Improving Software Installation Techniques at the National Center for Computational Sciences at Oak Ridge National Laboratory: The Smithy software installation tool.

Anthony DiGirolamo
National Center for Computational Sciences
Oak Ridge National Laboratory
1 Bethel Valley Rd
Oak Ridge, TN USA
lmd@ornl.gov

Robert D. French
National Center for Computational Sciences
Oak Ridge National Laboratory
1 Bethel Valley Rd
Oak Ridge, TN USA
frenchrd@ornl.gov

ABSTRACT

Smithy is a software compilation and installation tool that borrows ideas from SWTools[3] and the homebrew[2] package management system for Mac OS X. Smithy is designed to manage many software builds within an HPC Linux environment using modulefiles to load software into a user's shell. SWTools has set very good conventions for software installations at the NCCS. Smithy's goal is to make following the SWtools conventions easier and less error prone. Smithy replaces SWTools and improves upon it by providing a simpler command line interface, modulefile generation and management, and installations via formulas written in Ruby.

1. INTRODUCTION

The sane management of third-party software installation requests is a vital component of the user experience at any high-performance computing center. Users expect installation requests to be handled promptly, and for required software to be available on each computing resource they use. Unfortunately for the software administrator, the disparity between computing resources can mean hours of work deducing appropriate configuration options, testing for correct behavior, or even crafting patches to support esoteric systems.

SWTools[3] aided the software administrator by providing tools and conventions to facilitate reproducible software builds. Information specific to the configuration of a build could be kept in *rebuild scripts* that would house the information necessary to recompile and reinstall a package. However, SWTools is very specific in its aims: a single rebuild script supports one set of configuration options for a particular platform. This can be a hindrance, for example, for packages like Python Numpy, which can target multiple major versions of Python (2.X and 3.X) as well as multiple BLAS implementations. Supporting all combinations of Python and BLAS for a single compute resource can require many rebuild scripts, each only marginally dissimilar from the others.

Smithy improves over SWTools primarily in this regard. Drawing inspiration from Homebrew[2], a package manager for Mac OS X, Smithy encapsulates everything necessary for reproducible builds in a single ruby program called a "formula". Formulas can be written in a generalized fashion to support multiple software versions or build platforms; they can also be extended using ruby's "concerns" mechanism to support separate configurations per version or per platform when necessary.

2. INSTALLATION AND CONFIGURATION

Installation is a simple download and extract anywhere affair. Smithy comes bundled with its own x86, x86_64, and Mac OS X builds of Ruby that will run on most machines. See the releases page for download tar files at (github.com/AnthonyDiGirolamo/smithy/releases)

Smithy is designed to manage many software installations across many hosts and architectures. To support this it must be configured via a yaml formatted config file. As an example:

```
---
software-root: /sw
hostname-architectures:
  rhea:        redhat6
  titan-login: xk7
  titan-ext:   xk7
file-group-name: ccsstaff
formula-directories:
- /sw/tools/smithy/formulas
download-cache: /sw/sources
```

The above lines and many more configuration options are documented on the smithy manpage at (anthonydigirolamo.github.io/smithy/smithy.1.html). Notable are the `software-root` and `hostname-architectures` options. `software-root` is the root directory where all software will be installed. `hostname-architectures` maps machine hostnames to specific architecture directory names. For example when smithy is run on a machine with the

hostname `titan-ext1` it will place installations within the `/sw/xk7` directory.

3. FORMULAS

Smithy formulas are Ruby programs that deduce platform-specific build settings. They can manipulate the programming environment and load or unload modules as appropriate. Formulas can express dependency requirements, and even generate modulefiles for newly installed packages.

Formulas are installed with an SWTools target name. Targets are formatted as `name/version/build_name` and correspond to the directory structure used to install the software on the filesystem. For example a target can be specified such as:

```
python_numpy/1.9.2/python2.7_sles11.1_acml
```

This tells smithy to load the `python_numpy` formula, and build version `1.9.2` targeting Python2.7 on SuSE Linux Enterprise Server version 11.1 using AMD Core Math Library. This is a suitable target for Titan, a Cray XK7 running SLES 11.1 with AMD Opteron CPUs. When installed Smithy will layout the directory structure for the above target as:

```
/sw/xk7
\___ python_numpy
   \___ 1.9.2
      \___ python2.7_sles11.1_acml
         |--- bin
         |--- lib
         \___ source
```

The root location for all software installations is `/sw` and all software installations for Titan are installed under the `xk7` architecture subdirectory.

To install NumPy on Rhea, an NCCS analysis cluster running RedHat 6.5 on Intel Xeon CPUs, Smithy can be invoked with the target:

```
python_numpy/1.9.2/python2.7_rhel6.5_mkl
```

Which corresponds to the following directory structure:

```
/sw/rhea
\___ python_numpy
   \___ 1.9.2
      \___ python2.7_rhel6.5_mkl
         |--- bin
         |--- lib
         \___ source
```

Smithy will invoke the same `python_numpy` formula as before, but this time targeting RHEL 6.5 and the Intel Math Kernel Library.

As the configuration options and strategy for building numpy are largely the same between these two cases, the specifics needed to handle each can be expressed in conditional blocks if they are simple, or in "concerns" if they are more complex. This allows the software administrator to maintain portable build instructions for dissimilar platforms in a single file.

When we say portable we mean that Smithy formulas should be shareable between dissimilar architectures (Cray XK7, XE6, and XC-30) as well as related x86-Linux clusters. That is, one formula to capture all installation caveats of a package within the center.

See Figure 2 and Figure 3 for the complete `python_numpy` formula.

A further advantage of Smithy formulas over SWTools rebuild scripts, inspired by Homebrew formulas, is that they can be shared through public version-control hosting sites such as GitHub. HPC centers can choose to maintain their own formulas, pull changes downstream from the official Smithy formulas repository, or contribute additional improvements upstream.

3.1 Formula Basics

Formulas are written as Ruby classes and follow a simple structure. They provide a domain specific language (DSL) with the support of a full programming language. The structure of a formula is the same as a Ruby class. Here is a bare minimum example:

```ruby
class ExampleFormula < Formula
  homepage "http://example.com"
  url "http://example.com/example-1.2.3.tar.gz"
  md5 "44d667c142d7cda120332623eab69f40"

  def install
    system "./configure --prefix=#{prefix}"
    system "make install"
  end
end
```

The above is a working example of a formula written in the Ruby programming language. All examples shown are syntactically vaild Ruby. No pseudo code is used. The above formula can be installed via the command line by running:

Using the Ruby language to write formulas has been somewhat of a challenge at our center. To help address that we have included links and general Ruby programming information in the (Smithy formula writing documentation).

```
smithy formula install example/1.2.3/gnu4.3.4
```

Assuming that this is run on Titan, smithy will download the `example-1.2.3.tar.gz` from the supplied url, check its md5 sum, extract it to `/sw/xk7/example/1.2.3/gnu4.3.4/source`, switch to that working directory, and run the commands specified via the `system` method. Namely:

```
./configure \\
  --prefix=/sw/xk7/example/1.2.3/gnu4.3.4
make install
```

The `homepage`, `url`, and `md5` DSL methods take a string as their argument. The `def install...end` section is a Ruby method definition that will be run after Smithy extracts the software and sets up the shell environment. Inside that method any arbitrary Ruby code can be added to

perform the installation. The `system` method will run shell commands and the `prefix` method is a string variable that Smithy will populate with the full installation path based on the target name supplied to the `smithy formula install` command.

3.2 Formula Helpers

Formulas have a wealth of additional helper features each of which is documented on the formula writing manpage at (anthonydigirolamo.github.io/smithy/smithyformula.5.html). Here we will highlight some of the most useful ones.

3.2.1 `concern for_version()`

Concerns are used allow formulas to support multiple versions. When used, their contents will override the same methods defined at the root level of a formula class. To specify a concern for version 1.2.3 of a package add the following to a formula (replacing the comment with the methods you would like to override).

```
concern for_version("1.2.3") do
  included do
    # override methods here
  end
end
```

Any formula DSL method can be overridden including `def install`.

Here is an example of a python formula that supports version "2.7.9" and "3.4.3". This example only overrides the download url and md5.

```
class PythonFormula < Formula
  homepage "www.python.org/"

  depends_on "sqlite"

  module_commands ["unload python"]

  concern for_version("2.7.9") do
    included do
      url "https://www.python.org/"
          "ftp/python/2.7.9/Python-2.7.9.tgz"
      md5 "5eebcaa0030dc4061156d3429657fb83"
    end
  end

  concern for_version("3.4.3") do
    included do
      url "https://www.python.org/"
          "ftp/python/3.4.3/Python-3.4.3.tgz"
      md5 "4281ff86778db65892c05151d5de738d"
    end
  end

  def install
    module_list
    ENV["CPPFLAGS"] =
      "-I#{sqlite.prefix}/include"
    ENV["LDFLAGS"] =
      "-L#{sqlite.prefix}/lib"
      system "./configure --prefix=#{prefix}",
             "--enable-shared"
      system "make install"
    end
  end
```

Using the above formula it is possible to install python with the following and smithy will download the correct tarball for each version:

```
smithy formula install \\
  python/2.7.9/sles11.3_gnu4.3.4
smithy formula install \\
  python/3.4.3/sles11.3_gnu4.3.4
```

3.2.2 `depends_on`

This method expects either a single string or an array of strings that define dependencies for this formula. Here are some examples:

```
depends_on "curl"
depends_on [
  "cmake",
  "qt",
  "openssl",
  "sqlite" ]
```

Using this method ensures that if a given dependency is not met smithy will abort the installation. It also provides a way to query dependent packages information within the install method later on. For example if you write `depends_on "curl"` in your formula you gain access to an object named curl inside the install method. This allows you to do things like:

```
system "./configure --prefix=#{prefix}",
       "--with-curl=#{curl.prefix}"
```

In the above example `#{curl.prefix}` is an example of a ruby interpolated string, everything between the `#{ }` is ruby code. `curl.prefix` will return a string with the location curl is installed in.

The strings passed to `depends_on` are just the locations of installed software. If you required a specific version of a dependency you could specify the version or build numbers of existing installed software. For example:

```
depends_on [
  "cmake/2.8.11.2/sles11.1_gnu4.3.4",
  "qt/4.8.5",
  "sqlite" ]
```

Assuming your software root is `/sw/xk7` smithy would look for the above software installs in `/sw/xk7/cmake/2.8.11.2/sles11.1_gnu4.3.4`, `/sw/xk7/qt/4.8.5/*`, and `/sw/xk7/sqlite/*/*`. The wildcard `*` works similar to shell globbing. If you needed to install a python module that depends on a specific version of another python module you might use:

```
depends_on [ "python/3.3.0",
```

```
    "python_numpy/1.7.1/*python3.3.0*" ]
```

This would require a given formula to have access to both `/sw/xk7/python/3.3.0/*` and a python module with a build name that includes `python3.3.0` located at `/sw/xk7/python_numpy/1.7.1/*python3.3.0*`

You may also need to specify dependencies conditionally upon the type of build you are performing. If you add the type of build to the `build_name` when installing you can key off that to specify dependencies. Taking the python example further, lets extend it to support multiple versions of python. You can pass a ruby block to the `depends_on` method to make it more dynamic. The syntax for this is:

```
depends_on do
  # ...
end
```

Any ruby code may go in here the last executed line of the block should be an array of strings containing the dependencies. Lets use a ruby case statement for this:

```
depends_on do
  case build_name
  when /python3.3/
    [ "python/3.3.0",
      "python_numpy/1.7.1/*python3.3.0*" ]
  when /python2.7/
    [ "python/2.7.3",
      "python_numpy/1.7.1/*python2.7.3*" ]
  end
end
```

In this example case statement switches on the `build_name`. The `when /python3.3/` will be true if the `build_name` contains the `python3.3`. The `/python3.3/` syntax is a regular expression.

This allows the formula to set it's dependencies based off the type of build thats being performed. Lets say this formula is `python_matplotlib`. You could run either of these commands to install it and expect the dependencies to be set correctly:

```
smithy formula install \\
  python_matplotlib/1.2.3/python3.3.0
smithy formula install \\
  python_matplotlib/1.2.3/python2.7.3
```

3.2.3 `module_commands`

This method defines the module commands that must be run before `system` calls within the `def install` part of the modulefile. Similar to `depends_on` it expects an array of strings with each string being a module command. For example:

```
module_commands [ "load szip", "load hdf5" ]
```

A more complicated example:

```
module_commands [
  "unload PE-gnu PE-pgi PE-intel PE-cray",
  "load PE-gnu",
  "load cmake/2.8.11.2",
  "load git",
  "swap gcc gcc/4.7.1",
  "swap ompi ompi/1.6.3"
]
```

`module_commands` also accepts ruby blocks the syntax for this is:

```
module_commands do
  # ...
end
```

This can be used to dynamically set which modules to load based on the `build_name`. Here is an example that loads the correct python version:

```
module_commands do
  commands = [ "unload python" ]

  case build_name
  when /python3.3/
    commands << "load python/3.3.0"
  when /python2.7/
    commands << "load python/2.7.3"
  end

  commands << "load python_numpy"
  commands << "load szip"
  commands << "load hdf5/1.8.8"
  commands
end
```

This block starts by creating a variable named `commands` as an array with a single item `"unload python"`. Next a case statement is used to determine which version of python we are compiling for. `commands << "load python/3.3.0"` will append `"load python/3.3.0"` to the end of the array. After that, it appends a few more modules to load. The last line of the block must be the array itself so that when the block is evaluated by smithy, it recieves the expected value.

Assuming this is a formula for `python_h5py` running `smithy formula install python_h5py/2.1.3/python3.3` results in an array containing:

```
[ "unload python",
  "load python/3.3.0",
  "load python\_numpy",
  "load szip",
  "load hdf5/1.8.8" ]
```

3.2.4 `modulefile`

The modulefile command provides a way to define a complete modulefile within the formula itself. It's recommended to have one modulefile per application and version combination.

Writing modulefiles is a topic in and of itself. For details on the modulefile format see the modulefile manpage Module-

files are written in tcl and can take many forms.

Here is an example of a modulefile that points to a single build:

```
modulefile <<-MODULEFILE
  #%Module
  proc ModulesHelp { } {
    puts stderr \
    "<%= @package.name %> <%= @package.version %>"
    puts stderr ""
  }
  module-whatis \
    "<%= @package.name %> <%= @package.version %>"

  set PREFIX <%= @package.prefix %>

  prepend-path PATH            $PREFIX/bin
  prepend-path LD_LIBRARY_PATH $PREFIX/lib
  prepend-path MANPATH         $PREFIX/share/man
MODULEFILE
```

The <<-MODULEFILE syntax denotes the beginning of a multi-line string. The string ends with MODULEFILE. You can substitute any word for MODULEFILE.

The modulefile definition uses the erb format. Anything between the <%= ... %> delimiters will be interpreted as ruby code. There are a few helper methods that you can use inside these delimiters see the Modulefile Helper Methods section of the smithyformula manpage. Modulefile Helper Methods section of the smithyformula manpage for details.

A more complicated modulefile may examine a user's programming environment to determine which build to load. For instance if the user has gcc or a gnu programming environment module loaded then your modulefile will want to load the build compiled with the gcc compiler. Here is an example designed to dynamically set the build:

```
#%Module
proc ModulesHelp { } {
  puts stderr \
  "<%= @package.name %> <%= @package.version %>"
  puts stderr ""
}
# One line description
module-whatis \
  "<%= @package.name %> <%= @package.version %>"

<% if @builds.size > 1 %>
<%= module_build_list @package, @builds %>

set PREFIX \
  <%= @package.version_directory %>/$BUILD
<% else %>
set PREFIX <%= @package.prefix %>
<% end %>

# Helpful ENV Vars
setenv <%= @package.name.upcase %>_DIR \
  $PREFIX
setenv <%= @package.name.upcase %>_LIB \
  "-L$PREFIX/lib"
setenv <%= @package.name.upcase %>_INC \
  "-I$PREFIX/include"

# Common Paths
prepend-path PATH            $PREFIX/bin
prepend-path LD_LIBRARY_PATH $PREFIX/lib
prepend-path MANPATH         $PREFIX/share/man
prepend-path INFOPATH        $PREFIX/info
prepend-path PKG_CONFIG_PATH $PREFIX/lib/pkgconfig
```

The main difference from the first example is the <%= if @builds.size > 1 %> block. This basically checks to see if we have installed multiple builds or not. If that condition is true everything up until the <% else %> will be put in the modulefile including the output from the module_build_list method. Otherwise, if we have only one build, set PREFIX <%= @package.prefix %> will be put in the modulefile.

3.2.5 module_build_list

This is a helper method that will generate the TCL necessary to conditionally load builds based on what compiler programming environment modules a user has loaded. It takes @package and @builds as arguments. Here is an example of the output generated for a package installed with the following builds:

```
smithy formula install bzip2/1.0.4/gnu4.3.4
smithy formula install bzip2/1.0.4/gnu4.7.2
smithy formula install bzip2/1.0.4/pgi13.4
smithy formula install bzip2/1.0.4/intel12

if [ is-loaded PrgEnv-gnu ] {
  if [ is-loaded gcc/4.3.4 ] {
    set BUILD gnu4.3.4
  } elseif [ is-loaded gcc/4.7.2 ] {
    set BUILD gnu4.7.2
  } else {
    set BUILD gnu4.7.2
  }
} elseif [ is-loaded PrgEnv-pgi ] {
  set BUILD pgi13.4
} elseif [ is-loaded PrgEnv-intel ] {
  set BUILD intel12
}
if {![info exists BUILD]} {
  puts stderr "[module-info name] is only
  available for the following environments:"
  puts stderr "gnu4.3.4"
  puts stderr "gnu4.7.2"
  puts stderr "intel12"
  puts stderr "pgi13.4"
  break
}
```

4. SIMILAR EFFORTS

Smithy is not the only tool aiming to facilitate portable software builds. Some other tools include EasyBuild[1], Spack, and HashDist.

4.1 Smithy's Goals

Smithy was designed to support building software on Cray systems and related clusters. One of the main challenges of supporting software on Cray systems is dealing with complex Cray-specific build environments that can require cross-compilation and restrict the usage of shared libraries. Another challenge is the generation of environment modulefiles that are aware of compiler and MPI environments. We have not found this combination of features in other tools.

In our experience, software packages which do not specifically mention Cray support do not necessarily build and install correctly without modification. This is where using a full programming language such as Ruby helps in terms of patching, configuration, or writing out Cray-specific files in a readable and organized way. The advantage of using Ruby over markup formats such as YAML (as in HashDist) is the ability to make complex choices about how to satisfy dependencies at build time. This is also a strength of Spack and EasyBuild's use of Python.

It is not clear that other tools such as HashDist or Spack provide tools or DSL extensions to allow users to easily target Cray systems or to generate robust modulefiles. EasyBuild does have support for modulefile generation, but not support for Cray specific build requirements.

4.2 EasyBuild

EasyBuild supports a wide variety of scientific applications, over 500 packages in total. The Smithy formulas repository (github.com/olcf/smithy_formulas) contains formulas for over 150 packages written by multiple staff memebers including PETSc, OpenMPI, NetCDF, Magma, a large collection of host-native Python modules including NumPy and SciPy, as well as several domain-specific packages such as LAMMPS, NAMD, and Gromacs.

While far short of the number boasted by EasyBuild, Smithy formulas excel over Easybuild's "easyblocks" in two regards: they emphasize support for esoteric platforms such as Crays, and since formulas are contained in a single file, they are easier to write and maintain.

4.3 Spack

Spack is very close to Smithy but implemented in Python. It has a couple benefits over Smithy such as auto dependency resolution and pre-bundled formulas that are generally installable on most Linux systems. Smithy does not come with formulas by default but the NCCS publishes it's formulas at github.com/olcf/smithy_formulas. contains formulas for over 150 packages written by multiple staff memebers Some of the formulas there are very specific to our systems and target Cray environments with Nvidia GPUs.

The way Spack solves dependencies between software it builds is through setting the RPATH for each binary and library it builds. This approach works well in a clustered environment where installed software is accessible by every compute node. It also has the benefit of not requiring the user to load environment modules that modify the `$PATH` and `$LD_LIBRARY_PATH` variables in their shell. The RPATH method does not work well for us at the NCCS because often our modulefiles need to make decicions about what directory should be added to the `$LD_LIBRARY_PATH` depending on the host it is loaded on. Our software installation filesystem is not accessible from our compute nodes.

4.4 HashDist

is cool?

5. CONCLUSIONS

Smithy formulas provide a self-contained encapsulation of all logic necessary to build multiple versions of a scientific software package for various resources, operating systems, and programming environments. A software administrator employing smithy can support build requests for many disparate compute resources within the same center. The concise nature of the Ruby programming language allows formula authors to spare themselves the tedium of repetitive software build tasks. The formula helper methods provided by the Smithy formula DSL facilitate dealing with the caveats of deploying software to these environments.

Formulas are not a new idea in package management, they are similar to Homebrew formulas with additional functionality to support HPC specific requirements such as multiple compilers and environment modulefiles. They are analogous to Spack's packages or EasyBuild's EasyBlocks, with the advantage that all logic needed for an installation is contained in a single file. When managing multiple systems of varying architectures, with multiple compiler environments per system, minimizing the number of files to manage is a boon to the software maintainer's sanity.

Although Smithy does not support as many packages as EasyBuild at the time of writing, it is the authors' belief that Smithy's ease of adoption and rich feature set will allow HPC centers to collaborate, grow the number of packages, and maintain the existing ones.

6. ACKNOWLEDGMENTS

The authors would like to thank the US Department of Energy's Office of Science's Advanced Scientific Computing Research program.

7. FURTHER READING

The Smithy Homepage (anthonydigirolamo.github.io/smithy) has information on installing, configuring and running smithy on your system. In addition there are two in depth manpages that cover using smithy on the command line (anthonydigirolamo.github.io/smithy/smithy.1.html) and formula writing (anthonydigirolamo.github.io/smithy/smithyformula.5.html).

8. REFERENCES

[1] K. Hoste, J. Timmerman, A. Georges, and S. D. Weirdt. Easybuild: Building software with ease. In *High Performance Computing, Networking, Storage and Analysis (SCC)*. Ghent University, November 2012.

[2] M. Howell and et al. Homebrew. the missing package manager for os x. http://brew.sh/, 2015. Accessed: 2015-04-16.

[3] N. Jones and M. R. Fahey. Design, implementation, and experiences of third-party software administration at the ornl nccs. In *Proceedings of the 50th Cray User Group (CUG08)*. Oak Ridge National Laboratory, May 2008.

APPENDIX

```
class GitFormula < Formula
  homepage "https://git-core.googlecode.com/"
  url      "https://github.com/git/git/archive/v2.3.2.tar.gz"
  sha256   "7d8e15a2f41b8d6c391e527f461d61027cf3391c9ccc89b8c1a1a0785f18a0fb"

  depends_on ["curl/7.39.0", "zlib"]

  def install
    module_list
    system "make configure"
    system "./configure --prefix=#{prefix}",
           "--with-curl=#{curl.prefix}",
           "--with-zlib=#{zlib.prefix}"
    system "make install"

    system "mkdir -p #{prefix}/share/man"
    system "curl -O",
      "https://www.kernel.org/pub/software/scm/git/git-manpages-2.3.2.tar.gz"
    system "cd #{prefix}/share/man && ",
      "tar xf #{prefix}/source/git-manpages-2.3.2.tar.gz"
  end

  modulefile do
    <<-MODULEFILE
      #%Module
      proc ModulesHelp { } {
         puts stderr "<%= @package.name %> <%= @package.version %>"
         puts stderr ""
      }
      # One line description
      module-whatis "<%= @package.name %> <%= @package.version %>"

      set PREFIX <%= @package.prefix_directory %>

      prepend-path PATH      $PREFIX/bin
      prepend-path PERL5LIB  $PREFIX/lib64/perl5/site_perl
      prepend-path MANPATH   $PREFIX/share/man
      setenv       GITDIR    $PREFIX
    MODULEFILE
  end
end
```

Figure 1: Example Git Formula

```
class PythonNumpyFormula < Formula
  homepage "http://www.numpy.org/"
  additional_software_roots [ config_value("lustre-software-root").fetch(hostname) ]

  supported_build_names /python.*_gnu.*/

  concern for_version("1.8.0") do
    included do
      url "http://downloads.sourceforge.net/project/numpy/NumPy/1.8.0/numpy-1.8.0.tar.gz"
    end
  end

  concern for_version("1.9.2") do
    included do
      url "http://downloads.sourceforge.net/project/numpy/NumPy/1.9.2/numpy-1.9.2.tar.gz"
    end
  end

  depends_on do
    [ python_module_from_build_name, "cblas/20110120/*acml*" ]
  end

  module_commands do
    pe = "PE-"
    pe = "PrgEnv-" if cray_system?

    commands = [ "unload #{pe}gnu #{pe}pgi #{pe}cray #{pe}intel" ]
    commands << "load #{pe}gnu"
    commands << "swap gcc gcc/#{$1}" if build_name =~ /gnu([\d\.]+)/

    commands << "load acml"
    commands << "unload python"
    commands << "load #{python_module_from_build_name}"

    commands
  end

  def install
    module_list

    acml_prefix = module_environment_variable("acml", "ACML_BASE_DIR")
    acml_prefix += "/gfortran64"

    FileUtils.mkdir_p "#{prefix}/lib"
    FileUtils.cp "#{cblas.prefix}/lib/libcblas.a", "#{prefix}/lib", verbose: true
    FileUtils.cp "#{acml_prefix}/lib/libacml.a",   "#{prefix}/lib", verbose: true
    FileUtils.cp "#{acml_prefix}/lib/libacml.so",  "#{prefix}/lib", verbose: true

    ENV['CC']  = 'gcc'
    ENV['CXX'] = 'g++'
    ENV['OPT'] = '-O3 -funroll-all-loops'
```

Figure 2: Example Python NumPy Formula (1 of 2)

```
52      File.open("site.cfg", "w+") do |f|
53        f.write <<-EOF.strip_heredoc
54          [blas]
55          blas_libs = cblas, acml
56          library_dirs = #{prefix}/lib
57          include_dirs = #{cblas.prefix}/include
58
59          [lapack]
60          language = f77
61          lapack_libs = acml
62          library_dirs = #{acml_prefix}/lib
63          include_dirs = #{acml_prefix}/include
64
65          [fftw]
66          libraries = fftw3
67          library_dirs = /opt/fftw/3.3.0.1/x86_64/lib
68          include_dirs = /opt/fftw/3.3.0.1/x86_64/include
69        EOF
70      end
71
72      system "cat site.cfg"
73
74      system_python "setup.py build"
75      system_python "setup.py install --prefix=#{prefix} --compile"
76    end
77
78    modulefile do
79      <<-MODULEFILE
80        #%Module
81        proc ModulesHelp { } {
82           puts stderr "<%= @package.name %> <%= @package.version %>"
83           puts stderr ""
84        }
85        # One line description
86        module-whatis "<%= @package.name %> <%= @package.version %>"
87
88        prereq PrgEnv-gnu PE-gnu
89        prereq python
90        conflict python_numpy
91
92        <%= python_module_build_list @package, @builds %>
93        set PREFIX <%= @package.version_directory %>/$BUILD
94
95        set LUSTREPREFIX \
96          #{additional_software_roots.first}/#{arch}/<%= @package.name %>/<%= @package.version %>/$BUILD
97
98        prepend-path LD_LIBRARY_PATH $LUSTREPREFIX/lib
99        prepend-path LD_LIBRARY_PATH $LUSTREPREFIX/lib64
100       prepend-path PYTHONPATH      $LUSTREPREFIX/lib/$LIBDIR/site-packages
101       prepend-path PYTHONPATH      $LUSTREPREFIX/lib64/$LIBDIR/site-packages
102
103       prepend-path PATH            $PREFIX/bin
104       prepend-path LD_LIBRARY_PATH $PREFIX/lib
105       prepend-path LD_LIBRARY_PATH $PREFIX/lib64
106       prepend-path LD_LIBRARY_PATH /opt/gcc/4.8.2/snos/lib64
107       prepend-path LD_LIBRARY_PATH /ccs/compilers/gcc/rhel6-x86_64/4.8.2/lib
108       prepend-path LD_LIBRARY_PATH /ccs/compilers/gcc/rhel6-x86_64/4.8.2/lib64
109       prepend-path MANPATH         $PREFIX/share/man
110
111       prepend-path PYTHONPATH      $PREFIX/lib/$LIBDIR/site-packages
112       prepend-path PYTHONPATH      $PREFIX/lib64/$LIBDIR/site-packages
113     MODULEFILE
114   end
115 end
```

Figure 3: Example Python NumPy Formula (2 of 2)