Extending Python with C and C++

Overview

- A look at how Python is extended with code written in C and C++
- Building extensions by hand
- Extension building tools (Swig)
- Extension library module (ctypes)
- Some practicalities

Disclaimer

- This is an advanced topic
- Will cover some essentials, but you will have to consult a reference for hairy stuff
- We could do an entire course on this topic
- My main goal is to give you a survey

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 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;
}
```

Extension Example

- To access from Python, you write a wrapper 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",r);
}
```

- Sits between C and 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)) {
           rn 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

- 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 exrmethods[] = {
   {"gcd", py_gcd, METH_VARARGS},
   {NULL,NULL}
}
        li                             */
      gcd() {
   Py_InitModule("ext",extmethods);
}
```

List the wrapper functions here

Name used in Python

The C wrapper

Flags

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);
}
```

Module initializer

Creates the module and populates with methods

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 gcdmethods[] = {
   {"gcd", py_gcd, METH_VARARGS},
   {NULL,NULL}
}

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

These names must match

Extension Example

- 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 Example

- 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 Example

- 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 Example

- Manual compilation

```
% cc -c -I/usr/local/include/python2.5 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 Example

- To use the module, just run python

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

- import loads the shared library and adds extension functions
- If all goes well, it will just "work"

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 here for gory low-level details

Interlude

- Programming extensions by hand is possible, but extremely tedious and error prone
- Most Python programmers use extension building tools and code generators to do it
- Examples: Swig, Boost.Python, ctypes, SIP, etc.

Swig

- http://www.swig.org
- A special C/C++ compiler that automatically creates extension modules
- Parses C/C++ declarations in header files
- Generates all of the wrapper code needed

Disclaimer

- I am the original creator of Swig
- It is <u>not</u> the only solution to this problem
- I don't know if it is any better or worse than other tools
- Your mileage might vary

A Swig Example

- Wrapping a C function
- First you create a Swig interface file

```
// ext.i
%module ext                          Module name
%{
extern int gcd(int,int);             externals
%}

int gcd(int,int);                    declarations
```

- Contains module name, external definitions, and a list of declarations

A Swig Example

- Manually running Swig

```
% swig -python ext.i
%
```

- This creates two files

```
% ls
gcd.c   ext.i  ext.py  ext_wrap.c
%
```

- ext_wrap.c - A set of C wrappers
- ext.py - A set of high-level Python wrappers

A Swig Example

- To compile, create a distutils setup.py file

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

setup(name="ext",
      py_modules=['ext']
      ext_modules=[Extension("_ext",
                   ["gcd.c","ext.i"])]
      )
```

- Contains original source, Swig-related files
- Note: distutils already knows how to run Swig

A Swig Example

- Run setup.py

```
% python setup.py build_ext --inplace
running build_ext
building '_ext' extension
swigging ext.i to ext_wrap.c
swig -python -o ext_wrap.c ext.i
creating build/temp.macosx-10.3-fat-2.5
gcc ... -c gcd_wrap.c -o build/temp.macosx-10.3-fat-2.5/
ext_wrap.o
gcc ... -c gcd.c -o build/temp.macosx-10.3-fat-2.5/gcd.o
gcc ... build/temp.macosx-10.3-fat-2.5/gcd_wrap.o build/
temp.macosx-10.3-fat-2.5/gcd.o -o build/lib.macosx-10.3-
fat-2.5/_ext.so
%
```

- Creates a module ext.py and an extension module _ext.so

A Swig Example

- To use the module, run Python

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

Swig Usage

- Tools such as Swig are especially appropriate when working with more complex C/C++
- Automated tools know how to create wrappers for structures, classes, and other program constructs that would be difficult to handle in hand-written extensions

Swig and C

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

Example: Structures

- Structures are wrapped by Python classes

```
%module example
...
struct Vector {
    double x,y,z;
};
```

- Example:

```
>>> import example
>>> v = example.Vector()
>>> v
<example.Vector; proxy of <Swig Object of type 'Vector
*' at 0x60e970> >
>>> v.x = 3.4
>>> v.y = 2.0
>>> print v.x
3.4
>>>
```

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

- It gets wrapped into a Python proxy class

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

Example: Overloading

- Supported for the most part

```
%module example
...
void foo(int x);
void foo(double x);
void foo(char *x, int n);
```

- Example:

```
>>> import example
>>> example.foo(4)
>>> example.foo(4.5)
>>> example.foo("Hello",5)
```

- However, certain corner cases don't work

```
void foo(double x);
void foo(float x);
```

Swig Wrap-Up

- Swig is a very widely used extension tool
- Primary audience is programmers who want to use Python as a control language for large libraries of C/C++ code
- Example: Using Python to control software involving 300 C++ classes

ctypes

- ctypes module is new in Python2.5
- A library module that allows C functions to be executed in arbitrary shared libraries/DLLs
- Does not involve writing any C wrapper code or using a tool like Swig

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 wizardry)

ctypes Example

- Well, it *almost* works:

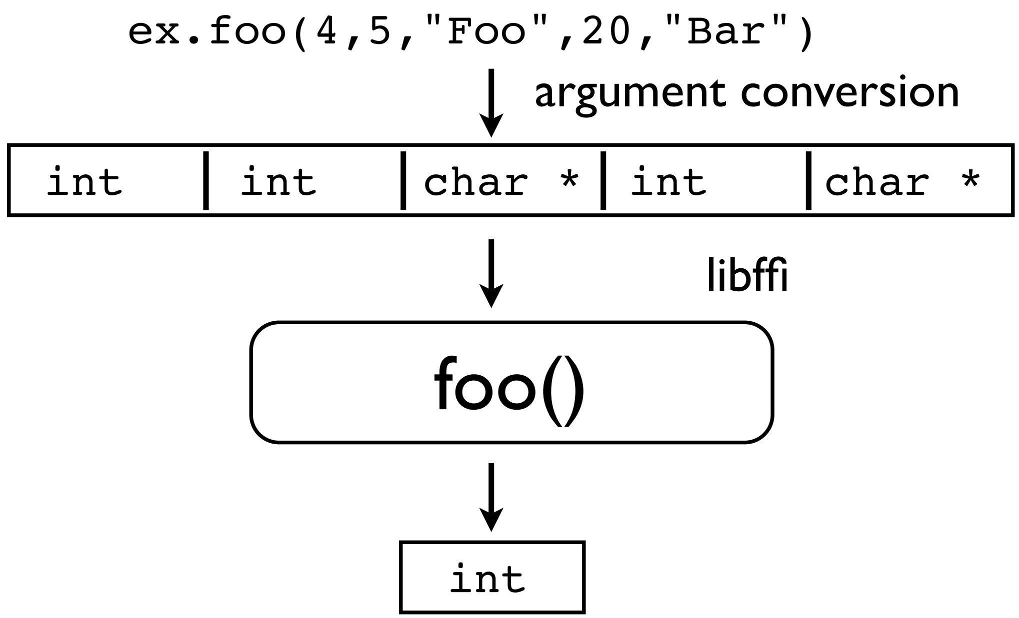
```
>>> import ctypes
>>> ex = ctypes.cdll.LoadLibrary("./libexample.so")
>>> ex.fact("Howdy")
1
>>> ex.cmp(4,5)
Segmentation Fault

>>> ex.half(5)
-1079032536
>>> ex.half(5.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <type
'exceptions.TypeError'>: Don't know how to convert
parameter 1
>>>
```

ctypes Internals

- ctypes is a module that implements a foreign function interface (FFI)
- Only has limited knowledge of C by itself
- By default, assumes all parameters are either integers or pointers (ints, strings)
- Assumes all functions return integers
- Performs no type checking (unless more information is known)

ctypes Internals

- A high level view:



- Relies on low-level details of C (native word size, int/pointer compatibility, etc.)

ctypes Types

- ctypes *can* handle other C datatypes
- You have to provide more information

```
>>> 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
```

ctypes Types

- Sampling of datatypes available

```
ctypes type        C Datatype
------------------ ---------------------------
c_byte             signed char
c_char             char
c_char_p           char *
c_double           double
c_float            float
c_int              int
c_long             long
c_longlong         long long
c_short            short
c_uint             unsigned int
c_ulong            unsigned long
c_ushort           unsigned short
c_void_p           void *
c_py_object        PyObject *
```

ctypes Limitations

- Requires detailed knowledge of underlying C library and how it operates
- Function names
- Argument types and return types
- Data structures
- Side effects/Semantics
- Memory management

ctypes and C++

- Not really supported
- This is more the fault of C++
- C++ creates libraries that aren't easy to work with (non-portable name mangling, vtables, etc.)
- C++ programs may use features not easily mapped to ctypes (e.g., templates, operator overloading, smart pointers, RTTI, etc.)

ctypes Summary

- A very cool module for simple C libraries
- Works on almost every platform (Windows, Linux, Mac OS-X, etc.)
- Great for quick access to a foreign function
- Actively being developed---there are even Swig-like tools for it
- Part of standard Python distribution

Practicalities

- Extension programming is hairy
- Want to discuss some general issues
  - Searching
  - Stealing
  - Performance tuning
  - Shared libraries and dynamic loading
  - Debugging
  - Tools

Search the Web

- Check to see if someone has already done it
- Most popular C libraries already have Python interfaces
- Don't re-invent the wheel
- Python Package Index

http://cheeseshop.python.org/pypi

Stealing

- If you must write an extension module, steal as much code as possible
- Best place to look: Python source code

```
Python/Modules        # Built-in library modules
Python/Objects        # Built-in types
```

- Find a built-in module that behaves most like the extension you're trying to build
- Tweak it

Performance Tuning

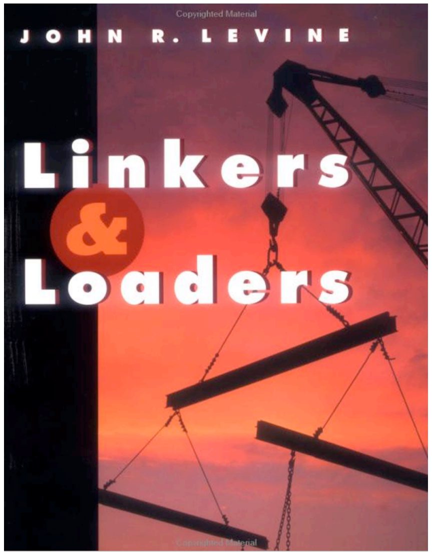
- Some programmers turn to extension modules for performance
- If performance is a problem, look for a better algorithm first
- An efficient algorithm in Python may beat an inefficient algorithm in C
- Consider optimizations of Python code

Shared Libraries

- All Python extensions are compiled as shared libraries/dynamically loadable modules
- DLLs
- Sadly, very few C/C++ programmers actually understand what's going on with shared libraries
- And even fewer understand dynamic loading

A General Reference

- Recommended reading:
- J. Levine, "Linkers and Loaders"
- A good overview of basic principles related to libraries, dynamic linking, dynamic loading, etc.



- Sadly, beyond the scope of what I cover here

Debugging

- Extension modules may crash Python

```
Access Violation
Segmentation Fault
Bus Error
Abort (failed assertion)
```

- Python debugger (pdb) is useless here
- Most common culprit: Memory/pointers
- To debug: Run a C/C++ debugger on the Python interpreter itself

Summary

- Python allows modules to be written in C/C++
- There is a documented programming API
- There are many tools that can simplify matters
- There are many subtle issues (e.g., debugging)
- I've only covered the tip of the iceberg.