PackageManager

Easily download and install GAP packages

1.4.3

12 January 2024

Michael Young

Michael Young
Email: mct25@st-andrews.ac.uk
Homepage: https://mct25.host.cs.st-andrews.ac.uk/
Address: School of Computer Science
University of St Andrews
Jack Cole Building, North Haugh
St Andrews, Fife, KY16 9SX
United Kingdom

Contents

Chapter 1

Introduction

1.1 What does the PackageManager package do?

This package provides the ability to install or remove a package using a single command: `InstallPackage` (2.1.1) or `RemovePackage` (2.2.1). The user can specify a package to install using its name, or using a URL to an archive, a repository, or a `PackageInfo.g` file. When installing, PackageManager also attempts to compile the package, build its documentation if necessary, and ensure that its dependencies are also installed.

1.2 What does the PackageManager package not do?

At present, PackageManager is fairly basic, without many of the advanced features available in package managers such as pip or apt. For instance, the user cannot update all packages in one command. Removing a package will not remove any of its dependencies, since we do not track how packages were installed. When a package is installed, no tests are run to ensure that it is compatible with the installed version of GAP. Any of these features might be added in the future. Other feature requests can be posted on the issue tracker at `https://github.com/gap-packages/PackageManager/issues`.

1.3 A quick example

To install the latest deposited version of the Digraphs package, use the following:

Example

```
gap> LoadPackage("PackageManager");
gap> InstallPackage("digraphs");
```

To uninstall it later, use the following:

Example

```
gap> LoadPackage("PackageManager");
gap> RemovePackage("digraphs");
```

PackageManager also supports version control repositories. To install the latest version of the curlInterface package from GitHub, use the following:

Example

```
gap> LoadPackage("PackageManager");
gap> InstallPackage("https://github.com/gap-packages/curlInterface.git");
```

Chapter 2

Commands

2.1 Installing and updating packages

2.1.1 InstallPackage

▷ `InstallPackage(`*`string[, version][, interactive]`*`)` (function)

Returns: `true` or `false`

Attempts to download and install a package. The argument *`string`* should be a string containing one of the following:

- the name of a package;
- the URL of a package archive, ending in `.tar.gz` or `.tar.bz2`;
- the URL of a git repository, ending in `.git`;
- the URL of a mercurial repository;
- the URL of a valid `PackageInfo.g` file.

The package will then be downloaded and installed, along with any additional packages that are required in order for it to be loaded. Its documentation will also be built if necessary. If this installation is successful, or if this package is already installed, `true` is returned; otherwise, `false` is returned.

By default, packages will be installed in the `pkg` subdirectory of the user's home directory, see `UserHomeExpand` (**Reference: UserHomeExpand**). Note that this location is not the default user pkg location on Mac OSX, but it will be created on any system if not already present. Note also that starting GAP with the `-r` flag will cause all packages in this directory to be ignored.

Certain decisions, such as installing newer versions of packages, will be confirmed by the user via an interactive shell – to avoid this interactivity and use sane defaults instead, the optional argument *`interactive`* can be set to `false`.

To see more information about this process while it is ongoing, see `InfoPackageManager` (2.1.4).

If *`string`* is the name of the package in question then one can specify a required package version via a string as value of the optional argument *`version`*, which is interpreted as described in Section (**Reference: Version Numbers**). In particular, if *`version`* starts with = then the function will try to install exactly the given version, and otherwise it will try to install a version that is not smaller than the given one. If an installed version satisfies the condition on the version then `true` is returned without an attempt to upgrade the package. If the package is not yet installed or if no installed version

satisfies the version condition then an upgrade is tried only if the package version that is listed on the GAP webpages satisfies the condition. (The function will not update a dev version of the package if a version number is prescribed; otherwise it could happen that one updates the installation and afterwards notices that the version condition is still not satisfied.)

Example

```
gap> InstallPackage("digraphs");
true
```

2.1.2 UpdatePackage

▷ `UpdatePackage(`*`name[, interactive]`*`)` (function)

Returns: `true` or `false`

Attempts to update an installed package to the latest version. The first argument *`name`* should be a string specifying the name of a package installed in the user GAP root (for example, one installed using `InstallPackage` (2.1.1)), see (**Reference: GAP Root Directories**). The second argument *`interactive`* is optional, and should be a boolean specifying whether to confirm interactively before any directories are deleted (default value `true`).

If the package was installed via archive, the new version will be installed in a new directory, and the old version will be deleted. If installed via git or mercurial, it will be updated using `git pull` or `hg pull -u`, so long as there are no outstanding changes. If no newer version is available, no changes will be made.

This process will also attempt to fix the package if it is broken, for example if it needs to be recompiled or if one of its dependencies is missing or broken.

Returns `true` if a newer version was installed successfully, or if no newer version is available. Returns `false` otherwise.

Example

```
gap> UpdatePackage("io");
#I  io version 4.6.0 will be installed, replacing 4.5.4
#I  Saved archive to /tmp/tm7r5Ug7/io-4.6.0.tar.gz
Remove old version of io at /home/user/.gap/pkg/io-4.5.4 ? [y/N] y
true
```

2.1.3 CompilePackage

▷ `CompilePackage(`*`name`*`)` (function)

Returns: `true` or `false`

Attempts to compile an installed package. Takes one argument *`name`*, which should be a string specifying the name of a package installed in the user GAP root (for example, one installed using `InstallPackage` (2.1.1)), see (**Reference: GAP Root Directories**). Compilation is done automatically when a package is installed or updated, so in most cases this command is not needed. However, it may sometimes be necessary to recompile some packages if you update or move your GAP installation.

Compilation is done using the `etc/BuildPackages.sh` script bundled with PackageManager. If the specified package does not have a compiled component, this function should have no effect.

Returns `true` if compilation was successful or if no compilation was necessary. Returns `false` otherwise.

Example

```
gap> CompilePackage("orb");
#I  Running compilation script on /home/user/.gap/pkg/orb-4.8.3 ...
true
```

2.1.4 InfoPackageManager

▷ `InfoPackageManager` (info class)

Info class for the PackageManager package. Set this to the following levels for different levels of information:

- 0 - No messages
- 1 - Problems only: messages describing what went wrong, with no messages if an operation is successful
- 2 - Directories and versions: also displays informations about package versions and installation directories
- 3 - Progress: also shows step-by-step progress of operations
- 4 - All: includes extra information such as whether curlInterface is being used

Set this using, for example `SetInfoLevel(InfoPackageManager, 1)`. Default value is 3.

2.1.5 InstallPackageFromName

▷ `InstallPackageFromName(`*`name[, version][, interactive]`*`)` (function)

Returns: `true` or `false`

Attempts to download and install a package given only its name. Returns `false` if something went wrong, and `true` otherwise.

Certain decisions, such as installing newer versions of packages, will be confirmed by the user via an interactive shell – to avoid this interactivity and use sane defaults instead, the optional argument *`interactive`* can be set to `false`.

A required version can also be specified using the optional argument *`version`*. It works as described in the `InstallPackage` (2.1.1) function.

2.1.6 InstallPackageFromInfo

▷ `InstallPackageFromInfo(`*`info`*`)` (function)

Returns: `true` or `false`

Attempts to download and install a package from a valid `PackageInfo.g` file. The argument *`info`* should be either a valid package info record, or a URL that points to a valid `PackageInfo.g` file. Returns `true` if the installation was successful, and `false` otherwise.

2.1.7 InstallPackageFromArchive

▷ `InstallPackageFromArchive(`*`url`*`)` (function)

Returns: `true` or `false`

Attempts to download and install a package from an archive located at the given URL. Returns `true` if the installation was successful, and `false` otherwise.

2.1.8 InstallPackageFromGit

▷ `InstallPackageFromGit(`*`url[, interactive][, branch]`*`)` (function)

Returns: `true` or `false`

Attempts to download and install a package from a git repository located at the given URL. Returns `false` if something went wrong, and `true` otherwise.

If the optional string argument *`branch`* is specified, this function will install the branch with this name. Otherwise, the repository's default branch will be used.

Certain decisions, such as installing newer versions of packages, will be confirmed by the user via an interactive shell – to avoid this interactivity and use sane defaults instead, the optional second argument *`interactive`* can be set to `false`.

2.1.9 InstallPackageFromHg

▷ `InstallPackageFromHg(`*`url[, interactive][, branch]`*`)` (function)

Returns: `true` or `false`

Attempts to download and install a package from a Mercurial repository located at the given URL. Returns `false` if something went wrong, and `true` otherwise.

If the optional string argument *`branch`* is specified, this function will install the branch with this name. Otherwise, the repository's default branch will be used.

Certain decisions, such as installing newer versions of packages, will be confirmed by the user via an interactive shell – to avoid this interactivity and use sane defaults instead, the optional second argument *`interactive`* can be set to `false`.

2.1.10 InstallRequiredPackages

▷ `InstallRequiredPackages()` (function)

Returns: `true` or `false`

Attempts to download and install the latest versions of all packages required for GAP to run. Currently these packages are GAPDoc, primgrp, SmallGrp, and transgrp. Returns `false` if something went wrong, and `true` otherwise.

Clearly, since these packages are required for GAP to run, they must be loaded before this function can be executed. However, this function installs the packages in the `~/.gap/pkg` directory, so that they can be managed by PackageManager in the future, and are available for other GAP installations on the machine.

2.2 Removing packages

2.2.1 RemovePackage

▷ RemovePackage(*name[, interactive]*) (function)

Returns: `true` or `false`

Attempts to remove an installed package using its name. The first argument *name* should be a string specifying the name of a package installed in the user GAP root, see (**Reference: GAP Root Directories**). The second argument *interactive* is optional, and should be a boolean specifying whether to confirm certain decisions interactively (default value `true`).

Returns `true` if the removal was successful, and `false` otherwise.

Example

```
gap> RemovePackage("digraphs");
Really delete directory /home/user/.gap/pkg/digraphs-0.13.0 ? [y/N] y
true
```

Index