Section 10

Extending Python with C and C++

Overview

- C/C++ extension modules
- Building extensions by hand
- ctypes library
- Introduction to Swig

Extending Python

- Python can be extended with C/C++
- Many built-in modules are written in C
- Critical for interfacing to 3rd party libraries
- Also common for performance critical tasks

Extension API

- The Python interpreter is written in C
- There is a very specific programming API used to make callouts to C code
- There is also a packaging mechanism for loading shared libraries (DLLs)
- Let's look at the basics

Extension Example

- Suppose you had this C function

```
/* File: gcd.c */

/* Compute the greatest common divisor */
int gcd(int x, int y) {
    int g = y;
    while (x > 0) {
        g = x;
        x = y % x;
        y = g;
    }
    return g;
}
```

- It does not involve any Python

Extension Example

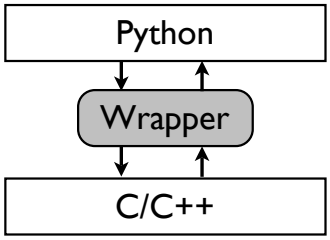
- To access from Python, you must wrap it

```
#include "Python.h"
extern int gcd(int, int);

/* Compute the greatest common divisor */
PyObject* py_gcd(PyObject *self, PyObject *args) {
    int x,y,r;
    if (!PyArg_ParseTuple(args,"ii",&x,&y)) {
       return NULL;
    }
    r = gcd(x,y);
    return Py_BuildValue("i",r);
}
```

- This function sits between C and Python

Wrapper Functions

- The wrapper serves as glue



- It converts values from Python to a low-level representation that C can work with
- It converts results from C back into Python

Extension Example

- Python header files

```
#include "Python.h"
extern int gcd(int, int);

/* Compute the greatest common divisor */
PyObject* py_gcd(PyObject *self, PyObject *args) {
    int x,y,r;
    if (!PyArg_ParseTuple(args,"ii",&x,&y)) {
       return NULL;
    }
    r = gcd(x,y);
    return Py_BuildValue("i",r);
}
```

All extension modules include this header file

Extension Example

- Wrapper function declaration

```
#include "Python.h"
extern int gcd(int, in

/* Compute the greatest common divisor */
PyObject* py_gcd(PyObject *self, PyObject *args) {
    int x,y,r;
    if (!PyArg_ParseTuple(args,"ii",&x,&y)) {
        n NULL;
    }
    x,y);
    y_BuildValue("i
}
```

All wrapper functions have the same C prototype

Return result (Python Object)

Arguments (A tuple)

Extension Example

- Conversion of Python arguments to C

```
#include "Python.h"
extern int gcd(int, int);

/* Compute the greatest common divisor */
PyObject* py_gcd(PyObject *self, PyObject *args) {
    int x,y,r;
    if (!PyArg_ParseTuple(args,"ii",&x,&y)) {
       return NULL;
    }
    r = gcd(x,y);
    return Py_BuildValue("i",r);
}
```

Convert Python arguments to C

PyArg_ParseTuple()

- Format codes are used for conversions

Format	Python Type	C Datatype
"s"	String	char *
"s#"	String with length	char *, int
"c"	String	char
"b"	Integer	char
"B"	Integer	unsigned char
"h"	Integer	short
"H"	Integer	unsigned short
"i"	Integer	int
"I"	Integer	unsigned int
"l"	Integer	long
"k"	Integer	unsigned long
"f"	Float	float
"d"	Float	double
"O"	Any object	PyObject *

PyArg_ParseTuple()

- Must pass the address of C variables into which the result of conversions are placed
- Example:

```
int      x;
double   y;
char     *s;

if (!PyArg_ParseTuple(args,"ids",&x,&y,&s)) {
   return NULL;
}
```

Extension Example

- Calling the C function

```
#include "Python.h"
extern int gcd(int, int);

/* Compute the greatest common divisor */
PyObject* py_gcd(PyObject *self, PyObject *args) {
    int x,y,r;
    if (!PyArg_ParseTuple(args,"ii",&x,&y)) {
       return NULL;
    }
    r = gcd(x,y);
    return Py_BuildValue("i
}
```

Call the real C function

Extension Example

- Creating a return result

```
#include "Python.h"
extern int gcd(int, int);

/* Compute the greatest common divisor */
PyObject* py_gcd(PyObject *self, PyObject *args) {
    int x,y,r;
    if (!PyArg_ParseTuple(args,"ii",&x,&y)) {
       return NULL;
    }
    r = gcd(x,y);
    return Py_BuildValue("i",r);
}
```

Create a Python Object with Result

Py_BuildValue()

- This function also relies on format codes

Format	Python Type	C Datatype
"s"	String	char *
"s#"	String with length	char *, int
"c"	String	char
"b"	Integer	char
"h"	Integer	short
"i"	Integer	int
"l"	Integer	long
"f"	Float	float
"d"	Float	double
"O"	Any object	PyObject *
"(*items*)"	Tuple	*format*
"[*items*]"	List	*format*
"{*items*}"	Dictionary	*format*

Py_BuildValue()

- Examples:

```
Py_BuildValue("")             // None
Py_BuildValue("i",37)         // 37
Py_BuildValue("d",3.14159)    // 3.14159
Py_BuildValue("s","Hello")    // 'Hello'

Py_BuildValue("(ii)",37,42)   // (37,42)
Py_BuildValue("[ii]",37,42)   // [37,42]
Py_BuildValue("{s:i,s:i}",    // {'x':37,'y':42}
              "x",37,"y",42)
```

- Last few examples show how to easily create tuples, lists, and dictionaries

Extension Example

- Once wrappers are written, you must tell Python about the functions
- Define a "method table" and init function

```
/* Method table for extension module */
static PyMethodDef extmethods[] = {
   {"gcd", py_gcd, METH_VARARGS},
   {NULL,NULL}
}

/* initialization function */
void initext() {
   Py_InitModule("ext",extmethods);
}
```


Extension Example

- The method table lists all of the names and and wrappers in the extension module
- Might contain a large number of entries

```
/* Method table for extension module */
static PyMethodDef extmethods[] = {
   {"gcd", py_gcd, METH_VARARGS},
   {NULL,NULL}
}
```

List the wrapper functions here

Name used in Python

The C wrapper

Flags

```
gcd() {
   Py_InitModule("ext",extmethods);
}
```

Extension Example

- Init function gets executed when the module is dynamically loaded (via import)
- Purpose is to initialize the module contents

```
/* Method table for extension module */
static PyMethodDef extmethods[] = {
   {"gcd", py_gcd, METH_VARARGS},
   {NULL,NULL}
}

/* initialization function */
void initext() {
   Py_InitModule("ext",extmethods);
}
```

Module initializer

Creates the module and populates with methods

Extension Example

- The naming of the init function is critical
- Python looks for a specific C function based on the name of the module

```
/* Method table for extension module */
static PyMethodDef extmethods[] = {
   {"gcd", py_gcd, METH_VARARGS},
   {NULL,NULL}
}

/* initialization function */
void initext() {
   Py_InitModule("ext",extmethods);
}
```

These names must match

Extension Compilation

- Compiling an extension module
- There are usually two sets of files

```
gcd.c              # Original C code
pyext.c            # Python wrappers
```

- These are compiled together into a shared lib
- Use of distutils is "recommended"

Extension Compilation

- Create a setup.py file

```
# setup.py
from distutils.core import setup, Extension

setup(name="ext",
      ext_modules=[Extension("ext",
                             ["gcd.c","pyext.c"])]
      )
```

- To build and test

```
% python setup.py build_ext --inplace
```

Extension Compilation

- Sample output of compiling

```
% python setup.py build_ext --inplace
running build_ext
building 'ext' extension
creating build
creating build/temp.macosx-10.3-fat-2.5
gcc ... -c gcd.c -o build/temp.macosx-10.3-fat-2.5/gcd.o
gcc ... -c pygcd.c -o build/temp.macosx-10.3-fat-2.5/pyext.o
gcc ... build/temp.macosx-10.3-fat-2.5/gcd.o build/
temp.macosx-10.3-fat-2.5/pyext.o -o ext.so
%
```

- Creates a shared library file (ext.so)

Extension Compilation

- Manual compilation

```
% cc -c -I/usr/local/include/python2.6 pyext.c
% cc -c gcd.c
% cc -shared pyext.o gcd.o -o ext.so
%
```

- This will vary depending on what system you're on, compiler used, installation location of Python, etc.

Extension Import

- To use the module, just use import

```
% python
>>> import ext
>>> ext.gcd(42,20)
2
>>>
```

- import loads the shared library and populates a module with extension functions
- If all goes well, it will just "work"

A Word on Filenames

- Expected file suffix on extension libraries
  - .so on Unix systems
  - .pyd on Windows
- Common mistake: using a bad filename

```
>>> import ext
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named ext
>>>
```

Commentary

- There are many steps
- Must have a C/C++ compiler
- Must be able to create DLLs/shared libs
- In my experience, compilation/linking is the most difficult step to figure out

More Information

- "Extending and Embedding the Python Interpreter", by Guido van Rossum

```
http://docs.python.org/ext/ext.html
```

- These is the official documentation on how the interpreter gets extended
- Look there for gory low-level details

Exercise 10.1

ctypes

- An alternative approach to accessing C code
- A module that allows C functions to be executed in arbitrary shared libraries/DLLs
- Instead of writing wrappers, you do everything entirely in Python

ctypes Example

- Consider this C code:

```
int fact(int n) {
   if (n <= 0) return 1;
   return n*fact(n-1);
}

int cmp(char *s, char *t) {
   return strcmp(s,t);
}
double half(double x) {
   return 0.5*x;
}
```

- Suppose it was compiled into a shared lib

```
% cc -shared example.c -o libexample.so
```


ctypes Example

- Using C types

```
>>> import ctypes
>>> ex = ctypes.cdll.LoadLibrary("./libexample.so")
>>> ex.fact(4)
24
>>> ex.cmp("Hello","World")
-1
>>> ex.cmp("Foo","Foo")
0
>>>
```

- It just works (heavy FFI wizardry)

ctypes Caution

- C libraries don't contain type information
- So, ctypes has to guess...

```
>>> import ctypes
>>> ex = ctypes.cdll.LoadLibrary("./libexample.so")
>>> ex.fact("Howdy")
1
>>> ex.half(5)
-1079032536
>>> ex.cmp(4,5)
Segmentation Fault
```

- And unfortunately, it usually gets it wrong
- However, you can help it out.

ctypes Types

- You can add type signatures after loading

```
>>> ex.half.argtypes = (ctypes.c_double,)
>>> ex.half.restype = ctypes.c_double
>>> ex.half(5.0)
2.5
>>>
```

- Creates a minimal prototype

```
.argtypes      # Tuple of argument types
.restype       # Return type of a function
```

- Using this, you can create an extension module that works "properly"

Using ctypes

- To use ctypes, three main steps are involved
- Step 1: Create a .py file for your module

```
# example.py
#
# An extension module for the libexample.so library
```

- Step 2: Load the associated shared library

```
# example.py
...
import ctypes
ex = ctype.cdll.LoadLibrary("libexample.so")
```


Using ctypes

- Step 3: Extract symbols and add signatures

```
# example.py
...
# int fact(int)
fact = ex.fact
fact.argtypes = (ctypes.c_int,)
fact.restype  = ctypes.c_int

# int cmp(char *s, char *t)
cmp = ex.cmp
cmp.argtypes = (ctypes.c_char_p, ctypes.c_char_p)
cmp.restype = ctypes.c_int
```

- Continue until you have finished

Using ctypes

- Using your module: import normally

```
>>> import example
>>> example.fact(4)
24
>>> example.half(5)
2.5
>>>
```

- If you have done it correctly, the end user should not be aware of ctypes
- ctypes is a hidden implementation detail

Type Signatures

- To use ctypes effectively, you have to map C type signatures to ctypes signatures
- This is straightforward, but you need to know the names of the primitive types

Primitive C Datatypes

- C integer types

ctypes type	C Datatype
c_bool	bool
c_byte, c_int8	signed char
c_ubyte, c_uint8	unsigned char
c_short, c_int16	short
c_ushort, c_uint16	unsigned short
c_int, c_int32	int
c_uint, c_uint32	unsigned int
c_long, c_int64	long
c_ulong, c_uint64	unsigned long
c_longlong	long long
c_ulonglong	unsigned long long
c_size_t	size_t

- Note: May vary by platform and 32 vs 64-bit (depends on how Python was compiled)

Primitive C Datatypes

- C floating point types

ctypes type	C Datatype
c_float	float
c_double	double
c_longdouble	long double

- C string and character types

c_char	char
c_char_p	char *
c_wchar	wchar
c_wchar_p	wchar *

- Generic pointer

c_void_p	void *

Practicalities

- Effective use of ctypes requires expert skills
- Must know everything about the C library
  - Function names
  - Type signatures
  - Data structures
  - Memory management model
  - Side effects and semantics
- If you're not sure, you will blow your leg off

ctypes and C++

- Not supported at all
- This is the fault of C++
- C++ "libraries" aren't designed to integrate with any other environment other than C++ (and not even different C++ compilers)
- However, you could put a C wrapper around C++ and access it through that interface

Commentary

- ctypes is something you should know about
- It can access C libraries entirely from Python without requiring a C compiler (might simplify deployment of an application)
- A good choice for applications that only involve a small amount of C extension code (applications that are mostly Python except for a few performance critical operations)

Exercise 10.2

Swig

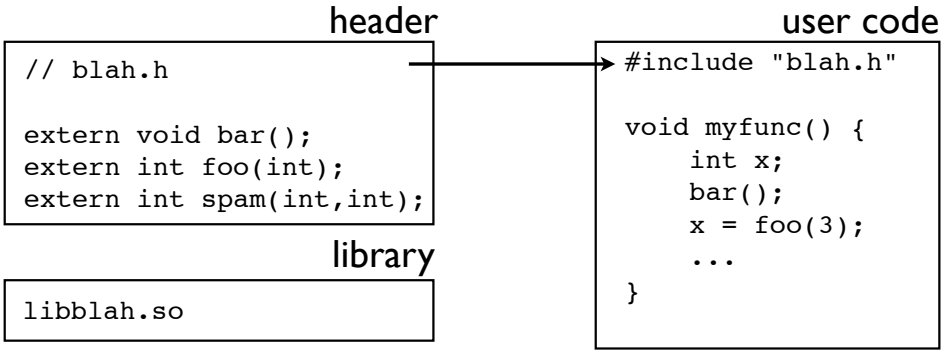
- http://www.swig.org
- A C/C++ code generator that creates extension modules for C libraries
- Creates code similar to what would be written in a hand-written extension module
- People use it to access large libraries and frameworks from Python (hundreds of functions, structures, classes, etc.)

Disclaimers

- I am the original creator of Swig
- It is not the only solution to this problem
- It is a large package that dates to 1996
- It has a plethora of advanced features that you can use to injure yourself and others if you don't know what's going on

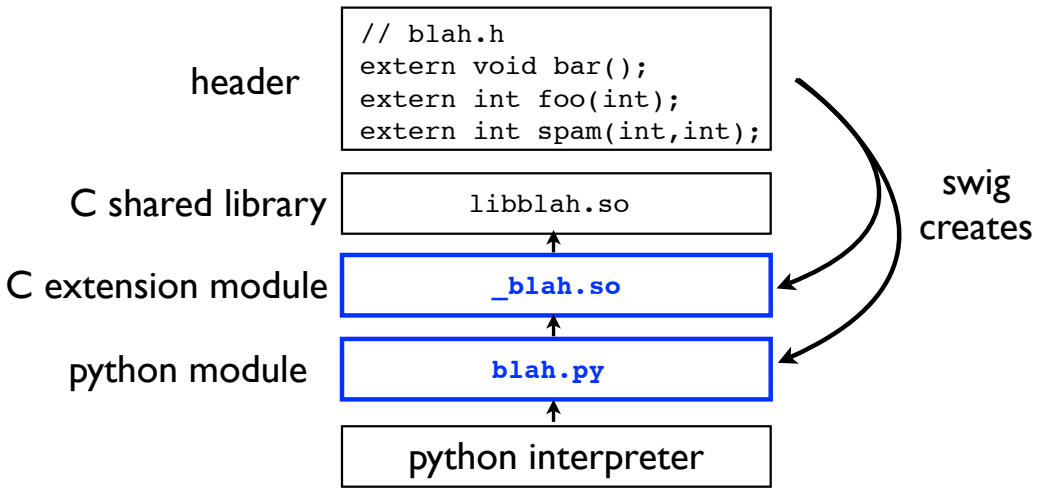
C Header Files

- Contain declarations of program components defined in separate object files or libraries
- Every programming library has header files in order to support separate compilation



Swig Big Picture

- Swig takes declarations from header files and uses them to build a glue layer for binding shared libraries to the Python interpreter



Swig Big Picture

- End-goal is to seamlessly import the shared library as a Python library module

```
>>> import blah
>>> blah.bar()
>>> blah.foo(3)
37
>>>
```

- Operations call C functions in the library
- End-user shouldn't care (the use of C is only a low-level implementation detail)

Interface Files

- To use Swig, you write an interface file

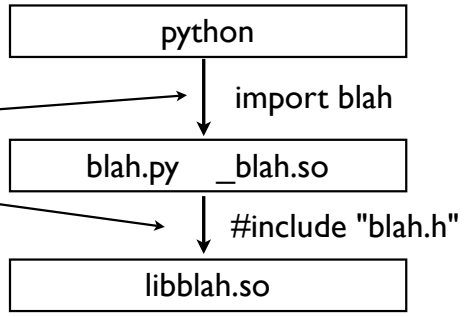
```
// blah.i
%module blah
%{
#include "blah.h"
%}

// Declarations (explained shortly)
...
```

- Minimal requirements:
  - Specify the module name (%module)
  - Include C header files (%{ ... %})

Interface Preamble

- The preamble at the top is required to glue the parts together and for compilation

```
// blah.i
%module blah
%{
#include "blah.h"
%}
...
```



- All Swig interface specifications start like this (if not, you're not using it correctly)

Interface Declarations

- The remainder of the interface specification contains declarations (usually from the header)

```
// blah.i
%module blah
%{
#include "blah.h"
%}

// Declarations
void bar();
int foo(int);
int spam(int, int);
```

List C/C++ declarations here

- The declarations section explicitly lists all functionality to be exposed to Python

Running Swig

- Swig is a compiler
- You run it as a command-line tool

```
% swig -python blah.i
%
```

- Two output files are created

```
blah.py           # Python module code
blah_wrap.c       # C extension code
```

- The C extension code must be compiled and linked by the C compiler

Compilation

- Swig extensions can be compiled the same way as hand-written Python extensions
- Use distutils : Here's a rough template

```
# setup.py
from distutils.core import setup, Extension

setup(name="blah",
      py_modules = ['blah.py'],
      ext_modules=[
          Extension("_blah",
                    ["blah_wrap.c"],
                    include_dirs = [],
                    library_dirs = ['.'],
                    libraries = ['blah']
                   )
      ])
```

Compilation

- Compilation with distutils

```
% python setup.py build_ext --inplace
```

- Hand-compilation (platform specific)

```
% cc -shared -I/usr/local/include/python2.6 \
  blah_wrap.c -L. -Xlinker -rpath . -lblah \
  -o _blah.so
%
```

- You should now have two files

```
blah.py         # Python module
_blah.so        # Compiled extension module
```

- These two files are always going to be paired

Module Import

- Using your module : It should just "work"

```
>>> import blah
>>> blah.foo(2)
37
>>> blah.bar()
>>>
```

- This is one of the most difficult parts
- Complexity comes from the complicated build environment (C, shared libraries, compiler options, linking, etc.)

Swig and C

- Swig supports virtually all of ANSI C
- Functions, variables, and constants
- All ANSI C datatypes
- Pointers and arrays
- Structures and Unions

C++ Wrapping

- Swig supports most of C++
- Classes and inheritance
- Overloaded functions/methods
- Operator overloading (with care)
- Templates
- Namespaces
- Not supported: Nested classes

Example: C++ Classes

- A sample C++ class

```
%module blah
...
class Foo {
public:
   int bar(int x, int y);
   int member;
   static int spam(char *c);
  };
```

- It works like a Python class

```
>>> import blah
>>> f = blah.Foo()
>>> f.bar(4,5)
9
>>> f.member = 45
>>> blah.Foo.spam("hello")
```


Exercise 10.3