# Package Management in Racket (Beta)

Version 5.3.6

Jay McCarthy <jay@racket-lang.org>

August 9, 2013

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

# Contents

# 1 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 can be computed as the SHA1 (see `openssl/sha1`) of the package's source.
- a *version* — a string of the form ⟨*maj*⟩`.`⟨*min*⟩, ⟨*maj*⟩`.`⟨*min*⟩`.`⟨*sub*⟩, or ⟨*maj*⟩`.`⟨*min*⟩`.`⟨*sub*⟩`.`⟨*rel*⟩, where ⟨*maj*⟩, ⟨*min*⟩, ⟨*sub*⟩, and ⟨*rel*⟩ are all canonical decimal representations of natural numbers, ⟨*min*⟩ has no more than two digits, and ⟨*sub*⟩ and ⟨*rel*⟩ 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").

A *package source* identifies a package representation. Each package source type has a different way of storing the checksum. The valid package source types are:

- a local file path naming an archive – 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"`. For example, `"~/tic-tac-toe.zip"`'s checksum would

be inside `"~/tic-tac-toe.zip.CHECKSUM"`. The valid archive formats are (currently) `".zip"`, `".tar"`, `".tgz"`, `".tar.gz"`, and `".plt"`.

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

- a local directory – The name of the package is the name of the directory. The checksum is not present. For example, `"~/tic-tac-toe/"` is 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 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:

  `github://github.com/`⟨*user*⟩`/`⟨*repository*⟩`/`⟨*branch-or-tag*⟩`/`⟨*optional-subpath*⟩

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

  The `zip`-formatted archive for the repository (generated by GitHub for every branch and tag) is used as a remote URL archive path, except the checksum is the hash identifying the branch (or tag).

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

- a package name – A package name resolver 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 `-`.

A *package name resolver* (*PNR*) is a server that converts package names to other package sources. A PNR is identified by a string representing a URL. This URL is combined with `pkg/`⟨*package*⟩ path segments (where ⟨*package*⟩ is a package name) plus a `version=`⟨*version*⟩ query (where ⟨*version*⟩ is the Racket version number) to form a URL that should refer to a `read`-able hash table with the keys: `'source` mapped to the package source string and `'checksum` mapped to the checksum value. Typically, the package source value for `'source` will be a remote URL.

PLT supports two package name resolvers that are enabled by default: `https://pkg.racket-lang.org` for new packages and `https://planet-compat.racket-lang.org` for automatically generated packages for old PLaneT packages. Anyone may host their own package name resolver. The source for the PLT-hosted resolvers is in the `(collection-file-path "pkg-index" "meta")` directory.

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

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 core Racket distribution. 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"` or `".ss"`.

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. 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.

A *package scope* determines the effect of package installations, updates, etc., with respect to different users, Racket versions, and Racket installations. The default package scope can be configured, but it is normally `user`, which is user-specific and version-specific; that is, package installation makes the package visible only for the installing user and with the installing version of Racket. The `installation` scope means that package installation makes the package visible to all users of the specific Racket installation that is used to install the package. Finally, the `shared` scope means user-specific, but for all versions and installations

of Racket.

# 2 Managing Packages

The Racket package manager has two user interfaces: a command line `raco` sub-command and a library. They have the exact same capabilities, as the command line interface invokes the library functions and reprovides all their options.

## 2.1 Command Line

The `raco pkg` sub-command provides the following sub-sub-commands:

- `raco pkg install` ⟨*option*⟩ ... ⟨*pkg-source*⟩ ... — Installs the given package sources with the given ⟨*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*⟩.
  - `--deps` ⟨*behavior*⟩ — Selects the behavior for dependencies, where ⟨*behavior*⟩ is one of
    - `fail` — Cancels the installation if dependencies are version requirements are unmet (default for most packages)
    - `force` — Installs the package(s) despite missing dependencies or version requirements (unsafe)
    - `search-ask` — Looks for the dependencies or updates via the configured package name resolvers (default if the dependency is an indexed name) but asks if you would like it installed or updated.
    - `search-auto` — Like `search-ask`, but does not ask for permission to install or update.
  - `--force` — Ignores conflicts (unsafe)
  - `--ignore-checksums` — Ignores errors verifying package checksums (unsafe).
  - `--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.
  - `--link` — Implies `--type dir` (and overrides any specified type), and links the existing directory as an installed package.
  - `--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 as user-specific and Racket version-specific.
    * `shared` — Install packages as user-specific, but for all Racket versions.

    The default package scope is normally `user`, but it can be configured with `raco pkg config -i --set default-scope` ⟨*scope*⟩.
  - `-i` or `--installation` — Shorthand for `--scope installation`.
  - `-u` or `--user` — Shorthand for `--scope user`.
  - `-s` or `--shared` — Shorthand for `--scope shared`.

- `raco pkg update` ⟨*option*⟩ ... ⟨*pkg*⟩ ... — Checks the specified packages for package updates. If an update is found, but it cannot be installed (e.g. it conflicts with another installed package), then this command fails without installing any of the ⟨*pkg*⟩s (or their dependencies). The `update` sub-command accepts the following ⟨*option*⟩s:
  - `--deps` ⟨*behavior*⟩ — Same as for `raco pkg install`.
  - `--all` or `-a` — Update all packages, if no packages are given in the argument list.
  - `--update-deps` — Checks the named packages, and their dependencies (transitively) for updates.
  - `--no-setup` — 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`.
  - `-s` or `--shared` — Shorthand for `--scope shared`.

- `raco pkg remove` ⟨*option*⟩ ... ⟨*pkg*⟩ ... — Attempts to remove the given packages. If a package is the dependency of another package that is not listed, this command fails without removing any of the ⟨*pkg*⟩s. It accepts the following ⟨*option*⟩s:
  - `--force` — Ignore dependencies when removing packages.
  - `--no-setup` — Same as for `raco pkg install`.
  - `--auto` — Remove packages that were installed by the `search-auto` or `search-ask` dependency behavior and are no longer required.
  - `--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`.
  - `-s` or `--shared` — Shorthand for `--scope shared`.

- `raco pkg show` ⟨*option*⟩ ... — Print information about currently installed packages. By default, packages are shown for all installation modes (installation-wide, user- and Racket-version-specific, and user-specific all-version). The command accepts the following ⟨*option*⟩s:

  - `-d` — Adds a column in the output for the directory the package is installed to.
  - `--scope` ⟨*scope*⟩ — Shows only packages in ⟨*scope*⟩, which is one of
    * `installation` — Show only installation-wide packages.
    * `user` — Show only user-specific, version-specific packages.
    * `shared` — Show only user-specific, all-version packages.

    The default is to show packages for all package scopes.
  - `-i` or `--installation` — Shorthand for `--scope installation`.
  - `-u` or `--user` — Shorthand for `--scope user`.
  - `-s` or `--shared` — Shorthand for `--scope shared`.
  - `--version` ⟨*vers*⟩ or `-v` ⟨*vers*⟩ — Show only user-specific packages for Racket version ⟨*vers*⟩.

- `raco pkg config` ⟨*option*⟩ ... ⟨*key*⟩ ⟨*val*⟩ ... — View and modify package configuration options. It accepts 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`.
  - `-i` or `--installation` — Shorthand for `--scope installation`.
  - `-u` or `--user` — Shorthand for `--scope user`.
  - `-s` or `--shared` — Shorthand for `--scope shared`.

  The valid keys are:
  - `indexes` — A list of URLs for package name resolvers.
  - `default-scope` — Either `installation`, `user`, or `shared`. This configuration option exists only with the `installation` scope (i.e., it's an installation-wide configuration of the default package scope for `raco pkg` commands).

- `raco pkg create` ⟨*option*⟩ ... ⟨*package-directory*⟩ — Bundles a package directory into a package archive. It accepts the following ⟨*option*⟩s:
  - `--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.

## 2.2 Programmatic

```
(require pkg)
```

The `pkg` module provides a programmatic interface to the command sub-sub-commands.

```
install : procedure?
update : procedure?
remove : procedure?
show : procedure?
config : procedure?
create : procedure?
```

Duplicates the command line interface.

Each long form option of the command-line interface is keyword argument. An argument corresponding to `--type`, `--deps`, `--format`, or `--scope` accepts its argument as a symbol. All other options accept booleans, where `#t` is equivalent to the presence of the option.

# 3 Developing Packages

To create a package, first make a directory for your package and select its name, ⟨*package*⟩:

```
mkdir ⟨package⟩
```

Next, link your development directory to your local package repository:

```
raco pkg install --link ⟨package⟩
```

Optionally, enter your directory and create a basic `"info.rkt"` file:

```
cd ⟨package⟩

echo "#lang setup/infotab" > info.rkt

echo "(define deps (list))" >> info.rkt
```

The `"info.rkt"` file is not necessary if you have no dependencies, but you may wish to create it to simplify adding dependencies in the future. (Note that this `"info.rkt"` is for the package, not for a collection; definitions such as `scribblings` or `raco-commands` work only in a collection's `"info.rkt"`.)

Next, inside the ⟨*package*⟩ directory, create directories for the collections and modules that your package will provide. For example, the developer of tic-tac-toe 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
```

After your package is ready to deploy, choose either §3.1 "GitHub Deployment" or §3.2 "Manual Deployment".

## 3.1 GitHub Deployment

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

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

Typically, ⟨*branch*⟩ will be `master`, but you may wish to use different branches for releases and development.

Whenever you

```
git push
```

your changes will automatically be discovered by those who used your package source when they use `raco pkg update`.

## 3.2 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*⟩

And then upload the archive and its checksum to your site:

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

Now, publish your package source as:

`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.

## 3.3 Helping Others Discover Your Package

By using either of the above deployment techniques, anyone will be able to use 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 name resolver.

If you'd like to use the official package name resolver, browse to https://pkg.racket-lang.org/manage/upload 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 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 official package name resolver will notice, and `raco pkg update` will work on your user's machines.

## 3.4 Naming and Designing Packages

Although of course not required, we suggest the following system 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 package name resolvers ignore version declarations; in particular, a package is a candidate for updating when its checksum changes, independent of whether the package's version changes or in which direction the version changes.
- 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 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.
- Packages are not allowed to start with plt, racket, or planet without special approval from PLT curation.

# 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 `setup/infotab` language.

The following fields are used by the package manager:

- `version` — a version string. The default version of a package is `"0.0"`.
- `deps` — a list of dependencies, where each dependency is either a package source strings or a list containing a package source string and a version string.

  Each elements 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.

  When provided, a version string specifies a lower bound on an acceptable version of the package.

  Use the package name `"racket"` to specify a dependency on the version of the Racket installation.
- `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).

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

```
#lang setup/infotab
(define version "1.0")
(define deps (list package-source-string ...))
```

# 5 PLaneT Compatibility

PLT maintains a package name resolver to serve packages that were developed using the original PLaneT package system. This compatibility resolver is at https://planet-compat.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.

# 6 FAQ

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

## 6.1 Are package installations versioned with respect to the Racket version?

By default, when you install a package, it is installed for a specific user and a specific version of Racket. That is, the package scope is user- and version-specific.

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- and version-specific package scope.

Finally, you can use the `-s` or `--shared` flag with `raco pkg` commands to install user-specific packages that apply to all Racket versions that you run. (In contrast, PLaneT requires reinstallation of all packages every version change.)

## 6.2 Where and how are packages installed?

User-specific and Racket-version-specific packages are in `(build-path (find-system-path 'addon-dir) (version) "pkgs")`, user-specific and all-version packages are in `(build-path (find-system-path 'addon-dir) "pkgs")`, and installation-wide packages are in `(build-path (find-lib-dir) "pkgs")`. They are linked as collection roots with `raco link`.

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

User-specific packages are checked against installation-wide packages for conflicts. Installation-wide packages are checked only against other installation-wide packages.

Beware that a new installation-wide package can invalidate previous conflict checks for user-specific packages. Similarly, new user-specific but all-version packages can invalidate previous user-specific conflict checks for a different Racket version.

## 6.4 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).

## 6.5 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.

## 6.6 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 §7.1 "Short Term".

# 7 Future Plans

## 7.1 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 PNR will divide packages into three categories: `planet`, `solar-system`, and `galaxy`. The definitions for these categories are:
  - `galaxy` — No restrictions.
  - `solar-system` — Must not conflict any package in `solar-system` or `planet`.
  - `planet` — Must not conflict any package in `solar-system` or `planet`. 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 the `solar-system`, with the `galaxy` 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 `galaxy` or `solar-system` 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 `solar-system`, but B will be in `galaxy`.) 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 `planet` 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 `solar-system` category, automatically.

- In order to mitigate the costs of external linking vis a vis the inability to understand code in isolation, we will create a module resolver that searches for providers of modules on the configured package name resolvers. For example, if a module requires `"data/matrix.rkt"`, and it is not available, then the PNR will be consulted to discover what packages provide it. *Only packages in `solar-system` or `planet` 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 `planet` packages but not `solar-system` packages, while others may not want to install any.)

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

## 7.2 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 PNR 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 name resolver with a public key shipped with Racket, and allow other resolvers to implement a similar security scheme.
- Packages in the `planet` 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.
- Packages can contain any kinds of files, including bytecode and documentation, which would reduce the time required to install a package (since we must run `raco setup`). However, packages with these included are painful to maintain and unreliable given users with different versions of Racket installed.

  One solution is to have a separate place where such "binary" packages are available. For example, PLT could run a PNR for every Racket version, i.e., `"https://binaries.racket-lang.org/5.3.1.4"`, that would contain the binaries for all the packages in the `planet` category. Thus, when you install package tic-tac-toe you could also install the binary version from the appropriate PNR.

There are obvious problems with this... it could be expensive for PLT in terms of space and time... Racket compilation is not necessarily deterministic or platform-independent.

This problem requires more thought.

- 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.
- The core distribution should be split apart into many more packages. For example, Redex, Plot, the Web Server, and the teaching languages are natural candidates for being broken off.
- The core should be able to be distributed with packages that will be installed as soon as the system is installed. Ideally, this would be customizable by instructors so they could share small distributions with just the right packages for their class.