Package Management in Racket

Version 6.1.1

Jay McCarthy <jay@racket-lang.org>

November 4, 2014

The Racket package manager lets you install new libraries and collections, and the Racket package catalog helps other Racket programmers find libraries that you make available.

Contents

1 Getting Started with Packages

There are two ways to manage Racket package installations:

- The package manager graphical interface.

  Most users access the package manager graphical interface through DrRacket, which provides a Package Manager... item in the File menu.

  You can also install the `gui-pkg-manager` package, with provides a Racket Package Manager application (which can be launched as `racket-package-manager` in a command-line environment).

- The `raco pkg` command-line tool.

  The `raco` executable launches various Racket command-line tools, and the `raco pkg` command groups various package-management sub-commands. The different `raco pkg` sub-commands are documented in §3 "Using `raco pkg`".

We'll use the `raco pkg` command to describe package-management operations here, but the graphical interface allows the same operations.

1.1 What is a Package?

A package is not something that you refer to directly in your Racket programs. Instead, a package is a set of libraries that fit into the collection hierarchy, and you refer to libraries through their collection-based paths. Libraries that are close in the hierarchy may be provided by different packages, while a single package may provide libraries that are far from each other in the hierarchy (but that are conceptually related, somehow).

Racket documentation tells you which package provides a given library. For example, the documentation for the `pict/face` library says that it is provided by the `pict-lib` package.

If you're reading this in a web browser, click `pict/face` to go straight to its documentation.

Over time, packages may be refactored so that a library moves to a different package, but the original package should continue to provide the library, too, by declaring a dependency on the new package. More generally, a package is intended to have an interface that only grows in terms of libraries, bindings, and functionality, which provides a basic level of backward compatibility. Incompatible changes should be implemented in a new package.

1.2 Inspecting Your Installation

To see the packages that you have installed already, use the `raco pkg show` subcommand:

```
raco pkg show
```

Unless you have an especially minimal Racket installation, you will have packages installed already, probably listed in the "Installation-wide" section. In fact, if you have a typical Racket installation, then `raco pkg show` will initially show a `main-distribution` package and a `racket-lib` package:

```
Installation-wide:
 Package              Checksum          Source
 main-distribution    01.........ef     (catalog main-distribution)
 racket-lib           fe.........01     (catalog racket-lib)
User-specific for installation ....:
 [none]
```

The "Checksum" column reports the specific implementation of each package that is installed. A package can have a version in a more traditional sense, but the checksum is the "version" as far as the package system is concerned. When you request an update, then a package installation is updated if the current implementation of the package has a different checksum than the installed package, whether or not the package author adjusted the package's version.

The "Source" column indicates how each package was installed. A `catalog` source indicates that the package was installed by consulting a package catalog. The name after `catalog` indicates the name of the package as requested from the catalog, which is normally (but not necessarily) the name of the package as it exists in your installation. We discuss other possibilities for "Source" in §1.4 "Installing Packages".

Neither the `main-distribution` package nor the `racket-lib` package actually provides any libraries on its own, but each declares dependencies on other packages. The `racket-lib` package depends on native-library packages, if any, for your platform. The `main-distribution` package depends on lots of packages that have been selected for inclusion in the main Racket distribution. If you provide the `--all` flag to `raco pkg show`, then you can see the packages that were automatically installed as a result of installing `main-distribution` and `racket-lib` (or whatever packages you have explicitly selected for your installation).

```
raco pkg show --all
```

An asterisk appears beside the name of every package that was "auto-installed" to satisfy a dependency. All auto-installed packages are as available for your use in the same way as explicitly installed packages, but normally your code should refer only to packages that you have explicitly installed. The difference between an auto-installed and an explicitly installed package is how various commands, such as `raco pkg show`, treat the package. If you specifically request installation of a package that is auto-installed, then the package is promoted and thereafter treated as a explicitly installed package.

1.3 Finding Packages

The PLT package catalog at

```
http://pkgs.racket-lang.org
```

provides a centralized listing of available Racket packages. The PLT package catalog normally will be the first place you check when looking for a package.

There are other ways to distribute and reference packages. For example, a package can be installed directly from a `".zip"` file—available locally or served from on a web site—or from a Github repository. Such direct references make sense when a package is not yet ready for wide distribution or when it will never be of interest to a wide audience. So, you may find non-catalog references in mailing-list posts, recommended by your friends, or advertised in e-mail spam.

There may be other package catalog services besides PLT's. Note that even if you discover a package name from PLT's package catalog, your installation may be configured to consult a different package catalog to locate the package's implementation (to obtain a pre-built version of the package, for example), but you should expect the installation-configured package catalog to deliver the package that is described on the PLT package catalog.

1.4 Installing Packages

If you find a package by name from a package catalog, then use the package's name with `raco pkg install`:

`raco pkg install` ⟨*pkg-name*⟩

If the package depends on other packages that you do not have installed already, then `raco pkg install` will alert you and ask whether it should install them, too. Use `--auto` to skip the question and make dependencies installed automatically. Either way, packages installed to satisfy dependencies are marked as auto-installed, which makes them easier to uninstall, and it also makes them hidden by default for `raco pkg show` (since packages that are installed for dependencies are an implementation detail that you usually do not care about).

The argument that you provide to `raco pkg install` does not have to be a package name that is recognized by a package catalog. In general, each argument to `raco pkg install` is a package source. A package source can refer to a `".zip"` file, a `".tar"` file, a Github repository, a directory-structured web site, or a few other possibilities. In each of those cases, a package name is inferred from the package source. After the package is installed, you use the package name with other `raco pkg` commands to refer to the installed package.

In fact, a package catalog does not actually serve package implementations. It simply maps

each package name to a package source. When the package manager consults a package catalog, it gets back a package source for the actual package implementation, so each package installed from a package catalog is actually installed from a `".zip"` file, Github repository, etc. Registering with a package catalog is just a way of making your package easier to find and update.

1.5 Updating Packages

If your package installations become out of date, you can update packages with `raco pkg update`:

```
raco pkg update ⟨pkg-name⟩
```

Either specify individual packages to update, or use `--all` to update all installed packages for which a new checksum is available.

The way that the package manager finds updates depends on the way that a package was installed. If it was installed by using a package name that was resolved by a package catalog, then the package catalog is consulted again to get the current checksum for the package, and the package is updated if the checksum doesn't match the current installation. If the package was installed directly from a Github reference, then Github is consulted to get the current commit of a particular branch, and the package is updated if the commit identifier doesn't match the checksum of the current installation.

In some cases, updating a package may require an update to one of the package's dependencies. That should happen only when the package requires a new binding, feature, or bug fix from the dependent package, since packages are meant to evolve in an otherwise backward-compatible way. Package versions provide a way for package authors to declare (and for the package manager to check) those dependencies. The end result is that `raco pkg update` might report a version-mismatch error that forces you to request more package updates than you originally requested.

Normally, you provide package names to `raco pkg update`. More generally, you can provide a package source to `raco pkg update`. In that case, a package with the same name must be installed already, and the installed package is replaced with the specified one. Replacing a package with a new package source is a generalization of fetching a replacement package that has a new checksum at a previously specified source.

1.6 Removing Packages

As you might expect, `raco pkg remove` removes a package:

```
raco pkg remove ⟨pkg-name⟩
```

If the installation of a package triggered auto-installs of other packages, then removing the package *does not* automatically remove the auto-installed packages. Supply the `--auto` flag for `raco pkg remove`, either by itself or when uninstalling packages, to also remove any auto-installed packages that are left without dependents.

The `raco pkg remove` command will not remove a package if other installed packages depend on it, unless you force the removal. If you want to demote a package from explicitly installed to auto-installed (for clean-up later, perhaps when other packages are removed), then supply the `--demote` flag to `raco pkg remove`.

1.7 Creating Packages

A package normally starts life as a directory containing module files and grows up to become a Github repository that is registered with a package catalog.

So, to create a package, first make a directory and select its name, ⟨*pkg-name*⟩:

```
mkdir ⟨pkg-name⟩
```

Although a package can provide libraries in any number of collections, it's common for a package to provide only libraries in a collection that matches the package name. If that's the case for your package, then files implementing modules in the ⟨*pkg-name*⟩ collection will go directly in the ⟨*pkg-name*⟩ directory that you have created.

If your package implements multiple collections, then you'll need to add a basic `"info.rkt"` file in the ⟨*pkg-name*⟩ directory:

```
cd ⟨pkg-name⟩

echo "#lang info" > info.rkt

echo "(define collection 'multi)" >> info.rkt
```

The `collection` definition tells the package manager that the package implements libraries in multiple collections, and each collection is represented by a sub-directory whose name matches the collection name. Libraries for each collection go in the collection's directory.

You can start with a single-collection package and later change it to a multi-collection package by restructuring the package directory, so you don't have to worry much about the choice when you get started.

Whether creating a single-collection package or a multi-collection package, the next step is to link your development directory as a locally installed package. Use `raco pkg install` in the ⟨*pkg-name*⟩ directory:

```
raco pkg install
```

If you use `raco pkg show` at this point, you'll see a line for ⟨*pkg-name*⟩. The "Source" column will show that it's a linked package, and the "Checksum" column will say `#f`, which means that there is no checksum. Sub-commands like `raco pkg update` will not work on a linked package, because "updates" to the package happen whenever you modify the package's implementation.

Finally, inside the ⟨*pkg-name*⟩ directory, add directories and/or files to implement the collections and/or modules that your package provides. For example, the developer of a `tic-tac-toe` multi-collection package that provides `games/tic-tac-toe/main` and `data/matrix` libraries might create directories and files like this:

```
mkdir -p games/tic-tac-toe

touch games/tic-tac-toe/info.rkt

touch games/tic-tac-toe/main.rkt

mkdir -p data

touch data/matrix.rkt
```

An `"info.rkt"` file is not necessary for a single-collection package with no dependencies, but you may wish to create one, anyway, to hold dependency declarations. Every package at least depends on `base`, which provides the collections and libraries of a minimal Racket installation. To make your package work best for other users, you will ultimately need to declare all dependencies. (Fortunately, `raco setup` can check dependencies and help you figure out what dependencies to declare.)

Even for a single-collection package, you may want to create `"info.rkt"` and include the definition

```
(define collection "⟨pkg-name⟩")
```

This definition may seem redundant, since ⟨*pkg-name*⟩ is available as the name of the enclosing directory, but declaring the collection name explicitly prevents the meaning of your package's implementation from depending on the way that the implementation is referenced.

Finally, in the case of a multi-collection package, note that the `"info.rkt"` file in ⟨*pkg-name*⟩ is for the package, not for a collection. Definitions such as `scribblings` or `raco-commands` work only in a collection's `"info.rkt"`. For a single-collection package, the `"info.rkt"` file serves double-duty for the package and collection.

1.8 Sharing Packages

After your package is ready to deploy, choose either §1.8.1 "GitHub Deployment" or §1.8.2 "Manual Deployment", and then go on to §1.8.3 "Helping Others Discover Your Package".

1.8.1 GitHub Deployment

First, create a free account on GitHub, then create a repository for your package. After that, your package source is:

`git://github.com/`⟨*user*⟩`/`⟨*package*⟩

If you want the package to be ⟨*branch*⟩ instead of `master`, then add `"#`⟨*branch*⟩`"` to the end of the package source.

Whenever you

```
git push
```

your changes will automatically be discovered by those who use `raco pkg update` after installing from your github-based package source.

1.8.2 Manual Deployment

Alternatively, you can deploy your package by publishing it on a URL you control. If you do this, it is preferable to create an archive from your package directory first:

`raco pkg create` ⟨*package*⟩

Then, upload the archive and its checksum to your site:

`scp` ⟨*package*⟩`.zip` ⟨*package*⟩`.zip.CHECKSUM your-host:public_html/`

Your package source is then something like

`http://your-host/~`⟨*user*⟩`/`⟨*package*⟩`.zip`

Whenever you want to provide a new release of a package, recreate and reupload the package archive (and checksum). Your changes will automatically be discovered by those who used your package source when they use `raco pkg update`.

By default, `raco pkg create` generates a `".zip"` archive. For more options, refer to the `raco pkg create` documentation. If you want to generate an archive through some other means, simply archive what you made in the first part of this section. For more formal details, refer to the package definition.

1.8.3 Helping Others Discover Your Package

By using either §1.8.1 “GitHub Deployment” or §1.8.2 “Manual Deployment”, anyone will be able to install your package by referring to your package source. However, they will not be able to refer to it by a simple name until it is listed on a package catalog.

If you'd like to use the PLT package catalog, browse to http://pkgs.racket-lang.org/ and upload a new package. You will need to create an account and log in first.

You only need to go to this site *once* to list your package. The server will periodically check the package source you designate for updates.

If you use this server, and if you use GitHub for deployment, then you will never need to open a web browser to update your package for end users. You just need to push to your GitHub repository, then within 24 hours, the PLT package catalog will notice, and `raco pkg update` will work on your user's machines.

1.8.4 Naming and Designing Packages

We suggest the following conventions for naming and designing packages:

- Packages should not include the name of the author or organization that produces them, but be named based on the content of the package. For example, `data-priority-queue` is preferred to `johns-amazing-queues`.
- Packages that provide an interface to a foreign library or service should be named the same as the service. For example, `cairo` is preferred to `Racket-cairo` or a similar name.
- Packages should not generally contain version-like elements in their names, initially. Instead, version-like elements should be added when backwards incompatible changes are necessary. For example, `data-priority-queue` is preferred to `data-priority-queue1`. Exceptions include packages that present interfaces to external, versioned things, such as `sqlite3` or `libgtk2`.
- A version declaration for a package is used only by other package implementors to effectively declare dependencies on provided features. Such declarations allow `raco pkg install` and `raco pkg update` to help check dependencies. Declaring and changing a version is optional, and the package catalog ignores version declarations; in particular, a package is a candidate for updating when its checksum changes, independent of whether the package's version changes or even in which direction the version changes. We suggest using a version smaller than `"1.0"` to indicate that a package's interface is unstable and changing it to `"1.0"` when you are ready to commit to backwards compatibility going forward.
- Packages should not combine large sets of utilities libraries with other functionality. For example, a package that contain many extensions to the `"racket"` collection, like `"racket/more-lists.rkt"` and `"racket/more-bools.rkt"` should not also contain complete applications, as other packages interested in the `"racket/more-bools.rkt"` library will not wish to depend on in such application.

- Packages should normally include both documentation and implementation. To make the implementation of a package available separately from its documentation (for use in environments where local documentation is not useful), define a package ⟨*pkg-name*⟩`-lib` to hold just the implementation, ⟨*pkg-name*⟩`-doc` to hold the documentation, and ⟨*pkg-name*⟩ that depends on both and that "re-exports" both with an `implies` declaration (see §4 "Package Metadata"). If you want to keep tests separate, put them a ⟨*pkg-name*⟩`-test` package that is *not* a dependency of ⟨*pkg-name*⟩. Similarly, use ⟨*pkg-name*⟩`-exe` for executables.

- Packages should generally provide one collection with a name similar to the name of the package. For example, `libgtk1` should provide a collection named `"libgtk"`. Exceptions include extensions to existing collection, such as new data-structures for the `"data"` collection, DrRacket tools, new games for PLT Games, etc.

1.8.5 Packages Compatible with Racket 5.3.5 and 5.3.6

A beta version of the package system was added to Racket starting in version 5.3.5. By the time version 6.0 was released, some features were added.

By using only certain older features, it is possible to make a package that can be used with Racket versions 5.3.5, 5.3.6, 6.0, and newer.

In your `info.rkt`, you should:

- Use `#lang setup/infotab` (not `#lang info`).

- Use `(define collection 'multi)`. Even if your package has a single collection, put it in a subdirectory and make a multi-collection package.

- If you depend on a specific version of another package, state this using the `(`*`other-package-name required-version`*`)` form (not the form with `#:version`).

Finally, when listing your package on `http://pkgs.racket-lang.org`, you should supply a GitHub source using the URL format `github://github.com/`⟨*user*⟩`/`⟨*repo*⟩`/`⟨*rev*⟩[`/`⟨*path*⟩] (not the `git:` format).

Version Exceptions

To make supporting multiple versions of Racket easier, the package catalog software supports *version exception*s. Version exceptions allow package authors to specify alternative package sources to be used when installing a given package using a specific version of Racket.

For example, a package that uses on Racket 6.0-specific features could provide a version exception for Racket 5.3.6 using a different branch in the package's Github repository, or a

different zip archive, as package source. Users installing the package from Racket 6.0 will use the default source for the package, while those using Racket 5.3.5 will use the alternative branch / archive.

For more details, see §7 “Package Catalog Protocol”.

2 Package Concepts

A *package* is a set of modules in some number of collections. Modules installed using the Racket package manager are `required` like any other modules. For example, if the package `tic-tac-toe` contains the module `"matrix.rkt"` in a `"data"` collection, then after `tic-tac-toe` is installed,

```
(require data/matrix)
```

imports the module. The package name is not mentioned with `require`, because packages are a way of managing library collections, not a way of referencing them. It is common, however, for a package to implement a collection whose name is the same as the package name—in which case a `require` might appear to be referencing a package, but it is actually referencing a collection provided by the package.

Each package has associated *package metadata*:

- a *package name* — a string made of the characters `a` through `z`, `A` through `Z`, `0` through `9`, `_`, and `-`.
- a *checksum* — a string that identifies different releases of a package. A package can be updated when its checksum changes, whether or not its version changes. The checksum normally can be computed as the SHA1 (see `openssl/sha1`) of the package's content.
- a *version* — a string of the form $\langle maj \rangle.\langle min \rangle$, $\langle maj \rangle.\langle min \rangle.\langle sub \rangle$, or $\langle maj \rangle.\langle min \rangle.\langle sub \rangle.\langle rel \rangle$, where $\langle maj \rangle$, $\langle min \rangle$, $\langle sub \rangle$, and $\langle rel \rangle$ are all canonical decimal representations of natural numbers, $\langle min \rangle$ has no more than two digits, and $\langle sub \rangle$ and $\langle rel \rangle$ has no more than three digits. A version is intended to reflect available features of a package, and should not be confused with different releases of a package as indicated by the checksum.
- a list of dependencies — a list of packages to be installed simultaneously, optionally with a lower bound on each package's version.

A package is typically represented by a directory with the same name as the package. The checksum is typically left implicit. The package directory can contain a file named `"info.rkt"` to declare other metadata (see §4 "Package Metadata").

2.1 Single-collection and Multi-collection Packages

A package can be a single-collection package or a multi-collection package:

- A *single-collection package*'s directory doubles as a collection directory. By default, the package name also doubles as the collection name, but if the package has an `"info.rkt"` file that defines `collection` to a string, then the string is used as the name of the package's collection.

- A *multi-collection package*'s directory contains subdirectories, each of which is a collection that is provided by the package (where the directory name is used as the collection name). A multi-collection package must have an `"info.rkt"` file that defines `collection` as `'multi`.

2.2 Package Sources

A *package source* identifies a package representation. Each package source type has a different way of storing the checksum and providing the package content (usually with single-collection package and multi-collection package variants).

The package source types are:

- a local file path naming an archive (as a plain path or `file://` URL) — The name of the package is the basename of the archive file. The checksum for archive `"f.`⟨*ext*⟩`"` is given by the file `"f.`⟨*ext*⟩`.CHECKSUM"`. The valid archive formats are (currently) `".zip"`, `".tar"`, `".tgz"`, `".tar.gz"`, and `".plt"`.

  For example, `"~/tic-tac-toe.zip"` is an archive package source, and its checksum would be inside `"~/tic-tac-toe.zip.CHECKSUM"`.

  An archive represents package content analogous to a directory, but if the archive's content is contained within a single top-level directory, then the directory's content (as opposed to the overall archive content) is used as the package content. The `".plt"` format does not accommodate either an extra directory layer or a single-collection package representation.

  A package source is inferred to refer to an archive file only when it has a suffix matching a valid archive format and when it starts with `file://` or does not start with alphabetic characters followed by `://`. The inferred package name is the filename without its suffix.

  Changed in version 6.0.1.12: Changed treatment of an archive that contains all content within a top-level directory.

- a local directory (as a plain path or `file://` URL) — The name of the package is the name of the directory. The checksum is not present.

  For example, `"~/tic-tac-toe/"` is a directory package source.

  A package source is inferred to refer to a directory only when it does not have a file-archive suffix, does not match the grammar of a package name, and either starts with starts with `file://` or does not start with alphabetic characters followed by `://`. The inferred package name is the directory name.

- a remote URL naming an archive — This type follows the same rules as a local file path, but the archive and checksum files are accessed via HTTP(S).

  For example, `"http://game.com/tic-tac-toe.zip"` is a remote URL package source whose checksum is found at `"http://game.com/tic-tac-toe.zip.CHECKSUM"`.

  A package source is inferred to be a URL only when it starts with `http://` or `https://`, and it is inferred to be a file URL when the URL ends with a path element that could be inferred as a file archive. The inferred package name is from the URL's file name in the same way as for a file package source.

- a remote URL naming a directory — The remote directory must contain a file named `"MANIFEST"` that lists all the contingent files. These are downloaded into a local directory and then the rules for local directory paths are followed. However, if the remote directory contains a file named `".CHECKSUM"`, then it is used to determine the checksum.

  For example, `"http://game.com/tic-tac-toe/"` is a directory URL package source whose checksum is found at `"http://game.com/tic-tac-toe/.CHECKSUM"`.

  A package source is inferred to be a URL the same for a directory or file, and it is treated as a directory URL when it does not end with a path element that has an archive file suffix. The inferred package name is the directory name.

- a remote URL naming a GitHub repository — The format for such URLs is:

  `git://github.com/`⟨*user*⟩`/`⟨*repo*⟩`[.git][/][?path=`⟨*path*⟩`][#`⟨*rev*⟩`]`

  where ⟨*path*⟩ can contain multiple `/`-separated elements to form a path within the repository, and defaults to the empty path. The ⟨*rev*⟩ can be a branch, tag, or commit, and it defaults to `master`.

  For example, `"git://github.com/game/tic-tac-toe#master"` is a GitHub package source.

  For backward compatibility, an older format is also supported:

  `github://github.com/`⟨*user*⟩`/`⟨*repo*⟩`/`⟨*rev*⟩`[/`⟨*path*⟩`]`

  The `zip`-formatted archive for the repository (generated by GitHub for any commit) is used as a remote URL archive path. The checksum is the hash identifying ⟨*rev*⟩ if ⟨*rev*⟩ is a branch or tag, otherwise ⟨*rev*⟩ itself serves as the checksum.

  A package source is inferred to be a GitHub reference when it starts with `git://` or `github://`; a package source that is otherwise specified as a GitHub reference is automatically prefixed with `"git://github.com/"`. The inferred package name is the last element of ⟨*path*⟩ if it is non-empty, otherwise the inferred name is ⟨*repo*⟩.

- a package name — A package catalog is consulted to determine the source and checksum for the package.

  For example, `tic-tac-toe` is a package name that can be used as a package source.

A package source is inferred to be a package name when it fits the grammar of package names, which means that it has only the characters `a` through `z`, `A` through `Z`, `0` through `9`, `_`, and `-`.

2.3 Package Catalogs

A *package catalog* is a server or database that converts package names to other package sources. A package catalog is identified by a string representing a URL, where a `http://` or `https://` URL indicates a remote server, and a `file://` URL indicates a local catalog in the form of an SQLite database or a directory tree.

PLT supports two package catalog servers that are enabled by default: `http://pkgs.racket-lang.org` for new packages and `http://planet-compats.racket-lang.org` for automatically generated packages for old PLaneT packages. Anyone may host a package catalog, and any file-serving HTTP host can act as a basic package catalog server. See §7 "Package Catalog Protocol" for information on how package information is extracted from a catalog.

2.4 Explicit vs. Auto-Installation

When a package is installed, the original source of its installation is recorded, as well as whether the instalation was an automatic installation. An *automatic installation* is one that was installed because it was a dependency of a non-automatic installation package.

2.5 Package Conflicts

Two packages are in *conflict* if they contain the same module. For example, if the package `tic-tac-toe` contains the module file `"data/matrix.rkt"` and the package `factory-optimize` contains the module file `"data/matrix.rkt"`, then `tic-tac-toe` and `factory-optimize` are in conflict.

A package may also be in conflict with Racket itself, if it contains a module file that is part of the base Racket implementation. For example, any package that contains `"racket/list.rkt"` is in conflict with Racket.

For the purposes of conflicts, a module is a file that ends in `".rkt"`, `".ss"`, or `".scrbl"`.

2.6 Package Updates

Package A is a *package update* of Package B if (1) B is installed, (2) A and B have the same name, and (3) A's checksum is different than B's. A single-collection package can be a package update of a multi-collection package and vice versa.

Note that a package version is not taken into account when determining a package update, although a change in a package's version (in either direction) implies a change in the checksum because the checksum is computed from the package source and the meta-data that specifies the version is part of the source.

2.7 Package Scopes

A *package scope* determines the effect of package installations, updates, etc., with respect to different users and Racket installations. The default package scope can be configured, but it is normally `user`, which makes actions specific to both the current user and the installation's name/version (in the sense of `get-installation-name`). The `installation` scope means that package operations affect all users of the Racket installation.

A directory path can be used as a package scope, in which case package operations affect the set of packages installations in the directory. An installation can be configured to include the directory in its search path for installed packages (see §18 "Installation Configuration and Search Paths").

Conflict checking disallows installation of the same or conflicting package in different scopes, but if such a configuration is forced, collections are found first in packages with `user` package scope. Search then proceeds in a configured order, where `installation` package scope typically precedes other directory package scopes.

3 Using `raco pkg`

The `raco pkg` command provides package-management tools via sub-commands.

3.1 `raco pkg install`

`raco pkg install` ⟨*option*⟩ ... ⟨*pkg-source*⟩ ... — Installs the given package sources (eliminating exact-duplicate ⟨*pkg-source*⟩s). If a given ⟨*pkg-source*⟩ is "auto-installed" (to satisfy some other package's dependency), then it is promoted to explicitly installed.

If no ⟨*pkg-source*⟩s are supplied, the current directory is installed as a link. See the `--link` flag below for more details.

The `install` sub-command accepts the following ⟨*option*⟩s:

- `--type` ⟨*type*⟩ or `-t` ⟨*type*⟩ — specifies an interpretation of the package source, where ⟨*type*⟩ is either `file`, `dir`, `file-url`, `dir-url`, `github`, or `name`.
- `--name` ⟨*pkg*⟩ or `-n` ⟨*pkg*⟩ — specifies the name of the package, which makes sense only when a single ⟨*pkg-source*⟩ is provided. The name is normally inferred for each ⟨*pkg-source*⟩.
- `--checksum` ⟨*checksum*⟩ — specifies a checksum for the package, which normally makes sense only when a single ⟨*pkg-source*⟩ is provided. The use of ⟨*checksum*⟩ depends on ⟨*pkg-source*⟩: for a GitHub source, ⟨*checksum*⟩ selects a checksum; for a package name, file path, or remote URL as a source, ⟨*checksum*⟩ specifies an expected checksum; for a directory path (including a remote directory URL without a `".CHECKSUM"` file) as a source, ⟨*checksum*⟩ assigns a checksum.
- `--deps` ⟨*behavior*⟩ — Selects the behavior for dependencies, where ⟨*behavior*⟩ is one of
  - `fail` — Cancels the installation if dependencies are uninstalled or version requirements are unmet. This behavior is the default for a ⟨*pkg-source*⟩ that is not a package name.
  - `force` — Installs the package(s) despite missing dependencies or version requirements. Forcing an installation may leave package content in an inconsistent state.
  - `search-ask` — Looks for dependencies (when uninstalled) or updates (when version requirements are unmet) via the configured package catalogs, but asks if you would like the packages installed or updated. This behavior is the default for a ⟨*pkg-source*⟩ that is a package name.
  - `search-auto` — Like `search-ask`, but does not ask for permission to install or update.

- `--auto` — Shorthand for `--deps search-auto`.

- `--update-deps` — With `search-ask` or `search-auto` dependency behavior, checks already-installed dependencies transitively for updates (even when not forced by version requirements), asking or automatically updating a package when an update is available. When a package is updated or installed, unless `--skip-implies` is specified, any package that it implies (see §4 "Package Metadata") is automatically updated independent of the behavior requested via `--update-deps` and `--deps`.

- `--skip-implies` — Disables special treatment of dependencies that are listed in `implies` (see §4 "Package Metadata") for an installed or updated package.

- `--link` — Implies `--type dir` (and overrides any specified type), and links the existing directory as an installed package, instead of copying the directory's content to install. Directory package sources are treated as links by default, unless `--copy` is specified.

  The package is identified as a single-collection package or a multi-collection package at the time that it is installed, and that categorization does not change even if the `collection` definition in `"info.rkt"` is changed (i.e., he package must be removed and re-installed for the change to take effect).

- `--static-link` — Implies `--link`, and also indicates that subdirectories of the given directory will not change for each given directory that implements a multi-collection package.

- `--pkgs` — Disables default installation of the current directory when no ⟨*pkg-source*⟩s are supplied.

- `--copy` — Disables default handling of directory package sources as links, and instead treats them like other sources: package content is copied to install.

- `--binary` — Strips source elements of a package before installing, and implies `--copy`.

- `--source` — Strips built elements of a package before installing, and implies `--copy`.

- `--scope` ⟨*scope*⟩ — Selects the package scope for installation, where ⟨*scope*⟩ is one of

  - `installation` — Install packages for all users of a Racket installation, rather than user-specific.
  - `user` — Install packages for the current user and current installation's name/version.

  The default package scope is normally `user`, but it can be configured with `raco pkg config --set default-scope` ⟨*scope*⟩. The default installation name is normally the Racket version, but it can be configured with `raco pkg config --set name` ⟨*name*⟩.

- `-i` or `--installation` — Shorthand for `--scope installation`.
- `-u` or `--user` — Shorthand for `--scope user`.
- `--scope-dir` ⟨*dir*⟩ — Select ⟨*dir*⟩ as the package scope.
- `--catalog` ⟨*catalog*⟩ — Use ⟨*catalog*⟩ instead of of the currently configured package catalogs.
- `--skip-installed` — Ignore any ⟨*pkg-source*⟩ whose name corresponds to an already-installed package, except for promoting auto-installed packages to explicitly installed.
- `--all-platforms` — Considers package dependencies independent of the current platform (instead of filtering dependencies to platforms other than the current one).
- `--force` — Ignores module conflicts, including conflicts due to installing a single package in multiple scopes. Forcing an installation may leave package content in an inconsistent state.
- `--ignore-checksums` — Ignores errors verifying package checksums (unsafe).
- `--strict-doc-conflicts` — Refuses to install in user package scope when documentation-directory names would conflict with other packages. "Conflicting" documentation names are normally allowed for installation in user package scope, but strict checking ensures that installation would succeed in other package scopes.
- `--no-cache` — Disables use of the download cache.
- `--no-setup` — Does not run `raco setup` after installation. This behavior is also the case if the environment variable `PLT_PKG_NOSETUP` is set to any non-empty value.
- `--jobs` ⟨*n*⟩ or `-j` ⟨*n*⟩ — Install and setup with ⟨*n*⟩ parallel jobs.
- `--fail-fast` — Breaks `raco setup` as soon as any error is encountered.

3.2 `raco pkg update`

`raco pkg update` ⟨*option*⟩ ... ⟨*pkg-source*⟩ ... — Checks the specified package names for package updates or replaces existing package installations with the given sources. If an update or replacement cannot be installed (e.g. it conflicts with another installed package), then this command fails without installing any of the ⟨*pkg-source*⟩s (or their dependencies).

If a package scope is not specified, the scope is inferred from the given ⟨*pkg-source*⟩s.

The `update` sub-command accepts the following ⟨*option*⟩s:

- `--all` or `-a` — Update all packages, if no packages are given in the argument list.

- `--lookup` — Checks Causes a package name as a ⟨*pkg-source*⟩ to be used as a replacement, instead of the name of a installed package that may have updates. (If the named package was installed through a package name, then there's effectively no difference.)
- `--type` ⟨*type*⟩ or `-t` ⟨*type*⟩ — Same as for `raco pkg install`.
- `--name` ⟨*pkg*⟩ or `-n` ⟨*pkg*⟩ — Same as for `raco pkg install`.
- `--checksum` ⟨*checksum*⟩ — Same as for `raco pkg install`.
- `--deps` ⟨*behavior*⟩ — Same as for `raco pkg install`.
- `--auto` — Shorthand for `--deps search-auto` plus `--update-deps`.
- `--update-deps` — Same as for `raco pkg install`, but implied by `--auto` only for `raco pkg update`.
- `--skip-implies` — Same as for `raco pkg install`.
- `--link` — Same as for `raco pkg install`.
- `--static-link` — Same as for `raco pkg install`.
- `--binary` — Same as for `raco pkg install`.
- `--copy` — Same as for `raco pkg install`.
- `--source` — Same as for `raco pkg install`.
- `--scope` ⟨*scope*⟩ — Selects a package scope, the same as for `raco pkg install`.
- `-i` or `--installation` — Shorthand for `--scope installation`.
- `-u` or `--user` — Shorthand for `--scope user`.
- `--scope-dir` ⟨*dir*⟩ — Selects ⟨*dir*⟩ as the package scope, the same as for `raco pkg install`.
- `--catalog` ⟨*catalog*⟩ — Same as for `raco pkg install`.
- `--all-platforms` — Same as for `raco pkg install`.
- `--force` — Same as for `raco pkg install`.
- `--ignore-checksums` — Same as for `raco pkg install`.
- `--strict-doc-conflicts` — Same as for `raco pkg install`.
- `--no-cache` — Same as for `raco pkg install`.
- `--no-setup` — Same as for `raco pkg install`.
- `--jobs` ⟨*n*⟩ or `-j` ⟨*n*⟩ — Same as for `raco pkg install`.

3.3 `raco pkg remove`

`raco pkg remove` ⟨*option*⟩ ... ⟨*pkg*⟩ ... — Attempts to remove the given packages. By default, if a package is the dependency of another package that is not listed, this command fails without removing any of the ⟨*pkg*⟩s.

If a package scope is not specified, the scope is inferred from the given ⟨*pkg*⟩s.

The `remove` sub-command accepts the following ⟨*option*⟩s:

- `--demote` — "Remove" explicitly installed packages by demoting them to auto-installed (leaving auto-installed packages as such). Combined with `--auto`, removes packages for which there are no dependencies.
- `--force` — Ignore dependencies when removing packages.
- `--auto` — In addition to removing each ⟨*pkg*⟩, remove auto-installed packages (i.e., installed by the `search-auto` or `search-ask` dependency behavior, or demoted via `--demote`) that are no longer required by any explicitly installed package.
- `--scope` ⟨*scope*⟩ — Selects a package scope, the same as for `raco pkg install`.
- `-i` or `--installation` — Shorthand for `--scope installation`.
- `-u` or `--user` — Shorthand for `--scope user`.
- `--scope-dir` ⟨*dir*⟩ — Selects ⟨*dir*⟩ as the package scope, the same as for `raco pkg install`.
- `--no-setup` — Same as for `raco pkg install`.
- `--jobs` ⟨*n*⟩ or `-j` ⟨*n*⟩ — Same as for `raco pkg install`.

3.4 `raco pkg show`

`raco pkg show` ⟨*option*⟩ ... — Print information about currently installed packages. By default, packages are shown for all package scopes, but only for packages not marked as auto-installed to fulfill dependencies.

The `show` sub-command accepts the following ⟨*option*⟩s:

- `-a` or `--all` — Includes auto-installed packages in the listing.
- `-d` or `--dir` — Adds a column in the output for the directory where the package is installed.
- `--scope` ⟨*scope*⟩ — Shows only packages in ⟨*scope*⟩, which is one of

  - `installation` — Show only installation-wide packages.
  - `user` — Show only user-specific packages for the current installation's name/version or the name/version specified with `--version` or `-v`.

  The default is to show packages for all package scopes.

- `-i` or `--installation` — Shorthand for `--scope installation`.
- `-u` or `--user` — Shorthand for `--scope user`.
- `--scope-dir` ⟨*dir*⟩ — Shows only packages installed in ⟨*dir*⟩.
- `--version` ⟨*vers*⟩ or `-v` ⟨*vers*⟩ — Show only user-specific packages for the installation name/version ⟨*vers*⟩.

3.5 `raco pkg migrate`

`raco pkg migrate` ⟨*option*⟩ ... ⟨*from-version*⟩ — Installs packages that were previously installed in `user` package scope for ⟨*from-version*⟩, where ⟨*from-version*⟩ is an installation name/version.

The `migrate` sub-command accepts the following ⟨*option*⟩s:

- `--deps` ⟨*behavior*⟩ — Same as for `raco pkg install`, except that `search-auto` is the default.
- `--binary` — Same as for `raco pkg install`.
- `--source` — Same as for `raco pkg install`.
- `--scope` ⟨*scope*⟩ — Same as for `raco pkg install`.
- `-i` or `--installation` — Shorthand for `--scope installation`.
- `-u` or `--user` — Shorthand for `--scope user`.
- `--scope-dir` ⟨*dir*⟩ — Select ⟨*dir*⟩ as the package scope.
- `--catalog` ⟨*catalog*⟩ — Same as for `raco pkg install`.
- `--all-platforms` — Same as for `raco pkg install`.
- `--force` — Same as for `raco pkg install`.
- `--ignore-checksums` — Same as for `raco pkg install`.
- `--strict-doc-conflicts` — Same as for `raco pkg install`.
- `--no-cache` — Same as for `raco pkg install`.
- `--no-setup` — Same as for `raco pkg install`.
- `--jobs` ⟨*n*⟩ or `-j` ⟨*n*⟩ — Same as for `raco pkg install`.

3.6 `raco pkg create`

`raco pkg create` ⟨*option*⟩ ... ⟨*directory-or-package*⟩ — Bundles a package into an archive. Bundling is not needed for a package that is provided directly from a GitHub repository or other non-archive formats. The `create` sub-command can create an archive from a directory (the default) or from an installed package. It can also adjust the archive's content to include only sources, only compiled bytecode and rendered documentation, or both—but packages are normally provided as source and converted to binary form by an automatic service, instead of by a package author.

The `create` sub-command accepts the following ⟨*option*⟩s:

- `--from-dir` — Treat ⟨*directory-or-package*⟩ as a directory path; this is the default mode.
- `--from-install` — Treat ⟨*directory-or-package*⟩ as the name of an installed package (instead of a directory).
- `--format` ⟨*format*⟩ — Specifies the archive format. The allowed ⟨*format*⟩s are: `zip` (the default), `tgz`, and `plt`. This option must be specified if `--manifest` is not present.
- `--manifest` — Creates a manifest file for a directory, rather than an archive.
- `--as-is` — Bundle all content of the package directory as is, with no filtering of sources, compiled files, or repository elements.
- `--source` — Bundle only sources in the package directory; see §5 "Source, Binary, and Built Packages".
- `--binary` — Bundle compiled bytecode and rendered documentation in the package directory; see §5 "Source, Binary, and Built Packages".
- `--built` — Bundle compiled sources, bytecode, and rendered documentation in the package directory, filtering repository elements; see §5 "Source, Binary, and Built Packages".
- `--dest` ⟨*dest-dir*⟩ — Writes generated bundles to ⟨*dest-dir*⟩.

3.7 `raco pkg config`

`raco pkg config` ⟨*option*⟩ ... [⟨*key*⟩] ⟨*val*⟩ ... — View and modify the configuration of the package manager. If ⟨*key*⟩ is not provided, the values for all recognized keys are shown. The ⟨*val*⟩ arguments are allowed only when `--set` is used, in which case the ⟨*val*⟩s are used as the new values for ⟨*key*⟩.

The `config` sub-command accepts with the following ⟨*option*⟩s:

- `--set` — Sets an option, rather than printing it.
- `--scope` ⟨*scope*⟩ — Selects a package scope, the same as for `raco pkg install`. A configuration value set at `installation` scope serves as the default value at `user` scope.
- `-i` or `--installation` — Shorthand for `--scope installation`.
- `-u` or `--user` — Shorthand for `--scope user`.
- `--scope-dir` ⟨*dir*⟩ — Same as for `raco pkg install`.

The valid ⟨*key*⟩s are:

- `name` — A string for the installation's name, which is used by `user` package scope and defaults to the Racket version.
- `catalogs` — A list of URLs for package catalogs.
- `default-scope` — Either `installation` or `user`. The value of this key at `user` scope (possibly defaulting from `installation` scope) is the default package scope for `raco pkg` commands for which a scope is not inferred from a given set of package names (even for `raco pkg config`, which is consistent but potentially confusing).
- `download-cache-dir` — A directory that holds copies of downloaded packages, used to avoid re-downloading if the same URL and checksum combination is requested again. The default cache directory is user-specific (but not specific to a Racket version or installation name).
- `download-cache-max-files` — A limit on the number of files to be kept in the download cache directory.
- `download-cache-max-bytes` — A limit on the total size of files that are kept in the download cache directory.
- `doc-open-url` — A URL to use in place of a local filesystem path for viewing (or at least searching) documentation; an empty string, which is the default, disables the URL so that the local filesystem is used. This key can be set only in `installation` scope.

3.8 `raco pkg catalog-show`

`raco pkg catalog-show` ⟨*option*⟩ ... ⟨*package-name*⟩ ... — Consults package catalogs for a package (that is not necessarily installed) and displays the catalog's information for the package, such as its source URL and a checksum.

The `catalog-show` sub-command accepts the following ⟨*option*⟩s:

- `--all` — Show information for all available packages. When using this flag, supply no ⟨*packaee-name*⟩s.
- `--only-names` — Show only package names. This option is mainly useful with `--all`, but when a ⟨*packaee-name*⟩ is provided, catalogs are consulted to ensure that he package is available.
- `--modules` — Show the modules that are implemented by a package.
- `--catalog` ⟨*catalog*⟩ — Query ⟨*catalog*⟩ instead of the currently configured package catalogs.
- `--version` ⟨*version*⟩ or `-v` ⟨*version*⟩ — Query catalogs for a result specific to ⟨*version*⟩, instead of the installation's Racket version.

3.9 `raco pkg catalog-copy`

`raco pkg catalog-copy` ⟨*option*⟩ ... ⟨*src-catalog*⟩ ... ⟨*dest-catalog*⟩ — Copies information from the package catalog named by ⟨*src-catalog*⟩s to a local database or directory ⟨*dest-catalog*⟩, which can be used as a new package catalog.

The ⟨*src-catalog*⟩s can be remote or local, while ⟨*dest-catalog*⟩ must be local (i.e., a directory path or a SQLite database path, as inferred from the path). If a ⟨*src-catalog*⟩ or ⟨*dest-catalog*⟩ does not start with a URL scheme, it is treated as a filesystem path. Information from multiple ⟨*src-catalog*⟩s is merged, with information from earlier ⟨*src-catalog*⟩s taking precedence over later ⟨*src-catalog*⟩s.

The `catalog-copy` sub-command accepts the following ⟨*option*⟩s:

- `--from-config` — Adds the currently configured package catalogs to the end of the ⟨*src-catalog*⟩s list.
- `--force` — Replaces ⟨*dest-catalog*⟩ if it exists already.
- `--merge` — Adds to ⟨*dest-catalog*⟩ if it exists already. By default, information already in ⟨*dest-catalog*⟩ takes precedence over new information.
- `--override` — Changes merging so that new information takes precedence over information already in ⟨*dest-catalog*⟩.
- `--relative` — Write package sources to ⟨*dest-catalog*⟩ in relative-path form, when possible.
- `--version` ⟨*version*⟩ or `-v` ⟨*version*⟩ — Copy catalog results specific to ⟨*version*⟩ (for catalogs that make a distinction), instead of the installation's Racket version.

3.10 `raco pkg catalog-archive`

`raco pkg catalog-archive` ⟨*option*⟩ ... ⟨*dest-dir*⟩ ⟨*src-catalog*⟩ ... — Copies information from the package catalog named by ⟨*src-catalog*⟩s to a `"catalog"` directory catalog in ⟨*dest-dir*⟩, and also copies all package sources to a `"pkgs"` directory in ⟨*dest-dir*⟩.

Packages sources are downloaded and repacked as needed, so that all packages are written to the `"pkgs"` directory as `".zip"` archives. This conversion may change the checksum on each archived package.

The `catalog-archive` sub-command accepts the following ⟨*option*⟩s:

- `--from-config` — Adds the currently configured package catalogs to the end of the ⟨*src-catalog*⟩s list.
- `--state` ⟨*state-database*⟩ — To enable incremental updating, Reads and writes the database ⟨*state-database*⟩, which must have the suffix `".sqlite"`, as the current state of ⟨*dest-dir*⟩.
- `--relative` — Write package sources to ⟨*dest-catalog*⟩ in relative-path form.
- `--version` ⟨*version*⟩ or `-v` ⟨*version*⟩ — Copy catalog results specific to ⟨*version*⟩ (for catalogs that make a distinction), instead of the installation's Racket version.

Added in version 6.0.17.

3.11 `raco pkg archive`

`raco pkg archive` ⟨*option*⟩ ... ⟨*dest-dir*⟩ ⟨*pkg*⟩ ... — Copies information from installed packages named by ⟨*pkgs*⟩s to a `"catalog"` directory catalog in ⟨*dest-dir*⟩, and also copies all package sources to a `"pkgs"` directory in ⟨*dest-dir*⟩.

Packages sources are copied and repacked as needed, so that all packages are written to the `"pkgs"` directory as `".zip"` archives. This conversion may change the checksum on each archived package.

The `archive` sub-command accepts the following ⟨*option*⟩s:

- `--include-deps` — Includes the dependencies of the specified packages in the resulting catalog.
- `--exclude` ⟨*pkg*⟩ — Omits the specified ⟨*pkg*⟩ from the resulting catalog. This also causes the dependencies of ⟨*pkg*⟩ to be omitted if `--include-deps` is specified. This flag can be provided multiple times.

- `--relative` — Write package sources to ⟨*dest-catalog*⟩ in relative-path form.

Added in version 6.1.0.8.

4 Package Metadata

Package metadata, including dependencies on other packages, is reported by an `"info.rkt"` module within the package. This module must be implemented in the `info` language.

For example, a basic `"info.rkt"` file might be

```
#lang info
(define version "1.0")
(define deps (list _package-source-string ...))
```

The following `"info.rkt"` fields are used by the package manager:

- `collection` — either `'multi` to implement a multi-collection package or a string or `'use-pkg-name` to implement a single-collection package. If `collection` is defined as a string, then the string is used as the name of the collection implemented by the package. If `collection` is defined as `'use-pkg-name`, then the package name is used as the package's collection name.

  Beware that omitting `collection` or defining it as `'use-pkg-name` means that a package's content effectively changes with the package's name. A package's content should normally be independent of the package's name, and so defining `collection` to a string is preferable for a single-collection package.

- `version` — a version string. The default version of a package is `"0.0"`.

- `deps` — a list of dependencies, where each dependency has one of the following forms:

  - A string for a package source.
  - A list of the form

    ```
    (list package-source-string
          keyword-and-spec ...)
    ```

    where each *keyword-and-spec* has a distinct keyword in the form

    ```
    keyword-and-spec = '#:version version-string
                     | '#:platform platform-spec

       platform-spec = string
                     | symbol
                     | regexp
    ```

    A *version-string* specifies a lower bound on an acceptable version of the needed package.

A *platform-spec* indicates that the dependency applies only for platforms with a matching result from (system-type) when *platforms-spec* is a symbol or (path->string (system-library-subpath #f)) when *platform-spec* is a string or regular expression. See also matching-platform?. For example, platform-specific binaries can be placed into their own packages, with one separate package and one dependency for each supported platform.

– A list of the form

```
(list package-source-string version-string)
```

which is deprecated and equivalent to

```
(list package-source-string '#:version version-string)
```

Each element of the deps list determines a dependency on the package whose name is inferred from the package source (i.e., dependencies are on package names, not package sources), while the package source indicates where to get the package if needed to satisfy the dependency.

Using the package name "racket" specifies a dependency on the Racket run-time system, which is potentially useful when a version is included in the dependency. For most purposes, it's better to specify a versioned dependency on "base", instead.

- build-deps — like deps, but for dependencies that can be dropped in a binary package, which does not include sources; see §5 "Source, Binary, and Built Packages". The build-deps and deps lists are appended, while raco pkg create strips away build-deps when converting a package for --binary mode.

- implies — a list of strings and 'core. Each string refers to a package listed in deps and indicates that a dependency on the current package counts as a dependency on the named package; for example, the gui package is defined to ensure access to all of the libraries provided by gui-lib, so the "info.rkt" file of gui lists "gui-lib" in implies. Packages listed in implies list are treated specially by updating: implied packages are automatically updated whenever the implying package is updated. The special value 'core is intended for use by an appropriate base package to declare it as the representative of core Racket libraries.

- update-implies — a list of strings. Each string refers to a package listed in deps or build-deps and indicates that the implied packages are automatically updated whenever the implying package is updated.

- setup-collects — a list of path strings and/or lists of path strings, which are used as collection names to set up via raco setup after the package is installed, or 'all to indicate that all collections need to be setup. By default, only collections included in the package are set up (plus collections for global documentation indexes and links).

- package-content-state — a list of two items; the first item is 'binary, 'binary-lib, or 'built, and the second item is either #f or a string to represent a Racket

version for compiled content. This information is used by `raco pkg install` or `raco pkg update` witj `--source`, `--binary`, or `--binary-lib` to ensure that the package content is consistent with the requested conversion; see also §5 “Source, Binary, and Built Packages”. Absence of this definition is treated the same as `(list 'source #f)`.

Changed in version 6.1.0.5: Added `update-implies`.

5 Source, Binary, and Built Packages

A package, especially in a repository format, normally provides module implementations and documentation in source form. Such *source packages* may work with multiple Racket versions, and modules are compiled to bytecode and documentation is rendered when the package is installed.

A *binary package* provides only compiled bytecode and rendered documentation, instead of package and documentation sources. Since compiled bytecode is specific to a version of Racket, a binary package is specific to a version of Racket. The benefit of a binary package is that it can have fewer dependencies (e.g., on Scribble to implement the documentation or on other packages whose documentation is referenced) and it can be installed faster. A drawback of a binary package is that it is version-specific and the source code may be less immediately accessible to other programmers.

A *binary library package* is like a binary package, but it further omits documentation.

A *built package* combines source and compiled elements. A built package can be installed more quickly than source, as long as it is installed for a suitable Racket version, but the source remains available as a back-up for other programmers to consult or to re-build for a different Racket version.

Programmers normally supply only source packages, while a package catalog service may convert each source package to a binary package, binary library package, or built package. Alternatively, programmers can create binary packages, binary library packages, or built packages by using the `raco pkg create` subcommand with `--binary`, `--binary-lib`, or `--built`. As a convenience, the `raco pkg create` subcommand can also create any kind of package from an installed package or repository checkout, dropping repository elements (such as a `".git"` directory) and compiled code. Note that `raco pkg create` by default bundles a package directory as-is, with no filtering or annotation.

Although a package can be specifically annotated as a source package, binary package, binary library package, or built package (see `package-content-state` in §4 "Package Metadata"), the different kinds of packages are primarily just conventions based on the content of the package. All forms of packages can be mixed in an installation, and a package can be updated from any form to any other form. Furthermore, `raco pkg install` and `raco pkg update` support `--source`, `--binary`, `--binary-lib` flags to convert to a package on installation; in that case, the package's existing annotation is checked to verify that it is consistent with the requested conversion.

Creating a source package, binary package, binary library package, or built package from a directory or package installation prunes the following files and directories:

- directories/files named `".svn"`;
- directories/files whose names start with `".git"`;

- directories/files whose names end with "~"; and
- directories/files whose names start and end with "#".

Any of the above can be suppressed, however, by a source-keep-files (for source package and built package bundling), binary-keep-files (for binary package, binary library package built package bundling), or binary-lib-keep-files (for binary library package bundling) definition in an "info.rkt" in the package or any subdirectory. A source-keep-files, binary-keep-files, or binary-lib-keep-files definition should bind the name to a list of paths relative to the "info.rkt" file.

Creating a source package prunes the following additional files and directories:

- directories/files named "compiled";
- directories/files named "doc";
- directories/files named "synced.rktd", which can appear as a marker in rendered documentation;
- directories/files named in an "info.rkt" file's source-omit-files definition.

Any of the above removals can be suppressed through source-keep-files—even for files and directories within an otherwise removed directory.

Creating a binary package prunes the following additional files and directories:

- directories/files with names ending in ".rkt" or ".ss" for which a corresponding compiled bytecode file is present (in a "compiled" subdirectory), not counting "info.rkt";
- directories/files with names ending in ".scrbl", "_scrbl.zo", or ".dep";
- directories/files ending with ".css" or ".js" immediately within a directory named "doc";
- directories/files named "tests" or "scribblings" (but see the exception below for "doc" and "info.rkt");
- directories/files named in an "info.rkt" file's binary-omit-files definition.

Any of the above removals can be suppressed through binary-keep-files—even files and directories within an otherwise removed directory. Furthermore, a "doc" or "info.rkt" directory/file is kept when it is within a "scribblings" directory and not within a "tests" directory. Creating a binary package further adjusts the following files (when they are not pruned):

- for any file whose name ends in ".zo", submodules named test, doc, or srcdoc are removed;
- for each ".html" file that refers to a "local-redirect.js" script, the path to the script is removed;
- each "info.rkt" is adjusted as follows: an assume-virtual-sources definition is added, any copy-foreign-libs definition is changed to move-foreign-libs, any copy-shared-files definition is changed to move-shared-files, any copy-man-pages definition is changed to move-man-pages, any build-deps definition is removed, any update-implies definition is removed, and a package-content-state is added to changed to (list 'binary (version)); and
- each collection within the path gets an "info.rkt" if it did not have one already, so that assume-virtual-sources can be defined.

Creating a binary library package prunes the following additional files and directories and makes additional adjustments compared to a binary package:

- directories/files named "doc" are removed;
- directories/files named in an "info.rkt" file's binary-lib-omit-files definition are removed; and
- each "info.rkt" is adjusted to remove any scribblings definition, and package-content-state is adjusted to (list 'binary-lib (version)).

Any of the removals can be suppressed through binary-lib-keep-files, in addition to suppressions through binary-keep-files.

Creating a built package removes any file or directory that would be removed for a source package *and* binary package, it performs the ".html" file updating of a binary package, and the package's "info.rkt" file (added if it does not exist already) is adjusted to define package-content-state as (list 'built (version)).

Finally, creating a binary package, binary library package, or built package "unmoves" files that were installed via move-foreign-libs, move-shared-files, or move-man-pages definitions in an "info.rkt" file, retrieving them if they are not present at referenced location but are present in a user-specific target directory (i.e., the directory reported by find-user-lib-dir, find-user-share-dir, or find-user-man-dir, respectively). On Mac OS X, when an unmoved file for move-foreign-libs is a Mach-O file that includes a reference to another library in one of the directories reported by (get-lib-search-dirs), then the reference is changed to a @loader_path/ reference.

(require pkg/strip) package: base

The pkg/strip module provides support for copying a package-style directory to a given destination with the same file/directory omissions and updates as raco pkg create.

```
(generate-stripped-directory mode
                             src-dir
                             dest-dir) → void?
  mode : (or/c 'source 'binary 'binary-lib 'built)
  src-dir : path-string?
  dest-dir : path-string?
```

Copies *src-dir* to *dest-dir* with file/directory omissions and updates corresponding the creation of a source package, binary package, binary library package, or built package as indicated by *mode*.

```
(check-strip-compatible mode
                        pkg-name
                        dir
                        error)   → any
  mode : (or/c 'source 'binary 'binary-lib 'built)
  pkg-name : string?
  dir : path-string?
  error : (string? . -> . any)
```

Check whether the content of *dir* is consistent with the given *mode* conversion according to the content of a `"info.rkt"` file in *dir*. If not, *error* is called with an error-message string to describe the mismatch.

```
(fixup-local-redirect-reference file
                                js-path) → void?
  file : path-string?
  js-path : string?
```

Assuming that *file* is an HTML file for documentation, adjusts the URL reference to `"local-redirect.js"`, if any, to use the prefix *js-path*.

```
(strip-binary-compile-info) → boolean?
(strip-binary-compile-info compile?) → void?
  compile? : any/c
```

A parameter that determines whether `"info.rkt"` files are included in bytecode form when converting package content for a binary packages, binary library packages, and built packages.

6 Package APIs

The pkg provides a programmatic interface to the `raco pkg` commands, but additional libraries provide smaller building blocks and local-database support.

6.1 Functions for `raco pkg`

`(require pkg)` package: `base`

The pkg module provides a programmatic interface to the `raco pkg` sub-subcommands.

Each-long form option of the command-line interface is a keyword argument to the functions described below. An argument corresponding to `--type`, `--deps`, `--format`, or `--scope` accepts its argument as a symbol, while other flags that take arguments expect strings. An argument corresponding to `--scope` is also allowed to be a path string, as would be provided to `--scope-dir`. Options without argument correspond to keyword arguments that accept booleans, where `#t` is equivalent to the presence of the option.

`pkg-install-command : procedure?`

Implements `raco pkg install`.

`pkg-update-command : procedure?`

Implements `raco pkg update`.

`pkg-remove-command : procedure?`

Implements `raco pkg remove`.

`pkg-show-command : procedure?`

Implements `raco pkg show`.

`pkg-migrate-command : procedure?`

Implements `raco pkg migrate`.

`pkg-config-command : procedure?`

Implements `raco pkg config`.

```
pkg-create-command : procedure?
```

Implements `raco pkg create`.

```
pkg-catalog-show-command : procedure?
```

Implements `raco pkg catalog-show`.

```
pkg-catalog-copy-command : procedure?
```

Implements `raco pkg catalog-copy`.

```
pkg-catalog-archive-command : procedure?
```

Implements `raco pkg catalog-archive`.

Added in version 6.0.17 of package `base`.

```
pkg-archive-command : procedure?
```

Implements `raco pkg archive`.

Added in version 6.1.0.8 of package `base`.

6.2 Package Management Functions

`(require pkg/lib)` package: `base`

The `pkg/lib` library provides building blocks on which the `pkg` library and `raco pkg` commands are built. It re-exports the bindings of `pkg/path`.

```
(with-pkg-lock body ...+)
(with-pkg-lock/read-only body ...+)
```

Evaluates the *body*s while holding a lock to prevent concurrent modification to the package database for the current package scope. Use the `with-pkg-lock/read-only` form for read-only access. The lock is reentrant but not upgradable from read-only.

Use these form to wrap uses of functions from `pkg/lib` that are documented to require the lock. Other functions from `pkg/lib` take the lock as needed.

```
(current-pkg-scope) → (or/c 'installation 'user
                           (and/c path? complete-path?))
(current-pkg-scope scope) → void?
  scope : (or/c 'installation 'user
                (and/c path? complete-path?))
(current-pkg-scope-version) → string?
(current-pkg-scope-version s) → void?
  s : string?
```

Parameters that determine package scope for management operations and, in the case of `'user` scope, the relevant installation name/version.

```
(current-pkg-lookup-version) → string?
(current-pkg-lookup-version s) → void?
  s : string?
```

Parameter that determines the relevant Racket version for extracting package information from a catalog.

Added in version 6.0.1.7 of package `base`.

```
(current-pkg-error) → procedure?
(current-pkg-error err) → void?
  err : procedure?
```

A parameter whose value is used to report errors that are normally intended for end uses. The arguments to the procedure are the same as for `error`, except that an initial symbol argument is omitted.

The default value uses `error` with `'pkg` as the first argument. The `raco pkg` command sets this parameter to use `raise-user-error` with the sub-command name as its first argument.

```
(current-pkg-catalogs) → (or/c #f (listof url?))
(current-pkg-catalogs catalogs) → void?
  catalogs : (or/c #f (listof url?))
```

A parameter that determines the package catalogs that are consulted to resolve a package name. If the parameter's value is `#f`, then the result of `pkg-config-catalogs` is used.

```
(pkg-config-catalogs) → (listof string?)
```

Returns a list of URL strings for the user's configured package catalogs.

```
(current-pkg-download-cache-dir)
 → (or/c #f (and/c path-string? complete-path?))
(current-pkg-download-cache-dir dir) → void?
  dir : (or/c #f (and/c path-string? complete-path?))
(current-pkg-download-cache-max-files) → (or/c #f real?)
(current-pkg-download-cache-max-files max-files) → void?
  max-files : (or/c #f real?)
(current-pkg-download-cache-max-bytes) → (or/c #f real?)
(current-pkg-download-cache-max-bytes max-bytes) → void?
  max-bytes : (or/c #f real?)
```

Parameters that determine the download cache location and limits. If a parameter's value is `#f`, then the user's configuration is used.

```
(pkg-directory name) → (or/c path-string? #f)
  name : string?
```

Returns the directory that holds the installation of the installed (in any scope) package *name*, or `#f` if no such package is installed.

```
(default-pkg-scope) → (or/c 'installation 'user
                            (and/c path? complete-path?))
```

Returns the user's configured default package scope.

```
(installed-pkg-names #:scope scope) → (listof string?)
  scope : (or/c #f 'installation 'user
                (and/c path? complete-path?))
```

Returns a list of installed package names for the given package scope, where `#f` indicates the user's default package scope.

```
(installed-pkg-table #:scope scope)
 → (hash/c string? pkg-info?)
  scope : (or/c #f 'installation 'user
                (and/c path? complete-path?))
```

Returns a hash table of installed packages for the given package scope, where `#f` indicates the user's default package scope.

```
(pkg-desc? v) → boolean?
  v : any/c
```

```
(pkg-desc source type name checksum auto?) → pkg-desc?
  source : string?
  type : (or/c #f 'file 'dir 'link 'static-link
               'file-url 'dir-url 'github 'name)
  name : (or/c string? #f)
  checksum : (or/c string? #f)
  auto? : boolean?
```

A `pkg-desc` value describes a package source plus details of its intended interpretation, where the *auto?* field indicates that the package is should be treated as installed automatically for a dependency.

```
(pkg-stage  desc
            [#:checksum checksum
             #:in-place? in-place?
             #:namespace namespace
             #:strip strip
             #:force-strip? force-string?
             #:use-cache? use-cache?
             #:quiet? quiet?])
    string?
    path?
  → (or/c #f string?)
    boolean?
    (listof module-path?)
  desc : pkg-desc?
  checksum : (or/c #f string?) = #f
  in-place? : boolean? = #f
  namespace : namespace? = (make-base-namespace)
  strip : (or/c #f 'source 'binary 'binary-lib) = #f
  force-string? : boolean? = #f
  use-cache? : boolean? = #f
  quiet? : boolean? = #t
```

Locates the implementation of the package specified by *desc* and downloads and unpacks it to a temporary directory (as needed).

If *desc* refers to an existing directory and *in-place?* is true, then the directory is used in place.

The *namespace* argument is passed along to `get-info/full` when the package's `"info.rkt"` is loaded.

If *strip* is not `#f`, then files and directories are removed from the prepared directory the same as when creating the corresponding kind of package. A directory that is staged in-place cannot be stripped. If `force-strip?` is true, then a consistency check (intended to avoid

stripping a source package as binary, for example) is skipped.

If *use-cache?* is true, then a local cache is consulted before downloading a particular package with a particular checksum. Note that the default for *use-cache?* is `#f`, while the default is `#t` for other functions that accept `#:use-cache?`.

The result is the package name, the directory containing the unpacked package content, the checksum (if any) for the unpacked package, whether the directory should be removed after the package content is no longer needed, and a list of module paths provided by the package.

```
(pkg-config  set?
             keys/vals
            [#:from-command-line? from-command-line?]) → void?
  set? : boolean?
  keys/vals : list?
  from-command-line? : boolean? = #f
```

Implements `pkg-config-command`.

If *from-command-line?* is true, error messages may suggest specific command-line flags for `raco pkg config`.

The package lock must be held (allowing writes if *set?* is true); see `with-pkg-lock`.

```
(pkg-create  format
             dir
             #:source source
             #:mode mode
             #:dest dest-dir
            [#:quiet? quiet?
             #:from-command-line? from-command-line?]) → void?
  format : (or/c 'zip 'tgz 'plt 'MANIFEST)
  dir : path-string?
  source : (or/c 'dir 'name)
  mode : (or/c 'as-is 'source 'binary 'binary-lib 'built)
  dest-dir : (or/c (and/c path-string? complete-path?) #f)
  quiet? : boolean? = #f
  from-command-line? : boolean? = #f
```

Implements `pkg-create-command`.

Unless *quiet?* is true, information about the output is reported to the current output port. If *from-command-line?* is true, error messages may suggest specific command-line flags for `raco pkg create`.

```
(pkg-install  descs
              [#:dep-behavior dep-behavior
               #:update-deps? update-deps?
               #:force? force?
               #:ignore-checksums? ignore-checksums?
               #:strict-doc-conflicts? strict-doc-conflicts?
               #:use-cache? use-cache?
               #:quiet? boolean?
               #:from-command-line? from-command-line?
               #:strip strip
               #:force-strip? force-string?
               #:link-dirs? link-dirs?])
    (or/c 'skip
          #f
→         (listof (or/c path-string?
                        (non-empty-listof path-string?))))
  descs : (listof pkg-desc?)
  dep-behavior : (or/c #f 'fail 'force 'search-ask 'search-auto)
               = #f
  update-deps? : boolean? = #f
  force? : boolean? = #f
  ignore-checksums? : boolean? = #f
  strict-doc-conflicts? : boolean? = #f
  use-cache? : boolean? = #t
  boolean? : quiet? = #f
  from-command-line? : boolean? = #f
  strip : (or/c #f 'source 'binary 'binary-lib) = #f
  force-string? : boolean? = #f
  link-dirs? : boolean? = #f
```

Implements pkg-install-command. The result indicates which collections should be setup via `raco setup`: 'skip means that no setup is needed, #f means all, and a list means only the indicated collections.

The *link-dirs?* argument determines whether package sources inferred to be directory paths should be treated as links or copied (like other package sources). Note that the default is #f, unlike the default built into pkg-install-command.

Status information and debugging details are mostly reported to a logger named 'pkg, but information that is especially relevant to a user (such as a download action) is reported to the current output port, unless quiet? is true.

If *from-command-line?* is true, error messages may suggest specific command-line flags for `raco pkg install`.

The package lock must be held; see `with-pkg-lock`.

```
(pkg-update  names
             [#:all? all?
              #:dep-behavior dep-behavior
              #:update-deps? update-deps?
              #:force? force?
              #:ignore-checksums? ignore-checksums?
              #:strict-doc-conflicts? strict-doc-conflicts?
              #:use-cache? use-cache?
              #:quiet? boolean?
              #:from-command-line? from-command-line?
              #:strip strip
              #:force-strip? force-string?
              #:link-dirs? link-dirs?])
    (or/c 'skip
          #f
→         (listof (or/c path-string?
                        (non-empty-listof path-string?))))
  names : (listof (or/c string? pkg-desc?))
  all? : boolean? = #f
  dep-behavior : (or/c #f 'fail 'force 'search-ask 'search-auto)
               = #f
  update-deps? : boolean? = #f
  force? : boolean? = #f
  ignore-checksums? : boolean? = #f
  strict-doc-conflicts? : boolean? = #f
  use-cache? : quiet? = #t
  boolean? : quiet? = #f
  from-command-line? : boolean? = #f
  strip : (or/c #f 'source 'binary 'binary-lib) = #f
  force-string? : boolean? = #f
  link-dirs? : boolean? = #f
```

Implements `pkg-update-command`. The result is the same as for `pkg-install`.

A string in *names* refers to an installed package that should be checked for updates. A `pkg-desc` in *names* indicates a package source that should replace the current installation.

If *from-command-line?* is true, error messages may suggest specific command-line flags for `raco pkg update`.

The package lock must be held; see `with-pkg-lock`.

```
(pkg-remove  names
             [#:demote? demote?
              #:auto? auto?
              #:force? force?
              #:quiet? boolean?
              #:from-command-line? from-command-line?])
    (or/c 'skip
          #f
→         (listof (or/c path-string?
                          (non-empty-listof path-string?))))
  names : (listof string?)
  demote? : boolean? = #f
  auto? : boolean? = #f
  force? : boolean? = #f
  boolean? : quiet? = #f
  from-command-line? : boolean? = #f
```

Implements `pkg-remove-command`. The result is the same as for `pkg-install`, indicating collects that should be setup via `raco setup`.

If *from-command-line?* is true, error messages may suggest specific command-line flags for `raco pkg remove`.

The package lock must be held; see `with-pkg-lock`.

```
(pkg-show indent [#:directory show-dir?]) → void?
  indent : string?
  show-dir? : boolean? = #f
```

Implements `pkg-show-command` for a single package scope, printing to the current output port. See also `installed-pkg-names` and `installed-pkg-table`.

The package lock must be held to allow reads; see `with-pkg-lock/read-only`.

```
(pkg-migrate  from-version
              [#:dep-behavior dep-behavior
               #:force? force?
               #:use-cache? use-cache?
               #:ignore-checksums? ignore-checksums?
               #:strict-doc-conflicts? strict-doc-conflicts?
               #:quiet? boolean?
               #:from-command-line? from-command-line?
               #:strip strip
               #:force-strip? force-string?])
```

```
   (or/c 'skip
         #f
→        (listof (or/c path-string?
                       (non-empty-listof path-string?))))
 from-version : string?
 dep-behavior : (or/c #f 'fail 'force 'search-ask 'search-auto)
              = #f
 force? : boolean? = #f
 use-cache? : boolean? = #t
 ignore-checksums? : boolean? = #f
 strict-doc-conflicts? : boolean? = #f
 boolean? : quiet? = #f
 from-command-line? : boolean? = #f
 strip : (or/c #f 'source 'binary 'binary-lib) = #f
 force-string? : boolean? = #f
```

Implements pkg-migrate-command. The result is the same as for pkg-install.

If *from-command-line?* is true, error messages may suggest specific command-line flags for `raco pkg migrate`.

The package lock must be held; see `with-pkg-lock`.

```
(pkg-catalog-show  names
                  [#:all? all?
                   #:only-names? only-names?
                   #:modules? modules?])      → void?
 names : (listof string?)
 all? : boolean? = #f
 only-names? : boolean? = #f
 modules? : boolean? = #f
```

Implements pkg-catalog-show-command. If *all?* is true, then *names* should be empty.

The current-pkg-lookup-version parameter determines the version included in the catalog query.

Changed in version 6.0.1.7 of package `base`: Use current-pkg-lookup-version instead of current-pkg-scope-version.

```
(pkg-catalog-copy  sources
                   dest
                  [#:from-config? from-config?
                   #:merge? merge?
                   #:force? force?
                   #:override? override?
                   #:relative-sources? relative-sources?])
```

```
→ void?
 sources : (listof path-string?)
 dest : path-string?
 from-config? : boolean? = #f
 merge? : boolean? = #f
 force? : boolean? = #f
 override? : boolean? = #f
 relative-sources? : boolean? = #f
```

Implements pkg-catalog-copy-command.

The current-pkg-lookup-version parameter determines the version for extracting existing catalog information.

Changed in version 6.0.1.7 of package base: Use current-pkg-lookup-version instead of current-pkg-scope-version.

```
(pkg-catalog-archive
 dest-dir
 sources
 [#:from-config? from-config?
  #:state-catalog state-catalog
  #:relative-sources? relative-sources?
  #:quiet? quiet?
  #:package-exn-handler package-exn-handler])
→ void?
 dest-dir : path-string?
 sources : (listof path-string?)
 from-config? : boolean? = #f
 state-catalog : (or/c #f path-string?) = #f
 relative-sources? : boolean? = #f
 quiet? : boolean? = #f
 package-exn-handler : (string? exn:fail? . -> . any)
                     = (lambda (pkg-name exn) (raise exn))
```

Implements pkg-catalog-archive-command.

The *package-exn-handler* argument handles any exception that is raised while trying to archive an individual package; the first argument is the package name, and the second is the exception. The default re-raises the exception, which aborts the archiving process, while a function that logs the exception message and returns would allow archiving to continue for other packages.

The current-pkg-lookup-version parameter determines the version for extracting existing catalog information.

Added in version 6.0.1.7 of package base. Changed in version 6.0.1.13: Added the #:package-exn-handler argument.

```
(pkg-archive-pkgs  dest-dir
                   pkgs
                  [#:include-deps? include-deps?
                   #:exclude exclude
                   #:relative-sources? relative-sources?
                   #:quiet? quiet?
                   #:package-exn-handler package-exn-handler])
 → void?
  dest-dir : path-string?
  pkgs : (listof path-string?)
  include-deps? : boolean? = #f
  exclude : (listof string?) = null
  relative-sources? : boolean? = #f
  quiet? : boolean? = #f
  package-exn-handler : (string? exn:fail? . -> . any)
                      = (lambda (pkg-name exn) (raise exn))
```

Implements pkg-archive-command.

The *package-exn-handler* argument handles any exception that is raised while trying to archive an individual package; the first argument is the package name, and the second is the exception. The default re-raises the exception, which aborts the archiving process, while a function that logs the exception message and returns would allow archiving to continue for other packages.

Added in version 6.1.0.8 of package base.

```
(pkg-catalog-update-local
 [#:catalogs catalogs
  #:catalog-file catalog-file
  #:quiet? quiet?
  #:set-catalogs? set-catalogs?
  #:consult-packages? consult-packages?])
 → void?
  catalogs : (listof string?) = (pkg-config-catalogs)
  catalog-file : path-string? = (current-pkg-catalog-file)
  quiet? : boolean? = #f
  set-catalogs? : boolean? = #t
  consult-packages? : boolean? = #f
```

Consults the package catalogs specified by *catalogs* (in the same way as pkg-catalog-copy) and the user's configured package servers (if *consult-packages?* is true) to populate the database *catalog-file* with information about available packages and the modules that they implement.

The local catalog *catalog-file* records the set of source catalogs, including *catalogs*, from which its information is drawn. If *set-catalogs?* is true (which is the default), then *catalogs* is recorded as the set of sources, and information from any other catalog is discarded. If *set-catalogs?* is `#f`, then *catalogs* must be a subset of the source catalogs already recorded in *catalog-file*.

Changed in version 6.0.1.6 of package `base`: Added `#:catalogs` and `#:set-catalogs?` arguments.

```
(pkg-catalog-suggestions-for-module
 module-path
 [#:catalog-file catalog-file])
→ (listof string?)
 module-path : module-path?
 catalog-file : path-string? = ....
```

Consults *catalog-file* and returns a list of available packages that provide the module specified by *module-path*.

The default *catalog-file* is `(current-pkg-catalog-file)` if that file exists, otherwise a file in the racket installation is tried.

```
(get-all-pkg-scopes)
 → (listof (or/c 'installation 'user path?))
```

Obtains a list of all the currently-available package scopes.

Added in version 6.1.0.5 of package `base`.

```
(get-all-pkg-names-from-catalogs) → (listof string?)
```

Consults package catalogs to obtain a list of available package names.

```
(get-all-pkg-details-from-catalogs)
 → (hash/c string? (hash/c symbol? any/c))
```

Consults package catalogs to obtain a hash table of available package names mapped to details about the package. Details for a particular package are provided by a hash table that maps symbols such as `'source`, `'checksum`, and `'author`.

```
(get-pkg-details-from-catalogs name)
 → (or/c #f (hash/c symbol? any/c))
  name : string?
```

Consults package catalogs to obtain information for a single package name, returning `#f` if the package name has no resolution. Details for the package are provided in the same form as from `get-all-pkg-details-from-catalogs`.

```
(pkg-single-collection  dir
                        [#:name name
                         #:namespace namespace])
 → (or/c #f string?)
  dir : path-string?
  name : string? = (elem "... from "(racket dir)" ...")
  namespace : namespace? = (make-base-namespapce)
```

Returns a string for a collection name if *dir* represents a single-collection package, or returns `#f` if *dir* represents a multi-collection package.

For some single-collection packages, the package's single collection is the package name; if the package name is different from the directory name, supply *name*.

Determining a single-collection package's collection name may require loading an `"info.rkt"` file, in which case *namespace* is passed on to `get-info/full`.

```
(get-pkg-content  desc
                  [#:extract-info extract-proc])
   (or/c #f string?)
 → (listof module-path?)
   any/c
  desc : pkg-desc?
                   ((or/c #f
  extract-proc :          ((symbol?) ((-> any)) . ->* . any))
                    . -> . any)
                = (lambda (get-pkg-info) ...)
```

Gets information about the content of the package specified by *desc*. The information is determined inspecting the package—resolving a package name, downloading, and unpacking into a temporary directory as necessary.

The results are as follows:

- The checksum, if any, for the downloaded package.
- A list of module paths that are provided by the package. Each module path is normalized in the sense of `collapse-module-path`.
- Information extracted from the package's metadata. By default, this information is the package's dependencies, but in general it is the result of *extract-proc*, which receives an information-getting function (or `#f`) as returned by `get-info`.

```
(extract-pkg-dependencies  info
                           [#:build-deps? build-deps?
                            #:filter? filter?
                            #:versions? versions?])
 → (listof (or/c string? (cons/c string? list?)))
  info : (symbol? (-> any/c) . -> . any/c)
  build-deps? : boolean? = #f
  filter? : boolean? = #f
  versions? : boolean? = #f
```

Returns packages dependencies reported by the *info* procedure (normally produced by `get-info`).

If *build-deps?* is true, then the result includes both run-time dependencies and build-time dependencies.

If *filter?* is true, then platform-specific dependencies are removed from the result list when they do not apply to the current platform, and other information is stripped so that the result list is always a list of either strings (when *versions?* is true) or a two-element list containing a string and a version (when *versions?* is `#f`).

Changed in version 6.0.1.6 of package `base`: Added the `#:versions?` argument.

```
(pkg-directory->module-paths  dir
                              pkg-name
                             [#:namespace namespace])
 → (listof module-path?)
  dir : path-string?
  pkg-name : string
  namespace : namespace? = (make-base-namespace)
```

Returns a list of module paths (normalized in the sense of `collapse-module-path`) that are provided by the package represented by *dir* and named *pkg-name*.

```
 (pkg-directory->additional-installs
   dir
   pkg-name
  [#:namespace namespace
   #:system-type sys-type
   #:system-library-subpath sys-lib-subpath])
 → (listof (cons/c symbol? string?))
  dir : path-string?
  pkg-name : string
  namespace : namespace? = (make-base-namespace)
  sys-type : (or/c #f symbol?) = (system-type)
  sys-lib-subpath : (or/c #f path?)
                  = (system-library-subpath #f)
```

Returns a list of pairs for items that are installed by the package represented by *dir* and named *pkg-name*. Installed items can include documentation, executables, foreign libraries, other shared files, and man pages—all as specified by "info.rkt" files. The symbol for each item gives it a category, such as 'doc or 'exe, and the string part is a normalized name, such as the destination name for a document or a case-folded executable name without a file suffix.

The *sys-type* and *sys-lib-subpath* arguments are used in the same way as for matching-platform? to determine platform-specific installations as determined by install-platform definitions in "info.rkt" files.

Added in version 6.0.1.13 of package base.

6.3 Package Paths and Database

(require pkg/path) package: base

The pkg/path library provides utilities for working with package paths and installed-package databases.

```
(struct pkg-info (orig-pkg checksum auto?)
    #:prefab)
            (or/c (list/c 'catalog string?)
  orig-pkg :      (list/c 'url string?)
                  (list/c 'link string?)
                  (list/c 'static-link string?))
  checksum : (or/c #f string?)
  auto? : boolean?
```

A structure type that is used to report installed-package information.

```
(struct sc-pkg-info pkg-info ())
```

A structure subtype that represents a package that is installed as single-collection.

```
(struct pkg-info/alt pkg-info (dir-name))
  dir-name : string?
(struct sc-pkg-info/alt sc-pkg-info (dir-name))
  dir-name : string?
```

Structure subtypes that are used when the installation directory for a package does not match the package name, but is instead dir-name. The directory name always includes a + (which is disallowed in a package name).

```
(path->pkg path #:cache cache) → (or/c string? #f)
  path : path-string?
  cache : (or/c #f (and/c hash? (not/c immutable?)))
```

Returns the installed package containing *path*, if any.

If *cache* is not #f, then it is consulted and modified to cache installed-package information across calls to path->pkg (with the assumption that the set of installed packages does not change across calls that receive the same *cache*).

```
(path->pkg+subpath path #:cache cache)
 → (or/c string? #f) (or/c path? 'same #f)
  path : path-string?
  cache : (or/c #f (and/c hash? (not/c immutable?)))
```

Like path->pkg, but returns a second value that represents the remainder of *path* within the package's directory.

```
(path->pkg+subpath+collect path
                           #:cache cache)
   (or/c string? #f)
 → (or/c path? 'same #f)
   (or/c string? #f)
  path : path-string?
  cache : (or/c #f (and/c hash? (not/c immutable?)))
```

Like path->pkg+subpath, but returns a third value for a collection name if the package is a single-collection package, #f otherwise.

```
(get-pkgs-dir scope [user-version]) → path?
  scope : (or/c 'installation 'user 'shared
                (and/c path? complete-path?))
  user-version : string? = (version)
```

Returns the path of the directory that holds installed packages in the given scope. The *user-version* argument is used to generate the result for 'user scope.

```
(read-pkgs-db scope) → (hash/c string? pkg-info?)
  scope : (or/c 'installation 'user 'shared
                (and/c path? complete-path?))
```

Returns a hash table representing the currently installed packages in the specified scope.

```
(read-pkg-file-hash path) → hash?
  path : path?
```

Reads a hash table from *path*, logging any errors and returning an empty hash table if *path* does not exist or if an error is encountered.

6.4 Package Source Parsing

```
(require pkg/name)
```
package: base

The pkg/name library provides functions for parsing and normalizing a package source, especially for extracting a package name.

```
(package-source-format? v) → boolean?
  v : any/c
```

Returns #t if *v* is 'name , 'file, 'dir, 'github, 'file-url, 'dir-url, 'link, or 'static-link, and returns #f otherwise.

The 'link and 'static-link formats are the same as 'dir in terms of parsing, but they are treated differently for tasks such as package installation.

```
(package-source->name source [type]) → (or/c #f string?)
  source : string?
  type : (or/c package-source-format? #f) = #f
```

Extracts the package name from a package source, where the package source type is inferred if *type* is #f. If a valid name cannot be inferred, the result is #f.

```
(package-source->name+type
 source
 [type
  #:complain complain-proc
  #:must-infer-name? must-infer-name?]
  #:link-dirs? link-dir?)
  (or/c #f string?)
→ (or/c package-source-format? #f)
 source : string?
 type : (or/c package-source-format? #f) = #f
 complain-proc : (string? string? . -> . any) = void
 must-infer-name? : boolean? = #f
 link-dir? : boolean?
```

Like package-source->name, but also returns the type of the source (which is useful when the type is inferred). If the source is not well-formed, the second result can be #f.

The *complain-proc* function is applied when *source* is ill-formed. The arguments to *complain-proc* are *source* and an error message.

If *must-infer-name?* is true, then *complain-proc* is called if a valid name cannot be inferred from *source*.

If link-dirs? is true, then a directory path is reported as type 'link instead of 'dir.

```
(package-source->path source [type]) → path?
  source : string?
  type : (or/c #f 'file 'dir 'link 'static-link) = #f
```

Converts a file or directory package source to a filesystem path.

The package-source->path function is different from string->path in the case that *source* starts with file://. Also, if *type* is 'dir, 'link, or 'static-link, then path->directory-path is used to ensure that the result path refers to a directory.

Added in version 10.0.1.11 of package base.

6.5 Package Catalog Database

```
(require pkg/db)
```
package: base

The pkg/db library provides tools for storing and retrieving package catalog information in a local database.

The functions provided by pkg/db do not actually manage packages; they do not change or consult the local database of installed modules in any package scope. The functions provided by pkg/db simply reflect a local copy of the information that a package catalog and individual package might provide (but with no guarantee of being in sync with an actual package catalog or package).

The database is implemented as an SQLite database with its own locking, so no additional locks are needed for database access, but beware of concurrent database changes that could break your program logic.

```
(struct pkg (name catalog author source checksum desc)
    #:extra-constructor-name make-pkg
    #:transparent)
  name : string?
  catalog : string?
  author : string?
  source : string?
  checksum : string?
  desc : string?
```

Represents a package implementation in the database. The name (package name) and catalog (package catalog, normally a URL) fields are always nonempty strings. Otherwise,

unknown fields are represented by empty strings.

```
(current-pkg-catalog-file) → path-string?
(current-pkg-catalog-file file) → void?
  file : path-string?
```

A parameter that determines the file path used to hold the SQLite database. The default value is in the user's add-on directory as determined by `(find-system-path 'addon-dir)` and within a subdirectory determined by `get-installation-name`.

```
(get-catalogs) → (listof string?)
(set-catalogs! catalogs) → void?
  catalogs : (listof string?)
```

Returns or sets the list of strings for all package catalog represented in the database. (Within the database, each package catalog gets its own identifying number.) The order of indices in the list represents a search order.

The `set-catalogs!` function removes information for any other package catalogs from the database.

```
(get-pkgs [#:name name #:catalog catalog]) → (listof pkg?)
  name : (or/c #f string?) = #f
  catalog : (or/c #f string?) = #f
```

Gets a list of package descriptions. If *name* or *catalog* is not `#f` (or if both are not `#f`), then the result includes only matching packages.

The result list is ordered by precedence of the package catalog.

```
(set-pkgs! catalog
           pkgs
           #:clear-other-checksums? clear-other-checksums?)
 → void?
  catalog : string?
  pkgs : (listof (or/c string? pkg?))
  clear-other-checksums? : #t
```

Sets the list of all packages that are recognized by the package catalog *catalog*.

Information about any other package for *catalog* is removed from the database. If a string is provided for *pkgs*, it is treated as a package name; if additional information is already recorded in the database for the package name, then the additional information is preserved.

If *clear-other-checksums?* is true, then for each element of *pkgs* that has a given checksum other than `""`, any information in the database specific to another checksum (such as a list of module paths) is removed from the database.

```
(set-pkg! name
          catalog
          author
          source
          checksum
          desc
          #:clear-other-checksums? clear-other-checksums?)
 → void?
  name : string?
  catalog : string?
  author : string?
  source : string?
  checksum : string?
  desc : string?
  clear-other-checksums? : (not (equal? checksum ""))
```

Sets the information for a specific package *name* as recognized by the package catalog *catalog*.

If *clear-other-checksums?* is true, then information (such as a list of module paths) is removed from the database when it is specific to a checksum other than *checksum*.

```
(get-pkg-tags name catalog) → (listof string?)
  name : string?
  catalog : string?
(set-pkg-tags! name catalog module-paths) → void?
  name : string?
  catalog : string?
  module-paths : (listof string?)
```

Gets or sets a list of tags for the package *name* as recognized by the package catalog *catalog*.

```
(get-pkg-dependencies name catalog checksum) → (listof list?)
  name : string?
  catalog : string?
  checksum : string?
(set-pkg-dependencies! name
                       catalog
                       checksum
                       dependencies) → void?
  name : string?
  catalog : string?
  checksum : string?
  dependencies : (listof any/c)
```

Gets or sets a list of dependencies for the package *name* as recognized by the package catalog *catalog* and for a specific checksum.

The list of dependencies must have the shape described for a `deps` `"info.rkt"` field as described in §4 “Package Metadata”. The result from `get-pkg-dependencies` is normalized: each dependency is represented by a list, a version in a dependency is always preceded by `'#:version`, and if both version and platform specification are included, `'#:version` appears before `'#:platform`.

```
(get-pkg-modules name catalog checksum) → (listof module-path?)
  name : string?
  catalog : string?
  checksum : string?
(set-pkg-modules! name
                  catalog
                  checksum
                  module-paths) → void?
  name : string?
  catalog : string?
  checksum : string?
  module-paths : (listof module-path?)
```

Gets or sets a list of module paths that are provided for the package *name* as recognized by the package catalog *catalog* and for a specific checksum. The module paths should be normalized in the sense of `collapse-module-path`.

```
(get-module-pkgs module-path) → (listof pkg?)
  module-path : module-path?
```

Reports a list of packages that implement the given *module-path*, which should be normalized in the sense of `collapse-module-path`.

```
(get-pkgs-without-modules [#:catalog catalog]) → (listof pkg?)
  catalog : (or/c #f string?) = #f
```

Returns a list of packages (optionally constrained to *catalog*) for which the database has no modules recorded.

Each resulting `pkg` has its `name`, `catalog`, and `checksum` field set, but other fields may be `""`.

7 Package Catalog Protocol

A package catalog is specified by a URL in one of three forms:

- `http://` or `https://` — a remote URL
- `file://` ending with `.sqlite` — a local SQLite database
- `file://` without `.sqlite` — a local directory

7.1 Remote and Directory Catalogs

In the case of a remote URL or a local directory naming a package catalog, the URL/path is extended as follows to obtain information about packages:

- `pkg` and ⟨*package*⟩ path elements, where ⟨*package*⟩ is a package name, plus a `version=`⟨*version*⟩ query (where ⟨*version*⟩ is a Racket version number) in the case of a remote URL.

  This URL/path form is use to obtain information about ⟨*package*⟩. An HTTP request for a remote URL should respond with a `read`-able hash table, as described below. A path in a local directory formed by adding `"pkg"` and ⟨*package*⟩ should refer to a file that similarly contains a `read`-able hash table.

  The hash table should supply the following keys:

  - `'source` (required) — a package source string, typically a remote URL. If this source is a relative URL, then it is treated as relative to the catalog.

    Changed in version 6.0.1.7: Added relative-path support to clients of a catalog server.
  - `'checksum` (requires) — a string for a checksum.
  - `'name` — a string that is the same as ⟨*package*⟩.
  - `'author` — a string for the author of the package, normally an e-mail address.
  - `'description` — a string describing the package.
  - `'tags` — a list of strings that describe the package's categorization.
  - `'dependencies` — a list of dependencies for the package, in the same shape as a `deps` `"info.rkt"` field as described in §4 "Package Metadata".
  - `'modules` — a list of module paths for modules that are provided by the package; each module path should be normalized in the sense of `collapse-module-path`.
  - `'versions` (optional) — a hash table mapping version strings and `'default` to hash tables, where each version-specific hash table provides mappings to override the ones in the main hash table, and `'default` applies to any version not otherwise mapped.

Clients of a remote catalog may request information for a specific version, but they should also check for a `'versions` entry in a catalog response, in case a catalog with version-specific mappings is implemented as a directory or by a file-serving HTTP server. A `'default` mapping, meanwhile, allows the main hash table to provide information that is suitable for clients at version 5.3.6 and earlier (which do not check for `'versions`).

- `pkgs` path element: Obtains a list of package names that are mapped by the package catalog. An HTTP request for a remote URL should respond with a `read`-able list of strings. A path in a local directory formed by adding `"pkgs"` should refer to a file that similarly contains a `read`-able list of strings.

  This URL/path form is used by `raco pkg catalog-copy` and tools that allow a user to browse an catalog.

  In the case of a local directory, if no `"pkgs"` file is available, a list is created by listing all files in the `"pkg"` directory.

- `pkgs-all` path element: Obtains a hash table mapping package names to package details. An HTTP request for a remote URL should respond with a `read`-able hash table mapping strings to hash tables. A path in a local directory formed by adding `"pkgs-all"` should refer to a file that similarly contains a `read`-able hash table.

  This URL/path form is a shortcut for a `pkgs` URL/path form combined with a `pkgs/`⟨*package*⟩ query for each package.

  In the case of a local directory, if no `"pkgs-all"` file is available, a list is created from files in the `"pkg"` directory.

Note that a local directory served as files through an HTTP server works as a remote URL, as long as the `"pkgs"` and `"pkgs-all"` files are present.

The source for the PLT-hosted package catalog is in the `(collection-file-path "pkg-catalog" "meta")` directory of the full Racket distribution.

7.2 SQLite Catalogs

A SQLite database package catalog is meant to be constructed and queries using the `pkg/db` library, but the database can be constructed in any way as long as it contains the following tables:

- A `catalog` table with the format

```
(id SMALLINT,
 url TEXT,
 pos SMALLINT)
```

Normally, the only row in this table is `(0, "local", 0)`, but a database that records the content of a set of other catalogs can also be used as an catalog, in which case each row represents an catalog; the `id` field is a unique identifier for each catalog, the `url` field is the catalog's URL, and the `pos` column orders the catalog relative to others (where a lower `pos` takes precedence).

- A `pkg` table with the format

  ```
  (name TEXT,
   catalog SMALLINT,
   author TEXT,
   source TEXT,
   checksum TEXT,
   desc TEXT)
  ```

  The `catalog` field is normally `0`; in the case that the database reflects multiple other catalogs, the `catalog` field indicates the package entry's source catalog.

  The `pkg` and `catalog` fields together determine a unique row in the table.

- A `tags` table with the form

  ```
  (pkg TEXT,
   catalog SMALLINT,
   tag TEXT)
  ```

  where the `pkg` and `catalog` combination identifies a unique row in `pkg`.

- A `modules` table with the form

  ```
  (name TEXT,
   pkg TEXT,
   catalog SMALLINT,
   checksum TEXT)
  ```

  where the `pkg` and `catalog` combination identifies a unique row in `pkg`, and `name` is a printed module path.

  This table is not currently used by any `raco pkg` command, but it can be used to suggest package installations to provide a particular library.

- A `dependencies` table with the form

  ```
  (onpkg TEXT,
   onversion TEXT,
   onplatform TEXT,
   pkg TEXT,
   catalog SMALLINT,
   checksum TEXT)
  ```

  where the `pkg` and `catalog` combination identifies a unique row in `pkg`, and `onpkg`, `onversion`, and `onplatform` represent the dependency; `onversion` or `onplatform` is an empty string if the dependency has no version or platform specification.

  This table is not currently used by any `raco pkg` command.

8 PLaneT Compatibility

PLT maintains a package catalog to serve packages that were developed using the original PLaneT package system. This compatibility catalog is at http://planet-compats.racket-lang.org/, which is included by default in the package-server search path.

Copies of PLaneT packages are automatically created by the server according to the following system: for all packages that are in the `4.x` PLaneT repository, the latest minor version of ⟨*user*⟩/⟨*package*⟩`.plt/`⟨*major-version*⟩ will be available as `planet-`⟨*user*⟩`-`⟨*package*⟩⟨*major-version*⟩. For example, `jaymccarthy/opencl.plt/1` minor version 2, will be available as `planet-jaymccarthy-opencl1`.

The contents of these copies is a single collection with the name `"`⟨*user*⟩/⟨*package*⟩⟨*major-version*⟩`"` with all the files from the original PLaneT package in it.

Each file has been transliterated to use direct Racket-style requires rather than PLaneT-style requires. For example, if any file contains `(planet jaymccarthy/opencl/module)`, then it is transliterated to `jaymccarthy/opencl1/module`. *This transliteration is purely syntactic and is trivial to confuse, but works for most packages, in practice.* Any transliterations that occurred are automatically added as dependencies for the compatibility package.

We do not intend to improve this compatibility system much more over time, because it is simply a stop-gap as developers port their PLaneT packages to the new system. Additionally, the existence of the compatibility server is not meant to imply that we will be removing PLaneT from existence in the near future.

9 FAQ

This section answers anticipated frequently asked questions about the package manager.

9.1 Are package installations versioned with respect to the Racket version?

Most Racket installations are configured to that installing a package installs it for a specific user and a specific version of Racket. That is, the package scope is user- and version-specific. More precisely, it is user-specific and installation-name-specific, where an installation name is typically a Racket version.

You can change the default package scope (for a particular Racket installation) with `raco pkg config -i --set default-scope installation`, in which case package operations apply for all users of a Racket installation. You can also use the `-i` or `--installation` flag with a specific `raco pkg` command, instead of changing the default scope for all uses of `raco pkg`. Note that an installation-wide package is not exactly version-specific, because the version of an installation can change if it corresponds to a source-code checkout that is periodically updated and rebuilt.

If you change the default package scope, you can use the `-u` or `--user` flag with a specific `raco pkg` command to perform the command with user-specific package scope.

9.2 Where and how are packages installed?

User-specific and Racket-version-specific packages are in `(find-user-pkgs-dir)`, and installation-wide packages are in `(find-pkgs-dir)`. They are linked as collections (for single-collection packages) or collection roots (for multi-collection packages) with `raco link`.

9.3 How are user-specific and installation-wide package scopes related for conflict checking?

User-specific packages are checked against installation-wide packages for package-name conflicts and provided-module conflicts. Installation-wide packages are checked against user-specific packages only for provided-module conflicts.

Beware that a conflict-free, installation-wide change by one user can create conflicts for a different user.

9.4 Do I need to change a package's version when I update a package with error fixes, etc.?

If you have new code for a package, then it should have a new checksum. When package updates are searched for, the checksum of the installed package is compared with the checksum of the source, if they are different, then the source is re-installed. This allows code changes to be distributed. You do not need to declare an update a version number, except to allow other package implementors to indicate a dependency on particular features (where a bug fix might be considered a feature, but it is not usually necessary to consider it that way).

9.5 How can I specify which version of a package I depend on if its interface has changed and I need an *old* version?

In such a situation, the author of the package has released a backwards incompatible edition of a package. The package manager provides no help to deal with this situation (other than, of course, not installing the "update"). Therefore, package authors should not make backwards incompatible changes to packages. Instead, they should release a new package with a new name. For example, package `libgtk` might become `libgtk2`. These packages should be designed to not conflict with each other, as well.

9.6 How can I fix my installation to a specific set of package implementations or checksums?

Packages are updated only when you run a tool such as `raco pkg update`, so packages are never updated implicitly. Furthermore, you can snapshot a set of package archives and install from those archives, instead of relying on package name resolution through a package catalog.

If you want to control the resolution of package names (including specific checksums) but not necessary keep a copy of all package code (assuming that old checksums remain available, such as through Github), you can create a snapshot of the package name to package source mapping by using `raco pkg catalog-copy`. For example,

```
raco pkg catalog-copy --from-config /home/joe/snapshot.sqlite
```

creates a snapshot `"/home/joe/snapshot.sqlite"` of the current package name resolution, and then

```
raco pkg config --set catalogs file:///home/joe/snapshot.sqlite
```

directs all package-name resolution to the snapshot. You can configure resolution for specific package names by editing the snapshot.

You can go even further with

```
raco pkg catalog-archive --from-config /home/joe/snapshot/
```

which not only takes a snapshot of the catalog, but also takes a snapshot of all package sources (so that you do not depend on the original sources).

9.7 How can I install a package without its documentation?

If package is available in the form of a built package, then you can use `raco pkg install --binary-lib` to strip source, tests, and documentation from a package before installing it.

Built packages are typically provided by a snapshot or release site (where a Racket distribution downloaded from the site is configured to consult the site for packages), at least for packages associated with the distribution. Beware that `http://pkgs.racket-lang.org/` generally refers to source packages, not built packages. In the near future, built variants of the `http://pkgs.racket-lang.org/` packages will be provided at `http://pkg-build.racket-lang.org/catalog/`.

Some packages have been split at the source level into separate library, test, and documentation packages. For example, `net-lib` provides modules such as `net/cookie` without documentation, while `net-doc` provides documentation and `net-test` provides tests. The `net` package depends on `net-lib` and `net-doc`, and it implies `net-lib`, so you can install `net` in a minimal Racket distribution to get the libraries with documentation (and lots of additional packages to support documentation), or install `net-lib` to get just the libraries.

If you develop a package that is especially widely used or is especially useful in a constrained installation environment, then splitting your package into `-lib`, `-doc`, and `-test` components may be worthwhile. Most likely, you should keep the packages together in a single source-code repository and use metedata such as `implies` and `update-implies` (see §4 “Package Metadata”) so that the packages are updated in sync.

9.8 Why is the package manager so different than PLaneT?

There are two fundamental differences between PLaneT and this package manager.

The first is that PLaneT uses “internal linking” whereas the current package manager uses “external linking.” For example, an individual module requires a PLaneT package directly in a require statement:

```
(require (planet game/tic-tac-toe/data/matrix))
```

whereas using the package manager, the module would simply require the module of interest:

```
(require data/matrix)
```

and would rely on the external system having the `tic-tac-toe` package installed.

This change is good because it makes the origin of modules more flexible—so that code can migrate in and out of the core, packages can easily be split up, combined, or taken over by other authors, etc.

This change is bad because it makes the meaning of your program dependent on the state of the system.

The second major difference is that PLaneT is committed to guaranteeing that packages that never conflict with one another, so that any number of major and minor versions of the same package can be installed and used simultaneously. The package manager does not share this commitment, so package authors and users must be mindful of potential conflicts and plan around them.

This change is good because it is simpler and lowers the burden of maintenance (provided most packages don't conflict.)

The change is bad because users must plan around potential conflicts.

In general, the goal of the package manager is to be a lower-level system, more like the package systems used by operating systems. The goals of PLaneT are not bad, but we believe they are needed infrequently and a system like PLaneT could be more easily built atop the package manager than the reverse.

In particular, our plans to mitigate the downsides of these changes are documented in §10.1 "Short Term".

10 Future Plans

10.1 Short Term

This section lists some short term plans for the package manager. These are important, but didn't block its release. The package manager will be considered out of beta when these are completed.

- The official catalog server will divide packages into three categories: ring-0, ring-1, and ring-2. The definitions for these categories are:
  - ring-2 — No restrictions.
  - ring-1 — Must not conflict any package in ring-1 or ring-0.
  - ring-0 — Must not conflict any package in ring-1 or ring-0. Must have documentation and tests. The author must be responsive about fixing regressions against changes in Racket, etc.

  These categories will be curated by PLT.

  Our goal is for all packages to be in ring-1, with ring-2 as a temporary place while the curators work with the authors of conflicting packages to determine how modules should be renamed for unity.

  However, before curation is complete, each package will be automatically placed in ring-2 or ring-1 depending on its conflicts, with preference being given to older packages. (For example, if a new package B conflicts with an old package A, then A will be in ring-1, but B will be in ring-2.) During curation, however, it is not necessarily the case that older packages have preference. (For example, `tic-tac-toe` should probably not provide `"data/matrix.rkt"`, but that could be spun off into another package used by both `tic-tac-toe` and `factory-optimize`.)

  In contrast, the ring-0 category will be a special category that authors may apply for. Admission requires a code audit and implies a "stamp of approval" from PLT. In the future, packages in this category will have more benefits, such as automatic regression testing on DrDr, testing during releases, provided binaries, and advertisement during installation.

  The PLaneT compatibility packages will also be included in the ring-1 category, automatically.

- In order to mitigate the costs of external linking vis a vis the inability to understand code in isolation, we will create exception printers that search for providers of modules on the configured package catalogs. For example, if a module requires `"data/matrix.rkt"`, and it is not available, then the catalog will be consulted to discover what packages provide it. *Only packages in ring-1 or ring-0 will be returned.* (This category restriction ensures that the package to install is unique.)

Users can configure their systems to then automatically install the package provided is has the appropriate category (i.e., some users may wish to automatically install ring-0 packages but not ring-1 packages, while others may not want to install any.)

This feature will be generalized across all package catalogs, so users could maintain their own category definitions with different policies.

10.2 Long Term

This section lists some long term plans for the package manager. Many of these require a lot of cross-Racket integration.

- The official catalog server is bare bones. It could conceivably do a lot more: keep track of more statistics, enable "social" interactions about packages, link to documentation, problem reports, licenses, etc. Some of this is easy and obvious, but the community's needs are unclear.
- It would be nice to encrypt information from the official package catalog with a public key shipped with Racket, and allow other catalogs to implement a similar security scheme.
- Packages in the ring-0 category should be tested on DrDr. This would require a way to communicate information about how they should be run to DrDr. This is currently done via the `"meta/props"` script for things in the core. We should generalize this script to a `"meta/props.d"` directory so that packages can install DrDr metadata to it.
- We hope that this package system will encourage more incremental improvements to pieces of Racket. In particular, it would be wonderful to have a very thorough `"data"` collection of different data-structures. However, our existing setup for Scribble would force each new data structue to have a different top-level documentation manual, rather than extending the documentation of the existing `"data"` collection. Similar issues will exist for the `"net"` and `"file"` collections. We should design a way to have such "documentation plugins" in Scribble and support similar "plugin" systems elsewhere in the code-base.
- The user interface could be improved, including integration with DrRacket and a GUI. For example, it would be good if DrRacket would poll for package updates periodically and if when it was first started it would display available, popular packages.

11 How Package Installation and Distribution Works

The package manager builds on three main pieces of infrastructure:

- Collection links files as supported by the Racket runtime system.

  Installation of a package installs collection links, so the package's collections can be found to compile and load modules that use the package's modules.

  If you use `raco link -l` to view installed links, you will see links that were put in place by the package system. Obviously, you should not directly modify those links.

- The `raco setup` tool for building installed collections, including their documentation.

  The `raco setup` tool drives `raco make` to compile Racket sources to bytecode form. Recompilation is determined by changes to file timestamps, SHA-1 hashes, and dependencies recorded in `"dep"` files.

  Since package installations are reflected as collection links, `raco setup` operations on collections implicitly handle packages. The `raco setup` tool is "aware" of packages to only a limited extent: it uses functions like `path->pkg` to print progress information in terms of packages, and it uses similar package-inspection functions to connect modules to package and check actual dependencies against declared package dependencies.

- The `racket-index` package, which extends `raco setup` to drive Scribble for collection-based documentation.

  The `racket-index` package implements the documentation-rendering analogue of `raco make`, detecting changes in documentation declarations and re-rendering documents as needed to pick up cross-reference changes. The `racket-index` package also implements special documents for the entry point to HTML-rendered documentation (i.e., the listing of all installed documentation), the HTML search page, the local-redirection page (which server-search links to locally installed files), and so on.

Each of the three levels accommodate the `user` and `installation` package scopes, where the details in each case often differ between the scopes. Generally, references in the `installation` scope must be implemented as relative, so that an in-place installing of Racket can be moved to a different location. References in the `user` scope, meanwhile, may refer directly to the installation at some level; most references are collection-relative or installation-relative, so package content can be built in `user` scope and then assembled into a built package or binary package for installation elsewhere.

11.1 Relative References

Functions like `path->collects-relative` and `path->main-collects-relative` are used to serialize paths into relative form, and then the paths can be deserialized with

functions like collects-relative->path and main-collects-relative->path. The make-path->relative-string function generalizes support for such serialization and deserialization relative to a given set of directories.

Dependencies in a raco make-generated ".dep" file use collection-relative paths whenever possible, and it should always be possible for dependencies within a collection. Similarly, cross-reference information for documentation uses collection-relative paths when possible.

In a collection links file, paths are relative to the link file's directory. Installation-wide links then work when an in-place installation is moved.

In cross-reference information for documentation that installation-wide, paths can be stored relative to the installation's "doc" directory. For documentation that is built in user scope, cross-reference information within the built document is recorded relative to the document's directory via the root-path initialization argument to render%; the cross-reference information can be unpacked to a different destination, where the use-time path is provided the #:root argument to load-xref and/or make-data+root structures.

11.2 Separate Documentation Rendering

Unlike module references, which must create no reference cycles, documentation can have reference cycles. Documentation also tends to be less compact than code, and while we attempt to minimize module dependencies in code, documentation should freely reference any other documentation that is relevant. Finally, documentation references are less static than module references; for example, a document references cons by referring to racket/base, and the documentation system must figure out which other document defines cons. A naive implementation of documentation rendering would load all documents to render any one document, which is prohibitively expensive in both time and space.

Scribble supports separate document rendering by marshaling and unmarshaling cross-reference information. The racket-index extension of raco setup stores a document's information in ".sxref" files. Some documents, such as the reference, export a large volume of cross-reference information, so raco setup breaks up a document's exported cross-reference information into multiple "out⟨*n*⟩.sxref" files. Information about "imported" cross-reference information—that is, the cross references that were used the last time a document was built—is kept in "in.sxref" files. Finally, to detect which "out⟨*n*⟩.sxref" files need to be loaded while building a document, a mapping of cross-reference keys to "out⟨*n*⟩.sxref" files is kept in a SQLite database, "docindex.sqlite". Lazy loading of "out⟨*n*⟩.sxref" files is implemented though the #:demand-source argument to load-xref, providing a function that consults "docindex.sqlite" to map a key to a cross-reference file.

Various kinds of paths within cross-reference files are stored with various relative-path conventions. The "docindex.sqlite" file in an installation can be moved unmodified with the installation. The "docindex.sqlite" file for user-scoped packages is non-portable (and

outside any package), while the "in.sxref" and "out⟨n⟩.sxref" files can be included as-is in a binary package or built package.

11.3 Cross-Document HTML References

The HTML generated for a Scribble document needs relative links. Unlike data that is unmarshaled by Racket code, however, there is no way to turn paths that are relative to various installation directories into paths that a browser understands—at least, not using only HTML. Generated HTML for documentation therefore relies on JavaScript to rewrite certain references, with a fallback path through a server to make documentation also work as plain HTML.

References within a single document are rendered as relative links in HTML. A reference from one document to another is rendered as a query to, say, http://docs.racket-lang.org/. However, every document also references "local-redirect.js" and (in the case of documentation for user-specific collections) "local-user-redirect.js". Those fragments of JavaScript dynamically rewrite query references to direct filesystem references—to installation-wide and user-specific targets, respectively—when local targets are available. When local targets are not available, the query link is left unmodified to go through a server.

The "local-redirect.js" and "local-user-redirect.js" files map documentation-directory names to specific paths. Most query references contain a documentation-directory name and a relative path within the directory, in which case the mapping from directory names to paths is sufficient. Indirect links, such as those created by (seclink #:indirect? #t ...), embed a cross-reference key, and so "local-redirect.js" and "local-user-redirect.js" must also embed a part of the cross-reference database. (This copy of the database is broken into multiple files, each of which is loaded on demand.) The "local-redirect.js" and "local-user-redirect.js" files are generated as part of the special "local-redirect" document that is implemented by the racket-index package.

The indirection through "local-redirect.js" and "local-user-redirect.js" reduces the problem of relative links to the problem of referencing those two files. They are referenced as absolute paths in a user-specific document build. To create a built package or binary package that includes documentation, each ".html" file must be modified to remove the absolute paths, and then each ".html" file must be modified again on installation to put the target installation's paths in path.

The racket-index package's extension of raco setup to build Scribble documentation puts these indirections in place using the set-external-tag-path method of render-mixin from scribble/html-render. The http://docs.racket-lang.org/ path is not hardwired, but instead based on the installation's configuration as reported by get-doc-search-url. That configuration, in turn, can be determined when building a Racket distribution; the main distributions from PLT set the URL to a version-specific site, so that searches work even after new Racket versions are released, while snapshots similarly set the

URL to a snapshot-specific site.

11.4 HTML Documentation Searching and Start Page

The `racket-index` package provides a special document to implement the initial page for installed HTML documentation. The document uses `"info.rkt"`-file `scribblings` flags to depend on all documents for their titles.

The `racket-index` package also provides a special document to implement searching. The search document uses JavaScript and a copy of the cross-reference database (similar to `"local-redirect.js"`) to implement interactive searching.

If any user-specific collections have been installed, then `racket-index` generates two copies of the start and search documents: one for the installation, and one specific to the user. The user pages are an extension of the installation pages. The user-specific search page reads the installation-wide search page's database, which both avoids duplication and allows the search to pick up any additions to the installation without requiring a rebuild of the user-specific search page. The user-specific start page, in contrast, must be rebuilt after any installation-wide additions to pick up the additions.

When DrRacket or `raco docs` opens documentation in a browser, it opens the user-specific start or search page, if it exists. If those pages are visited for any reason, browser local storage or (if local storage is not supported) a cookie is installed. The local-storage key or cookie is named "PLT_Root.⟨*version*⟩," it points to the location of the user-specific documentation. Thereafter, using the local value of cookie, searching in any documentation page or going to the "top" page goes to the user-specific page, even from an installation-wide page.