Porting Radar Simulation Software to Python: A Case Study in the Benefits of Python

Ryan May
Enterprise Electronics Corporation
Norman, OK

1 Introduction

Python, as a dynamic programming language, is frequently used as a "glue" language–the reference being that Python is used to glue together different parts of existing software. As a dynamic language, development time tends to be very quick, at a cost of run-time performance. However, for much of the code that is written, run-time performance is unimportant; therefore a trade off that gives rapid development can produce large gains. This is often true in a research setting, where code turnover is frequent as many different solutions are tried. Part of this utility also comes from Python's feature-filled standard library [1], which helps a developer find needed functionality without spending time searching for options.

As a case study in using Python, this work will examine the process of porting an existing radar simulation software package from straight C to a mixture of C and Python. Various Python modules used in the porting process and their impacts on the size, and more importantly, functionality, of the code base will be discussed. The ease of development for adding these features is also discussed.

2 Simulation Software

The software under study is a package for simulating weather radar data, which uses high resolution (100m grid spacing) numerical simulation output as an input source [2]. The simulator relies on a set of configuration data to control the radar hardware characteristics as well as the operating parameters. The original code was written in C and heavily optimized for run-time performance. However, a single simulation still took several hours on a single processor machine.

Two separate changes in the needs of the software were the motivation behind incorporating the use of Python. The first was a desire to begin using the emulator in a classroom setting. Classroom use by students required a large upgrade to the ease of use so as not to be a support burden for the research staff. To aid in this role, it was recognized that the terminal interactivity (commandline invocation and output to the user) and the configuration file format would need to be improved. The second requirement was the need to upgrade the code to support simulation of dual-polarimetric radars. This required incorporating code for more sophisticated scattering models as well as the capability to select a model at run-time–this is needed to examine the differences that result from different approximations. Generating simulated dual-polarimetric data also required changes to the input and output (I/O) sections of the code to read in new parameters as well as write out new generated fields. A major consideration in switching to Python was the simplicity with which NetCDF files (the format used by the simulator) can be manipulated using Python. All of these changes were constrained to what will be referred to as the "frontend". The heavily optimized computational core (the "backend") of the simulator did not require significant changes, and thus kept mostly unchanged.

The author's recent (at the time) experience with

Python in other domains gave indications that Python was a promising choice for this additional development. Adding to this was the fact that the author had already created a flexible Python module for performing a variety of scattering calculations. Pupynere (or any of the many other Python NetCDF libraries) provided a clear and simple path for upgrading the I/O section of the code. Python's standard library, which contains modules for configuration file parsing, command line parsing, and logging, implied that the usability improvements could be obtained with no need to search for reliable libraries to serve these purposes. The standard library also contains the ctypes module, which served as a straightforward (and included) way to join the new Python front-end to the C-based backend. The availability of many features within the standard library was the final tipping point for deciding to port to Python.

3 Python Standard Library

In addition to language syntax and features, part of the rapid development times in Python comes from its feature-filled standard library [1]. This leads many Python developers to refer to Python as having "batteries included." The standard library gives developers a starting point to look for a given set of functionality: running external processes, working with files and paths, regular expressions, etc. In addition to the benefit as a starting point, which should not be underestimated, the standard library reduces complications from software dependencies. The modules in the standard library can be relied upon to be present; thus no need to test for their presence and alert the user to install if necessary.

Some of the standard library modules used in the simulator include:

- collections - item collections
- itertools - iterating over groups of items
- math - mathematical functions
- warnings - issuing and suppressing warnings
- os.path - path/file-name manipulation
- datetime, time, calendar - dates and times
- ConfigParser - configuration file parsing
- optparse - command-line option parsing
- logging - controlling output
- ctypes - calling into dll/.so libraries

In this work, we examine specifically the use of standard library modules for: command line parsing (optparse), configuration files (ConfigParser), and message logging (logging).

3.1 Configuration Files

One of the most significant upgrades required to improve the program's usability was to improve the configuration file format. An example from the original format is given below:

```
NumSweeps 1
Sweep1NumAz 2
Sweep1AzAngle1 200
Sweep1AzAngle2 345
Sweep1NumEl 1
Sweep1ElAngle1 0.5
Sweep1PRT 0.00098667
Sweep1PulseLength 0.00000157
Sweep1RotationRate 19.2261
Sweep1NumPulses 52
Sweep1GateLength  250
```

Old Configuration Format

While the format is not too difficult to understand at first glance, it is somewhat deceptive. Each line is made up of a string with a value. While the string indicates to the user what the value represents, these strings are not recognized by the program at all–the lines were simply read in a fixed order. Any accidental shuffling of the lines, or adding new ones at the wrong location, would result in cryptic error messages. This is a very unfriendly format for new users, and prone to errors when new files are created. Additionally, it was impossible to have any comments in the file to help explain values and units.

When porting to Python, the standard library's ConfigParser module was selected. [3] This module provides simple and flexible parsing of configuration files. A sample of the new configuration file format is given below:

```
# Scan is made up of a set of sweeps
[Sweep1]
#Degrees
AzAngles = 200, 345

#Degrees
ElAngles = 0.5

#Seconds
PRT = 0.00098667

# seconds
PulseLength = 0.00000157

# deg. per sec.
RotationRate = 19.2261

#pulses per radial
NumPulses = 52

# meters
GateLength = 250
```

New Configuration Format

The new format is much more flexible, allowing for comments as well as having no fixed order for the different entries. Additionally, the configuration file can be broken into separate sections, which can be helpful to organize the file as well as for the programmer to keep the configuration compartmentalized. This helps users by having a much more "human-readable" file. From a programmer's standpoint, the use of ConfigParser yields much simpler and maintainable code. The actual parsing of lines into keys and values is offloaded to the library, so that one only needs to request certain parameters by name. Also, the library supports parsing a flexible (i.e. unknown) number of values as a single parameter, which was not possible with the old format.

The end result of the conversion of the configuration parsing code from plain C to using Python's ConfigParser module was a reduction of 600 lines of code (LOC). The loss in code size represents a sure gain in code maintainability (fewer lines gives fewer opportunities for bugs). The code's extensibility is improved as well since adding support for new parameters is as simple as looking up the new parameter name and having a sensible default if it is not found. Unspecified parameters using a default would have been exceedingly difficult, if not impossible, using the old C-based code. All of these gains together represent a large gain in program usability.

3.2 Message Logging

A second change needed to improve the simulator's usability was to improve the messages displayed to the user. As a large research code base under heavy development, the original code displayed a great deal of output to the console. Such messages would be distracting and/or confusing to the user. The messages provided useful debugging information, however, so simply removing the messages was undesirable. What was needed was a way to turn on the messages when requested but have them off by default.

Python provides the logging module in the standard library [4], which serves the purpose of logging messages. The full framework provides a way of controlling message formatting, messages at different levels (warn, error, debug, etc.), and sending individual messages to a variety of end points (files, screen, email, etc.). Much of the complete functionality was considered overkill for this application. The only features used here were separate message levels (for controlling program verbosity) and optionally sending messages to a file instead of the terminal.

The inclusion of logging messages to a file allows the user to easily collect a log of the debugging messages, which could be submitted as feedback in the event of a problem. This streamlines the process of trying to fix issues that inevitably arise as code gets used more widely under configurations that have not previously been tested.

The change in the code base to utilize logging were a gain of 40 lines of Python code. Additionally, all print statements in the C code became calls to logging functions in Python. For such a small gain in the code base size, useful features for controlling and saving messages were gained, as well as putting more polish on the user-visible face of the program.

3.3 Command Line Parsing

Besides the configuration file, the command line represents the only user interface with the simulator. While the original command line interface worked fine for the original feature set, new options were required to support the logging module. Also, the original help screen rep-

resented essentially a text file embedded into the source. Any changes to the interface required editing code in separate location to ensure that the documentation accurately reflected actual program use.

While the Python standard library's optparse module did not motivate the port to Python, once the port was underway, it was a clear choice for implementing command line parsing. optparse provides a clear advantage in terms of creating code that can easily be extended to support new options [5]. Each option is specified individually, giving:

- long and short command names
- optional default values
- a description of the command (for the program's help message)
- what the parser should do when the option is encountered, such as
  - Store the subsequent string (such as a filename)
  - Store true/false
  - Increment a counter
  - Run a function

From the descriptions of each individual option, the library generates a help screen for the program, which can be displayed by invoking the program with the '-h' option or any other case the programmer wishes.

A typical invocation of the program from the command line, which looks like any typical Linux/UNIX program, is:

```
radarsim -vvv -d -l run.log config/example.config
```

The code used to add an option looks like:

```
opt_parser.add_option("-d", "--detailed",
    action="store_true", dest="detailed",
    help="Use detailed logging messages.")
```

The use of this module increased the size of the code base by 10 lines. (This does not include the old hard-coded help page). This additional code added options for the logging framework that were not available in the old code, as well as provided a generated, UNIX-like help screen. An example of the help screen is given below:

```
Usage: radarsim [options] configfiles

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  -v, --verbose         Produce more verbose messages. Specify more than once
                        for more messages.
  -q, --quiet           Make output messages more quiet. Specify more than
                        once for less output.
  -d, --detailed        Use detailed logging messages.
  -l FILE, --log-output=FILE
                        Log output messages to FILE. Only error messages will
                        be displayed to the console.
  -L, --log-only        Used to specify that output only goes to logfile. Need
                        to also specify --log-output.
```

All of these changes required minimal development effort. It should be noted that as of Python's 2.7 release, the optparse library has been deprecated in favor of the argparse module. This module provides a very similar interface in many respects, but improves handling of required arguments and default parameter values. [5]

4 NetCDF

In addition to the utility of the standard library, the syntax and features of Python permit library developers to create more expressive interfaces than would be possible in may other languages. Much of this stems from Python's dynamic nature and key language features. For instance, Python includes mapping character strings to arbitrary objects (a dictionary) as one of its core data structures; adding such support to user-created data structures is very straightforward. That these mappings can contain arbitrary objects and do not need to be homogeneous is due to Python's dynamic nature.

NetCDF is a specific example of such a library. The available Python interfaces for creating NetCDF files are vastly more simple than the reference C interface. Part of the reason for simplicity is the availability of a de facto standard array type for Python in the form of the NumPy library. NumPy provides a mathematical array type that handles memory allocation and provides array-based mathematical operations [6]. The other reason for the simplicity is again Python's dynamic nature. As a self-describing, flexible file format, the data types for different fields in NetCDF files are not known a priori. Combined with the need to know types when compiling, this leads to the creation of large amount of code (functions, data structures, etc.) to handle the possible cases. A dy-

namic language lends itself much more readily to n such a flexible-typed file format, and this shows in the difference in interfaces.

There exist many Python libraries for reading NetCDF files:

- ScientificPython (Scientific.IO.NetCDF)
- PyNIO
- pupynere
- scipy.io.netcdf
- netcdf4-python

The interface for all of these libraries is nearly identical; ScientificPython's was the first and the others followed the same interface since it proved simple to use. For this project, Pupynere was selected as it is implemented purely in Python [7], making it easy to install as a dependency; there is no need to compile code or install the reference C-based NetCDF library.

As an example of the verbosity in reading a NetCDF file in C, see the listing below, which retrieves an array of values corresponding to the "Temperature" variable:

```
int nc_file, var_id, ndims;
nc_open("Test_data.nc", NC_NOWRITE, &nc_file);

nc_inq_varid(nc_file, "Temperature", &var_id);
nc_inq_varndims(nc_file, var_id, &ndims);
int* dims = malloc(ndims * sizeof(int));
nc_inq_vardimid(nc_file, var_id, dims);

size_t dim_len, total_size = 1;
for(int i=0; i<ndims; ++i)
{
    nc_inq_dimlen(nc_file, dims[i], &dim_len);
    total_size *= dim_len;
}
float* var = malloc(total_size * sizeof(float));
nc_get_var_float(nc_file, var_id, var);

free(dims);
free(var);
nc_close(nc_file);
```

This stands in stark contrast to the equivalent Python code using pupynere:

```
from pupynere import netcdf_file
nc = netcdf_file('Test_data.nc', 'r')

temp_var = nc.variables('Temperature')

nc.close()
```

In the C version of the code, after looking up an ID using the variable's name, a lot of code is spent figuring out how much memory to allocate to hold the array. All of this work is hidden behind a single function call in Python.

Another interesting contrast between the two versions is that to read a different data type (say Temperature stored as integers), the C version would need to be updated; the Python version would look identical to the version above. In order to make a C version that would have the same flexibility, many more lines of code would be need to check types and to allocate a pointer of the appropriate type.

While part of Python's advantage comes from the NumPy library, it would be possible to write such a data structure and interface functions in C (indeed, much of the numpy library is in C). However, this would be burdensome on the developer compared with simply downloading and installing a package. Even if such a C-based library existed, it still would not have the benefit of being integrated with the C NetCDF API like pupynere (and others) integrate with NumPy.

It should be noted that these examples are rather simple, as errors are not handled. However, this again works in favor of Python. In C, the errors are handled by checking return values of each function invocation, and appropriate action taken if the error code is not 0. In Python, the task is simpler since exception handling is available; at the end of the appropriate code block, one must simply catch the appropriate exception. This block can encompass many API calls.

The impact of converting the NetCDF I/O code was a drastic reduction in code volume: 800 lines of code were removed by converting from C to Python. In addition, extending the I/O code was made much simpler, which encouraged some improvements to the metadata which was output in the files. These metadata include the code version, date and time of simulation, and random num-

ber generator seed (for later reproduction and testing of results).

5 Porting and Wrapping existing code

Several methods were considered for wrapping the low-level C code which was to be kept:

- Python C-API
- Ctypes
- F2Py
- Cython

The Python C-API was discounted as being too low-level and fragile, as well as time-consuming to develop and get correct. F2Py's C support was deemed insufficient for the needs of this project, since its main focus was on FORTRAN. Cython, (a fork of the Pyrex project), is a tool to generate C code (utilizing Python's C-API) from a Python-like syntax. This has use both in optimizing Python code as well as interfacing with external libraries. At the time, the project was deemed too immature to be relied upon; since that time the project has gone onto wide success and is used by many scientific Python projects as the method of choice for interfacing Python with compiled code. Ctypes was chosen as the mature solution, having gained acceptance into Python's standard library as of version 2.5. [8]

Ctypes provides a way of opening a shared library (a .so file on UNIX or a .dll on Windows) and obtaining references to the individual functions contained within. To pass the proper data structures to these functions, Ctypes provides a set of classes that map to the various C data types (pointers, int, float, etc.). These low level types can be assembled into full C-style structures. Once a reference to the desired function is obtained, one sets the proper types for the function's arguments as well as its return value. The function is called by creating the proper structures, filling them with information, and passing them to the function. This approach has the unique property of having the entire interface to the compiled code being maintained in Python. It should also be noted that numpy arrays can be passed as pointers to the appropriate data type using Ctypes. A simplified version of some of the simulator's Ctypes code is given below:

```
from ctypes import cdll, Structure, POINTER,
     c_float, c_double, c_int

class RadarStatus(Structure):
    _fields_ = [('time',c_double),
                ('az_pointing',c_float),
                ('cos_az',c_float),
                ('sin_az',c_float),
                ('el_pointing',c_float),
                ('cos_el',c_float),
                ('sin_el',c_float),
                ('cur_az_ind',c_int),
                ('cur_el_ind',c_int)]
from ctypes import cdll
_rslib = cdll.LoadLibrary('_librs.so')

radar_scan = _rslib.RS_radar_scan
radar_scan.argtypes = [POINTER(RadarStatus)]
radar_scan.restype = c_int
```

6 Conclusion

The use of Python enabled the rapid development of the additional required functionality; total development time was approximately 1.5 months for a graduate student working full-time. Through the use of Python's standard library, the required features were added without a large increase in code size (and hence maintenance burden). In fact, porting to Python, in terms of lines of code, actually reduced the size of the code base. The initial size of the code was 5400 lines of C code. The final ported version has 2000 lines of Python and 2900 lines of C (not including the separate library for scattering calculations). This port came with no measurable performance penalty; while the new Python code is no doubt slower, the sections of the code responsible for the overwhelming majority of run time remains in C. This shows the utility of being able to combine the two languages so that one can benefit from their respective strengths.

Porting this software package to Python has demonstrated that Python possesses many attributes that make it easier for developing than in traditional, statically typed languages. Python's standard library offers many built-in

facilities that make accomplishing a task, like parsing a configuration file, a rapid endeavor. Python's 3rd-party support for NetCDF files is a large improvement over the support in C or FORTRAN. The ability to use Python's Ctypes library (or the 3rd party Cython package) to link to existing code allows one to keep performance-critical sections of the code in a language like C, while permitting rapid development in Python for the rest. This allows developers to use the best tools for the job at hand and maximize their development time. For a scientist, this means spending less time developing code and more time applying it to research problems.

References

[1] *The Python Standard Library*, http://docs.python.org/library/index.html

[2] May, Ryan M. and Biggerstaff, Michael I. and Xue, Ming, 2007. *A Doppler Radar Emulator with an Application to the Detectability of Tornadic Signatures*, J. Atmos. Ocean. Tech., **12**, 1973–1996.

[3] *ConfigParser – Configuration file parser*, http://docs.python.org/library/configparser.html

[4] *logging – Logging facility for Python*, http://docs.python.org/library/logging.html

[5] *optparse – Parser for command line options*, http://docs.python.org/library/optparse.html

[6] *NumPy Reference*, http://docs.scipy.org/doc/numpy/reference/

[7] *pupynere 1.0.13*, http://pypi.python.org/pypi/pupynere/

[8] *ctypes – A foreign function library for Python*, http://docs.python.org/library/ctypes.html