Swig Master Class

David "Mr. Swig" Beazley
http://www.dabeaz.com

Presented at PyCon'2008

An Introduction

- Perhaps you've heard about this little Python-extension building tool called "Swig" and wondered what it was all about
- Maybe you used it on some simple code and were a little "surprised" that it worked.
- Maybe you tried to use it for something more complicated and were a) Overwhelmed, b) Confused, c) Horrified, or d) All of the above.

Swig Essentials

- The official website:

  http://www.swig.org

- The short history
  - First implementation (summer, 1995)
  - First release (Feb. 1996)
  - First paper (June 1996, Python Workshop)
  - And now thousands of users...

About Me

- I am the original creator of Swig
- I wanted to make it easy for scientists to put scripting interfaces on physics software
- Have since worked on a variety of other projects (parsing tools, debuggers, etc.)
- I am still involved with Swig, but am coming off of a bit of sabbatical from working on it.

About Swig

- Swig has been around for many years and has been actively maintained by a large group of developers
- However, its complexity has grown along with its capabilities
- Although it is still easy to use on "simple" projects, more advanced tasks can (regrettably) involve a rather steep learning curve.

About this Class

- How Swig is put together
- How Swig thinks about extension building
- How Swig code generation works
- How various customization features work

Disclaimers

- This is an advanced course
- I will assume the following:
  - You can write Python programs
  - You can write C/C++ programs
  - You have made extensions to Python before (by hand or with a tool)

Disclaimers

- This class is about important concepts and the "big picture."
- This isn't a Swig reference manual or even a detailed tutorial for beginners.
- But, the Swig reference manual will (hopefully) make a lot more sense after this class

No Advocacy!

- This is not a Swig sales pitch
- It's a look inside to see how Swig works
- Mostly, I want to demystify parts of it

Format

- Course is organized as a discussion with some live examples/demos
- You can follow along, but you will need Python, Swig, C/C++ installed on your system
- I'm not going to talk about how to configure your environment.
- Please stop me to ask questions!

Part I

Python Extension Building and Swig

Python Extensions

- Python can be extended with C functions

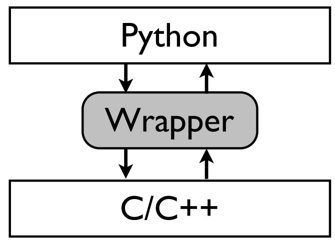
```
/* A simple C function */
double square(double x) {
    return x*x;
}
```

- To do this, you have to write a wrapper

```
PyObject *py_square(PyObject *self, PyObject *args) {
    double x, result;
    if (!PyArg_ParseTuple(self,"d",&x)) {
        return NULL;
    }
    result = square(x);
    return Py_BuildValue("d",result);
}
```

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 Modules

- An extension module is just a collection of wrapper functions
- Additional initialization code sets it up

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

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


Packaging of Extensions

- Extension modules usually compiled into shared libraries or DLLs (ext.so, ext.pyd, etc.)
- The import statement knows to look for such files (along with .py, .pyc, and .pyo files)

```
>>> import ext
>>> ext.square(4)
16.0
>>>
```

- There are many details related to the compilation of these modules (but, that's an entirely different tutorial)

The Problem

- Writing extension code by hand is annoying
- Extremely tedious and error prone
- Difficult to maintain
- Not at all obvious when you start getting into gnarly C/C++ code (structs, classes, arrays, pointers, templates, etc.)

Extension Tools

- Python has a large number of tools that aim to "simplify" the extension building process
  - Boost.Python
  - ctypes
  - SIP
  - pyfort
  - Pyrex
  - Swig
- Apologies to anyone I left out

Swig

- Swig generates wrappers from C++ headers
- Basic idea : You just list everything you want in your extension module using normal C-style declarations
- Swig parses those declarations and creates an output file of C/C++ code which you compile to make your module

Sample Swig Interface

- Here is a sample Swig specification:

```
%module sample
%{
#include "myheader.h"
#include "otherheader.h"
%}

#define PI 3.14159;
int    foo(int x, int y);
double bar(const char *s);

struct Spam {
    int a, b;
};
```


Sample Swig Interface

- Here is a sample Swig specification:

```
%module sample
%{
#include "myheader.h"
#include "otherheader.h"
%}

#define PI 3.14159;
int    foo(int x, int y);
double bar(const char *s);

struct Spam {
    int a, b;
};
```

Preamble.

Gives the module name and provides declarations needed to get the code to compile (usually header files).

Sample Swig Interface

- Here is a sample Swig specification:

```
%module sample
%{
#include "myheader.h"
#include "otherheader.h"
%}

#define PI 3.14159;
int    foo(int x, int y);
double bar(const char *s);

struct Spam {
    int a, b;
};
```

Declarations.

List everything that you want in the extension module.

Running Swig

- Swig is a command-line tool

```
shell % swig -python sample.i
shell %
```

- Unless there are errors, it is silent
- Invocation of Swig may be hidden away.
- For instance, distutils/setuptools runs Swig automatically if you list a .i file as a source.

Swig Output

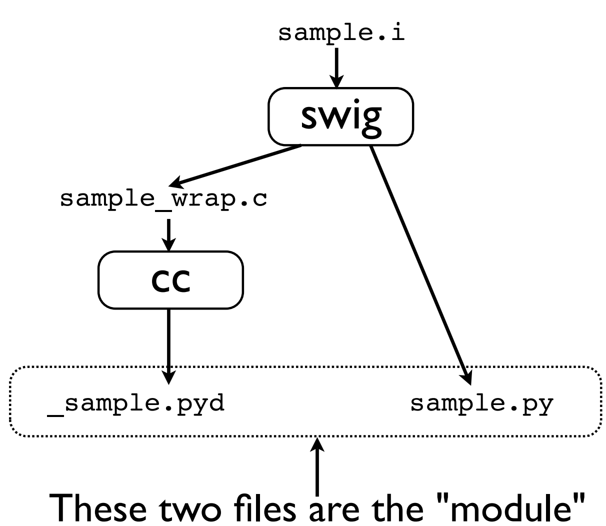
- As output, Swig produces two files

```
shell % ls
sample.i sample_wrap.c sample.py
shell %
```

- The _wrap.c file is C code that must be compiled in a shared library
- The .py file is Python support code that serves as a front-end to the low-level C module
- Users import the .py file

Building Swig Extensions

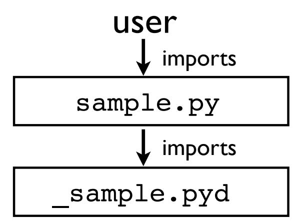
- Swig extension modules always come in pairs



These two files are the "module"

Dual-Module Architecture

- Swig uses a dual-module architecture where some code is in C and other code is in Python



- This same approach is used by Python itself (socket.py, _socket.pyd, thread.pyd, threading.py)

Using a Swig Module

- Usually no big surprises

```
>>> import sample
>>> sample.foo(73,37)
42
>>> sample.PI
3.1415926
>>> x = sample.bar("123.45")
>>> s = sample.Spam()
>>> s.a = 1
>>> s.b = 2
>>> print s.a + s.b
3
>>>
```

- Everything in the declaration list is available and "works" as you would expect

General Philosophy

- The main goal of Swig is to make a “natural” interface to C/C++ code

```
%module sample

class Spam {
public:
    int a,b;
    int bar(int);
    static foo(void);
};
```

```
>>> import sample
>>> s = sample.Spam()
>>> s.a = 42
>>> x = s.bar(37)
>>> sample.Spam.foo()
>>>
```

- A very large subset of C/C++ is supported
- Use in Python is the same as use in C++

Swig-generated Code

- Swig generates the same kind of code that you would normally write by hand
- It creates wrapper functions
- It creates a module initialization function
- It packages everything up into a file that you can compile into a extension module

The Horror, The Horror

- When people first come to Swig, they might look at the output files and have their head explode.
- That code is not meant to be read.
- However, there are a number of critical things going on in the output...

Compiler Dependencies

- Output includes a lot of compiler switches, macros, and other definitions for portability

```
#ifndef SWIGEXPORT
# if defined(_WIN32) || defined(__WIN32__) || defined(__CYGWIN__)
#   if defined(STATIC_LINKED)
#     define SWIGEXPORT
#   else
#     define SWIGEXPORT __declspec(dllexport)
#   endif
# else
#   if defined(__GNUC__) && defined(GCC_HASCLASSVISIBILITY)
#     define SWIGEXPORT __attribute__ ((visibility("default")))
#   else
#     define SWIGEXPORT
#   endif
# endif
#endif
```

Runtime Support

- The wrapper code also includes a runtime library of about 3000 lines of code
- Library functions, macros, etc.
- This is needed to deal with more complex aspects of extension modules (especially C++)
- A critical part of how modules are packaged

Self-Containment

- Many people don't realize that the output of Swig is <u>identical on all platforms</u>
- The wrapper code has no third-party dependencies and does not rely on any part of a Swig installation (headers or libraries)
- Code generated by Swig can be distributed independently from Swig
- End users don't need Swig installed

Part 2

Extension Building and Type Systems

The Extension Problem

- The problem of building extension modules is not new---people have been doing this from the beginning days of Python.
- What is the true nature of this problem?
- Is it simply a code generation problem?
- Is it some kind of text "parsing" problem?

Concept : Types

- Programming languages operate on different kinds of data.
- Data has a “type” associated with it

```
/* C++ */                  # Python

int    a;                  a = 37
double b;                  b = 3.14159
char   *c;                 c = "Hello"
```

- In C, variables have explicit types
- In Python, values have an implicit type

Concept : Type Systems

- There are rules that dictate what you can and can not do with various types

```
x = 42 + "Hello"   # TypeError
```

- These rules make up the "type system"
- In Python, checking occurs at run-time (dynamic typing)
- In C++, checking occurs in the compiler (static typing)

Type System Elements

- The type system is more than just the representation of data
- Example : Mutability of data

```
const int x = 42;
...
x = 37;      // Error. x is "const"
```

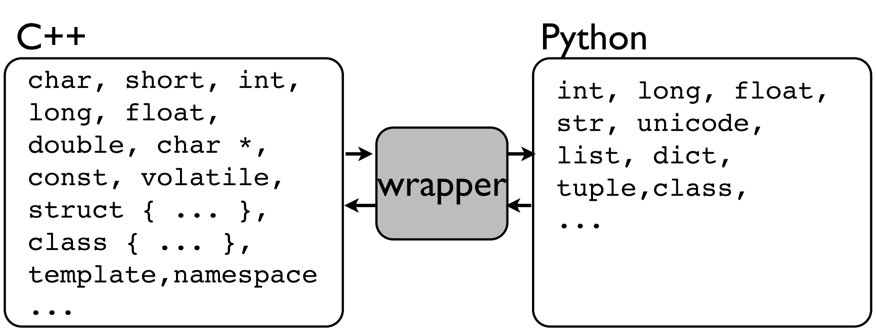
- Example: Inheritance in OO

```
class Foo {};
class Bar : public Foo {};
class Spam {};

Foo  *a = new Foo();     // Ok
Foo  *b = new Bar();     // Ok (Bar is a Foo)
Foo  *c = new Spam();    // Error (Spam is not a Foo)
```


Extension Building

- Extensions are mainly a type-system problem
- When you write "wrappers", you are creating glue that sits between two type systems



This Makes Sense

- When you write Python extension code, about 95% of the time, you're futzing around with various forms of type conversion
  - Converting arguments from Python to C
  - Converting results from C to Python
- It's clearly a problem that is at least strongly related to type systems.

Extension Building Tools

- If you start using extension building tools, much of your time is also oriented around type handling
- It just looks different than if you're writing code by hand.
- Example: Using the ctypes module

ctypes Example

- A C function

```
double half(double x) { return x/2; }
```

- Loading a DLL

```
>>> import ctypes
>>> ext = ctypes.cdll.LoadLibrary("./libext.so")
>>> ext.half(5)
-1079032536
>>>
```

- Fixing the types

```
>>> ext.half.argtypes = (ctypes.c_double,)
>>> ext.half.restype = ctypes.c_double
>>> ext.half(5)
2.5
>>>
```


The Problem

- Understanding the type system is a lot harder than it looks
- There's much more to it than just converting data back and forth
- Example : How many C/C++ programmers would claim that they really understand the C++ type system?

C Type System

- Example: Explain the difference

```
const char *s;
char *const s;
const char *const s;
```

- Example: What is the following?

```
void (*s(int, void (*)(int)))(int);
```

- Example: Explain the difference

```
int **x;
int   y[10][10];
```


C++ Type System

- Example: Explain this code

```
template<int N> struct F {
    enum { value = N*F<N-1>::value };
};
template<> struct F<0> {
    enum { value = 1 };
};

int x = F<4>::value;
```

Part 3

Inside the C/C++ Type System

Primitive C Types

- C is based on a primitive set of types

Byte	`char`	`8 bits`
Integer	`int`	`Typically 32 bits`
Floating point	`float` `double`	`32 bit single precision` `64 bit double precision`

- These types are a direct reflection of low-level computer hardware (the integer/floating point units of a microprocessor).

long/short Modifiers

- Integers can have "short" and "long" modifiers added to them to get different word sizes

```
short int        # 16 bit integer
long  int        # 32 or 64 bit integer
long long int    # 64 bit integer (support varies)
```

- Since the "int" is redundant, it is often dropped

```
short            # 16 bit integer
long             # 32 or 64 bit integer
long long        # 64 bit integer (support varies)
```

- long can also be applied to double (sometimes)

```
long double      # 128 quad-precision float
```


Signed/Unsigned Modifiers

- Integer types can also have a sign modifier

```
signed char,  unsigned char
signed short, unsigned short
signed int,   unsigned int
signed long,  unsigned long
```

- This modifier only provides information to the compiler on how the underlying data should be <u>interpreted</u>.

```
 (bunch of bits)
[1111111011101111] signed short   -> -275
[1111111011101111] unsigned short -> 65261
```

- No effect on the underlying data representation

Simple Datatypes

- If you take the primitive types and associated modifiers you get a complete list of the simple C types that are used to represent data.

```
char                      // 8-bit signed int
unsigned char             // 8-bit unsigned int
short                     // 16-bit signed int
unsigned short            // 16-bit unsigned int
int                       // 32-bit signed int
unsigned int              // 32-bit unsigned int
long                      // 32/64-bit signed int
unsigned long             // 32/64-bit unsigned int
long long                 // 64-bit signed int
unsigned long long        // 64-bit unsigned int
float                     // 32-bit single precision
double                    // 64-bit double precision
```


Python and C Data

- The Python C API mirrors this set of types (PyArg_ParseTuple() conversion codes)

```
Format  Python Type              C Datatype
------  ------------------------ ------------------------
"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
"L"     Integer                  long long
"K"     Integer                  unsigned long long
"f"     Float                    float
"d"     Float                    double
```

Type Declarators

- C has more complicated kinds of types
- For example: pointers, arrays, and qualified types

```
int *            // Pointer to an int
int  [40]        // Array of 40 integers
int *[40]        // Array of 40 pointers to integers
int *const       // Pointer to a constant int
int *const [40]  // Array of 40 pointers to const int
```

- These are "constructed" by taking a basic type and applying a sequence of "declarators" to it

Commentary

- The syntax for declarators is mind-boggling

```
void (*s(int, void (*)(int)))(int);
int *(*x)[10][20];
```

- That's almost impossible to read
- They are much easier to understand if you write them out as a sequence

C Declarator Operators

- There are four basic declarators
  - *. Pointer to something
  - [N]. An array of N items
  - *qualifier.* A qualifier (const, volatile)
  - (*args*). A function taking args

C Declarator Operators

- You can rewrite C types as a sequence that more easily shows its construction...
- Examples:

```
int                        int
int *                      *.int
int [40]                   [40].int
int *[40]      ------->    [40].*.int
const int *                *.const.int
int *(*)[10][20]           *.[10].[20].*.int
int (*)(int *,int)         *.(*.int,int).int
```

- Read the alternative syntax left to right

Declarations/Statements

- In C, there is a distinction between statements and declarations
- Statements make up the implementation

```
x = a + b;
foo(x,y);
for (i = 0; i < 100; i++) { ... }
```

- Declarations specify type information

```
double x;
double a,b;
void foo(double, int);
```


Declarations/Statements

- For extension modules, we do not care about the implementation
- We are only interested in declarations
- And only those declarations that are visible
- The public interface to C/C++ library code
- So, let's look further at declarations...

Declarations

- A declaration binds a name and storage specifier to a type

```
int a,*b;
extern  int foo(int x, int y);
typedef int Integer;
static  void bar(int x);
```

- Name : A valid C identifier
- Storage : extern, static, typedef, virtual, etc.

Declaration Tables

- Declarations are easily stored in a table

```
int a,*b;
extern  int foo(int x, int y);
typedef int Integer;
static  void bar(int x);
```

↓

Name	storage	type
'a'	None	int
'b'	None	*.int
'foo'	extern	(int,int).int
'Integer'	typedef	int
'bar'	static	(int).void
...		

Namespaces

- There is always a global declaration table (::)
- However, declarations can also appear in
  - Structures and unions
  - C++ classes
  - C++ namespaces
- Each of these is just a named declaration table

Namespaces

- Example:

```
class Spam {
public:
    int a,b;
    virtual int bar(int);
    static int foo(void);
};
```

- Class declaration table

Spam

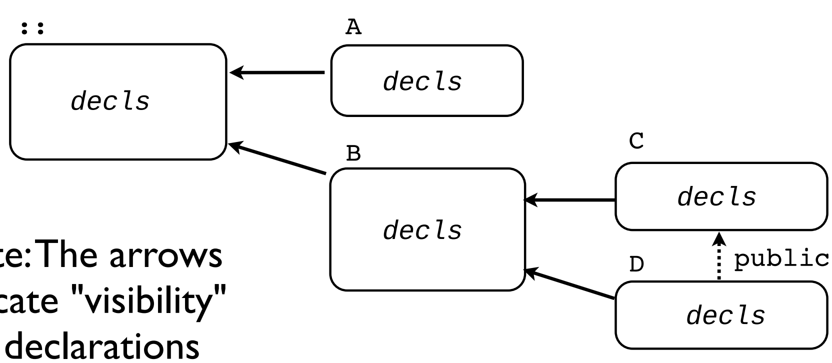
Name	storage	type
'a'	None	int
'b'	None	int
'bar'	virtual	(int).int
'foo'	static	(void).int

Namespaces

- The tricky bit with namespaces is that different namespaces can be nested and linked together
- Inner namespaces see declarations in outer namespaces
- Class namespaces see declarations in namespace for parent class (inheritance)
- All of this gets implemented as a tree

C++ Namespace Example

```
decls
class A { decls } ;
namespace B {
    decls
    class C { decls };
    class D : public C { decls };
}
```



Note: The arrows indicate "visibility" of declarations

Overloading

- C++ allows overloaded functions/methods

```
int  foo(int x);
int  foo(double x);
int  foo(int x, int y);
void foo(char *s, int n);
```

- Each func declaration must have unique args

Name	storage	type
'foo'	None	**(int)**.int
'foo'	None	**(double)**.int
'foo'	None	**(int,int)**.int
'foo'	None	**(*.char,int)**.void

- The return type is irrelevant

Templates

- C++ allows a declaration to be parameterized

```
template<parms> decl;
```

- Parameters are specified as a list of types

```
template<class T> decl;
template<int n> decl;
template<class T, int n> decl;
...
```

- To refer to a template, you use the declaration name with a set of parameters

```
name<args>
```

Template Implementation

- Implementing templates is slightly tricky (sic)
- It's a declaration with arguments, so just add an extra table column for that

```
int a;
int foo(int x, int *y);
template<class T> T max(T a, T b);
...
```

↓

Name	storage	template	type
'a'	None	None	int
'foo'	None	None	(int,*.int).int
'max'	None	(class T)	(T,T).T
...			

Template Implementation

- Identifiers may also carry arguments

```
int a;
int foo(int x, int *y);
template<class T> T max(T a, T b);
vector<int> blah(vector<int> *x, int n);
```

↓

Name	storage	template	type
'a'	None	None	int
'foo'	None	None	(int,*.int).int
'max'	None	(class T)	(T,T).T
'blah'	None	None	**(*.vector<int>,int).vector<int>**

- Nothing changes in the table, just horrid names

Putting it All Together

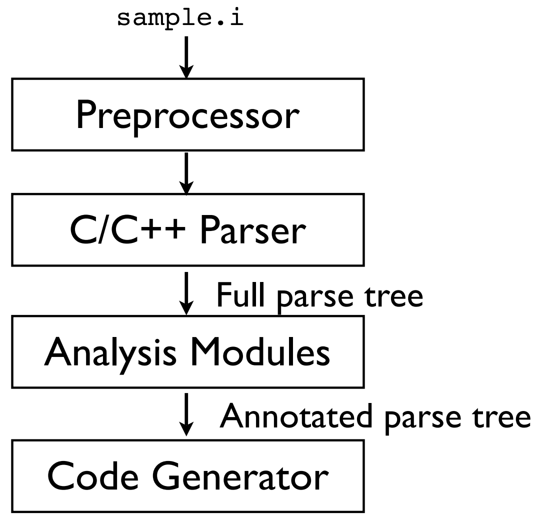
- The key to everything is knowing that C/C++ header files basically just define a bunch of declaration tables
- These tables have a very simple structure (even with features such as C++ templates)
- If you can assemble the declaration tables, you can generate wrappers

Seque to Swig

- This is essentially how Swig works
  - Parse a C++ header file
  - Create declaration tables
  - Manipulate the declaration tables
  - Generate wrapper code

Swig Architecture

- Swig is a multi-pass compiler



Discussion

- The phases build upon each other
- Each phase has various customization features that can be applied to control processing
- These are controlled by special directives which are always prefixed by %
- Let's look at each phase...

Part 4

The Preprocessor

Preprocessor

- There is a full ANSI C preprocessor
- Supports file includes, conditional compilation, macro expansion, variadic macros, etc.
- Also implements a number of Swig-specific extensions related to file inclusion and macros

Preprocessing

- The preprocessor is the primary entry point
- Here's what happens when you run Swig

```
% swig -python sample.i
```

expands to...

```
%include <swig.swg>

%include <python.swg>

%include "sample.i"
```

The first two files are part of Swig and contain definitions needed to process the .i file that follows

Preprocessor

Digression

- The previous slide explains the cryptic error you get if you don't install Swig properly

```
% swig -python sample.i
```

```
%include <swig.swg>

%include <python.swg>

%include "sample.i"
```

```
:1: Error: Unable to find 'swig.swg'
:3: Error: Unable to find 'python.swg'
```

(oops!)

Viewing Preprocessed Text

- The result of preprocessed input is easy to view

```
% swig -E -python sample.i
```

- This will show you the exact input that actually gets fed into the Swig parser
- Some of this will be rather cryptic, but the goal is to make life easier for the parser

Preprocessor Extensions

- Swig makes the following extensions to the normal C preprocessor
  - A different set of file-inclusive directives
  - Code literals
  - Constant value detection
  - Macro extensions

File Includes

- Swig uses its own file inclusion directives
- %include : Include a file for wrapping

```
%include "other.i"
```

- %import : Include for declarations only

```
%import "other.i"
```

- Rationale : Sometimes you want it wrapped and sometimes you don't.

File Includes

- By default, Swig ignores all preprocessor #include statements
- Rationale : Swig doesn't know what you want to do with those files (so it plays safe)
- All of this can be controlled:

```
swig -I/new/include/dir    # Add a search path
swig -importall            # All #includes are %import
swig -includeall           # All #includes are %include
```

Getting File Dependencies

- Listing the file dependencies : swig -M

```
% swig -python -M sample.i
sample_wrap.c: \
 /usr/local/share/swig/1.3.31/swig.swg \
 /usr/local/share/swig/1.3.31/swigwarnings.swg \
 /usr/local/share/swig/1.3.31/swigwarn.swg \
 /usr/local/share/swig/1.3.31/python/python.swg \
 /usr/local/share/swig/1.3.31/python/pymacros.swg \
 /usr/local/share/swig/1.3.31/typemaps/swigmacros.swg \
 /usr/local/share/swig/1.3.31/python/pyruntime.swg \
 /usr/local/share/swig/1.3.31/python/pyuserdir.swg \
 /usr/local/share/swig/1.3.31/python/pytypemaps.swg \
 /usr/local/share/swig/1.3.31/typemaps/fragments.swg \
...
```

- This will show you the files in the same order as they are included and will be parsed

Code Literals

- Raw C/C++ code often has to pass through the preprocessor so that it can go into the wrapper output files
- The preprocessor ignores all code in %{..%}

```
%{
#include "myheader.h"
...
%}
```

Constant Value Detection

- C/C++ headers often use #define to denote constants

```
#define  PI         3.1415926
#define  PI2        (PI/2)
#define  LOGFILE    "app.log"
```

- But macros are often used for other things

```
#define EXTERN      extern
```

- The preprocessor uses a heuristic to try and detect the constants for wrapping

Constant Value Detection

- Example:

```
%module simple.i

#define  PI         3.1415926
#define  PI2        (PI/2)
#define  LOGFILE    "app.log"
#define  EXTERN     extern
```

- swig -E simple.i

```
...
%constant PI = 3.1415926;
%constant PI2 = (3.1415926/2);
%constant LOGFILE = "app.log";
```

Swig Macros

- Swig also has its own macros that extend the capabilities of the normal C preprocessor

```
// sample.i
%define %greet(who)
%echo "Hello " #who ", I'm afraid you can't do that"
%enddef

%greet(Dave)
```

- Example:

```
% swig -python sample.i
Hello Dave, I'm afraid you can't do that
%
```


Swig Macros

- The macro system is a critical part of Swig
- The macro system is used to reduce typing by automatically generating large blocks of code
- For better or worse, a lot of Swig low-level internals are heavily based on macros

Swig Macros

- Frankly, the macro system is frightening.

```
%define FACTORIAL(N)
#if N == 0
1
#else
(N)*FACTORIAL(N-1)
#endif
%enddef

int x = FACTORIAL(6);        // 720
```

- Supports recursive preprocessing
- Macros can define other macros (yow!)

Part 5

The C++ Parser

Parsing in a Nutshell

- The parser constructs a full parse tree from the input
- Each node in this tree is identified by a "tag" that describes what it is
- These tags mimic the struct of the input file.
- You can easily view the tree structure

```
% swig -python -debug-tags sample.i
% swig -python -dump_tags sample.i
```


Viewing the Tree Structure

```
%module sample
%{
#include "myheader.h"
#include "otherheader.h"
%}

#define PI 3.14159
int    foo(int x,
double bar(const

struct Spam {
    int a, b;
};
```

```
% swig -python -debug-tags sample.i
    ...
  . top . include (sample.i:0)
  . top . include . module (sample.i:1)
  . top . include . insert (sample.i:5)
  . top . include . constant (sample.i:7)
  . top . include . cdecl (sample.i:8)
  . top . include . cdecl (sample.i:9)
  . top . include . class (sample.i:11)
  . top . include . class . cdecl (sample.i:12)
  . top . include . class . cdecl (sample.i:12)
```

Parse Tree Nodes

- All parse tree nodes are dictionaries
- They're not Python dictionaries, but virtually identical---a mapping of keys (strings) to values
- The values are either numbers, strings, lists, or other dictionaries
- The parse tree nodes are also easy to view

```
% swig -python -debug-module 1 sample.i
% swig -python -dump_parse_module sample.i
```


Viewing the Tree Nodes

```
% swig -python -debug-module 1 sample.i
...
+++ cdecl --------------------------
| sym:name     - "bar"
| name         - "bar"
| decl         - "f(p.q(const).char)."
| parms        - char const *
| kind         - "function"
| type         - "double"
| sym:symtab   - 0x32db70
| sym:overname - "__SWIG_0"
|
...
```

```
double bar(const char *s);
```

There's a lot of information here...

Viewing the Tree Nodes

```
% swig -python -debug-module 1 sample.i
...
+++ cdecl ------------------------
| sym:name      - "bar"
| name          - "bar"
| decl          - "f(p.q(const).char)."
| parms         - char const *
| kind          - "function"
| type
| sym:symtab
| sym:overnam
|
...
```

double bar(const char *s);

This is the node tag
(A C declaration)

Viewing the Tree Nodes

```
% swig -python -debug-module 1 sample.i
...
+++ cdecl ------------------------
| sym:name      - "bar"
| name          - "bar"
| decl          - "f(p.q(const).char)."
| parms         - char const *
| kind          - "function
| type          - "double"
| sym:symtab    - 0x32db70
| sym:overname  - "__SWIG_0
|
...
```

double **bar**(const char *s);

These attributes hold the
declaration name

name	The name used in C
sym:name	The name used in Python

Viewing the Tree Nodes

```
% swig -python -debug-module 1 sample.i
...
+++ cdecl ------------------------
| sym:name      - "bar"
| name          - "bar"
| decl          - "f(p.q(const).char)."
| parms         - char const *
| kind          - "function"
| type          - "double"
| sym:symtab    - 0x32db70
| sym:overname  - "__SWIG_0"
|
...
```

```
double bar(const char *s);
```

Declaration base type and type operators

type	The base type
decl	Type operators

Type Syntax Explained

- Swig uses a similar representation of the declarator operators we looked at earlier

```
p.                  *.
a(N).               [N].
q(qualifiers).      qualifiers.
f(parms).           (parms).
```

- An Example:

```
double bar(const char *);
        ↓
f(p.q(const).char).double
```

Viewing the Tree Nodes

```
% swig -python -debug-module 1 sample.i
...
+++ cdecl ----------------------------------------
| sym:name      - "bar"
| name          - "bar"
| decl          - "f(p.q(const).char)."
| parms         - char const *
| kind          - "function"
| type          - "double"
| sym:symtab    - 0x32db70
| sym:overname  - "__SWIG_
|
...
```

```
double bar(const char *s);
```

Other fields have certain information split out so that later processing is easier

kind	The kind of declaration
parms	List of argument types

Different cdecl Types

```
% swig -python -debug-module 1 sample.i
...
+++ cdecl ----------------------------------------
| sym:name      - "bar"
| name          - "bar"
| decl          - "f(p.q(const).char)."
| parms         - char const *
| kind          - "function"
| type          - "double"
|
+++ cdecl ----------------------------------------
| sym:name      - "spam"
| name          - "spam"
| decl          - ""
| kind          - "variable"
| type          - "int"
```

```
double bar(const char *s);
```

```
int spam;
```

Viewing the Tree Nodes

```
% swig -python -debug-module 1 sample.i
...
+++ cdecl ------------------------
| sym:name     - "bar"
| name         - "bar"
| decl         - "f(p.q(const).char)
| parms        - char const *
| kind         - "function"
| type         - "double"
| sym:symtab   - 0x32db70
| sym:overname - "__SWIG_0"
|
...
```

```
double bar(const char *s);
```

Symbol tables

sym:symtab is the (C++) namespace where the declaration lives.

sym:overname is related to overloaded symbols

Parsing Directives

- There are two important directives that relate to the construction of the parse tree
- %extend : Class/structure extension
- %template : Template instantiation

%extend Directive

- Extends a class with additional declarations

```
%extend Spam {
   void set(int a, int b) {    /* Added method */
       $self->a = a;
       $self->b = b;
   }
};
...

struct Spam {
    int a, b;
};
```

- The purpose of doing this is to provide additional functionality in the final wrappers

%extend Directive

- Usage of the extended class

```
>>> import sample
>>> s = sample.Spam()
>>> s.set(3,4)             # extended method
>>>
```

- Clever use of this feature can result in Python wrappers that look very different than the original C/C++ source

%extend Directive

- %extend works by collecting the extra declarations and attaching them to the end of the parse tree node of the named class

```
%extend Spam {
   void set(int a, int b) {
       $self->a = a;
       $self->b = b;
   }
};
...

struct Spam {
    int a, b;
};
```


Extended Parse Tree

```
% swig -python -debug-module 1 sample.i
  +++ class ----------------------------------------
  | name         - "Spam"
  | kind         - "struct"
        +++ cdecl ----------------------------------------
        | name         - "a"
        | type         - "int"
        +++ cdecl ----------------------------------------
        | name         - "b"
        | type         - "int"
        +++ extend ----------------------------------------
              +++ cdecl ------------------------------
              | name         - "set"
              | decl         - "f(int,int)."
              | code         - "{\n        $self->a = a;\n
              | kind         - "function"
              | type         - "void"
              | feature:extend - "1"
```

The extension

%extend Directive

- %extend can appear anywhere

```
struct Spam {
    int a, b;
};
...
%extend Spam {
    ...
};
```

- %extend is open (can appear more than once)

```
%extend Spam { .... };
...
%extend Spam { ... };
...
struct Spam { ... };
```

- Repeats are collected together

%template Directive

- %template instantiates a template

```
template<class T> T max(T a, T b) { ... }
...
%template(maxint) max<int>;
%template(maxdouble) max<double>;
```

- This is needed for a few reasons
- First, if you're going to use a template, you have to give it a valid Python identifier
- Swig doesn't really know what templates you actually want to use---you need to tell it

%template directive

```
% swig -c++ -python -debug-module 1 sample.i

 +++ template ---------------  template<class T> T max(T a,T b)
 | templatetype - "cdecl"
 | name         - "max"
 | sym:name     - "max"
 | decl         - "f(T,T)."
 | parms        - T,T
 | type         - "T"
 |
 ...
 +++ cdecl --------------------  %template(maxint) max<int>;
 | name         - "max<(int)>"
 | kind         - "function"
 | sym:name     - "maxint"
 | decl         - "f(int,int)."
 | parms        - int,int
 | type         - "int"
 |
```


%template Directive

- %template is really just a macro expansion in the parse tree
- Every use inserts a copy of the templated declaration into the parse tree where all of the types have been appropriately expanded

Template Discussion

- Manual instantiation of templates is one area where Swig is weak
- Can get real messy if you present Swig with code that makes heavy use of advanced C++ idioms (e.g., template metaprogramming)
- Swig is coming into that code as an "outsider"
- Compare to Boost.Python which uses C++ templates to wrap C++ (a neat trick BTW)

Part 6

Code Analysis

Code Analysis Phases

- After parsing, the parse tree is analyzed by other parts of Swig
- There are currently 3 phases

```
Phase 1     : C/C++ Parsing (just covered)
Phase 2     : Type processing
Phase 3     : Allocation analysis
```

- View the results using

```
% swig -python -debug-module PhaseN sample.i
```


Type Processing

- This phase looks at all of the types, classes, typedefs, namespaces, and prepares the parse tree for later code generation
- Fully expands all of the type names
- Example: Namespaces

```
namespace Spam {
    typedef double Real;
    Real foo(Real x);
};
```

Type Processing

```
% swig -c++ -python -debug-module 1 sample.i
     +++ cdecl   ----------------------------------------
     | sym:name     - "foo"
     | name         - "foo"
     | decl         - "f(Real)."
     | parms        - Real
     | kind         - "function"
     | type         - "Real"
```

(Type Processing)

```
% swig -c++ -python -debug-module 2 sample.i
     +++ cdecl ----------------------------------------
     | sym:name     - "foo"
     | name         - "Spam::foo"
     | decl         - "f(Spam::Real)."
     | parms        - Spam::Real
     | kind         - "function"
     | type         - "Spam::Real"
```

Notice expansion of type and declaration names

Allocation Analysis

- This phase mainly analyzes the memory management properties of the classes
- Default constructor/destructors
- Detecting the use of smart pointers
- Marking classes used as C++ exceptions
- Virtual function elimination optimization

Alloc Analysis : Example

- Detecting whether or not it is safe to create default constructor and destructor wrappers

```
%module sample
...
struct Spam {
    int a, b;
};
```

```
>>> import sample
>>> s = sample.Spam()
>>> s.a = 32
>>> s.b = 13
...
>>> del s
```

- In the interface, nothing is specified about creation/destruction.

After Parsing

```
% swig -python -debug-module 1 sample.i
+++ class -----------------------------
| name          - "Spam"
| kind          - "struct"
      +++ cdecl ----------------------
      | name          - "a"
      | decl          - ""
      | type          - "int"
      +++ cdecl ----------------------------------------
      | name          - "b"
      | decl          - ""
      | type          - "int"
```

```
struct Spam {
   int a,b;
};
```

After Allocation Analysis

```
% swig -python -debug-module 3 sample.i
+++ class ----------------------------------------
| name          - "Spam"
| kind          - "struct"
| allocate:default_constructor - "1"
| allocate:default_destructor - "1"
| allocate:copy_constructor - "1"
      +++ cdecl ----------------------------------------
      | ismember      - "1"
      | name          - "a"
      | decl          - ""
      | type          - "int"
      +++ cdecl ----------------------
      | ismember      - "1"
      | name          - "b"
      | decl          - ""
      | type          - "int"
```

```
struct Spam {
   int a,b;
};
```

These added fields indicate whether or not it's safe to create default constructor/ destructor functions

Discussion

- Code analysis phases look at the parse tree and add additional attributes
- Essentially, Swig is building a more complete picture of what's happening in the module
- Keep in mind, all of this occurs before Swig ever generates a line of output
- It's prepping the module for the Python code generator that will run at the end

Part 7

Decorating the Parse Tree

Giving Swig Hints

- Swig only looks at the contents of headers
- There are a lot of things that can be determined automatically
- Especially certain semantics of classes
- However, there are other aspects of making an extension module where user input is required

Name Conflicts

- Suppose a C++ header uses a reserved word

```
class Foo {
public:
    virtual void print(FILE *f);
    ...
};
```

- There is no way this can be wrapped using the given method name
- Must pick an alternative...

%rename Directive

- %rename renames a declaration

```
%rename(cprint) print;
...
class Foo {
public:
    virtual void print(FILE *f);
    ...
};
```

- This slightly alters the parse tree...

%rename Directive

```
% swig -python -debug-module 1 sample.i
  +++ class ----------------------------------------
  | sym:name      - "Foo"
  | name          - "Foo"
  | kind          - "class"
        +++ access ----------------------------------------
        | kind          - "public"
        |
        +++ cdecl ----------------------------------------
        | sym:name      - "cprint"
        | name          - "print"
        | decl          - "f(p.FILE)."
        | parms         - FILE *
        | kind          - "function"
        | type          - "void"
```

Here is the renamed declaration

%ignore Directive

- %ignore ignores a declaration (in the wrappers)

```
%ignore print;
...
class Foo {
public:
    virtual void print(FILE *f);
    ...
};
```

- This is also a parse tree manipulation...

%ignore Directive

```
% swig -python -debug-module 1 sample.i
  +++ class ----------------------------------------
  | sym:name     - "Foo"
  | name         - "Foo"
  | kind         - "class"
        +++ access ----------------------------------------
        | kind         - "public"
        |
        +++ cdecl ----------------------------------------
        | name         - "print"
        | feature:ignore - "1"
        | decl         - "f(p.FILE)."
        | parms        - FILE *
        | kind         - "function"
        | type         - "void"
```

Declaration is marked as "ignored"

%immutable Directive

- %immutable makes a declaration read-only

```
%immutable Spam::a;
%immutable Spam::b;

...
struct Spam {
    int a,b;
};
```

%immutable Directive

```
% swig -python -debug-module 1 sample.i
  +++ class ----------------------------------------
  | sym:name     - "Spam"
  | name         - "Spam"
        +++ cdecl ----------------------------------------
        | sym:name     - "a"
        | name         - "a"
        | feature:immutable - "1"
        | kind         - "variable"
        | type         - "int"
        +++ cdecl ----------------------------------------
        | sym:name     - "b"
        | name         - "b"
        | feature:immutable - "1"
        | kind         - "variable"
        | type         - "int"
```

Declaration is marked as "immutable"

%newobject Directive

- %newobject marks a declaration as returning newly allocated memory

```
%newobject strdup;
...
char *strdup(const char *s);
```

- Maybe you want to know this so it can be cleaned up properly

%newobject Directive

```
% swig -python -debug-module 1 sample.i
  +++ cdecl ----------------------------------------
  | sym:name     - "strdup"
  | name         - "strdup"
  | decl         - "f(p.q(const).char).p."
  | parms        - char const *
  | feature:new  - "1"
  | kind         - "function"
  | type         - "char"
  | sym:symtab   - 0x32a3b0
  | sym:overname - "__SWIG_0"
```

Declaration is marked as "new"

Declaration Annotation

- The last few examples are all the same idea
- You provide a hint regarding a specific declaration
- The hint shows up as a "feature" in the parse tree
- The code generator is programmed to look for various "features" as part of its processing

The %feature Directive

- Tags specific declarations with additional information

```
%feature("new","1") strdup;
...
char *strdup(const char *s);
```

- It attaches a value to parse tree nodes

```
+++ cdecl ----------------------------------------
| sym:name      - "strdup"
| name          - "strdup"
| decl          - "f(p.q(const).char).p."
| parms         - char const *
| feature:new   - "1"           Added feature
| kind          - "function"
| type          - "char"
```


The %feature Directive

- %feature can be narrowed to any single declaration in the input file
- Uses the same matching rules that C/C++ uses to uniquely identify declarations

```
%feature("blah","1")  Spam::foo(int) const;

class Spam {
public:
    void foo(const char *s, int);
    void foo(const char *s);
    void foo(int);
    void foo(int) const;
    void foo(double);
    ...
};
```

%feature Use

- Virtually all <u>declaration based</u> customizations in Swig are built using %feature (using macros)

```
%ignore                    %feature("ignore","1")
%newobject    ---------->  %feature("new","1")
%immutable                 %feature("immutable","1")
```

- Where it gets confusing : %feature is open-ended. There is no fixed set of "features" and any part of Swig can be programmed to look for specific feature names of interest.

%feature and Code

- Some features operate with code-blocks

```
%feature("except") Spam::bar {
   try {
        $action
   } catch (SomeException) {
        // Handle the exception in some way
   }
}
```

- Here, the entire block of code is captured and attached to the matching declaration
- In this case, we're attaching exception code

%feature Wildcards

- %feature can pinpoint exact declarations
- However, it can match ranges of declarations

```
%feature("blah","1");          // Tag everything!
%feature("blah","1") bar;      // Tag all 'bar' decls
%feature("blah","1") *::bar;   // All 'bar' in classes
%feature("blah","1") ::bar;    // All global 'bar'
```

- In these cases, all declarations that match will be tagged with the appropriate feature

%feature Commentary

- %feature is closely related in concept to Python decorators and Aspect Oriented Prog.
- You're basically "decorating" declarations with additional information
- This information is used by the low-level code generators to guide wrapper creation.

Discussion

- If you understand that Swig works by decorating the parse tree, you start to see how interfaces get put together
- Typical Swig interface

```
%module sample
%{
#include "sample.h"          <------- Preamble
%}

%feature(...) bar;
%feature(...) foo;           <------- Decorations
...
%include "sample.h"          <------- A header
...
```


Difficulties

- There are too many features! (dozens)

```
%immutable              %feature("immutable")
%nodefault              %feature("nodefault")
%nodefaultctor          %feature("nodefaultctor")
%nodefaultdtor          %feature("nodefaultdtor")
%copyctor               %feature("copyctor")
%exception              %feature("except")
%allowexcept            %feature("allowexcept")
%exceptionvar           %feature("exceptvar")
%catches                %feature("catches")
%exceptionclass         %feature("exceptionclass")
%newobject              %feature("new")
%delobject              %feature("del")
%refobject              %feature("ref")
%unrefobject            %feature("unref")
%callback               %feature("callback")
%fastdispatch           %feature("fastdispatch")
%director               %feature("director")
%contract               %feature("contract")
```

Difficulties

- The features are not randomly implemented
- There to solve some sort of customization
- Almost always related to underlying semantics of the code being wrapped
- However, you need to look at a manual to know all of the available options

Example

- Contract checking (a little known feature)

```
%contract sqrt(double x) {
require:
    x >= 0;
}
...
double sqrt(double);
```

- Specific language backends might define even more exotic features

Part 7

Code Generation

Code Generation

- The last phase of Swig processing is the generation of low-level wrapper code
- There are four basic building blocks
  - Inserting literal code into the output
  - Creating a wrapper function
  - Installing a constant value
  - Wrapping a global variable

Swig Output

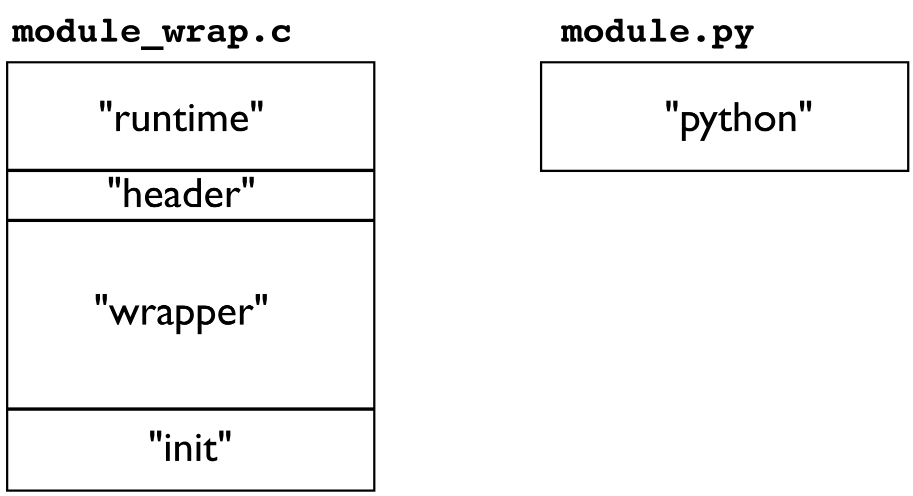
- Swig creates two different output files

```
shell % swig -python sample.i
shell % ls
sample.i sample_wrap.c sample.py
shell %
```

- The _wrap.c file is C code that must be compiled in a shared library
- The .py file is Python support code that serves as a front-end to the low-level C module

Output Code Sections

- Internally, there are 5 named file "targets"



%insert(section)

- Inserts literal code into any named file section

```
%insert("runtime") %{
   static void helloworld() { printf("Hello World\n"); }
%}

%insert("python") %{
# Print a welcome message
print "Welcome to my Swig module"
%}
```

- Note: These are usually aliased by macros

```
%runtime %{ ... %}
%header %{ ... %}          (Same as bare %{ ... %})
%wrapper %{ ... %}
%init %{ ... %}
```


Wrapper Functions

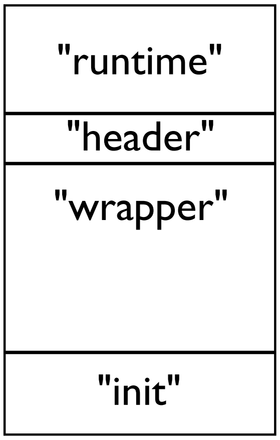
- The most elementary form of a Python extension function is the following:

```
PyObject *wrapper(PyObject *self, PyObject *args) {
    ...
}
```

- Swig wraps almost all C/C++ declarations with simple Python extension functions like this

Wrapper Creation

module_wrap.c




```
%module sample
...
int foo(int x, int y);
```

Create a wrapper

```
PyObject *
_wrap_foo(PyObject *self, PyObject *args)
{
    ...
}
```


Wrapper Initialization

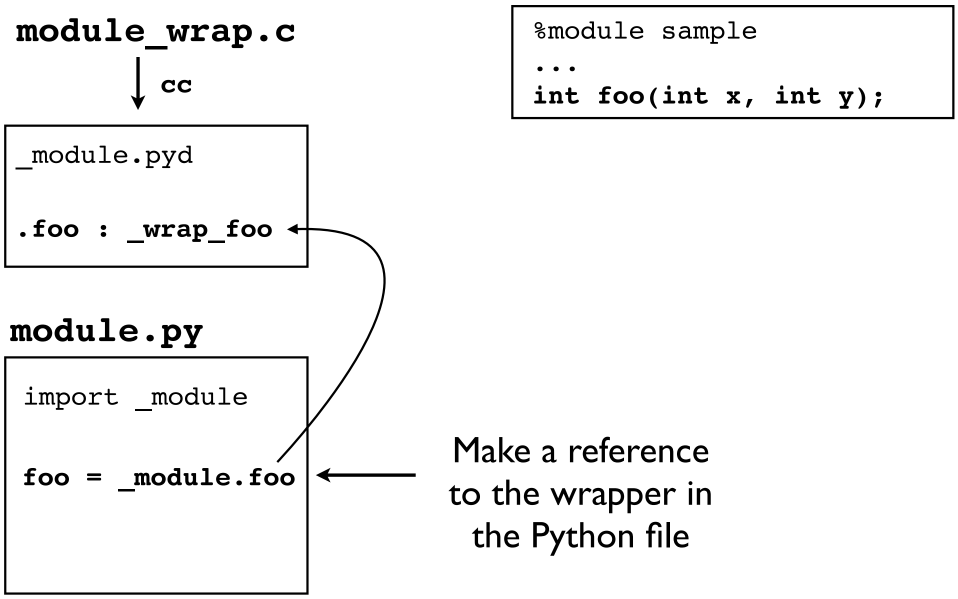
module_wrap.c


```
%module sample
...
int foo(int x, int y);
```

Installation of the wrapper into the module table

```
static PyMethodDef SwigMethods[] = {
   { "foo", _wrap_foo, METH_VARARGS, NULL}
   ...
};
```

Python Wrapper



Wrapper Functions

- Swig reduces all declarations to low-level wrappers
- Example : A C structure

```
%module sample
...
struct Spam {
    int a, b;
};
```

Reduction to Functions

```
struct Spam {
    int a, b;
};
```

Reduction to functions

```
Spam *new_Spam() { return (Spam *) malloc(sizeof(Spam)); }
void  delete_Spam(Spam *s) { free(s); }
int   Spam_a_get(Spam *s) { return s->a; }
void  Spam_a_set(Spam *s, int a) { s->a = a; }
int   Spam_b_get(Spam *s) { return s->b; }
void  Spam_b_set(Spam *s, int b) { s->b = b; }
```

This is a collection of "accessor" functions that provide access to the implementation of the structure

Wrapper Generation

```
Spam *new_Spam() { return (Spam *) malloc(sizeof(Spam)); }
void  delete_Spam(Spam *s) { free(s); }
int   Spam_a_get(Spam *s) { return s->a; }
void  Spam_a_set(Spam *s, int a) { s->a = a; }
int   Spam_b_get(Spam *s) { return s->b; }
void  Spam_b_set(Spam *s, int b) { s->b = b; }
```

Wrapper code

```
PyObject *_wrap_new_Spam(PyObject *self, PyObject *args);
PyObject *_wrap_delete_Spam(PyObject *self, PyObject *args);
PyObject *_wrap_Spam_a_get(PyObject *self, PyObject *args);
PyObject *_wrap_Spam_a_set(PyObject *self, PyObject *args);
PyObject *_wrap_Spam_b_get(PyObject *self, PyObject *args);
PyObject *_wrap_Spam_b_set(PyObject *self, PyObject *args);
```

Proxy Generation

_module.pyd

```
.new_Spam:     _wrap_new_Spam
.delete_Spam: _wrap_delete_Spam
.Spam_a_get:  _wrap_Spam_a_get
.Spam_a_set:  _wrap_Spam_a_set
.Spam_b_get:  _wrap_Spam_b_get
.Spam_b_set:  _wrap_Spam_b_set
```

module.py

```
class Spam(object):
    def __init__(self):
        self.this = _module.new_Spam()
    a = property(_module.Spam_a_get,_module.Spam_a_set)
    b = property(_module.Spam_b_get,_module.Spam_b_set)
    ...
```


Commentary

- There are a lot of low-level details I'm omitting
- A critical point : Swig never wraps C/C++ objects with Python types defined in C.
- Objects are always wrapped by proxies implemented partly in Python as shown

Part 8

Customizing Code Generation with Typemaps

Typemaps

- A customization feature applied to specific datatypes that appear in the input
- Background: The primary role of a wrapper function is to convert data between Python/C.
- Typemaps allow you to hook into that conversion process and customize it
- Without a doubt : This is the most mind-boggling part of Swig.

Introduction

- Consider this C function and a hand-written Python wrapper (from intro)

```
/* A simple C function */
double square(double x) {
    return x*x;
}

PyObject *py_square(PyObject *self, PyObject *args) {
    double x, result;
    if (!PyArg_ParseTuple(self,"d",&x)) {
        return NULL;
    }
    result = square(x);
    return Py_BuildValue("d",result);
}
```


Introduction

- In the wrapper, there is a <u>mapping from types</u> in the declaration to conversion code

```
/* A simple C function */
double square(double x) {
    return x*x;
}
          output          input
PyObject *py_square(PyObject *self, PyObject *args) {
    double x, result;
    if (!PyArg_ParseTuple(self,"d",&x)) {
        return NULL;
    }
    result = square(x);
    return Py_BuildValue("d",result);
}
```

%typemap directive

- Allows complete customization of what happens during type conversion

```
%typemap(in) double {
    // Custom input conversion code
}
%typemap(out) double {
    // Custom output conversion code
}


/* The C function to wrap */
double square(double x);
```


Sample Wrapper Code

```
PyObject *_wrap_square(PyObject *self, PyObject *args) {
  PyObject *resultobj = 0;
  double arg1 ;
  double result;
  PyObject * obj0 = 0 ;

  if (!PyArg_ParseTuple(args,(char *)"O:square",&obj0)) SWIG_fail;
  {
    // Custom input conversion code
  }
  result = (double)square(arg1);
  {
    // Custom output conversion code
  }
  return resultobj;
fail:
  return NULL;
}
```

%typemap variables

- A typemap is just a fragment of C code
- In that fragment, there are special substitutions

```
$1            - The value in C
$input        - The Python input value
$result       - The Python result value
```

- Example:

```
%typemap(in) double {
    $1 = PyFloat_AsDouble($input);
}
%typemap(out) double {
    $result = PyFloat_FromDouble($1);
}
```


Sample Wrapper Code

```
PyObject *_wrap_square(PyObject *self, PyObject *args) {
  PyObject *resultobj = 0;
  double arg1 ;
  double result;
  PyObject * obj0 = 0 ;

  if (!PyArg_ParseTuple(args,(char *)"O:square",&obj0)) SWIG_fail;
  {
    arg1 = PyFloat_AsDouble(obj0);
  }
  result = (double)square(arg1);
  {
    resultobj = PyFloat_FromDouble(result);
  }
  return resultobj;
fail:
  return NULL;
}
```

%typemap matching

- A typemap binds to both to types and names
- Can use that fact to pinpoint types

```
%typemap(in) double nonnegative {
     $1 = PyFloat_AsDouble($input);
     if ($1 < 0) {
         PyErr_SetString(PyExc_ValueError,"must be >=0");
         return NULL;
     }
}

double sqrt(double nonnegative);
```


%typemap matching

- Typemaps can also bind to typedef names

```
typedef double nndouble;
%typemap(in) nndouble {
     $1 = PyFloat_AsDouble($input);
     if ($1 < 0) {
         PyErr_SetString(PyExc_ValueError,"must be >=0");
         return NULL;
     }
}

double sqrt(nndouble x);
```

- The typemap only applies to types that exactly match that name

Interlude

- Normally, you don't have to define typemaps
- Swig already knows how to convert primitive datatypes, handle C/C++ pointers, etc.
- Typemaps only come into play if you want to make an extension module do something other than the default behavior

Example: Multiple Outputs

- Wrapping a function with multiple outputs

```
double do_sqrt(double x, int *status) {
    double result;
    if (x >= 0) {
        result = sqrt(x);
        *status = 1;
    } else {
        result = 0;
        *status = 0;
    }
    return result;
}
```

- Here, the function returns a result and a status
- Suppose you wanted both returned as a tuple?

Example: Multiple Outputs

- Typemaps to do this

```
%typemap(in,numinputs=0) int *status(int stat_value) {
   $1 = &stat_value;
}

%typemap(argout) int *status {
   PyObject *newobj = Py_BuildValue(("O",i),$result,*$1);
   Py_DECREF($result);
   $result = newobj;
}
...
double do_sqrt(double x, int *status);
```

- Now, let's look at what happens

Example: Multiple Outputs

```
%typemap(in,numinputs=0) int *status(int stat_value) {
   $1 = &stat_value;
}
```

```
PyObject *_wrap_do_sqrt(self, PyObject *args) {
  PyObject *resultobj = 0;
  double arg1 ;
  int *arg2 = (int *) 0 ;
  double result;
  int stat_value2 ;
  PyObject * obj0 = 0 ;

  {
    arg2 = &stat_value2;
  }
  if (!PyArg_ParseTuple(args,(char *)"O:do_sqrt",&obj0)) SWIG_fail;
  {
    arg1 = PyFloat_AsDouble(obj0);
  }
```

Variable to hold value

Set pointer to temporary variable

Example: Multiple Outputs

```
%typemap(argout) int *status {
   PyObject *newobj = Py_BuildValue("(O,i)",$result,*$1);
   Py_DECREF($result);
   $result = newobj;
}
```

```
PyObject *_wrap_do_sqrt(self, PyObject *args) {
   ...
   result = (double)do_sqrt(arg1,arg2);
  {
    resultobj = PyFloat_FromDouble(result);
  }
  {
    PyObject *newobj = Py_BuildValue("(O,i)",resultobj,*arg2);
    Py_DECREF(resultobj);
    resultobj = newobj;
  }
  return resultobj;
```


Example: Multiple Outputs

- Example use:

```
>>> import sample
>>> sample.do_sqrt(4)
(2.0, 1)
>>> sample.do_sqrt(-4)
(0.0, 0)
>>>
```

Commentary

- Writing typemap code is extremely non-trivial
- Requires knowledge of both Swig and Python
- May have to get into blood and guts of memory management (reference counting)
- Code that you write is really ugly
- However, you need to realize that people have already written a lot of this code

Typemap Libraries

- Swig comes with libraries of typemaps
- Use the %apply directive to use them

```
%include "typemaps.i"
%apply int *OUTPUT { int *status };

...
double do_sqrt(double x, int *status);
```

- Someone already figured out that output argument problem. We're using that code.
- %apply applies a set of typemaps to a new type

More Commentary

- People like to complain about typemaps
- Yet, it is not necessary to manually write typemap code in most situations
- If you are new to Swig and you are trying to write typemaps, you need to stop what you're doing and go re-read the documentation.
- Always intended as an advanced feature

Cautions

- The default set of Python typemaps is of considerable complexity (even I can't quite wrap my brain around all of it right now)
- Considerable effort concerning memory management, error handling, threads, etc.
- UTL (Universal Typemap Library). An effort to unify core typemaps across Python/Ruby/Tcl and other languages (heavy use of macros)

Part 9

Where to go from here?

Wrap-up

- Swig is built from a few basic components
- Preprocessor - For defining macros
- Parser - Grabs the an entire parse tree
- Features - Decoration of the parse tree
- Typemaps - Code fragments used in wrappers

Using Swig

- The best way to think of Swig is as a code generator based on pattern matching
- You're going to define various rules/ customizations for specific declarations and datatypes
- Those rules then get applied across a header file

Complaints

- Although each part of Swig is conceptually simple, all of the features we've described can interact with each other
- Can create interfaces that are a mind-boggling combination of macros/features/typemaps
- Swig is so flexible internally, that contributers have added a vast array of customizations
- I don't even fully understand all of it!

Start Simple

- To get the most out of Swig, it's best to start small and build over time
- Most of the really exotic features are not needed to get going
- Although Swig is complicated, I think the power of the implementation grows on you
- There are some really sick things you can do...

More Information

- http://www.swig.org
- There is extensive documentation
- Past papers describing Swig and how it works