SCRIPTING I/II

EXTENDING PYTHON

References:

- https://docs.python.org/3.4/c-api/
- **https://docs.python.org/3.4/extending/index.html#extending-index**

GOALS

- **Part I:**
  - Create a module (a C dll), callable from python [**extension**]
    - Functions
    - Class (with methods)
  - Illustrate type conversions
  - Introduce **reference counting**
- **Part II:**
  - Embed an interpreter into ssuge.
  - Make a (partial) mirror of our C GameObject in Python.
    - We'll extend (in python scripts) from this (e.g. NinjaObject)
    - The C++ engine will call python methods (e.g. onUpdate) if that game object is "script-aware"

PYTHON C API OVERVIEW

- A *massive* collection of C (not C++) functions
- Naming conventions:
  - Py_XXX: where XXX is a *top-level* function
  - PyYYY_ZZZ: where YYY is a python type and ZZZ is the method.
    - E.g. PyTuple_GetSize
- PyObject: nearly everything uses (and is one of) these
  - primitives: int, floats
  - complex primitives: strings, lists, tuples
  - functions
  - classes (and instances)
  - modules
  - exceptions
  - You have to check the type (PyYYY_Check(PyObject*)).
- A lot of our work involves converting C <-> Python data

STEP 0.

- Download a python distribution (3.6 prefereably) if you don’t have one.
  - Make note: is it 32-bit or 64-bit?
  - Ensure there is an include (.h files) and libs (.lib files) folder
- In VS, create a new Win32 console **dll** project
  - (despite the name, we can make 64-bit console apps too)
  - [side-note about dll projects]
  - copy the dependencies here
  - Remove the Debug configuration
  - Remove the 32-bit or 64-bit configuration
    - Whichever *doesn’t* match your python distro.
  - Add the C++/include and Linker settings necessary to add the python dependencies
  - Change the output of the extension from dll to pyd

CREATING THE MODULE

TESTING THE MODULE

- ensure there is a pyd file being built
- create a new .py file
- run it in your IDE / command-line

```
import testMod

print("testMod.__name__ = " + testMod.__name__)
print("testMod.__doc__ = " + testMod.__doc__)
print(dir(testMod))
```

TANGENT: REFERENCE-COUNTING AND GARBAGE COLLECTION INTRO

```
L = [["a", 4.7], ["b", 1.1]]      # RC on all are 1
L[0] = ["c", 9.2]                 # RC on "a"-list dropped to
                                  #   0 – GC
G = L                             # RC on main list is 2
H = L[1]                          # RC on "b"-list is 2
L = "hi!"                         # RC on main list is 1
G = "bye!"                        # RC on main list is 0 – GC
                                  #   "b"-list has RC of 1
                                  #   so no GC
H = "hasta la vista"              # Now "b"-list has RC
                                  # of 0 - GC
```

ADDING CONSTANTS

- (For example math.pi in the built-in math module)

```
import testMod

print("testMod.mysteryValue = " + str(testMod.mysteryValue))
```

ADDING A FUNCTION (GOAL)

```
# We want to add this functionality to the module:

# … (as before)
T = testMod.buildTuple(6, 0.3, 1)
print(T)        # A tuple of 6 floats in range 0.3 – 1.0
                # I was the last two params to be either
                # floats or int (any number type)

# These should cause an error
# T = testMod.buildTuple(3)    # Too few parameters
# T = testMod.buildTuple(6, 1.4, 1.2)  # max before min
```

ADDING A FUNCTION (IMPLEMENTATION)

PART II: PYTHON/C "CLASSES"

PYTHON OOP BASICS (REVIEW)

```
# We're going to do this in C on later slides (and the labs).
# Just for comparison here.
class Foo(object):
    """ Class-level docstring """
    def __init__(self, name):
        self.mName = name
        self.mVal = 0
        self.mList = []
    def __str__(self):
        return '["' + self.mName + " : " + str(self.mVal) + "]"
    def bar(self):
        self.mVal += 1
        self.mList.append(chr(random.randint(97,122)))
        return self.mVal
x = Foo("Bob")
print(x.bar())          # 1
print(x)                # {"Bob" : 1 : ["q"]}
```

ADDING A CLASS (GOAL)

```
# (Add this to our existing test code)
x = testMod.testClass("Bob")
y = testMod.testClass("Sue")
print(x, y)              # ["Bob" : 0], ["Sue" : 0]
print(x.bar())           # 1
print(x, y)              # ["Bob" : 1], ["Sue" : 0]
for i in range(10):
      y.bar()
print(x, y)               # ["Bob" : 1], ["Sue" : 10]
```

OVERVIEW

- Create 4 additional structures:
  1. [**`testClass`**] A C struct defining the C side of that "object"
     - Can contain one or more basic types (floats, ints, etc.)
     - Can contain PyObject's (e.g. for Python strings, lists, etc.)
  2. [**`testClass_methods`**] An array of PyMethodDefs structures: one element for each non-special method in the class.
  3. [**`testClassType`**] A PyTypeObject: The definition of the new type (in Python).
     - Includes "slots" for all the special functions (e.g. __init__, __add__, etc)
  4. [**`testClass_members`**] An array of PyMemeberDef structures: one for each attribute of the class.
- The rest will be defining the C implementation of all methods.
  - Most will have the signature

    PyObject* func(PyObject * self, PyObject * args)

IMPLEMENTATION

QUESTIONS?