Ape user manual and API documentation

Release 1.00dev

Lukas Nellen

February 27, 2017

CHAPTER ONE

APE USER GUIDE

Note: update notice

The conventions for package names used in ape got changed to all lower-case. If you have used and configured ape for older versions, you might have to change the package names used in your configuration files.

1.1 Getting started

1.1.1 Dowloading ape

To download ape, visit the ape trac pages.

You can always get the current development version of ape from the repository. The instructions for downloading are on the ape trac pages.

1.1.2 Prerequisites for using ape

To be able to use ape, you have to have some packages installed on your system. You need python and a C++-compiler as well as some development libraries not yet provided by ape. Please consult the ape trac pages for more information.

The required packages are readily available as part of all linux distributions. For the Mac, you have to install XCode. It comes on DVD with OS X.

Please consult the documentation for *Build support for mysql connector package* for considerations about MySQL.

1.1.3 Simple use

It is possible to use ape with an empty configuration file, without command line options. In this case, ape will install the packages into the directory ape resides in by setting the *base* there.

```
# Create empty ape config - not the recommended way of using ape!
touch $HOME/.aperc
```

To download and install the current offline software and all packages required to built it, invoke:

```
ape install offline
```

If you only want the external packages required to build the offline software, call:

```
ape install externals
```

Note: Depending on the setting of your `PATH`, you might have to specify `./ape` to invoke `ape`.

To install to a different location, you can set the *base* variable either by specifying the option `-bpath` or by providing a simple configuration file. The latter is the recommended way. For a personal installation, create `~/.aperc`. If you prepare a site-wide installation, you should provide `ape.rc` in a central location and set the `APERC` environment variable. The way to set environment variables at session start is system dependent. Please consult with your system administrator about the means provided to configure the environment for all users or for certain groups. A typical configuration file will be very short. Here is a simple example:

```
# A simple ~/.aperc
[DEFAULT]
base = %(home)s/auger/software/ApeInstalled

[ape]
jobs = 2
mirrors = mx us
```

The variables set here are:

base The path to install all packages to. The hierarchy below is determined by the configuration of the individual packages installed. The example sets it relative to the user's `HOME` environment variable.

It is generally recommended to provide at least this variable in your configuration file to decouple the installation area from the `ape` code area.

NB: The syntax used for variable substitution is Python's string interpolation syntax: `%(var)s`. Don't forget the trailing `s`. It is required and will not appear in the result.

jobs The number of jobs to use for those packages which support parallel builds. The default value is 1. Set this to the number of cores in the machine your running `ape` on.

mirrors The preferred mirror list. The example selects the Mexican mirror ahead of the US mirror, followed by the remaining mirrors known to ape. Setting this variable to a reasonable value can reduce the time required for downloading the packages significantly. Use `ape mirrors` to get the list of mirrors.

On a cluster with a shared home area, you might want to build on a local disk, for example in the `\tmp` area. Your configuration file for compiling on a cluster with multi-core servers might be:

```
# A simple ~/.aperc
[DEFAULT]
base = %(home)s/auger/software/ApeInstalled
build = /tmp/ape-build-%(user)s

[ape]
jobs = 10
mirrors = mx us
```

1.1.4 Setting up your environment

The sub-commands **sh** and **csh** dump shell instructions to set the environment variables needed to use all or some of the packages installed by `ape`.

It is possible to install several independent packages with `ape`. They can require different, sometimes incompatible, configurations of the environment. You should specify the name of the target package(s) as argument(s) to the **sh** and

csh subcommands of `ape`. For example, to set up the environment for the use of the Auger offline package, specify:

```
# sh/bash/ksh
eval `ape sh offline`

# {t}csh
eval `ape csh offline`
```

Dependencies are tracked. If a package depends on other packages, the environment is also set up for the dependencies. Typical Auger users will use the configuration for offline, as shown in the example. Developers should use `Externals` as the target for their environment. You can also set up a customized environment for running the adst package if you use `ape` to install it.

You can try to set up the environment for all installed packages by no providing the name of a target package.

```
# sh/bash/ksh
eval `ape sh`

# {t}csh
eval `ape csh`
```

Note: This form of invoking the **sh** and **csh** sub-commands is more fragile than targeting a specific package. It will probably get removed in a future release of `ape`.

More information can be found in the section *The ape configuration files*. You can specify more than one package name as an argument to **sh** and **csh**.

1.2 Details of the `ape` command

The syntax of the `ape` command is

```
ape [options...] verb [packages...]
```

The `verb` indicates the action to be taken.

1.2.1 Verbs for installation

install Installs the selected package(s) and their dependencies. If the option *`--jobs`* is given, building will be done in parallel when possible.

fetch Download the source archives for the packages listed on the command line. Unless the option *`--no-dependencies`* is given, this will also download the source archives of all dependencies.

unpack Unpack the source archives for the packages listed on the command line. Unless the option *`--no-dependencies`* is given, this will also unpack the source archives of all dependencies.

clean Clean the build area for all installed packages. This way, one can remove the build directories which preserved due to the *`--keep`* option or due to some configuration settings.

> **Note:** There is no way to force the removal of a build directory which is preserved due to the conditions listed in *`ApeTools.Build.Package.clean()`*.

1.2.2 Verbs to provide information

installed List the installed packages.

mirrors List the mirror sites known to `ape`. This list will be ordered as specified in the selection of preferred mirrors, e.g., by using the *`--mirrors`* option.

packages List all packages known to `ape`. The dependencies are also liste, unless suppressed using the option *`--no-dependencies`*. If running in *`--verbose`* mode, the version number are also listed.

package-config Dump the configuration data for the listed packages. If no package is listed, dump the information for all known packages.

ape-config Dump the global configuration of `ape`.

dump-config Write the information provided by **package-config** and **ape-config** into a log-file name `ape-dump.apelog` in the *logs* directory.

1.2.3 Verbs to propagate information

sh Dump a set of commands to configure the user's environment as specified by the listed packages. If no package name(s) given, produce an environment for the use of all installed packages.This verb provides commands for a shell of the *Bourne-Shell* family of shells.

The commands emitted for path-like variables, e.g., `PATH` or `LD_LIBRARY_PATH` take into account if the variables exist already and emit the correct code for immediate use. No conditional shell code is produced.

csh Dump a set of commands to configure the user's environment as specified by the listed packages. If no package name(s) given, produce an environment for the use of all installed packages.This verb provides commands for a shell of the *C-Shell* family of shells.

The commands emitted for path-like variables, e.g., `PATH` or `LD_LIBRARY_PATH` take into account if the variables exist already and emit the correct code for immediate use. No conditional shell code is produced.

1.2.4 Options

Options can be used to modify details of the action or output when running `ape`.

`--base``=<path>`, **`-b`** `<path>`
Specify the base of the tree to install the packages into. It is preferred, though, to set it in a *configuration file*.

By default, `ape` installs into the current directory.

`--jobs``=<jobs>`, **`-j`** `<jobs>`
Set the number of *jobs* for building in parallel. Some packages cannot be built in parallel. This option will be ignored in that case.

`--verbose`, **`-v`**
Provide more information.

`--dry-run`, **`-n`**
Do not actually download, unpack, build, or install anything. This provides the user with a way to see what `ape` intends to do.

`--keep`, **`-k`**
Keep the directories where the packages where build. In general, `ape` will remove the build area, since it is no longer needed after a successful install.

--force, -f
Force `ape` to perform the selection on the packages listed on the command line. This can be used, e.g., to re-install a package.

--force-all
Like *--force*, but applies also to all dependencies, not just the packages listed on the command line.

--no-dependencies, -x
Ignore the dependencies of a package. This option can be useful for fetching exactly the source files for selected packages, or to reduce the output when listing the packages known to `ape`. It should not be used during an installation, or you might end up with an inconsistent set-up.

--mirrors, -m
Provide a colon-separated list of mirror sites which should be tried first when fetching package sources.

--debug, -d
Provide additional debugging information.

--rc=<file>
Specifies an additional configuration file to be read.

Note: This should be used only for debugging, in scripts or for one-off installations. Otherwise, the results might be hard to reproduce. This is especially true if you intend to use the *Verbs to propagate information* to set up your environment.

--pw-file=<file>
Specifies the file which contains user/password pairs for fetching externals. For more details, including the format of the file, see the section on *Using a password file*.

Note: This suppresses interactive prompting for a password.

--option=<assignment>, **-o** <assignment>
Set a configuration variable directly from the command line. For example, to provide a list of preferred mirrors, you could use any of the following:

- Provide the information as an option:

```
ape --mirrors=mx:us fetch externals
```

- Set the *ape.mirrors* configuration variable on the command line:

```
ape --option='ape.mirrors = mx us' fetch externals
```

- Provide the information in a `.rc` file:

```
# part of ~/.aperc
[ape]
mirrors = mx us
```

And run `ape` without any options:

```
ape fetch externals
```

1.3 Troubleshooting

The output of all commands executed while building the packages is collected in a single directory *logs*. Unless set explicitly, this directory is below the *base* of the installation. There is a separate log-file for each stage of building and installing a package. All log-files have the extension `.apelog`. In case of failure, the final error message will also be logged in a file named `ape-failure.apelog`. The information in this file indicates the stage where `ape` encountered a problem and an indication of what failed. The first two lines of a log-file indicate the host where the command was executed and the command being run:

```
ape >>> on host firefly.lan
ape >>> make
```

This extra information can help to reconstruct what went wrong. Knowing the host can be important to diagnose spurious problems in a heterogeneous cluster.

If you cannot solve the problem, consult with the experts on the user and developer mailing lists. You might get asked to provide some or all of the files in the *logs* directory.

If you think you found a bug in `ape`, please report it at the ape trac pages.

1.4 The `ape` configuration files

Most data needed by `ape` is provided in configuration files and not in the code itself. In this section, we deal with the customization of `ape` for individual and site set-ups. Advanced issues, including teaching `ape` about new packages, get discussed in the section *Extending ape*. The section *Simple use* contains a *simple configuration example*.

1.4.1 Format of configuration files

The configuration files are in INI-format, which can be handled by Python's `ConfigParser` modules. This format looks like:

```
# a comment
[section]
var = value
another.var = yet another value

[section 2]
# var has a different value in this section
var = value2
# the value of var.extendend is "value2 and more".
var.extended = %(var)s and more
```

A variable is identified by the section and a name within a section. By convention, `ape` does not allow `.` in section names, only in variable names. Values of variables can be substituted into other variables, using the Pythonic notation `%(`*var*`)s`. This is used extensively by `ape` to build up path names.

Comments are allowed and can be introduced with `#` or `;`.

1.4.2 Variable types

Different variable types are supported in configuration files:

string An arbitrary string value. Leading white-space is stripped.

integer A string which can be converted to an integer.

boolean A string which can be converted to a boolean. The values `on`, `yes`, `1`, and `true` indicate a `True` value. The values `off`, `no`, `0`, and `false` indicate a `False` value. The conversion is case-insensitive.

list of strings A white-space separated list of strings. The strings cannot contain any white-space.

To make customization simpler, a variable *var* has the following variables associated:

var.delete A list of items to remove from *var*

var.prepend A list of items to prepend to *var*

var.append A list of items to append to *var*

Note: These customization variables are reserved for the user or local administrator and cannot be used in the configuration files supplied with `ape`.

1.4.3 Section names

The sub-components of `ape` look in different sections for configuration variables. Some sections provide default values for other sections.

`[DEFAULT]` This is the global default section. Variables set in this section will appear visible in **all** other sections. A few variables are provided automatically by `ape`: *home*, *user*, *apeDir*, *base*. The build directory *build* is also set in this section. Do not add variables to this section unless you are 200% sure of what you are doing.

Most likely, you are going to change *base*. On clusters, where *base* is on a remote file-system, you might set *build* to a path on a local disk.

```
[DEFAULT]
base = /shared/ApeInstalled
build = /local/build/path
```

`[ape]` This section contains the global variables for `ape`. The variables you are likely to set are *jobs* and *mirrors*.

```
[ape]
# list of tags for mirrors in order of preference
mirrors = mx us es
# number of processed for parallel builds
jobs = 4
```

Additionally, you can add variables *env.var* to this section to set environment variables:

```
[ape]
# force packages which look at the environment variables CC and CXX
# to use the gnu compilers.
env.cc = gcc
env.cxx = g++
```

`[ape internal]` **Do not touch.**

`[package]` Default values of most variables used to configure the building of packages.

`[package` name`]` Variables to configure the package called **name**. `Ape` determines the set of known packages by looking for all sections who's names start with `package` followed by a package name.

`[mirror]` Default values for setting up data mirror sites.

```
    [mirror]
    user = cuyam
```

[mirror tag] Defines the variables for a mirror identified by **tag**. The configuration for the *us* mirror is:

```
    [mirror us]
    location = Northeastern University, Boston, USA
    url = http://129.10.132.228/apeDist
```

A site might want to set up a local mirror and make it the preferred mirror to use.

1.4.4 Using a password file

Note: Storing a password in clear text in a file is in general a security risk and should be avoided. This feature is primarily intended to support frequent, automated installations for test purposes.

You can store the download password(s) in a file, which can be specified either in the configuration file or using a the command line option *--pw-file*. To specify the file in your configuration file, you have to set the `pwFile` variable it in the `[ape]` section of the file:

```
[ape]
 # other options ...
 # password file is .apepw in my home directory
 pwFile = %(home)s/.apepw
```

The file contains one or more lines with user/password pairs:

```
user 12345
luser qwerty
noone password
```

The file has to be protected to be accessible only by the user, without execute permissions i.e., file mode 0400 or 0600. The presence of a password file will suppress the interactive prompt.

1.5 Customizing packages

A few packages have variables which can be customized.

1.5.1 Disabling the installation of a package

You might want to avoid the installation of a package by `ape`, e.g., because you prefer to use a version of the package provided by your system.

Note: Don't do this unless you are know what you are doing and you are absolutely convinced that this is the right thing to do.

You hide a package from the installation by setting its attribute `isHidden`:

```
[package excludeme]
isHidden = True
```

The package will be excluded from the list of known packages when no package is requested explicitly and from all dependencies of other packages Operations line installation and the emission of the environment will ignore the package. Some some operations of the package a still possible, though. For example, information about the package can still be interrogated, which means `ape package-config excludeme` will always provide a dump of the important attributes of the package.

Note: It is **strongly** recommended that you remove your existing installation and re-install if you change this flag for *any* package.

Disabling by removing a package from dependencies

Trying to prevent the installation of a package by removing it from the `dependencies` and `weakDependencies` of *all* other packages is possible, but not reliable. You can miss dependencies and dependencies in the default configuration are likely to change in the future.

Disabling by removing sources and environment

An alternative is to turn a package into a *dummy* by clearing the list of sources and environment variables for that package:

```
    [package excludeme]
    tarballs =
    extraFiles =
    environment =
```

This declares that there are no files and no environment variables for the package. As a consequence, there is nothing to do for this package. Its dependencies are still processed, unless you remove those, too.

1.5.2 Customizing aires

You can select the Fortran compiler used for building aires by setting the *fc* variable.

```
[package aires]
# use g77 instead of gfortran, needed if you have the 3.x GNU compilers
fc = g77
```

1.5.3 Customizing boost

You might want to change the individual libraries built as part of your installation of boost. You can do this by adding and removing options from the *configureArgs* variable.

```
[package boost]
configureArgs.delete = --without-python
configureArgs.append = --without-iostreams
```

1.5.4 Customizing root

The configuration script of root takes a large number of options to select which sub-systems to include and exclude in a build. The options, without the strings `--enable-` or `--disable-`, are listed in the string lists *enable* and *disable*.

Assuming you would like to disable Python and include Ruby (if possible), you specify:

```
[package root]
disable.delete = ruby
disable.append = python
```

1.5.5 Customizing `externals`

You can customize the list of dependencies of the `externals` package. By default, this package depends on all required and optional packages the Auger Offline software depends on. Assuming you would like to exclude aires, you specify:

```
[package externals]
dependencies.delete = aires
```

1.5.6 Selecting a non-standard compiler

The `gccselect` package

The `gccselect` package allows the user to override the compilers used. It provides a `bin` directory with the standard names of compilers symbolically linked to names with version suffixes and pre-pending this to the `PATH` environment variable.

To use the package, the user has to insert the following lines into the personal ape configuration in `~/.aperc` or equivalent locations, as documented earlier:

```
[DEFAULT]
# suffix to identify versioned compilers
gccsuffix = -4.8

[ape]
# prepend the next package to all dependencies
platformPreDependencies = gccselect
```

This mechanism was originally added to be able to use the gcc compiler suite on OS X.

CHAPTER

TWO

EXTENDING `APE`

2.1 Simple packages

2.2 Customizing `ApeTools.Build.Package`

CHAPTER

THREE

CODE DOCUMENTATION

3.1 Support utilities: the `ApeTools` package and sub-modules

The *`ApeTools`* package and its sub-modules provide utilities to handle the installation of the Auger Offline software.

exception `ApeTools.`**`InstallError`**(*args=[]*, *cwd=None*, *pack=None*, *stage=None*, *log=None*, *cmd=[]*)

The generic installation error exception.

The string representation of *`InstallError`* instances generates an error report.

This class is the basis for all other exceptions defined in ape. It is caught at the top-level to generate an error report to the user before aborting execution.

3.1.1 Sub-Module `Version`

This module provides the version information of `ape`. It is used in python code and, as a tool, in Makefiles.

It provides two variables with the version and release number of `ape`.

If run as a script, it prints *`fullVersion`*. If run with the option `--short`, it prints *`version`*.

`ApeTools.Version.`**`version`**

The ape major and minor version, without release number.

Type string, without spaces.

`ApeTools.Version.`**`fullVersion`**

The full `ape` version, including patch number and possible dev/alpha/beta/rc suffixes. It has to start with *`version`*.

Type string, without spaces.

3.1.2 Sub-module `Build`

This module provides the general tools for package building, dependency analysis, and a registry of known packages.

exception `ApeTools.Build.`**`CircularDependencyError`**(*name*)

Exception thrown when a circular dependency is detected.

class `ApeTools.Build.`**`DependencyNode`**(*package*, *strong*)

A node in the dependency tree of packages.

We separate *strong* and *weak* dependencies. *Weak* dependencies are only for ordering the build process, but do not indicate an obligatory dependency.

Parameters

- **`package`** – An instance of *`ApeTools.Build.Package`*
- **`strong`** – A flag indicating if this node marks a strong or weak dependency

`add`(*package*, *strong*)
Add *package* to the dependencies of this node.

Parameters

- **`package`** – Package this node depends on
- **`strong`** – Flag indicating if the dependency is strong or weak

`flatten`(*flatTree*)
Flatten the sub-tree rooted at this node. Append itself after the child nodes.

Parameters **`flatTree`** – The flattened tree to append to.

`getPackage`()
Return the package represented by this node.

`isStrong`()
Returns flag if this node indicates a strong or weak dependency.

class `ApeTools.Build.`**`DependencyTree`**(*packages*)
A specialization of *`DependencyNode`*, representing the root of the dependency tree.

Packages listed in the configuration variable *preDependencies* are automatically prepended. This allows the injection of packages at the head of the dependency list.

Parameters **`packages`** – The initial list of packages requested. These are automatically strong dependencies. This ensures that they and all their dependencies will be built.

`buildList`()
Get list of packages to build.

Flattens the list and returns a list of packages marked as strong somewhere in the tree.

class `ApeTools.Build.`**`Package`**(*name*)
Base class to provide default implementations for all steps of the fetching and building process of a package.

For simple packages, the process can be customized using configuration files. For packages requiring a more complicated build process, you can derive your installer from this class and re-implement some methods.

The dependencies of a package are listed and have to be build before its dependents. Additionally, there are *weak* dependencies, which are only built if requested. But if requested, they ought to be build before packages depending weakly on them. Full dependencies activate packages. Both types of dependencies are used to determine the order in which to build packages.

Parameters **`name`** – Used to set the attribute *`name`*

`name`
The name of the package. This is used to refer to the package on the command line and to retrieve configuration information.

`build`(*env*)
Build the package by invoking the individual stages.

1. patch
2. configure
3. make

4.install

5.clean

The process can be customized by setting configuration variables or by replacing this method or the methods implementing one of the stages.

clean()

This method removes the source and build directories, unless one of the following conditions is met:

- The package was built outside the general build area. The condition is important to avoid removing packages which do an in-place build directly in the install area.
- The configuration variable *keep* is set in section `[ape]`.
- The option *--keep* was given on the command line. This works by setting the configuration variable *keep* is set in section `[ape]`.
- The package configuration variable *alwaysKeep* is set for this package. This is set in the configuration section for the current package.

```
[package myPackageName]
alwaysKeep = yes
```

configure(*logFile*, *env*)

Configure the current package. This will typically call a **./configure** script or use **cmake**.

dependants()

env()

The additions to the environment for this package.

The environment can be set using configuration variables or by overriding this function.

fetch(*manager*)

This function tries fetch all files specified in `tarballs` and `extraFiles`, using services from *ApeTools.Fetch.DownloadManager*.

info()

Returns a list of variables and their values for this package.

install(*logFile*, *env*)

Install the package. This is typically done invoking **make** with the argument `install`.

isDummy()

Is this package a dummy package? Dummy packages do not require any files. They are typically used to collect dependencies in one place.

isInstalled()

Evaluates if this package is already installed.

A regular package is installed if the top-level installation directory `prefix` exists.

A dummy package has no files to install. It is installed if all dependencies are installed.

logFile(*stageNumber*, *stageName*)

Determine the name of the logfile to use.

Parameters

- **stageNumber** – The number of the stage we are currently processing
- **stageName** – The name of the current processing stage

make (*logFile*, *env*)
Build the package. This is typically done invoking **make**.

patch (*logFile*)
Apply the patches for this package.

removePrefixDir ()
Remove installation directory for this package.

- Use when building or installing fail
- Only removes directories in ape's install area (protection against configuration errors)

setAttributes (*names*, *getter=<function get>*)
Set instance attributes from configuration variables.

Parameters

- **names** (`list or string`) – List of attribute names
- **getter** – One of the `get...` functions from the `ApeTools.Config` modules. Defaults to the string getter.

setDependencyActive ()
Mark package as visited already during dependency checking.

Raise `CircularDependencyError` exception if the package is already in the current branch of the dependency tree.

setDependencyInactive ()
Set package to be not active during dependency checking.

unpack (*parentDir=None*)
This function assumes the tarballs listed in `tarballs` are located in the distfile Directory and unpacks them.

`ApeTools.Build.`**call** (*args*, *cwd=None*, *output=None*, *env=None*, *append=False*, *package=None*, *stage=None*, *useException=True*)
Wrapper to execute command in sub-process.

We always tie `stdout` and `stderr` together for logging, so we have only one output argument.

Parameters

- **args** (`list or string`) – Command to execute
- **cwd** – Directory to execute the command in
- **output** – Name of output file. Write to `sys.stdout` if not set.
- **env** – Enviroment to execute command in
- **append** – Flag to determine if to overwrite or append to output file
- **package** – Name of package being worked on. Leave empty for global processing.
- **stage** – Processing stage
- **useException** – Flag to determine if to report failure by throwing a `ApeTools.InstallError` or by returning a non-zero return code.

Raises `ApeTools.InstallError` is used to report the case when the return code of the command is not 0. This can be suppressed setting *useException* to `False`.

Returns The return code of the command executed. A non-zero result will only be returned when *useException* is set to `False`.

`ApeTools.Build.`**`getPackage`**`(`*name*`)`

Get an instance of `Package` or a sub-class. The class is meant to handle the building of *name*.

This function returns an existing instance if possible. Otherwise, it tries to create a new one. The instance is created calling a callable object registered with, `registerPackage()`, typically a sub-class of `Package`. The name used to locate the creator is given by the *builder* configuration variable for the package *name*.

If *name* is not a string, we assume it is already a package and return it as is.

Raises `ApeTools.InstallError` if the package name or the associated builder are unkown.

`ApeTools.Build.`**`getPackageNames`**`()`

Returns the list of all package names configured, except hidden packages.

`ApeTools.Build.`**`getPackages`**`(`**args*`)`

Returns instances for all packages requested. If no argument is passed, return instances of all configured packages.

Parameters **`args`** – The list of packages requested as individual arguments.

class `ApeTools.Build.`**`package_meta`**

Metaclass for packages. Takes care of the registration of the package builder.

`ApeTools.Build.`**`registerPackage`**`(`*name*, *creator*`)`

Register a callable object

Parameters

- **`name`** – Package name
- **`creator`** – Callable object to create *name*

`ApeTools.Build.`**`untar`**`(`*arName*, *outDir*`)`

Untar an archive by wrapping system calls to tar (for old pythons)

Parameters

- **`arName`** – name of the tarball to be extracted
- **`outDir`** – location of the extracted output

3.1.3 Sub-module `Config`

Configuration management for ape.

Provides the tools to read the information from the `ape.rc` files. The format of the `.rc` files follows the INI-style:

```
[section]
var = value
```

exception `ApeTools.Config.`**`ConfigError`**`(`*value*`)`

Exception class for problems during configuration and command-line processing.

`ApeTools.Config.`**`addOptions`**`(`*optParser*`)`

Set command line options processed by the configuration management.

The main script is expected to call this function to create all the options which the configuration management expects.

`ApeTools.Config.`**`get`**`(`*section*, *tag*, *fallbacks=[]*`)`

Retrieve the variable `tag` from `section`.

The `fallbacks` are provided to reduce the clutter in the `[DEFAULT]` section and avoids the contamination of individual sections with irrelevant variables from the `[DEFAULT]` section. The

Parameters

- **`section`** (*`string`*) – Config section for variable lookup
- **`tag`** (*`string`*) – Name of variable to look up
- **`fallbacks`** (*`list`*) – List of sections to look up variables not found in `section`

Return type string

`ApeTools.Config.`**`getboolean`**(*section*, *tag*, *fallbacks=[]*)

Retrieve the variable *tag* from *section*. The values `on`, `true`, `1`, and `yes` are converted to `True`. The values `off`, `false`, `0`, and `no` are converted to `False`.

Raise `ValueError` if the value cannot be converted.

Return type boolean

`ApeTools.Config.`**`getint`**(*section*, *tag*, *fallbacks=[]*)

Retrieve the variable *tag* from *section*. Converts the result to integer.

Raise `ValueError` if the value cannot be converted.

Return type integer

`ApeTools.Config.`**`getlist`**(*section*, *tag*, *fallbacks=[]*)

Retrieve the variable *tag* from *section*. Converts the result to a list of strings.

The variable `` `tag`.delete `` is used to specify values which should be removed from the list. The variables `` `tag`.prepend `` and `` `tag`.append `` specify items which are prepended and appended to the list.

Note: The `.delete`, `.prepend`, and `.append` variables are reserved for customization of the configuration by the user. They should **not** be used in system configurations provided with `ape`.

Return type list of strings

`ApeTools.Config.`**`init`**(*options=None*)

Prepare the configuration managment by reading the `.rc` files. Information from the command line can be passed through the *options* parameter. It will be propagated in to the right variables in the `[ape]` and `[DEFAULT]` sections.

This functions processes first all the files matching `config/*.rc` in the ape distribution and and platform specific configurations `` config/ape.rc.`sys.platform` `` and `` config/ape.rc.`sys.platform`.`os.uname()[4]` ``. The later allows for special settings for 32bit and 64bit platforms. It then looks for user specified files:

1. If the environment variable `APERC` is set, the file it points to. It is a fault if it does not exist.
2. `~/.aperc`.
3. The file specified on the commandline using the `--rc` option. It is a fault if it does not exist. It replaces the files found by options 1. and 2.

Only on of 1. or 2. are allowed.

Note: Only the first and second case should be used in production settings. The other two options are mainly useful for testing new configurations.

Parameters **`options`** – An options object returned by an options parsers.

`ApeTools.Config.`**`sections`**`()`

Returns the list of known sections in the configuration files.

`ApeTools.Config.`**`variables`**`(`*section*`)`

Returns the list of variables in the section *section*.

Note: This will include *all* the variables defined in the `[DEFAULT]` section as well as the variables defined in *section*.

3.1.4 Sub-module `Environment`

class `ApeTools.Environment.`**`Environment`**`(`*env={}*, *baseEnv={}*`)`

Environment class. This is used to accumulate dictionaries that define environment variables. It provides two convenience functions to convert the dictionary into sh and csh code for setting the user's environment.

Instances are mapping objects.

Parameters

- **`env`** – Provides the initial environment stored in `_env`.
- **`baseEnv`** (*`read-only`*) – Set the base environment to extend. Typical use is to set it to `os.environ`.

`csh``()`

Create the code to set the environment for a shell of the *C-shell* family.

`sh``()`

Create the code to set the environment for a shell of the *Bourne-shell* family.

`toString``(`*template*`)`

Convert environment to a string of [c]sh commands.

Handles substitutions of variables, e.g., LD_LIBRARY_PATH to DYLD_LIBRARY_PATH on OS X.

The substitution is customized through templates.

Parameters **`template`** – A template for setting variables.

`update``(`*env={}*, ***kwargs*`)`

Add the variables from one environment into 'self'. Path-like values are concatenated to form a `:` separated list. System paths, listed in *`sysPaths`*, are terminated by `:` if previously empty. Other variables are copied, existing values get replaced.

The additions to the environment can also be specified as keyword arguments.

`ApeTools.Environment.`**`cshTemplate`** **= 'setenv %(var)s %(value)s;'**

Template to set a variable for shells of the *C-shell* family.

`ApeTools.Environment.`**`paths`** **= ['MANPATH', 'PATH', 'LD_LIBRARY_PATH', 'PKG_CONFIG_PATH', 'PYTHONPA**

Path variables. Extending should create a ":" separated list.

`ApeTools.Environment.`**`replace`** **= {'LD_LIBRARY_PATH': 'DYLD_LIBRARY_PATH'}**

Environment variable names to change from the unix default, e.g., for OS X

`ApeTools.Environment.`**`shTemplate`** **= 'export %(var)s=%(value)s'**

Template to set a variable for shells of the *Bourne-shell* family.

`ApeTools.Environment.`**`sysPaths`** **= ['MANPATH']**

System path variables. The system default should not be disabled. An empty variable means that a single ":" has to be appended when updating the system environment

3.1.5 Sub-module `Fetch`

class `ApeTools.Fetch.`**`DataMirror`**(*owner*, *tag*, *location*, *url*, *user*)

Class to store data mirror properties and fetch requested source files.

Parameters

- **`owner`** – The *`DownloadManager`* instance which owns this object.
- **`tag`** – A tag to identify the mirror. This is used to select the preferred mirror(s).
- **`location`** – Specifies the country and institution where the mirror is hosted.
- **`url`** – The base url for this mirror. Files are located relative to this url.
- **`user`** – The user-name to specify for downloads from this site. A password is requeste when needed.

`fetchFile`(*fileName*)

Retrieve the file *filename* from this mirror. The password is for the `user` is requested from our `owner`.

Raise *`DownloadError`* in case of problems.

`fetchFileNative`(*fileName*, *url*)

Fetch one file from remote mirror. This routine provides a pure python based implementation, based on `urllib2`.

Note: Currently, in python 2.6 and earlier, this method cannot be used to retrieve `https` urls through a firewall. The preferred method is *`fetchFileWget()`*.

Raise *`DownloadError`* in case of problems.

`fetchFileWget`(*fileName*, *url*)

Fetch a file from a remote mirror, using `wget` to do the actual transfer.

exception `ApeTools.Fetch.`**`DownloadError`**(*value*)

Exception class for problems with the download.

class `ApeTools.Fetch.`**`DownloadManager`**

Manager for downloads.

`addMirror`(*tag*)

Add the mirror identified by *tag*. The information is retrieved from the configuration management, section `[mirror 'tag']`. Defautls for all mirrors can be provided in the section `[mirror]`.

```
[mirror]
mirrorUser = globalUser

[mirror exa]
location = Exampolonia University
url = http://www.mirror.example.edu/ape-files
```

Raise *`ApeTools.InstallError`* if it cannot locate the information for mirror *tag*.

`checkSha1`(*fileName*)

Compare checksum for *fileName*.

Returns Result of comparison of checksum with precomputed value, `None` if no precomputed checksum available.

computeSha1(*f*)
Compute the sha1 checksum of file *f*. *f* has to be an open file.

Returns `sha1` checksum object

fetchFile(*fileName*, *directory=None*)
Retrieve *fileName*. If set, the result is stored in the directory *directory*, otherwise it is stored in the current directory.

The *DataMirror*s known to this manager are tried in a round-robin fashion. We remember the last site used successfully and start trying the next download from there.

fillMirrors()
Extract the information for all mirrors. The list of mirror sites starts with the sites listed in the `ape.mirrors` configuration variable. This is followed by all sites not yet listed, identified by looking for sections named `[mirror ‘tag‘]`.

fillSha1()
Read the sha1 checksums from a file. The file is specified by the configuration variable `sha1File` in section `[ape internal]`. The file is in the format generated by `openssl sha1`.

Note: The SHA1 file has to be re-generated when the distribution mirrors get updated.

getPassword(*user*)
Return the password for *user*. If it is not known, try reading the password from file or prompt on command line. Cache the result.

getPasswordFile(*user*)
Read password from file. The file has to be accessible only by the owner, mode, permisson 0400 or 0600. The file contains space-separated pairs (user, password).

getPasswordInteractive(*user*)
Prompt user for password

3.2 Specialized build support for individual packages: `ApePackages`

This module contains sub-modules for the packages which require special code for building. The default implementation for all the build steps is in *ApeTools.Build.Package*. Packages which are not listed here are configured using only the options documented in the section *The ape configuration files*.

The name of the builder is normaly the name of the package, in lower-case letters. Exceptions are noted in the corresponding builder modules.

All sub-modules of *ApePackages* are imported to give them a chance to register their creator functions.

3.2.1 Build support for adst package

class `ApePackages.ADST.`**ADST**(*name*)

build(*env*)
Add the `ADSTROOT` environment variable before before calling *ApeTools.Build.Package.build()* to build adst.

isInstalled()
Determine if the adst package is installed by looking for **EventBrowser**.

unpack()
: Unpack adst. The top level directory in the tar-ball is always called `ADST`. We rename it to the requested source-directory. Since adst does an in-place build, this is also the place of the final installation.

3.2.2 Build support for the aires air shower simulation package

class `ApePackages.Aires.`**Aires**(*name*)

__init__(*name*)
: Set up additional attributes:

  fc The fortran compiler

  prefixBase The base of the prefix, without version number. Needed for configuration.

configure(*_logName*, *_env*)
: Generate the `config` file in the aires top level directory.

  The package variable `fc` can be used to select the Fortran compiler used to build aires.

make(*logName*, *env*)
: Run the **doinstall** script to build and install Aires.

3.2.3 Build support for augercmake extra modules for cmake

class `ApePackages.AugerCMake.`**AugerCMake**(*name*)

isInstalled()
: Special function to check if the package is installed. This is needed because of the non-standard installation path needed to make sure cmake can locate the extra modules.

3.2.4 Build support for the cdas package

class `ApePackages.CDAS.`**CDAS**(*name*)
: Customizations for the cdas package.

  The cdas package is built in place. Is also provides a large number of environment variables which can cannot be set statically.

__init__(*name*)
: Set attribute `cmtSystem`.

  It follows the algorithm of CMT for normal system. The special case of running on a system with AFS is not take into account.

env()
: Set the environment.

  Determine the location of all library packages and set the `...HOME` variables as well as the `LD_LIBRARY_PATH`. Update `PATH` to locate **ED.exe**.

isInstalled()
: Determine if the cdas package is installed by looking for **ED.exe**.

packageVersion(*package*)

Determine the version of one of the sub-packages (in CMT nomenclature).

Expects that there is only one directory in the package directory. Uses the name of this directory as the package version.

Example if we have `IoSd/v2r8`, the version of the `IoSd` package is `v2r8`.

unpack()

Unpack into the install location.

Unpack the cdas package into its final location. It has to be built there, since CMT does not install packages after building. The only activity of the *install* phase is to clean up temporary files.

3.2.5 Build support for cppunit testing package

class `ApePackages.Cppunit.`**Cppunit**(*name*)

configure(*logFile*, *env*)

Configuration fails on OS X, at least for version `1.12.1`. Nevertheless, it is possible to build the package with the files which got created. So we catch and ignore all errors in the *configure* phase if we are on OS X.

3.2.6 Build support for geant4

class `ApePackages.Geant4.`**Geant4**(*name*)

__init__(*name*)

Add the `g4system` attribute and read it from the configuration..

make(*logFile*, *env*)

3.2.7 Build support for mysql connector package

This package mainly for users of old linux distributions, for users of OS X, or on computing centres where the administrators refuse to install the mysql client software.

The builder is registered with the name `mysql`.

Note: If you run a MySQL server, on a linux machine, it is strongly recommended that you install the client headers and libraries from the same source. Otherwise, connecting to databases on `localhost` can fail if the path of the unix socket used for communication can differ between the server and the client.

class `ApePackages.Mysql.`**Mysql**(*name*)

Install the connector distributed from the MySQL download site.

install(*logFile*, *env*)

Extend standard install to set additional symbolic link `%(prefix)/include/mysql -> .` to support Auger Offline's use of `#include`.

3.2.8 Build support for CERN's root package

class `ApePackages.Root.`**`Root`**(*name*)

> **`__init__`**(*name*)
>
> Defines extra `enable` and `disable` list attributes and reads their values from the configuration. These attributes are used to determine the list of `--enable-*` and `--disable-*` arguments to **`./configure`**. This implementation keeps the option lists in the configuration files shorter.
>
> **`build`**(*env*)
>
> Remove the `ROOTSYS` environment variable before before calling *`ApeTools.Build.Package.build()`* to build root.
>
> **`unpack`**()
>
> Unpack root. The top level directory in the tar-ball is always called `root`. We rename it to `root-`*version*.

3.2.9 Build support for the xercesc xml parser

The builder is registered with the name `xerces`.

class `ApePackages.Xerces.`**`Xerces`**(*name*)

Add the attribute `sourceDirectory` and read it from the configuration.

> **`build`**(*env*)
>
> Set the environment variable `XERCESCCROOT` to the `sourceDirectory` before calling *`ApeTools.Build.Package.build()`* to build xercesc.

3.3 The main script

The main top-level `ape` script to manage software packages.

Please consult the *Ape user guide* on how to use `ape`.

class `ape.`**`Ape`**

This class bundles methods implementing the individual actions of the `ape` script and a few shared attributed.

> **`apeConfig`**(*_packages*, *outFile=<open file '<stdout>', mode 'w'>*)
>
> Provide information about the configuration of `ape` to *outFile*.
>
> **`clean`**(*packages*)
>
> Clean up build directories of *packages*. If *package* is empty, cleans up build directories of all installed packages.
>
> **`dispatch`** = **<ape.Dispatch instance>**
>
> Dictionary mapping verbs to methods implementing
>
> **`dumpConfig`**(*_packages*)
>
> Write the configuration information of `ape` and of the *packages* to logfile.
>
> **`envCsh`**(*packages*)
>
> Print a list of instructions which can be **`source`**ed to provide the environment variables defined by the currently installed packages to the user. This version is for the *C-shell* family of shells.
>
> **`envSh`**(*packages*)
>
> Print a list of instructions which can be **`source`**ed to provide the environment variables defined by the currently installed packages to the user. This version is for the *Bourne-shell* family of shells.

fetch (*packages*)
Download the *packages*. Unless disabled, also fetches the dependencies.

getDownloadManager ()
Provide access to the `ApeTools.Fetch.DownloadManager`' instance we use. Create it if necessary.

install (*packages*)
Install the *packages* and their dependencies. Download if needed. If a package is already installed, don't install again, unless *--force*ed.

installed (*_packages*)
List the packages which are currently installed.

main ()
The main routine for `ape`. It handles command-line parsing, validation, and dispatch according to the requested action.

mirrors (*_packages*)
Print a list of known download-mirrors.

packageConfig (*packages*, *outFile=<open file '<stdout>', mode 'w'>*)
Provide information about the configuration of packages to *outFile*. If the list *packages* is empty, provide information about all known packages. Otherwise, provide information only on those in *packages*.

packages (*packages*)
Provide a list of packages known to `ape`. If *packages* is empty, list all packages known. Otherwise, provide information only on those listed in *packages*.

unpack (*packages*)
Unpack the *packages*. Unless disabled, also unpacks the dependencies.

class `ape.`**DescriptionFormatter** (*indent_increment=2*, *max_help_position=24*, *width=None*, *short_first=1*)
A formatter for `optparse.OptionParse` object.

format_description (*description*)
If *description* empty, return "", else return description with "\n".

class `ape.`**Dispatch**
Dispatch handler for `ape`'s actions.

exception DispatchError
Exception, so we can catch dispatch errors selectively.

`Dispatch.`**addAction** (*name*, *action*, *description=''*)
Add callable *action* to use when requesting *name*. Provides *description* for help.

`Dispatch.`**call** (*name*, **args*)
Call action *name* with arguments **args*.

Raises *Dispatch.DispatchError* if *name* not registered.

`Dispatch.`**getDescriptions** ()
Generate a short explanation from the descriptions of all actions. The descriptions will be listed in the order in which the actions where added.

The names are listed on the left. The space for the names is determined by the longest name in the list. The description for each action is wrapped to fit into the remaining space.

The result is inteded to be passed as a description to an `optparse.OptionParser` instance which uses an instance of *DescriptionFormatter* as its formatter.

exception `ape.`**`UsageError`**

Class to allow commands to report errors. This error will be caught to produce help and/or a message.

`ape.`**`abort`**`(`*error*, *parser=None*, *printHelp=False*`)`

Abort the processing in the case of an error.

This function can optionally print usage information for the script.

Parameters

- **`error`** – The error message to report.
- **`parser`** – An `optparse.OptionParser` instance. If present, use it to provide usage information or help.
- **`printHelp`** – Flag to indicate if to print the full help information or just a sort usage information.

`ape.`**`expandDependencies`**`(`*packages*, *expand=True*`)`

Expands a list of packages to include dependencies if requested. If *expand* is `False`, return the original list unchanged.

`ape.`**`haveToInstall`**`(`*pack*, *options*, *listed*`)`

Determine if a package has to be installed or if its installation can be safely skipped. A package has to be installed if one of the following conditions holds:

- The package is not yet installed.
- The option *`--force-all`* has be specified.
- The option *`--force`* has be specified and the package is listed in *listed*. This way, the option *`--force`* only acts on packages explicitly requested but not on their dependencies.
- The package attribute `alwaysInstall` is set to `True`.

`ape.`**`listInstalled`**`()`

Return a list of all packages which are currenly installed.

`ape.`**`packageEnv`**`(`*packages=None*`)`

Return an `ApeTools.Environment.Environment`` instance with is populated with the environment variables provided by all the packages listed in *packages*. If *packages* is not given, build environment for currently installed.

Helper to emit instructions for setting the [c]sh environment.

INDEX