;
    if ( pVal != NULL )
        assert( false ) ;
    else
    {
        // get the error details
        std::cout << "An error occurred:" << std::endl ;
        PyObject *pExcType , *pExcValue , *pExcTraceback ;
        PyErr_Fetch( &pExcType , &pExcValue , &pExcTraceback ) ;
        if ( pExcType != NULL )
        {
            PyObject* pRepr = PyObject_Repr( pExcType ) ;
            std::cout << "- EXC type: " << PyString_AsString(pRepr) << std::endl ;
            Py_DecRef( pRepr ) ;
            Py_DecRef( pExcType ) ;
        }
        if ( pExcValue != NULL )
        {
            PyObject* pRepr = PyObject_Repr( pExcValue ) ;
            std::cout << "- EXC value: " << PyString_AsString(pRepr) << std::endl ;
            Py_DecRef( pRepr ) ;
            Py_DecRef( pExcValue ) ;
        }
        if ( pExcTraceback != NULL )
        {
            PyObject* pRepr = PyObject_Repr( pExcValue ) ;
            std::cout << "- EXC traceback: " << PyString_AsString(pRepr) << std::endl ;
            Py_DecRef( pRepr ) ;
            Py_DecRef( pExcTraceback ) ;
        }
    }
```



C:\Azadeh\Sedeen Viewer SDK\v5.2.1.427\msvc2012\examples\plugins\PythonEmb...

```
An error occurred:
- EXC type: <type 'exceptions.ValueError'>
- EXC value: "invalid literal for int() with base 10: 'abc'"
```

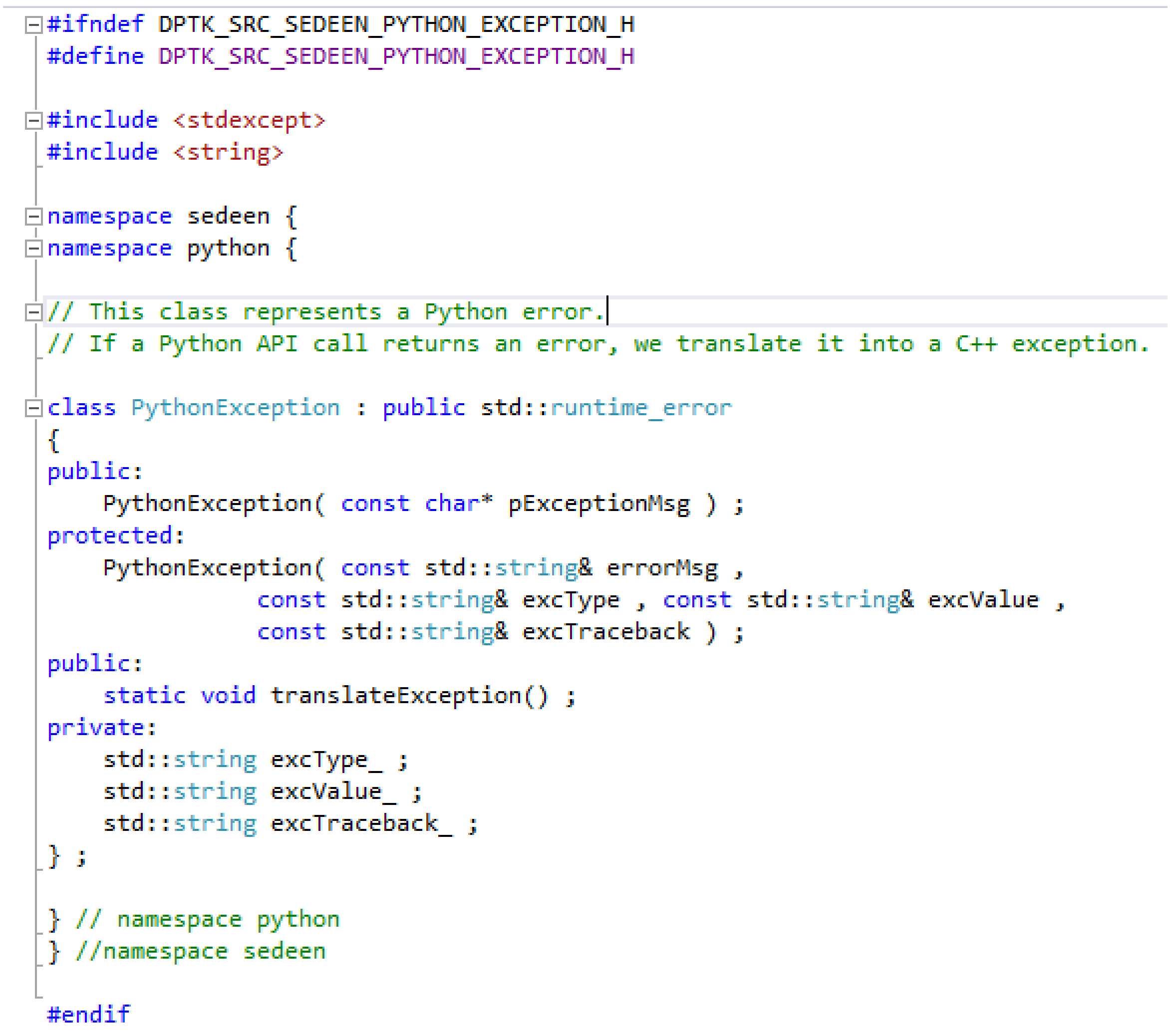
WRITING A C++ WRAPPER LIBRARY

You can also write a C++ wrapper library to make it easier for users to work with Python/C API. The first thing that You need to take care of is initializing Python and cleaning it up. The following code demonstrates one implementation. The C++ application must call *initPython()* to initialize Python and *releasePython()* function to shuts down the interpreter and frees any resources allocated during its lifetime.

```cpp
#pragma once

namespace python {

extern void initPython( bool enableThreads ) ;
extern void releasePython() ;

} // namespace python
```

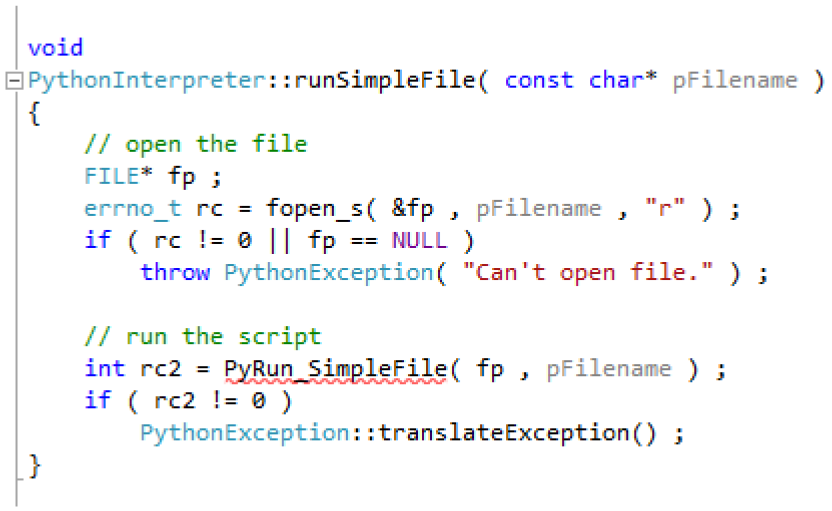
```cpp
namespace python {

static bool gIsInit = false ;

void
initPython( )
{
    assert( !gIsInit ) ;

    // initialize Python
    Py_Initialize() ;
    gIsInit = true ;
}

void
releasePython()
{
    assert( gIsInit ) ;

    // clean up Python
    Py_Finalize() ;
    gIsInit = false ;
}

} // namespace python
```

Now, you need to write an exception class to handle the errors. Whenever a Python/C API call returns an error status, it retrieves the details of the error, then converts it to a C++ exception. Below is an example of exception class:



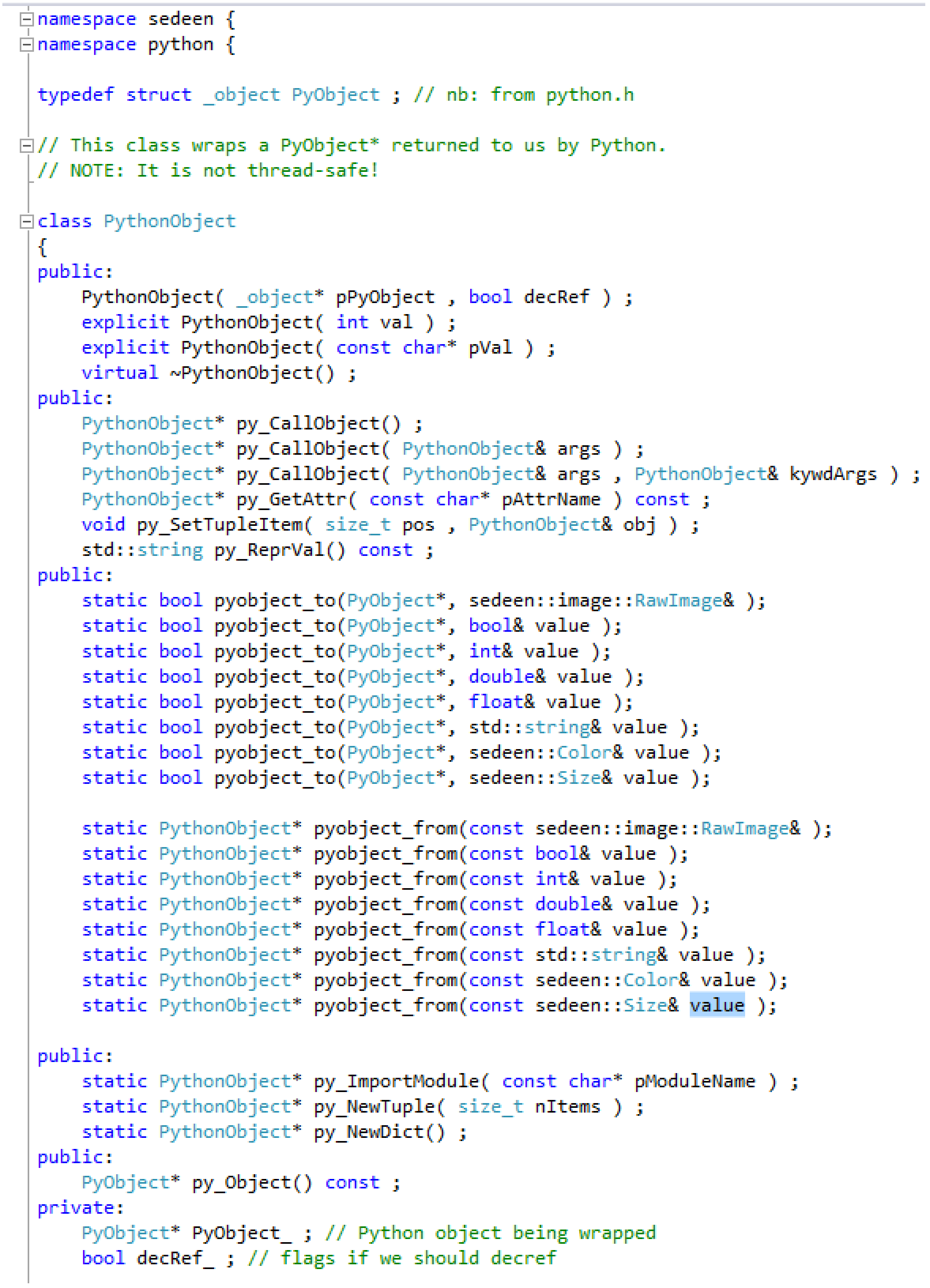
```
#ifndef DPTK_SRC_SEDEEN_PYTHON_EXCEPTION_H
#define DPTK_SRC_SEDEEN_PYTHON_EXCEPTION_H

#include <stdexcept>
#include <string>

namespace sedeen {
namespace python {

// This class represents a Python error.
// If a Python API call returns an error, we translate it into a C++ exception.

class PythonException : public std::runtime_error
{
public:
    PythonException( const char* pExceptionMsg ) ;
protected:
    PythonException( const std::string& errorMsg ,
                     const std::string& excType , const std::string& excValue ,
                     const std::string& excTraceback ) ;
public:
    static void translateException() ;
private:
    std::string excType_ ;
    std::string excValue_ ;
    std::string excTraceback_ ;
} ;

} // namespace python
} //namespace sedeen

#endif
```

Next, you need to write a class to manage the interpreter. For example, you can have a class called "Interpreter" and add a method to the class to run a specified script:



```
void
PythonInterpreter::runSimpleFile( const char* pFilename )
{
    // open the file
    FILE* fp ;
    errno_t rc = fopen_s( &fp , pFilename , "r" ) ;
    if ( rc != 0 || fp == NULL )
        throw PythonException( "Can't open file." ) ;

    // run the script
    int rc2 = PyRun_SimpleFile( fp , pFilename ) ;
    if ( rc2 != 0 )
        PythonException::translateException() ;
}
```

Finally, you need a class to wrap the PyObjects using in the application. Here is a definition of such a class. The *PythonObject* class has three constructors to initialize, a Python object based on a C++ data type passed to, and a destructor to clean up after. It implements methods to handle calls to a Python objects/modules. It also has methods to retrieve the return values from calling into python script, while converting Python object to C++ data type.



```
namespace sedeen {
namespace python {

typedef struct _object PyObject ; // nb: from python.h

// This class wraps a PyObject* returned to us by Python.
// NOTE: It is not thread-safe!

class PythonObject
{
public:
    PythonObject( _object* pPyObject , bool decRef ) ;
    explicit PythonObject( int val ) ;
    explicit PythonObject( const char* pVal ) ;
    virtual ~PythonObject() ;
public:
    PythonObject* py_CallObject() ;
    PythonObject* py_CallObject( PythonObject& args ) ;
    PythonObject* py_CallObject( PythonObject& args , PythonObject& kywdArgs ) ;
    PythonObject* py_GetAttr( const char* pAttrName ) const ;
    void py_SetTupleItem( size_t pos , PythonObject& obj ) ;
    std::string py_ReprVal() const ;
public:
    static bool pyobject_to(PyObject*, sedeen::image::RawImage& );
    static bool pyobject_to(PyObject*, bool& value );
    static bool pyobject_to(PyObject*, int& value );
    static bool pyobject_to(PyObject*, double& value );
    static bool pyobject_to(PyObject*, float& value );
    static bool pyobject_to(PyObject*, std::string& value );
    static bool pyobject_to(PyObject*, sedeen::Color& value );
    static bool pyobject_to(PyObject*, sedeen::Size& value );

    static PythonObject* pyobject_from(const sedeen::image::RawImage& );
    static PythonObject* pyobject_from(const bool& value );
    static PythonObject* pyobject_from(const int& value );
    static PythonObject* pyobject_from(const double& value );
    static PythonObject* pyobject_from(const float& value );
    static PythonObject* pyobject_from(const std::string& value );
    static PythonObject* pyobject_from(const sedeen::Color& value );
    static PythonObject* pyobject_from(const sedeen::Size& value );

public:
    static PythonObject* py_ImportModule( const char* pModuleName ) ;
    static PythonObject* py_NewTuple( size_t nItems ) ;
    static PythonObject* py_NewDict() ;
public:
    PyObject* py_Object() const ;
private:
    PyObject* PyObject_ ; // Python object being wrapped
    bool decRef_ ; // flags if we should decref
```

INTRODUCING TO PYTHONQT

PythonQt is a dynamic Python binding for the *Qt* framework. It provides a simple way to embed python scripting language into C++/Qt application. For more details on the *PythonQt*, visit the official website: http://pythonqt.sourceforge.net/index.html.

PythonQt is open source and examples are available in the example directory. Here is a simple example shows how to integrate *PythonQt* into a *Qt* application which I barrow from their website:

```
int main( int argc, char **argv )
{
  QApplication qapp(argc, argv);

  // init PythonQt and Python
  PythonQt::init();

  // get the __main__ python module
  PythonQtObjectPtr mainModule = PythonQt::self()->getMainModule();

  // evaluate a simple python script and receive the result a qvariant:
  QVariant result = mainModule.evalScript("19*2+4", Py_eval_input);

  // create a small Qt GUI
  QVBoxLayout*  vbox = new QVBoxLayout;
  QGroupBox*    box  = new QGroupBox;
  QTextBrowser* browser = new QTextBrowser(box);
  QLineEdit*    edit = new QLineEdit(box);
  QPushButton*  button = new QPushButton(box);
  button->setObjectName("button1");
  edit->setObjectName("edit");
  browser->setObjectName("browser");
  vbox->addWidget(browser);
  vbox->addWidget(edit);
  vbox->addWidget(button);
  box->setLayout(vbox);

  // make the groupbox to the python under the name "box"
  mainModule.addObject("box", box);

  // evaluate the python script which is defined in the resources
  mainModule.evalFile(":GettingStarted.py");

  // define a python method that appends the passed text to the browser
  mainModule.evalScript("def appendText(text):\n  box.browser.append(text)");
  // shows how to call the method with a text that will be append to the browser
  mainModule.call("appendText", QVariantList() << "The ultimate answer is ");

  return qapp.exec();
}
```

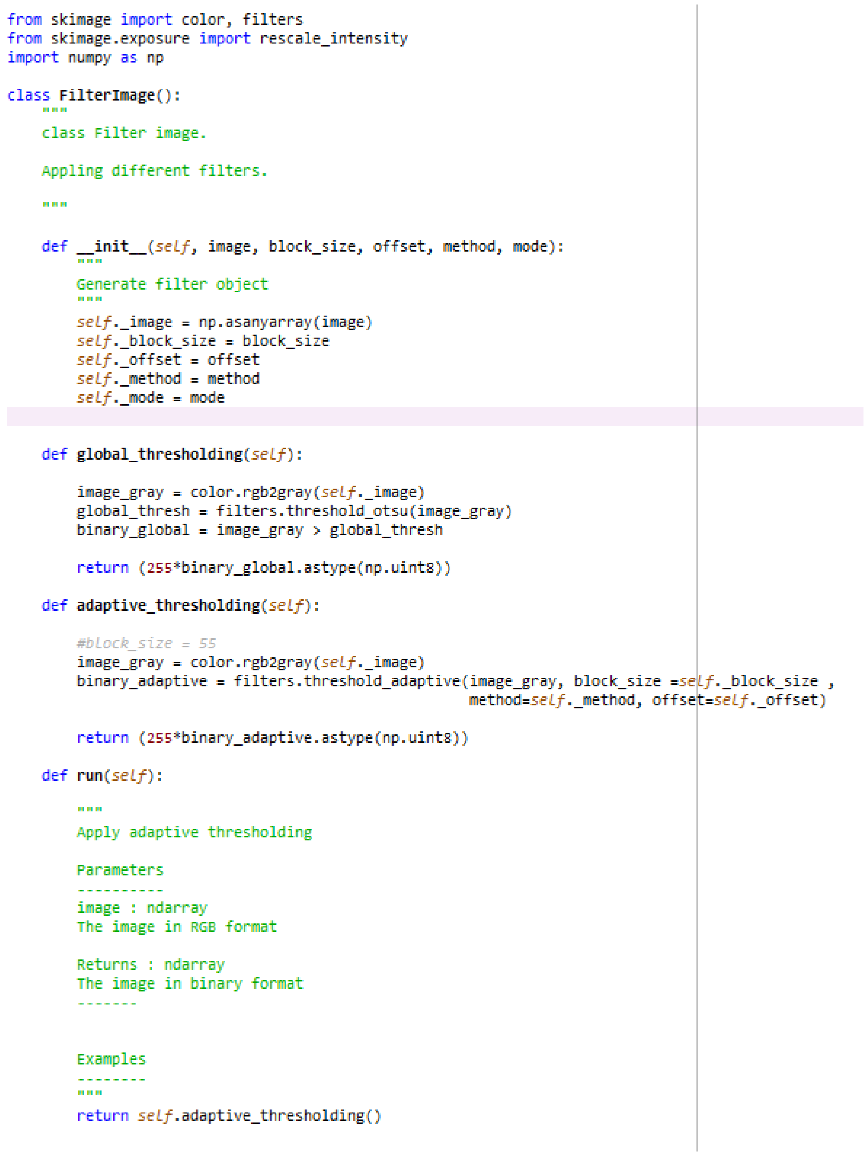
Also, it would be very useful to see how to use the 3rdparty libraries in the embedded Python. This tutorial aimed to cover the main issues that are required to be addressed for embedding Python in C++ application. However, this tutorial is certainly nowhere near complete in terms of access to the underlying Python/C API, but it is a good starting point to build upon.

WRITING PYTHON SCRIPT

Here are some useful notes that you need to consider when writing a Python script.

1- Before calling any Python script inside Sedeen, make sure it runs properly from Python interpolate.
2- We need to specify a guild line or standard for Python developer.
3- Most of the Python image processing methods returns image as double or float values. Double or float image type is not supported by Sedeen. You need to make sure that the return image is a numpy array and the data type is either uint8, int8, uint16 or int16 (image data types which are supported by Sedeen).
4- Python image processing packages need to be tested before being supported inside Sedeen.
5- Python needs to be recompiled to support both release and debug configurations. Python is only available in release mode.

I have also provided a Python scripts to apply different filters to image as an example. Each function of the class applies a specific filter to image. For example, *adaptive_thresholding* function applies an adaptive threshold also known as local or dynamic thresholding to an array. The threshold value is calculated as the weighted mean of the local neighborhood of a pixel subtracted by a constant (offset).



```
from skimage import color, filters
from skimage.exposure import rescale_intensity
import numpy as np

class FilterImage():
    """
    class Filter image.

    Appling different filters.

    """

    def __init__(self, image, block_size, offset, method, mode):
        """
        Generate filter object
        """
        self._image = np.asanyarray(image)
        self._block_size = block_size
        self._offset = offset
        self._method = method
        self._mode = mode


    def global_thresholding(self):

        image_gray = color.rgb2gray(self._image)
        global_thresh = filters.threshold_otsu(image_gray)
        binary_global = image_gray > global_thresh

        return (255*binary_global.astype(np.uint8))

    def adaptive_thresholding(self):

        #block_size = 55
        image_gray = color.rgb2gray(self._image)
        binary_adaptive = filters.threshold_adaptive(image_gray, block_size =self._block_size ,
                                                     method=self._method, offset=self._offset)

        return (255*binary_adaptive.astype(np.uint8))

    def run(self):

        """
        Apply adaptive thresholding

        Parameters
        ----------
        image : ndarray
        The image in RGB format

        Returns : ndarray
        The image in binary format
        -------


        Examples
        --------
        """
        return self.adaptive_thresholding()
```

The use of the plugin and call to Python script is described in details below:

1. Open the WSI image.
2. Load the “Python Extension with Kernel” plugin from the pulldown list of **Algorithm** (Fig1).

3. Users need to specify the name of the python class by typing the class name in “Python class name” filed.
4. Users also can specify the algorithm parameters which are **block_size**, **method**, **offset**, and **mode.** Blok-size is the size of kernel which is used to calculate the threshold value based on the specified method. Offset is a constant subtracted from weighted mean of neighborhood to calculate the local threshold value. Mode determines how the array boarder are handled (Fig2).

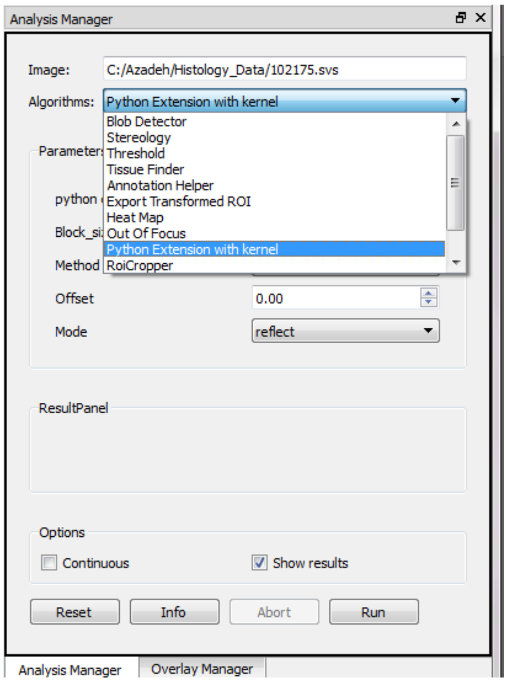

Analysis Manager

Image: C:/Azadeh/Histology_Data/102175.svs

Algorithms: Python Extension with kernel

Blob Detector
Stereology
Threshold
Tissue Finder
Annotation Helper
Export Transformed ROI
Heat Map
Out Of Focus
Python Extension with kernel
RoiCropper

Offset: 0.00

Mode: reflect

Fig1. Select the “Python Extenstion with Kernel” plugin**.**

5. Click on the Run button will execute the algorithm and ask for the specific Python script that user would like to call. Users can zoom in or out, or change the parameters to see the desired results (Fig3).

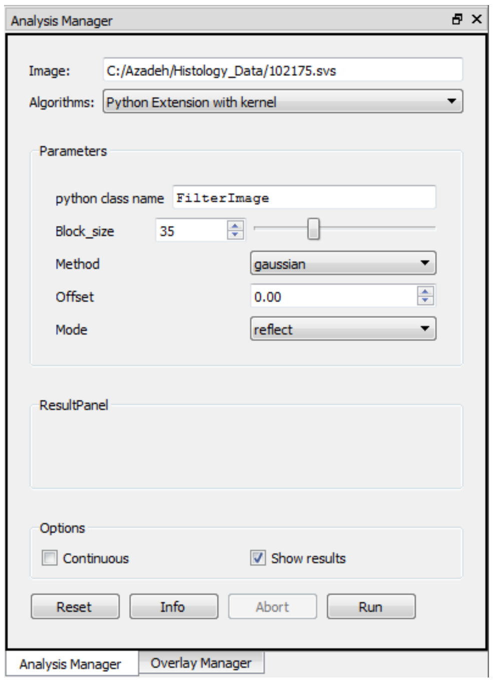

Analysis Manager

Image: C:/Azadeh/Histology_Data/102175.svs

Algorithms: Python Extension with kernel

Parameters

python class name: FilterImage

Block_size: 35

Method: gaussian

Offset: 0.00

Mode: reflect

Fig2. Algorithm parameters.

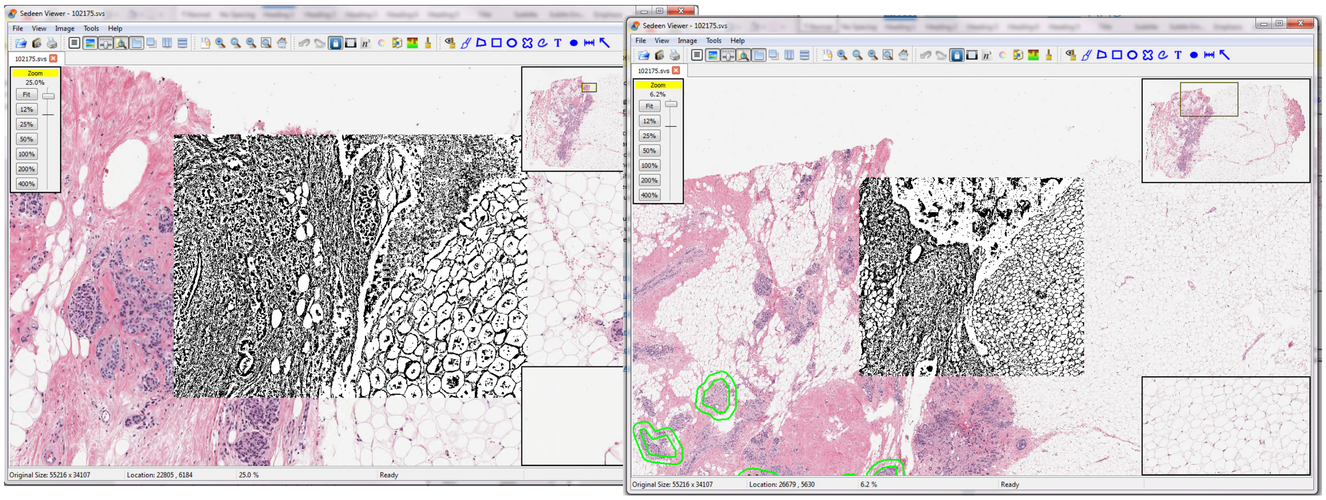

Sedeen Viewer - 102175.svs
File View Image Tools Help
Original Size: 55216 x 34107
Location: 22805 , 6184
25.0 %
Ready
Sedeen Viewer - 102175.svs
Original Size: 55216 x 34107
Location: 26679 , 5630
6.2 %
Ready

Fig3. Result of running plugin in different resolution.

RESOURCES

(python.org) https://docs.python.org/2/c-api/index.html

https://docs.python.org/2/c-api/intro.html#objects-types-and-reference-counts

https://docs.python.org/2/extending/embedding.html

http://www.linuxjournal.com/article/3641?page=0,0

https://doc.qt.io/archives/qq/qq23-pythonqt.html

https://www.codeproject.com/Articles/11805/Embedding-Python-in-C-C-Part-I