AMS 595: Fundamentals of Computing (Python Unit)

Lecture 10: Advanced Topics: Performance Optimization; Packaging

Xiangmin Jiao

Stony Brook University

Outline

Basics of Performance Optimization

- Python is interpreted, and it is expected to be performative than compiled languages, such as C/C++
- There are two general approaches to speed up Python
  1. Speed up pure Python code using a just-in-time (JIT) compiler
  2. Use C/C++ for some time-intensive kernels
- An excellent example of the first approach is to use Numba
- An excellent example of the second approach is to use Cython

Speeding Up Python Using Numba

- Numba is an open-source JIT compiler that translates a subset of Python and NumPy code into fast machine code
- The following Jupyter Notebooks are borrowed from official Numba demos
  - Numba Basics
  - Numpy Support in Numba
  - Multithreading with Numba

Using Cython and Other Methods

- Cython is an optimizing static compiler for both Python and extended Cython and can convert python into compiled C code
- Following examples here
- Other Methods
  - SWIG: simplified wrappers and interface generator
  - pybind11: seemless interoperatity between C++11 and Python
  - ctypes: calling functions in shared libraries
  - C-API for writing extension modules in Python or embed Python in C

  See also: Interface with C by Haenel as part of SciPy lecture notes

Outline

More about Modules

- Use "import file_base_name" to access a module
  - Module's name is available as __name__ in the module
  - Separate namespace by default
  - If run as "python modulename.py args" then __name__ is set "to __main__"
- Just like UNIX has a path that it searches, in order, for an executable, python uses this to find modules. Search order:
  - Current directory
  - PYTHONPATH environment variable
  - System-wide python installation default path
- sys.path will show the path

Useful Facts about Modules

- "Private" variables in a module
  - "from modulename import *" imports all variables and functions in the module, except for any names starting with "_"
  - If you change a module, you can re-import it as: "reload(modulename)"
- Python is interpreted, but it creates byte-code files (.pyc instead of .py) when a module is first imported
  - This speeds up the loading of the module—it does not change the speed of execution
  - .pyc is automatically recreated based on the file modification times
  - Note that .pyc files are not, in general, portable

Packaging of Python Modules

- Often you separate a project into multiple files / directories
- Example from python docs:

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Packaging of Python Modules

- You can do:
  - import sound.effects.echo (just get that single module)
    - Access as sound.effects.echo.echofilter(...)
  - from sound.effects import echo
    - Access as echo.echofilter(...)
  - from sound.effects.echo import echofilter (make that specific function available)
    - Access as echofilter(...)
- In order to do from sound.effects import *, we need to define what we mean by all
  - The package should have an __init__.py file—this tells python that a directory contains packages
    - Can be empty, but needs to be present (example with our previous module)
    - Set __all__ to the list of modules that should be imported by default

Packaging of Python Modules

- The main purpose of packaging is to put your python module in a system-wide location for you can import it from other python modules
- If you are just writing a simple script that will not be used by other python modules, then you don't need to worry about packaging— just distribute the source
  - Same for a more complex, but still standalone python program.
  - As seen in the path, there is a system-wide location and user-specific location $\times$ the number of python versions
- Some example cases:
  - You have a single .py (mymodule.py) file that you want people to be able to import.
    - This will be placed in the main library directory, like: /usr/lib/python2.7/site-packages/
    - You can then just import mymodule
  - You have extension modules for your application, and you want to make it easy on the user

Packaging of Python Modules

- There are several different options for packaging your python code for other users
  - Unfortunately, these appear to be in a state of flux at the moment
  - distutils and setup.py makes writing extensions easy
- Main methods (at the moment):
  - distutils: this is part of python (2 and 3)
  - setuptools: newer than distutils, offers more functionality. Introduced easy_install, and setuptools module to import into setup.py

distutils

- Simple example (from their docs):
  - We want to distribute foo.py
  - setup.py:

```
from distutils.core import setup

setup(name='foo',
      version='1.0',
      py_modules=['foo'],)
```

  - Keyword arguments to setup() are metadata and the list of modules that comprise the package
  - Create source distribution: python setup.py sdist
    - Creates a .tar.gz (or .zip)
  - Install the package: python setup.py install
  - Commands exist for creating Windows executable installers, RPMs, ...
    - See python setup.py bdist --help-formats