Python dispatchers for CCP4

David G. Waterman
CCP4, Research Complex at Harwell, Rutherford Appleton Laboratory, Didcot OX11 0FA, UK

Introduction

Python is a popular high-level language for scripting in science, offering a high degree of portability with concise and clear source code. The CCP4 Suite contains numerous components written in Python. A typical use case consists of Python 'glue', in which a top level Python program acts as a decision-making script, controlling the execution of various lower level programs that do the intensive calculations. This is the basis behind the expert system, of which MrBump[1] and xia2[2] are two prominent examples in CCP4. Another common use of Python is as a language to write graphical user interfaces, *via* extensions such as PyQt or wxPython. The forthcoming CCP4 GUI, CCP4i2[3], is an example of this, replacing Tcl/Tk with Python/PyQt for a more modern and powerful system.

In those cases where Python is used to wrap lower level executables there is a common need to write very similar 'wrapper' code. Typically this makes use of Python's built-in subprocess module. This is relatively straightforward to use, however care must be taken to ensure that it is used in a portable manner across platforms (there are some differences on Windows compared to Unix-like platforms). There are also some limitations to basic use of subprocess. For example, there is no simple interface by which jobs can be started but control returned immediately to the Python script, which is then free to monitor the output of that job, potentially aborting an unsuccessful run before it completes, or to start new jobs to run in parallel. These problems have been addressed multiple times in the CCP4 suite, as the suite lacked a consistent, common Python interface.

From version 6.4.0, the CCP4 suite will be distributed with a Python package called CCP4Dispatchers, containing wrappers for all of the executables in the suite. These dispatchers are designed with a common, simple, but flexible interface that works in the same way across platforms. It is hoped that writing these wrappers once more lifts the need for some boilerplate code in future CCP4 python projects. In addition, the CCP4Dispatchers package contains features that makes it more powerful than basic use of subprocess.

Automatic code generation

The CCP4 suite is under active development. New programs appear, occasionally old programs are removed, change name or are replaced by versions in a different, interpreted, language. The suite also makes relatively heavy use of environment variables. These may change names and values over time (although the recent trend is to pare back the list of CCP4 environment variables, which grew rather unwieldy over the years). The CCP4Dispatchers package must cope with the need to maintain the Python interface in step with the programs and environment. The effort required to do this is minimised by generating the dispatchers automatically as one of the final steps during installation of the suite. This ensures the dispatchers are up to date with the installed programs, and guarantees that a dispatcher is available for every program present at installation. When changes are made to the suite, e.g. by the automatic updater, the dispatcher package can simply be regenerated as this operation is very quick.

The 300+ programs distributed with CCP4 are diverse. Although the majority are native executables, these are supplemented by shell scripts, interpreted languages such as Python, Tcl, Perl or Ruby and

Java bytecode. This must be detected when the dispatchers are generated to create the correct dispatch command, which avoids the need to rely on the shell to interpret the target type. This is also of benefit to the user of the CCP4Dispatchers package, who need not care what the target type is. As long as it is a CCP4 executable located in the $CBIN directory, it should have a usable Python wrapper automatically written for it.

In CCP4 6.4.0, binary distributions of the suite on Mac or Linux will have the CCP4Dispatchers package generated under $CCP4/share/python by the BINARY.setup script. On Windows the package will be distributed pre-generated, just to make things easier for the installer.

Encapsulating the CCP4 environment

An important feature of the CCP4Dispatchers package is to wrap up everything needed to run a CCP4 job, including a correct set of CCP4 environment variables. These are captured when the package is generated and are automatically set when a dispatcher is created in Python code. This has some interesting consequences.

1. It is beneficial to third-party code, which may need to run CCP4 programs but without the inconvenience of having to set up the environment explicitly. It is only required that a CCP4Dispatchers package is placed in Python's module search path, so that it can be imported. Then the environment is handled automatically prior to runtime of the program.

2. It potentially alleviates the requirement to set the CCP4 environment for command line use too. Rather than start the program directly, a user may call the programs *via* their dispatchers, handling the environment set up implicitly just prior to runtime. In fact, a set of symbolic links (or .bat files on Windows), that have the same names as the underlying executables, are generated in a separate directory along with the CCP4Dispatcher package. If this directory is placed in the PATH then no other variables need to set in the shell in order to run any CCP4 programs.

3. Multiple dispatcher packages can be generated with different environments. This provides a means to access multiple versions of the suite without their environments colliding in the shell.

The second two points above hint at more advanced usage of the dispatcher generator. It may be run outside of the suite install process in order to generate packages with custom locations, names and environment definitions.

Custom use

The binary distributions of the suite come with a canonical set of CCP4 dispatchers, located in $CCP4/share/python. However, customised sets of dispatchers may be generated at any time using Python scripts located in $CCP4/libexec. Dispatcher generation is a two stage process. First a file of environment variable definitions must be created. To simplify this procedure the script envExtractor.py can be run passing in a number of POSIX shell scripts used to set up the environment. On Windows the file of environment variable definitions must be created manually, but the syntax is extremely simple. Secondly, dispatcherGenerator.py is run, with minimal input being the file of definitions and a path to an directory of executables to write dispatchers for.

This two stage procedure makes it easy to modify sets of environment variables prior to dispatcher generation. It also means that the dispatcher generator can remain a completely general tool. In fact, not even the name of the output package 'CCP4Dispatchers' is prescribed. By using options of dispatcherGenerator.py it is possible to write a package of Python dispatchers for any set of

executables, with any valid package name, in a user-specified location (usually in PYTHONPATH), with a set of links (or .bat files) in some other location (usually in PATH).

More detailed information about the use of envExtractor.py and dispatcherGenerator.py are given in the documentation at $CCP4/html/CCP4Dispatchers.html.

A simple example

The following code shows how the example script refmac5-simple.exam may be rewritten in Python, using the CCP4Dispatchers:

```
from string import Template
import os
from CCP4Dispatchers import dispatcher_builder

cmd = Template("HKLIN $CEXAM/rnase/rnase18.mtz " + \
               "HKLOUT $CCP4_SCR/rnase_simple_out.mtz " + \
               "XYZIN $CEXAM/rnase/rnase.pdb " + \
               "XYZOUT $CCP4_SCR/rnase_simple_out.pdb")
cmd = cmd.substitute(os.environ)
keywords = """LABIN FP=FNAT SIGFP=SIGFNAT FREE=FreeR_flag
NCYC 10
END
"""

d = dispatcher_builder("refmac5", cmd, keywords)
d.call()
```

Further examples of use are given in the documentation at $CCP4/html/CCP4Dispatchers.html.

Feedback is very welcome!

References

[1] Keegan, R.M. and Winn, M.D. “*MrBUMP*: an automated pipeline for molecular replacement” (2008) *Acta Cryst*. D**64**, 119-124

[2] Winter, G. "xia2: an expert system for macromolecular crystallography data reduction". (2010) *J. Appl. Cryst.* **43**, 186-190

[3] Potterton, L. "CCP4i2 for Programmers". (2012) *CCP4 Newsletter* **49**.