



Writing Easyblocks: The Basics

Kenneth Hoste
kenneth.hoste@ugent.be

EasyBuild hackathon - Basel, 20150211

What is an easyblock?

- Python module that implements a software build procedure
- 'plugin' for the EasyBuild framework
- can be *generic* (using standard tools) or *software-specific*
- lives (somewhere) in the `easybuild.easyblocks` namespace
- own easyblocks can live next to installed easyblocks
- overview of available easyblocks via `eb --list-easyblocks`
- names of software-specific easyblocks always start with 'EB_'
- EasyBuild v1.16.1 includes:
 - 141 software-specific easyblocks
 - 20 generic easyblocks

Easyblocks vs easyconfigs

- thin line between 'fat' easyconfigs and an easyblock
- easyblocks are “do once and forget”
- central solution for build peculiarities
- can significantly simplify easyconfigs
- implemented in Python on top of framework API: very flexible
- reasons to consider an easyblock alongside a simply easyconfig:
 - 'critical' values for easyconfig parameters
 - configure/build/install options that are toolchain-dependent
 - custom (configure) options for included dependencies
 - hackish usage of parameters for existing (generic) easyblocks

Implementing easyblocks

quick overview

- each easyblock (eventually) derives from `EasyBlock` base class
- defines/extends/replaces one or more 'step' methods
- the `configure/build/install` steps *must* be defined
- API documentation:
https://jenkins1.ugent.be/view/EasyBuild/job/easybuild-framework_unit-test_hpcugent_master/Documentation/

Example

```
from easybuild.framework.easyblock import EasyBlock
from easybuild.tools.filetools import run_cmd
class EB_Foo(EasyBlock):
    def configure_step(self):
        run_cmd("PREFIX=%s ./configure.sh" % self.installdir)
    def build_step(self):
        run_cmd("build.sh %s" % self.cfg['buildopts'])
    def install_step(self):
        run_cmd("install.sh")
```

Derive from existing easyblocks

avoid duplicate code

- easyblocks can be defined hierarchically through inheritance
- (generic) easyblock can serve as a basis for others
- step methods of 'parent' can be inherited/extended/redefined
- maximizes code reuse across easyblocks

Example

```
class EB_Score_minus_P(ConfigureMake):
    def configure_step(self):
        """
        Custom configure step for Score-P:
        set configure options and run configure script.
        """
        comp_opt = "--with-nocross-compiler-suite=intel"
        self.cfg.update("configureopts", comp_opt)
        super(EB_Score_minus_P, self).configure_step() # parent
```

Minimal easyconfigs

easyblock takes care of the hard work

- if an easyblock is available, easyconfigs should be kept minimal
- easyblock should take care of configure/build/install options:
 - that depend on toolchain being used
 - for listed dependencies
 - that are common across builds
- sanity check paths/commands should be defined via easyblock
- easyblock can define extra custom easyconfig parameters
- easyconfig file can:
 - specify additional configure/build/install options
 - override sanity check paths defined by easyblock

Detailed documentation

<https://github.com/hpcugent/easybuild/wiki/Writing-easyblocks>

- different aspects of writing easyblocks to be documented
- including examples and references to existing easyblocks
- **work in progress**

Fully worked out example easyblock/easyconfig for WRF:

[https://github.com/hpcugent/easybuild/wiki/Tutorial:
-building-WRF-after-adding-support-for-it](https://github.com/hpcugent/easybuild/wiki/Tutorial:-building-WRF-after-adding-support-for-it)



Writing Easyblocks: The Basics

Kenneth Hoste
kenneth.hoste@ugent.be

EasyBuild hackathon - Basel, 20150211