Python, C++ and SWIG

Robin Dunn
Software Craftsman

O’Reilly Open Source Convention
July 21–25, 2008

Slides available at
http://wxPython.org/OSCON2008/

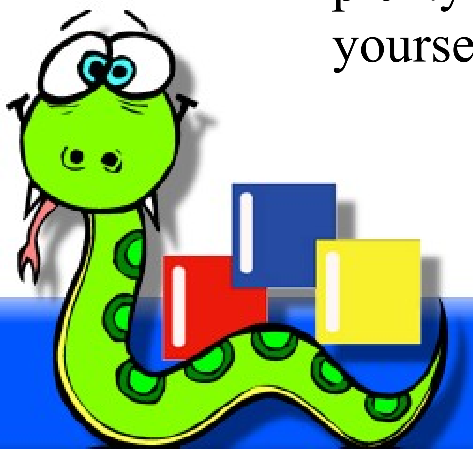
Python & C++ Comparisons

- Each is a general purpose programming language, applicable to all sorts of application domains
  - client/server
  - GUI/WUI
  - database
  - computational
  - business
  - games
  - etc.
- But each language also has its own set of strengths and weaknesses

Python & C++ Comparisons

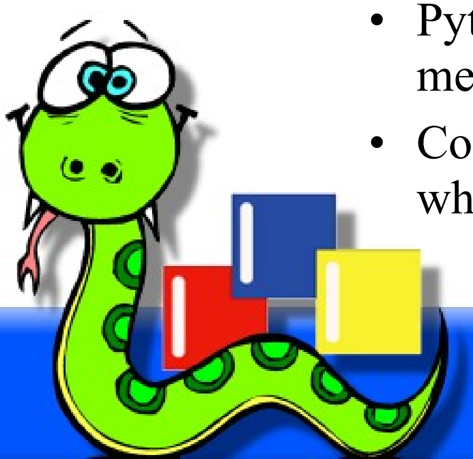
- C++
  - statically typed
  - compiled
  - complex
  - close to the metal
  - fast at runtime
  - long edit/compile/run cycles
  - typically lots of code needed to solve a problem
  - plenty of ways to shoot yourself in the foot

- Python
  - dynamically typed
  - interpreted
  - simple
  - higher level
  - slow, but usually fast enough
  - quick edit/run cycles
  - typically 5-10 times less code to solve same problem
  - your foot is much more safe



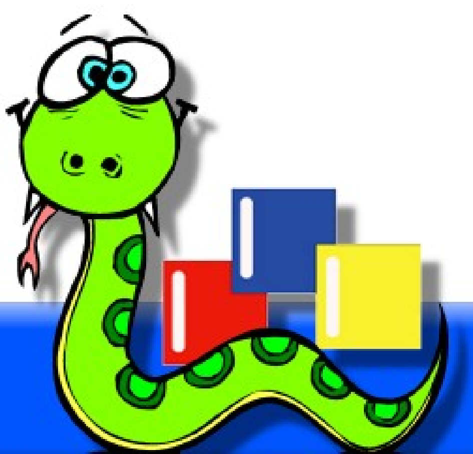
Python & C++ Comparisons

- In the end it all boils down to
  - Speed
    - Since C++ is “close to the metal” the compiled code has little runtime overhead and can be very quick executing. For many situations however there are much slower bottlenecks than the CPU so the speed of C++ isn't as valuable as you might think for these tasks.
  - Productivity
    - Because of the simpler language, dynamic typing (plus “duck typing”) and high level language support of advanced container and other types, much less code is needed and the programmer can be much more productive.
    - Python has no need for pointers, templates, macro preprocessor, explicit memory allocation, etc.
    - Common anecdotal rule: one Python programmer can do in 2 months what 2 C++ programmers are not able to finish in one year.



Best of Both?

- So is it possible to have the speed of C/C++ where it's needed, and the productivity, flexibility, ease of maintenance, and joy of using Python, all in the same application?

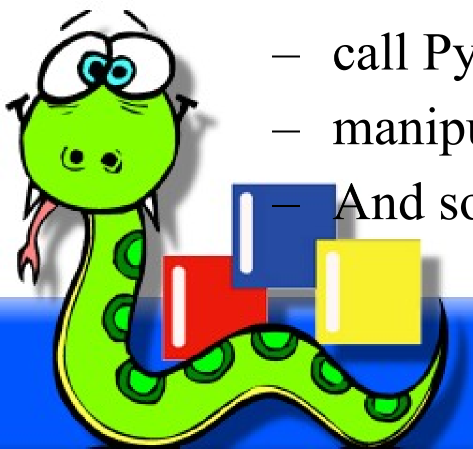


Best of Both? Yes!

- Python is designed from the beginning to be extensible in C
- The calling code doesn't need to know if the module is Python or C code
- Enables porting modules to C on an as-needed basis after prototyping and profiling, without disturbing the rest of the application
- Can be anything from simple wrappers around C library functions to very Python-esque things like new class types, etc.

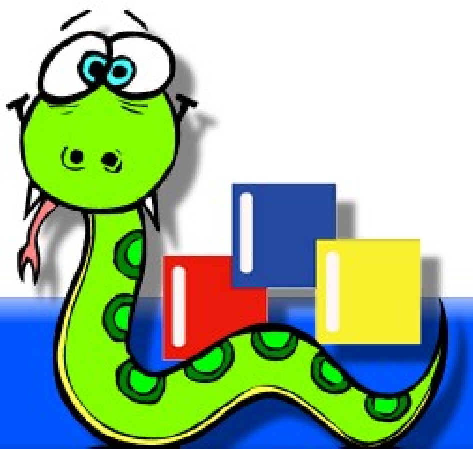
Python C API

- Broad and comprehensive APIs
- Used by extension modules to provide the glue between the Python code and the C/C++ code.
- Can deal with Python objects at an abstract level (e.g. anything that can act like a number)
- Or at a concrete level, (e.g. when you know you have an integer object)
- Can also
  - raise Python exceptions
  - call Python functions/methods
  - manipulate lists, dictionaries, etc.
  - And so on.



Extension modules

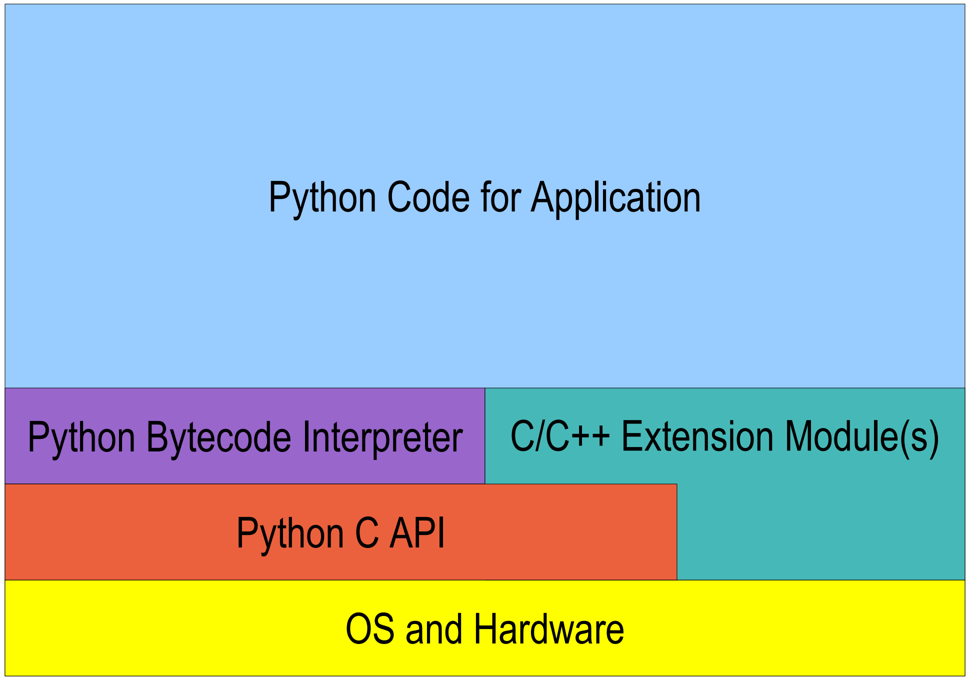
- Compiled and linked as a shared object file or DLL
- Placed on the Python path
- Used with the normal Python import statement
- Python uses the platform's normal APIs for dynamic loading
- Python calls a known initialization function, which creates the module object and all of its content
- You can then use the module the same as if it was written in Python instead of C.



Extension Modules

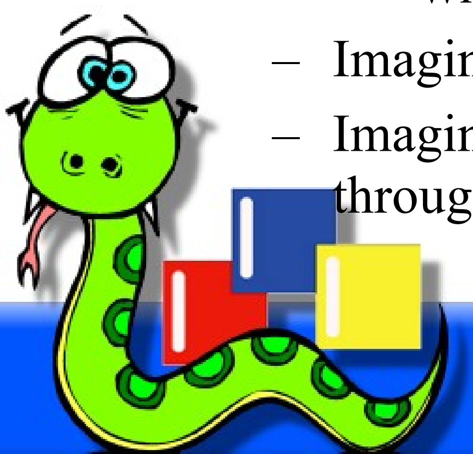
- You still have to deal with the negative aspects of C/C++, (complex code, memory allocations, longer edit/compile/run cycles, etc.) but:
  - It is for a relatively small portion of the whole application, just where speed or time critical things matter
  - Usually can be a self contained library
  - Usually can be easily unit tested to help maintain reliability, because the unit testing can happen from the Python side
  - Productivity gains of using Python for the rest of the application greatly outweigh extra work needed for making an extension module.

Basic Architecture


Python Code for Application
Python Bytecode Interpreter
C/C++ Extension Module(s)
Python C API
OS and Hardware

What's the Downside?

- Writing large extension modules by hand can be mind-numbingly tedious and error-prone grunt work
  - The process
    - Write a Python wrapper function for every C function you want to access
      - Convert input parameters
      - Call target function
      - Check for errors/exceptions
      - Convert return value
    - Create Python versions of C constants
    - Provide access to C variables, structures, and C++ classes
    - Write an initialization function
  - Imagine repeating that for a huge library with thousands of functions
  - Imagine propagating API changes in the target library by hand through the extension module code and the Python wrapper code

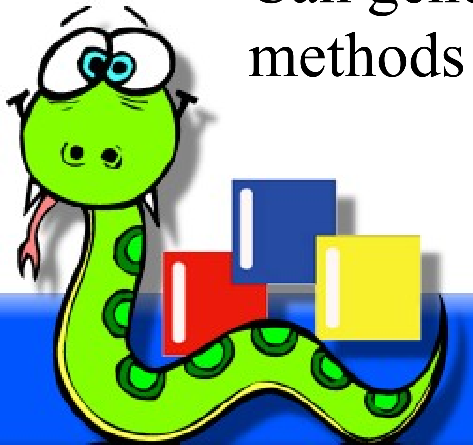


SWIG

- **S**implified **W**rapper **I**nterface **G**enerator
- Automates a huge amount of the work for extension modules, eliminating most of the time and tedium for development of extensions
- Facilitates the gluing of C/C++ libs to Python, allowing you to partition the application to the languages that can best handle each kind of task, building on strengths instead of living with weaknesses

SWIG

- A “compiler” that reads C/C++ declarations and writes C code implementing the extension
- Support for wrapping C structures and C++ classes, in addition to standalone functions, constants, etc.
- Also supports extensions for languages other than Python
- Understands the C++ class hierarchy and can do appropriate type checking of input parameters, etc.
- Can generate code for use with C++ templates
- Can generate code that allows overriding C++ virtual methods in Python classes

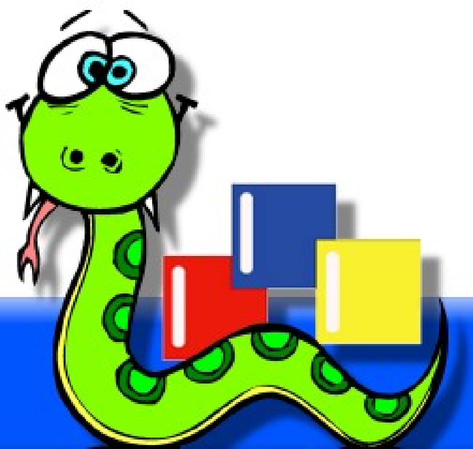


SWIG

- Many things work "out of the box" with no extra work, but SWIG also provides many ways to customize or add to the generated code
  - `%typemap`: controls what C code is generated in wrapper functions that receive or return objects of certain types
  - `%extend`: Can add new methods to a wrapped class that don't necessarily exist in the C++ class
  - `%inline`: Add new C functions that will automatically be wrapped
  - `%pythoncode`: Emit custom python code into the wrapper for the current class, or append or prepend it to the Python proxy function
  - Etc.

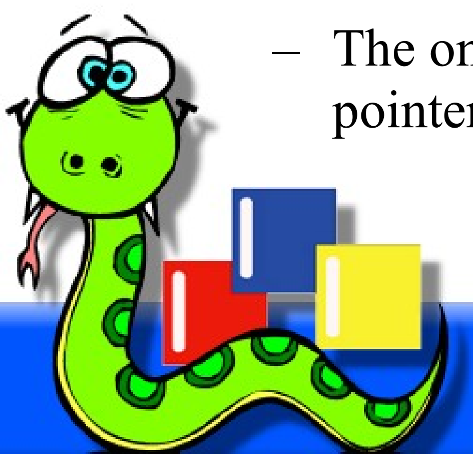
SWIG

- Includes a full C/C++ preprocessor, with some SWIG extensions for richer macros, etc.
  - Can use macros and conditional compilation directives in the interface declaration files
  - SWIG can understand the real headers for the library, and as long as they are "clean" it can sometimes make sense to use them directly with no extra work needed.

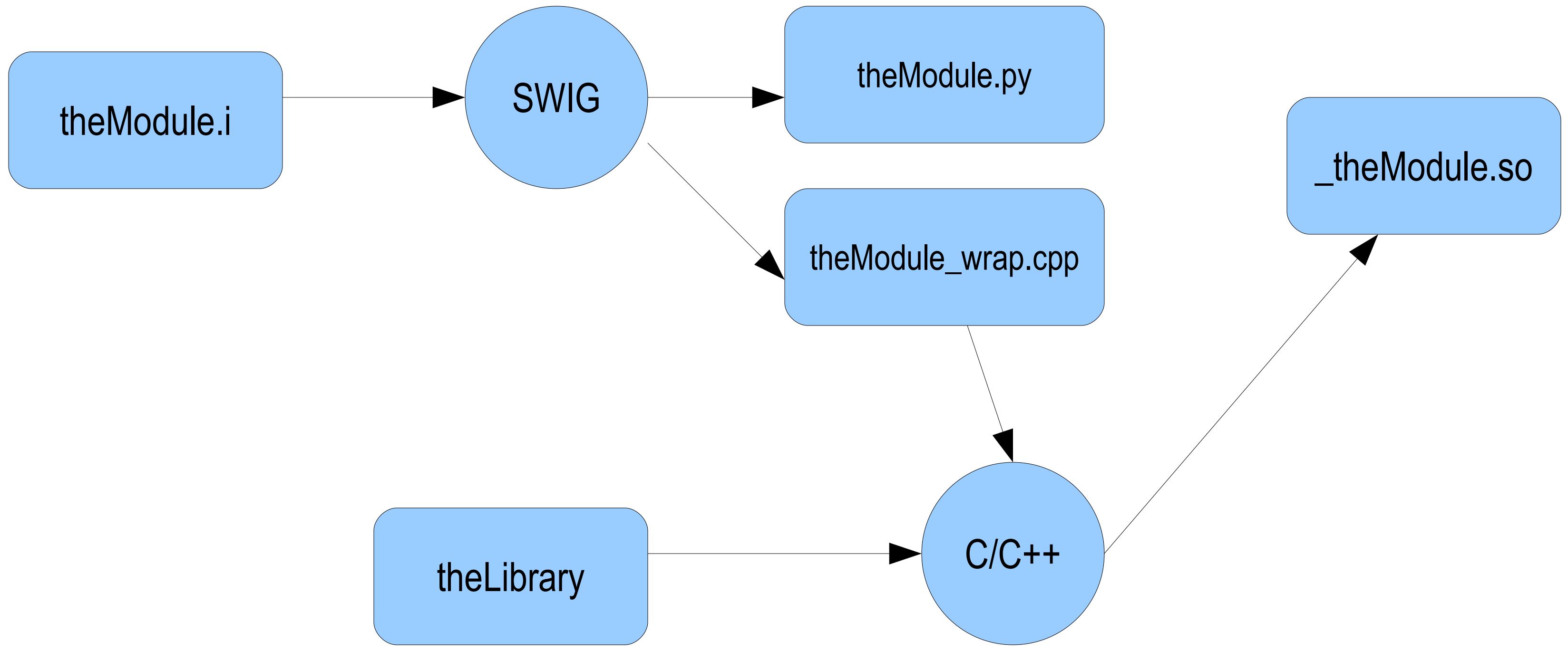


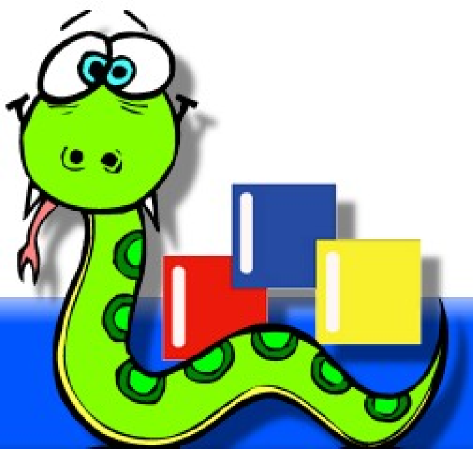
SWIG

- SWIG can either generate a “bare” extension module full of wrapped functions
- Or a combination of an extension module and a Python module with Proxy classes.
  - This is the best option for wrapping C++ classes by providing a natural mapping to Python classes
  - Offers lots of options for customizing or tweaking the API by emitting additional Python code
  - Can also easily create a class-based API from wrappers of plain C functions
  - The only attribute in the Python proxy is an object that contains a pointer to the real C++ object



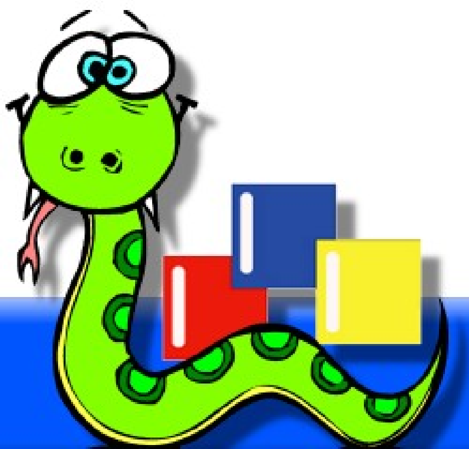
SWIG


theModule.i
SWIG
theModule.py
theModule_wrap.cpp
_theModule.so
theLibrary
C/C++



Why to not use SWIG

- Code generated by SWIG is not the best that an extension module can be.
  - Classes are flattened to a collection of wrapped functions, with the OO-ness added back on by the Proxy classes. A hand-coded extension module would probably just use extension types (classes) instead
  - The flexibility and generality of SWG generated code leads to some inefficiencies that could be avoided with hand-coding
  - Like probably any code generator, the code is messy and not much fun to read, but luckily you seldom need to do so



Why use SWIG

- SWIG saves massive amounts of time and effort
  - wxPython is 1.6 MLOC, 90-95% generated by SWIG, and not only maintained, but actively developed by one person
- Allows developers to partition the application into the languages best suited to performing each task, without needing to spend relatively large amounts of time on the glue code
- Developers can spend more time on the parts of the code that really matter