Building Packages for Releases

This page describes how all software packages are built, using the `pspackage` class. But if you are building a Python package, see Building Python Packages as well. And if you are building a package that uses the "./configure; make; make install" pattern, see Building Unix Packages.

When the packages for a release are built, **ps_make** does the following:

- It reads the list of packages in the release specification file;
- It adds to its internal list all packages that the specified packages depend upon;
- It verifies that the requested versions of the packages are consistent;
- It determines which packages have already been built and don't need to be rebuilt;
- It orders the unbuilt packages in dependency order (so dependencies are built first);
- And finally it iterates over the sorted list and builds each package.

If an individual package build fails, **ps_make** simply goes to the next package... unless that package is missing a dependency (due to a failed build).

The pspackage class

The **ps_make** program (and others) use the `pspackage` class to represent information about each package, including:

- name (e.g. `'atlas'`)
- version (e.g. `'3.8.3'`)
- dependencies (e.g. `'lapack/3.2.1'`)
- relative and absolute install dirs

To build an individual package, the **ps_make** program calls `package.build_and_install()`. This method does the following:

- Sets paths and other environment variables.
- Creates working directories.
- Calls the `package.do_build_and_install()` method – this does all the actual compiling and building.
- Fixes `RPATH`, symbolic links to absolute paths, etc. to fix relocation problems.
- Runs basic tests for the package, if any.
- Finally, if all succeeds, a special file (e.g. `'.atlas_is_installed'`) is created in the install directory. If this file is not present during the next run of **ps_make**, it will assume the package was not successfully built and will attempt to rebuild it.

The `pspackage` class provides some methods that it doesn't use directly but are for use by subclasses. The most important of these is `package.extract()`. This method looks for a corresponding zip, bzip2, or compressed tar file in the sources/tarball directory and uses the corresponding tool to extract the sources. It assumes that the tarball (zip file, etc) starts with the package name.

Helper modules

There are some helper Python modules that provide small wrappers around the Python `os` and `subprocess` modules.

The psrun module

The psrun module has one function:

```
run(command, continue_on_error = False)
```

which can be run in two ways.

In the first case, we are running the command as one does in a shell script without redirection. We don't need to capture the output; it can be simply printed to the console. We expect the command to succeed; if it doesn't, we want an exception to be thrown. In this case, we will do e.g.:

```
psrun.run('make all')
```

Note that in this case the command may be long running. That's part of the reason for letting output be printed directly to the console. If we save the output in a Python variable, then (1) the Python process might run out of memory, and (2) there will be no indication from `ps_build` that anything is happening.

In the second case, we are running the command so that we can process its output. In this case, we will do e.g.:

```
(retcode, error, output) = psrun.run('uname -s', True)
```

In this case, if the command fails, we don't want to throw an exception, as we may have alternative commands to try. Instead, we will test the value of retcode. If it is zero then the command succeeded.

The psenv module

The `psenv` module provides the following functions:

- set(var, dir): sets the env variable var to just dir (using os.environvar = dir)
- unset(var): unsets var (using del os.environvar)
- get(var): if var is set, return its value; otherwise, return the empty string

It also provides the following functions for dealing with colon-separated paths (e.g. `PATH`, `PYTHONPATH`, and `LD_LIBRARY_PATH`).

- add_to_front(var, val): if val is not in var, then add it to the front. If it is in var but is not at the front, move it to the front.
- add_to_back(var, val): if val is not in var, then add it to the back. If it is in var but is not at the back, move it to the back.
- reset_paths(): set `PATH` to the minimalist `'/usr/local/bin:/usr/bin:/bin'` and unset PYTHONPATH and LD_LIBRARY_PATH.

The `psfs` module

The `psfs` module provides the following functions:

- chdir(): does os.chdir() but with a message to output
- mkdir(): creates a directory, including required parent directories, if the directory does not already exist.
- freebytes(path): uses os.statvfs(), f_frsize, f_bavail to calculate the free space in bytes in the filesystem where path lives.
- pwd(): does os.getcwd()
- join(): does os.path.join()
- copy_from_slac(src, dst): does an rsync to copy files from SLAC (src) to the local host (dst).
- copy_to_slac(src,dst): does an rsync to copy files from the localhost (src) back to SLAC (dst).
- list_from_slac(dir): returns a list of the files in the specified directory at SLAC.
- slac_has_file(filename): checks if SLAC has a file with the specified name and returns True if it exists and False otherwise.