10. API Reference

10.1. distutils.core — Core Distutils functionality

The distutils.core module is the only module that needs to be installed to use the Distutils. It provides the setup() (which is called from the setup script). Indirectly provides the distutils.dist.Distribution and distutils.cmd.Command class.

distutils.core.**setup**(*arguments*)

The basic do-everything function that does most everything you could ever ask for from a Distutils method.

The setup function takes a large number of arguments. These are laid out in the following table.

argument name	value	type
name	The name of the package	a string
version	The version number of the package; see distutils.version	a string
description	A single line describing the package	a string
long_description	Longer description of the package	a string
author	The name of the package author	a string
author_email	The email address of the package author	a string
maintainer	The name of the current maintainer, if different from the author. Note that if the maintainer is provided, distutils	a string

argument name	value	type
	will use it as the author in `PKG-INFO`	
maintainer_email	The email address of the current maintainer, if different from the author	a string
url	A URL for the package (homepage)	a string
download_url	A URL to download the package	a string
packages	A list of Python packages that distutils will manipulate	a list of strings
py_modules	A list of Python modules that distutils will manipulate	a list of strings
scripts	A list of standalone script files to be built and installed	a list of strings
ext_modules	A list of Python extensions to be built	a list of instances of `distutils.core.Extension`
classifiers	A list of categories for the package	a list of strings; valid classifiers are listed on PyPI.
distclass	the `Distribution` class to use	a subclass of `distutils.core.Distribution`
script_name	The name of the setup.py script - defaults to `sys.argv[0]`	a string
script_args	Arguments to supply to the setup script	a list of strings
options	default options for the setup script	a dictionary
license	The license for the package	a string
keywords	Descriptive meta-data, see **PEP 314**	a list of strings or a comma-separated string

argument name	value	type
platforms		a list of strings or a comma-separated string
cmdclass	A mapping of command names to `Command` subclasses	a dictionary
data_files	A list of data files to install	a list
package_dir	A mapping of package to directory names	a dictionary

`distutils.core.`**`run_setup`**(*script_name*[, *script_args=None*, *stop_after='run'*])

Run a setup script in a somewhat controlled environment, and return the `distutils.dist.Distribution` instance that drives things. This is useful if you need to find out the distribution meta-data (passed as keyword args from *script* to `setup()`), or the contents of the config files or command-line.

script_name is a file that will be read and run with `exec()`. `sys.argv[0]` will be replaced with *script* for the duration of the call. *script_args* is a list of strings; if supplied, `sys.argv[1:]` will be replaced by *script_args* for the duration of the call.

stop_after tells `setup()` when to stop processing; possible values:

value	description
init	Stop after the `Distribution` instance has been created and populated with the keyword arguments to `setup()`
config	Stop after config files have been parsed (and their data stored in the `Distribution` instance)
commandline	Stop after the command-line (`sys.argv[1:]` or *script_args*) have been parsed (and the data stored in the `Distribution` instance.)
run	Stop after all commands have been run (the same as if `setup()` had been called in the usual way). This is the default value.

In addition, the `distutils.core` module exposed a number of classes that live elsewhere.

- `Extension` from `distutils.extension`
- `Command` from `distutils.cmd`
- `Distribution` from `distutils.dist`

A short description of each of these follows, but see the relevant module for the full reference.

class `distutils.core.`**`Extension`**

The Extension class describes a single C or C++ extension module in a setup script. It accepts the following keyword arguments in its constructor:

argument name	**value**	**type**
name	the full name of the extension, including any packages — ie. *not* a filename or pathname, but Python dotted name	a string
sources	list of source filenames, relative to the distribution root (where the setup script lives), in Unix form (slash-separated) for portability. Source files may be C, C++, SWIG (.i), platform-specific resource files, or whatever else is recognized by the **build_ext** command as source for a Python extension.	a list of strings
include_dirs	list of directories to search for C/C++ header files (in Unix form for portability)	a list of strings
define_macros	list of macros to define; each macro is defined using a 2-tuple `(name, value)`, where *value* is either the string to define it to or `None` to define it without a particular value (equivalent of `#define FOO` in source or `-DFOO` on Unix C compiler command line)	a list of tuples
undef_macros		a list of strings

argument name	value	type
	list of macros to undefine explicitly	
library_dirs	list of directories to search for C/C++ libraries at link time	a list of strings
libraries	list of library names (not filenames or paths) to link against	a list of strings
runtime_library_dirs	list of directories to search for C/C++ libraries at run time (for shared extensions, this is when the extension is loaded)	a list of strings
extra_objects	list of extra files to link with (eg. object files not implied by ‘sources’, static library that must be explicitly specified, binary resource files, etc.)	a list of strings
extra_compile_args	any extra platform- and compiler-specific information to use when compiling the source files in ‘sources’. For platforms and compilers where a command line makes sense, this is typically a list of command-line arguments, but for other platforms it could be anything.	a list of strings
extra_link_args	any extra platform- and compiler-specific information to use when linking object files together to create the extension (or to create a new static Python interpreter). Similar interpretation as for ‘extra_compile_args’.	a list of strings
export_symbols	list of symbols to be exported from a shared extension. Not used on all platforms,	a list of strings

argument name	value	type
	and not generally necessary for Python extensions, which typically export exactly one symbol: `init` + extension_name.	
depends	list of files that the extension depends on	a list of strings
language	extension language (i.e. `'c'`, `'c++'`, `'objc'`). Will be detected from the source extensions if not provided.	a string
optional	specifies that a build failure in the extension should not abort the build process, but simply skip the extension.	a boolean

class `distutils.core.`**`Distribution`**

A `Distribution` describes how to build, install and package up a Python software package.

See the `setup()` function for a list of keyword arguments accepted by the Distribution constructor. `setup()` creates a Distribution instance.

class `distutils.core.`**`Command`**

A `Command` class (or rather, an instance of one of its subclasses) implement a single distutils command.

10.2. `distutils.ccompiler` — CCompiler base class

This module provides the abstract base class for the `CCompiler` classes. A `CCompiler` instance can be used for all the compile and link steps needed to build a single project. Methods are provided to set options for the compiler — macro definitions, include directories, link path, libraries and the like.

This module provides the following functions.

`distutils.ccompiler.`**`gen_lib_options`**(*compiler*, *library_dirs*, *runtime_library_dirs*, *libraries*)

Generate linker options for searching library directories and linking with specific libraries. *libraries* and *library_dirs* are, respectively, lists of library names (not filenames!) and search directories. Returns a list of command-line options suitable for use with some compiler (depending on the two format strings passed in).

distutils.ccompiler.**gen_preprocess_options**(*macros*, *include_dirs*)

Generate C pre-processor options (`-D`, `-U`, `-I`) as used by at least two types of compilers: the typical Unix compiler and Visual C++. *macros* is the usual thing, a list of 1- or 2-tuples, where `(name,)` means undefine (`-U`) macro *name*, and `(name, value)` means define (`-D`) macro *name* to *value*. *include_dirs* is just a list of directory names to be added to the header file search path (`-I`). Returns a list of command-line options suitable for either Unix compilers or Visual C++.

distutils.ccompiler.**get_default_compiler**(*osname*, *platform*)

Determine the default compiler to use for the given platform.

osname should be one of the standard Python OS names (i.e. the ones returned by `os.name`) and *platform* the common value returned by `sys.platform` for the platform in question.

The default values are `os.name` and `sys.platform` in case the parameters are not given.

distutils.ccompiler.**new_compiler**(*plat=None*, *compiler=None*, *verbose=0*, *dry_run=0*, *force=0*)

Factory function to generate an instance of some CCompiler subclass for the supplied platform/compiler combination. *plat* defaults to `os.name` (eg. `'posix'`, `'nt'`), and *compiler* defaults to the default compiler for that platform. Currently only `'posix'` and `'nt'` are supported, and the default compilers are “traditional Unix interface” (`UnixCCompiler` class) and Visual C++ (`MSVCCompiler` class). Note that it’s perfectly possible to ask for a Unix compiler object under Windows, and a Microsoft compiler object under Unix—if you supply a value for *compiler*, *plat* is ignored.

distutils.ccompiler.**show_compilers**()

Print list of available compilers (used by the `--help-compiler` options to **build**, **build_ext**, **build_clib**).

class distutils.ccompiler.**CCompiler**([*verbose=0*, *dry_run=0*, *force=0*])

The abstract base class `CCompiler` defines the interface that must be implemented by real compiler classes. The class also has some utility methods used by several compiler classes.

The basic idea behind a compiler abstraction class is that each instance can be used for all the compile/link steps in building a single project. Thus, attributes common to all of those compile and link steps — include directories, macros to define, libraries to link against, etc. — are attributes of the compiler instance. To allow for variability in how individual files are treated, most of those attributes may be varied on a per-compilation or per-link basis.

The constructor for each subclass creates an instance of the Compiler object. Flags are *verbose* (show verbose output), *dry_run* (don't actually execute the steps) and *force* (rebuild everything, regardless of dependencies). All of these flags default to `0` (off). Note that you probably don't want to instantiate `CCompiler` or one of its subclasses directly - use the `distutils.CCompiler.new_compiler()` factory function instead.

The following methods allow you to manually alter compiler options for the instance of the Compiler class.

`add_include_dir`(*dir*)

Add *dir* to the list of directories that will be searched for header files. The compiler is instructed to search directories in the order in which they are supplied by successive calls to `add_include_dir()`.

`set_include_dirs`(*dirs*)

Set the list of directories that will be searched to *dirs* (a list of strings). Overrides any preceding calls to `add_include_dir()`; subsequent calls to `add_include_dir()` add to the list passed to `set_include_dirs()`. This does not affect any list of standard include directories that the compiler may search by default.

`add_library`(*libname*)

Add *libname* to the list of libraries that will be included in all links driven by this compiler object. Note that *libname* should *not* be the name of a file containing a library, but the name of the library itself: the actual filename will be inferred by the linker, the compiler, or the compiler class (depending on the platform).

The linker will be instructed to link against libraries in the order they were supplied to `add_library()` and/or `set_libraries()`. It is perfectly valid to duplicate library names; the linker will be instructed to link against libraries as many times as they are mentioned.

`set_libraries`(*libnames*)

Set the list of libraries to be included in all links driven by this compiler object to *libnames* (a list of strings). This does not affect any standard system libraries that the linker may include by default.

`add_library_dir`(*dir*)

Add *dir* to the list of directories that will be searched for libraries specified to `add_library()` and `set_libraries()`. The linker will be instructed to search for libraries in the order they are supplied to `add_library_dir()` and/or `set_library_dirs()`.

`set_library_dirs`(*dirs*)

Set the list of library search directories to *dirs* (a list of strings). This does not affect any standard library search path that the linker may search by default.

`add_runtime_library_dir`(*dir*)

Add *dir* to the list of directories that will be searched for shared libraries at runtime.

`set_runtime_library_dirs`(*dirs*)

Set the list of directories to search for shared libraries at runtime to *dirs* (a list of strings). This does not affect any standard search path that the runtime linker may search by default.

`define_macro`(*name*[, *value=None*])

Define a preprocessor macro for all compilations driven by this compiler object. The optional parameter *value* should be a string; if it is not supplied, then the macro will be defined without an explicit value and the exact outcome depends on the compiler used.

`undefine_macro`(*name*)

Undefine a preprocessor macro for all compilations driven by this compiler object. If the same macro is defined by `define_macro()` and undefined by `undefine_macro()` the last call takes precedence (including multiple redefinitions or undefinitions). If the macro is redefined/undefined on a per-compilation basis (ie. in the call to `compile()`), then that takes precedence.

`add_link_object`(*object*)

Add *object* to the list of object files (or analogues, such as explicitly named library files or the output of “resource compilers”) to be included in every link driven by this compiler object.

`set_link_objects`(*objects*)

Set the list of object files (or analogues) to be included in every link to *objects*. This does not affect any standard object files that the linker may include by default (such as system libraries).

The following methods implement methods for autodetection of compiler options, providing some functionality similar to GNU **autoconf**.

`detect_language`(*sources*)

Detect the language of a given file, or list of files. Uses the instance attributes `language_map` (a dictionary), and `language_order` (a list) to do the job.

`find_library_file`(*dirs*, *lib*[, *debug=0*])

Search the specified list of directories for a static or shared library file *lib* and return the full path to that file. If *debug* is true, look for a debugging version (if that makes sense on the current platform). Return `None` if *lib* wasn't found in any of the specified directories.

`has_function`(*funcname*[, *includes=None*, *include_dirs=None*, *libraries=None*, *library_dirs=None*])

Return a boolean indicating whether *funcname* is supported on the current platform. The optional arguments can be used to augment the compilation environment by providing additional include files and paths and libraries and paths.

`library_dir_option`(*dir*)

Return the compiler option to add *dir* to the list of directories searched for libraries.

`library_option`(*lib*)

Return the compiler option to add *lib* to the list of libraries linked into the shared library or executable.

`runtime_library_dir_option`(*dir*)

Return the compiler option to add *dir* to the list of directories searched for runtime libraries.

`set_executables`(***args*)

Define the executables (and options for them) that will be run to perform the various stages of compilation. The exact set of executables that may be specified here depends on the compiler class (via the 'executables' class attribute), but most will have:

attribute	**description**
compiler	the C/C++ compiler
linker_so	linker used to create shared objects and libraries
linker_exe	linker used to create binary executables
archiver	static library creator

On platforms with a command-line (Unix, DOS/Windows), each of these is a string that will be split into executable name and (optional) list of arguments. (Splitting the string is done similarly to how Unix shells operate: words are delimited by spaces, but quotes and backslashes can override this. See `distutils.util.split_quoted()`.)

The following methods invoke stages in the build process.

`compile`(*sources*[, *output_dir=None*, *macros=None*, *include_dirs=None*, *debug=0*, *extra_preargs=None*, *extra_postargs=None*, *depends=None*])

Compile one or more source files. Generates object files (e.g. transforms a `.c` file to a `.o` file.)

sources must be a list of filenames, most likely C/C++ files, but in reality anything that can be handled by a particular compiler and compiler class (eg. `MSVCCompiler` can handle resource files in *sources*). Return a list of object filenames, one per source filename in *sources*. Depending on the implementation, not all source files will necessarily be compiled, but all corresponding object filenames will be returned.

If *output_dir* is given, object files will be put under it, while retaining their original path component. That is, `foo/bar.c` normally compiles to `foo/bar.o` (for a Unix implementation); if *output_dir* is *build*, then it would compile to `build/foo/bar.o`.

macros, if given, must be a list of macro definitions. A macro definition is either a `(name, value)` 2-tuple or a `(name,)` 1-tuple. The former defines a macro; if the value is `None`, the macro is defined without an explicit value. The 1-tuple case undefines a macro. Later definitions/redefinitions/undefinitions take precedence.

include_dirs, if given, must be a list of strings, the directories to add to the default include file search path for this compilation only.

debug is a boolean; if true, the compiler will be instructed to output debug symbols in (or alongside) the object file(s).

extra_preargs and *extra_postargs* are implementation-dependent. On platforms that have the notion of a command-line (e.g. Unix, DOS/Windows), they are most likely lists of strings: extra command-line arguments to prepend/append to the compiler command line. On other platforms, consult the implementation class documentation. In any event, they are intended as an escape hatch for those occasions when the abstract compiler framework doesn't cut the mustard.

depends, if given, is a list of filenames that all targets depend on. If a source file is older than any file in depends, then the source file will be recompiled. This supports dependency tracking, but only at a coarse granularity.

Raises `CompileError` on failure.

`create_static_lib`(*objects*, *output_libname*[, *output_dir=None*, *debug=0*, *target_lang=None*])

Link a bunch of stuff together to create a static library file. The "bunch of stuff" consists of the list of object files supplied as *objects*, the extra object files supplied to `add_link_object()` and/or `set_link_objects()`, the libraries supplied to `add_library()` and/or `set_libraries()`, and the libraries supplied as *libraries* (if any).

output_libname should be a library name, not a filename; the filename will be inferred from the library name. *output_dir* is the directory where the library file will be put.

debug is a boolean; if true, debugging information will be included in the library (note that on most platforms, it is the compile step where this matters: the *debug* flag is included here just for consistency).

target_lang is the target language for which the given objects are being compiled. This allows specific linkage time treatment of certain languages.

Raises `LibError` on failure.

`link`(*target_desc*, *objects*, *output_filename*[, *output_dir=None*, *libraries=None*, *library_dirs=None*, *runtime_library_dirs=None*, *export_symbols=None*, *debug=0*, *extra_preargs=None*, *extra_postargs=None*, *build_temp=None*, *target_lang=None*])

Link a bunch of stuff together to create an executable or shared library file.

The "bunch of stuff" consists of the list of object files supplied as *objects*. *output_filename* should be a filename. If *output_dir* is supplied, *output_filename* is relative to it (i.e. *output_filename* can provide directory components if needed).

libraries is a list of libraries to link against. These are library names, not filenames, since they're translated into filenames in a platform-specific way (eg. *foo* becomes `libfoo.a` on Unix and `foo.lib` on DOS/Windows). However, they can include a directory component, which means the linker will look in that specific directory rather than searching all the normal locations.

library_dirs, if supplied, should be a list of directories to search for libraries that were specified as bare library names (ie. no directory component). These are on top of the system default and those supplied to `add_library_dir()` and/or `set_library_dirs()`. *runtime_library_dirs* is a list of directories that will be embedded into the shared library and used to search for other shared libraries that *it* depends on at run-time. (This may only be relevant on Unix.)

export_symbols is a list of symbols that the shared library will export. (This appears to be relevant only on Windows.)

debug is as for `compile()` and `create_static_lib()`, with the slight distinction that it actually matters on most platforms (as opposed to `create_static_lib()`, which includes a *debug* flag mostly for form's sake).

extra_preargs and *extra_postargs* are as for `compile()` (except of course that they supply command-line arguments for the particular linker being used).

target_lang is the target language for which the given objects are being compiled. This allows specific linkage time treatment of certain languages.

Raises `LinkError` on failure.

`link_executable`(*objects*, *output_progname*[, *output_dir=None*, *libraries=None*, *library_dirs=None*, *runtime_library_dirs=None*, *debug=0*, *extra_preargs=None*, *extra_postargs=None*, *target_lang=None*])

Link an executable. *output_progname* is the name of the file executable, while *objects* are a list of object filenames to link in. Other arguments are as for the `link()` method.

`link_shared_lib`(*objects*, *output_libname*[, *output_dir=None*, *libraries=None*, *library_dirs=None*, *runtime_library_dirs=None*, *export_symbols=None*, *debug=0*, *extra_preargs=None*, *extra_postargs=None*, *build_temp=None*, *target_lang=None*])

Link a shared library. *output_libname* is the name of the output library, while *objects* is a list of object filenames to link in. Other arguments are as for the `link()` method.

`link_shared_object`(*objects*, *output_filename*[, *output_dir=None*, *libraries=None*, *library_dirs=None*, *runtime_library_dirs=None*, *export_symbols=None*, *debug=0*, *extra_preargs=None*, *extra_postargs=None*, *build_temp=None*, *target_lang=None*])

Link a shared object. *output_filename* is the name of the shared object that will be created, while *objects* is a list of object filenames to link in. Other arguments are as for the `link()` method.

`preprocess`(*source*[, *output_file=None*, *macros=None*, *include_dirs=None*, *extra_preargs=None*, *extra_postargs=None*])

Preprocess a single C/C++ source file, named in *source*. Output will be written to file named *output_file*, or *stdout* if *output_file* not supplied. *macros* is a list of macro definitions as for `compile()`, which will augment the macros set with `define_macro()` and `undefine_macro()`. *include_dirs* is a list of directory names that will be added to the default list, in the same way as `add_include_dir()`.

Raises `PreprocessError` on failure.

The following utility methods are defined by the `CCompiler` class, for use by the various concrete subclasses.

`executable_filename`(*basename*[, *strip_dir=0*, *output_dir=''*])

Returns the filename of the executable for the given *basename*. Typically for non-Windows platforms this is the same as the basename, while Windows will get a `.exe` added.

`library_filename`(*libname*[, *lib_type='static'*, *strip_dir=0*, *output_dir=''*])

Returns the filename for the given library name on the current platform. On Unix a library with *lib_type* of `'static'` will typically be of the form `liblibname.a`, while a *lib_type* of `'dynamic'` will be of the form `liblibname.so`.

`object_filenames`(*source_filenames*[, *strip_dir=0*, *output_dir=''*])

Returns the name of the object files for the given source files. *source_filenames* should be a list of filenames.

`shared_object_filename`(*basename*[, *strip_dir=0*, *output_dir=''*])

Returns the name of a shared object file for the given file name *basename*.

`execute`(*func*, *args*[, *msg=None*, *level=1*])

Invokes `distutils.util.execute()`. This method invokes a Python function *func* with the given arguments *args*, after logging and taking into account the *dry_run* flag.

`spawn`(*cmd*)

Invokes `distutils.util.spawn()`. This invokes an external process to run the given command.

mkpath(*name*[, *mode=511*])

Invokes `distutils.dir_util.mkpath()`. This creates a directory and any missing ancestor directories.

move_file(*src*, *dst*)

Invokes `distutils.file_util.move_file()`. Renames *src* to *dst*.

announce(*msg*[, *level=1*])

Write a message using `distutils.log.debug()`.

warn(*msg*)

Write a warning message *msg* to standard error.

debug_print(*msg*)

If the *debug* flag is set on this `CCompiler` instance, print *msg* to standard output, otherwise do nothing.

10.3. `distutils.unixccompiler` — Unix C Compiler

This module provides the `UnixCCompiler` class, a subclass of `CCompiler` that handles the typical Unix-style command-line C compiler:

- macros defined with `-Dname[=value]`
- macros undefined with `-Uname`
- include search directories specified with `-Idir`
- libraries specified with `-llib`
- library search directories specified with `-Ldir`
- compile handled by **cc** (or similar) executable with `-c` option: compiles `.c` to `.o`
- link static library handled by **ar** command (possibly with **ranlib**)
- link shared library handled by **cc** `-shared`

10.4. `distutils.msvccompiler` — Microsoft Compiler

This module provides `MSVCCompiler`, an implementation of the abstract `CCompiler` class for Microsoft Visual Studio. Typically, extension modules need to be compiled with the same compiler that was used to compile Python. For Python 2.3 and earlier, the compiler was Visual Studio 6. For Python 2.4 and 2.5, the compiler is Visual Studio .NET 2003. The AMD64 and Itanium binaries are created using the Platform SDK.

`MSVCCompiler` will normally choose the right compiler, linker etc. on its own. To override this choice, the environment variables *DISTUTILS_USE_SDK* and *MSSdk* must be both set. *MSSdk* indicates that the current environment has been setup by the SDK's `SetEnv.Cmd` script, or that the environment variables had been registered when the SDK was installed; *DISTUTILS_USE_SDK* indicates that the distutils user has made an explicit choice to override the compiler selection by `MSVCCompiler`.

10.5. `distutils.bcppcompiler` — Borland Compiler

This module provides `BorlandCCompiler`, a subclass of the abstract `CCompiler` class for the Borland C++ compiler.

10.6. `distutils.cygwincompiler` — Cygwin Compiler

This module provides the `CygwinCCompiler` class, a subclass of `UnixCCompiler` that handles the Cygwin port of the GNU C compiler to Windows. It also contains the Mingw32CCompiler class which handles the mingw32 port of GCC (same as cygwin in no-cygwin mode).

10.7. `distutils.archive_util` — Archiving utilities

This module provides a few functions for creating archive files, such as tarballs or zipfiles.

`distutils.archive_util.`**`make_archive`**(*base_name*, *format*[, *root_dir=None*, *base_dir=None*, *verbose=0*, *dry_run=0*])

> Create an archive file (eg. `zip` or `tar`). *base_name* is the name of the file to create, minus any format-specific extension; *format* is the archive format: one of `zip`, `tar`, `gztar`, `bztar`, `xztar`, or `ztar`. *root_dir* is a directory that will be the root directory of the archive; ie. we typically `chdir` into *root_dir* before creating the archive. *base_dir* is the directory where we start archiving from; ie. *base_dir* will be the common prefix of all files and directories in the archive. *root_dir* and *base_dir* both default to the current directory. Returns the name of the archive file.
>
> *Changed in version 3.5:* Added support for the `xztar` format.

`distutils.archive_util.`**`make_tarball`**(*base_name*, *base_dir*[, *compress='gzip'*, *verbose=0*, *dry_run=0*])

‘Create an (optional compressed) archive as a tar file from all files in and under *base_dir*. *compress* must be `'gzip'` (the default), `'bzip2'`, `'xz'`, `'compress'`, or `None`. For the `'compress'` method the compression utility named by **compress** must be on the default program search path, so this is probably Unix-specific. The output tar file will be named `base_dir.tar`, possibly plus the appropriate compression extension (`.gz`, `.bz2`, `.xz` or `.Z`). Return the output filename.

Changed in version 3.5: Added support for the `xz` compression.

`distutils.archive_util.`**`make_zipfile`**(*base_name*, *base_dir*[, *verbose=0*, *dry_run=0*])

Create a zip file from all files in and under *base_dir*. The output zip file will be named *base_name* + `.zip`. Uses either the `zipfile` Python module (if available) or the InfoZIP `zip` utility (if installed and found on the default search path). If neither tool is available, raises `DistutilsExecError`. Returns the name of the output zip file.

10.8. `distutils.dep_util` — Dependency checking

This module provides functions for performing simple, timestamp-based dependency of files and groups of files; also, functions based entirely on such timestamp dependency analysis.

`distutils.dep_util.`**`newer`**(*source*, *target*)

Return true if *source* exists and is more recently modified than *target*, or if *source* exists and *target* doesn’t. Return false if both exist and *target* is the same age or newer than *source*. Raise `DistutilsFileError` if *source* does not exist.

`distutils.dep_util.`**`newer_pairwise`**(*sources*, *targets*)

Walk two filename lists in parallel, testing if each source is newer than its corresponding target. Return a pair of lists (*sources*, *targets*) where source is newer than target, according to the semantics of `newer()`.

`distutils.dep_util.`**`newer_group`**(*sources*, *target*[, *missing='error'*])

Return true if *target* is out-of-date with respect to any file listed in *sources* In other words, if *target* exists and is newer than every file in *sources*, return false; otherwise return true. *missing* controls what we do when a source file is missing; the default (`'error'`) is to blow up with an `OSError` from inside `os.stat()`;

if it is `'ignore'`, we silently drop any missing source files; if it is `'newer'`, any missing source files make us assume that *target* is out-of-date (this is handy in “dry-run” mode: it’ll make you pretend to carry out commands that wouldn’t work because inputs are missing, but that doesn’t matter because you’re not actually going to run the commands).

10.9. `distutils.dir_util` — Directory tree operations

This module provides functions for operating on directories and trees of directories.

`distutils.dir_util.`**`mkpath`**(*name*[, *mode=0o777*, *verbose=0*, *dry_run=0*])

Create a directory and any missing ancestor directories. If the directory already exists (or if *name* is the empty string, which means the current directory, which of course exists), then do nothing. Raise `DistutilsFileError` if unable to create some directory along the way (eg. some sub-path exists, but is a file rather than a directory). If *verbose* is true, print a one-line summary of each mkdir to stdout. Return the list of directories actually created.

`distutils.dir_util.`**`create_tree`**(*base_dir*, *files*[, *mode=0o777*, *verbose=0*, *dry_run=0*])

Create all the empty directories under *base_dir* needed to put *files* there. *base_dir* is just the name of a directory which doesn’t necessarily exist yet; *files* is a list of filenames to be interpreted relative to *base_dir*. *base_dir* + the directory portion of every file in *files* will be created if it doesn’t already exist. *mode*, *verbose* and *dry_run* flags are as for `mkpath()`.

`distutils.dir_util.`**`copy_tree`**(*src*, *dst*[, *preserve_mode=1*, *preserve_times=1*, *preserve_symlinks=0*, *update=0*, *verbose=0*, *dry_run=0*])

Copy an entire directory tree *src* to a new location *dst*. Both *src* and *dst* must be directory names. If *src* is not a directory, raise `DistutilsFileError`. If *dst* does not exist, it is created with `mkpath()`. The end result of the copy is that every file in *src* is copied to *dst*, and directories under *src* are recursively copied to *dst*. Return the list of files that were copied or might have been copied, using their output name. The return value is unaffected by *update* or *dry_run*: it is simply the list of all files under *src*, with the names changed to be under *dst*.

preserve_mode and *preserve_times* are the same as for `distutils.file_util.copy_file()`; note that they only apply to regular files, not to directories. If *preserve_symlinks* is true, symlinks will be copied as symlinks (on platforms that support them!); otherwise (the default), the destination of the symlink will be copied. *update* and *verbose* are the same as for `copy_file()`.

Files in *src* that begin with `.nfs` are skipped (more information on these files is available in answer D2 of the NFS FAQ page).

Changed in version 3.3.1: NFS files are ignored.

`distutils.dir_util.`**`remove_tree`**(*directory*[, *verbose=0*, *dry_run=0*])

Recursively remove *directory* and all files and directories underneath it. Any errors are ignored (apart from being reported to `sys.stdout` if *verbose* is true).

10.10. `distutils.file_util` — Single file operations

This module contains some utility functions for operating on individual files.

`distutils.file_util.`**`copy_file`**(*src*, *dst*[, *preserve_mode=1*, *preserve_times=1*, *update=0*, *link=None*, *verbose=0*, *dry_run=0*])

Copy file *src* to *dst*. If *dst* is a directory, then *src* is copied there with the same name; otherwise, it must be a filename. (If the file exists, it will be ruthlessly clobbered.) If *preserve_mode* is true (the default), the file's mode (type and permission bits, or whatever is analogous on the current platform) is copied. If *preserve_times* is true (the default), the last-modified and last-access times are copied as well. If *update* is true, *src* will only be copied if *dst* does not exist, or if *dst* does exist but is older than *src*.

link allows you to make hard links (using `os.link()`) or symbolic links (using `os.symlink()`) instead of copying: set it to `'hard'` or `'sym'`; if it is `None` (the default), files are copied. Don't set *link* on systems that don't support it: `copy_file()` doesn't check if hard or symbolic linking is available. It uses `_copy_file_contents()` to copy file contents.

Return a tuple `(dest_name, copied)`: *dest_name* is the actual name of the output file, and *copied* is true if the file was copied (or would have been copied, if *dry_run* true).

`distutils.file_util.`**`move_file`**(*src*, *dst*[, *verbose*, *dry_run*])

Move file *src* to *dst*. If *dst* is a directory, the file will be moved into it with the same name; otherwise, *src* is just renamed to *dst*. Returns the new full name of the file.

Warning: Handles cross-device moves on Unix using `copy_file()`. What about other systems?

`distutils.file_util.`**`write_file`**(*filename*, *contents*)

Create a file called *filename* and write *contents* (a sequence of strings without line terminators) to it.

10.11. `distutils.util` — Miscellaneous other utility functions

This module contains other assorted bits and pieces that don't fit into any other utility module.

`distutils.util.`**`get_platform`**()

Return a string that identifies the current platform. This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by 'os.uname()'), although the exact information included depends on the OS; eg. for IRIX the architecture isn't particularly important (IRIX only runs on SGI hardware), but for Linux the kernel version isn't particularly important.

Examples of returned values:

- `linux-i586`
- `linux-alpha`
- `solaris-2.6-sun4u`
- `irix-5.3`
- `irix64-6.2`

For non-POSIX platforms, currently just returns `sys.platform`.

For Mac OS X systems the OS version reflects the minimal version on which binaries will run (that is, the value of `MACOSX_DEPLOYMENT_TARGET` during the build of Python), not the OS version of the current system.

For universal binary builds on Mac OS X the architecture value reflects the universal binary status instead of the architecture of the current processor. For 32-bit universal binaries the architecture is `fat`, for 64-bit universal binaries the architecture is `fat64`, and for 4-way universal binaries the architecture is `universal`. Starting from Python 2.7 and Python 3.2 the architecture `fat3` is used for a 3-way universal build (ppc, i386, x86_64) and `intel` is used for a universal build with the i386 and x86_64 architectures

Examples of returned values on Mac OS X:

- `macosx-10.3-ppc`
- `macosx-10.3-fat`
- `macosx-10.5-universal`
- `macosx-10.6-intel`

`distutils.util.`**`convert_path`**(*pathname*)

Return 'pathname' as a name that will work on the native filesystem, i.e. split it on '/' and put it back together again using the current directory separator. Needed because filenames in the setup script are always supplied in Unix style, and have to be converted to the local convention before we can actually use them in the filesystem. Raises `ValueError` on non-Unix-ish systems if *pathname* either starts or ends with a slash.

`distutils.util.`**`change_root`**(*new_root*, *pathname*)

Return *pathname* with *new_root* prepended. If *pathname* is relative, this is equivalent to `os.path.join(new_root,pathname)` Otherwise, it requires making *pathname* relative and then joining the two, which is tricky on DOS/Windows.

`distutils.util.`**`check_environ`**()

Ensure that 'os.environ' has all the environment variables we guarantee that users can use in config files, command-line options, etc. Currently this includes:

- `HOME` - user's home directory (Unix only)
- `PLAT` - description of the current platform, including hardware and OS (see `get_platform()`)

`distutils.util.`**`subst_vars`**(*s*, *local_vars*)

Perform shell/Perl-style variable substitution on *s*. Every occurrence of `$` followed by a name is considered a variable, and variable is substituted by the value found in the *local_vars* dictionary, or in `os.environ` if it's not in *local_vars*. *os.environ* is first checked/augmented to guarantee that it contains certain values: see `check_environ()`. Raise `ValueError` for any variables not found in either *local_vars* or `os.environ`.

Note that this is not a fully-fledged string interpolation function. A valid `$variable` can consist only of upper and lower case letters, numbers and an underscore. No { } or ( ) style quoting is available.

`distutils.util.`**`split_quoted`**(*s*)

Split a string up according to Unix shell-like rules for quotes and backslashes. In short: words are delimited by spaces, as long as those spaces are not escaped by a backslash, or inside a quoted string. Single and double quotes are equivalent, and the quote characters can be backslash-escaped. The backslash is stripped from any two-character escape sequence, leaving only the escaped character. The quote characters are stripped from any quoted string. Returns a list of words.

`distutils.util.`**`execute`**(*func*, *args*[, *msg=None*, *verbose=0*, *dry_run=0*])

Perform some action that affects the outside world (for instance, writing to the filesystem). Such actions are special because they are disabled by the *dry_run* flag. This method takes care of all that bureaucracy for you; all you have to do is supply the function to call and an argument tuple for it (to embody the "external action" being performed), and an optional message to print.

`distutils.util.`**`strtobool`**(*val*)

Convert a string representation of truth to true (1) or false (0).

True values are `y`, `yes`, `t`, `true`, `on` and `1`; false values are `n`, `no`, `f`, `false`, `off` and `0`. Raises `ValueError` if *val* is anything else.

`distutils.util.`**`byte_compile`**(*py_files*[, *optimize=0*, *force=0*, *prefix=None*, *base_dir=None*, *verbose=1*, *dry_run=0*, *direct=None*])

Byte-compile a collection of Python source files to `.pyc` files in a `__pycache__` subdirectory (see **PEP 3147** and **PEP 488**). *py_files* is a list of files to compile; any files that don't end in `.py` are silently skipped. *optimize* must be one of the following:

- `0` - don't optimize
- `1` - normal optimization (like `python -O`)
- `2` - extra optimization (like `python -OO`)

If *force* is true, all files are recompiled regardless of timestamps.

The source filename encoded in each bytecode file defaults to the filenames listed in *py_files*; you can modify these with *prefix* and *basedir*. *prefix* is a string that will be stripped off of each source filename, and *base_dir* is a directory name that will be prepended (after *prefix* is stripped). You can supply either or both (or neither) of *prefix* and *base_dir*, as you wish.

If *dry_run* is true, doesn't actually do anything that would affect the filesystem.

Byte-compilation is either done directly in this interpreter process with the standard `py_compile` module, or indirectly by writing a temporary script and executing it. Normally, you should let `byte_compile()` figure out to use direct compilation or not (see the source for details). The *direct* flag is used by the script generated in indirect mode; unless you know what you're doing, leave it set to `None`.

Changed in version 3.2.3: Create `.pyc` files with an `import magic tag` in their name, in a `__pycache__` subdirectory instead of files without tag in the current directory.

Changed in version 3.5: Create `.pyc` files according to **PEP 488**.

`distutils.util.`**`rfc822_escape`**(*header*)

Return a version of *header* escaped for inclusion in an **RFC 822** header, by ensuring there are 8 spaces space after each newline. Note that it does no other modification of the string.

10.12. `distutils.dist` — The Distribution class

This module provides the `Distribution` class, which represents the module distribution being built/installed/distributed.

10.13. `distutils.extension` — The Extension class

This module provides the `Extension` class, used to describe C/C++ extension modules in setup scripts.

10.14. `distutils.debug` — Distutils debug mode

This module provides the DEBUG flag.

10.15. `distutils.errors` — Distutils exceptions

Provides exceptions used by the Distutils modules. Note that Distutils modules may raise standard exceptions; in particular, SystemExit is usually raised for errors that are obviously the end-user's fault (eg. bad command-line arguments).

This module is safe to use in `from ... import *` mode; it only exports symbols whose names start with `Distutils` and end with `Error`.

10.16. `distutils.fancy_getopt` — Wrapper around the standard getopt module

This module provides a wrapper around the standard `getopt` module that provides the following additional features:

- short and long options are tied together
- options have help strings, so `fancy_getopt()` could potentially create a complete usage summary
- options set attributes of a passed-in object
- boolean options can have "negative aliases" — eg. if `--quiet` is the "negative alias" of `--verbose`, then `--quiet` on the command line sets *verbose* to false.

`distutils.fancy_getopt.`**`fancy_getopt`**(*options*, *negative_opt*, *object*, *args*)

Wrapper function. *options* is a list of `(long_option, short_option, help_string)` 3-tuples as described in the constructor for `FancyGetopt`. *negative_opt* should be a dictionary mapping option names to option names, both the key and value should be in the *options* list. *object* is an object which will be used to store values (see the `getopt()` method of the `FancyGetopt` class). *args* is the argument list. Will use `sys.argv[1:]` if you pass `None` as *args*.

`distutils.fancy_getopt.`**`wrap_text`**(*text*, *width*)

Wraps *text* to less than *width* wide.

class `distutils.fancy_getopt.`**`FancyGetopt`**([*option_table=None*])

The option_table is a list of 3-tuples: `(long_option, short_option, help_string)`

If an option takes an argument, its *long_option* should have `'='` appended; *short_option* should just be a single character, no `':'` in any case. *short_option* should be `None` if a *long_option* doesn't have a corresponding *short_option*. All option tuples must have long options.

The `FancyGetopt` class provides the following methods:

`FancyGetopt.`**`getopt`**([*args=None*, *object=None*])

Parse command-line options in args. Store as attributes on *object*.

If *args* is `None` or not supplied, uses `sys.argv[1:]`. If *object* is `None` or not supplied, creates a new `OptionDummy` instance, stores option values there, and returns a tuple `(args, object)`. If *object* is supplied, it is modified in place and `getopt()` just returns *args*; in both cases, the returned *args* is a modified copy of the passed-in *args* list, which is left untouched.

`FancyGetopt.`**`get_option_order`**()

Returns the list of `(option, value)` tuples processed by the previous run of `getopt()` Raises `RuntimeError` if `getopt()` hasn't been called yet.

`FancyGetopt.`**`generate_help`**([*header=None*])

Generate help text (a list of strings, one per suggested line of output) from the option table for this `FancyGetopt` object.

If supplied, prints the supplied *header* at the top of the help.

10.17. `distutils.filelist` — The FileList class

This module provides the `FileList` class, used for poking about the filesystem and building lists of files.

10.18. `distutils.log` — Simple PEP 282-style logging

10.19. `distutils.spawn` — Spawn a sub-process

This module provides the `spawn()` function, a front-end to various platform-specific functions for launching another program in a sub-process. Also provides `find_executable()` to search the path for a given executable name.

10.20. `distutils.sysconfig` — System configuration information

The `distutils.sysconfig` module provides access to Python's low-level configuration information. The specific configuration variables available depend heavily on the platform and configuration. The specific variables depend on the build process for the specific version of Python being run; the variables are those found in the `Makefile` and configuration header that are installed with Python on Unix systems. The configuration header is called `pyconfig.h` for Python versions starting with 2.2, and `config.h` for earlier versions of Python.

Some additional functions are provided which perform some useful manipulations for other parts of the `distutils` package.

`distutils.sysconfig.`**`PREFIX`**

The result of `os.path.normpath(sys.prefix)`.

`distutils.sysconfig.`**`EXEC_PREFIX`**

The result of `os.path.normpath(sys.exec_prefix)`.

`distutils.sysconfig.`**`get_config_var`**(*name*)

Return the value of a single variable. This is equivalent to `get_config_vars().get(name)`.

`distutils.sysconfig.`**`get_config_vars`**(*...*)

Return a set of variable definitions. If there are no arguments, this returns a dictionary mapping names of configuration variables to values. If arguments are provided, they should be strings, and the return value will be a sequence giving the associated values. If a given name does not have a corresponding value, `None` will be included for that variable.

`distutils.sysconfig.`**`get_config_h_filename`**()

Return the full path name of the configuration header. For Unix, this will be the header generated by the **configure** script; for other platforms the header will have been supplied directly by the Python source distribution. The file is a platform-specific text file.

`distutils.sysconfig.`**`get_makefile_filename`**()

Return the full path name of the `Makefile` used to build Python. For Unix, this will be a file generated by the **configure** script; the meaning for other platforms will vary. The file is a platform-specific text file, if it exists. This function is only useful on POSIX platforms.

`distutils.sysconfig.`**`get_python_inc`**([*plat_specific*[, *prefix*]])

Return the directory for either the general or platform-dependent C include files. If *plat_specific* is true, the platform-dependent include directory is returned; if false or omitted, the platform-independent directory is returned. If *prefix* is given, it is used as either the prefix instead of `PREFIX`, or as the exec-prefix instead of `EXEC_PREFIX` if *plat_specific* is true.

`distutils.sysconfig.`**`get_python_lib`**([*plat_specific*[, *standard_lib*[, *prefix*]]])

Return the directory for either the general or platform-dependent library installation. If *plat_specific* is true, the platform-dependent include directory is returned; if false or omitted, the platform-independent directory is returned. If *prefix* is given, it is used as either the prefix instead of `PREFIX`, or as the exec-prefix instead of `EXEC_PREFIX` if *plat_specific* is true. If *standard_lib* is true, the directory for the standard library is returned rather than the directory for the installation of third-party extensions.

The following function is only intended for use within the `distutils` package.

`distutils.sysconfig.`**`customize_compiler`**(*compiler*)

Do any platform-specific customization of a `distutils.ccompiler.CCompiler` instance.

This function is only needed on Unix at this time, but should be called consistently to support forward-compatibility. It inserts the information that varies across Unix flavors and is stored in Python's `Makefile`. This information includes the selected compiler, compiler and linker options, and the extension used by the linker for shared objects.

This function is even more special-purpose, and should only be used from Python's own build procedures.

`distutils.sysconfig.`**`set_python_build`**()

Inform the `distutils.sysconfig` module that it is being used as part of the build process for Python. This changes a lot of relative locations for files, allowing them to be located in the build area rather than in an installed Python.

10.21. `distutils.text_file` — The TextFile class

This module provides the `TextFile` class, which gives an interface to text files that (optionally) takes care of stripping comments, ignoring blank lines, and joining lines with backslashes.

class `distutils.text_file.`**`TextFile`**([*filename=None*, *file=None*, ***options*])

This class provides a file-like object that takes care of all the things you commonly want to do when processing a text file that has some line-by-line syntax: strip comments (as long as `#` is your comment character), skip blank lines, join adjacent lines by escaping the newline (ie. backslash at end of line), strip leading and/or trailing whitespace. All of these are optional and independently controllable.

The class provides a `warn()` method so you can generate warning messages that report physical line number, even if the logical line in question spans multiple physical lines. Also provides `unreadline()` for implementing line-at-a-time lookahead.

`TextFile` instances are create with either *filename*, *file*, or both. `RuntimeError` is raised if both are `None`. *filename* should be a string, and *file* a file object (or something that provides `readline()` and `close()` methods). It is recommended that you supply at least *filename*, so that `TextFile` can include it in warning messages. If *file* is not supplied, `TextFile` creates its own using the `open()` built-in function.

The options are all boolean, and affect the values returned by `readline()`

option name	**description**	**default**
strip_comments		true

option name	description	default
	strip from `'#'` to end-of- line, as well as any whitespace leading up to the `'#'`—unless it is escaped by a backslash	
lstrip_ws	strip leading whitespace from each line before returning it	false
rstrip_ws	strip trailing whitespace (including line terminator!) from each line before returning it.	true
skip_blanks	skip lines that are empty *after* stripping comments and whitespace. (If both lstrip_ws and rstrip_ws are false, then some lines may consist of solely whitespace: these will *not* be skipped, even if *skip_blanks* is true.)	true
join_lines	if a backslash is the last non-newline character on a line after stripping comments and whitespace, join the following line to it to form one logical line; if N consecutive lines end with a backslash, then N+1 physical lines will be joined to form one logical line.	false
collapse_join	strip leading whitespace from lines that are joined to their predecessor; only matters if `(join_lines and not lstrip_ws)`	false

Note that since *rstrip_ws* can strip the trailing newline, the semantics of `readline()` must differ from those of the built-in file object's `readline()` method! In particular, `readline()` returns `None` for end-of-file: an empty string might just be a blank line (or an all-whitespace line), if *rstrip_ws* is true but *skip_blanks* is not.

`open`(*filename*)

Open a new file *filename*. This overrides any *file* or *filename* constructor arguments.

`close`()

Close the current file and forget everything we know about it (including the filename and the current line number).

warn(*msg*[, *line=None*])

Print (to stderr) a warning message tied to the current logical line in the current file. If the current logical line in the file spans multiple physical lines, the warning refers to the whole range, such as `"lines 3-5"`. If *line* is supplied, it overrides the current line number; it may be a list or tuple to indicate a range of physical lines, or an integer for a single physical line.

readline()

Read and return a single logical line from the current file (or from an internal buffer if lines have previously been “unread” with `unreadline()`). If the *join_lines* option is true, this may involve reading multiple physical lines concatenated into a single string. Updates the current line number, so calling `warn()` after `readline()` emits a warning about the physical line(s) just read. Returns `None` on end-of-file, since the empty string can occur if *rstrip_ws* is true but *strip_blanks* is not.

readlines()

Read and return the list of all logical lines remaining in the current file. This updates the current line number to the last line of the file.

unreadline(*line*)

Push *line* (a string) onto an internal buffer that will be checked by future `readline()` calls. Handy for implementing a parser with line-at-a-time lookahead. Note that lines that are “unread” with `unreadline()` are not subsequently re-cleansed (whitespace stripped, or whatever) when read with `readline()`. If multiple calls are made to `unreadline()` before a call to `readline()`, the lines will be returned most in most recent first order.

10.22. `distutils.version` — Version number classes

10.23. `distutils.cmd` — Abstract base class for Distutils commands

This module supplies the abstract base class `Command`.

class `distutils.cmd.`**Command**(*dist*)

Abstract base class for defining command classes, the “worker bees” of the Distutils. A useful analogy for command classes is to think of them as subroutines with local variables called *options*. The options are declared in `initialize_options()` and defined (given their final values) in `finalize_options()`, both of which must be defined by every command class.

The distinction between the two is necessary because option values might come from the outside world (command line, config file, …), and any options dependent on other options must be computed after these outside influences have been processed — hence `finalize_options()`. The body of the subroutine, where it does all its work based on the values of its options, is the `run()` method, which must also be implemented by every command class.

The class constructor takes a single argument *dist*, a `Distribution` instance.

10.24. Creating a new Distutils command

This section outlines the steps to create a new Distutils command.

A new command lives in a module in the `distutils.command` package. There is a sample template in that directory called `command_template`. Copy this file to a new module with the same name as the new command you're implementing. This module should implement a class with the same name as the module (and the command). So, for instance, to create the command `peel_banana` (so that users can run `setup.py peel_banana`), you'd copy `command_template` to `distutils/command/peel_banana.py`, then edit it so that it's implementing the class `peel_banana`, a subclass of `distutils.cmd.Command`.

Subclasses of `Command` must define the following methods.

`Command.`**`initialize_options`**()

Set default values for all the options that this command supports. Note that these defaults may be overridden by other commands, by the setup script, by config files, or by the command-line. Thus, this is not the place to code dependencies between options; generally, `initialize_options()` implementations are just a bunch of `self.foo = None` assignments.

`Command.`**`finalize_options`**()

Set final values for all the options that this command supports. This is always called as late as possible, ie. after any option assignments from the command-line or from other commands have been done. Thus, this is the place to code option dependencies: if *foo* depends on *bar*, then it is safe to set *foo* from *bar* as long as *foo* still has the same value it was assigned in `initialize_options()`.

`Command.`**`run`**()

A command's raison d'etre: carry out the action it exists to perform, controlled by the options initialized in `initialize_options()`, customized by other commands, the setup script, the command-line, and config files, and finalized in `finalize_options()`. All terminal output and filesystem interaction should be done by `run()`.

Command.**sub_commands**

sub_commands formalizes the notion of a “family” of commands, e.g. `install` as the parent with sub-commands `install_lib`, `install_headers`, etc. The parent of a family of commands defines *sub_commands* as a class attribute; it’s a list of 2-tuples `(command_name, predicate)`, with *command_name* a string and *predicate* a function, a string or `None`. *predicate* is a method of the parent command that determines whether the corresponding command is applicable in the current situation. (E.g. `install_headers` is only applicable if we have any C header files to install.) If *predicate* is `None`, that command is always applicable.

sub_commands is usually defined at the *end* of a class, because predicates can be methods of the class, so they must already have been defined. The canonical example is the **install** command.

10.25. `distutils.command` — Individual Distutils commands

10.26. `distutils.command.bdist` — Build a binary installer

10.27. `distutils.command.bdist_packager` — Abstract base class for packagers

10.28. `distutils.command.bdist_dumb` — Build a “dumb” installer

10.29. `distutils.command.bdist_msi` — Build a Microsoft Installer binary package

class distutils.command.bdist_msi.**bdist_msi**

Builds a Windows Installer (.msi) binary package.

In most cases, the `bdist_msi` installer is a better choice than the `bdist_wininst` installer, because it provides better support for Win64 platforms, allows administrators to perform non-interactive installations, and allows installation through group policies.

10.30. distutils.command.bdist_rpm — Build a binary distribution as a Redhat RPM and SRPM

10.31. distutils.command.bdist_wininst — Build a Windows installer

10.32. distutils.command.sdist — Build a source distribution

10.33. distutils.command.build — Build all files of a package

10.34. distutils.command.build_clib — Build any C libraries in a package

10.35. distutils.command.build_ext — Build any extensions in a package

10.36. distutils.command.build_py — Build the .py/.pyc files of a package

class distutils.command.build_py.**build_py**

class distutils.command.build_py.**build_py_2to3**

Alternative implementation of build_py which also runs the 2to3 conversion library on each .py file that is going to be installed. To use this in a setup.py file for a distribution that is designed to run with both Python 2.x and 3.x, add:

```
try:
    from distutils.command.build_py import build_py_2to3 as build_
except ImportError:
    from distutils.command.build_py import build_py
```

to your setup.py, and later:

```
cmdclass = {'build_py': build_py}
```

to the invocation of setup().

10.37. `distutils.command.build_scripts` — Build the scripts of a package

10.38. `distutils.command.clean` — Clean a package build area

This command removes the temporary files created by **build** and its subcommands, like intermediary compiled object files. With the `--all` option, the complete build directory will be removed.

Extension modules built in place will not be cleaned, as they are not in the build directory.

10.39. `distutils.command.config` — Perform package configuration

10.40. `distutils.command.install` — Install a package

10.41. `distutils.command.install_data` — Install data files from a package

10.42. `distutils.command.install_headers` — Install C/C++ header files from a package

10.43. `distutils.command.install_lib` — Install library files from a package

10.44. `distutils.command.install_scripts` — Install script files from a package

10.45. `distutils.command.register` — Register a module with the Python Package Index

The `register` command registers the package with the Python Package Index. This is described in more detail in **PEP 301**.

10.46. `distutils.command.check` — Check the meta-data of a package

The `check` command performs some tests on the meta-data of a package. For example, it verifies that all required meta-data are provided as the arguments passed to the `setup()` function.