Lecture6: Python Scripting

References:

https://docs.python.org/3.8/c-api/index.html

https://docs.python.org/3.8/extending/index.html#extending-index

Getting the dependencies

- Build from source
  - Python does have nice building tools (no cmake; it uses python!)
- Install python (any 3.x version is fine)
  - Install Debug libraries / symbols
  - Copy include and lib folders, then dll's
  - [You can then uninstall if you want]
    - We *don't* have to have python installed to run ssuge
- Two versions of lib / dll's
  - python38[_d].lib – everything in Python3 plus specific 3.8 stuff. I used this.
  - python3[_d].lib – just things common to all 3.x versions.

Part I: Extending

Python **Extending** (in-class exercise)

- Write a dll (renamed as pyd) in C/C++ that exposes certain functions
  - Just release (unless you have a debug interpreter)
- *Uses an installed interpreter*
  - Bit-depth must match (x86 vs. x64)
  - If using the python3.lib, any Python3.x will work
  - If using the python38.lib, must be Python 3.8
- Python loads that dll (pyd) in response to an “import xyz”
- The python user calls those python functions
- E.g. Pygame
- A large part of this task is converting between Python and C types

Python / C API naming convention

- PyXXX_YYY
  - Where XXX is a type
    - Module
    - Import
    - Tuple
    - …
  - and YYY is a function name
    - New
    - SetItem
    - …
- Py_ZZZ
  - Where ZZZ is a “top-level” function

Reference Counting

- How languages like C#, Python, etc. automatically manage memory
- Each dynamically allocated object has a ref-count
  - When the object holding the reference goes out of scope / gets reset, ref-count goes down
  - If another object stores it, the ref-count goes up.
  - If ref-count ever reaches zero, the memory is freed.
  - A “smart pointer” in C++11
- When writing an extension / embedding, you must be keenly aware of it
- The API documentation clearly states when a function creates a new reference
  - Also, “borrowing” and “stealing” a reference.
- Py_INCREF and Py_DECREF are used to manually increase / decrease the ref-count.

Python Ref-counting example

- import stmt loads the random module (or gets it from the dictionary of loaded modules). ref-count on that module++
- the 2 def statements create an executable-type object and stores it away.
- Line 12 allocates a new list w/ ref-count of 1. SubL holds the only reference. 4 and 5 are objects too (w/ refcount of 1). The list referred to by SubL holds them as well.
- Line 13 allocates another list w/ ref-count of 1. The ref-count on the [4, 5] list goes up by one (but 4 and 5's ref-count are unchanged)
- Line 14 runs func. alist is created and ref-count on this bigger list goes up by 1. The 3 object is created w/ ref-count of 1.
- Line 9. Each time through the loop, create_new_val is called. Each time:
  - ref-count on the big list goes up by one.
  - A new integer is created w/ refcount of 1.
  - Line 5 "steals" that reference. The ref-count on the random int is still one, but now the big list owns it.
  - When the function is done, the ref-count on the big list goes down by one (it's still above 0).
- When func ends, the ref-count on the big list goes down by 1 and on the 3 goes down by 1. The 3 is destroyed.
- If we added an L = [] to the end
  - Ref-count on the big list would go down. It reaches zero, so is deleted. The ref-count on all the onwed objects is 0 excepts for the sub-list, which is now 1.
- When the program ends, the ref-count on the random module goes down. If it reaches 0, it is reclaimed (it won't, thought, since python caches imported modules in a dictionary which now has its own reference to the module)

```
1  import random
2
3  def create_new_val(target):
4      x = random.randint(1, 10)
5      target.append(x)
6
7  def func(alist, num):
8      for i in range(num):
9          create_new_val(alist)
10     return len(alist)
11
12 SubL = [4, 5]
13 L = [1, 2, 3, SubL]
14 func(L, 3)
15
```

Look at Python / C api documentation

Steps in Extending

- Make a dll project
  - Incorporate dependencies (as normal)
  - Just need one function that exports one function: PyInit_xyz (where xyz is the module / dll name)
  - That module should create stuff (here: a function and a variable)
  - Make sure the extension type is pyd instead of dll
    - Configuration Properties => Advanced
- Have a PyMethodDef array (next slide) to list all functions we export
- Have a PyModuleDef (next slide) structure to describe the module
- Have a PyInit_xyz function (next slide) to initialize the module
- Create a (normal) python script next to the output pyd that imports our module. Run it.

Steps in Extending, cont.

- Structure of (nearly) all Python functions we expose

  PyObject* my_func(PyObject* self, PyObject* args)

  - All values passed to your function (from Python) are in args (a tuple)
  - You must either:
    - Raise an exception and return NULL if something goes wrong in your function
    - Construct a PyObject that holds the return value
      - A list, string, dictionary, whatever.
      - Or Py_None (don't forget to INCREF it) if it's a "void" type function
- PyMethod def array

```
PyMethodDef my_functions[] = {
    {"func_name1", my_func, METH_VARARGS, "func_name1's docstring"},
    {NULL, NULL, 0, NULL}   // <- "Sentinel" value
};
```

Steps in Extending, cont.

- PyModuleDef structure:

```
struct PyModuleDef my_module = {
    PyModuleDef_HEAD_INIT,          // data to place at beginning
    “name of module”,
    “module docstring”,
    -1,                             // -1 = keep module state in globals
    my_functions                    // (from last slide)
};
```

Steps in Extending, cont.

- The init function

```
PyMODINIT_FUNC PyInit_the_module(void)
{
   PyObject* m = PyModule_Create(&my_module);   // from last slide
   if (m == NULL)
      return NULL;      // will cause python to raise exception
   // Do other stuff to initialize module
   return m;
}
```

(Most) of the functions I used

- PyErr_SetString
- PyExc_ValueError // This is a variable, not a function
- PyLong_FromLong
- PyTuple_Check
- PyTuple_New
- PyTuple_Size
- PyTuple_GetItem
- PyTuple_SetItem
- PyUnicode_Check

Part II: Embedding

Python **Embedding:** Basic idea

- Python embedded interpreter
  - NO EXTERNAL INTERPRETER – it's part of ssuge.exe
  - High flexibility / lower performance [not an issue w/ scripting]
  - Some high-level functions (e.g. `ssuge.set_skybox("whatever")`)
  - [Lab7] Some high-level classes (e.g. ssuge.GameObject)
    - Contains a pointer to a C++ game
    - Has wrappers which will pick out python arguments and call a C++ game object method
    - We'll inherit classes from this (e.g. PlayerShip)
- C++ engine
  - Rendering, physics, etc. [high performance]
  - Links against python.lib / dll files
  - Has a ScriptManager that can execute scripts
  - [Lab7]
    - C++ GameObject will contain a PyObject* to the corresponding python object (if it is "script aware")
    - At key times, we'll call the PYObject's method
      - E.g. GameObject::update will looke for an Update methodof the PyObject*

Main Steps

- Script_functions.h: declarations of all functions we want to expose to Python
  - All will have this form: PyObject* the_func(PyObject* self, PyObject * args)
- Script_functions.cpp: bodies of those functions, plus…the function "table of contents"
- ScriptManager class
  - Py_Initialize(); // starts up our embedded interpreter (ScriptManager::ctor)
    - call PyImport_AppendInitTab before to add a PyInit_whatever function
    - The PyInit_whatever function called PyModule_Create, just as with extending
  - Py_Finalize(); // shuts it down (ScriptManager::dtor)
  - run_script method
    - Opens file, reads contents into char-buffer
    - Calls Py_CompileString, then PyImport_ExecCodeModule to run it
- Application should make a ScriptManager and run a test script (from create_scene)

Error-Handling

- Critical!
- Py_CompileString could error (usually syntax, structural problems w/ script)
- Py_ImportExecCodeMoudle could error (a run-time error, with stack trace)
- Python has tools to generate an error message (in python)

```
import traceback
import sys
# When an error occurs, do this:
   exception_type, exception_value, exception_tb = sys.exc_info()
   output_str = traceback.format_exception(exception_type,
                                           exception_value, exception_tb)
```

- [Develop a function for ScriptManager to do this]