xylem Documentation

Release 0.1.0

Open Source Robotics Foundation

August 10, 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 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 it 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)
  - installers, e.g. package managers (e.g. APT, PIP, Homebrew), but also for example the source installer.
  - frontend input of keys (e.g. directly from the command line or by parsing a directory of ROS packages)
  - rules specification (e.g. rules files or released ROS packages from rosdistro)

  - default sources (e.g. additional default rules files from robot vendors)
  - 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`, `bloom` or `catkin_lint`.
- 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 known 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`) [*details*]
  - support ranges of versions and definitions for versions greater or equal to a specific version [*details*]

  - support options to OSs and condition rules on them [details]
  - support package versions in rules files, e.g. parsed from `package.xml` files [details]
  - support different types of dependencies such as test dependencies
  - 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]
- 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 derivative operating systems (e.g. use Ubuntu rules on Ubuntu derivatives if no specific rules are available) [details]
- warn users when `xylem` is out of date [details]
- version the rules database and force update on version changes
- improve situation on Windows
- support proxies for any downloads as well as for the installer invocations, see ros-infrastructure/rosdep#335
- support package managers with options (such as formula options on homebrew, use flags on gentoo?)

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 installer 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

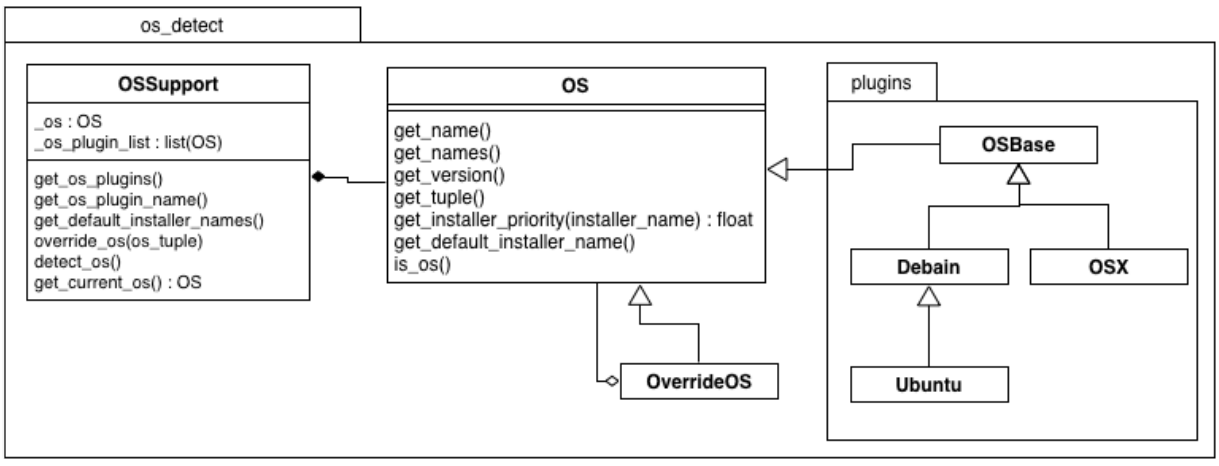
In order to be modular and extensible by independent Python packages, `xylem` uses the Entry Points concept of `setuptools`. The following discusses the pluggable parts of `xylem` laid out above in more detail.

1.4.1 OS support

Operating system support includes:

- detecting if current OS matches the OS plugin
- detecting the current OS version (or codename)
- specify supported installers, default installer and installer order of preference

OS plugins are derived from `xylem.os_support.OS` and `xylem.os_support.OSSupport` manages the list of os plugins as well as the current (possibly overridden) os. `xylem.os_support.OSSupport` is high-level API, but not necessarily used directly, but rather inside `xylem.installers.InstallerContext`



Notes:

- At the moment OS support plugins are not able to list all versions, but only ever detect the current version. The advantage is that no code update is necessary for each new OS release. The disadvantage is that the list of versions is not available e.g. to verify the structure of rules files or to distinguish between package manager and version names in rules definitions.

  - *Nikolaus*: I think we should leave it like that for now.

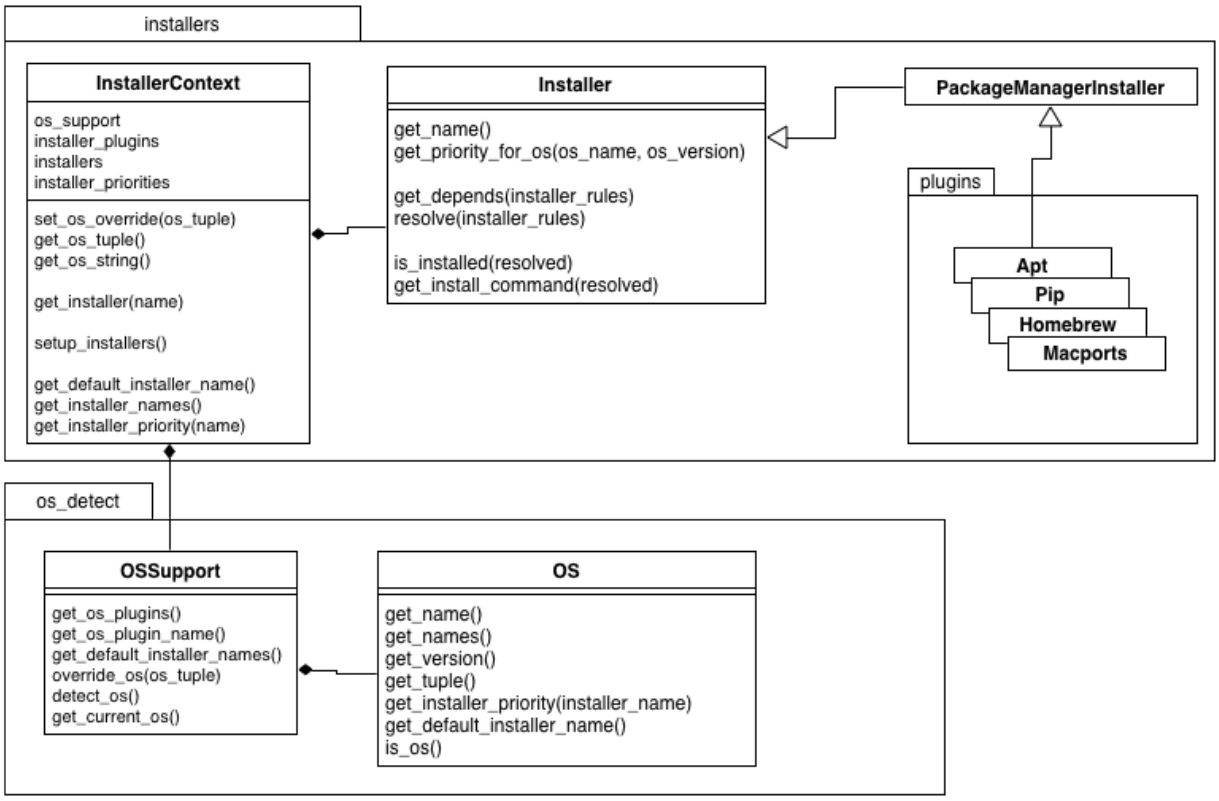
- For each OS plugin we have to choose if we use numbers or code names to specify versions. In general we try to use version code-names if possible. Version numbers have the disadvantage of being less memorable and some care needs to be taken because YAML might parse version numbers as floats, not strings. Codenames for some operating systems have the disadvantage that they are not in alphabetical order (e.g. OS X, debian), meaning the rules definition mappings in YAML are not in the chronological OS version order. Moreover, without the OS plugins listing the existing versions, version ranges cannot be specified because the order of versions is in general unknown. One might want to support shortcut notation in rules files like `ubuntu: "lucid - oneric": foo-pkg`.
  - *Nikolaus*: I'm not sure what we can do about this without listing the known OS versions. Even if they are known, we would need to have this information for formatting and verifying rules files (order of version dict).
- Should are OS configuration 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?
  - *Nikolaus*: I believe per OS is fine for now.

1.4.2 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

Installer plugins are derived from `Installer`. The list of known installer plugins is managed by a high-level API context object, the `InstallerContext`. The `InstallerContext` uses `OSDetect` to manage the detected/overridden OS. `setup_installers()` uses information from user configuration, os plugins and installer plugins to prepare the list of installers for the current os, their priorities, as well as the default installer. The idea is that information about which installer is used when multiple possible resolutions exist can come from different sources. In the default case, OS plugins specify which installers are used on that plugin (including a order of preference through priorities and a default installer). On top of that platform independent installer plugins can declare to be used on specific OSs (e.g. all OSs). This allows to write new installer plugins (e.g. for `go get`) that are available on platforms without touching the os plugins. Lastly, the user config can override all of that (available installers as well as their priorities).



The following are ideas for additional functionality of installer plugins. It is not quite clear how they are formalized in code. Maybe just methods that may be defined (duck typing or ABC mixin style). Some of these (like support for options) can be done transparently (as is done for homebrew in rosdep), but some require interaction with other components (e.g. uninstall, native reinstall, versions).

- support uninstall
  - e.g. source installer does not support this
- support native reinstall
  - Use the pm's native reinstall command as opposed to uninstall+install
  - *Nikolaus*: is this ever useful?
- support to attempt install without dependencies
  - this would be needed for a `specified-only` option to the `install` command.
  - *Nikolaus*: 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
  - *Nikolaus*: do we need this?

Notes:

- We need to allow the configuration to completely disable installers (for specific os), e.g. disable macports on OS X (in favour of homebrew).
- Can we change the default resolution on OS X based on which of PM (macports, homebrew) is installed? With that the resolution depends on the system state, which is maybe not so nice.
- See http://www.ros.org/reps/rep-0112.html and http://www.ros.org/reps/rep-0111.html

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:

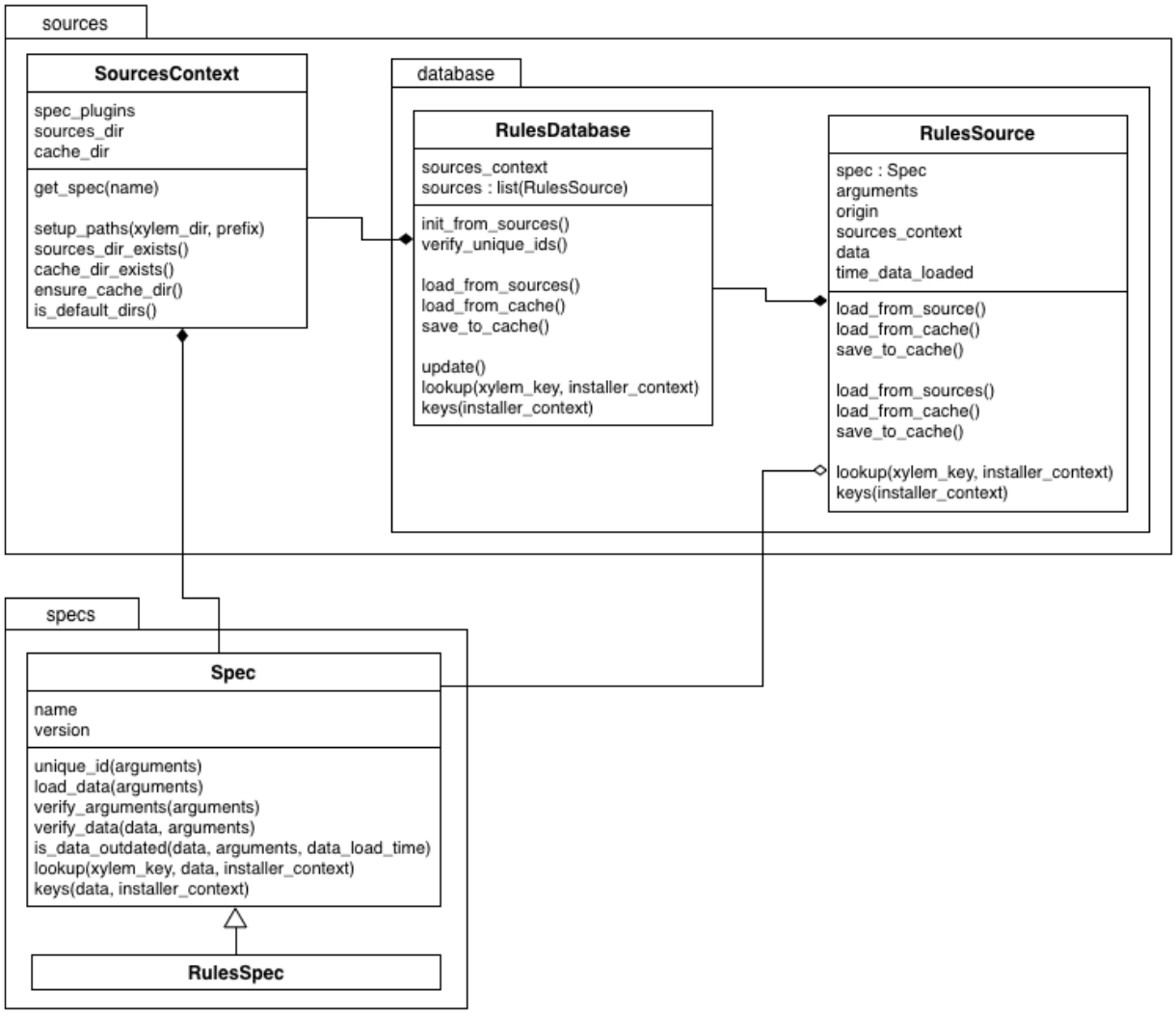
- *Nikolaus*: I'm not sure yet how exactly those plugins would look.
- Implementing these as new command verbs gives ultimate flexibility, but on the other hand it makes much more sense if the standard commands like `install` or `check` can be extended. E.g. ROS support plugins for `xylem` should be 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.
  - *Nikolaus*: 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 specification

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 base install and also allows to organize the base rules files (e.g. one file for all python packages rules). `xylem` will be using a similar format of source files listing rules files, with some (mostly) backwards- compatible (and already implemented) changes to the rules file format (`any_os`, `any_version`, see *xylem Rules files*). `spec` plugins can define new types of specifications for rules. The source files indicate which spec plugin to use for each entry. Right now we can foresee the following cases that might come as new spec plugins:

- New rules file format that is not compatible with the existing format.
  - This would work in a very similar fashion to the initial `RulesSpec` spec plugin.
- Rules derived from `rosdistro`.
  - This rules spec uses the `rosdistro` package to derive rules for each ROS distro.

The design for the rules sources and spec plugins is as follows:



Spec plugins derive from `Spec`. They define how rules are specified and at the core provide `load_data` and `lookup` methods. The plugin for rules files is `RulesSpec`

A `SourcesContext` object manages known spec plugins as well as the location of source and cache files (default: `/etc/xylem/sources.d/` and `/var/cache/xylem/sources`). Those locations can be either configured by specifying a `prefix` (for FHS comaptible folder layout) or a `xylem_dir` (for layout suitable for in-home-folder configuration).

The source files are ordered mappings of spec plugin names to arguments. In the case of the default `Rules` spec plugin the arguments are simple the rule file URL. For example:

```
# Latest rules in new format
- rules2: 'files://latest/rules/using/new/rules/format/base.yaml'
# Existing rules in legacy format
- rules: 'https://github.com/ros/rosdistro/raw/master/rosdep/base.yaml'
- rules: 'https://github.com/ros/rosdistro/raw/master/rosdep/python.yaml'
- rules: 'https://github.com/ros/rosdistro/raw/master/rosdep/ruby.yaml'
- rosdistro:
    rosdistro_url: 'https://github.com/ros/rosdistro...'
    use_ROSDISTRO_URL: yes
    some_more_optional_arguments: '...'
```

A `RulesDatabase` is initialized given a `SourcesContext`. It loads all source files to create an ordered list of `RulesSource` objects. Each `RulesSource` references the according spec plugin and arguments from the entry in the source file. Moreover, cache and meta data are managed by these objects. The data (== rules specifications) in the `RulesDatabase` can be loaded by invoking the spec plugins. Data and meta information can be saved to and loaded from cache. During `lookup`, all `RulesSource` objects are considered in order and the result merged. `lookup` returns a dictionary mapping installers to installer rules. The installer priority determines which of the returned installers is chosen.

A few simplified code examples to illustrate how this all comes together:

```
def update(prefix=None):
    sources_context = SourcesContext(prefix=prefix)
    sources_context.ensure_cache_dir()

    database = RulesDatabase(sources_context)
    database.update()

def lookup(xylem_key, prefix=None, os_override=None):

    sources_context = SourcesContext(prefix=prefix)
    database = RulesDatabase(sources_context)
    database.load_from_cache()

    ic = InstallerContext(os_override=os_override)

    installer_dict = database.lookup(xylem_key, ic)
    return installer_dict

def resolve(xylem_keys, prefix=None, os_override=None, all_keys=False):

    sources_context = SourcesContext(prefix=prefix)

    database = RulesDatabase(sources_context)
    database.load_from_cache()

    ic = InstallerContext(os_override=os_override)

    if all_keys:
        xylem_keys = database.keys(ic)
```

```
    result = []

    for key in xylem_keys:

        installer_dict = database.lookup(key, ic)

        if not installer_dict:
            raise LookupError("Could not find rule for xylem key '{0}' on "
                              "'{1}'.".format(key, ic.get_os_string()))

        rules = []
        for installer_name, rule in installer_dict.items():
            priority = ic.get_installer_priority(installer_name)
            if priority is None:
                debug("Ignoring installer '{0}' for resolution of '{1}' "
                      "because it is not registered for '{2}'".
                      format(installer_name, key, ic.get_os_string()))
                continue
            if 'priority' in rule:
                priority = rule['priority']

            installer = ic.get_installer(installer_name)
            resolutions = installer.resolve(rule)

            rules.append((priority, installer_name, resolutions))

        if not rules:
            debug("Could not find rule for xylem key '{0}' on '{1}' for "
                  "registered installers '{2}'. Found rules for "
                  "installers '{3}'. Ignoring from 'all' keys.".
                  format(key, ic.get_os_string(),
                         ", ".join(ic.get_installer_names()),
                         ", ".join(installer_dict.keys())))
        else:
            rules.sort(reverse=True)
            result.append((key, rules))

    return sorted(result)
```

Notes:

- 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).
  - *Nikolaus*: I believe it actually has little value at the moment.
- 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`.

Considered design questions:

- 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?

- 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.

Not considered for now:

- It has been considered to include `source plugins` that defines the format / structure of the source files. We have for now decided against it.

1.4.5 Default sources

The idea with default sources plugins is that robot vendors can provide additional default sources including prepackaged cache such that even those default sources work out of the box without initial `update`. How exactly this is realized is tightly related to Sources and cache location.

1.4.6 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 (resolve + dependencies + installer prerequisites checking)
  - options: `--reinstall`, `--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-config` to initialize config file, `sources.list.d` and cache (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` -> resolve a key for os/version; no dependency resolution / prerequisites checking
- `where-defined`

Maybe something to query/change the configuration:

- `config` with the following arguments:
  - `--list-plugins` to list all installed plugins (of all kinds)
  - `--list-sources` list information about all sources that would be considered during update

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.
- there should be some options that tell the user why some key is needed and why it was resolved the way it was resolved

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/`. I.e. the default system wide cache/source location is `/var/cache/xylem` / `/etc/xylem/sources.d`, but the user can configure it to locally be e.g. `~/.xylem/var/cache/xylem` / `.xylem/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.

  - *Dirk:* For rosdistro we actually do the first approach - the environment variable ROSDISTRO_INDEX_URL is also used for API calls (if not overridden by passing a custom index url). I think that approach has the advantage that any tool using rosdistro will use the custom url when it is defined in the environment.

    Wouldn't it be kind of unexepcted if the command line tool xylem uses the prefix from the environment but a different tools like bloom falls back to a different default? Then you would also lack a way to override the prefix for any tool using the API (or that tool would need to expose a custom way to override the prefix).

- 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`?
- When using a user-local cache, locations like `~/.xylem/var/cache/xylem` / `.xylem/etc/xylem/sources.d` are somewhat suboptimal. If we want something like `~/.xylem/cache` / `.xylem/sources.d`, we would likely need separate `XYLEM_SOURCES` and `XYLEM_CACHE` environment variables instead of or alternative to `XYLEM_PREFIX`.
- Additional default sources could also be realized as plugins, which provide source files as well as pickled cache files.

1.5.2 Settings and command line arguments

There should be a canonical way to supply arguments to `xylem`. We propose a system-wide config file, a user config file and command line options. The default settings might be captured on a config file that comes with the installation (this would also give a reference for what settings are available). The order of precedence of settings specified multiple times is:

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

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

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

`xylem` tries to avoid the use of environment variables for configuration. However, in order to allow users of tools like bloom (that make use of the xylem API) to configure xylem, without having those tools expose and pass through xylem-specific arguments, `xylem` uses the `XYLEM_CONFIG` environment variable to optionally point to a config file. There is also a CLI argument `--config`, with the same effect. The CLI argument takes precedence. If a custom location for a `xylem` config file is provided (via `XYLEM_CONFIG` or `--config`), user and system config files are ignored. In that case the order of precedence is:

```
command line > config file > default
```

All command line tools as well as API calls respect the configuration files (either `user > system > default` or `config file > default`). Default configuration can be achieved either by pointing `XYLEM_CONFIG`/`--config` to an empty file or supplying the empty string instead of a path.

Certain `xylem` plugins may respect environment variables, for example the rosdistro spec plugin would by default respect the `ROSDISTRO_INDEX_URL` environment variable.

Where it makes sense, options should be supported both by the CLI and config files.

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).

It also needs to be possible to supply arguments to the installer plugins (e.g. `as-root` or `additional-arguments`, see rosdep#307). Such options may be passed down to those plugins via the `InstallerContext`. The 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 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 Derivative operating systems

OS support e.g. for Ubuntu derivatives should be able to reuse most of the rules for Ubuntu, but maybe overwrite certain rules.

We propose to let OS plugins define a list of increasingly specific names. E.g. a Xubuntu os plugin might define the names `debian`, `ubuntu` and `xubuntu`. The most specific name corresponds to the OS name. It has to be considered that the version names of the less specific OSs might not match the version names of the derivative. In our example, `xubuntu:trusty` corresponds to `ubuntu:trusty`, but does not have a (released-) version correspondence in Debian. Therefore, instead of a list of OS names, os plugins specify a list of tuples of OS names and versions. A `None` version indicates that there is no version correspondence. In that case only `any_version` rules may apply to the derivative. For example, the Xubuntu plugin might return the following list of names/versions on Trusty: `[("debian", None), ("ubuntu", "trusty"), ("xubuntu", "trusty")]`.

The lookup of rules is done in the following way: For a given list of OS names and versions, lookup happens in such a way as if it was first done based on only the names (not versions) independently for each of the specified names (merging information from all sources). Then, the most specific OS name for which some definition exists (no matter for which OS version) is chosen as the sole definition. Only then is the according OS version name considered.

For example, if we have the following rules

```
foo:
  debian: libfoo
bar:
  ubuntu:
    precise: libbar
    trusty: libbar
  xubuntu:
    trusty: libbar-x
```

then on `xubuntu:trusty` the resolutions are `foo -> libfoo` and `bar -> libbar-x`, but on `xubuntu:precise` the key `bar` does not resolve.

1.5.6 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
  - python versions:
    * http://legacy.python.org/dev/peps/pep-0386/
    * http://pythonhosted.org//kitchen/api-versioning.html
- interesting blog about abstract vs concrete dependencies in python https://caremad.io/blog/setup-vs-requirement/

1.5.7 OS options

OS plugins should have options that are configured in the xylem config files. One example of such options are proposed OS "features" (another is "core installers", see Alternative resolutions). Features can be either "active" or not. The config file contains a list of of active features (all other features are inactive). For example, let us consider the Ubuntu OS plugin. For recent Ubuntu versions there exist two alternatives for each python package, one for python 2 and one for python 3. For this example, let us assume that something similar would hypothetically be the case for Ruby 2 and 3. Now if we want to use the latest and greatest, we might therefore put in our `/etc/xylem/config.yaml`:

```
os_options:
  features: [python3, ruby3]
```

In this scenario the Ubuntu plugin defines features `python3` and `ruby3` where their absence implies Python 2 and Ruby 2.

It (probably) does not make sense to define os features in a rules file (like installer options), however, the OS plugins might choose to set default features depending on OS version. For Ubuntu, `python3` might be active by default starting from a certain version. We therefore might also add config options to activate or deactivate certain features (as apposed to only defining the definitive list of features).

In rules files, we allow (optional) conditioning on the features at the OS level. In a shorthand notation (which gets expanded) this might look like the following:

```
rosdep:
  ubuntu: [python-rosdep]
  ubuntu & python3: [python3-rosdep]
  # note that the above is parsed as string "ubuntu & python3"
```

In order to keep things unambiguous, we need to ensure that in each file and for each key/os-name, only at most one rule applies to a configured set of features. In order to achieve this the list of features in os dicts are interpreted in the following way: For a given set of feature dependent os dict entries, we assume that any feature that appears in any of the entries is relevant for all entries. I.e. in the above example the `ubuntu` entry only applies when feature `python3` is not active, because it used in the next line for `ubuntu & python3`. However, since `ruby3` does not appear in any of the entries (in that file, for that key/os), the two rules both apply to feature `ruby3` active and inactive.

While we assume that in practice for each xylem key there is at most one OS feature that is relevant, here is an example of a definition involving two features:

```
mixed-python-ruby-pkg-foo:
  ubuntu: [python-ruby-foo]
  ubuntu & python3, ruby3: [python3-ruby3-foo]
  ubuntu & ruby3: [python-ruby3-foo]
  # note that the above is parsed as string "ubuntu & python3, ruby3"
  # python3-ruby-foo does not exist. List does not have to be exhaustive.
```

In the expanded rules dict, the feature conditions are organized in a binary decision tree (built from valid YAML, but optimized for lookup). Each inner node in the tree consists of a dict with at most 2 or 3 entries: `feature` mapping to the feature that is conditioned on in this node and one or two of `active` and `inactive`, mapping to subtrees for the corresponding decision. The leaves of the tree are version dicts. Since in practice at most one feature is relevant each key, this tree would have depth 0 or 1 for most keys. To illustrate the structure, we show the expanded definition for the example with two features:

```
mixed-python-ruby-pkg-foo:
  ubuntu:
    feature: python3
    active:
      feature: ruby3
      active:
        any_version:
          apt:
            packages: [python3-ruby3-foo]
    inactive:
      feature: ruby3
      active:
        any_version:
          apt:
            packages: [python-ruby3-foo]
      inactive:
        any_version:
          apt:
            packages: [python-ruby-foo]
```

For rules defintions not involving OS features the expaned definition is unchanged, i.e. the version dict comes directly underneath the OS dict.

Here os another example to illustrate the features used are checked on a per-file basis:

```
# 01-rules.yaml
mixed-python-ruby-pkg-foo:
  ubuntu: [python-foo]
# 02-rules.yaml
mixed-python-ruby-pkg-foo:
  ubuntu: [python-ruby-foo]
  ubuntu & python3, ruby3: [python3-ruby3-foo]
  ubuntu & ruby3: [python-ruby3-foo]
# merged result:
mixed-python-ruby-pkg-foo:
  ubuntu: [python-foo]
```

In the above, the first file takes precedence for all cases, even though it does not condition on any features. As explained above, the unconditioned rule in the first file applies to all possible sets of active features.

Notes:

- *Nikolaus*: I am open to suggestions for better compact syntax as well as expanded data structure.
- We might also want to change the OS override syntax to specify features, something like `--os ubuntu:trusty&python3`.
- An alternative proposal to support python 2 vs 3 rules on recent Ubuntu was using derivative OSs, but that doesn't scale very well. Considering multiple alternatives on the same OS, like the hypothetical Ruby 2 vs 3 in the example above, is already awkward, but when this is mixed with actual derivative OSs, it scales very poorly.

  Considering the examlpe above, we might define the following derivative OSs with listed names:

```
ubuntu => ['ubuntu']
ubuntu_py3 => ['ubuntu', 'ubuntu_py3']
ubuntu_rb3 => ['ubuntu', 'ubuntu_rb3']
ubuntu_py3_rb3 => ['ubuntu', 'ubuntu_py3', 'ubuntu_rb3', 'ubuntu_py3_rb3']
```

Now if I want to add an actual derivative OS like Xubuntu, I would also have to add 4 variants:

```
xubuntu => ['ubuntu', 'xubuntu']
xubuntu_py3 => ['ubuntu', 'ubuntu_py3', 'xubuntu', 'xubuntu_py3']
xubuntu_rb3 => ['ubuntu', 'ubuntu_rb3', 'xubuntu', 'xubuntu_rb3']
xubuntu_py3_rb3 => ['ubuntu', 'ubuntu_py3', 'ubuntu_rb3', 'ubuntu_py3_rb3', 'xubuntu', 'xubuntu_
```

Note that with this approach we would also have to include some setting in the config file to guide the os detection to choose the appropriate variant.

1.5.8 Installer options

Installer options configure installer plugins. They can be defined in config files or rules definitions. Definitions in the config files apply to all rules (of that installer). Definitions in rules files only apply to the rules for which they are defined. There is a shortcut to define installer options in rules files that apply to every rule. As an example we consider a set of options to the `apt` installer to support PPAs.

We would like to support custom PPAs for rules. With xylem being ROS- independent, the apt installer plugin has no knowledge of the ROS specific PPAs. We therefore define an option `required_ppas`, which maps to a list of necessary ppas. This list can be populated from xylem config files, or from xylem rules that are currently being resolved. For installation, xylem would then first check that all the required PPAs are installed and possibly offer to install missing ones, or at least give meaningful instructions to the user.

The definition in a config file might look like this:

```
installer_options:
  apt:
    required_ppas: ["ppa:osrf/ros"]
```

An rule in an installer file might look like this:

```
python-rosdep:
  ubuntu:
    any_version:
      apt:
        packages: [python-rosdep]
        required_ppas: ["ppa:osrf/ros"]
```

Recognizing that a rules file might contain many apt rules for packages from the same PPA, rules files may contain global definitions of options for each installer. They act as if they are part of any rule of the corresponding installer. For example, the above file with the single `python-rosdep` entry can alternatively be written:

```
_installer_options:
  apt:
    required_ppas: ["ppa:osrf/ros"]
python-rosdep: [python-rosdep]
```

The leading underscore distinguishes `_installer_options` from xylem keys and ensures that it appears at the top of the file. Having the PPA requirement always be liked to the rules themselves is advantageous. If none of the apt rules in a file with such installer options is part of the set of resolutions of the current `install` command, then of course the PPA requirement is not considered.

Support for mirrors could be added as another option, e.g.:

```
installer_options:
  apt:
    ppa_mirrors:
      "ppa:osrf/ros": ["ppa:freiburg/ros", "ppa:nyu/ros"]
```

Note that these mirrors could be defined in a config file, or again, in the rules file.

Sometimes it might be convenient to not only provide alternative PPAs, but actually replace a PPA with a different one, for example during testing. A third installer option could achieve this:

```
installer_options:
  apt:
    replace_ppas:
      "ppa:osrf/ros": ["ppa:/ros-testing"]
```

Replacing PPAs with a list of 0 or more different PPAs also allows to completely "disable" a PPA requirement without touching the rules files.

Something similar should be done for Taps for Homebrew. While it is possible to reference the Tap with the formula name for installation (`brew install osrf/ros/foopkg`), which should be supported for specific packages.

1.5.9 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, brew –prefix on PATH * pip: pip installed

1.5.10 Alternative resolutions

On a specific platform we want to allow alternative resolutions for the same key. By default some order on these alternatives should determine which resolution is chosen without user interaction. However, it should be possible for the user to permanently or temporarily override that order to install different alternatives.

For example on OS X we would like to offer installing packages either from macports, or from homebrew / pip / gem. Similarly on Ubuntu python packages are packaged with apt, but a user might prefer to install all or certain python packages from pip instead.

We also have to consider that information for which installers are available on each platform and what their priority is should come from multiple sources: os plugins, installer plugins and the user configuration.

We should consider the following use cases:

1. The default scenario should be that the OS plugin defines a preference on supported installers such that each key is resolved uniquely in a way suitable for that platform.
2. Installer plugins for platform independent package managers should be able to specify additional installers independent from the OS plugins. For example a new installer plugin for `go get` should be available on all platforms without the need to update all OS plugins. The installers take lower priority than the default installers for that platform. They are independent from one-another and (by default) don't need arbitration between them. For example, the `go get` installer does need have a relative priority to `pip` or `gem`. Enabling support for new installers should be possible just by installing the corresponding plugin and without additional user configuration.
3. Installer plugins might define installers that are supposed to be used instead of the core installers defined by an OS plugin. For example, someone might write a `linuxbrew` installer plugin and want to use that on Ubuntu instead of `apt`. In that case the installer needs to take higher priority than the OS plugin defined core installers. Requiring user configuration for this rather rare scenario is fine.
4. The user wants to make a different choice for the core installers on a platform where multiple alternatives are provided by xylem. For example while by default the on OS X homebrew together with pip and gem are used, an alternative is to use macports. In that case homebrew should be deactivated completely, i.e. keys that are not provided through macports should not be installed through homebrew, and vice versa, by default keys not available on homebrew should not be installed through macports. However it should still be possible to set up xylem to use homebrew and fallback to macports, if the user desires that. Requiring user configuration for anything non default is fine.
5. The user might want to specify the exact installer to use for specifc keys. E.g. while she uses apt (as per default) on Ubuntu for python packages, a given list of packages should still be installed through pip. The user might do that through config-file / command-line or custom rules file.

Initially, as described in the installer plugin section, a system of real-valued priorities for installers was devised. However it was deemed unnecessarily complex. In particular user configuration should not deal with number-valued priorities for installers.

The following therefore describes the newly proposed interplay of os plugins, installer plugins, rules files and user config to address the above use cases.

The os pluings define an ordered list of "core installers" that are used on this operating system by default. The order determines which installer is used, in case multiple resolutions for one key are available. Moreover, installer plugins may register themselves as "additional installers" for any or all OSs. To that end they define a function that takes os name and version and returns a boolean that indicates if the installer should be used as "additional installer". E.g. installers like `pip` might return `True` for all OSs. The additional installers have arbitrary order/preference among themselves, and lower priority than any of the core installers. If for the resolution of a key there is the choice between multiple additional installers, we might want to either raise an error, or make an arbitrary choice (and possibly provide a warning). Any installers already present in the list of core installers are ignored as "additional installers". This way of configuring core and additional installers should cover use cases 1 and 2.

The user may furthermore override the list of core installers in her config file. For example the following config file specifies that the `linuxbrew` installer should be used with highest precedence, and `apt` as a fallback, and thus supports use case 3 (see also OS options):

```
os_options:
  installers: [linuxbrew, apt]
```

This still allows installer plugins to register themselves as additional installers, i.e. python packages without resolutions for `linuxbrew` nor `apt` can still be installed from pip with the above config. This last part ensures the "no config" requirement of use case 2 (i.e. allow an alternative set of core installers, while still automatically picking up additional installers from newly installed installer plugins without touching the config file).

The above also covers use case 4. In the example the OS plugin for OSX, the core installers would be `[homebrew]` (`pip` and `gem` are additional installers), and can be overwritten in the config file to be `[macports]` by a macports user that does not want to use homebrew at all. Neither macports nor homebrew would register themselves as "additional" installers. More exotically, the user might also configure `[homebrew, pip, gem, macports]` as core installers, to use macports as a fallback for everything not available through homebrew/pip/gem. Even with this setup, the user can take immediately use a newly installer `go get` installer plugin without needing to touch her config file.

To give the user ultimate control over which installers can be used, she might specify that no "additional installers" should be used. With that, the specified list of core installers is the definitive list of used installers:

```
use_additional_installers: False
```

Lastly, we support use case 5, namely overrides for specific keys, in two ways. Firstly, the user can specify a list of keys to install from specific installers either in the config file

```
install_from:
  pip: [python-foo, python-bar]
  gem: [ruby-baz]
```

or on the command line:

```
xylem install some-pkg-with-deps --install-from="pip: [python-foo, python-bar]"
```

This is useful to quickly choose a different resolution for some keys, where this resolution is already defined, but not the highest priority with the current installer precedence setup.

A slightly different scenario is where one wants to override the resolution for a specific key in a custom rules file and make sure that this is the only rule, not merged with rules for other installers in existing rules files. For example:

```
# 01-local-rules.yaml
foo:
  ubuntu:
    any_version:
      my_installer: ["libfoo"]
# 20-default-rules.yaml
...
foo:
  ubuntu:
    any_version:
      apt: ["libfoo"]
...
```

In this setup, the rules for `my_installer` and `apt` would get merged upon lookup of `foo`. Since `apt` is the highest priority installer on Ubuntu, xylem would always choose the apt rule. In order to support true overriding of specific keys in local rules files (like is currently possible in rosdep) in a way that does not require to also list those keys in a config file, we propose a special `disable` keyword for installer rules, which makes sure that a definition further down the line is not merged. A special `any_installer` entry in the installer dict can be used to disable all other installers. So in our example the `01-local-rules.yaml` can be written as either

```
foo:
  ubuntu:
    any_version:
      my_installer: ["libfoo"]
      apt:
        disable: True
```

or

```
foo:
  ubuntu:
    any_version:
      my_installer: ["libfoo"]
      any_installer:
        disable: True
```

to achieve the desired effect.

Note that on top of the above configuration possibilities, there are also other ways in which power users might influence the list of installers. E.g. one can provide a custom os plugin and disable to one provided by xylem. The same can be done with installer plugins, to e.g. write a custom plugin for the `apt` installer and disable the one shipped with python. This last option allows to use all the existing rules for `apt` while fully customizing how the apt packages are actually installed (maybe someone wants to install all apt packages from source for some reason). Having to write and replace plugins should however not be the workflow for common use cases, which is granted by the above proposal.

Notes:

- The `xylem resolve` command can optionally list all alternative resolution and their order in order to debug.
- In the context of bloom, keep in mind that for debian releases only apt dependencies are allowed, whereas e.g. homebrew formulae can also depend on pip / gem).

1.5.11 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
- restriction on the characters used in xylem keys, os names, installer names, version strings: alphanumeric, period, dash, underscore. Is this too restrictive? Reserved names such as any_os, any_version, default_installer...
- for the rosdistro plugin, there should be a more meaningful error message when an operating system is not supported (it should not just be "key not resolved", nor should it simply try to install non- existent packages (and fail) like it does now on homebrew)
- before releasing, carefully consider security and ability for plugins to override completely what is installed from sources
- consider migration path ros-package -> system dependencies (in light of xylem supporting multiple ros distros) http://answers.ros.org/question/173773/depend-on-opencv-in-hydro/
- Look at Chef cookbook http://answers.ros.org/question/174507/is-there-interest-in-maintaining-chef-cookbooks-for-ros/

1.6 Terminology

[TODO: Define terms]

- xylem key
- key database
- rules file
- installer
- package manager
- platform –> os/version tuple
- installer
- installer context
- package -> pm package
- rules dict, os dict, version dict, installer dict, installer rule
- rules database (contains merged rules dict)
- rules source (entry in sources file, contains spec plugin name and data, typically url, must should have unique identifier)
- cache -> version, datetime, must be reproducible for the unique identifier

CHAPTER 2

xylem Rules files

2.1 Notes

- Rules files compacted with this syntax can be found here: https://github.com/NikolausDemmel/rosdistro/tree/xylem/rosdep
- With the proposed rules files the following entries are not valid any more. `homebrew` is interpreted as a version of osx and `packages` as the package-manager. No detection of this problem happens at the moment:

```
boost:
  osx:
    homebrew:
      packages: [boost]
```

  Correct would be:

```
boost:
  osx:
    any_version:
      homebrew:
        packages: [boost]
```

- `'any_version'` is somewhat limited in some cases:

```
gazebo:
  ubuntu:
    precise: [gazebo]
    quantal: [gazebo]
    raring: [gazebo]
    saucy: [gazebo2]
    trusty: [gazebo2]
```

  Using `any_version` for `saucy` and `trusty` does not contain information from which version the rule has been tested/confirmed. In this case it would also apply to `precise`, whereas the explicit list above would give a more meaningful error on precise ("No rules definition" instead of "Failed to install apt package gazebo2").

- Here is another limitation, where rules for all versions but the latest are the same:

```
ffmpeg:
  ubuntu:
    lucid: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    maverick: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    natty: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    oneiric: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    precise: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
```

```
    quantal: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    raring: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    saucy: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    trusty: [libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
```

We might want to allow tuple/list/set as version dict key or maybe range of versions `'lucid - saucy'` to alleviate that problem.

- For implementing verions ranges part of the problem is that versions are not known and cannot be listed, only detected. Can we change that easily? OS plugins could list all known versions, but that would require adding any new version explicitly. Maybe not so bad?
- Should special keys have special syntax like `*any_os*` or similar? + *Nikolaus*: IMHO no

2.2 any_version and version ranges

The above notes mention some problems with the initial proposal for `any_version`.

Firstly, `any_version` applies to all version, even old ones for which it the rule has not been tested. `any_version` is often used to make sure the rule applies to any newly released versions and therefore used instead of a definition for the currently latest version. This makes sure that when a new OS version is released, only the keys for which the rule actually has to change need to be touched in the rules file. At the moment it is not possible to express a minimal version for which the `any_version` rule is valid. Therefore, replacing a list of explicit definitions for each version with a `any_version` rule actually loses information.

Secondly, in the above example of `ffmpeg`, only the latest version has changed. Since we want to use `any_version` for the latest version, we have to retain the repeated explicit definition for all other versions (which are identical).

We propose the following two changes to alleviate those problems.

2.2.1 any_version with greater or equal condition

Firstly, the dictionary underneath `any_version` may optionally have a key `version_geq` mapping to a minimal version to which the rule applies. The installer dict is then pushed down one level accessed by a `installers` key. For example:

```
gazebo:
  ubuntu:
    precise: [gazebo]
    quantal: [gazebo]
    raring: [gazebo]
    any_version:
      version_geq: saucy
      installers:
        apt:
          packages: [gazebo2]
```

There exists a short notation without the intermediary dictionary:

```
gazebo:
  ubuntu:
    precise: [gazebo]
    quantal: [gazebo]
    raring: [gazebo]
    any_version>=saucy: [gazebo2]
    # note that the above is parsed as string "any_version>=saucy"
```

```
gazebo:
  ubuntu:
    precise: [gazebo]
    quantal: [gazebo]
    raring: [gazebo]
    any_version>=saucy:
      apt:
        packages: [gazebo2]
```

The both of the above shorthands expand to the initial example. There is also a short notation at the os dict level:

```
gazebo:
  ubuntu>=saucy: [gazebo2]
```

which expands to:

```
gazebo:
  ubuntu:
    any_version:
      version_geq: saucy
      installers:
        apt:
          packages: [gazebo2]
```

This means that for rule lookup the order on versions needs to be known. Therefore, each os plugin needs to provide an order over its version strings. For systems like OS X, that implies listing all known versions in the OS plugin.

Note that the order is *not* needed for rules file *expansion*.

2.2.2 Multiple versions in one defintion

We allow to define multiple versions in one definition by allowing the keys in the version dict to be a comma separated list of versions (as a string). Upon rule expansion the definitions are separated as one definition for each listed version. We do *not* provide a way to specify version ranges (like `precise - saucy`) to keep the implied versions explicit. This also helps to not require the list of all versions for rule expansion.

For example, the `ffmpeg` definition can be compacted as:

```
ffmpeg:
  ubuntu:
    any_version>=trusty: [libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    lucid, maverick, natty, oneiric, precise, quantal, raring, saucy: [ffmpeg, libavcodec-dev, libavi
```

which expands to:

```
ffmpeg:
  ubuntu:
    any_version>=trusty: [libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    lucid: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    maverick: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    natty: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    oneiric: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    precise: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    quantal: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    raring: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
    saucy: [ffmpeg, libavcodec-dev, libavformat-dev, libavutil-dev, libswscale-dev]
```

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.`**`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

3.2 Indices and tables

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

CHAPTER 4

xylem

4.1 xylem package

4.1.1 Subpackages

xylem.commands package

Submodules

xylem.commands.lookup module

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.resolve module

xylem.commands.resolve.**main**(*args=None*)

xylem.commands.resolve.**parse_os_tuple**(*os_arg*)

xylem.commands.resolve.**prepare_arguments**(*parser*)

xylem.commands.update module

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

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

Module contents

xylem.installers package

Subpackages

xylem.installers.plugins package

Submodules

xylem.installers.plugins.fake module This is a fake installer plugin for testing.

It is able to install any package and 'installs'/'removes' packages by touching/removing files in a folder.

var INSTALL_LOCATION the installation folder

var definition definition of the installer plugin to be referenced by the according entry point

class xylem.installers.plugins.fake.**FakeInstaller**
Bases: xylem.installers.package_manager_installer.PackageManagerInstaller

FakeInstaller class for testing.

The opaque installer items are simply strings (package names).

Packages are installed by touching files in INSTALL_LOCATION. The folder must exist, else installation fails and all packages are assumed uninstalled.

get_install_command(*resolved*, *interactive=True*, *reinstall=False*)

static get_name()

xylem.installers.plugins.fake.**detect_fn**(*resolved*)
Return list of subset of installed packages.

xylem.installers.plugins.fake.**get_installer_filename**(*resolved_item*)
Return the location of the file indicating installation of item.

xylem.installers.plugins.homebrew module TODO: Describe and implement this

var definition definition of the installer plugin to be referenced by the according entry point

class xylem.installers.plugins.homebrew.**HomebrewInstaller**
Bases: xylem.installers.package_manager_installer.PackageManagerInstaller

get_install_command(*resolved*, *interactive=True*, *reinstall=False*)

static get_name()

xylem.installers.plugins.homebrew.**fixme_detect**(*pkgs*, *exec_fn=None*)

xylem.installers.plugins.macports module TODO: Describe and implement this

var definition definition of the installer plugin to be referenced by the according entry point

class xylem.installers.plugins.macports.**MacportsInstaller**
Bases: xylem.installers.package_manager_installer.PackageManagerInstaller

get_install_command(*resolved*, *interactive=True*, *reinstall=False*)

static get_name()

xylem.installers.plugins.macports.**fixme_detect**(*pkgs*, *exec_fn=None*)

xylem.installers.plugins.pip module This is a installer plugin for the pip python package manager.

See https://pypi.python.org/pypi/pip

var definition definition of the installer plugin to be referenced by the according entry point

class xylem.installers.plugins.pip.**PipInstaller**
Bases: xylem.installers.package_manager_installer.PackageManagerInstaller

Installer support for pip.

get_install_command(*resolved*, *interactive=True*, *reinstall=False*)

static get_name()

get_priority_for_os(*os_name*, *os_version*)

xylem.installers.plugins.pip.**is_pip_installed**()
Return True if 'pip' can be executed.

xylem.installers.plugins.pip.**pip_detect**(*pkgs*, *exec_fn=None*)
Given a list of package, return the list of installed packages.

Parameters **exec_fn** – function to execute Popen and read stdout (for testing)

Module contents

Submodules

xylem.installers.impl module

class xylem.installers.impl.**Installer**
Bases: builtins.object

Installer class that custom installer plugins derive from.

The Installer API is designed around opaque *resolved* parameters. These parameters can be any type of sequence object, but they must obey set arithmetic. They should also implement __str__() methods so they can be pretty printed.

get_depends(*rule_args*)
Get list list of dependencies on other xylem keys.

Parameters **rule_args** (*dict*) – argument dictionary to the xylem rule for this package manager

Returns List of dependencies on other xylem keys. Only necessary if the package manager doesn't handle dependencies.

Return type list of str

get_install_command(*resolved*, *interactive=True*, *reinstall=False*)
Get command line invocations to install list of items.

Parameters

- **resolved** – [resolution]. List of opaque resolved installation items
- **interactive** – If False, disable interactive prompts, e.g. pass through -y or equivalent to package manager.

- **reinstall** – If `True`, install everything even if already installed

Returns List of commands, each command being a list of strings.

Return type `[[str]]`

static get_name()
Get the name of the installer this class implements.

This is the name that is referenced in the rules files, user configuration or OS plugins. There may only be one installer for a any given name at runtime, i.e. plugins defining installers with existing names might be ignored.

Return str installer name

get_priority_for_os(*os_name*, *os_version*)
Get the priority of this installer according to installer plugin.

Given an OS name/version tuple, the installer can declare that it should be used on that OS with the returned priority. If the installer does not want to declare itself for this OS, None is returned.

Return type number or None

is_installed(*resolved_item*)
Check if single opaque installation item is installed.

Parameters **resolved_item** – single opaque resolved installation item

Returns `True` if all of the *resolved* items are installed on the local system

resolve(*rule_args*)
Return list of resolutions from rules dictionary entry.

Parameters **rule_args** (*dict*) – argument dictionary to the xylem rule for this package manager

Returns [resolution]. Resolved objects should be printable to a user, but are otherwise opaque.

class xylem.installers.impl.**InstallerContext**(*setup_installers=True*, *os_override=None*)
Bases: builtins.object

`InstallerContext` manages the context of execution for xylem.

It combines OS plugins, installer plugins and user settings to manage the current OS, installers to be used including their priorities.

get_default_installer_name()
Get name of default installer for current os.

`setup_installers()` needs to be called beforehand.

get_installer(*name*)
Get installer object by name.

get_installer_names()
Get all configured installers for current os.

`setup_installers()` needs to be called beforehand.

get_installer_priority(*name*)
Get configured priority for specific installer and current os.

`setup_installers()` needs to be called beforehand.

get_os_string()
Get the OS name and version as 'name:version' string.

See `get_os_tuple()`

Return type str

Raises UnsupportedOsError if OS was not detected correctly

get_os_tuple()
Get the OS (name,version) tuple.

Return the OS name/version tuple to use for resolution and installation. This will be the detected OS name/version unless `InstallerContext.set_os_override()` has been called.

Returns (os_name, os_version)

Return type (str,str)

Raises UnsupportedOsError if OS was not detected correctly

set_os_override(*os_tuple*)
Override the OS detector with *os_name* and *os_version*.

See `InstallerContext.detect_os()`.

Parameters os_name (*(str,str)*) – OS (name,version) tuple to use

Raises UnsupportedOsError if os override was invalid

setup_installers()
For current os, setup configured installers.

Installers are set up with their priorities for the current os and based on user config, os plugins and installer plugins.

xylem.installers.impl.**get_installer_plugin_list**()
Return list of Installer plugin objects unique by name.

Load the Installer plugin descriptions from entry points, instantiating objects and ignoring duplicates (by Installer.get_name(), not entry point name).

Returns list of the loaded plugin objects

Raises InvalidPluginError if one of the loaded plugins is invalid

xylem.installers.impl.**load_installer_plugin**(*entry_point*)
Load Installer plugin from entry point.

Parameters entry_point – entry point object from `pkg_resources`

xylem.installers.package_manager_installer module

class xylem.installers.package_manager_installer.**PackageManagerInstaller**(*detect_fn*, *supports_depends=False*)
Bases: `xylem.installers.impl.Installer`

Base class from a variety of package manager installers.

General form of a package manager `Installer` implementation that assumes:

- installer rule-args spec is a list of package names stored with the key "packages"
- a detect function exists that given a list of packages, returns a list of the installed packages

Also, if *supports_depends* is set to `True`:

- installer rule-args spec can also include dependency specification with the key "depends"

Subclasses need to provide implementation of `get_install_command`.

In addition, if subclass provide their own `resolve` method, the resolved items need not be package names (i.e. strings). Methods other than `get_isntall_command`, `resolve` and the `detect_fn` treat the resolved items as opaque objects.

get_depends(*rule_args*)
Get list list of dependencies on other xylem keys.

Parameters **rule_args** (*dict*) – argument dictionary to the xylem rule for this package manager

Returns List of dependencies on other xylem keys read from the 'depends' key in `rule_args` if `self.supports_depends` is `True`.

get_packages_to_install(*resolved*, *reinstall=False*)

is_installed(*resolved_item*)

resolve(*rules_args*)
See `Installer.resolve()`.

Module contents

class xylem.installers.**InstallerContext**(*setup_installers=True*, *os_override=None*)
Bases: `builtins.object`

`InstallerContext` manages the context of execution for xylem.

It combines OS plugins, installer plugins and user settings to manage the current OS, installers to be used including their priorities.

get_default_installer_name()
Get name of default installer for current os.

`setup_installers()` needs to be called beforehand.

get_installer(*name*)
Get installer object by name.

get_installer_names()
Get all configured installers for current os.

`setup_installers()` needs to be called beforehand.

get_installer_priority(*name*)
Get configured priority for specific installer and current os.

`setup_installers()` needs to be called beforehand.

get_os_string()
Get the OS name and version as 'name:version' string.

See `get_os_tuple()`

Return type str

Raises **UnsupportedOsError** if OS was not detected correctly

get_os_tuple()
Get the OS (name,version) tuple.

Return the OS name/version tuple to use for resolution and installation. This will be the detected OS name/version unless `InstallerContext.set_os_override()` has been called.

Returns (os_name, os_version)

Return type (str,str)

Raises UnsupportedOsError if OS was not detected correctly

set_os_override(*os_tuple*)
Override the OS detector with *os_name* and *os_version*.

See `InstallerContext.detect_os()`.

Parameters **os_name** (*(str,str)*) – OS (name,version) tuple to use

Raises UnsupportedOsError if os override was invalid

setup_installers()
For current os, setup configured installers.

Installers are set up with their priorities for the current os and based on user config, os plugins and installer plugins.

class xylem.installers.**Installer**
Bases: `builtins.object`

Installer class that custom installer plugins derive from.

The `Installer` API is designed around opaque *resolved* parameters. These parameters can be any type of sequence object, but they must obey set arithmetic. They should also implement `__str__()` methods so they can be pretty printed.

get_depends(*rule_args*)
Get list list of dependencies on other xylem keys.

Parameters **rule_args** (*dict*) – argument dictionary to the xylem rule for this package manager

Returns List of dependencies on other xylem keys. Only necessary if the package manager doesn't handle dependencies.

Return type list of str

get_install_command(*resolved*, *interactive=True*, *reinstall=False*)
Get command line invocations to install list of items.

Parameters

- **resolved** – [resolution]. List of opaque resolved installation items
- **interactive** – If `False`, disable interactive prompts, e.g. pass through `-y` or equivalent to package manager.
- **reinstall** – If `True`, install everything even if already installed

Returns List of commands, each command being a list of strings.

Return type `[[str]]`

static get_name()
Get the name of the installer this class implements.

This is the name that is referenced in the rules files, user configuration or OS plugins. There may only be one installer for a any given name at runtime, i.e. plugins defining installers with existing names might be ignored.

Return str installer name

get_priority_for_os(*os_name*, *os_version*)
Get the priority of this installer according to installer plugin.

Given an OS name/version tuple, the installer can declare that it should be used on that OS with the returned priority. If the installer does not want to declare itself for this OS, None is returned.

Return type number or None

is_installed(*resolved_item*)
Check if single opaque installation item is installed.

Parameters **resolved_item** – single opaque resolved installation item

Returns `True` if all of the *resolved* items are installed on the local system

resolve(*rule_args*)
Return list of resolutions from rules dictionary entry.

Parameters **rule_args** (*dict*) – argument dictionary to the xylem rule for this package manager

Returns [resolution]. Resolved objects should be printable to a user, but are otherwise opaque.

xylem.os_support package

Submodules

xylem.os_support.impl module

class xylem.os_support.impl.**OS**
Bases: builtins.object

Abstract OS plugin base class.

OS plugins should define entry points as classes derived from this.

Operating systems are described by a list of increasingly specific names, where the most specific of those is referred to as the name of the operating system. The description furthermore includes a operating system version, which can be a version number string or code name.

Operating systems can name their default installer and furthermore list additional applicable installer names, each with a number as priority (higher number take precedence).

get_default_installer_name()
Get name of default installer as described by OS plugin.

Return type str

get_installer_priority(*installer_name*)
Get priority of installer as described by OS plugin.

Parameters **installer_name** (*str*) – name of installer in question

Returns priority of this installer if the os defines it, else None

Return type number or None

get_name()
Get the most specific name of the described operating system.

Return type string

get_names()
Get a list of names describing this operating system.

Returns list of increasingly specific os names

Return type list of strings

get_tuple()
Get (name,version) tuple.

Return type (str,str)

get_version()
Get version of this operating system.

Return type string

is_os()
Return true if the current OS matches the one this object describes.

Return type bool

class xylem.os_support.impl.**OSSupport**
Bases: builtins.object

OSSupport manages the OS plugins and options such as override_os.

Can detect the current OS from the installed OS plugins or use the override option. Moreover manages options such as disabling specific plugins.

In order to set up, either call `detect_os()` or `override_os()` and subsequently access it with `current_os()`

detect_os()
Detects and sets the current OS.

The first OS plugin that returns `True` for `OS.is_os()` is the detected one. If multiple os plugins would accept the current OS, a warning is printed to the user.

Raises UnsupportedOSError If no OS plugin accepts the current OS

get_current_os()
Return OS object of current OS.

Detect current OS if not yet detected or overridden.

Return type OS

Raises UnsupportedOSError If OS is not set and cannot be detected.

get_default_installer_names()
Return mapping of os name to default installer for all os.

get_os_plugin(*name*)
Return os plugin object for given os name or None if not known.

get_os_plugin_names()
Return list of known/configured os names.

get_os_plugins()
Return list of is plugin objects.

override_os(*os_tuple*)
Override to to (name,version) tuple.

A plugin with `name` must be installed.

Raises UnsupportedOSError if specified os name is not known

class xylem.os_support.impl.**OverrideOS**(*os*, *version*)
Bases: xylem.os_support.impl.OS

Special OS class that acts as a proxy to another OS with fixed version.

OverrideOS takes another OS object and delegates all queries to that, except for detection and version, which are fixed by the OverrideOS.

get_default_installer_name()
Return the delegate's default installer.

get_installer_priority(*installer_name*)
Return the delegate's installer priority.

get_name()
Return the delegate's name.

get_names()
Return the delegate's names.

get_version()
Return the saved version from setup.

is_os()
Detection for OverrideOS is always True.

exception xylem.os_support.impl.**UnsupportedOSError**
Bases: builtins.Exception

Operating system unsupported.

Detected operating system is not supported or could not be identified.

xylem.os_support.impl.**get_os_plugin_list**()
Return list of OS plugin objects unique by name.

Load the os plugin classes from entry points, instantiating objects and ignoring duplicates (by os.name(), not entry point name).

Returns list of the loaded plugin objects

Raises InvalidPluginError if one of the loaded plugins is invalid

xylem.os_support.impl.**load_os_plugin**(*entry_point*)
Load OS plugin from entry point.

Parameters **entry_point** – entry point object from pkg_resources

Raises InvalidPluginError if the plugin is not valid

xylem.os_support.os_detect module

Library for detecting the current OS, including detecting specific Linux distributions.

class xylem.os_support.os_detect.**Arch**(*release_file='/etc/arch-release'*)
Bases: xylem.os_support.os_detect.OsDetector

Detect Arch Linux.

get_codename()

get_version()

is_os()

class xylem.os_support.os_detect.**Cygwin**
Bases: xylem.os_support.os_detect.OsDetector

Detect Cygwin presence on Windows OS.

get_codename()

get_version()

is_os()

class xylem.os_support.os_detect.**Fedora**(*release_file='/etc/redhat-release'*, *issue_file='/etc/issue'*)
Bases: xylem.os_support.os_detect.OsDetector

Detect Fedora OS.

get_codename()

get_version()

is_os()

class xylem.os_support.os_detect.**FreeBSD**(*uname_file='/usr/bin/uname'*)
Bases: xylem.os_support.os_detect.OsDetector

Detect FreeBSD OS.

get_codename()

get_version()

is_os()

class xylem.os_support.os_detect.**Gentoo**(*release_file='/etc/gentoo-release'*)
Bases: xylem.os_support.os_detect.OsDetector

Detect Gentoo OS.

get_codename()

get_version()

is_os()

class xylem.os_support.os_detect.**LsbDetect**(*lsb_name*, *get_version_fn=None*)
Bases: xylem.os_support.os_detect.OsDetector

Generic detector for Debian, Ubuntu, and Mint

get_codename()

get_version()

is_os()

class xylem.os_support.os_detect.**OSX**(*sw_vers_file='/usr/bin/sw_vers'*)
Bases: xylem.os_support.os_detect.OsDetector

Detect OS X

get_codename()

get_version()

is_os()

class xylem.os_support.os_detect.**OpenSuse**(*brand_file='/etc/SuSE-brand'*, *release_file='/etc/SuSE-release'*)

Bases: xylem.os_support.os_detect.OsDetector

Detect OpenSuse OS.

get_codename()

get_version()

is_os()

class xylem.os_support.os_detect.**OsDetect**(*os_list=None*)

Bases: builtins.object

This class will iterate over registered classes to lookup the active OS and version

add_detector(*name*, *detector*)

Add detector to list of detectors used by this instance. *detector* will override any previous detectors associated with *name*.

Parameters

- **name** – OS name that detector matches
- **detector** – OsDetector instance

default_os_list = [('windows', <xylem.os_support.os_detect.Windows object at 0x7fc678aa07b8>), ('ubuntu', <xyler

detect_os(*env=None*)

Detect operating system. Return value can be overridden by the **:env:'ROS_OS_OVERRIDE'** environment variable.

Parameters **env** – override os.environ

Returns (os_name, os_version, os_codename), (str, str, str)

Raises OsNotDetected if OS could not be detected

get_codename()

get_detector(*name=None*)

Get detector used for specified OS name, or the detector for this OS if name is None.

Raises KeyError

get_name()

get_version()

static register_default(*os_name*, *os_detector*)

Register detector to be used with all future instances of OsDetect. The new detector will have precedence over any previously registered detectors associated with *os_name*.

Parameters

- **os_name** – OS key associated with OS detector
- **os_detector** – OsDetector instance

class xylem.os_support.os_detect.**OsDetector**

Bases: builtins.object

Generic API for detecting a specific OS.

get_codename()

Returns codename for this OS. (ala Ubuntu Hardy Heron = "hardy"). If codenames are not available for this OS, return empty string.

Raises `OsNotDetected` if called on incorrect OS.

get_version()

Returns standardized version for this OS. (ala Ubuntu Hardy Heron = "8.04")

Raises `OsNotDetected` if called on incorrect OS.

is_os()

Returns if the specific OS which this class is designed to detect is present. Only one version of this class should return for any version.

exception xylem.os_support.os_detect.**OsNotDetected**
Bases: builtins.Exception

Exception to indicate failure to detect operating system.

class xylem.os_support.os_detect.**QNX**(*uname_file='/bin/uname'*)
Bases: xylem.os_support.os_detect.OsDetector

Detect QNX realtime OS. @author: Isaac Saito

get_codename()

get_version()

is_os()

class xylem.os_support.os_detect.**Rhel**(*release_file='/etc/redhat-release'*)
Bases: xylem.os_support.os_detect.Fedora

Detect Redhat OS.

get_codename()

get_version()

is_os()

class xylem.os_support.os_detect.**Windows**
Bases: xylem.os_support.os_detect.OsDetector

Detect Windows OS.

get_codename()

get_version()

is_os()

xylem.os_support.os_detect.**read_issue**(*filename='/etc/issue'*)

Returns list of strings in issue file, or None if issue file cannot be read/split

xylem.os_support.os_detect.**uname_get_machine**()
Linux: wrapper around uname to determine if OS is 64-bit

xylem.os_support.plugins module

class xylem.os_support.plugins.**Debian**
Bases: xylem.os_support.plugins.OSBase

class xylem.os_support.plugins.**OSBase**
Bases: xylem.os_support.impl.OS

OS plugin base class for builtin plugins.

This is an internal base class used for the plugins shipped with xylem, which use the os_detect module. In general, external plugins would want to derive from OS directly.

Derived classes should fill in the following member variables:

Variables

- **names** (*list(str)*) – list of names
- **detect** – Detector object supporting is_os(), get_version() and get_codename()
- **use_codename** (*bool*) – boolean to decide if numbered version or codename should be used
- **installer_priorities** (*dict*) – dict of installer_name => priority
- **default_installer_name** (*str*) – name of the desired default installer

get_default_installer_name()

get_installer_priority(*installer_name*)

get_name()

get_names()

get_version()

is_os()

class xylem.os_support.plugins.**OSX**
Bases: xylem.os_support.plugins.OSBase

class xylem.os_support.plugins.**Ubuntu**
Bases: xylem.os_support.plugins.Debian

Module contents

Module to manage OS plugins and their use for OS detection.

class xylem.os_support.**OS**
Bases: builtins.object

Abstract OS plugin base class.

OS plugins should define entry points as classes derived from this.

Operating systems are described by a list of increasingly specific names, where the most specific of those is referred to as the name of the operating system. The description furthermore includes a operating system version, which can be a version number string or code name.

Operating systems can name their default installer and furthermore list additional applicable installer names, each with a number as priority (higher number take precedence).

get_default_installer_name()
Get name of default installer as described by OS plugin.

Return type str

get_installer_priority(*installer_name*)
Get priority of installer as described by OS plugin.

Parameters **installer_name** (*str*) – name of installer in question

Returns priority of this installer if the os defines it, else None

Return type number or None

get_name()
Get the most specific name of the described operating system.

Return type string

get_names()
Get a list of names describing this operating system.

Returns list of increasingly specific os names

Return type list of strings

get_tuple()
Get (name,version) tuple.

Return type (str,str)

get_version()
Get version of this operating system.

Return type string

is_os()
Return true if the current OS matches the one this object describes.

Return type bool

class xylem.os_support.**OSSupport**
Bases: builtins.object

OSSupport manages the OS plugins and options such as override_os.

Can detect the current OS from the installed OS plugins or use the override option. Moreover manages options such as disabling specific plugins.

In order to set up, either call `detect_os()` or `override_os()` and subsequently access it with `current_os()`

detect_os()
Detects and sets the current OS.

The first OS plugin that returns `True` for `OS.is_os()` is the detected one. If multiple os plugins would accept the current OS, a warning is printed to the user.

Raises UnsupportedOSError If no OS plugin accepts the current OS

get_current_os()
Return OS object of current OS.

Detect current OS if not yet detected or overridden.

Return type OS

Raises UnsupportedOSError If OS is not set and cannot be detected.

get_default_installer_names()
Return mapping of os name to default installer for all os.

get_os_plugin(*name*)
Return os plugin object for given os name or None if not known.

get_os_plugin_names()
Return list of known/configured os names.

get_os_plugins()
Return list of is plugin objects.

override_os(*os_tuple*)
Override to to (name,version) tuple.

A plugin with `name` must be installed.

Raises UnsupportedOSError if specified os name is not known

exception xylem.os_support.**UnsupportedOSError**
Bases: builtins.Exception

Operating system unsupported.

Detected operating system is not supported or could not be identified.

xylem.sources package

Submodules

xylem.sources.database module

class xylem.sources.database.**RulesDatabase**(*sources_context*)
Bases: builtins.object

init_from_sources()

keys(*installer_context*)
Return list of keys defined for current os/version.

load_from_cache()

load_from_source()

lookup(*xylem_key*, *installer_context*)
Return rules for xylem key in current os.

save_to_cache()

update()

verify_unique_ids()

class xylem.sources.database.**RulesSource**(*spec*, *arguments*, *origin*, *sources_context*)
Bases: builtins.object

cache_file_path()

clear_cache()
Remove cache file.

Raises OSError if cache file cannot be removed

is_cache_available()

is_cache_outdated()

keys(*installer_context*)
Return list of keys defined for current os/version.

load_from_cache()

load_from_source()

lookup(*xylem_key*, *installer_context*)

save_to_cache()

unique_id()

xylem.sources.impl module

class xylem.sources.impl.**SourcesContext**(*prefix=None*, *xylem_dir=None*, *spec_plugins=None*)
Bases: builtins.object

cache_dir_exists()

ensure_cache_dir()

get_spec(*spec_name*)

is_default_dirs()

setup_paths(*prefix=None*, *xylem_dir=None*)

sources_dir_exists()

exception xylem.sources.impl.**UnknownSpecError**
Bases: builtins.Exception

xylem.sources.impl.**cache_dir_from_prefix**(*prefix*)

xylem.sources.impl.**cache_dir_from_xylem_dir**(*xylem_dir*)

xylem.sources.impl.**get_default_source_descriptions**()
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_descriptions**(*sources_dir*)
Return a list of source urls.

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_sources_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_source_descriptions**(*data*, *file_path='<string>'*)
Parse a YAML string as source descriptions.

If parsing failes an error message is printed to console and an empty list is returned.

Parameters

- **data** (*str*) – string containing YAML representation of source descriptions
- **file_path** (*str*) – name of the file whose contents `data` contains

Returns tuple of `file_path` and parsed source descriptions

Return type `tuple(str, list)`

`xylem.sources.impl.`**`parse_source_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`))

`xylem.sources.impl.`**`sources_dir_from_prefix`**(*prefix*)

`xylem.sources.impl.`**`sources_dir_from_xylem_dir`**(*xylem_dir*)

`xylem.sources.impl.`**`verify_source_description`**(*descr*)
Verify that a source description has valid structure.

Parameters **descr_list** (*dict*) – source description

Raises ValueError if structure of source description is invalid

`xylem.sources.impl.`**`verify_source_description_list`**(*descr_list*)
Verify that a source description list has valid structure.

Parameters **descr_list** (*list*) – list of source descriptions

Raises ValueError if structure of source descriptions is invalid

xylem.sources.rules_dict module

Module contents

class `xylem.sources.`**`SourcesContext`**(*prefix=None*, *xylem_dir=None*, *spec_plugins=None*)
Bases: `builtins.object`

`cache_dir_exists`()

`ensure_cache_dir`()

`get_spec`(*spec_name*)

`is_default_dirs`()

`setup_paths`(*prefix=None*, *xylem_dir=None*)

`sources_dir_exists`()

class `xylem.sources.`**`RulesDatabase`**(*sources_context*)
Bases: `builtins.object`

`init_from_sources`()

`keys`(*installer_context*)
Return list of keys defined for current os/version.

`load_from_cache`()

load_from_source()

lookup(*xylem_key*, *installer_context*)
Return rules for xylem key in current os.

save_to_cache()

update()

verify_unique_ids()

xylem.specs package

Submodules

xylem.specs.impl module

class xylem.specs.impl.**Spec**
Bases: xylem.plugin_utils.PluginBase

Spec plugin abstract base class.

Spec plugins are stateless classes such that all functions get are their needed parameters passed on every invocation.

The data and arguments (e.g. url for the 'rules' spec plugin) are managed by the sources.database.RulesSource class in the sources.database.RulesDatabase.

is_data_outdated(*data*, *arguments*, *data_load_time*)

keys(*data*, *installer_context*)
Return list of keys defined for current os/version.

load_data(*arguments*)

lookup(*data*, *xylem_key*, *installer_context*)

name

unique_id(*arguments*)

verify_arguments(*arguments*)

verify_data(*data*, *arguments*)

version

xylem.specs.impl.**get_spec_plugin_list**()
Return list of spec plugin objects unique by name.

See get_plugin_list()

xylem.specs.impl.**verify_spec_name**(*spec_name*)
Verify that a spec_name is valid spec name.

Parameters **spec_name** (*str*) – spec name

Raises **ValueError** if spec name is invalid

xylem.specs.rules module

Module contents

xylem.specs.**get_spec_plugin_list**()
Return list of spec plugin objects unique by name.

See get_plugin_list()

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

Raised when an invalid spec element is encountered while parsing.

xylem.specs.**verify_spec_name**(*spec_name*)
Verify that a spec_name is valid spec name.

Parameters **spec_name** (*str*) – spec name

Raises **ValueError** if spec name is invalid

class xylem.specs.**Spec**
Bases: xylem.plugin_utils.PluginBase

Spec plugin abstract base class.

Spec plugins are stateless classes such that all functions get are their needed parameters passed on every invocation.

The data and arguments (e.g. url for the 'rules' spec plugin) are managed by the sources.database.RulesSource class in the sources.database.RulesDatabase.

is_data_outdated(*data*, *arguments*, *data_load_time*)

keys(*data*, *installer_context*)
Return list of keys defined for current os/version.

load_data(*arguments*)

lookup(*data*, *xylem_key*, *installer_context*)

name

unique_id(*arguments*)

verify_arguments(*arguments*)

verify_data(*data*, *arguments*)

version

4.1.2 Submodules

4.1.3 xylem.exception module

Exception classes for error handling xylem.

exception xylem.exception.**DownloadFailure**
Bases: builtins.Exception

Failure downloading data for I/O or other reasons.

exception xylem.exception.**InstallerNotAvailable**
Bases: builtins.Exception

Failure indicating a installer is not installed.

exception xylem.exception.**InvalidDataError**
Bases: builtins.Exception

Data is not in valid xylem format.

exception xylem.exception.**InvalidPluginError**
Bases: builtins.Exception

Plugin loaded from an entry point does not have the right type/data.

4.1.4 xylem.load_url module

Helper to download content from url.

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

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

Retunrs loaded data as string

Return type str

Raises DownloadFailure if loading fails even after retries

4.1.5 xylem.log_utils module

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

Works like `print()`, optionally uses terminal colors and tries to handle unicode correctly by encoding to `utf-8` before printing. 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.**enable_verbose**(*state=True*)
En- or disable printing verbose output to console.

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

Works like `print()`, optionally uses terminal colors and tries to handle unicode correctly by encoding to `utf-8` before printing.

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

Works like `print()`, optionally uses terminal colors and tries to handle unicode correctly by encoding to `utf-8` before printing.

xylem.log_utils.**is_debug**()
Return true if xylem is set to debug console output.

xylem.log_utils.**is_verbose**()
Return true if xylem is set to verbose console output.

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

Works like `print()`, optionally uses terminal colors and tries to handle unicode correctly by encoding to `utf-8` before printing. Can be enabled or disabled with `enable_debug()`.

4.1.6 xylem.lookup module

4.1.7 xylem.plugin_utils module

Helpers for loading plugin definitions.

class xylem.plugin_utils.**PluginBase**
Bases: `builtins.object`

Abstract base class for all plugin classes.

Plugin classes must define the `name` property. This name is used in other parts of the system. For example for installer plugins the installer name `"apt"` is used in rules definitions. The name of plugin classes is distinct from the plugin name from the plugin definition. The latter is only used to refer to the plugin definitions themselves and (de-)activate specific plugins. All loaded plugins of one kind are unique by the plugin class name.

name
Name of the plugin object.

xylem.plugin_utils.**get_plugin_list**(*kind*, *base_class*, *group*)
Load plugins form entry points.

Load the plugins of given `kind` from entry points `group`, instantiating objects and ignoring duplicates. The entry points must be valid plugin definitions (see `verify_plugin_definition()`). The list of plugins is free of duplicates by plugin class name (not plugin name).

Parameters

- **kind** (*str*) – kind of plugin (e.g. "installer")
- **base_class** – (abstract) base class plugins (must implement PluginBase)
- **group** – entry point group to load plugins from

Returns list of the loaded and instantiated plugin classes

Return type `list`

xylem.plugin_utils.**verify_plugin_class**(*class_*, *base_class*)
Verify class from plugin definition.

Raises ValueError if class is invalid

xylem.plugin_utils.**verify_plugin_definition**(*definition*, *kind*, *base_class*)
Verify plugin definition.

Parameters

- **definition** (*dict*) – definition of plugin as loaded from entry point
- **kind** (*str*) – kind of plugin (e.g. "installer")

- **base_class** (*type*) – (abstract) base class plugins must derive from

Raises InvalidPluginError if plugin definition is invalid

xylem.plugin_utils.**verify_plugin_description**(*decription*)
Verify decription from plugin definition.

Raises ValueError if decription is invalid

xylem.plugin_utils.**verify_plugin_name**(*name*)
Verify name from plugin definition.

Raises ValueError if name is invalid

4.1.8 xylem.resolve module

xylem.resolve.**resolve**(*xylem_keys*, *prefix=None*, *os_override=None*, *all_keys=False*)

4.1.9 xylem.terminal_color module

Module to enable color terminal output.

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

delimiter = '@'

pattern = re.compile('\n \\@(?:\n (?P<escaped>\\@) | # Escape sequence of two delimiters\n (?P<named>[_a-z][_a-z0-9]

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.

4.1.10 xylem.text_utils module

Utility module for dealing with unicode/str/bytes in a uniform way.

This has been inspired by parts of the kitchen package, which is not py3 compatible to date.

xylem.text_utils.**to_bytes**(*obj*, *encoding='utf-8'*, *errors='replace'*)
Helper for converting to encoded byte-strings in py2 and py3.

xylem.text_utils.**to_str**(*obj*, *encoding='utf-8'*, *errors='replace'*)
Helper for converting to (unicode) text in py2 and py3.

4.1.11 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.**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

4.1.12 xylem.util module

Provides common utility functions for xylem.

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

class xylem.util.**change_directory**(*directory=''*)
Bases: builtins.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.**dump_yaml**(*data*, *inline=False*)
Dump data to unicode string.

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

xylem.util.**load_yaml**(*data*)
Parse a unicode string containing yaml.

This calls `yaml.load(data)` but makes sure unicode is handled correctly.

See `yaml.load()`.

Raises yaml.YAMLError if parsing fails

xylem.util.**pdb_hook**()

xylem.util.**print_exc**(*formated_exc*)

xylem.util.**raise_from**(*exc_type*, *exc_args*, *from_exc*)
Raise new exception directly caused by `from_exc`.

On py3, this is equivalent to `raise exc_type(exc_args) from from_exc` and on py2 the messages are composed manually to retain the arguments of `from_exc` as well as the stack trace.

xylem.util.**read_stdout**(*cmd*)

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

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

4.1.13 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
```

CHAPTER 9

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
```

X