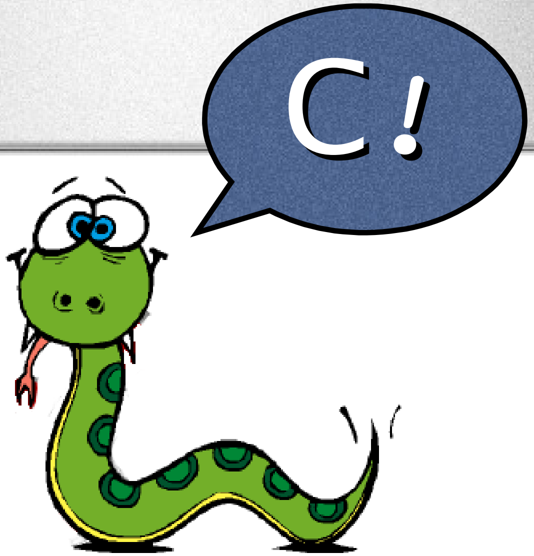


ctypes

extending python was never easier!

Anant Narayanan
Malaviya National Institute of Technology

So what is python?

- Dynamically typed, interpreted language
- Allows for fast prototyping, thanks to the awesome interpreter
- The interpreter revolutionized how programmers attacked problems
- Emphasizes simplicity

How does python work?

- Classic implementation written in the C language, also known as CPython
- Provides an API to communicate between “C-Land” and “Python-Land”
- Standard functions to convert C data types to python types and vice-versa

Python Fundamentals

- Simple set of data types
- string, int/long, float and unicode
- Every function, class or data type is an object
- These objects are, in reality, wrappers over corresponding C types
- Basic python functions implemented in C, higher level functions in python itself

C fundamentals

- Rich set of data types, including the infamous pointers!
    - Basic types: int, char, double, float, w_char_t
    - Arrays: char*, w_char_t*, int*
    - Structures and Unions

The need for bindings

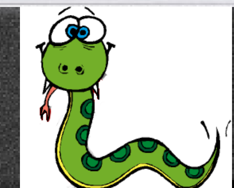
- The python standard library provides a lot of useful functionality
- However, python's success is mainly attributed to the availability of bindings to many popular libraries
- Gtk+, wxWidgets, openGL, SDL, cdrecord, Gnome etc...

The “good old” way

- use functions defined by “Python.h” to export objects:
    - PyObject *obj
    - PyArg_ParseTuple
    - Py_BuildValue
    - Py_INCREF, Py_DECREF
    - PyModule_AddObject, Py_InitModule

The “good old” way

- Good for simple libraries
- Tends to be very monotonous
- pyGTK uses code-generation instead
- Converts function prototypes found in C Header files to scheme-like `def’ files, which are then parsed to generated Python Module code



SWIG: The Next Step

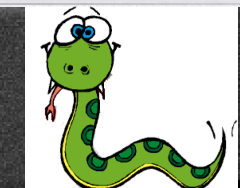
- Abstracts code-generation
- Single “interface” file defines function prototypes, which is then converted to appropriate C binding code
- Not only generates code for python, but PHP, Perl, Ruby and Java too
- Used by Subversion and other major projects

What's wrong with SWIG?

- Need to learn the swig "interface" language
- Produces computer-generated code
    - Ugly to read!
    - Impossible to debug!
- Additional dependency and build routine

Enter ctypes!

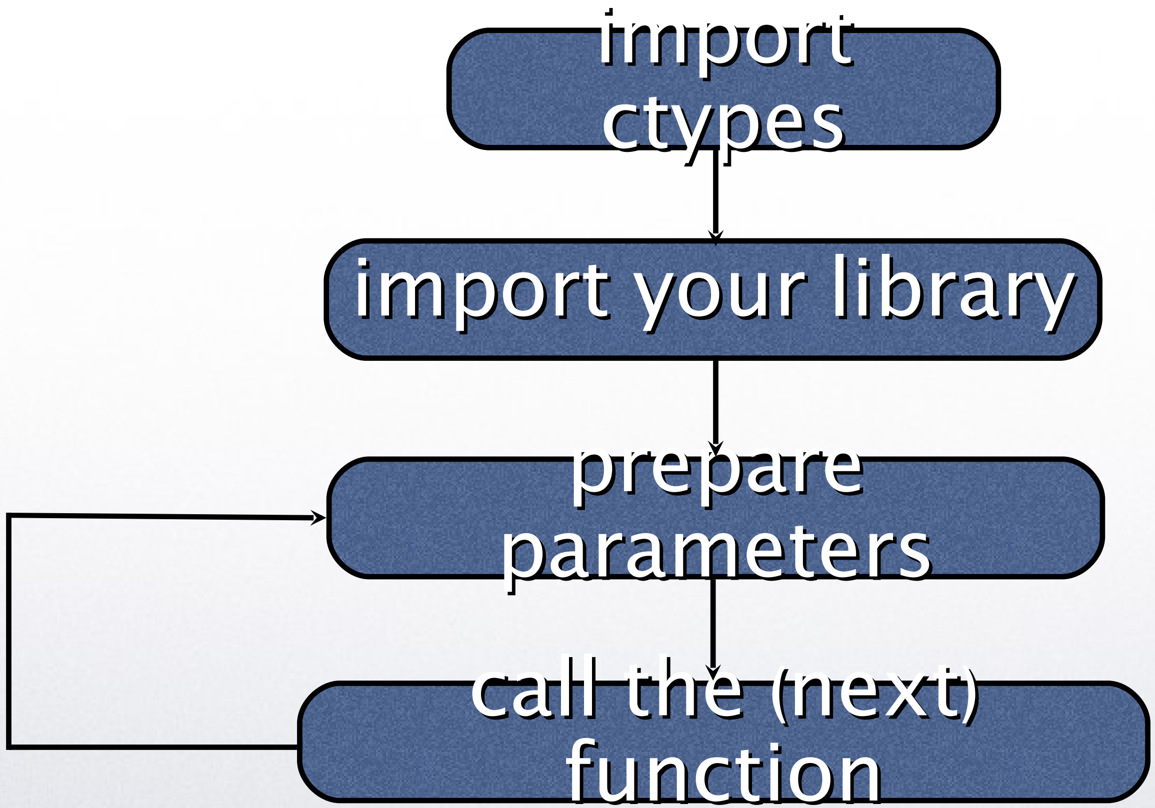
- An even higher layer of abstraction
- Code generation done at memory level
- Allows dynamic execution of functions in a shared library
- .dll, .so and .dylib supported



What does this mean?

- extend python with functions from C libraries - easy bindings written in pure python
- write your callback functions in python and call them from C
- use python as a true “glue” language: interconnecting several libraries to create large, coherent programs

How does it work?



Searching for libraries

- ctypes.util.find_library
  - Runs `ldconfig`, `gcc` and `objdump`
  - Returns filename of the library
  - OS X: Standard paths are looked in; absolute path of dylib returned

Importing libraries

- ctypes.CDLL (name, mode, handle)
  - Represents a loaded shared library
  - Functions in these libraries are called using the standard calling convention of C
  - Functions are assumed to return int
- ctypes.PyDLL (name, mode, handle)
  - Python GIL is not released, hence exceptions are caught and returned. Useful for using the python C API itself!

Importing libraries

- preset library loaders available (just call cdll)
- Or manually load with LibraryLoader and LoadLibrary
- pythonapi represents an instance of PyDLL with the CPython API loaded
- Also available: windll, oledll

Accessing functions

- All functions are exported as attributes of the CDLL/PyDLL class
- Functions are objects of type _FuncPtr
- Called just like regular python callables
- But remember to first convert the parameters!

Type conversion

- int/long, None, strings and unicode objects are automatically converted for you!
- any types other than these must first be converted using the data types provided by ctypes
- python type data type → ctype → C

Some common ctypes

- Mutable
  - c_char, c_wchar, c_byte, c_ubyte
  - c_short, c_ushort
  - c_int, c_uint, c_long, c_ulong, c_float, c_double
- Immutable
  - c_char_p, c_wchar_p, c_void_p

Mutability for strings

- ```
  st = “Hello World!”
  nt = c_char_p(st)
  print nt                    # returns c_char_p(“Hello World!”)
  nt.value = “Bye Bye World!”
  print nt                    # returns c_char_p(“Bye Bye World!”)
  print st        # returns “Hello World!”
  ```

- ## Use
  - create_string_buffer(<bytes>)
  - create_string_buffer(<string>)


  ```

Specifying parameter types

- Set the argtypes attribute for the function object
- This attribute is a sequence of ctypes:
    - myfunc.argtypes = [c_char_p, c_int, c_double]
- Setting this attribute will ensure that function is called with the correct number and types of attributes, and will also convert where possible

Specifying return types

- Set the restype attribute for the function object
- This attribute corresponds to any valid ctype
- Also possible to set it as a python callable if the actual return type is an integer
- In this case, the callable will be invoked with the actual result; and the result of the callable will appear to be

Using Pointers

- Use the quick and easy byref function
- Or construct a proper pointer ctype
  - Value can be accessed by the contents attribute, and also by offset
- Use byref when you don’t need the pointer object later on

Structures & Unions

- All structures are defined as children of the structure base class
- The _fields_ attribute is a list of 2-tuples, containing a field name and a field type
- The field name is any valid python identifier and the field type is any valid ctype.

Structures & Unions

- Similarly Unions are extended from the union base class
- Both are initialized by creating an instance of the class
- Bit fields are also possible
  - Pass the bit-length of the field as the 3rd tuple of each list in the _fields_ attribute

Forward declarations

- The _fields_ attribute can't contain elements that haven't been declared
- However, you can always define or add to the _fields_ attribute later!
- Be careful of using your structure before you have finalized your _fields_, this could lead to inconsistencies

Arrays

- Simply multiply the base element type with a positive integer!
  - ```
    myArray = (c_int * 5)(, 2, 3, 4, 5)
    for i in range(5):
            print myArray[i]
    ```
- ...Or create and object that represent your array and instantiate it
- Arrays are proper ctypes, so you can include them in your structures and other complex types
    ```

Typecasting

- ctypes will automatically accept arrays of a base type, where it was expecting just the base type
- In all other cases: strict type checking!
- Use the cast function to typecast one type to another
  - ```
    obj = (c_byte * 10)()
    castedObj = cast(obj, POINTER(c_int))
    ```



    ```

Callbacks

- Function pointers are of the CFUNCTYPE and can be created by instantiating that class
- The result type is the first argument, followed by the arguments that your callback function must expect
- Connect the function pointer to the actual python callback by instantiating the object returned by CFUNCTYPE

What now?

- Clearly, wrapping libraries in ctypes is far easier, and more maintainable
- Not much performance loss (code generation at runtime)