PackageManager

Easily download and install GAP packages

2.0.0

19 November 2026

Michael Young

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

Contents

1 PackageManager **3**
- 1.1 Basic usage . . . 3
- 1.2 Changes in version 2.0 . . . 4
- 1.3 Commands . . . 4
- 1.4 Directories . . . 8
- 1.5 User Preferences . . . 8
- 1.6 Info Warnings . . . 9

Index **10**

Chapter 1

PackageManager

PackageManager allows you to install and remove GAP packages on your system quickly and easily with a single command. The main functionality is provided by the following commands:

- `InstallPackage` (1.3.1) to install a package by its name, or by a URL to an archive, Git repository or package info file.
- `RemovePackage` (1.3.2) to remove a package by its name.
- `InstallRequiredPackages` (1.3.4) to install all the packages that GAP needs in order to run.

Compiling the package, building the documentation, and installing dependencies are all handled automatically.

The behaviour of PackageManager can be customised in various ways by setting appropriate user preferences, as described in Section 1.5. These can also be overridden for a single call by adding an optional argument *prefs* to the appropriate function call, as described in the documentation for the `InstallPackage` (1.3.1) command.

By default, packages will be installed in the `pkg` subdirectory of `GAPInfo.UserGapRoot`. To learn more, see Section 1.4.

1.1 Basic usage

PackageManager is distributed with GAP, so it should already be available on your system. If it is not already loaded, you can call it with:

Example

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

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

Example

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

To uninstall it later, use the following:

Example

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

1.2 Changes in version 2.0

PackageManager 2.0 introduced a range of new features, including the ability to download the structured info for the whole package distribution, and plan dependency trees in advance before installation begins. It also introduced a range of user preferences, which are described under the `InstallPackage` (1.3.1) command below.

The following patterns are now deprecated, and may stop working in a future release:

- `InstallPackage(name, "mybranch")` where the second argument is the name of a branch to check out from a Git repo. Instead you can use `InstallPackage(name, rec(branch := "mybranch"))` to specify a branch.
- `InstallPackage(name, false)` where the second argument is a boolean that can be set to `false` to disable interactivity. Instead you can use `InstallPackage(name, rec(interactive := false))` to disable interactivity for this call, or set the behaviour globally with `SetUserPreference("PackageManager", "interactive", false)`.
- `InstallPackage(name : keepDirectory)` where the `keepDirectory` option would prevent directories from being deleted if installation fails. This is now the default behaviour, so no option is needed.

1.3 Commands

1.3.1 InstallPackage (for a string)

▷ `InstallPackage(`*`string[, prefs]`*`)` (operation)

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, usually ending in `.git`;
- 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 the installation is completed successfully, or if this package is already installed, `true` is returned; otherwise, `false` is returned.

Example

```
gap> InstallPackage("digraphs");
#I  The following packages will be installed:
#I    Digraphs        1.15.0
#I    IO              4.11.0
#I    orb             5.1.0
#I    datastructures  0.4.3
Continue? [Y/n] y
true
```

This command's behaviour is affected by the user preferences that have been set for the PackageManager package. These can be updated using, for example, `SetUserPreference("PackageManager", "compile", true);` but these can also be overridden for this call by using the optional second argument *prefs* for this function, for example `InstallPackage(name, rec(compile := false));`

The available preferences are as follows:

- `dependencies` - whether to install the package's dependencies as well
- `suggested` - whether to include suggested as well as required dependencies
- `compile` - whether to attempt to compile newly installed packages
- `compileDeps` - whether to attempt to compile the dependencies of newly installed packages
- `upgrade` - whether to upgrade already installed packages if an upgrade is available
- `git` - whether to attempt to pull any checked out Git repos when upgrading
- `proceed` - whether to proceed with the installation at all (after printing the installation plan)
- `interactive` - set this to `false` to disable all interactive prompting (preferences set to "ask" are instead treated as `true`)
- `distroLocation` - the URL to download the GAP package distribution info from
- `distroVersion` - the version of the package distribution info to use (choose a version of GAP, or just "latest")
- `version` - the version of a particular package to attempt to install
- `branch` - the name of the branch to attempt to install when cloning a Git repository (or `fail` for the default branch)

All boolean preferences above (those described as "whether to...") can be set to `true`, `false`, or "ask" which will prompt the user interactively. This is the default setting for many of them.

Example

```
gap> InstallPackage("digraphs", rec(interactive := false,
>                                   suggested := true,
>                                   compileDeps := false));
#I  The following packages will be installed:
#I    Digraphs        1.15.0
#I    IO              4.11.0
```

```
#I    orb             5.1.0
#I    datastructures  0.4.3
#I    GRAPE           4.9.3
#I    AutoDoc         2026.09.09
true
```

To see more information about the install process while it is ongoing, see `InfoPackageManager` (1.6.1).

If the `version` preference is specified, then it is interpreted as described in Section (**Reference: Version Numbers**). In particular, the value will be treated as a minimum version number unless it starts with =.

If installation fails, then any new directories that were created will be preserved. To remove them, the `RemovePackage` (1.3.2) operation can be used.

1.3.2 RemovePackage (for a string)

▷ `RemovePackage(`*`name[, prefs]`*`)` (operation)

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**)).

This command's behaviour is affected by the user preferences that have been set for the PackageManager package. These can be set globally, or overridden for this command using the optional *`prefs`* argument, in the same way as for the `InstallPackage` (1.3.1) command.

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

Example

```
gap> RemovePackage("digraphs");
Delete directory /home/mtorpey/.gap/pkg/digraphs-1.15.0/ ? [Y/n] y
#I  Removed directory /home/mtorpey/.gap/pkg/digraphs-1.15.0/
true
```

1.3.3 CompilePackage (for a string)

▷ `CompilePackage(`*`name[, prefs]`*`)` (operation)

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` (1.3.1)). 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.

This command's behaviour is affected by the user preferences that have been set for the PackageManager package. These can be set globally, or overridden for this command using the optional *`prefs`* argument, in the same way as for the `InstallPackage` (1.3.1) command.

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

Example

```
gap> CompilePackage("digraphs");
#I  Running compilation script on /home/mtorpey/.gap/pkg/datastructures-0.4.3/ ...
#I  Running compilation script on /home/mtorpey/.gap/pkg/orb-5.1.0/ ...
#I  Running compilation script on /home/mtorpey/.gap/pkg/io-4.11.0/ ...
#I  Running compilation script on /home/mtorpey/.gap/pkg/digraphs-1.15.0/ ...
true
gap> CompilePackage("digraphs", rec(compileDeps := false));
#I  Running compilation script on /home/mtorpey/.gap/pkg/digraphs-1.15.0/ ...
true
```

1.3.4 InstallRequiredPackages (with no required arguments)

▷ `InstallRequiredPackages(`*`[prefs]`*`)` (operation)

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, PerfGrp, PrimGrp, SmallGrp, and TransGrp. Returns `false` if something went wrong, and `true` otherwise.

If these required packages are not currently installed, GAP can still be loaded with the `--bare` option, which is sufficient to allow PackageManager to be loaded and this command to be run.

This command's behaviour is affected by the user preferences that have been set for the PackageManager package. These can be set globally, or overridden for this command using the optional *`prefs`* argument, in the same way as for the `InstallPackage` (1.3.1) command.

1.3.5 UpdateAllPackages (with no required arguments)

▷ `UpdateAllPackages(`*`[prefs]`*`)` (operation)

Returns: `true` or `false`

Attempts to upgrade all packages to their latest version. Returns `false` if something went wrong, and `true` otherwise.

Only packages in the directory managed by PackageManager will be considered. See Section 1.4 for more details.

This command's behaviour is affected by the user preferences that have been set for the PackageManager package. These can be set globally, or overridden for this command using the optional *`prefs`* argument, in the same way as for the `InstallPackage` (1.3.1) command.

1.3.6 RefreshPackageMetadata (with no required arguments)

▷ `RefreshPackageMetadata(`*`[prefs]`*`)` (operation)

Returns: `true` or `false`

Attempts to download metadata for all available packages from the GAP package distribution.

Calling this function is not usually necessary, since the metadata is downloaded automatically the first time `InstallPackage` (1.3.1) is called, and the result is cached for the rest of the GAP session. However, this command can be called to force the download and retrieve any updated information, which may be useful, for example, if the `distroLocation` or `distroVersion` user preferences have been changed, or if new package versions may have been added to the distribution online since this GAP session started.

Returns `false` if something went wrong, and `true` otherwise.

This command's behaviour is affected by the user preferences that have been set for the PackageManager package. These can be set globally, or overridden for this command using the optional *prefs* argument, in the same way as for the InstallPackage (1.3.1) command.

1.4 Directories

By default, packages will be installed in the pkg subdirectory of GAPInfo.UserGapRoot (see (**Reference: GAP Root Directories**)). Note that this location is not the default user pkg location on MacOS, 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.

To manage a different directory, the following functions are available.

1.4.1 DirectoryPackageManager

▷ DirectoryPackageManager() (function)

Returns: a directory object

Returns the directory that is currently being used by PackageManager for installing and removing packages. This is the only directory in the file system where PackageManager will add, delete or modify any files. It can be modified using ChangeDirectoryPackageManager (1.4.2).

1.4.2 ChangeDirectoryPackageManager

▷ ChangeDirectoryPackageManager(*path*) (function)

Returns: true or fail

Changes the directory used by PackageManager for installing and removing packages. See DirectoryPackageManager (1.4.1).

The argument *path* should be a string that represents a path to a directory. It will be created if it does not exist.

Returns true on success and fail on failure.

1.5 User Preferences

PackageManager exposes several user preferences which can be set to customise the behaviour of the package globally. For example, calling

Example

```
gap> SetUserPreference("PackageManager", "compile", false);
```

will make it so that new packages are not compiled automatically when InstallPackage is called for the rest of the session. You could instead specify this as a one-off by passing a record to the command, as follows:

Example

```
gap> InstallPackage("digraphs", rec(compile := false));
```

You can make these preferences persist between sessions by calling WriteGapIniFile (**Reference: WriteGapIniFile**) and updating the settings in the gap.ini file that is produced.

A full list of user preferences recognised by the package is described in the documentation for InstallPackage (1.3.1).

1.6 Info Warnings

1.6.1 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 information about installation directories and package versions
- 3 - Progress: also shows step-by-step progress of operations
- 4 - All: includes extra information such as whether curlInterface is being used, package info validation and compilation output

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

Index

ChangeDirectoryPackageManager, 8
CompilePackage
    for a string, 6

DirectoryPackageManager, 8

InfoPackageManager, 9
InstallPackage
    for a string, 4
InstallRequiredPackages
    with no required arguments, 7

RefreshPackageMetadata
    with no required arguments, 7
RemovePackage
    for a string, 6

UpdateAllPackages
    with no required arguments, 7