PLaneT: Automatic Package Distribution

Version 7.4.0.10

Jacob Matthews
and Robert Bruce Findler

September 22, 2019

This page is for legacy support of PLaneT, Racket's deprecated, old package system. All new work should use the new Racket package system instead.

The PLaneT system is a method for automatically sharing code packages, both as libraries and as full applications, that gives every user of a PLaneT client the illusion of having a local copy of every code package on the server. It consists of the central PLaneT package repository, a server that holds all PLaneT packages, and the PLaneT client, built into Racket, which transparently interacts with the server on your behalf when necessary.

Contents

1 Using PLaneT

To use a PLaneT package in a program, require it using the `planet require` form (see §3.2 "Importing and Exporting: `require` and `provide`" for a full reference on the features of the `require` statement in general and the exact allowed grammar of PLaneT require statements). Here we explain how to use PLaneT by example.

1.1 Finding a Package

If you are new to PLaneT, the first thing to do is visit the PLaneT repository web site and see what packages are available. People contribute new PLaneT packages all the time — if you want to be notified whenever a new or updated package is released, you can subscribe to the (announcement-only) PLaneT-announce mailing list or use an RSS reader to subscribe to PLaneT's RSS feed.

To use a package from PLaneT in your program, the easiest thing to do is copy the `require` code snippet off of that package's page and paste it ino your program. For instance, to use Schematics' spgsql.plt package (a library for interacting with the PostgresQL database), as of this writing you would copy and paste the line:

```
(require (planet "spgsql.rkt" ("schematics" "spgsql.plt" 2 3)))
```

into your program. This line requires the file `"spgsql.rkt"` in package version 2.3 of the `"spgsql.plt"` package written by `"schematics"`. That does two things: first, it downloads and installs a version of `"spgsql.plt"` that is compatible with package version 2.3 from the central PLaneT repository if a compatible version hasn't already been installed. Second, it requires the module in file `"spgsql.rkt"` from that package, making all of its exported bindings available for use.

Unlike with most package-distribution systems, package downloading and installation in PLaneT is *transparent*: there's no need for you to do anything special the first time you want to use a package, and there's no need for you to even know whether or not a particular package is installed on your computer or the computers where your code will be deployed.

If you want to find all of the latest versions of the packages that planet has available, visit

`http://planet.racket-lang.org/servlets/pkg-info.ss`

It returns a list matching the contract

```
(listof (list/c string?
                string?
                (list/c exact-positive-integer?
                        exact-nonnegative-integer?)))
```

Each sublist represents the latest version of one of the packages and contains the userid, the package name (including ".plt"), and the version (major and minor numbers).

1.2 Shorthand Syntax

The code snippet above can also be written using a new shorter syntax:

```
(require (planet schematics/spgsql:2:3/spgsql))
```

The two forms behave identically. In the abbreviated syntax, however, it is illegal to write the trailing `".rkt"` suffix on the file name to be required or the trailing `".plt"` on the package file name. (They are mandatory for the long-form syntax.) It is also legal in the abbreviated syntax to omit a filename to be required entirely; in that case, PLaneT requires the file `"main.rkt"` in the given package.

1.3 Networking troubles

Sometimes, when PLaneT tries to download and install a package for the first time, your operating system may block it from access to the network. If you are uncomfortable giving DrRacket free access to the network (or if your attempts to do so do not seem to work), then you can use your browser to manually install a planet package.

To see how this works, lets assume you want to install the PLAI package and

```
(require (planet plai/plai:1))
```

is not working for you.

- First, fire up a command-line window and use `raco planet url` to determine the url for downloading the package. To find the url for version `(1 1)` of the plai package, do this:

  ```
  % raco planet url plai plai.plt 1 1
  ```

  and get this as a response:

  ```
  http://planet.racket-lang.org/servlets/planet-
  servlet.rkt?lang=%224.1.5.3%22&name=%22plai.plt%22&maj=1&min-
  lo=1&min-hi=%23f&path=%28%22plai%22%29
  ```

- Copy and paste that url into your browser, which should trigger the dowload of a file called `plai.plt`. Note that your browser will probably try to call the file something else. Rename it to `plai.plt`.

- Now run the command-line tool one more time to install the plt file:

  ```
  % raco planet fileinject plai plai.plt 1 1
  ```

  This command should be run from the same directory where you saved `plai.plt`.

  This command may fail, since version `(1 1)` of the PLAI package depends on `cce/scheme:4:1`. If it does, simply repeat the above steps for that package first, and then continue with the `fileinject` command for PLAI.

- Finally, to check that the installation is successful, run `raco planet show`. You should see output like this (possibly with slightly different version numbers, if the packages have been updated since this was written):

  ```
  Normally-installed packages:
    cce   scheme.plt      4 1
    plai  plai.plt        1 1
  ```

Once that is complete, PLaneT will use that version of the package for any subsequent `require`s and won't try to use the network.

If you wish to ensure that PLaneT won't use the network even if your operating system allows it, you can use the `download?` parameter of the `planet/resolver` module to control whether it attempts to download files. Similarly, you can use the `install?` parameter to prevent installation. Finally, you can block access at the operating system level to the path returned by `(PLANET-BASE-DIR)` to control which operating system users can install PLaneT packages.

1.4 Fine-Grained Control Over Package Imports

The PLaneT client is designed to balance two competing goals: transparent upgradability and control over the effect of a package requirement. To that end, the most basic PLaneT require form offers maximum upgradability, but several more specialized forms allow finer-grained control over what versions of the named package may be downloaded.

Package versions should not be confused with program or library versions; a *package version* is a PLaneT-specific version number that encodes backwards-compatibility information.

The most basic planet require line, which is what is used in the form

```
(require (planet "spgsql.rkt" ("schematics" "spgsql.plt" 2 3)))
```

in longhand notation, or

```
(require (planet schematics/spgsql:2:3/spgsql))
```

in shorthand notation, should be read "Require from PLaneT *any* release of Schematics' `"spgsql.plt"` package that is backwards-compatible with package version 2.3." (The actual package version used is determined by the PLaneT search order.) To signal this explicitly, it is possible to write

```
(require (planet "spgsql.rkt" ("schematics" "spgsql.plt" 2 (+ 3))))
```

or

```
(require (planet schematics/spgsql:2:>=3/spgsql))
```

both of which mean the same thing as the first pair of require lines.

See §6.4 "Determine Your Package's Backwards-Compatibility" for a more detailed discussion of backwards-compatibility obligations for PLaneT packages.

The notion of "backwards-compatibility" has a specific meaning in PLaneT: by definition, for the purposes of automation, a package is considered to be backwards-compatible with any other package of the same owner, name, and major version, and any *lower* minor version. Package maintainers are responsible for marking new releases that break backwards-compatibility by incrementing their major-version number. This means that all of the above require specifications will match any release of `"unlib.plt"` with major package version 3 (and any minor version), but will *never* match releases of `"unlib.plt"` with higher (or lower) major version numbers.

Of course a package author may make a mistake and introduced a backwards-incompatibility unintentionally, or may fix a bug that code in third-party libraries was already working around. In those cases, it may help to make use of the "upper bound" form of the planet require, in longhand form:

```
(require (planet "reduction-semantics.rkt"
                 ("robby" "redex.plt" 4 (- 3))))
```

and using shorthand notation:

```
(require (planet robby/redex:4:<=3/reduction-semantics))
```

In this require line, any version of the package `"redex.plt"` from package version 4.0 to package version 4.3 will match the require spec (though as with any PLaneT require specification, the PLaneT package search order determines which package is actually loaded).

It is also possible to specify both an upper and a lower bound, using the planet require's "range" form:

```
(require (planet "test.rkt" ("schematics" "schemeunit.plt" 2 (9 10))))
```

or

```
(require (planet schematics/schemeunit:2:9-10/test))
```

This form matches any package in the specified range (inclusive on both ends), in this example the specifications match either package version 2.9 or 2.10 of the `"schemeunit.plt"`

package, but do not match version with higher or lower minor version numbers (or any other major version number).

Using the range form, it is possible to require a specific version of a package as a special case (choosing the upper and lower bounds to be equal), but this is a common enough case that it has special support with the "exact-match" form:

```
(require (planet "unzip.rkt" ("dherman" "zip.plt" 2 (= 1))))
```

or

```
(require (planet dherman/zip:2:=1/unzip))
```

match only the exact package version 2.1 of the `"zip.plt"` package.

1.5 Monitoring PLaneT's progress

PLaneT logs information about what it is doing to the `info` log (via `log-info`). In DrRacket, you can view the logs from the Show Log menu item in the View menu, and Racket's logging output can be controlled via command-line options and via environment variables. See §15.5 "Logging" for more details.

2 The PLaneT Search Order

PLaneT has four strategies it uses in order to match a request with an appropriate package that.

2.1 Previous Linkage

Whenever a file requires a package via PLaneT and that requirement is satisfied, the system makes a note of exactly which package satisfied that requirement and from then on always uses that exact same package, even if a newer version is available. This is done to prevent "magic upgrades" in which a program stops working after installation because an unrelated package was installed. Such connections are called links and are stored in a user-specific table called the linkage table.

2.2 Acceptable Local Package

If the PLaneT client doesn't have any previous linkage information, it checks its list of already-installed PLaneT packages for one that meets the requirement, and uses it if available. Both PLaneT-installed packages and packages established through a development link (see §6.1.1 "Development Links") are checked simultaneously at this stage.

2.3 Acceptable Remote Package

If there is no acceptable local package, the PLaneT client sends a request to the PLaneT server for a new package that would satisfy the requirement. The server then finds the newest matching package and sends it back to the client, which then installs it and uses it to satisfy the original requirement.

2.4 Cached Installation Archive

If the remote server cannot be contacted (or fails in any way to deliver an acceptable package), the PLaneT client consults the uninstalled-packages cache, a cache of all previously-downloaded packages, even those that are not currently installed. Racket users who frequently upgrade their installations may have many packages downloaded but not installed at any given time; this step is intended to ensure that these users can still run programs even if they temporarily lose network connection.

3 The `raco planet` Command-Line Tool

The `raco planet` command-line tool allows a command-line interface to the most commonly-performed PLaneT tasks. It is invoked from the command line as

```
raco planet subcommand arg ...
```

where *subcommand* is a subcommand from the following list, and *arg* is a sequence of arguments determined by that subcommand:

- `create`: create a PLaneT archive from a directory
- `install`: download and install a given package
- `remove`: remove the specified package from the local cache
- `show`: list the packages installed in the local cache
- `clearlinks`: clear the linkage table, allowing upgrades
- `fileinject`: install a local file to the planet cache
- `link`: create a development link
- `unlink`: remove development link associated with the given package
- `fetch`: download a package file without installing it
- `url`: get a URL for the given package
- `open`: unpack the contents of the given package
- `structure`: display the structure of a given .plt archive
- `print`: display a file within of the given .plt archive

Each of these commands is described in more detail below. All the functionality of the command-line tool is also provided with a programmatic interface by the `"util.rkt"` library.

3.1 `create`

Usage:

```
raco planet create [ <option> ... ] <path>
```

Create a PLaneT archive in the current directory whose contents are the directory `<path>`.

`<option>` is one of:

- `-f, --force`: force a package to be created even if its info.rkt file contains errors.

3.2 `install`

Usage:

```
raco planet install <owner> <pkg> <maj> <min>
```

Download and install the package that `(require (planet "file.rkt" (<owner> <pkg> <maj> <min>)))` would install.

3.3 `remove`

Usage:

```
raco planet remove [ <option> ... ] <owner> <pkg> <maj> <min>
```

Remove the specified package from the local cache, optionally also removing its distribution file.

`<option>` is one of:

- `-e, --erase`: also remove the package's distribution file from the uninstalled-package cache

3.4 `show`

Usage:

```
raco planet show [ <option> ... ]
```

List the packages installed in the local cache.

`<option>` is one of:

- `-p, --packages`: show packages only (default)
- `-l, --linkage`: show linkage table only
- `-a, --all`: show packages and linkage

3.5 clearlinks

Usage:

```
raco planet clearlinks
```

Clear the linkage table, allowing upgrades.

3.6 fileinject

Usage:

```
raco planet fileinject <owner> <plt-file> <maj> <min>
```

Install local file <plt-file> into the planet cache as though it had been downloaded from the planet server. It is treated as though it had the given owner name as its owner name, the given file's filename as the its package name, and the given major and minor version numbers.

3.7 link

Usage:

```
raco planet link <owner> <pkg> <maj> <min> <path>
```

Create a development link (see §6.1.1 "Development Links") between the given package specifier and the specified directory name.

3.8 unlink

Usage:

```
raco planet unlink [ <option> ] <owner> <pkg> <maj> <min>
```

Remove any development link (see §6.1.1 "Development Links") associated with the given package.

<option> can only be:

- `-q, --quiet`: don't signal an error on nonexistent links

3.9 `fetch`

Usage:

```
raco planet fetch <owner> <pkg> <maj> <min>
```

Download the given package file from the central PLaneT repository without installing it.

3.10 `url`

Usage:

```
raco planet url <owner> <pkg> <maj> <min>
```

Get a URL for the given package.

This is never necessary for normal use of planet, but may be helpful in some circumstances for retrieving packages.

3.11 `open`

Usage:

```
raco planet open <plt-file> <target>
```

Unpack the contents of the given package into the given directory without installing.

This command is not necessary for normal use of planet. It is intended to allow you to inspect package contents offline without needing to install the package.

3.12 `structure`

Usage:

```
raco planet structure <plt-file>
```

Print the structure of the PLaneT archive named by <plt-file> to the standard output port.

This command does not unpack or install the named .plt file.

3.13 `print`

Usage:

```
raco planet print <plt-file> <path>
```

Print the contents of the file named by <path>, which must be a relative path within the PLaneT archive named by <plt-file>, to the standard output port.

This command does not unpack or install the named .plt file.

4 The `planet` Language

`#lang planet` package: `base`

When used with `#lang`, `planet` must be followed by a short-form PLaneT path. The path is used in the same way that `#lang` uses plain identifiers: `/lang/reader` is added to the given path to determine a module that supplies a module reader.

The `planet` module (as opposed to the reader used with `#lang`) implements the `raco planet` command-line tool.

5 Utility Libraries

The planet collection provides configuration and utilities for using PLaneT.

5.1 Resolver

```
(require planet/resolver)
```
package: base

The primary purpose of this library to for `require` to find PLaneT packages. It also, however, provides some utilities for manipulating the resolvers behavior.

```
(planet-module-name-resolver r-m-p) → void?
  r-m-p : resolved-module-path?
(planet-module-name-resolver spec
                             module-path
                             stx
                             load
                             orig-paramz)
 → resolved-module-path?
  spec : (or/c module-path? path?)
  module-path : (or/c #f resolved-module-path?)
  stx : (or/c #f syntax?)
  load : boolean?
  orig-paramz : parameterization?
```

This implements the PLaneT module resolution process. It is `dynamic-require`d by racket when the first PLaneT module require is needed. It acts much like a `current-module-name-resolver` would, but racket provides it with a special `parameterization?` (giving it special privileges) that it uses when installing new packages.

```
(get-planet-module-path/pkg spec
                            module-path
                            stx)        → path? pkg?
  spec : (or/c module-path? path?)
  module-path : (or/c #f resolved-module-path?)
  stx : (or/c #f syntax?)
```

Returns the path corresponding to the package (interpreting the arguments the same way as `planet-module-name-resolver` and `(current-module-name-resolver)`).

```
(resolve-planet-path planet-path) → path?
  planet-path : any/c
```

Returns the file system path to the file specified by the given quoted PLaneT require specification. This function downloads and installs the specified package if necessary, but does not verify that the actual file within it actually exists.

If the PLaneT package is not actually installed, then this function expects to be called with a very powerful security guard, one that is available to the built-in module name resolver, but not generally available to user code. So probably this function will fail (possibly deadlock).

This is the same function as the one with the same name, exported by `planet/util`.

```
(download?) → boolean?
(download? dl?) → void?
  dl? : boolean?
```

A parameter that controls if PLaneT attempts to download a package that isn't already present. If the package isn't present, the resolver will raise the `exn:fail:planet?` exception instead of downloading it.

```
(install?) → boolean?
(install? inst?) → void?
  inst? : boolean?
```

A parameter that controls if PLaneT attempts to install a package that isn't already installed. If the package isn't installed, the resolver will raise the `exn:fail:planet?` exception instead of installing it.

5.1.1 Resolver file locking

When PLaneT is asked to resolve a module path for loading the file (e.g., when the last argument to the `(current-module-name-resolver)` is `#t` and that resolver triggers a call to the PLaneT resolver), it finds the directory where the files are installed, say in this directory, which corresponds to version 1.2 of dyoo's closure-compile.plt package:

```
"(CACHE-DIR)/dyoo/closure-compile.plt/1/2/"
```

If the file

```
"(CACHE-DIR)/dyoo/closure-compile.plt/1/2.SUCCESS"
```

is there, it assumes that there is no installation needed and it just continues, using the path to the file inside that directory.

If the `"2.SUCCESS"` file is not there, then it attempts to grab an `'exclusive` filesystem lock on this file (via `port-try-file-lock?`)

```
"(CACHE-DIR)/dyoo/closure-compile.plt/1/2.LOCK"
```

If it gets the lock, it then proceeds with the installation, calling raco setup to do the unpacking, compilation, and docs building. After the unpacking has finished, but before beginning compilation and docs building, it creates the `"2.UNPACKED"` file:

```
"(CACHE-DIR)/dyoo/closure-compile.plt/1/2.UNPACKED"
```

When compilation and docs build are complete, it creates the `"2.SUCCESS"` file:

```
"(CACHE-DIR)/dyoo/closure-compile.plt/1/2.SUCCESS"
```

and releases the lock on the `"2.LOCK"` file.

If it fails to get the lock on `"2.LOCK"` and it does not already hold the lock (due to a reentrant call to the resolver (the resolver knows about locks it holds via an internal parameter that gets created when the `planet/resolver` module is instantiated) then it goes into a loop that polls for the existence of the `"2.SUCCESS"` file; when it that file appears, it just continues, without installing anything (since that means someone else installed it).

In some situations (e.g., when a new namespace is created and a fresh instantiation of `planet/resolver` is created), PLaneT can be fooled into thinking that it does not hold the lock on some installation. In order to cope with these situations somewhat, PLaneT takes an easier path when the resolver is only looking for information about package files (i.e., when the last argument to the resolver is `#f`, or when `get-planet-module-path/pkg` is called directly (as opposed to being called via module resolution). In those cases, PLaneT will look only for the `"2.UNPACKED"` file instead of the `"2.SUCCESS"` file.

5.2 Client Configuration

`(require planet/config)` package: `base`

The `planet/config` library provides several parameters useful for configuring how PLaneT works.

Note that while these parameters can be useful to modify programmatically, PLaneT code runs at module-expansion time, so most user programs cannot set them until PLaneT has already run. Therefore, to meaningfully change these settings, it is best to manually edit the `"config.rkt"` file.

```
(PLANET-BASE-DIR) → path-string?
(PLANET-BASE-DIR dir) → void?
  dir : path-string?
```

The root of the tree where planet stores all of its files. Defaults to

```
(let ([plt-planet-dir-env-var (getenv "PLTPLANETDIR")])
  (if plt-planet-dir-env-var
      (string->path plt-planet-dir-env-var)
      (build-path (find-system-path 'addon-dir)
                  "planet"
                  (PLANET-CODE-VERSION))))
```

```
(PLANET-DIR) → path-string?
(PLANET-DIR dir) → void?
  dir : path-string?
```

The root of the version-specific PLaneT files. Defaults to `(build-path (PLANET-BASE-DIR) (get-installation-name))`.

```
(CACHE-DIR) → path-string?
(CACHE-DIR dir) → void?
  dir : path-string?
```

The root of the PLaneT client's cache directory.

```
(UNINSTALLED-PACKAGE-CACHE) → path-string?
(UNINSTALLED-PACKAGE-CACHE dir) → void?
  dir : path-string?
```

The root of the PLaneT client's uninstalled-packages cache. PLaneT stores package distribution files in this directory, and searches for them in this directory for them if necessary. Unlike the main PLaneT cache, which contains compiled files and is specific to each particular version of Racket, the uninstalled package cache is shared by all versions of Racket that use the same package repository, and it is searched if a package is not installed in the primary cache and cannot be downloaded from the central PLaneT repository (for instance due to a loss of Internet connectivity). This behavior is intended to primarily benefit users who upgrade their Racket installations frequently.

```
(LINKAGE-FILE) → path-string?
(LINKAGE-FILE file) → void?
  file : path-string?
```

The file to use as the first place PLaneT looks to determine how a particular PLaneT dependence in a file should be satisfied. The contents of this file are used to ensure that no "magic upgrades" occur after a package is installed. The default is the file `"LINKAGE"` in the root PLaneT directory.

```
(LOG-FILE) → (or/c path-string? false?)
(LOG-FILE file) → void?
  file : (or/c path-string? false?)
```

If #f, indicates that no logging should take place. Otherwise specifies the file into which logging should be written. The default is the file "INSTALL-LOG" in the root PLaneT directory.

```
(USE-HTTP-DOWNLOADS?) → boolean?
(USE-HTTP-DOWNLOADS? bool) → void?
  bool : any/c
```

PLaneT can use two different protocols to retrieve packages. If #t, PLaneT will use the HTTP protocol; if #f it will use the custom-built PLaneT protocol. The default value for this parameter is #t and setting this parameter to #f is not recommended.

```
(HTTP-DOWNLOAD-SERVLET-URL) → string?
(HTTP-DOWNLOAD-SERVLET-URL url) → void?
  url : string?
```

The URL for the servlet that will provide PLaneT packages if USE-HTTP-DOWNLOADS? is #t, represented as a string. This defaults to the value of the PLTPLANETURL environment variable if it is set and otherwise is "http://planet.racket-lang.org/servlets/planet-servlet.rkt".

```
(PLANET-SERVER-NAME) → string?
(PLANET-SERVER-NAME host) → void?
  host : string?
```

The name of the PLaneT server to which the client should connect if USE-HTTP-DOWNLOADS? is #f. The default value for this parameter is "planet.racket-lang.org".

```
(PLANET-SERVER-PORT) → natural-number?
(PLANET-SERVER-PORT port) → void?
  port : natural-number?
```

The port on the server the client should connect to if USE-HTTP-DOWNLOADS? is #f. The default value for this parameter is 270.

```
(HARD-LINK-FILE) → path?
(HARD-LINK-FILE file) → void?
  file : path?
```

The name of the file where hard links are saved. Defaults to (build-path (PLANET-BASE-DIR) (get-installation-name) "HARD-LINKS").

```
(PLANET-ARCHIVE-FILTER) → (or/c #f string? regexp?)
(PLANET-ARCHIVE-FILTER regexp-filter) → void?
  regexp-filter : (or/c #f string? regexp?)
```

A regular-expression based filter that is used to skip files when building a PLaneT archive.

```
(PLANET-CODE-VERSION) → string?
(PLANET-CODE-VERSION vers) → void?
  vers : string?
```

Used to compute PLANET-BASE-VERSION.

```
(DEFAULT-PACKAGE-LANGUAGE) → string?
(DEFAULT-PACKAGE-LANGUAGE vers) → void?
  vers : string?
```

The package language used when communicating with the server to find which package to download.

Defaults to (version).

5.3 Package Archives

(require planet/planet-archives) package: base

```
(get-all-planet-packages)
 → (listof (list/c (and/c path? absolute-path?) string? string? (listof string?)
                   exact-nonnegative-integer?
                   exact-nonnegative-integer?))
```

Returns the installed planet package. Each element of the result list corresponds to a single package. The first element in an inner list is the location of the installed files. The second and third elements are the owner and package names. The last two elements are the major and minor verisons

```
(get-installed-planet-archives)
 → (listof (list/c (and/c path? absolute-path?) string? string? (listof string?)
                   exact-nonnegative-integer?
                   exact-nonnegative-integer?))
```

Like get-all-planet-archives, except that it does not return packages linked in with "raco planet link".

```
(get-hard-linked-packages)
 → (listof (list/c (and/c path? absolute-path?) string? string? (listof string?)
                   exact-nonnegative-integer?
                   exact-nonnegative-integer?))
```

Like `get-all-planet-archives`, except that it return only packages linked in with "raco planet link".

5.4 Package Utils

`(require planet/util)` package: `planet-lib`

The `planet/util` library supports examination of the pieces of PLaneT. It is meant primarily to support debugging and to allow easier development of higher-level package-management tools. The functionality exposed by the `raco planet` command-line tool is also available programmatically through this library.

```
(download/install-pkg owner pkg maj min) → (or/c pkg? #f)
  owner : string?
  pkg : (and/c string? #rx"[.]plt$")
  maj : natural-number/c
  min : natural-number/c
```

Downloads and installs the package specifed by the given owner name, package name, major and minor version number. Returns false if no such package is available; otherwise returns a package structure for the installed package.

The *pkg* argument must end with `".plt"`.

```
(install-pkg pkg-spec file maj min) → (or/c pkg-spec? #f)
  pkg-spec : pkg-spec?
  file : path-string?
  maj : natural-number/c
  min : natural-number/c
```

Installs the package represented by the arguments, using the *pkg-spec* argument to find the path and name of the package to install.

See `get-package-spec` to build a *pkg-spec* argument.

Returns a new `pkg-spec?` corresponding to the package that was actually installed.

```
(get-package-spec owner pkg [maj min]) → pkg-spec?
```

```
  owner : string?
  pkg : (and/c string? #rx"[.]plt$")
  maj : (or/c #f natural-number/c) = #f
  min : (or/c #f natural-number/c) = #f
```

Builds a pkg-spec? corresponding to the package specified by *owner*, *pkg*, *maj*, and *min*.

The *pkg* argument must end with the string ".plt".

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

Recognizes the result of get-package-spec (and install-pkg).

```
(current-cache-contents)
 → (listof
    (list/c string?
            (listof
             (list/c string?
                     (cons/c natural-number/c
                             (listof natural-number/c))))))
(current-cache-contents contents) → void?
  contents : (listof
              (list/c string?
                      (listof
                       (list/c string?
                               (cons/c natural-number/c
                                       (listof natural-number/c))))))
```

Holds a listing of all package names and versions installed in the local cache.

```
(current-linkage)
 → (listof (list/c path-string?
                   (list/c string?
                           (list/c string?)
                           natural-number/c
                           natural-number/c)))
```

Returns the current linkage table.

The linkage table is an association between file locations (encoded as path strings) and concrete planet package versions. If a require line in the associated file requests a package, this table is consulted to determine a particular concrete package to satisfy the request.

```
(make-planet-archive directory [output-file]) → path-string?
```

```
  directory : path-string?
  output-file : (or/c path? path-string?)
              = (string-append (path->string name) ".plt")
```

Makes a .plt archive file suitable for PLaneT whose contents are all files in the given directory and returns that file's name. If the optional filename argument is provided, that filename will be used as the output file's name.

See also `build-scribble-docs?` and `force-package-building?`

```
(build-scribble-docs?) → boolean?
(build-scribble-docs? b) → void?
  b : boolean?
```

Determines if `make-planet-archive` builds scribble docs (or not).

```
(force-package-building?) → boolean?
(force-package-building? b) → void?
  b : boolean?
```

Determines if `make-planet-archive` signals an error and refuses to continue packaging for certain, more significant errors.

Defaults to `#t`, and thus packaging will signal errors.

```
(download-package pkg-spec)
 → (or/c (list/c #true path? natural-number/c natural-number/c)
         string?
         (list/c #false string?))
  pkg-spec : pkg-spec?
```

Downloads the package given by *pkg-spec*. If the result is a list whose first element is `#true`, then the package was downloaded successfully and the rest of the elements of the list indicate where it was downloaded, and the precise version number.

The other two possible results indicate errors. If the result is a list, then the server is saying that there is no matching package; otherwise the error is some lower-level problem (perhaps no networking, etc.)

```
(pkg->download-url pkg) → url?
  pkg : pkg?
```

Returns the url for a given package.

```
(get-package-from-cache pkg-spec) → (or/c #false path?)
  pkg-spec : pkg-spec?
```

Returns the location of the already downloaded package, if it exists (and `#false` otherwise).

```
(lookup-package-by-keys owner
                        name
                        major
                        minor-lo
                        minor-hi)
 → (or/c (list/c path?
                 string?
                 string?
                 (listof string?)
                 exact-nonnegative-integer?
                 exact-nonnegative-integer?)
         #false)
  owner : string?
  name : string?
  major : exact-nonnegative-integer?
  minor-lo : exact-nonnegative-integer?
  minor-hi : exact-nonnegative-integer?
```

Looks up and returns a list representation of the package named by the given owner, package name, major and (range of) minor version(s).

```
(unpack-planet-archive plt-file output-dir) → any
  plt-file : (or/c path? path-string?)
  output-dir : (or/c path? path-string?)
```

Unpacks the PLaneT archive with the given filename, placing its contents into the given directory (creating that path if necessary).

```
(remove-pkg owner pkg maj min) → any
  owner : string?
  pkg : (and/c string? #rx"[.]plt$")
  maj : natural-number/c
  min : natural-number/c
```

Removes the specified package from the local planet cache, deleting the installed files.

```
(erase-pkg owner pkg maj min) → any
  owner : string?
  pkg : (and/c string? #rx"[.]plt$")
  maj : natural-number/c
  min : natural-number/c
```

Like `remove-pkg`, removes the specified package from the local planet cache and deletes all of the files corresponding to the package, but also deletes the cached `".plt"` file (so it will be redownloaded later).

```
(display-plt-file-structure plt-file) → any
  plt-file : (or/c path-string? path?)
```

Print a tree representing the file and directory structure of the PLaneT archive .plt file named by *plt-file* to (current-output-port).

```
(display-plt-archived-file plt-file
                           file-to-print) → any
  plt-file : (or/c path-string? path?)
  file-to-print : string?
```

Print the contents of the file named *file-to-print* within the PLaneT archive .plt file named by *plt-file* to (current-output-port).

```
(unlink-all) → any
```

Removes the entire linkage table from the system, which will force all modules to relink themselves to PLaneT modules the next time they run.

```
(add-hard-link owner pkg maj min dir) → any
  owner : string?
  pkg : (and/c string? #rx"[.]plt$")
  maj : natural-number/c
  min : natural-number/c
  dir : path?
```

Adds a development link between the specified package and the given directory; once a link is established, PLaneT will treat the cache as having a package with the given owner, name, and version whose files are located in the given path. This is intended for package development; users only interested in using PLaneT packages available online should not need to create any development links.

If the specified package already has a development link, this function first removes the old link and then adds the new one.

The *pkg* argument must end with the string ".plt".

```
(remove-hard-link  owner
                   pkg
                   maj
                   min
                  [#:quiet? quiet?]) → any
  owner : string?
  pkg : (and/c string? #rx"[.]plt$")
```

```
  maj : natural-number/c
  min : natural-number/c
  quiet? : boolean? = #false
```

Removes any hard link that may be associated with the given package.

The *pkg* argument must end with the string ".plt". The *maj* and *min* arguments must be integers. This procedure signals an error if no such link exists, unless #:quiet? is #true.

```
(resolve-planet-path spec) → path?
  spec : quoted-planet-require-spec?
```

This is the same function as the one with the same name, exported by planet/resolver.

```
(path->package-version p)
 → (or/c (list/c string? string? natural-number/c natural-number/c) #f)
  p : path?
```

Given a path that corresponds to a PLaneT package (or some part of one), produces a list corresponding to its name and version, exactly like (this-package-version). Given any other path, produces #f.

```
(struct exn:fail:planet exn:fail (message continuation-marks)
    #:extra-constructor-name make-exn:fail:planet)
  message : string?
  continuation-marks : continuation-mark-set?
```

This exception record is used to report planet-specific exceptions.

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

Determines if its argument is a pkg, the representation of an installed package.

5.5 Package Version

Provides bindings for PLaneT developers that automatically produce references to the name and version of the containing PLaneT package so the same code may be reused across releases without accidentally referring to a different version of the same package.

```
(require planet/version)
```
package: planet-lib

```
(this-package-version)
```

```
(this-package-version-symbol)
(this-package-version-symbol suffix-id)
(this-package-version-name)
(this-package-version-owner)
(this-package-version-maj)
(this-package-version-min)
```

Macros that expand into expressions that evaluate to information about the name, owner, and version number of the package in which they appear. `this-package-version` returns a list consisting of a string naming the package's owner, a string naming the package, a number indicating the package major version and a number indicating the package minor version, or `#f` if the expression appears outside the context of a package. The macros `this-package-version-name`, `this-package-version-owner`, `this-package-version-maj`, and `this-package-version-min` produce the relevant fields of the package version list.

`this-package-version-symbol` produces a symbol suitable for use in `planet` module paths. For instance, in version `1:0` of the package `package.plt` owned by `author`, `(this-package-version-symbol dir/file)` produces `'author/package:1:0/dir/file`. In the same package, `(this-package-version-symbol)` produces `'author/package:1:0`.

```
(this-package-in suffix-id ...)
```

A `require` sub-form that requires modules from within the same PLaneT package version as the require, as referred to by each *suffix-id*. For instance, in version `1:0` of the package `package.plt` owned by `author`, `(require (this-package-in dir/file))` is equivalent to `(require (planet author/package:1:0/dir/file))`.

Note: Use `this-package-in` when documenting PLaneT packages with Scribble to associate each documented binding with the appropriate package.

```
(make-planet-symbol stx [suffix]) → (or/c #false symbol?)
  stx : syntax?
  suffix : (or/c #false string?) = #false
```

Returns a symbol representing a require spec for the location of *stx*, as a planet package.

```
(package-version->symbol ver [suffix]) → (or/c #false symbol?)
  ver : (or/c (list/c string? string? exact-nonnegative-integer? exact-nonnegative-integer?)
              #false)
  suffix : (or/c #false string?) = #false
```

Returns a symbol representing the require spec for *ver*, as a planet package.

5.6 Macros and Syntax Objects

```
(require planet/syntax)
```
package: planet-lib

Provides bindings useful for PLaneT-based macros.

```
(syntax-source-planet-package stx) → (or/c list? #f)
  stx : syntax?
(syntax-source-planet-package-owner stx) → (or/c string? #f)
  stx : syntax?
(syntax-source-planet-package-name stx) → (or/c string? #f)
  stx : syntax?
(syntax-source-planet-package-major stx) → (or/c integer? #f)
  stx : syntax?
(syntax-source-planet-package-minor stx) → (or/c integer? #f)
  stx : syntax?
(syntax-source-planet-package-symbol  stx
                                      [suffix])
 → (or/c symbol? #f)
  stx : syntax?
  suffix : (or/c symbol? #f) = #f
```

Produce output analogous to `this-package-version`, `this-package-version-owner`, `this-package-version-name`, `this-package-version-maj`, `this-package-version-min`, and `this-package-version-symbol` based on the source location of *stx*.

```
(make-planet-require-spec stx [suffix]) → syntax?
  stx : syntax?
  suffix : (or/c symbol? #f) = #f
```

Produces a `require` sub-form for the module referred to by *suffix* in the PLaneT package containing the source location of *stx*.

5.7 Scribble Documentation

```
(require planet/scribble)
```
package: planet-doc

Provides bindings for documenting PLaneT packages.

```
(this-package-in suffix-id ...)
```

This binding from `planet/version` is also exported from `planet/scribble`, as it is useful for `for-label` imports in Scribble documentation.

```
(racketmod/this-package maybe-file suffix-id datum ...)
(racketmodname/this-package suffix-id)
(racketmodname/this-package (unsyntax suffix-expr))
(racketmodlink/this-package suffix-id pre-content-expr ...)
(defmodule/this-package maybe-req suffix-id maybe-sources pre-
flow ...)
(defmodulelang/this-package suffix-id maybe-sources pre-flow ...)
(defmodulelang/this-package suffix-id
  #:module-paths (mod-suffix-id ...) maybe-sources
  pre-flow ...)
(defmodulereader/this-package suffix-id maybe-sources pre-flow ...)
(defmodule*/this-package maybe-req (suffix-id ...+)
  maybe-sources pre-flow ...)
(defmodulelang*/this-package (suffix-id ...+)
  maybe-sources pre-flow ...)
(defmodulelang*/this-package (suffix-id ...+)
  #:module-paths (mod-suffix-id ...) maybe-sources
  pre-flow ...)
(defmodulereader*/this-package (suffix-id ...+)
  maybe-sources pre-flow ...)
(defmodule*/no-declare/this-package maybe-req (suffix-id ...+)
  maybe-sources pre-flow ...)
(defmodulelang*/no-declare/this-package (suffix-id ...+)
  maybe-sources pre-flow ...)
(defmodulelang*/no-declare/this-package (suffix-id ...+)
  #:module-paths (mod-suffix-id ...) maybe-sources pre-flow ...)
(defmodulereader*/no-declare/this-package (suffix-id ...+)
  maybe-sources pre-flow ...)
(declare-exporting/this-package suffix-id ... maybe-sources)
```

Variants of `racketmod`, `racketmodname`, `racketmodlink`, `defmodule`, `defmodulereader`, `defmodulelang`, `defmodule*`, `defmodulelang*`, `defmodulereader*`, `defmodule*/no-declare`, `defmodulelang*/no-declare`, `defmodulereader*/no-declare`, and `declare-exporting`, respectively, that implicitly refer to the PLaneT package that contains the enclosing module.

The full module name passed to `defmodule`, etc is formed by appending the *`suffix-id`* or *`mod-suffix-id`* to the symbol returned by `(this-package-version-symbol)`, separated by a `/` character, and tagging the resulting symbol as a `planet` module path. As a special case, if *`suffix-id`* is `main`, the suffix is omitted.

For example, within a package named `package.plt` by `author`, version `1:0`, the following are equivalent:

```
(defmodule/this-package dir/file)
  = (defmodule (planet author/package:1:0/dir/file))
```

and

```
(defmodule/this-package main)
  = (defmodule (planet author/package:1:0))
```

5.8 Terse Status Updates

```
(require planet/terse-info)
```
package: base

This module provides access to some PLaneT status information. This module is first loaded by PLaneT in the initial namespace (when PLaneT's resolver is loaded), but PLaneT uses `dynamic-require` to load this module each time it wants to announce information. Similarly, the state of which procedures are registered (via `planet-terse-register`) is saved in the namespace, making the listening and information producing namespace-specific.

```
(planet-terse-register proc) → void?
  proc : (-> (or/c 'download 'install 'docs-build 'finish)
             string?
             any/c)
```

Registers *proc* as a function to be called when `planet-terse-log` is called.

Note that *proc* is called asynchronously (ie, on some thread other than the one calling `planet-terse-register`).

```
(planet-terse-log id msg) → void?
  id : (or/c 'download 'install 'finish)
  msg : string?
```

This function is called by PLaneT to announce when things are happening. See also `planet-terse-set-key`.

```
(planet-terse-set-key key) → void?
  key : any/c
```

This sets a thread cell to the value of *key*. The value of the thread cell is used as an index into a table to determine which of the functions passed to `planet-terse-register` to call when `planet-terse-log` is called.

The table holding the key uses ephemerons and a weak hash table to ensure that when the *key* is unreachable, then the procedures passed to `planet-terse-log` cannot be reached through the table.

5.9 The Cache File's Path

```
(require planet/cachepath)
```
package: base

```
(get-planet-cache-path) → (and/c path? absolute-path?)
```

Returns the path to the `"cache.rktd"` file for the planet installation.

6 Developing Packages for PLaneT

To put a package on PLaneT, or release an upgrade to an already-existing package:

6.1 Write Your Package

PLaneT can distribute whatever programs you write, but keep these guidelines in mind as you write:

- Organize your code into modules. Since the PLaneT client is integrated into the `require` form, it works best if your code is arranged into modules.
- When one module in your program depends on another, it is best to require it using the relative-file-name form rather than the planet require form. For instance, if your program contains files primary.rkt and helper.rkt where primary.rkt requires helper, use the form

  ```
  (require "helper.rkt")
  ```

  instead of

  ```
  (require (planet "helper.rkt" ("username" "packagename.plt" 1
  0)))
  ```

  in files that will also be a part of the package.

6.1.1 Development Links

To aid development, PLaneT allows users to establish direct associations between a particular planet package with an arbitrary directory on the filesystem, for instance connecting the package named by the require line

```
(require (planet "file.rkt" ("my" "mypackage.plt" 1 0)))
```

to the directory `"/home/myname/svn/mypackages/devel/"`.

These associations are intended to allow developers to use their own directory structures, version control systems, and so on while still being able to use the packages they create as though they were distributed directly by PLaneT. Development links are local to a particular user and repository (but not to a particular Racket minor revision).

To establish a development link, use the `raco planet` command-line tool:

```
raco planet link myname mypackage.plt 1 0 ~/svn/mypackages/devel
```

Once you are finished developing a package, you should remove any development links you have established for it, again using the planet command-line tool:

```
raco planet unlink myname mypackage.plt 1 0
```

You may alternately use the functions `add-hard-link` and `remove-hard-link`.

6.2 Prepare Your Distribution

6.2.1 Arrange Files Into a Directory

Make sure that all source files, documentation, etc. that you want to be a part of the package are in a single directory and its subdirectories. Furthermore make sure that nothing else, *e.g.* unneeded backup files, is in that directory (with the exception that the meta-subdirectories and files Git/Subversion/CVS uses are automatically skipped by the packaging tool).

6.2.2 Create Documentation [Optional]

Use Scribble to write documentation for your package. See §5.7 "Scribble Documentation" for macros that ensure proper bindings and version numbers in documentation for PLaneT packages, and *Scribble: The Racket Documentation Tool* for instructions on how to write Scribble documentation.

When testing your documentation, set up a development link and use

```
raco setup -P <owner> <package-name> <maj> <min>
```

with arguments based on the development link to build and test your documentation.

Note: Always use `this-package-in` in `for-label` bindings when documenting PLane T packages, and always use the bindings in `planet/scribble` rather than `scribble/manual`. These macros automatically produce `planet`-based module paths with appropriate version numbers. Other `require` subforms and Scribble declarations may refer to the wrong version of a package, or may not be recognized as part of a PLaneT package at all when documentation is produced.

6.2.3 Create an `"info.rkt"` File [Optional]

If you put a file named `"info.rkt"` in your package's root directory, the PLaneT system (as well as the rest of the Racket tool suite) will look in it for descriptive metadata about your package. The PLaneT system looks for certain names in that file:

- The `'blurb` field: If present, the blurb field should contain a list of XHTML fragments encoded as x-expressions (see the xml collection for details) that PLaneT will use as a short description of your project.
- The `'release-notes` field: If present, the release-notes field should contain a list of XHTML fragments encoded as x-expressions (see the xml collection for details) that PLaneT will use as a short description of what's new in this release of your package.
- The `'categories` field: If present, the categories field should be a list of symbols corresponding to the categories under which this package should be listed.

  The valid categories are:

  - `'devtools`: Development Tools
  - `'net`: Networking and Protocols
  - `'media`: Graphics and Audio
  - `'xml`: XML-Related
  - `'datastructures`: Data Structures and Algorithms
  - `'io`: Input/Output and Filesystem
  - `'scientific`: Mathematical and Scientific
  - `'system`: Hardware/Operating System-Specific Tools
  - `'ui`: Textual and Graphical User Interface
  - `'metaprogramming`: Metaprogramming Tools
  - `'planet`: PLaneT-Related
  - `'misc`: Miscellaneous

  If you put symbols other than these the categories field, they will be ignored. If you put no legal symbols in the categories field or do not include this field in your info.rkt file, your package will be categorized as "Miscellaneous."
- The `'can-be-loaded-with` field: If present, the can-be-loaded-with field should be a quoted datum of one of the following forms:

```
can-be-loaded-with ::= 'all
                     | 'none
                     | (list 'all-except 'VER-SPEC ...)
                     | (list 'only 'VER-SPEC ...)
VER-SPEC           ::= Nat
                     | (Nat MINOR)
MINOR              ::= Nat
                     | (Nat Nat)
                     | (= Nat)
                     | (+ Nat)
                     | (- Nat)
```

  where `VER-SPEC` is a PLaneT package version specification in a manner like using `planet` in `require`.

Depending on your package's behavior, it may or may not be okay for multiple versions of the same package to be loaded at one time on the entire system — for instance, if your package relies on writing to a particular file and assumes that nothing else writes to that same file, then multiple versions of the same package being loaded simultaneously may be a problem. This field allows you to specify whether your package can be loaded simultaneously with older versions of itself. If its value is `'all`, then the package may be loaded with any older version. If it is `'none`, then it may not be loaded with older versions at all. If it is `(list 'all-except VER-SPEC ...)` then any package except those that match one of the given `VER-SPEC` forms may be loaded with this package; if it is `(list 'only VER-SPEC ...)` then only packages that match one of the given `VER-SPEC` forms may be loaded with this package.

When checking to see if a package may be loaded, PLaneT compares it to all other currently-loaded instances of the same package with any version: for each comparison, it checks to see if the newer package's can-be-loaded-with field allows the older package to be loaded. If all such comparisons succeed then the new package may be loaded; otherwise PLaneT signals an error.

The default for this field is `'none` as a conservative protection measure. For many packages it is safe to set this field to `'all`.

- The `'homepage` field: If present, the URL field should be a string corresponding to a URL for the package. PLaneT provides this link with the description of your package on the main PLaneT web page.

- The `'primary-file` field: If present, the primary-file field should be a either a string corresponding to the name (without path) of the main Racket source file of your package, or a list of such strings. The PLaneT web page corresponding to this package will present all files listed here as interface files for your package; it will give direct links to each package and a listing of all names provided by the package along with their contracts (if present).

  If you include only a single string, it will be used as the require line printed on your package's page. If you include a list of strings, then the first legal file string in the list will be used.

- The `'required-core-version` field: If present, the required-core-version field should be a string with the same syntax as the output of the `version` function. Defining this field indicates that PLaneT should only allow users of a version of Racket equal to or more recent than the version specified by this field. This allows you finer-grained control of your package's core-language requirements than its inclusion in a particular repository; for instance, setting this field to `"5.1.3"` would cause the PLaneT server not to serve it to Racket v5.1.2 or older clients.

- The `'version` field: If present, the version field should be a string that describes the version number of this code that should be presented to users (e.g., `"0.15 alpha"`). This field does not override or in any way interact with your package's package version number, which is assigned by PLaneT, but may be useful to users.

- The `'repositories` field: If present, the repositories field should be a list consisting of some subset of the strings `"4.x"` and `"3xx"`. The string `"4.x"` indicates that this package should be included in the v4.x repository (which contains packages that are intended to run in Racket and PLT Scheme versions at or above version 4.0, including the 5.0 series), and the string `"3xx"` indicates that the package should be included in the v3xx repository (containing packages intended to run in PLT Scheme versions in the 3xx series). A single package (and a single version of a package) may be included in multiple repositories with the same PLaneT version number.

In addition, PLaneT uses the `raco setup` installer to install packages on client machines, so most fields it looks for can be included with their usual effects. In particular, adding a `'name` field indicates that the Racket files in the package should be compiled during installation; it is a good idea to add it.

An example info.rkt file looks like this:

```
#lang info
(define name "My Application")
(define blurb
  '("My application runs 60% faster on 20% less peanut "
    "butter. It even shows a fancy graphic!"))
(define primary-file "my-app.rkt")
(define categories '(system xml))
```

See §6.4 “`"info.rkt"` File Format” for more information on `"info.rkt"` files.

6.3 Build a Distribution Archive

1. So that the next step can find `for-label` documentation in your own package, first set up a development link (if it is not already set), using

   ```
   raco planet link <owner> pkg.plt> <maj> <min> <path-to-files>
   ```

   This step is not necessary if your package has no documentation.

2. Use the planet command-line tool in its archive-creation mode to create a planet archive:

   ```
   raco planet create /home/jacobm/my-app/
   ```

   This will create a planet archive named `"my-app.plt"` in the current directory whose contents are the contents of `"/home/jacobm/my-app"` and all its subdirectories.

   Alternately, you can run `make-planet-archive` with the name of the directory you’ve prepared as its argument:

   ```
   (make-planet-archive "/home/jacobm/my-app/")
   ```

This function will build a packaged version of your directory and return the path to that package. The path will always be a file named `"X.plt"`, where `"X"` is the name of the directory you gave to `make-planet-archive`, located in that same directory.

3. Remove the development link from the first step (assuming you added one) using

   ```
   raco planet unlink <owner> <packagename.plt> <maj> <min>
   ```

4. Now test that your archive file works as intended using the planet command-line tool in its install mode:

   ```
   raco planet fileinject <owner> <path to .plt file> <maj> <min>
   ```

   installs the specified file into your local PLaneT cache as though it had been downloaded from the PLaneT server with the given owner name and major and minor versions. After you run this command, you can require your package on your local machine using

   ```
   (require (planet <file> (<owner> <.plt file name> <maj> <min>)))
   ```

   to verify everything works.

5. Finally, use

   ```
   raco planet remove <owner> <.plt file name> <maj> <min>
   ```

   to remove the test package from your local cache. (Not removing it is safe as long as you use the same name and version numbers the package will have on the PLaneT server; otherwise you may experience problems.)

6.4 Determine Your Package's Backwards-Compatibility

If you are updating a previously-released package, you must decide whether your package is a backwards-compatible change or not. A rule of thumb is to remember that modules written to work with the previously-released version of your package should unmodified with the new package. This means that at a minimum, a backwards compatible update should:

- Contain all the same Racket source files in that the previous version contained in directories intended for public access
- In each public file, provide at least all the bindings that the previous version provided
- For each name provided with a contract (see §7 "Contracts"), provide it with a contract that is at least as permissive as the previous contract

A backwards-compatible upgrade may, however:

- Change any behavior that reasonable consumers of your package would not consider guaranteed (*e.g.*, by fixing bugs or improving the efficiency of operations).

- Remove files in clearly-marked private sections. By convention, the contents of any directory called `"private"` are considered private and should not be relied upon by external users of your package.
- Extend the set of names exported by a module.

Currently these rules are guidelines only, but in the future some or all of them may be enforced programmatically. Ultimately, though, no technical device can precisely capture what it means for a package to be backwards-compatible with a previous version, so you should use your best judgment.

6.5 Submit Your Package

Go to the central PLaneT package repository web page and click on the link marked "contribute a package / log in" in the upper-right-hand corner. If you have not yet created an account, then do so on that page by providing your name, a user name, an email address, and a password and then responding to the confirmation message delivered to the email address you provide.

Once you have an account, then if this is a new package then upload it using the "Contribute a package" section in your user account page. If this is a package update then click "update this package" next to its name in the "Manage your packages" section of your user account page, then upload the .plt file and indicate on the form whether your update is backwards-compatible with the prior version or not.