# Keywords

The following are keywords `setuptools.setup()` accepts. They allow configuring the build process for a Python distribution or adding metadata via a `setup.py` script placed at the root of your project. All of them are optional; you do not have to supply them unless you need the associated `setuptools` feature.

Metadata and configuration supplied via `setup()` is complementary to (and may be overwritten by) the information present in `setup.cfg` and `pyproject.toml`. Some important metadata, such as `name` and `version`, may assume a default *degenerate* value if not specified.

Users are strongly encouraged to use a declarative config either via setup.cfg or pyproject.toml and only rely on `setup.py` if they need to tap into special behaviour that requires scripting (such as building C extensions).

**Note**

When using declarative configs via `pyproject.toml` with `setuptools<64.0.0`, users can still keep a very simple `setup.py` just to ensure editable installs are supported, for example:

```
from setuptools import setup

setup()
```

Versions of `setuptools` `>=64.0.0` do not require this extra minimal `setup.py` file.

`name`
: A string specifying the name of the package.

`version`
: A string specifying the version number of the package.

`description`
: A string describing the package in a single line.

`long_description`
: A string providing a longer description of the package.

`long_description_content_type`
: A string specifying the content type is used for the `long_description` (e.g. `text/markdown`)

`author`
: A string specifying the author of the package.

`author_email`
: A string specifying the email address of the package author.

`maintainer`
: A string specifying the name of the current maintainer, if different from the author. Note that if the maintainer is provided, setuptools will use it as the author in `PKG-INFO`.

`maintainer_email`
: A string specifying the email address of the current maintainer, if different from the author.

`url`
: A string specifying the URL for the package homepage.

`download_url`
: A string specifying the URL to download the package.

`packages`
: A list of strings specifying the packages that setuptools will manipulate.

`py_modules`
: A list of strings specifying the modules that setuptools will manipulate.

`scripts`
: A list of strings specifying the standalone script files to be built and installed.

`ext_package`
: A string specifying the base package name for the extensions provided by this package.

`ext_modules`
: A list of instances of `setuptools.Extension` providing the list of Python extensions to be built.

`classifiers`
: A list of strings describing the categories for the package.

`distclass`
: A subclass of `Distribution` to use.

`script_name`
: A string specifying the name of the setup.py script – defaults to `sys.argv[0]`

`script_args`
: A list of strings defining the arguments to supply to the setup script.

`options`
: A dictionary providing the default options for the setup script.

`license`
: A string specifying the license of the package.

`license_file`

> **Warning**
>
> `license_file` is deprecated. Use `license_files` instead.

`license_files`

A list of glob patterns for license related files that should be included. If neither `license_file` nor `license_files` is specified, this option defaults to `LICEN[CS]E*`, `COPYING*`, `NOTICE*`, and `AUTHORS*`.

`keywords`

A list of strings or a comma-separated string providing descriptive meta-data. See: Core Metadata Specifications.

`platforms`

A list of strings or comma-separated string.

`cmdclass`

A dictionary providing a mapping of command names to `Command` subclasses.

`data_files`

> **Warning**
>
> `data_files` is deprecated. It does not work with wheels, so it should be avoided.

A list of strings specifying the data files to install.

`package_dir`

A dictionary that maps package names (as they will be imported by the end-users) into directory paths (that actually exist in the project's source tree). This configuration has two main purposes:

1. To effectively "rename" paths when building your package. For example, `package_dir={"mypkg": "dir1/dir2/code_for_mypkg"}` will instruct setuptools to copy the `dir1/dir2/code_for_mypkg/...` files as `mypkg/...` when building the final wheel distribution.

   > **Attention**
   >
   > While it is *possible* to specify arbitrary mappings, developers are **STRONGLY ADVISED AGAINST** that. They should try as much as possible to keep the directory names and hierarchy identical to the way they will appear in the final wheel, only deviating when absolutely necessary.

2. To indicate that the relevant code is entirely contained inside a specific directory (instead of directly placed under the project's root). In this case, a special key is required (the empty string, `""`), for example: `package_dir={"": "<name of the container directory>"}`.

All the directories inside the container directory will be copied directly into the final wheel distribution, but the container directory itself will not.

This practice is very common in the community to help separate the package implementation from auxiliary files (e.g. CI configuration files), and is referred to as src-layout, because the container directory is commonly named `src`.

All paths in `package_dir` must be relative to the project root directory and use a forward slash (`/`) as path separator regardless of the operating system.

> **Tip**
>
> When using package discovery together with setup.cfg or pyproject.toml, it is very likely that you don't need to specify a value for `package_dir`. Please have a look at the definitions of src-layout and flat-layout to learn common practices on how to design a project's directory structure and minimise the amount of configuration that is needed.

`requires`

> **Warning**
>
> `requires` is superseded by `install_requires` and should not be used anymore.

`obsoletes`

> **Warning**
>
> `obsoletes` is currently ignored by `pip`.

List of strings describing packages which this package renders obsolete, meaning that the two projects should not be installed at the same time.

Version declarations can be supplied. Version numbers must be in the format specified in Version specifiers (e.g. `foo (<3.0)`).

This field may be followed by an environment marker after a semicolon (e.g. `foo; os_name == "posix"`)

The most common use of this field will be in case a project name changes, e.g. Gorgon 2.3 gets subsumed into Torqued Python 1.0. When you install Torqued Python, the Gorgon distribution should be removed.

`provides`

> **Warning**
>
> `provides` is currently ignored by `pip`.

List of strings describing package- and virtual package names contained within this package.

A package may provide additional names, e.g. to indicate that multiple projects have been bundled together. For instance, source distributions of the ZODB project have historically included the transaction project, which is now available as a separate distribution. Installing such a source distribution satisfies requirements for both ZODB and transaction.

A package may also provide a "virtual" project name, which does not correspond to any separately-distributed project: such a name might be used to indicate an abstract capability which could be supplied by one of multiple projects. E.g., multiple projects might supply RDBMS bindings for use by a given ORM: each project might declare that it provides ORM-bindings, allowing other projects to depend only on having at most one of them installed.

A version declaration may be supplied and must follow the rules described in Version specifiers. The distribution's version number will be implied if none is specified (e.g. `foo (<3.0)`).

Each package may be followed by an environment marker after a semicolon (e.g. `foo; os_name == "posix"`).

`include_package_data`

If set to `True`, this tells `setuptools` to automatically include any data files it finds inside your package directories that are specified by your `MANIFEST.in` file. For more information, see the section on Including Data Files.

`exclude_package_data`

A dictionary mapping package names to lists of glob patterns that should be *excluded* from your package directories. You can use this to trim back any excess files included by `include_package_data`. For a complete description and examples, see the section on Including Data Files.

`package_data`

A dictionary mapping package names to lists of glob patterns. For a complete description and examples, see the section on Including Data Files. You do not need to use this option if you are using `include_package_data`, unless you need to add e.g. files that are generated by your setup script and build process. (And are therefore not in source control or are files that you don't want to include in your source distribution.)

`zip_safe`

A boolean (True or False) flag specifying whether the project can be safely installed and run from a zip file. If this argument is not supplied, the `bdist_egg` command will have to analyze all of your project's contents for possible problems each time it builds an egg.

`install_requires`

A string or list of strings specifying what other distributions need to be installed when this one is. See the section on Declaring required dependency for details and examples of the format of this argument.

`entry_points`

A dictionary mapping entry point group names to strings or lists of strings defining the entry points. Entry points are used to support dynamic discovery of services or plugins provided by a project. See Advertising Behavior for details and examples of the format of this argument. In addition, this keyword is used to support Automatic Script Creation.

`extras_require`

A dictionary mapping names of "extras" (optional features of your project) to strings or lists of strings specifying what other distributions must be installed to support those features. See the section on Declaring required dependency for details and examples of the format of this argument.

`python_requires`

A string corresponding to a version specifier (as defined in PEP 440) for the Python version, used to specify the Requires-Python defined in PEP 345.

`setup_requires`

> **Warning**
>
> Using `setup_requires` is discouraged in favor of **PEP 518**.

A string or list of strings specifying what other distributions need to be present in order for the *setup script* to run. `setuptools` will attempt to obtain these before processing the rest of the setup script or commands. This argument is needed if you are using distutils extensions as part of your build process; for example, extensions that process setup() arguments and turn them into EGG-INFO metadata files.

(Note: projects listed in `setup_requires` will NOT be automatically installed on the system where the setup script is being run. They are simply downloaded to the ./.eggs directory if they're not locally available already. If you want them to be installed, as well as being available when the setup script is run, you should add them to `install_requires` **and** `setup_requires`.)

`dependency_links`

> **Warning**
>
> `dependency_links` is deprecated. It is not supported anymore by pip.

A list of strings naming URLs to be searched when satisfying dependencies. These links will be used if needed to install packages specified by `setup_requires` or `tests_require`. They will also be written into the egg's metadata for use during install by tools that support them.

`namespace_packages`

> **Warning**
>
> `namespace_packages` is deprecated in favor of native/implicit namespaces (**PEP 420**). Check the Python Packaging User Guide for more information.

A list of strings naming the project's "namespace packages". A namespace package is a package that may be split across multiple project distributions. For example, Zope 3's `zope` package is a namespace package, because subpackages like `zope.interface` and `zope.publisher` may be distributed separately. The egg runtime system can automatically merge such subpackages into a single parent package at runtime, as long as you declare them in each project that contains any subpackages of the namespace package, and as long as the namespace package's `__init__.py` does not contain any code other than a namespace declaration. See the section on Finding namespace packages for more information.

`test_suite`

A string naming a `unittest.TestCase` subclass (or a package or module containing one or more of them, or a method of such a subclass), or naming a function that can be called with no arguments and returns a `unittest.TestSuite`. If the named suite is a module, and the module has an `additional_tests()` function, it is called and the results are added to the tests to be run. If the named suite is a package, any submodules and subpackages are recursively added to the overall test suite.

Specifying this argument enables use of the test command to run the specified test suite, e.g. via `setup.py test`. See the section on the test command below for more details.

> **Warning**
>
> *Deprecated since version 41.5.0:* The test command will be removed in a future version of `setuptools`, alongside any test configuration parameter.

`tests_require`

If your project's tests need one or more additional packages besides those needed to install it, you can use this option to specify them. It should be a string or list of strings specifying what other distributions need to be present for the package's tests to run. When you run the `test` command, `setuptools` will attempt to obtain these. Note that these required projects will *not* be installed on the system where the tests are run, but only downloaded to the project's setup directory if they're not already installed locally.

> **Warning**
>
> *Deprecated since version 41.5.0:* The test command will be removed in a future version of `setuptools`, alongside any test configuration parameter.

`test_loader`

If you would like to use a different way of finding tests to run than what setuptools normally uses, you can specify a module name and class name in this argument. The

named class must be instantiable with no arguments, and its instances must support the `loadTestsFromNames()` method as defined in the Python `unittest` module's `TestLoader` class. Setuptools will pass only one test "name" in the `names` argument: the value supplied for the `test_suite` argument. The loader you specify may interpret this string in any way it likes, as there are no restrictions on what may be contained in a `test_suite` string.

The module name and class name must be separated by a `:`. The default value of this argument is `"setuptools.command.test:ScanningLoader"`. If you want to use the default `unittest` behavior, you can specify `"unittest:TestLoader"` as your `test_loader` argument instead. This will prevent automatic scanning of submodules and subpackages.

The module and class you specify here may be contained in another package, as long as you use the `tests_require` option to ensure that the package containing the loader class is available when the `test` command is run.

> **Warning**
>
> *Deprecated since version 41.5.0:* The test command will be removed in a future version of `setuptools`, alongside any test configuration parameter.

`eager_resources`

A list of strings naming resources that should be extracted together, if any of them is needed, or if any C extensions included in the project are imported. This argument is only useful if the project will be installed as a zipfile, and there is a need to have all of the listed resources be extracted to the filesystem *as a unit*. Resources listed here should be '/'-separated paths, relative to the source root, so to list a resource `foo.png` in package `bar.baz`, you would include the string `bar/baz/foo.png` in this argument.

If you only need to obtain resources one at a time, or you don't have any C extensions that access other files in the project (such as data files or shared libraries), you probably do NOT need this argument and shouldn't mess with it. For more details on how this argument works, see the section below on Automatic Resource Extraction.

`project_urls`

An arbitrary map of URL names to hyperlinks, allowing more extensible documentation of where various resources can be found than the simple `url` and `download_url` options provide.