**SciPy**

**Scilab Python**

**Tool Box Development**

1. **Selecting APIs for implementation of SciPy**

This document is split into three segments, each segment aims to solve a particular requirement. The three segments being

* Creating a custom type to hold python variables
* Creating gateway functions using custom variable
* Building the toolbox

We will be using the ***types*** api offered by scilab to create the gateway functions and our custom python variable. The toolbox will also be following the structure explained by the ***toolbox\_skeleton*** toolbox.

1. **Creating the custom python variable type**

For this segment, a basic understanding of the following is essential

1. C++ inheritance
2. Creating C++ headerfiles
3. Python’s extending and embedding api for C/C++, provided by the header file python.h

Scilab provides the ability to make custom user types through the ***class UserType*** provided by the ***headerfile user.hxx*** located in {scilab installation}/modules/ast/includes/types/

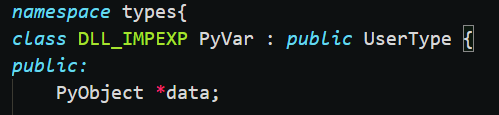
This class implements all the core functionalities that makes up a scilab data type, while allowing the user to implement the data type itself through *virtual functions.* The class offers a variety of virtual functions, The important functions would be outlined further

The process is summarized as

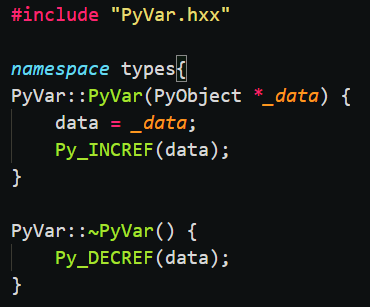
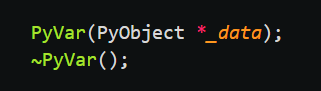
* Create a new headerfile in the toolbox location
* Include the headerfiles ‘python.h’, ‘alltypes.hxx’ and ‘user.hxx’ in the newly created header file
* Now create a class in the ‘***types’ namespace*** that inherits from ***UserType.***
* Add appropriate storage class specifiers to enable dll interfacing when building.
* Plan for the new features to be added to this custom data type.



* The next step would be to actually add a data member for storing our python variable. Many different data members could be added to hold the different attributes of the python variable. To keep things simpler, only one data member will be added. This data member will hold the actual PyObject, with the type as PyObject\*.



* Now we need to provide a constructor and destructor for the class, with best possible customization
* A constructor is created that takes a PyObject\* as argument and assigns it to our data member.
* A destructor will be reducing the reference count for the PyObject.
* Declare the functions in the newly created header file and define them in the respective source C++ file.



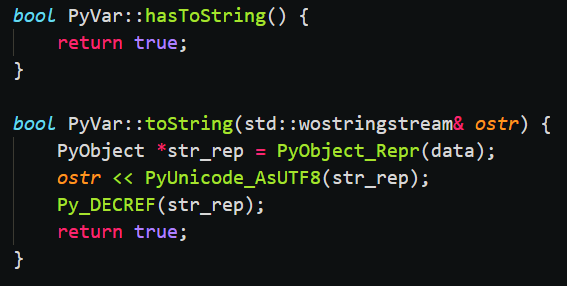
* overloading the virtual functions is the next step. As previously stated, scilab provides a number of these functions, and some essestial functions are highlighted in this section
* ***hasToString, toString, extract, isInvokable and invoke.*** There are also a few trivial functions like ***getTypeStr and getShortTypeStr*** which are self-explanatory.

**hasToString and toString**

The function hasToString needs to return a bool value. If it returns true, then scilab will call toString method whenever a string representation of our custom object is required – like printing to the scilab console for example. If it returns false, then scilab will never call the toString method and will instead call the inbuilt macro corresponding to your data type.

The function toString has to take a wide string stream as argument and write the string representation of our custom python data and then return true. To implement this function, we can make use of the ***PyObject\_Repr*** offered by the python api.

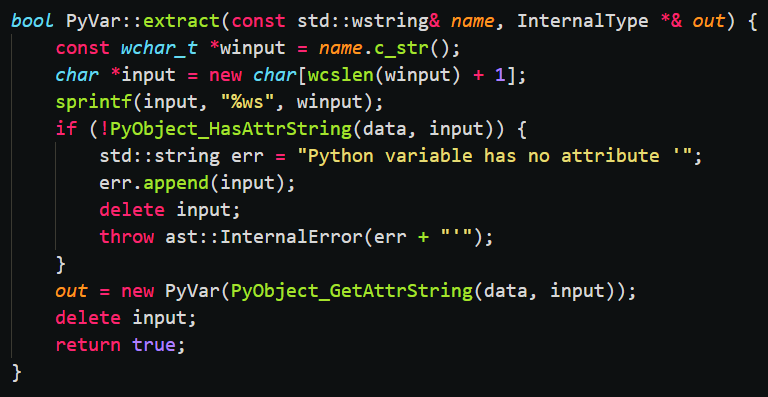
Declare the two functions in the headerfile and define them in the respective C++ source file.



**extract**

The function extract will be called whenever some attribute is extracted from our custom data type – like x.**attribute.** The function has to take a wide string ‘name’ and a pointer to internal type ‘out’ as arguments and must return true. The wide string will contain the name of the attribute – for x.abc, name = “abc” – and the extracted output will be assigned to the internal type pointer.

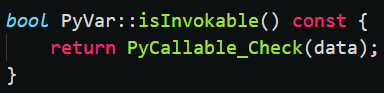
We can check if our python data has the attribute by using the function ***PyObject\_HasAttrString*** and we can retrieve the attribute as a PyObject by using the function ***PyObject\_GetAttrString.*** We can then create our variable of our custom data type with the extracted attribute and assign it to ‘out’. Declare the function in the headerfile and define it in the respective C++ source file.



**isInvokable and invoke**

The function isInvokable needs to return a bool value. If It returns true, then scilab will call the invoke method whenever our custom data type is called like a function – x(arg1, arg2…). If it returns false, scilab will never call the invoke function and will treat the data as uncallable.

We will need to return true if the python data is a method and false otherwise, for this purpose we can use the function ***PyCallable\_Check*.**

****

The invoke function takes

a typed list in; which will contain the arguments passed,

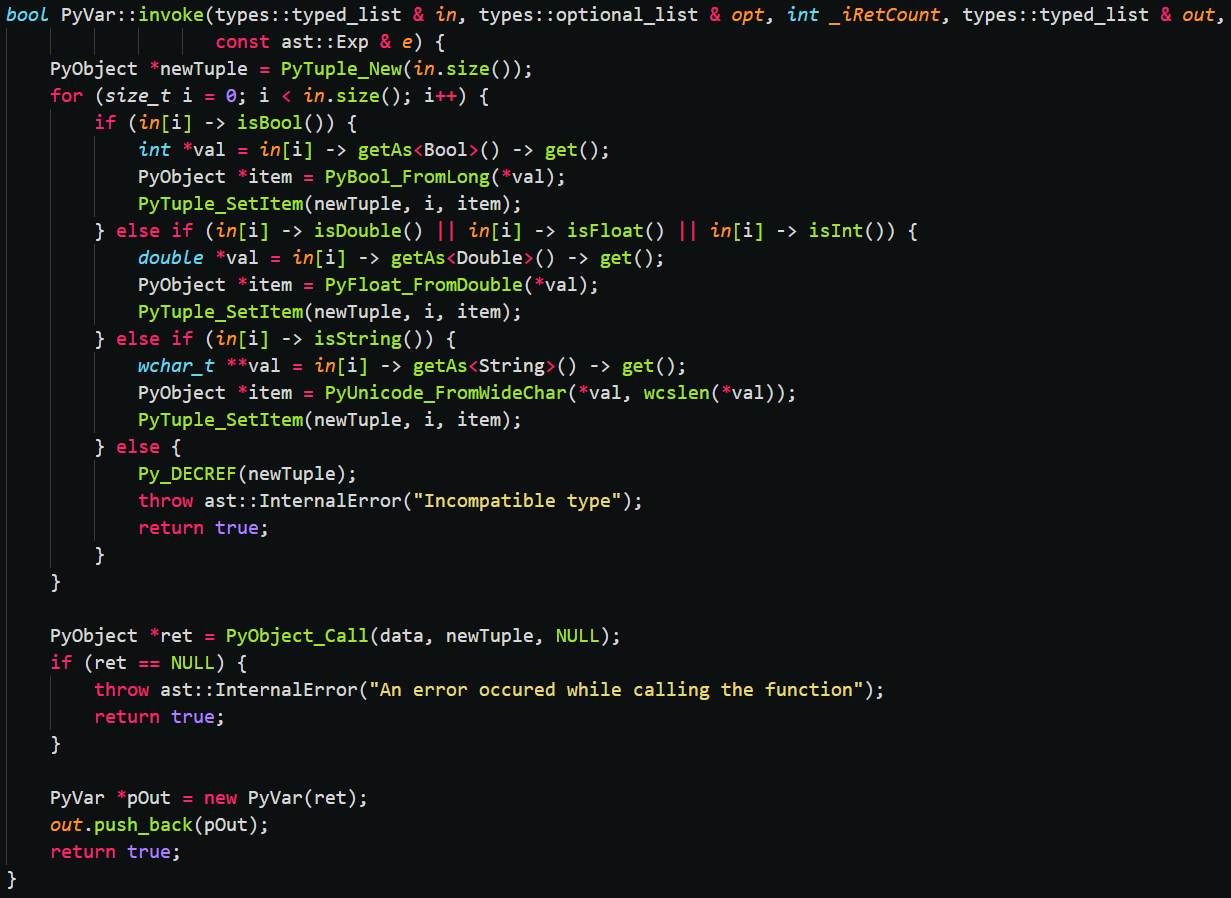
an optional list opt; which will contain optional arguments if configured,

an integer \_iRetCount; which will contain the number of output arguments,

a typed list out; which will have to be written with the output data,

and the abstract expression node e; which can be used while throwing exceptions and errors.

For this purpose, we can make use of the function ***PyObject\_Call*** function provided by the python api. We will have to convert the list of arguments into a python tuple to call the function, the detailed description of doing this is given in the next segment.



Declare the functions in the headerfile and define them in the source C++ file.

Note that when we access methods like x.foo(arg), scilab will first call extract function for x.foo and then call the invoke method on the returned object with the arguments.

x.foo(arg) 🡺 (x.extract(‘foo’)).invoke(arg)

Now our custom data type is ready, we can move on to writing gateway functions using our new data type.

1. **Creating Gateway Functions**

For this segment, you must have a basic understanding of

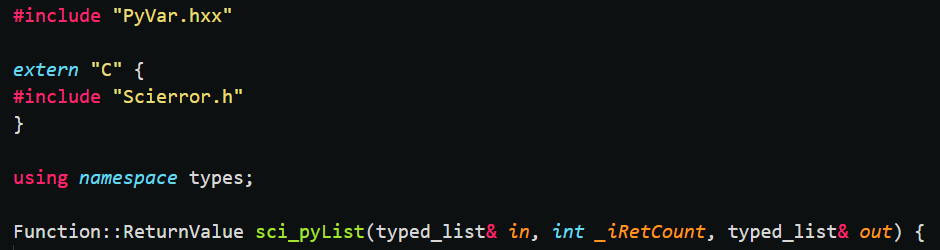
1. Python’s api for C/C++
2. Scilab data types

Scilab provides a format for functions that uses the same types api, using this we can write functions in C++ which can be called through scilab, these functions are called gateways.

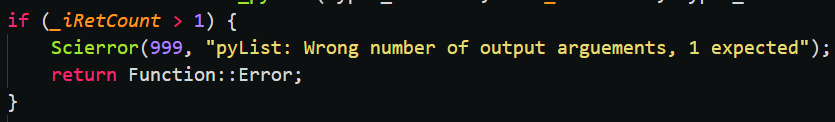
The general format to be followed for the gateway functions is

* Function must take in 3 arguments, a typed list ‘in’ which contains the arguments passed, an integer ‘\_iRetCount’ which contains the number of output arguments and a typed list ‘out’ which is supposed to be updated with the return value
* Function must return Function::Return Value, which is just an integer enumeration corresponding to the status of the function, if an error has occurred, return Function::Error otherwise, if it was successful return Function::OK
* It is also good convention to name your function and source file sci\_YourFunction, where ‘YourFunction’ will be the function’s name in scilab.

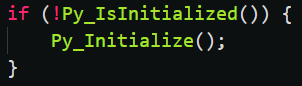
Let’s implement a function to create a python list as an example. Create a C++ file named sci\_pyList.cpp and include our *headerfile from the previous segment* and ***Scierror.h*** for error handling. Now create a function with the arguments as mentioned above and name it the same as the file.



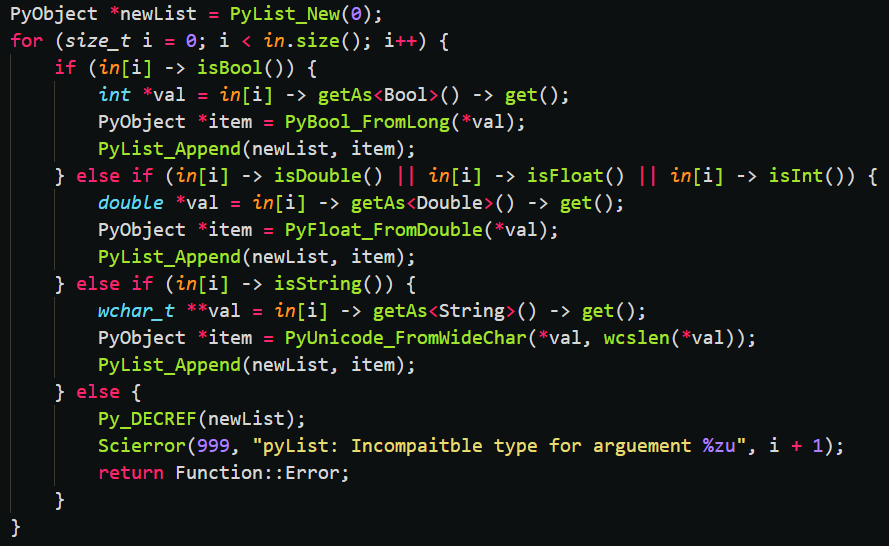
First, check if the number of output variables is one, if it is not then send an error message using the ***Scierror*** function and return error status.



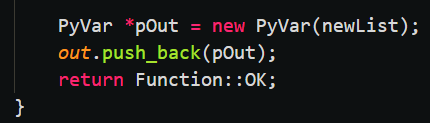
Now we can move on to actual implementation part, we would normally want to call ***Py\_Initialize*** through another function or when loading the toolbox, but for the purpose of keeping this function standalone, lets call Py\_Initialize if it has not already been called before. We can check if Py\_Initialize has been called by using the function ***Py\_IsInitialized***.



Next, we need to create an empty list to which the passed arguments will be added. We can do this by using the ***PyList\_New*** function. Now we need to convert all the passed scilab arguments into python data. For this the *scilab types api offers functions to check if the data is of specific datatypes*, using those function we first initially convert the scilab data into C++ data and then call the *appropriate python api function to create a python object with that data*. Then we can append it to the newly created list. If the data is not of appropriate type, we need to display an error message and return error status.



Finally, we need to create a variable of our custom data type with the list, the constructor we defined comes in handy here. And then we add it to the ‘out’ list and return success status.



After making all the gateway functions, we can move to the final stage of building the toolbox.

1. **Building The Toolbox**

For this segment, you must have a basic understanding of how to build a scilab toolbox. We will be following the format as described by the ***toolbox\_skeleton*** toolbox.

Place the custom data type headerfile and source file in the /src/cpp directory.

Place the source files of the gateway functions in the /sci\_gateway/cpp directory.

We will need an installation of python to be included with the toolbox so that we can add the library files. Place the python installation inside a newly created folder /python/

We will be following the basic procedures for building the toolbox, the configuration of the ***builder\_cpp.sce*** in /src/cpp and ***builder\_gateway\_cpp.sce*** in /sci\_gateway/cpp will be the main focus.

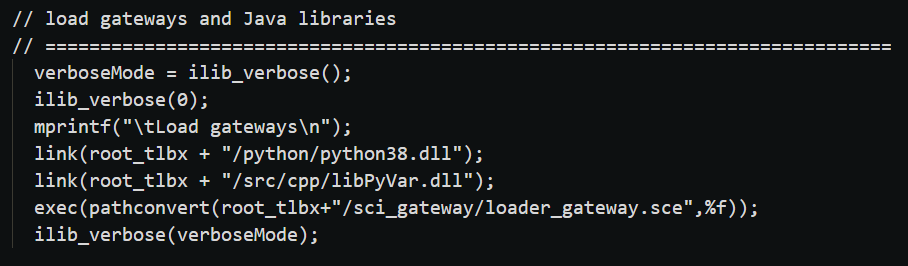
In builder\_cpp.sce, add the python include directory, which should be /python/include, to the CFLAGS along with the flag for handling import/export storage class specifier. Add the path to python38.lib, which should be /python/libs/python38.lib, to the LDFLAGS.



In builder\_gateway\_cpp.sce, add the python include path, and the types api include path, which should be {scilab installation}/modules/ast/includes, and the path to our headerfile, which should be /src/cpp, to the CFLAGS. Add the path to python38.lib and the lib file that would be created by our headerfile, which should be /src/cpp/libPyVar.lib, to the LDFLAGS.



Finally, we need to link the dll files when we load the toolbox, so we add the code to link python38.dll and the dll file which will be generated by our headerfile in our \*\*\*.start file located at /etc/. We add these before we load the gateways.



After making all the configuration settings, run the builder.sce file to build the toolbox. Once the toolbox is successfully built, you can test it out by loading it using the loader.sce file. When your toolbox is ready for full release, you can run the builder, then zip the entire root folder, now your toolbox can be installed through the atominstall command.