CIS192 Python Programming
Mixing C with Python/Modules and Packages

Eric Kutschera

University of Pennsylvania

April 24, 2015
Final Project

- Sign up for a demo slot
- Link is on the website for week 14
- Email me if none of the demo slots work for your group
1. Mixing C with Python
   - Python/C API
   - Higher Level Tools

2. Module and Package System
   - Modules
Why Mix C with Python

- Some things are not so easy to do in Pure Python
  - Making system calls
- Some things are not possible
  - Accessing CPython interpreter internals
  - Specifying exact memory layouts of objects
- Some things are Slow in Python
  - Basically everything
- Some problems are already solved with good C code
  - Don’t reinvent the wheel
Potential Disadvantages of C Extensions

- Only work on a specific version of CPython
- Introduce non-Python dependencies
- Platform specific compiled binaries
- Partially why numpy/scipy can be a pain to install
C Extensions from scratch

- The Python/C API allows direct extensions to CPython
  - Very low level CPython details
  - Have to worry about reference counting (garbage collection)
  - “It is intended primarily for creators of those tools, rather than being a recommended way to create your own C extensions.”
    - Python docs
- Allows you to do things not possible in pure Python
  - Define new built-in object types
  - Call C library functions and system calls
- Can be much faster
- CPython specific
  - Not portable across implementations (Jython)
  - Other alternative that are more portable (See later)
#include <Python.h> must be the first line

Defines types/functions used to extend the CPython Interpreter

After that you can include any c libraries

Define Python functions as C functions

  - Must return a PyObject
  - Have to parse Python inputs into C types (str → char*)

Must explicitly define

  - Method table
  - Module initialization
Building the .c

- Need to compile into a binary and tell Python where to find it
- `python setup.py build`
- `(sudo)python setup.py install`

```python
from distutils.core import setup, Extension

module1 = Extension('pycapi',
    sources=['pycapimodule.c'],
    extra_compile_args=['-std=c99'])

setup(name='PycapiPackage', version='1.0',
    description='Demo of Python/C API',
    ext_modules=[module1])
```
1. Mixing C with Python
   - Python/C API
   - Higher Level Tools

2. Module and Package System
   - Modules
Allows loading of C libraries

- `from ctypes import cdll`
- `libc = cdll.LoadLibrary('libc.so.6')`

Once a library is loaded you can call its functions

- `libc.printf(b'%s
', b'a string')`

Some Python types are automatically converted

- `byte strings: s = b'bytes' → char *s = 'bytes';`
- `None: None → NULL`
- `Integers: i = 42 → int i = 42;`

All other types must be constructed with a `ctypes` function

- `ctypes.c_bool(True)`
- `ctypes.c_uint(3)`

Supports Jython and PyPy
C Foreign Function Interface
Aims to just require separate knowledge of C and Python

```python
cffi = cffi.FFI()  # Create a Foreign Function Interface
cffi.cdef('Some C declarations')  # Define the interface
C = cffi.verify('paste your .c file')  # Compiles C
C.some_function(*args)  # Use things in cdef
```

- Can load existing C libraries `cffi.dlopen('lib_name')`
- Pass C objects to functions with `cffi.new('int[50]')`
- Has support for PyPy, Jython, and IronPython
Cython was released 2007 (based on a 2002 project)

Write C extensions to Python mostly with Python

Supports:
- Pure Python
- Python annotated with static types
- Calling C code

Cython Extensions must be compiled
- Allows for static compile time optimizations
- Can’t interpret directly with `$python cython_file.py`

Cython isn’t a perfect implementation of Python
- `a is b`: Identity comparisons are likely to behave strangely
- Introspection of Cython functions not quite right
Cython basic

- cythonhello.pyx
  
  ```python
  print('Hello Cython')
  ```

- setup.py
  
  ```python
  from distutils.core import setup
  from Cython.Build import cythonize

  setup(ext_modules=cythonize('cythonhello.pyx'))
  ```

- `python setup.py build_ext --inplace`

  ```python
  >>> import cythonhello
  Hello Cython
  ```
Other Cython Builds

If you just need the boiler plate setup

- Skip the setup.py

```python
>>> import pyximport; pyximport.install()
>>> import cythonhello
Hello Cython
```

If you want to use C libraries

- Mention the library in setup.py

```python
from distutils.extension import import Extension
ext_modules = [Extension('modname',
                         sources=['modname.pyx'],
                         libraries=['c_lib_name'])
```
Cython Using C functions

- Some C libraries are included with Cython
  - Special `cimport` keyword
  - `from libc.math cimport sin`
  - Converts types for you $\sin(1.5) \rightarrow 0.997...$

- Otherwise link from setup.py
  - import with `cdef` keyword

```python
cdef extern from 'math.h':
    double sin(double x)
```

Eric Kutschera (University of Pennsylvania)  CIS 192  April 24, 2015  15 / 24
Cython Static Types

- Static types annotations `int, double, int[10]` can be used.
- This allows Cython to avoid using a `PyObject` in the `.c`.
- Types can be given to local variables.
  - `cdef int n`
- To give a function input a type.
  - `def my_func(int arg)`
  - The input will be converted from a python type at runtime.
  - An error is thrown if the argument can’t be converted.
- If all variables in a loop are annotated.
  - Compiler can convert the entire loop into C code.
  - Allows the compiler to do lots of fancy tricks.
  - Huge win to annotate entire loop.
Numba Overview

- Numba is an LLVM compiler for Python (Released 2012)
- Designed with Numpy in mind
- Compiles Pure Python to fast machine code
- Even easier to use than Cython

```python
from numba import jit

@jit
def some_function(args):
    ...
```
Both Numba and Cython compile Python to machine code
  - Cython goes through C (gcc)
  - Numba goes through LLVM

Cython pre-compiles / Numba Compiles Just in Time (JIT)

Cython and Numba give similar performance

Modification from Pure Python
  - Cython requires lots of type annotations for best performance
  - Numba uses a single decorator and infers types

Installation
  - Cython requires Python and GCC (easy)
  - Numba requires Numpy and LLVM (pain)

Numba doesn’t support calling arbitrary C code
1. Mixing C with Python
   - Python/C API
   - Higher Level Tools

2. Module and Package System
   - Modules
**Imports Explained**

- **import modulename** does 2 things
  - Looks for the module on the system
  - Binds that module to a name in the local scope

- A module is python file on the system *module.py*
  - Any top-level definition is available after importing a module

- Modules are organized into a package hierarchy (file-system)
  - Any directory on the file-system becomes a package
  - A package can contain an __init__.py
  - When the **import** statement is executed so is __init__.py
  - Packages can contain sub-packages
  - **import pkg.subpkg.mod**
  - Packages are also modules so they can be imported
  - **import pkg**
In order for `import modulename` to succeed
  - The module search must succeed
First `sys.modules` is checked. It caches imported modules
Then `sys.meta_path`, a list of module finders, is used
finders are just functions that can search for modules
The default finder looks in the file-system starting at `sys.path`
Anything that matches the module name gets loaded and bound
When loading a module within a package
  - `import pkg.module`
  - `pkg.__path__` will be checked in addition to `sys.path`
What does `pip install` actually do?
It downloads the source from PyPi (Python Package Index)
Executes some setup code (Compiles or otherwise prepares files)
Copies the resulting package to “pythonX.x/site-packages”
Since “site-packages” is in `sys.path` you can then import it
Virtual Environments

- Python 3.3 introduces a built-in `pyvenv`
- It is a standard library solution similar to `virtualenv`
- The basic idea of both of these tools is to isolate dependencies

```
$ pyvenv venv_dir_name creates a directory
  - Scripts to “turn on” the environment
  - Links to or copies of the Python binaries
  - A place to locally install new packages

$ source venv_dir_name/bin/activate
$ deactivate
```

- “turning on” the venv just changes where python looks
  - Modifies the version of Python in `$ echo "$PATH"`
  - Changes Python specific Environment Variables
Benefits of Virtual Environments

- Allows you to use different versions of Python on one system
- Specify exactly which packages are needed for a project
  - Once everything is install generate a requirements file
  - `$ pip freeze > requirements.txt`
  - Use that file to install packages somewhere else
  - `$ pip install -r requirements.txt`
- Can install packages without root permissions