1. Installation of Python 3.8 on Ubuntu Linux OS

1. Open the Terminal by pressing **Ctrl + Alt + t** keys
2. Enter the below commands to install **Python 3.8**
```
>sudo add-apt-repository ppa:deadsnakes/ppa
>sudo apt update
>sudo apt install python3.8
>sudo apt install libpython3.8-dev
>sudo apt-get install python3.8-dev
```

2. Scithon - Scilab Python Tool Box Development

1. Selecting APIs for implementation of Scithon

This document has three segments. Each segment aims to solve a particular requirement. The three segments are:

- 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.

2. Creating the custom python variable type

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

- C++ inheritance
- Creating C++ header files
- Python's extended and embedded API for C/C++, provided by the header file **python.h**

Scilab provides the ability to make custom user types through the ***class UserType***. This is provided by the header file ***user.hxx*** located in

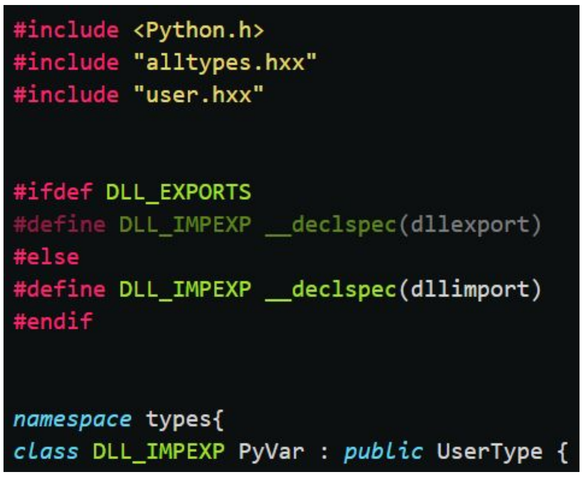
{scilab installation}/modules/ast/includes/types/

This class implements all the core functionalities that make 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: (Fig 1.1)

- Create a new header file in the toolbox location.
- Include the header files '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.

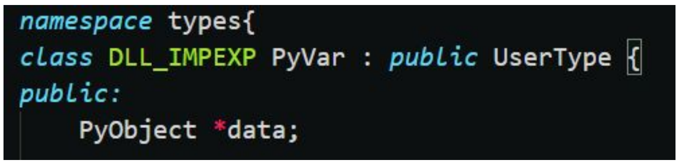
Fig 1.1



```
#include <Python.h>
#include "alltypes.hxx"
#include "user.hxx"


#ifdef DLL_EXPORTS
#define DLL_IMPEXP __declspec(dllexport)
#else
#define DLL_IMPEXP __declspec(dllimport)
#endif


namespace types{
class DLL_IMPEXP PyVar : public UserType {
```

- 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*.(Fig 1.2)

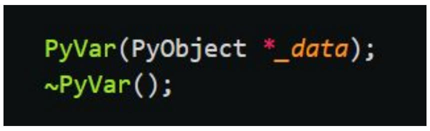
Fig 1.2



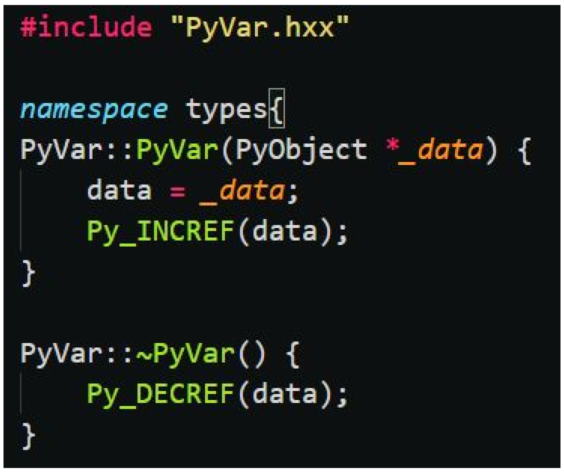
```
namespace types{
class DLL_IMPEXP PyVar : public UserType {
public:
    PyObject *data;
```

- 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 an argument and assigns it to our data member. (Fig 1.3)
- 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.

Fig 1.3



```
PyVar(PyObject *_data);
~PyVar();
```



```
#include "PyVar.hxx"

namespace types{
PyVar::PyVar(PyObject *_data) {
    data = _data;
    Py_INCREF(data);
}

PyVar::~PyVar() {
    Py_DECREF(data);
}
```

- Overloading the virtual functions is the next step. As previously stated, scilab provides a number of these functions, and some essential 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.

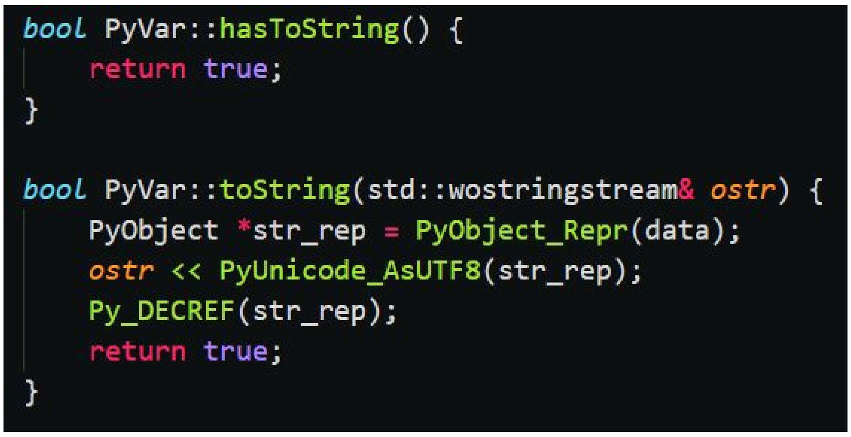
<u>hasToString and toString :</u> (Fig 1.4)

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 header file and define them in the respective C++ source file.

Fig 1.4



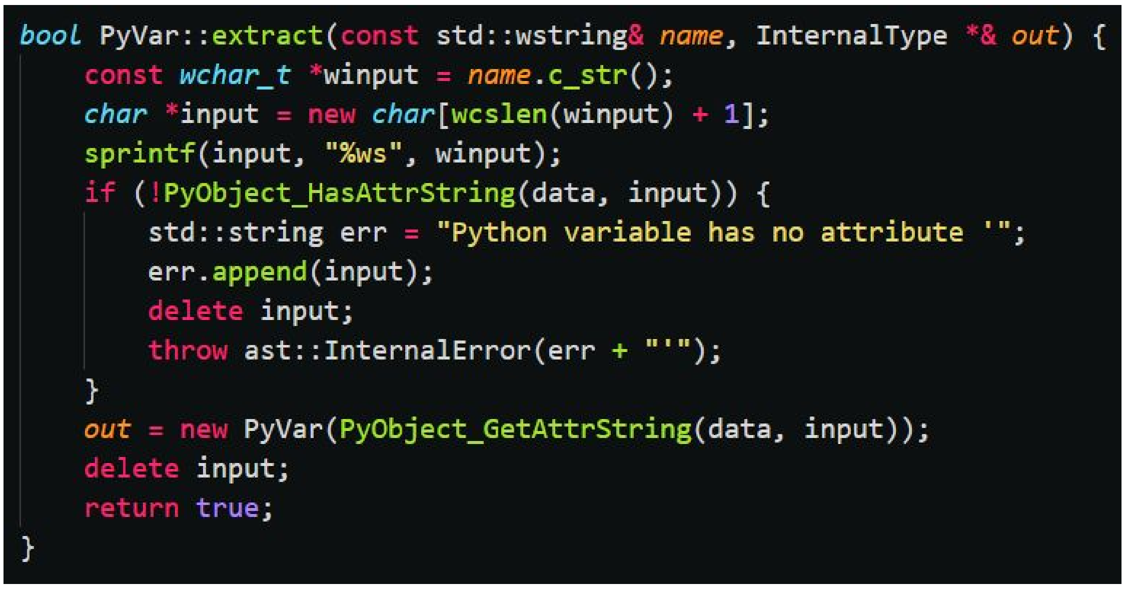
```
bool PyVar::hasToString() {
    return true;
}

bool PyVar::toString(std::wostringstream& ostr) {
    PyObject *str_rep = PyObject_Repr(data);
    ostr << PyUnicode_AsUTF8(str_rep);
    Py_DECREF(str_rep);
    return true;
}
```

extract (Fig 1.5)

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 header file and define it in the respective C++ source file.

Fig 1.5



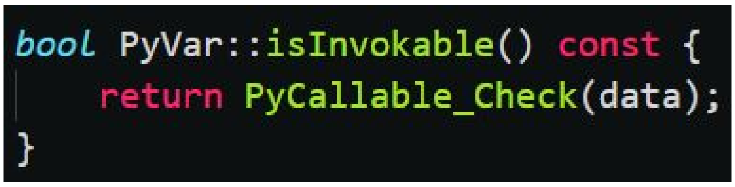
```
bool PyVar::extract(const std::wstring& name, InternalType *& out) {
    const wchar_t *winput = name.c_str();
    char *input = new char[wcslen(winput) + 1];
    sprintf(input, "%ws", winput);
    if (!PyObject_HasAttrString(data, input)) {
        std::string err = "Python variable has no attribute '";
        err.append(input);
        delete input;
        throw ast::InternalError(err + "'");
    }
    out = new PyVar(PyObject_GetAttrString(data, input));
    delete input;
    return true;
}
```

isInvokable and invoke (Fig 1.6)

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.***

Fig 1.6



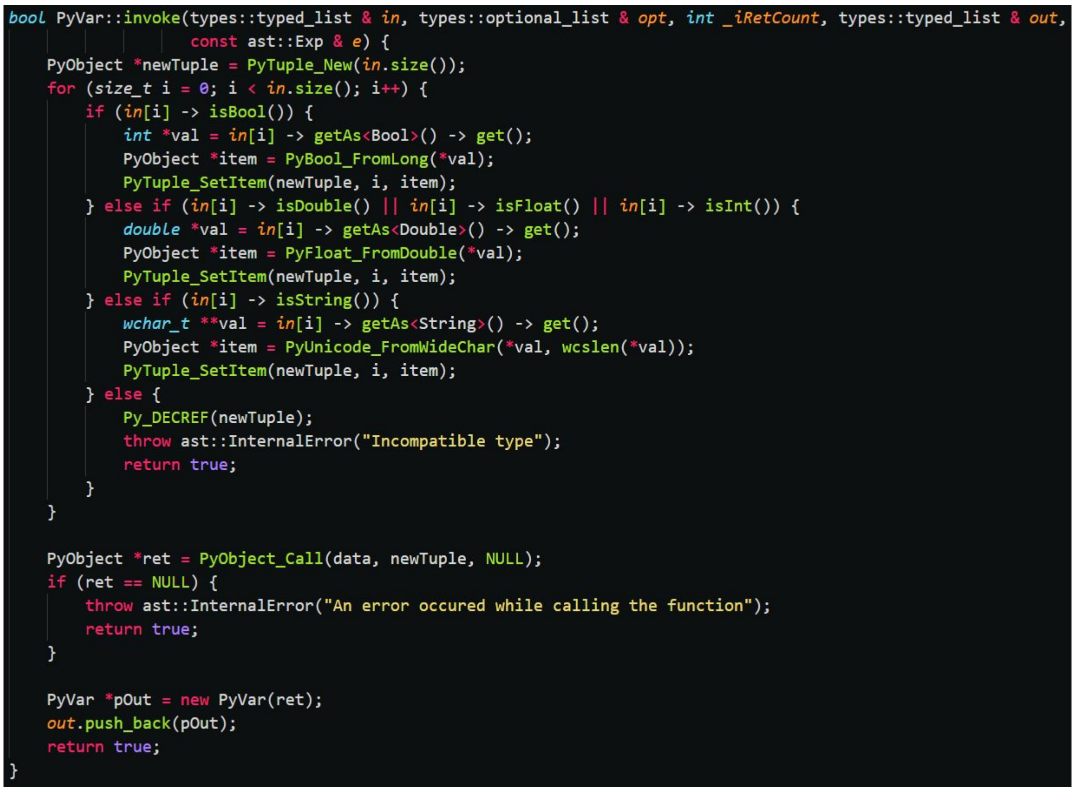
```cpp
bool PyVar::isInvokable() const {
    return PyCallable_Check(data);
}
```

The invoke function takes: (Fig 1.7)

- 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.

Fig 1.7



```cpp
bool PyVar::invoke(types::typed_list & in, types::optional_list & opt, int _iRetCount, types::typed_list & out,
                    const ast::Exp & e) {
    PyObject *newTuple = PyTuple_New(in.size());
    for (size_t i = 0; i < in.size(); i++) {
        if (in[i] -> isBool()) {
            int *val = in[i] -> getAs<Bool>() -> get();
            PyObject *item = PyBool_FromLong(*val);
            PyTuple_SetItem(newTuple, i, item);
        } else if (in[i] -> isDouble() || in[i] -> isFloat() || in[i] -> isInt()) {
            double *val = in[i] -> getAs<Double>() -> get();
            PyObject *item = PyFloat_FromDouble(*val);
            PyTuple_SetItem(newTuple, i, item);
        } else if (in[i] -> isString()) {
            wchar_t **val = in[i] -> getAs<String>() -> get();
            PyObject *item = PyUnicode_FromWideChar(*val, wcslen(*val));
            PyTuple_SetItem(newTuple, i, item);
        } else {
            Py_DECREF(newTuple);
            throw ast::InternalError("Incompatible type");
            return true;
        }
    }

    PyObject *ret = PyObject_Call(data, newTuple, NULL);
    if (ret == NULL) {
        throw ast::InternalError("An error occured while calling the function");
        return true;
    }

    PyVar *pOut = new PyVar(ret);
    out.push_back(pOut);
    return true;
}
```

Declare the functions in the header file 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.

3. Creating Gateway Functions

For this segment, you must have a basic understanding of:

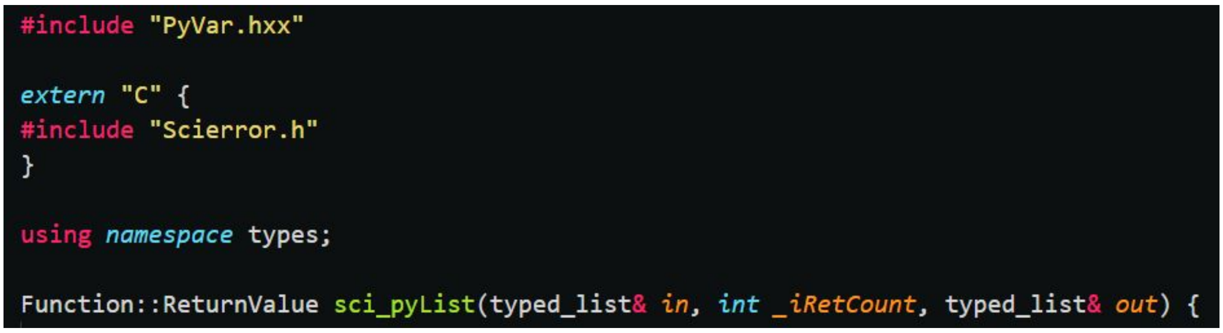
- Python's api for C/C++
- Scilab data types

Scilab provides a format for functions that uses the same types of 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: (Fig 1.8)

- 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 a good convention to name your function and source file sci_YourFunction, where 'YourFunction' will be the function's name in scilab.

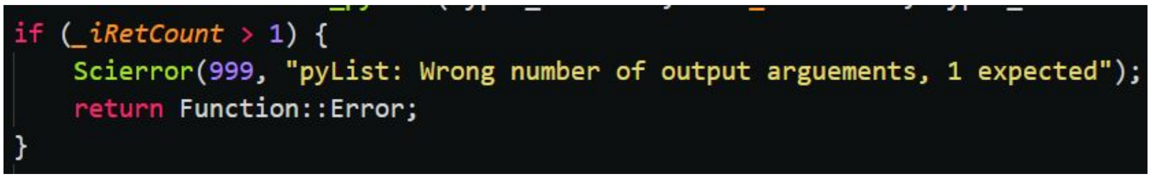
Let's implement a function to create a python list as an example. Create a C++ file named sci_pyList.cpp and include our header file from the previous segment and ***Scierror.h*** for error handling. Now create a function with the arguments as mentioned above and name it the same as the file.

Fig 1.8



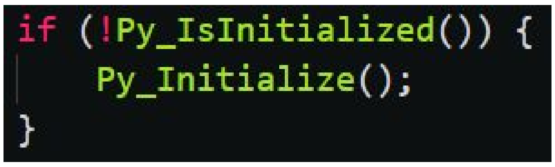
```
#include "PyVar.hxx"

extern "C" {
#include "Scierror.h"
}

using namespace types;

Function::ReturnValue sci_pyList(typed_list& in, int _iRetCount, typed_list& out) {
```

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.



```
if (_iRetCount > 1) {
    Scierror(999, "pyList: Wrong number of output arguements, 1 expected");
    return Function::Error;
}
```

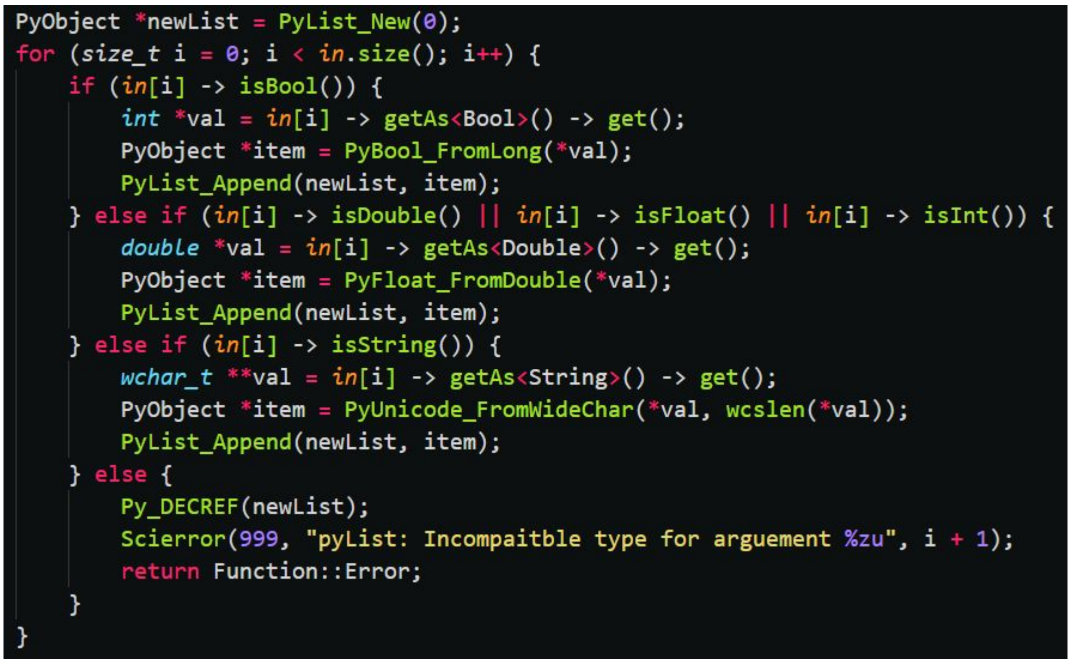
Now we can move on to the actual implementation part, we would normally want to call Py_Initialize through another function or when loading the toolbox, but for the purpose of keeping this function standalone, let's call Py_Initialize if it has not already been called before. We can check if Py_Initialize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



```
if (!Py_IsInitialized()) {
    Py_Initialize();
}
```

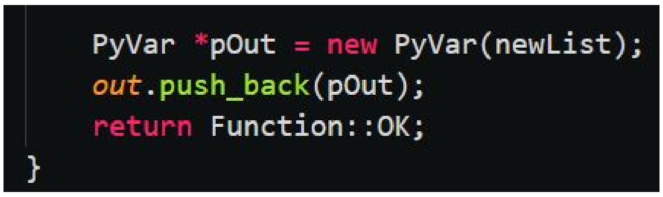
Next, we need to create an empty list to which the passed arguments will be added. We can do this by using the ***PyList_New*** function (Fig 1.9). Now we need to convert all the passed scilab arguments into python data. For this the scilab types api offers functions to check if the data is of specific data types, using those functions 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.

Fig 1.9



```
PyObject *newList = PyList_New(0);
for (size_t i = 0; i < in.size(); i++) {
    if (in[i] -> isBool()) {
        int *val = in[i] -> getAs<Bool>() -> get();
        PyObject *item = PyBool_FromLong(*val);
        PyList_Append(newList, item);
    } else if (in[i] -> isDouble() || in[i] -> isFloat() || in[i] -> isInt()) {
        double *val = in[i] -> getAs<Double>() -> get();
        PyObject *item = PyFloat_FromDouble(*val);
        PyList_Append(newList, item);
    } else if (in[i] -> isString()) {
        wchar_t **val = in[i] -> getAs<String>() -> get();
        PyObject *item = PyUnicode_FromWideChar(*val, wcslen(*val));
        PyList_Append(newList, item);
    } else {
        Py_DECREF(newList);
        Scierror(999, "pyList: Incompaitble type for arguement %zu", i + 1);
        return Function::Error;
    }
}
```

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.



```
    PyVar *pOut = new PyVar(newList);
    out.push_back(pOut);
    return Function::OK;
}
```

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

4. 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 header file and source file in the /src/cpp directory.

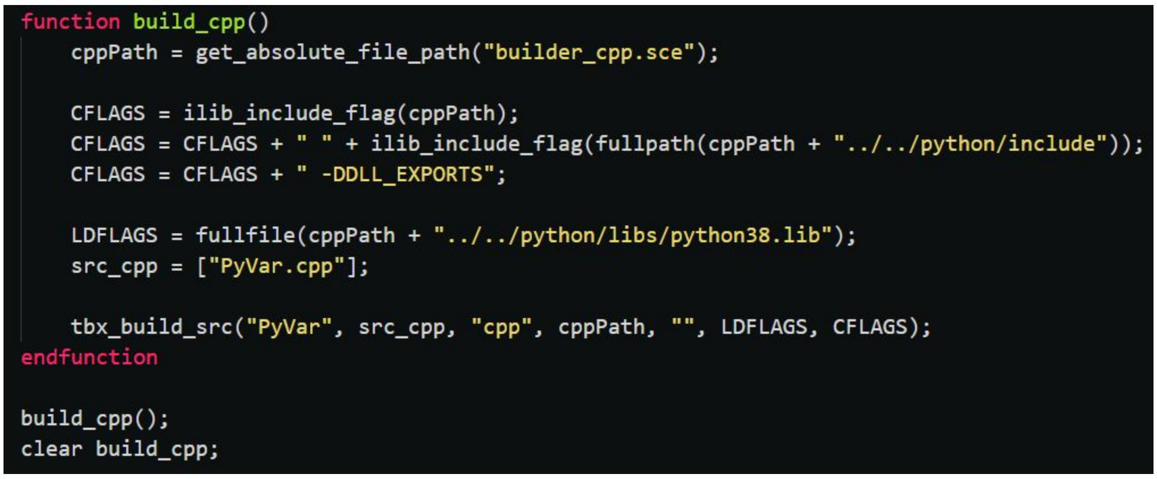
Place the source files of the gateway functions in the /sci_gateway/cpp directory.

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. (Fig 2.0)

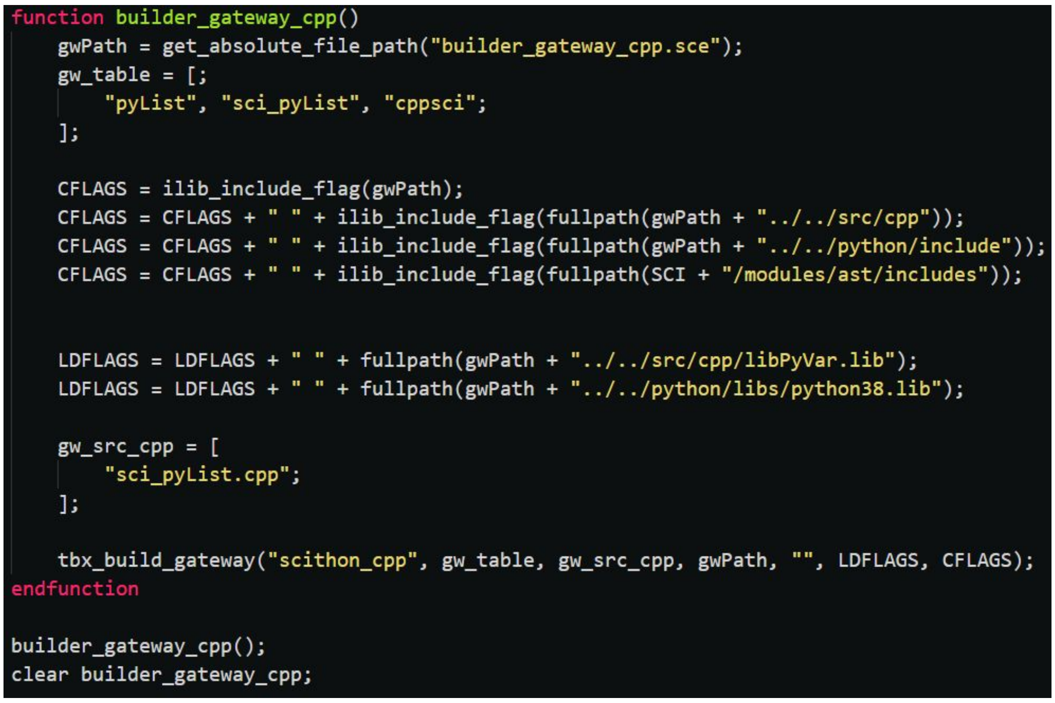
In builder_cpp.sce, add the python include directory, which should be /python/include, to the CFLAGS along with the flag for handling import/export storage class specifier. Add the path to python38.lib, which should be /python/libs/python38.lib, to the LDFLAGS.

Fig 2.0



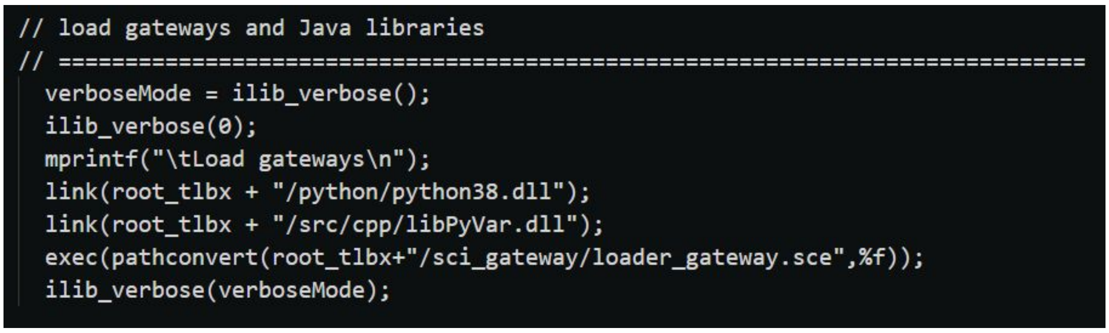
```
function build_cpp()
    cppPath = get_absolute_file_path("builder_cpp.sce");

    CFLAGS = ilib_include_flag(cppPath);
    CFLAGS = CFLAGS + " " + ilib_include_flag(fullpath(cppPath + "../../python/include"));
    CFLAGS = CFLAGS + " -DDLL_EXPORTS";

    LDFLAGS = fullfile(cppPath + "../../python/libs/python38.lib");
    src_cpp = ["PyVar.cpp"];

    tbx_build_src("PyVar", src_cpp, "cpp", cppPath, "", LDFLAGS, CFLAGS);
endfunction

build_cpp();
clear build_cpp;
```

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 header file, which should be /src/cpp, to the CFLAGS. Add the path to python38.lib and the lib file that would be created by our header file, which should be /src/cpp/libPyVar.lib, to the LDFLAGS.



```
function builder_gateway_cpp()
    gwPath = get_absolute_file_path("builder_gateway_cpp.sce");
    gw_table = [;
        "pyList", "sci_pyList", "cppsci";
    ];

    CFLAGS = ilib_include_flag(gwPath);
    CFLAGS = CFLAGS + " " + ilib_include_flag(fullpath(gwPath + "../../src/cpp"));
    CFLAGS = CFLAGS + " " + ilib_include_flag(fullpath(gwPath + "../../python/include"));
    CFLAGS = CFLAGS + " " + ilib_include_flag(fullpath(SCI + "/modules/ast/includes"));


    LDFLAGS = LDFLAGS + " " + fullpath(gwPath + "../../src/cpp/libPyVar.lib");
    LDFLAGS = LDFLAGS + " " + fullpath(gwPath + "../../python/libs/python38.lib");

    gw_src_cpp = [
        "sci_pyList.cpp";
    ];

    tbx_build_gateway("scithon_cpp", gw_table, gw_src_cpp, gwPath, "", LDFLAGS, CFLAGS);
endfunction

builder_gateway_cpp();
clear builder_gateway_cpp;
```

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 header file in our ***. start file located at /etc/. We add these before we load the gateways.



```
// load gateways and Java libraries
// =============================================================================
  verboseMode = ilib_verbose();
  ilib_verbose(0);
  mprintf("\tLoad gateways\n");
  link(root_tlbx + "/python/python38.dll");
  link(root_tlbx + "/src/cpp/libPyVar.dll");
  exec(pathconvert(root_tlbx+"/sci_gateway/loader_gateway.sce",%f));
  ilib_verbose(verboseMode);
```

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.

3. Go through the below links to understand the code inside the PyVar.CPP file.

https://docs.python.org/3/extending/embedding.html
https://docs.python.org/3/c-api/object.html
https://docs.python.org/3/c-api/sequence.html
https://docs.python.org/3/c-api/dict.html
https://docs.python.org/3.8/c-api/list.html
https://docs.python.org/3/c-api/tuple.html

4. Explanation of Gateway Functions in Scithon

1. **startPy**

The first few functions contain error/syntax checking. If no error is encountered, the Initialize function defined in /src/c/Pythoninstance.c is called, it sets up a way to capture the python's console output and starts the python interpreter.

2. quitPy

Similar to startPy, this function performs error/syntax checking and then calls the Finalize function defined in /src/c/Pythoninstance.c, this simply calls the Py_Finalize function to stop the interpreter.

3. **py**

First few lines perform error/syntax checking. Then the passed parameter is retrieved as a widestring and then converted to a character string. Then the PyRun_SimpleString function is called which will run the code in the string. The output of the code is then retrieved by use of some functions defined in /src/c/Pythoninstance.c and then converted to a Scilab string object and then added to the return list.

4. **pyExec**

First few lines perform error/syntax checking. Similar to the py function, the passed arguments is retrieved as a widestring and converted to a character string. We then open the file at the location in the string, if the file doesn't exist an appropriate error message is displayed and then the function ends with an error. Otherwise, the file is run by calling the PyRun_SimpleFileEx function and the output is retrieved by using some functions defined in /src/c/Pythoninstance.c and then added to the return list.

5. **pyGet**

First few lines perform error/syntax checking. Similar to the previous two functions, the passed arguments is retrieved as a widestring and then converted to a character string. Then the function GetPyObject which is defined in src/c/Pythoninstance.c is called which returns null if an object of the given name doesn't exist or if an error is encountered, if not then the function returns the PyObject pointer to the named variable. A new PyVar is created with this PyObject and then added to the return list.

6. **pyImport**

First few lines contain error/syntax checking. Once again, the passed argument is retrieved as a widestring and then converted to a character string. Then the PyImport_ImportModule function is called, this function would import and return a reference to the module named in the string, on an error it would return null. So we check if the output is null and print appropriate error messages and return error if it is. Otherwise a PyVar is created from the reference to the imported module and then added to the return list.

7. **pyList**

First few lines contain error/syntax checking. We first create an empty list by using the PyList_New function and then loop through the arguments passed. Each argument is converted to a PyVar and then their data is appended to the list. After all the arguments have been looped through, a PyVar is made from the list and then added to the return list.

8. **pyTuple**

First few lines contain error/syntax checking. Identical to the pyList function, but since tuples are immutable the size is set when creating the tuple using the PyTuple_New function and then the arguments passed are looped through, each argument is converted to a PyVar and their data is set in their respecting indices in the tuple by using the PyTuple_SetItem function. After the loop is finished, a PyVar is made from the tuple and added to the return list.

9. **pyDict**

First few lines contain error/syntax checking. Similar to the pyList function, an empty dictionary is created by using the PyDict_New function. The arguments passed are then iterated through two at a time, first of the pair is considered as the key while the latter is taken as the value. Both the scilab values are converted to PyVar and the pair is inserted into the dictionary by using the PyDict_SetItem function, this function would return null if the key is non-hashable, so the return value is checked for null and an appropriate error message is displayed. After the loop, a PyVar is made from the dictionary and then added to the return list.

10. **pySet**

First few lines contain error/syntax checking. An empty PyObject pointer is created, this will contain the set that will be created in the function. This function is overloaded to either take individual arguments to make a set out of, or convert a python iterable like tuple or list into a set. If a python iterable is passed, then it is converted to a set by passing the iterable to the function PySet_New. If an error occurred, then the function would’ve returned null, so the return value is checked for null and an appropriate error message is displayed. If the argument passed is not a python iterable, then an empty set is created using the PySet_New function and then the arguments are looped through. Each argument is converted to a PyVar and its data is added to the set by using the PySet_Add function. If the arguments passed to the function are not hashable, then it would return null, and hence the return value is checked for null and an appropriate error message is displayed. Finally a PyVar is made from the set and then added to the return list.

11. pyEquals

First few lines contain error/syntax checking. Both the passed arguments are converted to PyVars, then they are compared by passing the second object to the first object's __eq__ function, this is done by using the PyObject_CallMethod function. The method would return the true or false on a successful call and a null when the two variables or not of same type, so a scilab True value is added to the return list if the functions returned a Boolean true value otherwise a scilab False value is added to the return list.