xylem Documentation

Release 0.1.0

Open Source Robotics Foundation

May 28, 2014

Contents

`xylem` is a package manager abstraction tool. It can be used to install dependencies on any supported platform.

For example, if you want to install `boost` on your machine you would simply run `xylem install boost`. This command would cause `xylem` to determine your OS and OS Version, look up the corresponding package managers for that OS, OS Version tuple, look up the appropriate value for `boost` for that OS pair, and finally invoke the package manager to install boost, e.g. for Ubuntu that might be `sudo apt-get install libboost-all-dev`.

This tool allows you to generalize your installation instructions and define your software package's dependencies once. `xylem` also has an API which can be used to automate installation of resources, like for automated tests or for simplified installation scripts.

Contents:

CHAPTER 1

xylem's Design Overview

1.1 Motivation

What is the motivation for developing xylem as a new tool as opposed to updating rosdep?

rosdep was originally designed for use with rosbuild and both code and command line interface are structured for that purpose. The notion of stacks, packages and manifests where where rosdep keys were defined at a stack level is deeply baked into the design. Later adaptations to work with catkin were bolted on to that design in a suboptimal way and in became increasingly hard to extend rosdep with new features. Thus, rosdep has **a lot of unused or overly complicated code**.

Moreover, rosdep is currently linked tightly to several other ROS tools like bloom, such that even minor changes in rosdep can have deep ramifications in the tool chain. Due to this fragility, releases are slow and infrequent. Moreover, rosdep is not modular enough to facilitate extensions through third-party python packages. Together, all this implies that it is extremely **difficult to improve rosdep, implement new features, and get them released**.

Therefore it was concluded that it be more efficient to start fresh, borrowing ideas and code from rosdep, but designing it the way it should be rather than the way it used to be. Hence, xylem was born.

1.2 Goals

xylem is supposed to supersede rosdep as a package manager abstraction tool that can be used to install dependencies on any supported platform in a uniform manner. In particular, the goals of xylem are the following.

1.2.1 Separation of concerns

xylem addresses one of the key shortcomings of rosdep, namely its tight coupling with other ROS tools, with a modular design that considers the following building blocks.

- A **core library** that provides the infrastructure to read in rule files, resolve keys depending on the user platform and invokes package managers to install the desired software.
- A set of **plugins** that provide specific functionality:
  - operating system support (e.g. Ubuntu, OS X, cygwin)
  - backend installers, i.e. package managers (e.g. APT, PIP, Homebrew)
  - frontend input of keys (e.g. directly from the command line or by parsing a directory of ROS packages)
  - sources of rules (e.g. rules files or released ROS packages from rosdistro)

   - command verbs (e.g. `xylem install`, `xylem update`)

`xylem` comes with default plugins for all of the above points of extension.

1.2.2 Extensibility

Plugins should be able to extend the core tool from within other Python packages, such that extensions can be made without the need to touch the core package. This allows extensions to be developed and distributed somewhat independently of `xylem` releases. General purpose plugins that have proven to be useful to a range of users should be considered for inclusion into the core library.

1.2.3 Independence from ROS

One aim with designing `xylem` in a modular and extensible way is allowing it to be completely independent from ROS. In particular the core library should not have any ROS specific special cases or assumptions. Any functionality that is specific to ROS should be implemented as plugins, and possibly distributed as a separate package `xylem-ros`.

The ways in which `rosdep` is currently tied to ROS are:

- Frontend input, for example by scanning a directory for ROS packages and checking / installing their dependencies.
- Extracting resolution rules from `rosdistro` information.
- API access from tools like `catkin` or `bloom`.
- Use of other ROS specific packages, e.g. `rospkg.os_detect`.

1.2.4 Replace rosdep

One aim for `xylem` together with its ROS specific plugins is to provide a full future replacement for `rosdep`. This entails providing command line tools to check and install dependencies of ROS packages as well as providing an appropriate python API that allows tools such as `catkin` or `bloom` to query xylem for dependency information. We do not aim at backward compatibility at the CLI or API level, but at the level of provided features.

In particular, this also means that the keys currently specified in package.xml files of ROS packages should continue to work with `xylem` (for non-EOL distributions at the very least).

Full backward compatibility in particular to EOL tools such as `rosbuild` does *not* have to be achieved.

1.2.5 Consider improvements

The design of `xylem` should consider the know limitations of `rosdep` and improve beyond the functionality of `rosdep`. While proposed enhancements possibly are not implemented right away, it should be ensured that future extensions allow their realization without the need to break backwards-compatibility or for heavy redesign.

The following list of exemplar improvements is not necessarily exhaustive, nor definitive. More details on some of these ideas can be found further blow.

- improve rule files
   - smaller backwards-compatible changes, mostly syntactic sugar for less repetition for different platforms (`any_version`, `any_os`)
   - support versions in rules files, e.g. parsed from `package.xml` files [details]
   - support different types of dependencies such as test dependencies

  - support package managers with options (such as formula options on homebrew, use flags on gentoo?)
  - consider precedence of conflicting rules [details]
  - inter-key dependencies [details]
- support package manager sources (e.g. PPAs for APT on Ubuntu) [details]
- support package manager prerequisites (such as PM is installed, PM cache is up-to-date, correct PPA is installed) [details]
- support multiple resolution alternatives on the same platform with sensible defaults as well as user-configurable arbitration between them (e.g. macports vs homebrew, apt vs pip) [*details <Alternative resolutions_>*_]
- configure source/cache location and supply working cache with installation [details]
- configure package manager plugins from config/cli (e.g. whether to use sudo or not, supply additional command line arguments) [details]
- support concurrent invocations of `xylem`, in particular the `update` verb for tools such as `bloom` running in parallel. [details]
- support automatic cache updates (integrate update with native package manager, cronjob, ...)
- support virtual packages and/or `A OR B` logic
- support proxies
- support derivative operating systems (e.g. use Ubuntu rules on Ubuntu derivatives if no specific rules are available)
- warn users when `xylem` is out of date [details]
- version the rules database and force update on version changes
- improve situation on Windows

1.2.6 Anti-Goals

`xylem` does not aim to replace package managers or package software itself. While support for package-manager-less platforms can be achieved with backend plugins such as the source installer, it is not an objective of xylem to systematically maintain such installation scripts.

1.3 Supported platforms

`xylem` aims to support at least the following platforms (which is what `rosdep` currently supports) with their native package managers

- arch (pacman)
- windows/cygwin (apt-cyg)
- debian (apt)
- freebsd (pkg_add)
- gentoo (portage)
- opensuse (zypper)
- osx (homebrew, macports)
- redhat (yum)

as well as the following language-specific cross-platform packages managers

- ruby (gem)
- python (pip)

and a platform independent source installer:

- source

On the wish list is better support for Windows, but it is unclear how this could be achieved.

1.4 Plugins

In order to be modular and extensible by independent Python packages, `xylem` uses the Entry Points concept of `setuptools`. The following discusses the pluggable parts of `xylem` laid out above in more detail.

1.4.1 OS support

Operating system support includes:

- detecting OS name, version, codename (currently in `rospkg.os_detect`)
- register installers, default installer, installer order of preference etc with installer context (`rosdep2.installers.InstallerContext`)

Notes:

- Should OS support be plugin at all?
- Should are OS settings like registered installers and installer order of preference always per-OS as is in `rosdep`, or do we possibly need optional per-version distinction for these?
- What is relation between OS support plugins and installer plugins? Should OS plugin register all supported installers? Should installer plugin be able to register themselves for specific or all platforms?
- consider the distinction version_type vs codename_type
- support overriding detected OS from settings/cli

1.4.2 Backend installers

The supported installers are defined as plugins such that support for new installers can be added by external Python packages. Installers typically represent support for a specific package manager like APT, but not necessarily, as is the case for the source installer. The minimal functionality an installer needs to provide is:

- check if specific packages are installed
- install packages

Additional functionality is optional (these are ideas):

- support uninstall
  - e.g. source installer does not support this
- support native reinstall
  - is using the pm's native reinstall command as opposed to uninstall+install ever needed?
- support to attempt install without dependencies

  - this would be needed for a `specified-only` option to the `install` command.
  - not sure if we need this at all.
- support package versions
  - check which version of package is installed
  - check if installed package is outdated
  - upgrade installed package to latest version
  - install specific version of package
- support cache update
  - check if package manager cache is outdated
  - update cache (like `apt-get update`) or provide instructions for user how to update pm
- support options
  - some package managers additional options supplied when installing a package (homebrew, gentoo (use flags)?)
  - pass correct options to installer
  - check if options for installed package satisfy the requested options (e.g. they are superset)
- native dependencies
  - list all package manager dependencies of specific packages
  - the idea is that we let the package manager install the dependencies and only issue the install command for the necessary leafs
  - do we need this?

Notes:

- how is support for optional features formalized in the code?
- if new package managers can be added as plugins, then they need to be able to register themselves for specific or all operating systems

1.4.3 Frontend input

It needs to be possible to extend the way the user passes keys to be resolved to `xylem`. The basic usage would be directly passing a list of keys on the command line or API function. Another input would be parsing of ROS packages and checking the `package.xml` files. Another one would be a new file format `.xylem`, which allows non ROS packages to specify dependencies for convenient installation.

Notes:

- I'm not sure how exactly this would look.
- Implementing these as new command verbs gives ultimate flexibility, but on the other hand it makes much more sense if the standard commands like `install` or `check` can be extended. E.g. ROS support plugins for `xylem` should be able to provide an option like `--from-path` for the `install` verb.
- For compatibility of different frontends there are the following ideas:
  - Either the desired frontend has to be specified at the command line, e.g. `xylem install --frontend=ros desktop_full --rosdistro=hydro`, `xylem install --ros --from-path src`,

  - or the frontends register command line options that are unique, e.g. `xylem install --rospkg desktop_full`, `xylem install --ros-from- path .`,
  - or `xylem` can work some magic to find out which frontend the user desires, i.e. it determines if the input from the positional command line arguments consists of keys, directories, or ROS-packages. For directories is checks if they contain ROS packages with `package.xml` files or `.xylem` files. There is an order on which frontend takes precedence, which can be overwritten by explicitly specifying the frontend. This last alternative might make for the best *just works* user experience, but needs to be carefully thought through in order to not appear confusing.

1.4.4 Rules sources

The `rosdep` model for the definition of rules is configured in source files (e.g. `20-default-sources.yaml`) that contain the URLs of rules files (`base.yaml`). Multiple source files are considered in their alphabetical order. Having multiple files allows robot vendors to ship their own source files independently of the xylem base install. Possibly, rules plugins could also make use of this by shipping with additional default sources files. Initially, `xylem` will be using the same format, with some backwards- compatible (and already implemented) changes to the rules file format (`any_os`, `any_version`). Plugins can define new types of sources for rules. Right now we can foresee the following cases that might come as new source plugins:

- New rules file format that is not compatible with the existing format.
  - This would work in a very similar fashion to the initial plugin.
- Rules derived from `rosdistro`.
  - This is somewhat different, since it's sources are not specified by URLs but rather implicit using the `rosdistro` package.

Notes:

- Do we only support the *cache* model for sources, where a static rules database is built with the `update` command, but no new information is generated upon key resolution? This implies that rules sources that query some other database format (rosdistro?) or online sources at resolution time are not possible. In particular the `rosdistro` plugin would generate a list of rules for all released packages upon `update` (and not on-demand upon key resolution).
- What do the rules plugins return? The parsed rules from a given file in a (clearly defined) rules database format (something like the current `dict` database)? In any case the returned data should be in some versioned format, to allow future extensions to that format. This is probably the same format in which `xylem` keeps cached the database.
- Should we consider allowing for the possibility of loading parsed (and pickled) rules databases with the `update` command (for increased speed of `update`)? Here the original rules files would always be specified, but a binary version can be additionally added (somewhat like in homebrew all formula need to specify the source to build them, but some can additionally provide the binary package as a bottle).
- When are the different rules sourced merged (including arbitration of precedence)? During update, or while loading the cache database for resolution? Do we keep all possible resolutions in the database, or only the one that takes highest precedence?
- How is order of precedence defined between different rules plugins? Only by the order of the rules files? Do platform support plugins play a role in defining the precedence of different installers on a per-OS or per-version basis? Can user settings influence the order of precedence?
- Should the `.list` files be able to reference sources from multiple rules source plugins within the same file (which would also allow to control precedence if the entries are ordered within the file)?

  One can imagine a source files to look like this (not sure if this is correct YAML, but the idea should be clear):

```
# Overriding rules with highest precedence, but with legacy format
- format: rules
  sources:
    - 'some/special/rules.yaml'
# Latest rules in new format
- format: rules2
  sources:
    - 'latest/rules/using/new/rules/format/base.yaml'
# Existing rules in legacy format
- format: rules
  sources:
    - 'https://github.com/ros/rosdistro/raw/master/rosdep/base.yaml'
    - 'https://github.com/ros/rosdistro/raw/master/rosdep/python.yaml'
    - 'https://github.com/ros/rosdistro/raw/master/rosdep/ruby.yaml'
# this entry for the rosdistro rules plugin has no URLs, but is
  present to mark it as least-precedent
- format: rosdistro
```

- Do we support rules plugins that do not have an entry in any sources file (like `rosdistro`), or do we force all plugins to have at least an empty entry (example file above) in order to be 'activated' upon `update`.
- Should rules plugins include an abstraction to tell if the database is out of date (for a specific URL)? Something like comparing the last- changed timestamp of the cached databased with the last-changed timestamp of the online rules file. This might be used to speed up `update` and also to determine whether to remind the user to call `update`.

1.4.5 Commands

The top level command verbs to the `xylem` executable should be plugins. These can pretty much define any new functionality. It is not quite clear how exactly other plugins can interact with commands, e.g. frontend plugins should somehow be able to extend the `install` verb.

These are the core commands:

- `update` to update the rules database
  - If partial updates are supported, where only outdated rules files are pulled, there should be an option to force updating everything.
  - Needs to make sure to remove stale database cache files even on partial update, which are no longer referenced from the source files. Possibly add a `clean` command, that wipes the cache completely.
- `install` to install packages
  - options: `--reinstalll`, `--simulate`, `--skip-keys`, `--default-yes`, `--continue-on-error`, `--specified-only` (would this mean to not resolve dependencies on xylem level, or also stop possible dependency resolution of package manager, if that is even possible)
- `check` to check if packages installed
  - options: `--skip-keys`, `--continue-on-error`, `--specified- only`
- `init` to initialize config file and `sources.list.d` (possibly in custom location according to `XYLEM_PREFIX`). By default the built- in default sources / config is copied to the new location. Is a no-op with warning if sources / config is present.

  options:
  - `--from-prefix` to copy the config/sources that would be used with this given prefix
  - `--from-system` to copy the config/sources that would be used with empty prefix

  - `--force` to clear the config/sources even if they are present

These commands for dependency resolution could be useful:

- `depends` (options: `--depth` where 0 means no limit)
- `depends-on` (options: `--depth` where 0 means no limit)

There should also be some commands for checking how a key resolves on a specific operating system, possibly listing alternative resolutions (pip vs apt) highlighting the one that would be chosen with `install`. It should also be possible to determine where these resolutions come from, e.g. which source files.

- `resolve`
- `where-defined`

Notes:

- we might want to steal the alias mechanism from `catkin_tools`, but that is maybe low priority, since `xylem` command invocations would be much less frequent than `catkin build` invocations.

1.5 Improvements over rosdep

In the following we elaborate on some of the concrete improvements over `rosdep` listed above. Some of them are far future, some should be implemented right away.

1.5.1 Sources and cache location

The `xylem` model of a lookup database cache that is updated with and `update` command is somewhat analogous to `apt-get`. By default a system-wide cache is maintained that needs to be updated with `sudo`. We assume that many developer machines are single-user and/or are maintained by an admin that ensures regular `update` invocations (e.g. cronjob).

On top of the general scenario the following specific use-cases need to be supported with regards to the database cache:

- `xylem` needs to allow users to maintain their own cache in their home folder and use `xylem` independent from the system-wide installation and without super user privileges.
- Robot vendors need to be able to add to the default sources independently from the core `xylem` install and without post- installation work.
- `xylem` needs to be functional out of the box after installation. `update` requires internet connectivity, which is not given in some lab/robot environments. Therefore we need to make sure that `xylem` can be packaged (e.g. as debian) with a pre-generated binary cache. This needs to be possible for the default sources bundled with `xylem` as well as vendor supplied additional source files.
- Tools like `bloom` need to be able to create temporary caches independent from the system wide install and without super-user privileges.

We propose the following solution:

- Firstly, we assume that each URL/entry in the source files has it's own binary database cache file, all of which get merged upon lookup.
- The user can specify the `XYLEM_PREFIX` environment variable (overwritten by a command line option, maybe `--config-prefix` or `-c`). By default an empty prefix is assumed.
- The cache will live in `<prefix>/var/cache/xylem` and the sources in `<prefix>/etc/xylem/sources.d/`

- A xylem installation comes bundled with default source files and default cache files. However, in particular the cache is not installed into the `/var/cache` location directly.
- The `init` command installs the default sources and default cache into the corresponding locations. There are command line options to copy existing sources/cache from another prefix, but by default the built-in files are used. The source files are only installed if they are not present. The cache files are only installed, if the corresponding source file was either not present, or was present and identical to the default. Existing cache files are not overwritten. There is a flag (maybe `--force`), that causes it to overwrite the default files (sources and cache). Additional source files/cache files are not overwritten.
- `init` is called as part of the post-installation work at least for debians, maybe also pip? Note that this does not require internet connection and sets up a working config and cache.
- The default source files could be handled as conffiles in the debians, such that they are updated upon `apt-get upgrade`, where the user is queried what should happen if he has changed the default sources.
- `update` does not automatically use the the built-in sources if none exist under the given prefix. However, if the default source files do not exist, it warns the user and possibly tells him to call `xylem init` (or even offers to call it). This warning can be disabled in the settings for users that want to explicitly delete the default config files.
- Robot vendors that want to supply additional default sources can hook into `init` (with an entry point) and register their additional default sources as well as binary caches. All the above mechanisms work for those vendors. For example, if the additional vendor package gets installed, a subsequent post-install `init` does recognize the missing caches for installed default sources and installs them to ensure out-of-the-box operation. Likewise, calling `update` in a custom prefix after installing an additional vendor package will warn the user, that some of the default sources are not installed and urge her to call `init`, which will add these additional default sources (and cache files), while not touching the existing default source files from the core library.

For `rosdep`, there is pull request for a slightly different solution. However, what we suggest addresses some of the remaining issues:

- (re-)installing from debs does not overwrite existing cache files.
- python2 and python3 debians can be installed side-by-side (at least if the default source files are not handled as conffiles)

Notes:

- Should it be `sources.list.d` or `sources.d`? Note that we probably change the source files from `.list` to `.yaml`, so does `sources.list.d` still make sense?
- Can we ensure that the binary (pickled) database format is compatible between python2 and python3?
- If the default files have been updated, and the user updates the xylem installation, init will not change the existing default sources. Do we need to / can we detect if they are unchanged and replace them automatically if they are unchanged? If they are changed, ask the user what to do (like debian conffile).
- Do the API calls respect the `XYLEM_PREFIX` environment variable or need explicit setting of a `prefix` parameter? I think the latter.
- It was mentioned that the debian install needs to work out-of-the-box "without any post-installation work". Why exactly? Is post-install work (like calling `init`) ok if it does not require internet connectivity?
- Maybe the system wide settings file is also affected by `XYLEM_PREFIX`, i.e. lives in `<prefix>/etc/xylem/config`?

1.5.2 Settings and command line arguments

There should be a canonical way to supply arguments to `xylem`. We propose a system config file, a user config file and command line options. The order of precedence of arguments specified multiple times is:

```
command line > user > system
```

We use `yaml` syntax for the configuration files, and suggest the following locations:

- system: `<prefix>/etc/xylem/config.yaml`
- user: `$HOME/.xylem.yaml`

In general all options should be supported both by the CLI and the config files (where it makes sense). One exception is the environment variable `XYLEM_PREFIX`, because this configures the location of the system-wide config file in the first place.

Command line arguments can be grouped in the following way:

- global command line arguments applicable to all commands such as `disable-plugins` or `os`
- command specific command line arguments
- In order to achieve a good user experience, the command specific options should be further grouped. For example, all commands that take a list of keys as arguments, should do so in the same way, e.g. offering `skip-keys`)

It has to be seen if and how either or both kinds of arguments can be injected by plugins (e.g. frontend plugins inject new arguments to all commands that take a list of keys as input).

In particular it needs to be possible to supply arguments to the backend installer plugins (e.g. `as-root` or `additional-arguments`, see rosdep#307). `yaml` format gives a lot of flexibility, but there should also be some conventions (not necessarily enforced) to ensure that the plugins name their options in a uniform way, such that it may even be possible and reasonable to pass certain options to all installer plugins.

Notes:

- Should user file be in `$HOME/.config/xylem.yaml`, or even `$HOME/.config/xylem/config.yaml` (see stackexchange.com)? What about config locations on Windows?

1.5.3 Inter-key dependencies in rules files

In general, we rely on the backend package manager to install dependencies for resolved keys. Dependencies between keys in rules files is at the moment only used for the interplay between homebrew and pip on OS X it seems. Should this be a general feature for rules to depend on other keys? In particular if we reactivate the source installer this would be needed. In particular when considering adding versions to the rules files, doing dependency resolution right is not quite trivial I guess.

Dependencies on other keys might be reasonable on different levels. Currently they are part of the installer section, but maybe they could be defined also at the rule level.

1.5.4 Notify user about outdated database

Ideally, if the source plugins can tell when they are outdated, we would fork a process on every invocation to check if database is out of date and inform the user that an update would be good on the next run. Maybe limit the update check to only fire if the database has not been updated for a certain amount of time (a day, a week, could be customizable).

1.5.5 Versions in rules files

In general the user should expect a command `xylem install boost` to install the latest version of `boost` on the given system, i.e. on Ubuntu the version that `apt-get install boost` would install. For some package managers, like apt for a specific Ubuntu release, this might be always the same version of boost, for other package managers such as pip or homebrew, this will always refer to the latest version. This gives rise to two challenges with respect to software versions. Firstly, at any given time the key `boost` refers to different versions of the boost library on different platforms. Secondly, at two different points in time the key `boost` refers to two different versions of the boost library on the same platform. These challenges need to be taken into consideration, since the goal of `xylem` is to allow specification of dependencies in a uniform way that is robust over time, i.e. can be supplied as part of install instructions today and still be valid tomorrow.

At the moment, `rosdep` does not really consider versions, which users find confusing in particular in conjunction with ROS packages that may specify versioned dependencies (rosdep#325).

In general we assume that package managers can only install one version of a specific package at a time (largely true for apt, homebrew, pip). We also assume that we never install a specific version of a package with the package manager, but only the latest version, or possibly upgrade an already installed package to the latest version. Nevertheless, the package manager should be able to tell us, which version of a package is installed and which version would be installed/upgraded (i.e. the latest version on that platform).

For some libraries multiple incompatible major versions need to be present at the same time. Here `xylem` follows suite with package managers such as apt and homebrew and introduces new keys for the specific versions (as `rosdep` does currently). For example, for Eigen there are the version specific `eigen2` and `eigen3` keys, as well as a general `eigen` key that points to the latest version (i.e. is currently the same as `eigen3`).

What could be considered, is that `xylem` allows for input keys to be associated with version requirements (==, <=, >= etc) and then check, if the installed or would-be installed version matches. This would solve the use case with ROS packages above, where there is a one-to-one relation between xylem key and apt package. However, it is unclear how the version is handled if a key resolves to 0 or more than 1 packages. However, the most we would offer in terms of action is upgrading an already installed package to the latest version, and informing the user if a matching version cannot be achieved by upgrading or if the version requirements are incompatible themselves (i.e. user installs foo and bar, which depend on baz>1.0 and baz<1.0 respectively). Special care needs to be taken to correctly merge multiple versioned resolutions of the same key.

Another level of support for versions in rules would be to allow the resolution rules themselves to be conditional on a version, e.g. allowing to specify that `eigen` would resolve to `libeigen2-dev` or `libeigen3-dev`, depending on the version. With this, the versioned key `eigen==2` and `eigen==3` could be resolved at the same time. Things could get really complicated and I'm not sure we want to go down that route unless there is a good concrete use case where this is beneficial.

Notes:

- check how package managers deal with versions, in particular the capabilities (install multiple version of same package, install specific version of package not only latest) and syntax for versioned dependencies
  - apt
  - homebrew
  - pip: https://pip.pypa.io/en/latest/user_guide.html#requirements-files, http://pythonhosted.org/setuptools/setuptools.html#dec dependencies
- interesting blog about abstract vs concrete dependencies in python https://caremad.io/blog/setup-vs-requirement/

1.5.6 Improved package manager abstraction

[TODO: these are only random thoughts. transform them into a coherent and comprehensible description]

- support stuff like custom ppa's for apt, taps for homebrew
- the ros-ppa should not be special in xylem
- possibly specified on a per-rules-file basis? (identify real world use cases / needs)

- if custom ppa's are supported, provide tools to list the ppa's for bunch of keys / rules sources
- rules should never specify the ppa location, but rather have some sort of names prerequisite. this way the user could configure/overwrite the prerequisite in the config file if he e.g. has a customized mirror of that ppa or tap.
- issue of trust for the user (auto add alternavte pm sources? query user?)
- issue of reliability of sources for the maintainer
  - tool support to ensure ROS core packages are only using ubuntu or osrf ppa?
- maybe the right abstraction is *package manager prerequisites*
  - possibly not support undoing these prerequisites
  - prerequisites should be performed before any packages is installed
  - could query user or be automatic (with explicit option) or fail with instructions to user
  - allow user to configure and also skip specific or all prerequisite checks.
  - special prerequisite is the 'availability', which checks if the pm is installed. This should be treated specially, because maybe the selection of used package manager should depend on which is installed (e.g. macports vs homebrew). Ability to list available package managers
  - maybe with the previous it makes sense to distinguish general prerequisites (apt is installed and possibly up-to-date) and per-key prerequisites (certain ppa is installed)
  - concrete examples: * apt: ppa installed * source installer: tools installed (gcc etc) * brew: homebrew installed, Tap tapped * pip: pip installed

1.5.7 Alternative resolutions

Allow for alternatives with resolutions on a specific platforms, e.g. the use can choose macports vs homebrew on OS X, or to use pip over apt for python packages on Ubuntu.

Notes:

- multiple resolutions for one key on a specific os/version
- how to do the right thing by default? (e.g. detect if either homebrew or macports is installed to determine the default. Maybe some people never want to fallback to macports, maybe some want to fall back to macports if a key is not defined for homebrew)
- have preferred order of the different alternatives, customizable (at what granularity?)
- for debian releases only apt dependencies are allowed, for stuff like homebrew we can also depend on pip / gem
- per rules file or per key
- `xylem resolve` command should list all alternatives and help to arbitrate

1.5.8 Random points

- bring back the source installer
- improve windows situation; possibly source installer? windows 8 app store :-)
- integrate/interact with http://robotpkg.openrobots.org somehow? Check their solution for ideas for xylem.
- continue on error option for `install`
- authority on rules and versions

1.6 Terminology

[TODO: Define terms]

- xylem key
- key database
- rules file
- (backend) installer
- package manager
- platform –> os/version tuple
- installer
- installer context
- package -> pm package
- rules file

CHAPTER 2

xylem Rules

This module implements parsing and validating of rules spec files.

Note: These xylem rules files are compatible with rosdep's files.

2.1 xylem Rules File Specification

First, the top level rules file has these properties:

- The rules file is a YAML 1.1 compliant file
- The rules file contains a single dictionary, the rules dict

2.1.1 the rules dict

The rules dict contains a mapping between xylem keys and definitions for specific operating systems and package managers.

The rules dict has these properties:

- The keys of the rules dict are package manager agnostic xylem keys
- The values of the rules dict must be a dictionaries also, os definition dicts

2.1.2 the os specific definition dict

An os specific definition dict is a mapping of operating system names to os specific definitions. They have these properties:

- It has keys which map to os_names, e.g. ubuntu, osx
- There is a special 'any' key, i.e. 'any_os'
- It can have values of type dict, with os_version keys
- It can have values of type list, a set of packages for the default installer
- It can have values of type str, a package for the default installer
- A value of null or '[]' indicates no action is required to resolve for this os

When the value is a dict, then the keys of that dict are os_version's and the values of that dict are os_name and os_version specific definitions for those xylem keys.

the 'any_os' key

The 'any_os' key for the os specific definition dict can be used to indicate that the following definition matches any operating systems. This is useful for situations like the pip installer, which works on most operating systems, but is not specific to any one operating system. When the 'any_os' key is used, then the installer must be specified explicitly, i.e. no default key is used. Conversely, if the 'any_os' key is used, then the 'any_version' key must be used for the os_version as well, which makes sense because it doesn't make sense to have a definition which matches all operating systems but only a specific operating system version.

For example, consider this rule snippet:

```
foo:
  any_os:
    any_version:
      pip:
        packages: [foo]
  ubuntu: [python-foo]
  debian: [python-foo]
  osx:
    any_version:
      homebrew: [foo]
```

In this case `foo` will be resolved like this:

```
windows:7          -> pip:foo
ubuntu:precise     -> apt:python-foo
debian:wheezy      -> apt:python-foo
osx:mountain_lion -> homebrew:foo
```

You can imagine the logic to implement this behavior as such:

```
# foo_dict is a dictionary like the above YAML structure
os_name = get_os_name()
os_version = get_os_version()
if os_name in foo_dict:
    if os_version in foo_dict[os_name]:
        return foo_dict[os_name][os_version]
    elif 'any_version' in foo_dict[os_name]:
        return foo_dict[os_name]['any_version']
elif 'any_os' in foo_dict:
    if 'any_version' in foo_dict['any_os']:
        return foo_dict['any_os']['any_version']
raise NotFound
```

list and string expansion into os_version's any_version

When the value of the os definition dict is a str, then it is converted into a list containing that str.

Whether the value is a str converted into a list or originally a list, the list is expanded into an any version ('any_version') key-value pair, where the list is the value. Then the processing continues as normal.

For example, this snippet:

```
foo:
  ubuntu: [libfoo]
```

is expanded to:

```
foo:
  ubuntu:
    any_version: [libfoo]
```

The above snippet is a intermediate expansion, as `any_version: [libfoo]` will get expanded further later.

The case where no action is required, can occur when the package exists on the system by default. For example, on OS X many times the software package comes with OS X and no action is required for it to be resolved. This is represented with an empty list or null.

Whether originally a dict or expanded to a dict from a list, the resulting os specific definition dict always has os_names for keys and os_version specific definitions as values.

2.1.3 the os_version specific definition dict

The os_version specific definition dict's have these properties:

- It has keys which map to os_version's for the parent os_name.
- There can be one any_version key, i.e. 'any_version'
- The values can be a list or str, and are converted like the os definition dict
- The final values must be an installer specific definition dict

The os_version specific definition dict's are exactly like the os_name specific definition dict's, except that the resulting values are installer specific definition dict's, and there is a wild card key.

the 'any_version' key

The any_version key, 'any_version', indicates that the value, which is an installer specific definition dict, applies to all os_version's which do not have an explicit definition. For example, consider this snippet:

```
foo:
  ubuntu:
    lucid: [libfoo-1.8]
    any_version: [libfoo]
```

The above snippet will provide `libfoo-1.8` if you ask xylem to resolve `foo` for `ubuntu:lucid`, but will return `libfoo` for any other version of `ubuntu`, e.g. for `ubuntu:precise` xylem will resolve it as `libfoo`.

2.1.4 the installer specific definition dict

The installer specific definition dict has keys for installers, e.g. `apt`, `pip`, or `homebrew`. The installer specific definition dict is similar to the os_version specific definition dict except the default_installer key is interpreted differently.

the 'default_installer' key

When the default installer key, 'default_installer', is used in the installer specific definition dict, that indicates that the following definition is for the *default* installer for that operating system. This key cannot be used under the *any_os* key.

2.2 Examples

Basic:

```
foo:  # xylem key
  ubuntu:  # os name
    precise:  # os version (or codename)
      apt:  # installer
        packages: [libfoo]  # packages are a list
```

This is an example of using some of the available shortcuts:

```
foo:
  ubuntu: libfoo
  debian: libfoo
bar:
  ubuntu:
    lucid: libbar-1.2
    any_version: libbar
baz:
  any_os:
    any_version:
        pip: [baz]
  ubuntu: [libbaz]
```

Which expands to:

```
foo:
  ubuntu:
    any_version:
      default_installer:
        packages: [libfoo]
  debian:
    any_version:
      default_installer:
        packages: [libfoo]
bar:
  ubuntu:
    lucid:
      default_installer:
        packages: [libbar-1.2]
    any_version:
      default_installer:
        packages: [libbar]
baz:
  any_os:
    any_version:
        pip:
          packages: [baz]
  ubuntu:
    any_version:
      default_installer:
        packages: [libbaz]
```

CHAPTER 3

xylem Python API

Experimental: the xylem Python library is still unstable.

The `xylem` Python module supports both the *xylem* command-line tool as well as libraries that wish to use xylem data files to resolve dependencies.

As a developer, you may wish to extend `xylem` to add new OS platforms or package managers.

Table of Contents

3.1 Database

Implements the update functionality.

This includes the functions to collect and process source files. Part of this process is to load and run the spec parser, which are given by name in the source files.

`xylem.update.`**`handle_spec_urls`**(*spec*, *urls*)

Load a given spec parser by spec name and processed all urls.

Returns a list of new rules from parsed urls

Parameters

- **spec** (*str*) – name of a spec parser to load
- **urls** (`list` of `str`) – list of urls to load for the given spec parser

`xylem.update.`**`load_url`**(*url*, *retry=2*, *retry_period=1*, *timeout=10*)

Load a given url with retries, retry_periods, and timeouts.

Based on https://github.com/ros-infrastructure/rosdistro/blob/master/src/rosdistro/loader.py

Parameters

- **url** (*str*) – URL to load and return contents of
- **retry** (*int*) – number of times to retry the url on 503 or timeout
- **retry_period** (*float*) – time to wait between retries in seconds

- **timeout** (*float*) – timeout for opening the URL in seconds

`xylem.update.`**`update`**(*prefix=None*, *dry_run=False*)

Update the xylem cache.

If the prefix is set then the source lists are searched for in the prefix. If the prefix is not set (None) or the source lists are not found in the prefix, then the default, builtin source list is used.

Parameters

- **prefix** (`str` or `None`) – The config and cache prefix, usually '/' or someother prefix
- **dry_run** (*bool*) – If True, then no actual action is taken, only pretend to

`xylem.update.`**`verify_rules`**(*rules*, *spec*)

Verify that a set of rules are valid for internal storage.

Parameters **rules** (*dict*) – set of nested dictionaries which is the internal DB format

3.2 Indices and tables

- *genindex*
- *modindex*
- *search*

CHAPTER 4

xylem package

4.1 Subpackages

4.1.1 xylem.commands package

Submodules

xylem.commands.main module

`xylem.commands.main.`**`create_subparsers`**`(`*parser, cmds*`)`

`xylem.commands.main.`**`list_commands`**`()`

`xylem.commands.main.`**`load_command_description`**`(`*command_name*`)`

`xylem.commands.main.`**`main`**`(`*sysargs=None*`)`

`xylem.commands.main.`**`print_usage`**`()`

xylem.commands.update module

`xylem.commands.update.`**`main`**`(`*args=None*`)`

`xylem.commands.update.`**`prepare_arguments`**`(`*parser*`)`

Module contents

4.1.2 xylem.sources package

Submodules

xylem.sources.impl module

`xylem.sources.impl.`**`get_default_source_urls`**`()`

Return the list of default source urls.

Returns lists of source urls keyed by spec type

Return type `dict`(:py:obj:`str: list`(:py:obj:`str`))

`xylem.sources.impl.`**`get_source_urls`**(*prefix*)
Return a list of source urls.

Parameters **prefix** – prefix on which to look for etc/xylem/sources.list.d

Type prefix: str

Returns lists of source urls keyed by spec, or None if no configs found

Return type `dict`‘(:py:obj:‘`str`: `list`‘(:py:obj:‘`str`))

`xylem.sources.impl.`**`load_source_lists_from_path`**(*path*)
Return a list of source urls from a given directory of source lists.

Only files which have the .yaml extension are processed, other files, hidden files, and directories are ignored.

Parameters **path** (*str*) – directory containing source list files

Returns lists of source urls keyed by spec type

Return type `dict`‘(:py:obj:‘`str`: `list`‘(:py:obj:‘`str`))

`xylem.sources.impl.`**`parse_list`**(*data*, *file_path='<string>'*)
Parse a given list of urls and returns them as a list of source urls.

Parameters **data** (*str*) – string containing a list of source urls

Returns lists of source urls keyed by spec type

Return type `dict`‘(:py:obj:‘`str`: `list`‘(:py:obj:‘`str`))

Raises ValueError, yaml.YAMLError

`xylem.sources.impl.`**`parse_list_file`**(*file_path*)
Parse a given list file and returns a list of source urls.

Parameters **file_path** (*str*) – path to file containing a list of source urls

Returns lists of source urls keyed by spec type

Return type `dict`‘(:py:obj:‘`str`: `list`‘(:py:obj:‘`str`))

Module contents

`xylem.sources.`**`get_default_source_urls`**()
Return the list of default source urls.

Returns lists of source urls keyed by spec type

Return type `dict`‘(:py:obj:‘`str`: `list`‘(:py:obj:‘`str`))

`xylem.sources.`**`get_source_urls`**(*prefix*)
Return a list of source urls.

Parameters **prefix** – prefix on which to look for etc/xylem/sources.list.d

Type prefix: str

Returns lists of source urls keyed by spec, or None if no configs found

Return type `dict`‘(:py:obj:‘`str`: `list`‘(:py:obj:‘`str`))

4.1.3 xylem.specs package

Submodules

xylem.specs.impl module

exception `xylem.specs.impl.`**`SpecParsingError`**(*msg*, *related_snippet=None*)

Bases: `exceptions.ValueError`

Raised when an invalid spec element is encountered while parsing.

`xylem.specs.impl.`**`get_spec_parser`**(*name*)

Return a spec parser of a given name, or None if it is not found.

Parameters **name** (*str*) – name of the requested spec parser

Returns the requested spec parser, or None if it isn't found

Return type types.FunctionType

`xylem.specs.impl.`**`list_spec_parsers`**()

List available spec parsers, by name.

Returns list of spec parsers by name

Return type `list`(:py:obj:`str)`

xylem.specs.rules module

This module implements parsing and validating of rules spec files.

Note: These xylem rules files are compatible with rosdep's files.

xylem Rules File Specification

First, the top level rules file has these properties:

- The rules file is a YAML 1.1 compliant file
- The rules file contains a single dictionary, the rules dict

the rules dict The rules dict contains a mapping between xylem keys and definitions for specific operating systems and package managers.

The rules dict has these properties:

- The keys of the rules dict are package manager agnostic xylem keys
- The values of the rules dict must be a dictionaries also, os definition dicts

the os specific definition dict An os specific definition dict is a mapping of operating system names to os specific definitions. They have these properties:

- It has keys which map to os_names, e.g. ubuntu, osx
- There is a special 'any' key, i.e. 'any_os'
- It can have values of type dict, with os_version keys
- It can have values of type list, a set of packages for the default installer

- It can have values of type str, a package for the default installer
- A value of null or '[]' indicates no action is required to resolve for this os

When the value is a dict, then the keys of that dict are os_version's and the values of that dict are os_name and os_version specific definitions for those xylem keys.

the 'any_os' key The 'any_os' key for the os specific definition dict can be used to indicate that the following definition matches any operating systems. This is useful for situations like the pip installer, which works on most operating systems, but is not specific to any one operating system. When the 'any_os' key is used, then the installer must be specified explicitly, i.e. no default key is used. Conversely, if the 'any_os' key is used, then the 'any_version' key must be used for the os_version as well, which makes sense because it doesn't make sense to have a definition which matches all operating systems but only a specific operating system version.

For example, consider this rule snippet:

```
foo:
  any_os:
    any_version:
      pip:
        packages: [foo]
  ubuntu: [python-foo]
  debian: [python-foo]
  osx:
    any_version:
      homebrew: [foo]
```

In this case `foo` will be resolved like this:

```
windows:7          -> pip:foo
ubuntu:precise     -> apt:python-foo
debian:wheezy      -> apt:python-foo
osx:mountain_lion -> homebrew:foo
```

You can imagine the logic to implement this behavior as such:

```python
# foo_dict is a dictionary like the above YAML structure
os_name = get_os_name()
os_version = get_os_version()
if os_name in foo_dict:
    if os_version in foo_dict[os_name]:
        return foo_dict[os_name][os_version]
    elif 'any_version' in foo_dict[os_name]:
        return foo_dict[os_name]['any_version']
elif 'any_os' in foo_dict:
    if 'any_version' in foo_dict['any_os']:
        return foo_dict['any_os']['any_version']
raise NotFound
```

list and string expansion into os_version's any_version When the value of the os definition dict is a str, then it is converted into a list containing that str.

Whether the value is a str converted into a list or originally a list, the list is expanded into an any version ('any_version') key-value pair, where the list is the value. Then the processing continues as normal.

For example, this snippet:

```
foo:
  ubuntu: [libfoo]
```

is expanded to:

```
foo:
  ubuntu:
    any_version: [libfoo]
```

The above snippet is a intermediate expansion, as `any_version:   [libfoo]` will get expanded further later.

The case where no action is required, can occur when the package exists on the system by default. For example, on OS X many times the software package comes with OS X and no action is required for it to be resolved. This is represented with an empty list or null.

Whether originally a dict or expanded to a dict from a list, the resulting os specific definition dict always has os_names for keys and os_version specific definitions as values.

the os_version specific definition dict The os_version specific definition dict's have these properties:

- It has keys which map to os_version's for the parent os_name.
- There can be one any_version key, i.e. 'any_version'
- The values can be a list or str, and are converted like the os definition dict
- The final values must be an installer specific definition dict

The os_version specific definition dict's are exactly like the os_name specific definition dict's, except that the resulting values are installer specific definition dict's, and there is a wild card key.

the 'any_version' key The any_version key, 'any_version', indicates that the value, which is an installer specific definition dict, applies to all os_version's which do not have an explicit definition. For example, consider this snippet:

```
foo:
  ubuntu:
    lucid: [libfoo-1.8]
    any_version: [libfoo]
```

The above snippet will provide `libfoo-1.8` if you ask xylem to resolve `foo` for `ubuntu:lucid`, but will return `libfoo` for any other version of `ubuntu`, e.g. for `ubuntu:precise` xylem will resolve it as `libfoo`.

the installer specific definition dict The installer specific definition dict has keys for installers, e.g. `apt`, `pip`, or `homebrew`. The installer specific definition dict is similar to the os_version specific definition dict except the default_installer key is interpreted differently.

the 'default_installer' key When the default installer key, 'default_installer', is used in the installer specific definition dict, that indicates that the following definition is for the *default* installer for that operating system. This key cannot be used under the *any_os* key.

Examples

Basic:

```
foo:  # xylem key
  ubuntu:  # os name
    precise:  # os version (or codename)
      apt:  # installer
        packages: [libfoo]  # packages are a list
```

This is an example of using some of the available shortcuts:

```
foo:
  ubuntu: libfoo
  debian: libfoo
bar:
  ubuntu:
    lucid: libbar-1.2
    any_version: libbar
baz:
  any_os:
    any_version:
        pip: [baz]
  ubuntu: [libbaz]
```

Which expands to:

```
foo:
  ubuntu:
    any_version:
      default_installer:
        packages: [libfoo]
  debian:
    any_version:
      default_installer:
        packages: [libfoo]
bar:
  ubuntu:
    lucid:
      default_installer:
        packages: [libbar-1.2]
    any_version:
      default_installer:
        packages: [libbar]
baz:
  any_os:
    any_version:
        pip:
          packages: [baz]
  ubuntu:
    any_version:
      default_installer:
        packages: [libbaz]
```

`xylem.specs.rules.`**`expand_definition`**(*definition*)

`xylem.specs.rules.`**`expand_installer_definition`**(*installer_dict*)

`xylem.specs.rules.`**`expand_os_definition`**(*os_dict*)

`xylem.specs.rules.`**`expand_os_version_definition`**(*version_dict*)

`xylem.specs.rules.`**`expand_rules`**(*rules*)

`xylem.specs.rules.`**`rules_spec_parser`**(*data*)

Module contents

`xylem.specs.`**`get_spec_parser`**(*name*)

Return a spec parser of a given name, or None if it is not found.

Parameters **name** (*str*) – name of the requested spec parser

Returns the requested spec parser, or None if it isn't found

Return type types.FunctionType

`xylem.specs.`**`list_spec_parsers`**`()`
List available spec parsers, by name.

Returns list of spec parsers by name

Return type `list`(:py:obj:`str)`

exception `xylem.specs.`**`SpecParsingError`**`(`*msg*, *related_snippet=None*`)`
Bases: `exceptions.ValueError`

Raised when an invalid spec element is encountered while parsing.

4.2 Submodules

4.3 xylem.log_utils module

`xylem.log_utils.`**`debug`**`(`*msg*, *file=None*, **args*, ***kwargs*`)`
Print debug to console or file.

Works like `print` and optionally uses terminal colors. Can be enabled or disabled with `enable_debug()`.

`xylem.log_utils.`**`enable_debug`**`(`*state=True*`)`
En- or disable printing debug output to console.

`xylem.log_utils.`**`error`**`(`*msg*, *file=None*, *exit=False*, **args*, ***kwargs*`)`
Print error statement and optionally exit.

Works like `print` and optionally uses terminal colors.

`xylem.log_utils.`**`info`**`(`*msg*, *file=None*, **args*, ***kwargs*`)`
Print info to console or file.

Works like `print` and optionally uses terminal colors.

`xylem.log_utils.`**`warning`**`(`*msg*, *file=None*, **args*, ***kwargs*`)`
Print warning to console or file.

Works like `print` and optionally uses terminal colors.

4.4 xylem.terminal_color module

Module to enable color terminal output.

class `xylem.terminal_color.`**`ColorTemplate`**`(`*template*`)`
Bases: `string.Template`

`delimiter` = '@'

`pattern` = **<_sre.SRE_Pattern object at 0x7f8f3a3dab10>**

`xylem.terminal_color.`**`ansi`**`(`*key*`)`
Return the escape sequence for a given ansi color key.

`xylem.terminal_color.`**`disable_ANSI_colors`**`()`
Disable output of ANSI color serquences with `ansi()`.

Set all the ANSI escape sequences to empty strings, which effectively disables console colors.

`xylem.terminal_color.`**`enable_ANSI_colors`**`()`
Enable output of ANSI color serquences with `ansi()`.

Colors are enabled by populating the global module dictionary `_ansi` with ANSI escape sequences.

`xylem.terminal_color.`**`fmt`**`(`*msg*`)`
Replace color annotations with ansi escape sequences.

`xylem.terminal_color.`**`sanitize`**`(`*msg*`)`
Sanitize the existing msg, use before adding color annotations.

4.5 xylem.update module

Implements the update functionality.

This includes the functions to collect and process source files. Part of this process is to load and run the spec parser, which are given by name in the source files.

`xylem.update.`**`handle_spec_urls`**`(`*spec*, *urls*`)`
Load a given spec parser by spec name and processed all urls.

Returns a list of new rules from parsed urls

Parameters

- **spec** (*str*) – name of a spec parser to load
- **urls** (`list` of `str`) – list of urls to load for the given spec parser

`xylem.update.`**`load_url`**`(`*url*, *retry=2*, *retry_period=1*, *timeout=10*`)`
Load a given url with retries, retry_periods, and timeouts.

Based on https://github.com/ros-infrastructure/rosdistro/blob/master/src/rosdistro/loader.py

Parameters

- **url** (*str*) – URL to load and return contents of
- **retry** (*int*) – number of times to retry the url on 503 or timeout
- **retry_period** (*float*) – time to wait between retries in seconds
- **timeout** (*float*) – timeout for opening the URL in seconds

`xylem.update.`**`update`**`(`*prefix=None*, *dry_run=False*`)`
Update the xylem cache.

If the prefix is set then the source lists are searched for in the prefix. If the prefix is not set (None) or the source lists are not found in the prefix, then the default, builtin source list is used.

Parameters

- **prefix** (`str` or `None`) – The config and cache prefix, usually '/' or someother prefix
- **dry_run** (*bool*) – If True, then no actual action is taken, only pretend to

`xylem.update.`**`verify_rules`**`(`*rules*, *spec*`)`
Verify that a set of rules are valid for internal storage.

Parameters **rules** (*dict*) – set of nested dictionaries which is the internal DB format

4.6 xylem.util module

Provides common utility functions for xylem.

`xylem.util.`**`add_global_arguments`**(*parser*)

class `xylem.util.`**`change_directory`**(*directory=''*)
    Bases: `object`

`xylem.util.`**`create_temporary_directory`**(*prefix_dir=None*)
    Create a temporary directory and return its location.

`xylem.util.`**`custom_exception_handler`**(*type, value, tb*)

`xylem.util.`**`handle_global_arguments`**(*args*)

`xylem.util.`**`pdb_hook`**()

`xylem.util.`**`print_exc`**(*exc*)

class `xylem.util.`**`redirected_stdio`**
    Bases: `object`

class `xylem.util.`**`temporary_directory`**(*prefix=''*)
    Bases: `object`

4.7 Module contents

CHAPTER 5

Installing from Source

Given that you have a copy of the source code, you can install `xylem` like this:

```
$ python setup.py install
```

Note: If you are installing to a system Python you may need to use `sudo`.

If you do not want to install `xylem` into your system Python, or you don't have access to `sudo`, then you can use a virtualenv.

CHAPTER 6

Hacking

Because `xylem` uses setuptools you can (and should) use the develop feature:

```
$ python setup.py develop
```

Note: If you are developing against the system Python, you may need `sudo`.

This will "install" `xylem` to your Python path, but rather than copying the source files, it will instead place a marker file in the `PYTHONPATH` redirecting Python to your source directory. This allows you to use it as if it were installed but where changes to the source code take immediate affect.

When you are done with develop mode you can (and should) undo it like this:

```
$ python setup.py develop -u
```

Note: If you are developing against the system Python, you may need `sudo`.

That will "uninstall" the hooks into the `PYTHONPATH` which point to your source directory, but you should be wary that sometimes console scripts do not get removed from the bin folder.

CHAPTER 7

Code Style

The source code of `xylem` aims to follow the Python style guide and the **PEP 8** guidelines. In particular a line width of 79 characters is enforced for python code, while multiline comments or docstrings as well as text files should use a line width of 72.

The test-suite checks that all xylem code passes the `flake8`. On top of that identifer names should follow the rules layed out in **PEP 8** and docstrings should adhere to **PEP 257**, however these are not automatically checked.

The most important rules are readability and consistency and use of common sense.

CHAPTER 8

Testing

In order to run the tests you will need to install nosetests and flake8.

Once you have installed those, then run `nosetest` in the root of the `xylem` source directory:

```
$ nosetests
```

Building the Documentation

In order to build the docs you will need to first install Sphinx. We use the Read the Docs Sphinx Theme, which you can install with:

```
$ sudo pip install sphinx_rtd_theme
```

You can build the documentation by invoking the Sphinx provided make target in the `docs` folder:

```
$ # In the docs folder
$ make html
$ open _build/html/index.html
```

Sometimes Sphinx does not pickup on changes to modules in packages which utilize the `__all__` mechanism, so on repeat builds you may need to clean the docs first:

```
$ # In the docs folder
$ make clean
$ make html
$ open _build/html/index.html
```