Hawkey Documentation

Release 0.8.1-1

Aleš Kozumplík

Mar 30, 2017

Contents

Contents:

CHAPTER 1

API Changes

Contents

Introduction

This document describes the API changes the library users should be aware of before upgrading to each respective version. It is our plan to have the amount of changes requiring changing the client code go to a minimum after the library hits the 1.0.0 version.

Depracated API items (classes, methods, etc.) are designated as such in this document. The first release where support for such items can be dropped entirely must be issued at *least five months* after the issue of the release that announced the deprecation and at the same time have, relatively to the deprecating release, either:

- a higher major version number, or
- a higher minor version number, or
- a patchlevel number that is *by at least five* greater.

These criteria are likely to tighten in the future as hawkey matures.

Actual changes in the API are then announced in this document as well. ABI changes including changes in functions' parameter counts or types or removal of public symbols from `libhawkey` imply an increase in the library's SONAME version.

Changes in 0.2.10

Python bindings

`Query.filter()` now returns a new instance of `Query`, the same as the original with the new filtering applied. This allows for greater flexibility handling the `Query` objects and resembles the way `QuerySets` behave in Django.

In practice the following code will stop working as expected:

```
q = hawkey.Query(self.sack)
q.filter(name__eq="flying")
# processing the query ...
```

It needs to be changed to:

```
q = hawkey.Query(self.sack)
q = q.filter(name__eq="flying")
# processing the query ...
```

The original semantics is now available via the `Query.filterm()` method, so the following will also work:

```
q = hawkey.Query(self.sack)
q.filterm(name__eq="flying")
# processing the query ...
```

Changes in 0.2.11

Python bindings

In Python's `Package` instances accessors for string attributes now return None instead of the empty string if the attribute is missing (for instance a `pkg.sourcerpm` now returns None if `pkg` is a source rpm package already).

This change is towards a more conventional Python practice. Also, this leaves the empty string return value free to be used when it is actually the case.

Changes in 0.3.0

Core

Query: key for reponame filtering

The Query key value used for filtering by the repo name is `HY_PKG_REPONAME` now (was `HY_PKG_REPO`). The old value was misleading.

Repo initialization

`hy_repo_create()` for Repo object initialization now needs to be passed a name of the repository.

Query installs obsoleted

All Goal methods accepting Query as the means of selecting packages, such as `hy_goal_install_query()` have been replaced with their Selector counterparts. Selector structures have been introduced for the particular purpose of specifying a package that best matches the given criteria and at the same time is suitable for installation. For a discussion of this decision see *Selectors are not Queries*.

Python bindings

Query: filtering by repository with the reponame key

Similar change happened in Python, the following constructs:

```
q = q.filter(repo="updates")
```

need to be changed to:

```
q = q.filter(reponame="updates")
```

The old version of this didn't allow using the same string to both construct the query and dynamically get the reponame attribute from the returned packages (used e.g. in DNF to search by user-specified criteria).

Package: removed methods for direct EVR comparison

The following will no longer work:

```
if pkg.evr_eq(some_other_pkg):
    ...
```

Instead use the result of `pkg.evr_cmp`, for instance:

```
if pkg.evr_cmp(some_other_pkg) == 0:
    ...
```

This function compares only the EVR part of a package, not the name. Since it rarely make sense to compare versions of packages of different names, the following is suggested:

```
if pkg == some_other_pkg:
    ...
```

Repo initialization

All instantiations of *hawkey.Repo* now must be given the name of the Repo. The following will now fail:

```
r = hawkey.Repo()
r.name = "fedora"
```

Use this instead:

```
r = hawkey.Repo("fedora")
```

Query installs obsoleted

See *Query installs obsoleted* in the C section. In Python Queries will no longer work as goal target specifiers, the following will fail:

```
q = hawkey.Query(sack)
q.filter(name="gimp")
goal.install(query=q)
```

Instead use:

```
sltr = hawkey.Selector(sack)
sltr.set(name="gimp")
goal.install(select=sltr)
```

Or a convenience notation:

```
goal.install(name="gimp")
```

Changes in 0.3.1

Query: `hy_query_filter_package_in()` takes a new parameter

`keyname` parameter was added to the function signature. The new parameter allows filtering by a specific relation to the resulting packages, for instance:

```
hy_query_filter_package_in(q, HY_PKG_OBSOLETES, HY_EQ, pset)
```

only leaves the packages obsoleting a package in `pset` a part of the result.

Removed `hy_query_filter_obsoleting()`

The new version of `hy_query_filter_package_in()` handles this now, see above.

In Python, the following is no longer supported:

```
q = query.filter(obsoleting=1)
```

The equivalent new syntax is:

```
installed = hawkey.Query(sack).filter(reponame=SYSTEM_REPO_NAME)
q = query.filter(obsoletes=installed)
```

Changes in 0.3.2

Removed `hy_packagelist_of_obsoletes`.

The function was not systematic. Same result is achieved by obtaining obsoleting reldeps from a package and then trying to find the installed packages that provide it. In Python:

```
q = hawkey.Query(sack).filter(reponame=SYSTEM_REPO_NAME, provides=pkg.obsoletes)
```

Changes in 0.3.3

Renamed `hy_package_get_nvra` to `hy_package_get_nevra`

The old name was by error, the functionality has not changed: this function has always returned the full NEVRA, skipping the epoch part when it's 0.

Changes in 0.3.4

Python bindings

`pkg.__repr__()` is more verbose now

Previously, `repr(pkg)` would yield for instance `<_hawkey.Package object, id: 5>`. Now more complete information is present, including the package's NEVRA and repository: `<hawkey.Package object id 5, foo-2-9\.noarch, @System>`.

Also notice that the representation now mentions the final `hawkey.Package` type, not `_hawkey.Package`. Note that these are currently the same.

Changes in 0.3.8

Core

New parameter `rootdir` to `hy_sack_create()`

`hy_sack_create()` now accepts third argument, `rootdir`. This can be used to tell Hawkey that we are intending to do transactions in a changeroot, not in the current root. It effectively makes use of the RPM database found under `rootdir`. To make your code compile in 0.3.8 without changing functionality, change:

```
HySack sack = hy_sack_create(cachedir, arch);
```

to:

```
HySack sack = hy_sack_create(cachedir, arch, NULL);
```

Python bindings

Forms recognized by `Subject` are no longer an instance-scope setting

It became necessary to differentiate between the default forms used by `subject.nevra_possibilities()` and `subject.nevra_possibilities_real()`. Therefore there is little sense in setting the default form for an entire `Subject` instance. The following code:

```
subj = hawkey.Subject("input", form=hawkey.FORM_NEVRA)
result = list(subj.nevra_possibilities())
```

is thus replaced by:

```
subj = hawkey.Subject("input")
result = list(subj.nevra_possibilities(form=hawkey.FORM_NEVRA))
```

Changes in 0.3.9

Core

Flags for `hy_sack_create`

`hy_sack_create()` now accepts fourth argument, `flags`, introduced to modify the sack behavior with boolean flags. Currently only one flag is supported, `HY_MAKE_CACHE_DIR`, which causes the cache directory to be created if it doesn't exist yet. To preserve the previous behavior, change the following:

```
HySack sack = hy_sack_create(cachedir, arch, rootdir);
```

into:

```
HySack sack = hy_sack_create(cachedir, arch, rootdir, HY_MAKE_CACHE_DIR);
```

`hy_sack_get_cache_path` is renamed to `hy_sack_get_cache_dir`

Update your code by mechanically replacing the name.

Python bindings

`make_cache_dir` for Sack's constructor

A new sack by default no longer automatically creates the cache directory. To get the old behavior, append `make_cache_dir=True` to the *Sack.__init__()* arguments, that is change the following:

```
sack = hawkey.Sack(...)
```

to:

```
sack = hawkey.Sack(..., make_cache_dir=True)
```

`cache_path` property of `Sack` renamed to `cache_dir`

Reflects the similar change in C API.

Changes in 0.3.11

Core

`hy_goal_package_obsoletes()` removed, `hy_goal_list_obsoleted_by_package()` provided instead

`hy_goal_package_obsoletes()` was flawed in that it only returned a single obsoleted package (in general, package can obsolete arbitrary number of packages and upgrade a package of the same name which is also reported as an obsolete). Use `hy_goal_list_obsoleted_by_package()` instead, to see the complete set of packages that inclusion of the given package in an RPM transaction will cause to be removed.

`hy_goal_list_erasures()` does not report obsoletes

In other words, `hy_goal_list_erasures()` and `hy_goal_list_obsoleted()` return disjoint sets.

Python bindings

Directly reflecting the *core changes*. In particular, instead of:

```
obsoleted_pkg = goal.package_obsoletes(pkg)
```

use:

```
obsoleted = goal.obsoleted_by_package(pkg) # list
obsoleted_pkg = obsoleted[0]
```

Changes in 0.4.5

Core

Query: `hy_query_filter_latest()` now filter latest packages ignoring architecture

For old function behavior use new function `hy_query_filter_latest_per_arch()`

Python bindings

In Python's `Query` option `latest` in `Query.filter()` now filter only the latest packages ignoring architecture. The original semantics for filtering latest packages for each arch is now available via `latest_per_arch` option.

For example there are these packages in sack:

```
glibc-2.17-4.fc19.x86_64
glibc-2.16-24.fc18.x86_64
glibc-2.16-24.fc18.i686

>>> q = hawkey.Query(self.sack).filter(name="glibc")
>>> map(str, q.filter(latest=True))
['glibc-2.17-4.fc19.x86_64']

>>> map(str, q.filter(latest_per_arch=True))
['glibc-2.17-4.fc19.x86_64', 'glibc-2.16-24.fc18.i686']
```

Changes in 0.4.13

Core

Deprecated `hy_package_get_update_*`

The functions were deprecated because there can be multiple advisories referring to a single package. Please use the new function `hy_package_get_advisories()` which returns all these advisories. New functions `hy_advisory_get_*` provide the data retrieved by the deprecated functions.

The only exception is the `hy_package_get_update_severity()` which will be dropped without any replacement. However advisory types and severity levels are distinguished from now and the type is accessible via `hy_advisory_get_type()`. Thus enum `HyUpdateSeverity` was also deprecated. A new `HyAdvisoryType` should be used instead.

The old functions will be dropped after 2014-07-07.

Changes in 0.4.15

Core

`hy_goal_write_debugdata()` takes a directory parameter

`hy_goal_write_debugdata()` has a new *const char *dir* argument to communicate the target directory for the debugging data. The old call:

```
hy_goal_write_debugdata(goal);
```

should be changed to achieve the same behavior to:

```
hy_goal_write_debugdata(goal, "./debugdata");
```

Python bindings

`Goal.write_debugdata()` takes a directory parameter

Analogous to *core changes*.

Package: string attributes are represented by Unicode object

Attributes `baseurl`, `location`, `sourcerpm`, `version`, `release`, `name`, `arch`, `description`, `evr`, `license`, `packager`, `reponame`, `summary` and `url` of Package object return Unicode string.

Changes in 0.4.18

Core

Deprecated `hy_advisory_get_filenames`

The function was deprecated because we need more information about packages listed in an advisory than just file names. Please use the new function `hy_advisory_get_packages()` in combination with `hy_advisorypkg_get_string()` to obtain the data originally provided by the deprecated function.

The old function will be dropped after 2014-10-15 AND no sooner than in 0.4.21.

Python bindings

`Repo()` does not accept `cost` keyword argument

Instead of:

```
r = hawkey.Repo('name', cost=30)
```

use:

```
r = hawkey.Repo('name')
r.cost = 30
```

Also previously when no `cost` was given it defaulted to 1000. Now the default is 0. Both these aspects were present by mistake and the new interface is consistent with the C library.

Deprecated `_hawkey.Advisory.filenames`

The attribute was deprecated because the underlying C function was also deprecated. Please use the new attribute `packages` and the attribute `filename` of the returned objects to obtain the data originally provided by the deprecated attribute.

The old attribute will be dropped after 2014-10-15 AND no sooner than in 0.4.21.

Changes in 0.4.19

Python bindings

Advisory attributes in Unicode

All string attributes of `Advisory` and `AdvisoryRef` objects (except the deprecated `filenames` attribute) are Unicode objects now.

Changes in 0.5.2

Core

`hy_chksum_str` returns NULL

Previously, the function `hy_chksum_str` would cause a segmentation fault when it was used with incorrect type value. Now it correctly returns NULL if type parameter does not correspond to any of expected values.

Changes in 0.5.3

Core

New parameter `logfile` to `hy_sack_create()`

`hy_sack_create()` now accepts fifth argument, `logfile` to customize log file path. If NULL parameter as `logfile` is given, then all debug records are written to `hawkey.log` in `cachedir`. To make your code compile in 0.5.3 without changing functionality, change:

```
HySack sack = hy_sack_create(cachedir, arch, rootdir, 0);
```

to:

```
HySack sack = hy_sack_create(cachedir, arch, rootdir, NULL, 0);
```

Deprecated `hy_create_cmdline_repo()`

The function will be removed since *hy_add_cmdline_package* creates cmdline repository automatically.

The function will be dropped after 2015-06-23 AND no sooner than in 0.5.8.

Python bindings

New optional parameter `logfile` to `Sack` constructor

This addition lets user specify log file path from *Sack.__init__()*

`cache_path` property of `Sack` renamed to `cache_dir`

This change was already announced but it actually never happened.

Deprecated `Sack` method `create_cmdline_repo()`

The method will be removed since *Sack.add_cmdline_package()* creates cmdline repository automatically.

The method will be dropped after 2015-06-23 AND no sooner than in 0.5.8.

Changes in 0.5.4

Python bindings

Goal: `install()` takes a new optional parameter

If the `optional` parameter is set to `True`, hawkey silently skips packages that can not be installed.

Changes in 0.5.5

Core

Renamed `hy_sack_load_yum_repo` to `hy_sack_load_repo`

Hawkey is package manager agnostic and the `yum` phrase could be misleading.

The function will be dropped after 2015-10-27 AND no sooner than in 0.5.8.

Python bindings

Sack method *load_yum_repo* has been renamed to `Sack.load_repo()`

Hawkey is package manager agnostic and the `yum` phrase could be misleading.

The method will be dropped after 2015-10-27 AND no sooner than in 0.5.8.

Changes in 0.5.7

Python bindings

Package: file attribute is represented by list of Unicode objects

Sack: *list_arches* method returns list of Unicode objects

Changes in 0.5.9

Core

Deprecated `hy_goal_req_has_distupgrade()`, `hy_goal_req_has_erase()` and `hy_goal_req_has_upgrade()` functions

To make your code compile in 0.5.9 without changing functionality, change:

```
hy_goal_req_has_distupgrade_all(goal)
hy_goal_req_has_erase(goal)
hy_goal_req_has_upgrade_all(goal)
```

to:

```
hy_goal_has_actions(goal, HY_DISTUPGRADE_ALL)
hy_goal_has_actions(goal, HY_ERASE)
hy_goal_has_actions(goal, HY_UPGRADE_ALL)
```

respectively

Python bindings

Deprecated Goal methods `Goal.req_has_distupgrade_all()`, `Goal.req_has_erase()` and `Goal.req_has_upgrade_all()`

To make your code compatible with hawkey 0.5.9 without changing functionality, change:

```
goal.req_has_distupgrade_all()
goal.req_has_erase()
goal.req_has_upgrade_all()
```

to:

```
goal.actions | hawkey.DISTUPGRADE_ALL
goal.actions | hawkey.ERASE
goal.actions | hawkey.UPGRADE_ALL
```

respectively

Changes in 0.6.2

Core

The `hy_advisory_get_filenames()` API call, the corresponding Python property `filenames` of class `Advisory` are removed. Instead, iterate over `hy_advisory_get_packages()` with `hy_advisorypkg_get_string()` and `HY_ADVISORYPKG_FILENAME`. No known hawkey API consumers were using this call.

Hawkey now has a dependency on GLib. Aside from the above `hy_advisory_get_filenames()` call, the Python API is fully preserved. The C API has minor changes, but the goal is to avoid causing a significant amount of porting work for existing consumers.

The `hy_package_get_files` API call now returns a `char **`, allocated via *g_malloc*. Free with *g_strfreev*.

The `HyStringArray` type is removed, as nothing now uses it.

`HyPackageList` is now just a `GPtrArray`, though the existing API is converted into wrappers. Notably, this means you can now use `g_ptr_array_unref()`.

Python bindings

Aside from the one change below, the Python bindings should be unaffected by the C API changes.

Advisory: The `filename` property is removed along with the C API

CHAPTER 2

FAQ

Contents

Getting Started

How do I build it?

See the README.

Are there examples using hawkey?

Yes, look at:

- unit tests
- The Hawkey Testing Hack
- a more complex example is DNF, the Yum fork using hawkey for backend.

Using Hawkey

How do I obtain the repo metadata files to feed to Hawkey?

It is entirely up to you. Hawkey does not provide any means to do this automatically, for instance from your */etc/yum.repos.d* configuration. Use or build tools to do that. For instance, both Yum and DNF deals with the same problem and inside they employ urlgrabber to fetch the files. A general solution if you work in C is for instance libcurl. If you are building a nice downloading library that integrates well with hawkey, let us know.

Why is a tool to do the downloads not integrated into Hawkey?

Because downloading things from remote servers is a differnt domain full of its own complexities like HTTPS, parallel downloads, error handling and error recovery to name a few. Downloading is a concern that can be naturally separated from other parts of package metadata managing.

CHAPTER 3

python-hawkey Tutorial

Contents

Important: Please consult every usage of the library with *python-hawkey Reference Manual* to be sure what are you doing. The examples mentioned here are supposed to be as simple as possible and may ignore some minor corner cases.

Setup

First of, make sure hawkey is installed on your system, this should work from your terminal:

```
>>> import hawkey
```

The Sack Object

Sack is an abstraction for a collection of packages. Sacks in hawkey are toplevel objects carrying much of hawkey's of functionality. You'll want to create one:

```
>>> sack = hawkey.Sack()
>>> len(sack)
0
```

Initially, the sack contains no packages.

Loading RPMDB

hawkey is a lib for listing, querying and resolving dependencies of *packages* from *repositories*. On most linux distributions you always have at least *the system repo* (in Fedora it is the RPM database). To load it:

```
>>> sack.load_system_repo()
>>> len(sack)
1683
```

Hawkey always knows the name of every repository. The system repository is always set to *hawkey.SYSTEM_REPO_NAME*. and the client is responsible for naming the available repository metadata.

Loading Repositories

Let's be honest here: all the fun in packaging comes from packages you haven't installed yet. Information about them, their *metadata*, can be obtained from different sources and typically they are downloaded from an HTTP mirror (another possibilities are FTP server, NFS mount, DVD distribution media, etc.). Hawkey does not provide any means to discover and obtain the metadata locally: it is up to the client to provide valid readable paths to the repository metadata XML files. Structures used for passing the information to hawkey are the hawkey *Repos*. Suppose we somehow obtained the metadata and placed it in `/home/akozumpl/tmp/repodata`. We can then load the metadata into hawkey:

```
>>> path = "/home/akozumpl/tmp/repodata/%s"
>>> repo = hawkey.Repo("experimental")
>>> repo.repomd_fn = path % "repomd.xml"
>>> repo.primary_fn = path %
↪"f7753a2636cc89d70e8aaa1f3c08413ab78462ca9f48fd55daf6dedf9ab0d5db-primary.xml.gz"
>>> repo.filelists_fn = path %
↪"0261e25e8411f4f5e930a70fa249b8afd5e86bb9087d7739b55be64b76d8a7f6-filelists.xml.gz"
>>> sack.load_repo(repo, load_filelists=True)
>>> len(sack)
1685
```

The number of packages in the Sack will increase by the number of packages found in the repository (two in this case, it is an experimental repo after all).

Case for Loading the Filelists

What the `load_filelists=True` argument to *load_repo()* above does is instruct hawkey to process the `<hash>filelists.xml.gz` file we passed in and which contains structured list of absolute paths to all files of all

packages within the repo. This information can be used for two purposes:

- Finding a package providing given file. For instance, you need the file `/usr/share/man/man3/fprintf.3.gz` which is not installed. Consulting filelists (directly or through hawkey) can reveal the file is in the `man-pages` package.
- Depsolving. Some packages require concrete files as their dependencies. To know if these are resolvable and how, the solver needs to know what package provides what files.

Some files provided by a package (e.g those in `/usr/bin`) are always visible even without loading the filelists. Well-behaved packages requiring only those can be thus resolved directly. Unortunately, there are packages that don't behave and it is hard to tell in advance when you'll deal with one.

The strategy for using `load_filelists=True` is thus:

- Use it if you know you'll do resolving (i.e. you'll use `Goal`).
- Use it if you know you'll be trying to match files to their packages.
- Use it if you are not sure.

Building and Reusing the Repo Cache

Internally to hold the package information and perform canonical resolving hawkey uses Libsolv. One great benefit this library offers is providing writing and reading of metadata cache files in libsolv's own binary format (files with `.solv` extension, typically). At a cost of few hundreds of milliseconds, using the solv files reduces repo load times from seconds to tens of milliseconds. It is thus a good idea to write and use the solv files every time you plan to use the same repo for more than one Sack (which is at least every time your hawkey program is run). To do that use `build_cache=True` with *`load_repo()`* and *`load_system_repo()`*:

```
>>> sack = hawkey.Sack(make_cache_dir=True)
>>> sack.load_system_repo(build_cache=True)
```

By default, Hawkey creates `@System.cache` under the `/var/tmp/hawkey-<your_login>-<random_hash>` directory. This is the hawkey cache directory, which you can always delete later (deleting the cache files in the process). The `.solv` files are picked up automatically the next time you try to create a hawkey sack. Except for a much higher speed of the operation this will be completely transparent to you:

```
>>> s2 = hawkey.Sack()
>>> s2.load_system_repo()
```

By the way, the cache directory (if not set otherwise) also contains a logfile with some boring debugging information.

Queries

Query is the means in hawkey of finding a package based on one or more criteria (name, version, repository of origin). Its interface is loosely based on Django's QuerySets, the main concepts being:

- a fresh Query object matches all packages in the Sack and the selection is gradually narrowed down by calls to `Query.filter()`
- applying a `Query.filter()` does not start to evaluate the Query, i.e. the Query is lazy. Query is only evaluated when we explicitly tell it to or when we start to iterate it.
- use Python keyword arguments to `Query.filter()` to specify the filtering criteria.

For instance, let's say I want to find all installed packages which name ends with `gtk`:

```
>>> q = hawkey.Query(sack).filter(reponame=hawkey.SYSTEM_REPO_NAME, name__glob='*gtk')
>>> for pkg in q:
...     print str(pkg)
...
NetworkManager-gtk-1:0.9.4.0-9.git20120521.fc17.x86_64
authconfig-gtk-6.2.1-1.fc17.x86_64
clutter-gtk-1.2.0-1.fc17.x86_64
libchamplain-gtk-0.12.2-1.fc17.x86_64
libreport-gtk-2.0.10-3.fc17.x86_64
pinentry-gtk-0.8.1-6.fc17.x86_64
python-slip-gtk-0.2.20-2.fc17.noarch
transmission-gtk-2.50-2.fc17.x86_64
usermode-gtk-1.109-1.fc17.x86_64
webkitgtk-1.8.1-2.fc17.x86_64
xdg-user-dirs-gtk-0.9-1.fc17.x86_64
```

Or I want to find the latest version of all `python` packages the Sack knows of:

```
>>> q.clear()
>>> q = q.filter(name='python', latest_per_arch=True)
>>> for pkg in q:
...     print str(pkg)
...
python-2.7.3-6.fc17.x86_64
```

You can also test a `Query` for its truth value. It will be true whenever the query matched at least one package:

```
>>> q = hawkey.Query(sack).filter(file='/boot/vmlinuz-3.3.4-5.fc17.x86_64')
>>> if q:
...     print 'match'
...
match
>>> q = hawkey.Query(sack).filter(file='/booty/vmlinuz-3.3.4-5.fc17.x86_64')
>>> if q:
...     print 'match'
...
>>> if not q:
...     print 'no match'
...
no match
```

Note: If the Query hasn't been evaluated already then it is evaluated whenever it's length is taken (either via `len(q)` or `q.count()`), when it is tested for truth and when it is explicitly evaluated with `q.run()`.

Resolving things with Goals

Many *`Sack`* sessions culminate in a bout of dependency resolving, that is answering a question along the lines of "I have a package X in a repository here, what other packages do I need to install/update to have X installed and all its dependencies recursively satisfied?" Suppose we want to install the RTS game Spring. First let's locate the latest version of the package in repositories:

```
>>> q = hawkey.Query(sack).filter(name='spring', latest_per_arch=True)
>>> pkg = hawkey.Query(sack).filter(name='spring', latest_per_arch=True)[0]
>>> str(pkg)
'spring-88.0-2.fc17.x86_64'
>>> pkg.reponame
'fedora'
```

Then build the `Goal` object and tell it our goal is installing the `pkg`. Then we fire off the libsolv's dependency resolver by running the goal:

```
>>> g = hawkey.Goal(sack)
>>> g.install(pkg)
>>> g.run()
True
```

`True` as a return value here indicates that libsolv could find a solution to our goal. This is not always the case, there are plenty of situations when there is no solution, the most common one being a package should be installed but one of its dependencies is missing from the sack.

The three methods `Goal.list_installs()`, `Goal.list_upgrades()` and `Goal.list_erasures()` can show which packages should be installed/upgraded/erased to satisfy the packaging goal we set out to achieve (the mapping of `str()` over the results below ensures human readable package names instead of numbers are presented):

```
>>> map(str, g.list_installs())
['spring-88.0-2.fc17.x86_64', 'spring-installer-20090316-10.fc17.x86_64',
↪'springlobby-0.139-3.fc17.x86_64', 'spring-maps-default-0.1-8.fc17.noarch', 'wxBase-
↪2.8.12-4.fc17.x86_64', 'wxGTK-2.8.12-4.fc17.x86_64', 'rb_libtorrent-0.15.9-1.fc17.
↪x86_64', 'GeoIP-1.4.8-2.1.fc17.x86_64']
>>> map(str, g.list_upgrades())
[]
>>> map(str, g.list_erasures())
[]
```

So what does it tell us? That given the state of the given system and the given repository we used, 8 packages need to be installed, `spring-88.0-2.fc17.x86_64` itself included. No packages need to be upgraded or erased.

Selector Installs

For certain simple and commonly used queries we can do installs directly. Instead of executing a query however we instantiate and pass the `Goal.install()` method a `Selector`:

```
>>> g = hawkey.Goal(sack)
>>> sltr = hawkey.Selector(sack).set(name='emacs-nox')
>>> g.install(select=sltr)
>>> g.run()
True
>>> map(str, g.list_installs())
['spring-88.0-2.fc17.x86_64', 'spring-installer-20090316-10.fc17.x86_64',
↪'springlobby-0.139-3.fc17.x86_64', 'spring-maps-default-0.1-8.fc17.noarch', 'wxBase-
↪2.8.12-4.fc17.x86_64', 'wxGTK-2.8.12-4.fc17.x86_64', 'rb_libtorrent-0.15.9-1.fc17.
↪x86_64', 'GeoIP-1.4.8-2.1.fc17.x86_64']
>>> len(g.list_upgrades())
0
>>> len(g.list_erasures())
0
```

Notice we arrived at the same result as before, when a query was constructed and iterated first. What `Selector` does when passed to `Goal.install()` is tell hawkey to examine its settings and without evaluating it as a `Query` it instructs libsolv to find *the best matching package* for it and add that for installation. It saves user some decisions like which version should be installed or what architecture (this gets very relevant with multiarch libraries).

So Selectors usually only install a single package. If you mean to install *all packages* matching an arbitrarily complex query, just use the method describe above:

```
>>> map(goal.install, q)
```

CHAPTER 4

python-hawkey Reference Manual

Contents

Introduction

This reference manual describes Python API to the library. For a quick start take a look at *python-hawkey Tutorial*. To be sure that you are familiar with our deprecation policy, see *API Changes*.

Note: The API consists of exactly those elements described in this document, items not documented here can change from release to release. Opening a bugzilla if certain needed functionality is not exposed is the right thing to do.

Warning: The manual is not complete yet - the features are being added incrementally these days.

Contents

API Documentation Contents

Sack—The fundamental hawkey structure

class hawkey.**Sack**

Instances of *hawkey.Sack* represent collections of packages. An application typically needs at least one instance because it provides much of the hawkey's functionality.

> **Warning:** Any package instance is not supposed to work interchangeably between hawkey.Query, hawkey.Selector or hawkey.Goal created from different *hawkey.Sack*. Usually for common tasks there is no need to initialize two or more *Sacks* in your program. Sacks cannot be deeply copied.

cache_dir

A read-only string property giving the path to the location where a metadata cache is stored.

installonly

A write-only sequence of strings property setting the provide names of packages that should only ever be installed, never upgraded.

installonly_limit

A write-only integer property setting how many installonly packages with the same name are allowed to be installed concurrently. If 0, any number of packages can be installed.

__init__(*cachedir=_CACHEDIR*, *arch=_ARCH*, *rootdir=_ROOTDIR*, *pkgcls=hawkey.Package*, *pkginitval=None*, *make_cache_dir=False*, *logfile=_LOGFILE*)

Initialize the sack with a default cache directory, log file location set to hawkey.log in the cache directory, an automatically detected architecture and the current root (/) as an installroot. The cache is disabled by default.

cachedir is a string giving a path of a cache location.

arch is a string specifying an architecture.

rootdir is a string giving a path to an installroot.

pkgcls is a class of packages retrieved from the sack. The class' __init__ method must accept two arguments. The first argument is a tuple of the sack and the ID of the package. The second argument is the *pkginitval* argument. *pkginitval* cannot be None if *pkgcls* is specified.

make_cache_dir is a boolean that specifies whether the cache should be used to speedup loading of repositories or not (see *Building and Reusing the Repo Cache*).

logfile is a string giving a path of a log file location.

__len__()

Returns the number of the packages loaded into the sack.

add_cmdline_package(*filename*)

Add a package to a command line repository and return it. The package is specified as a string *filename* of an RPM file. The command line repository will be automatically created if doesn't exist already. It could be referenced later by *hawkey.CMDLINE_REPO_NAME* name.

add_excludes(*packages*)

Add a sequence of packages that cannot be fetched by Queries nor Selectors.

add_includes(*packages*)

Add a sequence of the only packages that can be fetched by Queries or Selectors.

This is the inverse operation of *add_excludes()*. Any package that is not in the union of all the included packages is excluded. This works in conjunction with exclude and doesn't override it. So, if you both include and exclude the same package, the package is considered excluded no matter of the order.

disable_repo(*name*)
Disable the repository identified by a string *name*. Packages in that repository cannot be fetched by Queries nor Selectors.

enable_repo(*name*)
Enable the repository identified by a string *name*. Packages in that repository can be fetched by Queries or Selectors.

> **Warning:** Execution of *add_excludes()*, *add_includes()*, *disable_repo()* or *enable_repo()* methods could cause inconsistent results in previously evaluated `Query`, `Selector` or `Goal`. The rule of thumb is to exclude/include packages, enable/disable repositories at first and then do actual computing using `Query`, `Selector` or `Goal`. For more details see developer discussion.

evr_cmp(*evr1*, *evr2*)
Compare two EVR strings and return a negative integer if *evr1* < *evr2*, zero if *evr1* == *evr2* or a positive integer if *evr1* > *evr2*.

get_running_kernel()
Detect and return the package of the currently running kernel. If the package cannot be found, `None` is returned.

list_arches()
List strings giving all the supported architectures.

load_system_repo(*repo=None*, *build_cache=False*)
Load the information about the packages in the system repository (in Fedora it is the RPM database) into the sack. This makes the dependency solving aware of the already installed packages. The system repository is always set to *hawkey.SYSTEM_REPO_NAME*. The information is not written to the cache by default.

repo is an optional *Repo* object that represents the system repository. The object is updated during the loading.

build_cache is a boolean that specifies whether the information should be written to the cache (see *Building and Reusing the Repo Cache*).

load_repo(*repo*, *build_cache=False*, *load_filelists=False*, *load_presto=False*, *load_updateinfo=False*)
Load the information about the packages in a *Repo* into the sack. This makes the dependency solving aware of these packages. The information is not written to the cache by default.

repo is the *Repo* object to be processed. At least its *Repo.repomd_fn* must be set. If the cache has to be updated, *Repo.primary_fn* is needed too. Some information about the loading process and some results of it are written into the internal state of the repository object.

build_cache is a boolean that specifies whether the information should be written to the cache (see *Building and Reusing the Repo Cache*).

load_filelists, *load_presto* and *load_updateinfo* are booleans that specify whether the *Repo.filelists_fn*, *Repo.presto_fn* and *Repo.updateinfo_fn* files of the repository should be processed. These files may contain information needed for dependency solving, downloading or querying of some packages. Enable it if you are not sure (see *Case for Loading the Filelists*).

Error handling

When an error or an unexpected event occurs during a Hawkey routine, an exception is raised:

- if it is a general error that could be common to other Python programs, one of the standard Python built-in exceptions is raised. For instance, `IOError` and `TypeError` can be raised from Hawkey.
- programming errors within Hawkey that cause unexpected or invalid states raise the standard `AssertionError`. These should be reported as bugs against Hawkey.
- programming errors due to incorrect use of the library usually produce `hawkey.ValueException` or one of its subclasses, `QueryException` (poorly formed Query) or `ArchException` (unrecognized architecture).
- sometimes there is a close call between blaming the error on an input parameter or on something else, beyond the programmer's control. `hawkey.RuntimeException` is generally used in this case.
- `hawkey.ValidationException` is raised when a function call performs a preliminary check before proceeding with the main operation and this check fails.

The class hierarchy for Hawkey exceptions is:

```
+-- hawkey.Exception
     +-- hawkey.ValueException
     |    +-- hawkey.QueryException
     |    +-- hawkey.ArchException
     +-- hawkey.RuntimeException
     +-- hawkey.ValidationException
```

Module level constants

`hawkey.`**`CMDLINE_REPO_NAME`**
The string name of the command line repository.

`hawkey.`**`SYSTEM_REPO_NAME`**
The string name of the system repository.

Repositories

class `hawkey.`**`Repo`**
Instances of *`hawkey.Repo`* point to metadata of packages that are available to be installed. The metadata are expected to be in the "rpm-md" format. librepo may help you with downloading the required files.

`cost`
An integer specifying a relative cost of accessing this repository. This value is compared when the priorities of two repositories are the same. The repository with *the lowest cost* is picked. It is useful to make the library prefer on-disk repositories to remote ones.

`filelists_fn`
A valid string path to the readable "filelists" XML file if set.

`name`
A string name of the repository.

`presto_fn`
A valid string path to the readable "prestodelta.xml" (also called "deltainfo.xml") file if set.

`primary_fn`
A valid string path to the readable "primary" XML file if set.

`priority`
An integer priority value of this repository. If there is more than one candidate package for a particular operation, the one from the repository with *the lowest priority* value is picked, possibly despite being less convenient otherwise (e.g. by being a lower version).

repomd_fn
A valid string path to the readable "repomd.xml" file if set.

updateinfo_fn
A valid string path to the readable "updateinfo.xml" file if set.

__init__(*name*)
Initialize the repository with empty *repomd_fn*, *primary_fn*, *filelists_fn* and *presto_fn*. The priority and the cost are set to `0`.

name is a string giving the name of the repository. The name should not be equal to *hawkey.SYSTEM_REPO_NAME* nor *hawkey.CMDLINE_REPO_NAME*

Indices:

- genindex

CHAPTER 5

Design Rationale

Selectors are not Queries

Since both a Query and a Selector work to limit the set of all Sack's packages to a subset, it can be suggested the two concepts should be the same and e.g. Queries should be used for Goal specifications instead of Selectors:

```
// create sack, goal, ...
HyQuery q = hy_query_create(sack);
hy_query_filter(q, HY_PKG_NAME, HY_EQ, "anaconda")
hy_goal_install_query(q)
```

This arrangment was in fact used in hawkey prior to version 0.3.0, just because Queries looked like a convenient structure to hold this kind of information. It was unfortunately confusing for the programmers: notice how evaluating the Query `q` would generally produce several packages (`anaconda` for different architectures and then different versions) but somehow when the same Query is passed into the goal methods it always results in up to one pacakge selected for the operation. This is a principal discrepancy. Further, Query is universal and allows one to limit the package set with all sorts of criteria, matched in different ways (substrings, globbing, set operation) while Selectors only support few. Finally, while a fresh Query with no filters applied corresponds to all packages of the Sack, a fresh Selector with no limits set is of no meaning.

An alternative to introducing a completely different concept was adding a separate constructor function for Query, one that would from the start designate the Query to only accept settings compatible with its purpose of becoming the selecting element in a Goal operation (in Python this would probably be implemented as a subclass of Query). But that would break client's assumptions about Query (the unofficial C++ FAQ takes up the topic).

Implementation note: Selectors reflect the kind of specifications that can be directly translated into Libsolv jobs, without actually searching for a concrete package to put there. In other words, Selectors are specifically designed not to iterate over the package data (with exceptions, like glob matching) like Queries do. While Hawkey mostly aims to hide any twists and complexities of the underlying library, in this case the combined reasons warrant a concession.

Indices and tables

- genindex
- modindex

- search

Index

Symbols

A

C

D

E

F

G

H

I

L

N

P

R

U