cookiecutter Documentation

Release 2.6.0

Audrey M. Roy Greenfeld

Mar 02, 2026

CONTENTS

Cookiecutter creates projects from **cookiecutters** (project templates), e.g. Python package projects from Python package templates.

CHAPTER

ONE

BASICS

1.1 Cookiecutter

Create projects swiftly from **cookiecutters** (project templates) with this command-line utility. Ideal for generating Python package projects and more.

- Documentation
- GitHub
- PyPI
- License (BSD)

1.1.1 Installation

Install Cookiecutter as a CLI tool with uv:

```
uv tool install cookiecutter
```

1.1.2 Features

- **Cross-Platform:** Supports Windows, Mac, and Linux.
- **User-Friendly:** No Python knowledge required.
- **Versatile:** Compatible with Python 3.10 to 3.14.
- **Multi-Language Support:** Use templates in any language or markup format.

For Users

Quickstart

The most common way to use Cookiecutter is as a command line utility with a GitHub-hosted Cookiecutter template such as https://github.com/audreyfeldroy/cookiecutter-pypackage

Use a GitHub-hosted Cookiecutter template

```
# You'll be prompted to enter values.
# Then it'll create your Python package in the current working directory,
# based on those values.
# For the sake of brevity, repos on GitHub can just use the 'gh' prefix
$ uvx cookiecutter gh:audreyfeldroy/cookiecutter-pypackage
```

Use a local template

```
$ uvx cookiecutter cookiecutter-pypackage/
```

Use it from Python

If you plan to use Cookiecutter programmatically, please run `uv add cookiecutter` to add it to your project. Then you can import and use it like this:

```
from cookiecutter.main import cookiecutter

# Create project from the cookiecutter-pypackage/ template
cookiecutter('cookiecutter-pypackage/')

# Create project from the cookiecutter-pypackage.git repo template
cookiecutter('gh:audreyfeldroy/cookiecutter-pypackage')
```

If Cookiecutter saves you time, star it on GitHub so other developers can find it too.

Detailed Usage

- Generate projects from local or remote templates.
- Customize projects with `cookiecutter.json` prompts.
- Utilize pre-prompt, pre- and post-generate hooks.

Learn More

For Template Creators

- **Any language, any framework.** A Cookiecutter template is just a directory with variables. It works for Python, Rust, Terraform, docs sites, whatever you build repeatedly.
- **Hooks for the rest of the setup.** Pre- and post-generate scripts (shell or Python) handle git init, dependency installs, or anything else your boilerplate needs.
- **One file defines the interface.** `cookiecutter.json` declares every variable and its default. Users answer prompts; the template does the rest.

Learn More

1.1.3 Available Templates

Discover a variety of ready-to-use templates on GitHub.

Special Templates

- cookiecutter-pypackage
- cookiecutter-django
- cookiecutter-pytest-plugin
- cookiecutter-plone-starter

1.1.4 Community

Join the community, contribute, or seek assistance.

- Troubleshooting Guide
- Stack Overflow
- Discord
- File an Issue
- *Contributors*
- *Contribution Guide*

Feedback

We value your feedback. Share your criticisms or complaints constructively to help us improve.

- File an Issue

Waiting for a Response?

- Be patient and consider reaching out to the community for assistance.
- For enterprise support, contact support@feldroy.com.

1.1.5 Code of Conduct

Adhere to the PyPA Code of Conduct during all interactions in the project's ecosystem.

1.1.6 Acknowledgements

Created and led by Audrey M. Roy Greenfeld, supported by a dedicated team of maintainers and contributors.

1.2 Overview

Cookiecutter takes a template provided as a directory structure with template-files. Templates can be located in the filesystem, as a ZIP-file or on a VCS-Server (Git/Hg) like GitHub.

It reads a settings file and prompts the user interactively whether or not to change the settings.

Then it takes both and generates an output directory structure from it.

Additionally the template can provide code (Python or shell-script) to be executed before and after generation (pre-gen- and post-gen-hooks).

1.2.1 Input

This is a directory structure for a simple cookiecutter:

```
cookiecutter-something/
├── {{ cookiecutter.project_name }}/  <--------- Project template
│   └── ...
├── blah.txt                      <--------- Non-templated files/dirs
│                                            go outside
│
└── cookiecutter.json             <--------- Prompts & default values
```

You must have:

- A `cookiecutter.json` file.
- A `{{ cookiecutter.project_name }}/` directory, where `project_name` is defined in your `cookiecutter.json`.

Beyond that, you can have whatever files/directories you want.

See https://github.com/audreyfeldroy/cookiecutter-pypackage for a real-world example of this.

1.2.2 Output

This is what will be generated locally, in your current directory:

```
mysomething/  <---------- Value corresponding to what you enter at the
│                         project_name prompt
│
└── ...       <-------- Files corresponding to those in your
                        cookiecutter's `{{ cookiecutter.project_name }}/` dir
```

1.3 Installation

1.3.1 Prerequisites

- Python interpreter
- Adjust your path
- Packaging tools

Python interpreter

Install Python for your operating system. On Windows and macOS this is usually necessary. Most Linux distributions come with Python pre-installed. Consult the official Python documentation for details.

You can install the Python binaries from python.org. Alternatively on macOS, you can use the homebrew package manager.

```
brew install python3
```

Adjust your path

Ensure that your `bin` folder is on your path for your platform. Typically `~/.local/` for UNIX and macOS, or `%APPDATA%\Python` on Windows. (See the Python documentation for site.USER_BASE for full details.)

UNIX and macOS

For bash shells, add the following to your `.bash_profile` (adjust for other shells):

```
# Add ~/.local/ to PATH
export PATH=$HOME/.local/bin:$PATH
```

Remember to load changes with `source ~/.bash_profile` or open a new shell session.

Windows

Ensure the directory where cookiecutter will be installed is in your environment's `Path` in order to make it possible to invoke it from a command prompt. To do so, search for "Environment Variables" on your computer (on Windows 10, it is under `System Properties` –> `Advanced`) and add that directory to the `Path` environment variable, using the GUI to edit path segments.

Example segments should look like `%APPDATA%\Python\Python3x\Scripts`, where you have your version of Python instead of `Python3x`.

You may need to restart your command prompt session to load the environment variables.

See also

See Configuring Python (on Windows) for full details.

Unix on Windows

You may also install Windows Subsystem for Linux or GNU utilities for Win32 to use Unix commands on Windows.

Packaging tools

See the Python Packaging Authority's (PyPA) documentation Requirements for Installing Packages for full details.

1.3.2 Install cookiecutter

At the command line:

```
python3 -m pip install --user cookiecutter
```

Or, if you do not have pip:

```
easy_install --user cookiecutter
```

Though, pip is recommended, easy_install is deprecated.

Or, if you are using conda, first add conda-forge to your channels:

```
conda config --add channels conda-forge
```

Once the conda-forge channel has been enabled, cookiecutter can be installed with:

```
conda install cookiecutter
```

1.3.3 Alternate installations

Homebrew (Mac OS X only):

```
brew install cookiecutter
```

Void Linux:

```
xbps-install cookiecutter
```

Pipx (Linux, OSX and Windows):

```
pipx install cookiecutter
```

1.3.4 Upgrading

from 0.6.4 to 0.7.0 or greater

First, read the release notes in detail. There are a lot of major changes. The big ones are:

- Cookiecutter no longer deletes the cloned repo after generating a project.
- Cloned repos are saved into *~/.cookiecutters/*.
- You can optionally create a *~/.cookiecutterrc* config file.

Or with pip:

```
python3 -m pip install --upgrade cookiecutter
```

Upgrade Cookiecutter either with easy_install (deprecated):

```
easy_install --upgrade cookiecutter
```

Then you should be good to go.

1.4 Usage

1.4.1 Grab a Cookiecutter template

First, clone a Cookiecutter project template:

```
$ git clone https://github.com/audreyfeldroy/cookiecutter-pypackage.git
```

1.4.2 Make your changes

Modify the variables defined in *cookiecutter.json*.

Open up the skeleton project. If you need to change it around a bit, do so.

You probably also want to create a repo, name it differently, and push it as your own new Cookiecutter project template, for handy future use.

1.4.3 Generate your project

Then generate your project from the project template:

```
$ cookiecutter cookiecutter-pypackage/
```

The only argument is the input directory. (The output directory is generated by rendering that, and it can't be the same as the input directory.)

> **Note**
>
> see *Command Line Options* for extra command line arguments

Try it out!

1.4.4 Works directly with git and hg (mercurial) repos too

To create a project from the cookiecutter-pypackage.git repo template:

```
$ cookiecutter gh:audreyfeldroy/cookiecutter-pypackage
```

Cookiecutter knows abbreviations for Github (`gh`), Bitbucket (`bb`), and GitLab (`gl`) projects, but you can also give it the full URL to any repository:

```
$ cookiecutter https://github.com/audreyfeldroy/cookiecutter-pypackage.git
$ cookiecutter git+ssh://git@github.com/audreyfeldroy/cookiecutter-pypackage.git
$ cookiecutter hg+ssh://hg@bitbucket.org/audreyr/cookiecutter-pypackage
```

You will be prompted to enter a bunch of project config values. (These are defined in the project's *cookiecutter.json*.)

Then, Cookiecutter will generate a project from the template, using the values that you entered. It will be placed in your current directory.

And if you want to specify a branch you can do that with:

```
$ cookiecutter https://github.com/audreyfeldroy/cookiecutter-pypackage.git --checkout␣
↪develop
```

1.4.5 Works with private repos

If you want to work with repos that are not hosted in github or bitbucket you can indicate explicitly the type of repo that you want to use prepending *hg+* or *git+* to repo url:

```
$ cookiecutter hg+https://example.com/repo
```

In addition, one can provide a path to the cookiecutter stored on a local server:

```
$ cookiecutter file://server/folder/project.git
```

1.4.6 Works with Zip files

You can also distribute cookiecutter templates as Zip files. To use a Zip file template, point cookiecutter at a Zip file on your local machine:

```
$ cookiecutter /path/to/template.zip
```

Or, if the Zip file is online:

```
$ cookiecutter https://example.com/path/to/template.zip
```

If the template has already been downloaded, or a template with the same name has already been downloaded, you will be prompted to delete the existing template before proceeding.

The Zip file contents should be the same as a git/hg repository for a template - that is, the zipfile should unpack into a top level directory that contains the name of the template. The name of the zipfile doesn't have to match the name of the template - for example, you can label a zipfile with a version number, but omit the version number from the directory inside the Zip file.

If you want to see an example Zipfile, find any Cookiecutter repository on Github and download that repository as a zip file - Github repository downloads are in a valid format for Cookiecutter.

Password-protected Zip files

If your repository Zip file is password protected, Cookiecutter will prompt you for that password whenever the template is used.

Alternatively, if you want to use a password-protected Zip file in an automated environment, you can export the *COOKIECUTTER_REPO_PASSWORD* environment variable; the value of that environment variable will be used whenever a password is required.

1.4.7 Keeping your cookiecutters organized

As of the Cookiecutter 0.7.0 release:

- Whenever you generate a project with a cookiecutter, the resulting project is output to your current directory.
- Your cloned cookiecutters are stored by default in your *~/.cookiecutters/* directory (or Windows equivalent). The location is configurable: see *User Config* for details.

Pre-0.7.0, this is how it worked:

- Whenever you generate a project with a cookiecutter, the resulting project is output to your current directory.
- Cloned cookiecutters were not saved locally.

1.5 Command Line Options

1.5.1 cookiecutter

Create a project from a Cookiecutter project template (TEMPLATE).

Cookiecutter is free and open source software, developed and managed by volunteers. If you would like to help out or fund the project, please get in touch at https://github.com/cookiecutter/cookiecutter.

Usage

```
cookiecutter [OPTIONS] [TEMPLATE] [EXTRA_CONTEXT]...
```

Options

-V, --version

Show the version and exit.

--no-input

Do not prompt for parameters and only use cookiecutter.json file content. Defaults to deleting any cached resources and redownloading them. Cannot be combined with the –replay flag.

-c, --checkout <checkout>

branch, tag or commit to checkout after git clone

--directory <directory>

Directory within repo that holds cookiecutter.json file for advanced repositories with multi templates in it

-v, --verbose

Print debug information

--replay

Do not prompt for parameters and only use information entered previously. Cannot be combined with the –no-input flag or with extra configuration passed.

--replay-file <replay_file>
: Use this file for replay instead of the default.

-f, --overwrite-if-exists
: Overwrite the contents of the output directory if it already exists

-s, --skip-if-file-exists
: Skip the files in the corresponding directories if they already exist

-o, --output-dir <output_dir>
: Where to output the generated project dir into

--config-file <config_file>
: User configuration file

--default-config
: Do not load a config file. Use the defaults instead

--debug-file <debug_file>
: File to be used as a stream for DEBUG logging

--accept-hooks <accept_hooks>
: Accept pre/post hooks

  Options
  yes | ask | no

-l, --list-installed
: List currently installed templates.

--keep-project-on-failure
: Do not delete project folder on failure

Arguments

TEMPLATE
: Optional argument

EXTRA_CONTEXT
: Optional argument(s)

1.6 Tutorials

Tutorials by @audreyfeldroy

1.6.1 Getting to Know Cookiecutter

> **Note**
>
> Before you begin, please install Cookiecutter 0.7.0 or higher. Instructions are in *Installation*.

Cookiecutter is a tool for creating projects from *cookiecutters* (project templates).

What exactly does this mean? Read on!

Case Study: cookiecutter-pypackage

cookiecutter-pypackage is a cookiecutter template that creates the starter boilerplate for a Python package.

> **Note**
>
> There are several variations of it, but for this tutorial we'll use the original version at https://github.com/audreyfeldroy/cookiecutter-pypackage/.

Step 1: Generate a Python Package Project

Open your shell and cd into the directory where you'd like to create a starter Python package project.

At the command line, run the cookiecutter command, passing in the link to cookiecutter-pypackage's HTTPS clone URL like this:

```
$ cookiecutter https://github.com/audreyfeldroy/cookiecutter-pypackage.git
```

Local Cloning of Project Template

First, cookiecutter-pypackage gets cloned to *~/.cookiecutters/* (or equivalent on Windows). Cookiecutter does this for you, so sit back and wait.

Local Generation of Project

When cloning is complete, you will be prompted to enter a bunch of values, such as *full_name*, *email*, and *project_name*. Either enter your info, or simply press return/enter to accept the default values.

This info will be used to fill in the blanks for your project. For example, your name and the year will be placed into the LICENSE file.

Step 2: Explore What Got Generated

In your current directory, you should see that a project got generated:

```
$ ls
boilerplate
```

Looking inside the *boilerplate/* (or directory corresponding to your *project_slug*) directory, you should see something like this:

```
$ ls boilerplate/
AUTHORS.rst      MANIFEST.in      docs             tox.ini
CONTRIBUTING.rst Makefile         requirements.txt
HISTORY.rst      README.rst       setup.py
LICENSE          boilerplate      tests
```

That's your new project!

If you open the AUTHORS.rst file, you should see something like this:

```
=======
Credits
=======
```

(continues on next page)

(continued from previous page)

```
Development Lead
----------------

* Audrey M. Roy Greenfeld <audreyroygreenfeld@example.com>

Contributors
------------

None yet. Why not be the first?
```

Notice how it was auto-populated with your (or my) name and email.

Also take note of the fact that you are looking at a ReStructuredText file. Cookiecutter can generate a project with text files of any type.

Great, you just generated a skeleton Python package. How did that work?

Step 3: Observe How It Was Generated

Let's take a look at cookiecutter-pypackage together. Open https://github.com/audreyfeldroy/cookiecutter-pypackage in a new browser window.

{{ cookiecutter.project_slug }}

Find the directory called *{{ cookiecutter.project_slug }}*. Click on it. Observe the files inside of it. You should see that this directory and its contents corresponds to the project that you just generated.

This happens in *find.py*, where the *find_template()* method looks for the first jinja-like directory name that starts with *cookiecutter*.

AUTHORS.rst

Look at the raw version of *{{ cookiecutter.project_slug }}/AUTHORS.rst*, at https://raw.github.com/audreyfeldroy/cookiecutter-pypackage/master/%7B%7Bcookiecutter.project_slug%7D%7D/AUTHORS.rst.

Observe how it corresponds to the *AUTHORS.rst* file that you generated.

cookiecutter.json

Now navigate back up to *cookiecutter-pypackage/* and look at the *cookiecutter.json* file.

You should see JSON that corresponds to the prompts and default values shown earlier during project generation:

```
{
    "full_name": "Audrey M. Roy Greenfeld",
    "email": "audreyroygreenfeld@example.com",
    "github_username": "audreyr",
    "project_name": "Python Boilerplate",
    "project_slug": "{{ cookiecutter.project_name.lower().replace(' ', '_') }}",
    "project_short_description": "Python Boilerplate contains all the boilerplate you␣
↪need to create a Python package.",
    "pypi_username": "{{ cookiecutter.github_username }}",
    "version": "0.1.0",
    "use_pytest": "n",
    "use_pypi_deployment_with_travis": "y",
```

(continues on next page)

(continued from previous page)

```
    "create_author_file": "y",
    "open_source_license": ["MIT", "BSD", "ISCL", "Apache Software License 2.0", "Not␣
↪open source"]
}
```

Questions?

If anything needs better explanation, please take a moment to file an issue at https://github.com/audreyfeldroy/cookiecutter/issues with what could be improved about this tutorial.

Summary

You have learned how to use Cookiecutter to generate your first project from a cookiecutter project template.

In tutorial 2 (*Create a Cookiecutter From Scratch*), you'll see how to create cookiecutters of your own, from scratch.

1.6.2 Create a Cookiecutter From Scratch

In this tutorial, we are creating *cookiecutter-website-simple*, a cookiecutter for generating simple, bare-bones websites.

Step 1: Name Your Cookiecutter

Create the directory for your cookiecutter and cd into it:

```
$ mkdir cookiecutter-website-simple
$ cd cookiecutter-website-simple/
```

Step 2: Create cookiecutter.json

cookiecutter.json is a JSON file that contains fields which can be referenced in the cookiecutter template. For each, default value is defined and user will be prompted for input during cookiecutter execution. Only mandatory field is *project_slug* and it should comply with package naming conventions defined in PEP8 Naming Conventions .

```
{
  "project_name": "Cookiecutter Website Simple",
  "project_slug": "{{ cookiecutter.project_name.lower().replace(' ', '_') }}",
  "author": "Anonymous"
}
```

Step 3: Create project_slug Directory

Create a directory called *{{ cookiecutter.project_slug }}*.

This value will be replaced with the repo name of projects that you generate from this cookiecutter.

Step 4: Create index.html

Inside of *{{ cookiecutter.project_slug }}*, create *index.html* with following content:

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>{{ cookiecutter.project_name }}</title>
```

(continues on next page)

(continued from previous page)

```
    </head>

    <body>
        <h1>{{ cookiecutter.project_name }}</h1>
        <p>by {{ cookiecutter.author }}</p>
    </body>
</html>
```

Step 5: Pack cookiecutter into ZIP

There are many ways to run Cookiecutter templates, and they are described in details in Usage chapter. In this tutorial we are going to ZIP cookiecutter and then run it for testing.

By running following command *cookiecutter.zip* will get generated which can be used to run cookiecutter. Script will generate *cookiecutter.zip* ZIP file and echo full path to the file.

```
$ (SOURCE_DIR=$(basename $PWD) ZIP=cookiecutter.zip && # Set variables
pushd .. && # Set parent directory as working directory
zip -r $ZIP $SOURCE_DIR --exclude $SOURCE_DIR/$ZIP --quiet && # ZIP cookiecutter
mv $ZIP $SOURCE_DIR/$ZIP && # Move ZIP to original directory
popd && # Restore original work directory
echo  "Cookiecutter full path: $PWD/$ZIP")
```

Step 6: Run cookiecutter

Set your work directory to whatever directory you would like to run cookiecutter at. Use cookiecutter full path and run the following command:

```
$ cookiecutter <replace with Cookiecutter full path>
```

You can expect similar output:

```
$ cookiecutter /Users/admin/cookiecutter-website-simple/cookiecutter.zip
project_name [Cookiecutter Website Simple]: Test web
project_slug [test_web]:
author [Anonymous]: Cookiecutter Developer
```

Resulting directory should be inside your work directory with a name that matches *project_slug* you defined. Inside that directory there should be *index.html* with generated source:

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Test web</title>
    </head>

    <body>
        <h1>Test web</h1>
        <p>by Cookiecutter Developer</p>
    </body>
</html>
```

1.6.3 External Links

- Learn the Basics of Cookiecutter by Creating a Cookiecutter - first steps tutorial with example template by @BruceEckel
- Project Templates Made Easy by @pydanny
- Cookiedozer Tutorials by @hackebrot
  - Part 1: Create your own Cookiecutter template
  - Part 2: Extending our Cookiecutter template
  - Part 3: Wrapping up our Cookiecutter template

1.7 Advanced Usage

Various advanced topics regarding cookiecutter usage.

1.7.1 Hooks

Cookiecutter hooks are scripts executed at specific stages during the project generation process. They are either Python or shell scripts, facilitating automated tasks like data validation, pre-processing, and post-processing. These hooks are instrumental in customizing the generated project structure and executing initial setup tasks.

Types of Hooks

Hook	Execution Timing	Working Directory	Template Variables	Version
pre_prompt	Before any question is rendered.	A copy of the repository directory	No	2.4.0
pre_gen_project	After questions, before template process.	Root of the generated project	Yes	0.7.0
post_gen_projec	After the project generation.	Root of the generated project	Yes	0.7.0

Creating Hooks

Hooks are added to the `hooks/` folder of your template. Both Python and Shell scripts are supported.

Python Hooks Structure:

```
cookiecutter-something/
├── {{cookiecutter.project_slug}}/
├── hooks
│   ├── pre_prompt.py
│   ├── pre_gen_project.py
│   └── post_gen_project.py
└── cookiecutter.json
```

Shell Scripts Structure:

```
cookiecutter-something/
├── {{cookiecutter.project_slug}}/
├── hooks
│   ├── pre_prompt.sh
```

(continues on next page)

(continued from previous page)

```
    ├── pre_gen_project.sh
    └── post_gen_project.sh
└── cookiecutter.json
```

Python scripts are recommended for cross-platform compatibility. However, shell scripts or *.bat* files can be used for platform-specific templates.

Hook Execution

Hooks should be robust and handle errors gracefully. If a hook exits with a nonzero status, the project generation halts, and the generated directory is cleaned.

Working Directory:

- `pre_prompt`: Scripts run in the root directory of a copy of the repository directory. That allows the rewrite of `cookiecutter.json` to your own needs.
- `pre_gen_project` and `post_gen_project`: Scripts run in the root directory of the generated project, simplifying the process of locating generated files using relative paths.

Template Variables:

The `pre_gen_project` and `post_gen_project` hooks support Jinja template rendering, similar to project templates. For instance:

```
module_name = '{{ cookiecutter.module_name }}'
```

Examples

Pre-Prompt Sanity Check:

A `pre_prompt` hook, like the one below in `hooks/pre_prompt.py`, ensures prerequisites, such as Docker, are installed before prompting the user.

```
import sys
import subprocess

def is_docker_installed() -> bool:
    try:
        subprocess.run(["docker", "--version"], capture_output=True, check=True)
        return True
    except Exception:
        return False

if __name__ == "__main__":
    if not is_docker_installed():
        print("ERROR: Docker is not installed.")
        sys.exit(1)
```

Validating Template Variables:

A `pre_gen_project` hook can validate template variables. The following script checks if the provided module name is valid.

```
import re
import sys
```

(continues on next page)

(continued from previous page)

```
MODULE_REGEX = r'^[_a-zA-Z][_a-zA-Z0-9]+$'
module_name = '{{ cookiecutter.module_name }}'

if not re.match(MODULE_REGEX, module_name):
    print(f'ERROR: {module_name} is not a valid Python module name!')
    sys.exit(1)
```

Conditional File/Directory Removal:

A `post_gen_project` hook can conditionally control files and directories. The example below removes unnecessary files based on the selected packaging option.

```
import os

REMOVE_PATHS = [
    '{% if cookiecutter.packaging != "pip" %}requirements.txt{% endif %}',
    '{% if cookiecutter.packaging != "poetry" %}poetry.lock{% endif %}',
]

for path in REMOVE_PATHS:
    path = path.strip()
    if path and os.path.exists(path):
        os.unlink(path) if os.path.isfile(path) else os.rmdir(path)
```

1.7.2 User Config

New in Cookiecutter 0.7

If you use Cookiecutter a lot, you'll find it useful to have a user config file. By default Cookiecutter tries to retrieve settings from a *.cookiecutterrc* file in your home directory.

New in Cookiecutter 1.3

You can also specify a config file on the command line via `--config-file`.

```
cookiecutter --config-file /home/audreyr/my-custom-config.yaml cookiecutter-pypackage
```

Or you can set the `COOKIECUTTER_CONFIG` environment variable:

```
export COOKIECUTTER_CONFIG=/home/audreyr/my-custom-config.yaml
```

If you wish to stick to the built-in config and not load any user config file at all, use the CLI option `--default-config` instead. Preventing Cookiecutter from loading user settings is crucial for writing integration tests in an isolated environment.

Example user config:

```
default_context:
    full_name: "Audrey M. Roy Greenfeld"
    email: "audreyroygreenfeld@example.com"
    github_username: "audreyfeldroy"
cookiecutters_dir: "/home/audreyr/my-custom-cookiecutters-dir/"
replay_dir: "/home/audreyr/my-custom-replay-dir/"
abbreviations:
```

(continues on next page)

(continued from previous page)

```
    pp: https://github.com/audreyfeldroy/cookiecutter-pypackage.git
    gh: https://github.com/{0}.git
    bb: https://bitbucket.org/{0}
```

Possible settings are:

default_context:
A list of key/value pairs that you want injected as context whenever you generate a project with Cookiecutter. These values are treated like the defaults in `cookiecutter.json`, upon generation of any project.

cookiecutters_dir
Directory where your cookiecutters are cloned to when you use Cookiecutter with a repo argument.

replay_dir
Directory where Cookiecutter dumps context data to, which you can fetch later on when using the *replay feature*.

abbreviations
A list of abbreviations for cookiecutters. Abbreviations can be simple aliases for a repo name, or can be used as a prefix, in the form `abbr:suffix`. Any suffix will be inserted into the expansion in place of the text `{0}`, using standard Python string formatting. With the above aliases, you could use the `cookiecutter-pypackage` template simply by saying `cookiecutter pp`, or `cookiecutter gh:audreyr/cookiecutter-pypackage`. The `gh` (GitHub), `bb` (Bitbucket), and `gl` (Gitlab) abbreviations shown above are actually **built in**, and can be used without defining them yourself.

Read also: *Injecting Extra Context*

1.7.3 Calling Cookiecutter Functions From Python

You can use Cookiecutter from Python:

```
from cookiecutter.main import cookiecutter

# Create project from the cookiecutter-pypackage/ template
cookiecutter('cookiecutter-pypackage/')

# Create project from the cookiecutter-pypackage.git repo template
cookiecutter('https://github.com/audreyfeldroy/cookiecutter-pypackage.git')
```

This is useful if, for example, you're writing a web framework and need to provide developers with a tool similar to *django-admin.py startproject* or *npm init*.

See the *API Reference* for more details.

1.7.4 Injecting Extra Context

You can specify an `extra_context` dictionary that will override values from `cookiecutter.json` or `.cookiecutterrc`:

```
cookiecutter(
    'cookiecutter-pypackage/',
    extra_context={'project_name': 'TheGreatest'},
)
```

This works as command-line parameters as well:

```
cookiecutter --no-input cookiecutter-pypackage/ project_name=TheGreatest
```

You will also need to add these keys to the `cookiecutter.json` or `.cookiecutterrc`.

Example: Injecting a Timestamp

If you have `cookiecutter.json` that has the following keys:

```
{
    "timestamp": "{{ cookiecutter.timestamp }}"
}
```

This Python script will dynamically inject a timestamp value as the project is generated:

```
from cookiecutter.main import cookiecutter

from datetime import datetime

cookiecutter(
    'cookiecutter-django',
    extra_context={'timestamp': datetime.utcnow().isoformat()}
)
```

How this works:

1. The script uses `datetime` to get the current UTC time in ISO format.
2. To generate the project, `cookiecutter()` is called, passing the timestamp in as context via the `extra_context` dict.

1.7.5 Suppressing Command-Line Prompts

To suppress the prompts asking for input, use `no_input`.

Note: this option will force a refresh of cached resources.

Basic Example: Using the Defaults

Cookiecutter will pick a default value if used with `no_input`:

```
from cookiecutter.main import cookiecutter
cookiecutter(
    'cookiecutter-django',
    no_input=True,
)
```

In this case it will be using the default defined in `cookiecutter.json` or `.cookiecutterrc`.

> **Note**
>
> values from `cookiecutter.json` will be overridden by values from `.cookiecutterrc`

Advanced Example: Defaults + Extra Context

If you combine an `extra_context` dict with the `no_input` argument, you can programmatically create the project with a set list of context parameters and without any command line prompts:

```
cookiecutter('cookiecutter-pypackage/',
             no_input=True,
             extra_context={'project_name': 'TheGreatest'})
```

See also *Injecting Extra Context* and the *API Reference* for more details.

1.7.6 Templates in Context Values

The values (but not the keys!) of *cookiecutter.json* are also Jinja2 templates. Values from user prompts are added to the context immediately, such that one context value can be derived from previous values. This approach can potentially save your user a lot of keystrokes by providing more sensible defaults.

Basic Example: Templates in Context

Python packages show some patterns for their naming conventions:

- a human-readable project name
- a lowercase, dashed repository name
- an importable, dash-less package name

Here is a *cookiecutter.json* with templated values for this pattern:

```
{
  "project_name": "My New Project",
  "project_slug": "{{ cookiecutter.project_name|lower|replace(' ', '-') }}",
  "pkg_name": "{{ cookiecutter.project_slug|replace('-', '') }}"
}
```

If the user takes the defaults, or uses *no_input*, the templated values will be:

- *my-new-project*
- *mynewproject*

Or, if the user gives *Yet Another New Project*, the values will be:

- `yet-another-new-project`
- `yetanothernewproject`

1.7.7 Private Variables

Cookiecutter allows the definition private variables by prepending an underscore to the variable name. The user will not be required to fill those variables in. These can either be not rendered, by using a prepending underscore, or rendered, prepending a double underscore. For example, the `cookiecutter.json`:

```
{
    "project_name": "Really cool project",
    "_not_rendered": "{{ cookiecutter.project_name|lower }}",
    "__rendered": "{{ cookiecutter.project_name|lower }}"
}
```

Will be rendered as:

```
{
    "project_name": "Really cool project",
    "_not_rendered": "{{ cookiecutter.project_name|lower }}",
    "__rendered": "really cool project"
}
```

The user will only be asked for `project_name`.

Non-rendered private variables can be used for defining constants. An example of where you may wish to use private **rendered** variables is creating a Python package repository and want to enforce naming consistency. To ensure the repository and package name are based on the project name, you could create a `cookiecutter.json` such as:

```
{
    "project_name": "Project Name",
    "__project_slug": "{{ cookiecutter.project_name|lower|replace(' ', '-') }}",
    "__package_name": "{{ cookiecutter.project_name|lower|replace(' ', '_') }}",
}
```

Which could create a structure like this:

```
project-name
├── Makefile
├── README.md
├── requirements.txt
└── src
    ├── project_name
    │   └── __init__.py
    ├── setup.py
    └── tests
        └── __init__.py
```

The `README.md` can then have a plain English project title.

1.7.8 Copy without Render

New in Cookiecutter 1.1

To avoid rendering directories and files of a cookiecutter, the `_copy_without_render` key can be used in the `cookiecutter.json`. The value of this key accepts a list of Unix shell-style wildcards:

```
{
    "project_slug": "sample",
    "_copy_without_render": [
        "*.html",
        "*not_rendered_dir",
        "rendered_dir/not_rendered_file.ini"
    ]
}
```

Note: Only the content of the files will be copied without being rendered. The paths are subject to rendering. This allows you to write:

```
{
    "project_slug": "sample",
    "_copy_without_render": [
```

(continues on next page)

(continued from previous page)

```
        "{{cookiecutter.repo_name}}/templates/*.html",
    ]
}
```

In this example, `{{cookiecutter.repo_name}}` will be rendered as expected but the html file content will be copied without rendering.

1.7.9 Replay Project Generation

New in Cookiecutter 1.1

On invocation **Cookiecutter** dumps a json file to `~/.cookiecutter_replay/` which enables you to *replay* later on.

In other words, it persists your **input** for a template and fetches it when you run the same template again.

Example for a replay file (which was created via `cookiecutter gh:hackebrot/cookiedozer`):

```
{
    "cookiecutter": {
        "app_class_name": "FooBarApp",
        "app_title": "Foo Bar",
        "email": "raphael@example.com",
        "full_name": "Raphael Pierzina",
        "github_username": "hackebrot",
        "kivy_version": "1.8.0",
        "project_slug": "foobar",
        "short_description": "A sleek slideshow app that supports swipe gestures.",
        "version": "0.1.0",
        "year": "2015"
    }
}
```

To fetch this context data without being prompted on the command line you can use either of the following methods.

Pass the according option on the CLI:

```
cookiecutter --replay gh:hackebrot/cookiedozer
```

Or use the Python API:

```
from cookiecutter.main import cookiecutter
cookiecutter('gh:hackebrot/cookiedozer', replay=True)
```

This feature comes in handy if, for instance, you want to create a new project from an updated template.

Custom replay file

New in Cookiecutter 2.0

To specify a custom filename, you can use the `--replay-file` option:

```
cookiecutter --replay-file ./cookiedozer.json gh:hackebrot/cookiedozer
```

This may be useful to run the same replay file over several machines, in tests or when a user of the template reports a problem.

1.7.10 Choice Variables

New in Cookiecutter 1.1

Choice variables provide different choices when creating a project. Depending on a user's choice the template renders things differently.

Basic Usage

Choice variables are regular key / value pairs, but with the value being a list of strings.

For example, if you provide the following choice variable in your `cookiecutter.json`:

```
{
    "license": ["MIT", "BSD-3", "GNU GPL v3.0", "Apache Software License 2.0"]
}
```

you'd get the following choices when running Cookiecutter:

```
Select license:
1 - MIT
2 - BSD-3
3 - GNU GPL v3.0
4 - Apache Software License 2.0
Choose from 1, 2, 3, 4 [1]:
```

Depending on an user's choice, a different license is rendered by Cookiecutter.

The above `license` choice variable creates `cookiecutter.license`, which can be used like this:

```
{%- if cookiecutter.license == "MIT" -%}
# Possible license content here

{%- elif cookiecutter.license == "BSD-3" -%}
# More possible license content here

{% endif %}
```

Cookiecutter is using Jinja2's if conditional expression to determine the correct license.

The created choice variable is still a regular Cookiecutter variable and can be used like this:

```
License
-------

Distributed under the terms of the `{{cookiecutter.license}}`_ license,
```

Overwriting Default Choice Values

Choice Variables are overwritable using a *User Config* file.

For example, a choice variable can be created in `cookiecutter.json` by using a list as value:

```
{
    "license": ["MIT", "BSD-3", "GNU GPL v3.0", "Apache Software License 2.0"]
}
```

By default, the first entry in the values list serves as default value in the prompt.

Setting the default `license` agreement to *Apache Software License 2.0* can be done using:

```
default_context:
    license: "Apache Software License 2.0"
```

in the *User Config* file.

The resulting prompt changes and looks like:

```
Select license:
1 - Apache Software License 2.0
2 - MIT
3 - BSD-3
4 - GNU GPL v3.0
Choose from 1, 2, 3, 4 [1]:
```

Note

As you can see the order of the options changed from `1 - MIT` to `1 - Apache Software License 2.0`. **Cookiecutter** takes the first value in the list as the default.

1.7.11 Boolean Variables

Added in version 2.2.0.

Boolean variables are used for answering True/False questions.

Basic Usage

Boolean variables are regular key / value pairs, but with the value being `True`/`False`.

For example, if you provide the following boolean variable in your `cookiecutter.json`:

```
{
    "run_as_docker": true
}
```

you will get the following user input when running Cookiecutter:

```
run_as_docker [True]:
```

User input will be parsed by *read_user_yes_no()*. The following values are considered as valid user input:

- `True` values: “1”, “true”, “t”, “yes”, “y”, “on”
- `False` values: “0”, “false”, “f”, “no”, “n”, “off”

The above `run_as_docker` boolean variable creates `cookiecutter.run_as_docker`, which can be used like this:

```
{%- if cookiecutter.run_as_docker -%}
# In case of True add your content here

{%- else -%}
# In case of False add your content here
```

(continues on next page)

(continued from previous page)

```
{% endif %}
```

Cookiecutter is using Jinja2's if conditional expression to determine the correct `run_as_docker`.

Input Validation

If a non valid value is inserted to a boolean field, the following error will be printed:

```
run_as_docker [True]: docker
Error: docker is not a valid boolean
```

1.7.12 Dictionary Variables

New in Cookiecutter 1.5

Dictionary variables provide a way to define deep structured information when rendering a template.

Basic Usage

Dictionary variables are, as the name suggests, dictionaries of key-value pairs. The dictionary values can, themselves, be other dictionaries and lists - the data structure can be as deep as you need.

For example, you could provide the following dictionary variable in your `cookiecutter.json`:

```
{
    "project_slug": "new_project",
    "file_types": {
        "png": {
            "name": "Portable Network Graphic",
            "library": "libpng",
            "apps": [
                "GIMP"
            ]
        },
        "bmp": {
            "name": "Bitmap",
            "library": "libbmp",
            "apps": [
                "Paint",
                "GIMP"
            ]
        }
    }
}
```

The above `file_types` dictionary variable creates `cookiecutter.file_types`, which can be used like this:

```
{% for extension, details in cookiecutter.file_types|dictsort %}
<dl>
  <dt>Format name:</dt>
  <dd>{{ details.name }}</dd>

  <dt>Extension:</dt>
```

(continues on next page)

(continued from previous page)

```
  <dd>{{ extension }}</dd>

  <dt>Applications:</dt>
  <dd>
      <ul>
      {% for app in details.apps -%}
          <li>{{ app }}</li>
      {% endfor -%}
      </ul>
  </dd>
</dl>
{% endfor %}
```

Cookiecutter is using Jinja2's for expression to iterate over the items in the dictionary.

1.7.13 Templates inheritance (2.2+)

New in Cookiecutter 2.2+

Sometimes you need to extend a base template with a different configuration to avoid nested blocks.

Cookiecutter introduces the ability to use common templates using the power of jinja: *extends*, *include* and *super*.

Here's an example repository:

```
https://github.com/user/repo-name.git
├── {{cookiecutter.project_slug}}/
|   └── file.txt
├── templates/
|   └── base.txt
└── cookiecutter.json
```

every file in the *templates* directory will become referable inside the project itself, and the path should be relative from the *templates* folder like

```
# file.txt
{% extends "base.txt" %}

... or ...

# file.txt
{% include "base.txt" %}
```

see more on https://jinja.palletsprojects.com/en/2.11.x/templates/

1.7.14 Template Extensions

New in Cookiecutter 1.4

A template may extend the Cookiecutter environment with custom Jinja2 extensions. It can add extra filters, tests, globals or even extend the parser.

To do so, a template author must specify the required extensions in `cookiecutter.json` as follows:

```
{
    "project_slug": "Foobar",
    "year": "{% now 'utc', '%Y' %}",
    "_extensions": ["jinja2_time.TimeExtension"]
}
```

On invocation Cookiecutter tries to import the extensions and add them to its environment respectively.

In the above example, Cookiecutter provides the additional tag now, after installing the jinja2_time.TimeExtension and enabling it in `cookiecutter.json`.

Please note that Cookiecutter will **not** install any dependencies on its own! As a user you need to make sure you have all the extensions installed, before running Cookiecutter on a template that requires custom Jinja2 extensions.

By default Cookiecutter includes the following extensions:

- `cookiecutter.extensions.JsonifyExtension`
- `cookiecutter.extensions.RandomStringExtension`
- `cookiecutter.extensions.SlugifyExtension`
- `cookiecutter.extensions.TimeExtension`
- `cookiecutter.extensions.UUIDExtension`

Warning

The above is just an example to demonstrate how this is used. There is no need to require `jinja2_time.TimeExtension`, since its functionality is included by default (by `cookiecutter.extensions.TimeExtension`) without needing an extra install.

Jsonify extension

The `cookiecutter.extensions.JsonifyExtension` extension provides a `jsonify` filter in templates that converts a Python object to JSON:

```
{% {'a': True} | jsonify %}
```

Would output:

```
{"a": true}
```

It supports an optional `indent` param, the default value is `4`:

```
{% {'a': True, 'foo': 'bar'} | jsonify(2) %}
```

Would output:

```
{
  "a": true,
  "foo": "bar"
}
```

Random string extension

New in Cookiecutter 1.7

The `cookiecutter.extensions.RandomStringExtension` extension provides a `random_ascii_string` method in templates that generates a random fixed-length string, optionally with punctuation.

Generate a random n-size character string. Example for n=12:

```
{{ random_ascii_string(12) }}
```

Outputs:

```
bIIUczoNvswh
```

The second argument controls if punctuation and special characters `!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~` should be present in the result:

```
{{ random_ascii_string(12, punctuation=True) }}
```

Outputs:

```
fQupUkY}W!)!
```

Slugify extension

The `cookiecutter.extensions.SlugifyExtension` extension provides a `slugify` filter in templates that converts string into its dashed ("slugified") version:

```
{% "It's a random version" | slugify %}
```

Would output:

```
it-s-a-random-version
```

It is different from a mere replace of spaces since it also treats some special characters differently such as `'` in the example above. The function accepts all arguments that can be passed to the `slugify` function of python-slugify. For example to change the output from `it-s-a-random-version` to `it_s_a_random_version`, the `separator` parameter would be passed: `slugify(separator='_')`.

UUID4 extension

New in Cookiecutter 1.x

The `cookiecutter.extensions.UUIDExtension` extension provides a `uuid4()` method in templates that generates a uuid4.

Generate a uuid4 string:

```
{{ uuid4() }}
```

Outputs:

```
83b5de62-31b4-4a1e-83fa-8c548de65a11
```

1.7.15 Organizing cookiecutters in directories

New in Cookiecutter 1.7

Cookiecutter introduces the ability to organize several templates in one repository or zip file, separating them by directories. This allows using symlinks for general files. Here's an example repository demonstrating this feature:

```
https://github.com/user/repo-name.git
    ├── directory1-name/
    |   ├── {{cookiecutter.project_slug}}/
    |   └── cookiecutter.json
    └── directory2-name/
        ├── {{cookiecutter.project_slug}}/
        └── cookiecutter.json
```

To activate one of templates within a subdirectory, use the `--directory` option:

```
cookiecutter https://github.com/user/repo-name.git --directory="directory1-name"
```

1.7.16 Customizing the Jinja2 environment

The special template variable `_jinja2_env_vars` can be used to customize the [Jinja2 environment](https://jinja.palletsprojects.com/en/3.1.x/api/#jinja2.Environment).

This example shows how to control whitespace with `lstrip_blocks` and `trim_blocks`:

```
{
    "project_slug": "sample",
    "_jinja2_env_vars": {"lstrip_blocks": true, "trim_blocks": true}
}
```

1.7.17 Working with line-ends special symbols LF/CRLF

New in Cookiecutter 2.0

> **Note**
>
> Before version 2.0 Cookiecutter silently used system line end character. LF for POSIX and CRLF for Windows. Since version 2.0 this behaviour changed and now can be forced at template level.

By default Cookiecutter checks every file at render stage and uses the same line end as in source. This allow template developers to have both types of files in the same template. Developers should correctly configure their `.gitattributes` file to avoid line-end character overwrite by git.

The special template variable `_new_lines` enforces a specific line ending. Acceptable variables: `'\r\n'` for CRLF and `'\n'` for POSIX.

Here is example how to force line endings to CRLF on any deployment:

```
{
    "project_slug": "sample",
    "_new_lines": "\r\n"
}
```

1.7.18 Local Extensions

New in Cookiecutter 2.1

A template may extend the Cookiecutter environment with local extensions. These can be part of the template itself, providing it with more sophisticated custom tags and filters.

To do so, a template author must specify the required extensions in `cookiecutter.json` as follows:

```
{
    "project_slug": "Foobar",
    "year": "{% now 'utc', '%Y' %}",
    "_extensions": ["local_extensions.FoobarExtension"]
}
```

This example uses a simple module `local_extensions.py` which exists in the template root, containing the following (for instance):

```
from jinja2.ext import Extension


class FoobarExtension(Extension):
    def __init__(self, environment):
        super(FoobarExtension, self).__init__(environment)
        environment.filters['foobar'] = lambda v: v * 2
```

This will register the `foobar` filter for the template.

For many cases, this will be unnecessarily complicated. It's likely that we'd only want to register a single function as a filter. For this, we can use the `simple_filter` decorator:

```
{
    "project_slug": "Foobar",
    "year": "{% now 'utc', '%Y' %}",
    "_extensions": ["local_extensions.simplefilterextension"]
}
```

```
from cookiecutter.utils import simple_filter


@simple_filter
def simplefilterextension(v):
    return v * 2
```

This snippet will achieve the exact same result as the previous one.

For complex use cases, a python module `local_extensions` (a folder with an `__init__.py`) can also be created in the template root. Here, for example, a module `main.py` would have to export all extensions with `from .main import FoobarExtension, simplefilterextension` or `from .main import *` in the `__init__.py`.

1.7.19 Nested configuration files

New in Cookiecutter 2.5.0

If you wish to create a hierarchy of templates and use cookiecutter to choose among them, you need just to specify the key `templates` in the main configuration file to reach the other ones.

Let's imagine to have the following structure:

```
main-directory/
    ├── project-1
    │   ├── cookiecutter.json
    │   ├── {{cookiecutter.project_slug}}
    |   |   |   ├── ...
    ├── package
    │   ├── cookiecutter.json
    │   ├── {{cookiecutter.project_slug}}
    |   |   |   ├── ...
    └── cookiecutter.json
```

It is possible to specify in the main `cookiecutter.json` how to reach the other config files as follows:

```
{
    "templates": {
        "project-1": {
            "path": "./project-1",
            "title": "Project 1",
            "description": "A cookiecutter template for a project"
        },
        "package": {
            "path": "./package",
            "title": "Package",
            "description": "A cookiecutter template for a package"
        }
    }
}
```

Then, when `cookiecutter` is launched in the main directory it will ask to choose among the possible templates:

```
Select template:
1 - Project 1 (A cookiecutter template for a project)
2 - Package (A cookiecutter template for a package)
Choose from 1, 2 [1]:
```

Once a template is chosen, for example 1, it will continue to ask the info required by `cookiecutter.json` in the `project-1` folder, such as `project-slug`

Old Format

New in Cookiecutter 2.2.0

In the main `cookiecutter.json` add a *template* key with the following format:

```
{
    "template": [
        "Project 1 (./project-1)",
        "Project 2 (./project-2)"
    ]
}
```

Then, when `cookiecutter` is launched in the main directory it will ask to choose among the possible templates:

```
Select template:
1 - Project 1 (./project-1)
2 - Project 2 (./project-2)
Choose from 1, 2 [1]:
```

Once a template is chosen, for example 1, it will continue to ask the info required by `cookiecutter.json` in the `project-1` folder, such as `project-slug`

1.7.20 Human readable prompts

You can add human-readable prompts that will be shown to the user for each variable using the `__prompts__` key. For multiple choices questions you can also provide labels for each option.

See the following cookiecutter config as example:

```
{
    "package_name": "my-package",
    "module_name": "{{ cookiecutter.package_name.replace('-', '_') }}",
    "package_name_stylized": "{{ cookiecutter.module_name.replace('_', ' ').capitalize()␣
↪}}",
    "short_description": "A nice python package",
    "github_username": "your-org-or-username",
    "full_name": "Firstname Lastname",
    "email": "email@example.com",
    "init_git": true,
    "linting": ["ruff", "flake8", "none"],
    "__prompts__": {
        "package_name": "Select your package name",
        "module_name": "Select your module name",
        "package_name_stylized": "Stylized package name",
        "short_description": "Short description",
        "github_username": "GitHub username or organization",
        "full_name": "Author full name",
        "email": "Author email",
        "command_line_interface": "Add CLI",
        "init_git": "Initialize a git repository",
        "linting": {
            "__prompt__": "Which linting tool do you want to use?",
            "ruff": "Ruff",
            "flake8": "Flake8",
            "none": "No linting tool"
        }
    }
}
```

1.8 Troubleshooting

1.8.1 I created a cookiecutter, but it doesn't work, and I can't figure out why

- Try upgrading to Cookiecutter 0.8.0, which prints better error messages and has fixes for several common bugs.

1.8.2 I'm having trouble generating Jinja templates from Jinja templates

Make sure you escape things properly, like this:

```
{{ "{{" }}
```

Or this:

```
{% raw %}
<p>Go <a href="{{ url_for('home') }}">Home</a></p>
{% endraw %}
```

Or this:

```
{{ {{ url_for('home') }} }}
```

See https://jinja.palletsprojects.com/en/latest/templates/#escaping for more info.

You can also use the _copy_without_render key in your *cookiecutter.json* file to escape entire files and directories.

1.8.3 Other common issues

TODO: add a bunch of common new user issues here.

This document is incomplete. If you have knowledge that could help other users, adding a section or filing an issue with details would be greatly appreciated.

CHAPTER

TWO

API REFERENCE

2.1 API

This is the Cookiecutter modules API documentation.

2.1.1 cookiecutter.cli module

Main *cookiecutter* CLI.

cookiecutter.cli.**list_installed_templates**(*default_config*, *passed_config_file*)

List installed (locally cloned) templates. Use cookiecutter –list-installed.

Return type
None

cookiecutter.cli.**validate_extra_context**(*_ctx*, *_param*, *value*)

Validate extra context.

Return type
OrderedDict[str, str] | None

cookiecutter.cli.**version_msg**()

Return the Cookiecutter version, location and Python powering it.

Return type
str

2.1.2 cookiecutter.config module

Global configuration handling.

cookiecutter.config.**get_config**(*config_path*)

Retrieve the config from the specified path, returning a config dict.

Return type
dict[str, Any]

cookiecutter.config.**get_user_config**(*config_file=None*, *default_config=False*)

Return the user config as a dict.

If `default_config` is True, ignore `config_file` and return default values for the config parameters.

If `default_config` is a dict, merge values with default values and return them for the config parameters.

If a path to a `config_file` is given, that is different from the default location, load the user config from that.

Otherwise look up the config file path in the COOKIECUTTER_CONFIG environment variable. If set, load the config from this path. This will raise an error if the specified path is not valid.

If the environment variable is not set, try the default config file path before falling back to the default config values.

Return type
dict[str, Any]

cookiecutter.config.**merge_configs**(*default*, *overwrite*)

Recursively update a dict with the key/value pair of another.

Dict values that are dictionaries themselves will be updated, whilst preserving existing keys.

Return type
dict[str, Any]

2.1.3 cookiecutter.environment module

Jinja2 environment and extensions loading.

class cookiecutter.environment.**ExtensionLoaderMixin**(*, *context=None*, ***kwargs*)

Bases: object

Mixin providing sane loading of extensions specified in a given context.

The context is being extracted from the keyword arguments before calling the next parent class in line of the child.

class cookiecutter.environment.**StrictEnvironment**(***kwargs*)

Bases: *ExtensionLoaderMixin*, Environment

Create strict Jinja2 environment.

Jinja2 environment will raise error on undefined variable in template- rendering context.

2.1.4 cookiecutter.exceptions module

All exceptions used in the Cookiecutter code base are defined here.

exception cookiecutter.exceptions.**ConfigDoesNotExistException**

Bases: *CookiecutterException*

Exception for missing config file.

Raised when get_config() is passed a path to a config file, but no file is found at that path.

exception cookiecutter.exceptions.**ContextDecodingException**

Bases: *CookiecutterException*

Exception for failed JSON decoding.

Raised when a project's JSON context file can not be decoded.

exception cookiecutter.exceptions.**CookiecutterException**

Bases: Exception

Base exception class.

All Cookiecutter-specific exceptions should subclass this class.

exception cookiecutter.exceptions.**EmptyDirNameException**

Bases: *CookiecutterException*

Exception for a empty directory name.

Raised when the directory name provided is empty.

exception cookiecutter.exceptions.**FailedHookException**

Bases: *CookiecutterException*

Exception for hook failures.

Raised when a hook script fails.

exception cookiecutter.exceptions.**InvalidConfiguration**

Bases: *CookiecutterException*

Exception for invalid configuration file.

Raised if the global configuration file is not valid YAML or is badly constructed.

exception cookiecutter.exceptions.**InvalidModeException**

Bases: *CookiecutterException*

Exception for incompatible modes.

Raised when cookiecutter is called with both *no_input==True* and *replay==True* at the same time.

exception cookiecutter.exceptions.**InvalidZipRepository**

Bases: *CookiecutterException*

Exception for bad zip repo.

Raised when the specified cookiecutter repository isn't a valid Zip archive.

exception cookiecutter.exceptions.**MissingProjectDir**

Bases: *CookiecutterException*

Exception for missing generated project directory.

Raised during cleanup when remove_repo() can't find a generated project directory inside of a repo.

exception cookiecutter.exceptions.**NonTemplatedInputDirException**

Bases: *CookiecutterException*

Exception for when a project's input dir is not templated.

The name of the input directory should always contain a string that is rendered to something else, so that input_dir != output_dir.

exception cookiecutter.exceptions.**OutputDirExistsException**

Bases: *CookiecutterException*

Exception for existing output directory.

Raised when the output directory of the project exists already.

exception cookiecutter.exceptions.**RepositoryCloneFailed**

Bases: *CookiecutterException*

Exception for un-cloneable repo.

Raised when a cookiecutter template can't be cloned.

exception cookiecutter.exceptions.**RepositoryNotFound**

Bases: *CookiecutterException*

Exception for missing repo.

Raised when the specified cookiecutter repository doesn't exist.

exception cookiecutter.exceptions.**UndefinedVariableInTemplate**(*message*, *error*, *context*)

Bases: *CookiecutterException*

Exception for out-of-scope variables.

Raised when a template uses a variable which is not defined in the context.

exception cookiecutter.exceptions.**UnknownExtension**

Bases: *CookiecutterException*

Exception for un-importable extension.

Raised when an environment is unable to import a required extension.

exception cookiecutter.exceptions.**UnknownRepoType**

Bases: *CookiecutterException*

Exception for unknown repo types.

Raised if a repo's type cannot be determined.

exception cookiecutter.exceptions.**UnknownTemplateDirException**

Bases: *CookiecutterException*

Exception for ambiguous project template directory.

Raised when Cookiecutter cannot determine which directory is the project template, e.g. more than one dir appears to be a template dir.

exception cookiecutter.exceptions.**VCSNotInstalled**

Bases: *CookiecutterException*

Exception when version control is unavailable.

Raised if the version control system (git or hg) is not installed.

2.1.5 cookiecutter.extensions module

Jinja2 extensions.

class cookiecutter.extensions.**JsonifyExtension**(*environment*)

Bases: Extension

Jinja2 extension to convert a Python object to JSON.

identifier = 'cookiecutter.extensions.JsonifyExtension'

class cookiecutter.extensions.**RandomStringExtension**(*environment*)

Bases: Extension

Jinja2 extension to create a random string.

identifier = 'cookiecutter.extensions.RandomStringExtension'

class cookiecutter.extensions.**SlugifyExtension**(*environment*)

Bases: Extension

Jinja2 Extension to slugify string.

identifier = 'cookiecutter.extensions.SlugifyExtension'

class cookiecutter.extensions.**TimeExtension**(*environment*)

Bases: Extension

Jinja2 Extension for dates and times.

parse(*parser*)

Parse datetime template and add datetime value.

Return type
Output

identifier = 'cookiecutter.extensions.TimeExtension'

tags = {'now'}

if this extension parses this is the list of tags it's listening to.

class cookiecutter.extensions.**UUIDExtension**(*environment*)

Bases: Extension

Jinja2 Extension to generate uuid4 string.

identifier = 'cookiecutter.extensions.UUIDExtension'

2.1.6 cookiecutter.find module

Functions for finding Cookiecutter templates and other components.

cookiecutter.find.**find_template**(*repo_dir*, *env*)

Determine which child directory of repo_dir is the project template.

Parameters
repo_dir (Path | str) – Local directory of newly cloned repo.

Return type
Path

Returns
Relative path to project template.

2.1.7 cookiecutter.generate module

Functions for generating a project from a project template.

cookiecutter.generate.**apply_overwrites_to_context**(*context*, *overwrite_context*, ***, *in_dictionary_variable=False*)

Modify the given context in place based on the overwrite_context.

Return type
None

cookiecutter.generate.**generate_context**(*context_file='cookiecutter.json'*, *default_context=None*, *extra_context=None*)

Generate the context for a Cookiecutter project template.

Loads the JSON file as a Python object, with key being the JSON filename.

Parameters

- **context_file** (str) – JSON file containing key/value pairs for populating the cookiecutter's variables.
- **default_context** (Optional[dict[str, Any]]) – Dictionary containing config to take into account.
- **extra_context** (Optional[dict[str, Any]]) – Dictionary containing configuration overrides

Return type
dict[str, Any]

cookiecutter.generate.**generate_file**(*project_dir*, *infile*, *context*, *env*, *skip_if_file_exists=False*)

Render filename of infile as name of outfile, handle infile correctly.

Dealing with infile appropriately:

a. If infile is a binary file, copy it over without rendering.

b. If infile is a text file, render its contents and write the rendered infile to outfile.

Precondition:

When calling *generate_file()*, the root template dir must be the current working directory. Using *utils.work_in()* is the recommended way to perform this directory change.

Parameters

- **project_dir** (str) – Absolute path to the resulting generated project.
- **infile** (str) – Input file to generate the file from. Relative to the root template dir.
- **context** (dict[str, Any]) – Dict for populating the cookiecutter's variables.
- **env** (Environment) – Jinja2 template execution environment.

Return type
None

cookiecutter.generate.**generate_files**(*repo_dir*, *context=None*, *output_dir='.'*, *overwrite_if_exists=False*, *skip_if_file_exists=False*, *accept_hooks=True*, *keep_project_on_failure=False*)

Render the templates and saves them to files.

Parameters

- **repo_dir** (Path | str) – Project template input directory.
- **context** (Optional[dict[str, Any]]) – Dict for populating the template's variables.
- **output_dir** (Path | str) – Where to output the generated project dir into.
- **overwrite_if_exists** (bool) – Overwrite the contents of the output directory if it exists.
- **skip_if_file_exists** (bool) – Skip the files in the corresponding directories if they already exist

- **accept_hooks** (bool) – Accept pre and post hooks if set to *True*.
- **keep_project_on_failure** (bool) – If *True* keep generated project directory even when generation fails

Return type
str

cookiecutter.generate.**is_copy_only_path**(*path*, *context*)

Check whether the given *path* should only be copied and not rendered.

Returns True if *path* matches a pattern in the given *context* dict, otherwise False.

Parameters

- **path** (str) – A file-system path referring to a file or dir that should be rendered or just copied.
- **context** (dict[str, Any]) – cookiecutter context.

Return type
bool

cookiecutter.generate.**render_and_create_dir**(*dirname*, *context*, *output_dir*, *environment*, *overwrite_if_exists=False*)

Render name of a directory, create the directory, return its path.

Return type
tuple[Path, bool]

2.1.8 cookiecutter.hooks module

Functions for discovering and executing various cookiecutter hooks.

cookiecutter.hooks.**find_hook**(*hook_name*, *hooks_dir='hooks'*)

Return a dict of all hook scripts provided.

Must be called with the project template as the current working directory. Dict's key will be the hook/script's name, without extension, while values will be the absolute path to the script. Missing scripts will not be included in the returned dict.

Parameters

- **hook_name** (str) – The hook to find
- **hooks_dir** (str) – The hook directory in the template

Return type
list[str] | None

Returns
The absolute path to the hook script or None

cookiecutter.hooks.**run_hook**(*hook_name*, *project_dir*, *context*)

Try to find and execute a hook from the specified project directory.

Parameters

- **hook_name** (str) – The hook to execute.
- **project_dir** (Path | str) – The directory to execute the script from.
- **context** (dict[str, Any]) – Cookiecutter project context.

Return type
None

cookiecutter.hooks.**run_hook_from_repo_dir**(*repo_dir*, *hook_name*, *project_dir*, *context*, *delete_project_on_failure*)

Run hook from repo directory, clean project directory if hook fails.

Parameters

- **repo_dir** (Path | str) – Project template input directory.
- **hook_name** (str) – The hook to execute.
- **project_dir** (Path | str) – The directory to execute the script from.
- **context** (dict[str, Any]) – Cookiecutter project context.
- **delete_project_on_failure** (bool) – Delete the project directory on hook failure?

Return type
None

cookiecutter.hooks.**run_pre_prompt_hook**(*repo_dir*)

Run pre_prompt hook from repo directory.

Parameters
repo_dir (Path | str) – Project template input directory.

Return type
Path | str

cookiecutter.hooks.**run_script**(*script_path*, *cwd='.'*)

Execute a script from a working directory.

Parameters

- **script_path** (str) – Absolute path to the script to run.
- **cwd** (Path | str) – The directory to run the script from.

Return type
None

cookiecutter.hooks.**run_script_with_context**(*script_path*, *cwd*, *context*)

Execute a script after rendering it with Jinja.

Parameters

- **script_path** (Path | str) – Absolute path to the script to run.
- **cwd** (Path | str) – The directory to run the script from.
- **context** (dict[str, Any]) – Cookiecutter project template context.

Return type
None

cookiecutter.hooks.**valid_hook**(*hook_file*, *hook_name*)

Determine if a hook file is valid.

Parameters

- **hook_file** (str) – The hook file to consider for validity
- **hook_name** (str) – The hook to find

Return type
bool

Returns
The hook file validity

2.1.9 cookiecutter.log module

Module for setting up logging.

cookiecutter.log.**configure_logger**(*stream_level='DEBUG'*, *debug_file=None*)

Configure logging for cookiecutter.

Set up logging to stdout with given level. If `debug_file` is given set up logging to file with DEBUG level.

Return type
Logger

2.1.10 cookiecutter.main module

Main entry point for the *cookiecutter* command.

The code in this module is also a good example of how to use Cookiecutter as a library rather than a script.

cookiecutter.main.**cookiecutter**(*template*, *checkout=None*, *no_input=False*, *extra_context=None*, *replay=None*, *overwrite_if_exists=False*, *output_dir='.'*, *config_file=None*, *default_config=False*, *password=None*, *directory=None*, *skip_if_file_exists=False*, *accept_hooks=True*, *keep_project_on_failure=False*)

Run Cookiecutter just as if using it from the command line.

Parameters

- **template** (str) – A directory containing a project template directory, or a URL to a git repository.
- **checkout** (Optional[str]) – The branch, tag or commit ID to checkout after clone.
- **no_input** (bool) – Do not prompt for user input. Use default values for template parameters taken from *cookiecutter.json*, user config and *extra_dict*. Force a refresh of cached resources.
- **extra_context** (Optional[dict[str, Any]]) – A dictionary of context that overrides default and user configuration.
- **replay** (Union[bool, str, None]) – Do not prompt for input, instead read from saved json. If True read from the `replay_dir`. if it exists
- **overwrite_if_exists** (bool) – Overwrite the contents of the output directory if it exists.
- **output_dir** (str) – Where to output the generated project dir into.
- **config_file** (Optional[str]) – User configuration file path.
- **default_config** (bool) – Use default values rather than a config file.
- **password** (Optional[str]) – The password to use when extracting the repository.
- **directory** (Optional[str]) – Relative path to a cookiecutter template in a repository.
- **skip_if_file_exists** (bool) – Skip the files in the corresponding directories if they already exist.
- **accept_hooks** (bool) – Accept pre and post hooks if set to *True*.

- **keep_project_on_failure** (bool) – If *True* keep generated project directory even when generation fails

Return type
str

2.1.11 cookiecutter.prompt module

Functions for prompting the user for project info.

class cookiecutter.prompt.**JsonPrompt**(*prompt=''*, ***, *console=None*, *password=False*, *choices=None*, *case_sensitive=True*, *show_default=True*, *show_choices=True*)

Bases: PromptBase[dict]

A prompt that returns a dict from JSON string.

response_type

alias of dict

static process_response(*value*)

Convert choices to a dict.

Return type
dict[str, Any]

default = None

validate_error_message = '[prompt.invalid] Please enter a valid JSON string'

class cookiecutter.prompt.**YesNoPrompt**(*prompt=''*, ***, *console=None*, *password=False*, *choices=None*, *case_sensitive=True*, *show_default=True*, *show_choices=True*)

Bases: Confirm

A prompt that returns a boolean for yes/no questions.

process_response(*value*)

Convert choices to a bool.

Return type
bool

no_choices = ['0', 'false', 'f', 'no', 'n', 'off']

yes_choices = ['1', 'true', 't', 'yes', 'y', 'on']

cookiecutter.prompt.**choose_nested_template**(*context*, *repo_dir*, *no_input=False*)

Prompt user to select the nested template to use.

Parameters

- **context** (dict[str, Any]) – Source for field names and sample values.
- **repo_dir** (Path | str) – Repository directory.
- **no_input** (bool) – Do not prompt for user input and use only values from context.

Return type
str

Returns
Path to the selected template.

cookiecutter.prompt.**process_json**(*user_value*)

Load user-supplied value as a JSON dict.

Parameters
user_value (str) – User-supplied value to load as a JSON dict

cookiecutter.prompt.**prompt_and_delete**(*path*, *no_input=False*)

Ask user if it's okay to delete the previously-downloaded file/directory.

If yes, delete it. If no, checks to see if the old version should be reused. If yes, it's reused; otherwise, Cookiecutter exits.

Parameters

- **path** (Path | str) – Previously downloaded zipfile.
- **no_input** (bool) – Suppress prompt to delete repo and just delete it.

Return type
bool

Returns
True if the content was deleted

cookiecutter.prompt.**prompt_choice_for_config**(*cookiecutter_dict*, *env*, *key*, *options*, *no_input*, *prompts=None*, *prefix=''*)

Prompt user with a set of options to choose from.

Parameters
no_input (bool) – Do not prompt for user input and return the first available option.

Return type
OrderedDict[str, Any] | str

cookiecutter.prompt.**prompt_choice_for_template**(*key*, *options*, *no_input*)

Prompt user with a set of options to choose from.

Parameters
no_input (bool) – Do not prompt for user input and return the first available option.

Return type
OrderedDict[str, Any]

cookiecutter.prompt.**prompt_for_config**(*context*, *no_input=False*)

Prompt user to enter a new config.

Parameters

- **context** (*dict*) – Source for field names and sample values.
- **no_input** (bool) – Do not prompt for user input and use only values from context.

Return type
OrderedDict[str, Any]

cookiecutter.prompt.**read_repo_password**(*question*)

Prompt the user to enter a password.

Parameters
question (str) – Question to the user

Return type
str

cookiecutter.prompt.**read_user_choice**(*var_name*, *options*, *prompts=None*, *prefix=''*)

Prompt the user to choose from several options for the given variable.

The first item will be returned if no input happens.

Parameters

- **var_name** (str) – Variable as specified in the context
- **options** (*list*) – Sequence of options that are available to select from

Returns
Exactly one item of `options` that has been chosen by the user

cookiecutter.prompt.**read_user_dict**(*var_name*, *default_value*, *prompts=None*, *prefix=''*)

Prompt the user to provide a dictionary of data.

Parameters

- **var_name** (str) – Variable as specified in the context
- **default_value** – Value that will be returned if no input is provided

Returns
A Python dictionary to use in the context.

cookiecutter.prompt.**read_user_variable**(*var_name*, *default_value*, *prompts=None*, *prefix=''*)

Prompt user for variable and return the entered value or given default.

Parameters

- **var_name** (*str*) – Variable of the context to query the user
- **default_value** – Value that will be returned if no input happens

cookiecutter.prompt.**read_user_yes_no**(*var_name*, *default_value*, *prompts=None*, *prefix=''*)

Prompt the user to reply with 'yes' or 'no' (or equivalent values).

- These input values will be converted to `True`: "1", "true", "t", "yes", "y", "on"
- These input values will be converted to `False`: "0", "false", "f", "no", "n", "off"

Actual parsing done by `prompt()`; Check this function codebase change in case of unexpected behaviour.

Parameters

- **question** (*str*) – Question to the user
- **default_value** – Value that will be returned if no input happens

cookiecutter.prompt.**render_variable**(*env*, *raw*, *cookiecutter_dict*)

Render the next variable to be displayed in the user prompt.

Inside the prompting taken from the cookiecutter.json file, this renders the next variable. For example, if a project_name is "Peanut Butter Cookie", the repo_name could be be rendered with:

{{ cookiecutter.project_name.replace(" ", "_") }}.

This is then presented to the user as the default.

Parameters

- **env** (*Environment*) – A Jinja2 Environment object.
- **raw** (bool | dict[_Raw, _Raw] | list[_Raw] | str | None) – The next value to be prompted for by the user.

- **cookiecutter_dict** (*dict*) – The current context as it's gradually being populated with variables.

Return type
str

Returns
The rendered value for the default variable.

2.1.12 cookiecutter.replay module

cookiecutter.replay.

---

cookiecutter.replay.**dump**(*replay_dir*, *template_name*, *context*)

Write json data to file.

Return type
None

cookiecutter.replay.**get_file_name**(*replay_dir*, *template_name*)

Get the name of file.

Return type
str

cookiecutter.replay.**load**(*replay_dir*, *template_name*)

Read json data from file.

Return type
dict[str, Any]

2.1.13 cookiecutter.repository module

Cookiecutter repository functions.

cookiecutter.repository.**determine_repo_dir**(*template*, *abbreviations*, *clone_to_dir*, *checkout*, *no_input*, *password=None*, *directory=None*)

Locate the repository directory from a template reference.

Applies repository abbreviations to the template reference. If the template refers to a repository URL, clone it. If the template is a path to a local repository, use it.

Parameters

- **template** (str) – A directory containing a project template directory, or a URL to a git repository.
- **abbreviations** (dict[str, str]) – A dictionary of repository abbreviation definitions.
- **clone_to_dir** (Path | str) – The directory to clone the repository into.
- **checkout** (str | None) – The branch, tag or commit ID to checkout after clone.
- **no_input** (bool) – Do not prompt for user input and eventually force a refresh of cached resources.
- **password** (Optional[str]) – The password to use when extracting the repository.
- **directory** (Optional[str]) – Directory within repo where cookiecutter.json lives.

Return type
tuple[str, bool]

Returns
A tuple containing the cookiecutter template directory, and a boolean describing whether that directory should be cleaned up after the template has been instantiated.

Raises
RepositoryNotFound if a repository directory could not be found.

cookiecutter.repository.**expand_abbreviations**(*template*, *abbreviations*)

Expand abbreviations in a template name.

Parameters

- **template** (str) – The project template name.
- **abbreviations** (dict[str, str]) – Abbreviation definitions.

Return type
str

cookiecutter.repository.**is_repo_url**(*value*)

Return True if value is a repository URL.

Return type
bool

cookiecutter.repository.**is_zip_file**(*value*)

Return True if value is a zip file.

Return type
bool

cookiecutter.repository.**repository_has_cookiecutter_json**(*repo_directory*)

Determine if *repo_directory* contains a *cookiecutter.json* file.

Parameters
repo_directory (str) – The candidate repository directory.

Return type
bool

Returns
True if the *repo_directory* is valid, else False.

2.1.14 cookiecutter.utils module

Helper functions used throughout Cookiecutter.

cookiecutter.utils.**create_env_with_context**(*context*)

Create a jinja environment using the provided context.

Return type
StrictEnvironment

cookiecutter.utils.**create_tmp_repo_dir**(*repo_dir*)

Create a temporary dir with a copy of the contents of repo_dir.

Return type
Path

cookiecutter.utils.**force_delete**(*func*, *path*, *_exc_info*)

Error handler for *shutil.rmtree()* equivalent to *rm -rf*.

Usage: *shutil.rmtree(path, onerror=force_delete)* From https://docs.python.org/3/library/shutil.html#rmtree-example

Return type
None

cookiecutter.utils.**make_executable**(*script_path*)

Make *script_path* executable.

Parameters
script_path (Path | str) – The file to change

Return type
None

cookiecutter.utils.**make_sure_path_exists**(*path*)

Ensure that a directory exists.

Parameters
path (Path | str) – A directory tree path for creation.

Return type
None

cookiecutter.utils.**rmtree**(*path*)

Remove a directory and all its contents. Like rm -rf on Unix.

Parameters
path (Path | str) – A directory path.

Return type
None

cookiecutter.utils.**simple_filter**(*filter_function*)

Decorate a function to wrap it in a simplified jinja2 extension.

Return type
type[Extension]

cookiecutter.utils.**work_in**(*dirname=None*)

Context manager version of os.chdir.

When exited, returns to the working directory prior to entering.

Return type
Iterator[None]

2.1.15 cookiecutter.vcs module

Helper functions for working with version control systems.

cookiecutter.vcs.**clone**(*repo_url*, *checkout=None*, *clone_to_dir='.'*, *no_input=False*)

Clone a repo to the current directory.

Parameters

- **repo_url** (str) – Repo URL of unknown type.
- **checkout** (Optional[str]) – The branch, tag or commit ID to checkout after clone.

- **clone_to_dir** (Path | str) – The directory to clone to. Defaults to the current directory.
- **no_input** (bool) – Do not prompt for user input and eventually force a refresh of cached resources.

Return type
str

Returns
str with path to the new directory of the repository.

cookiecutter.vcs.**identify_repo**(*repo_url*)

Determine if *repo_url* should be treated as a URL to a git or hg repo.

Repos can be identified by prepending "hg+" or "git+" to the repo URL.

Parameters
repo_url (str) – Repo URL of unknown type.

Return type
tuple[Literal['git', 'hg'], str]

Returns
('git', repo_url), ('hg', repo_url), or None.

cookiecutter.vcs.**is_vcs_installed**(*repo_type*)

Check if the version control system for a repo type is installed.

Parameters
repo_type (str)

Return type
bool

2.1.16 cookiecutter.zipfile module

Utility functions for handling and fetching repo archives in zip format.

cookiecutter.zipfile.**unzip**(*zip_uri*, *is_url*, *clone_to_dir='.'*, *no_input=False*, *password=None*)

Download and unpack a zipfile at a given URI.

This will download the zipfile to the cookiecutter repository, and unpack into a temporary directory.

Parameters

- **zip_uri** (str) – The URI for the zipfile.
- **is_url** (bool) – Is the zip URI a URL or a file?
- **clone_to_dir** (Path | str) – The cookiecutter repository directory to put the archive into.
- **no_input** (bool) – Do not prompt for user input and eventually force a refresh of cached resources.
- **password** (Optional[str]) – The password to use when unpacking the repository.

Return type
str

2.1.17 Module contents

Main package for Cookiecutter.

CHAPTER

THREE

PROJECT INFO

3.1 Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

3.1.1 Types of Contributions

You can contribute in many ways:

Report Bugs

Report bugs at https://github.com/cookiecutter/cookiecutter/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- If you can, provide detailed steps to reproduce the bug.
- If you don't have steps to reproduce the bug, just note your observations in as much detail as you can. Questions to start a discussion about the issue are welcome.

Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" is open to whoever wants to implement it.

Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "please-help" is open to whoever wants to implement it.

Please do not combine multiple feature enhancements into a single pull request.

Note: this project is very conservative, so new features that aren't tagged with "please-help" might not get into core. We're trying to keep the code base small, extensible, and streamlined. Whenever possible, it's best to try and implement feature ideas as separate projects outside of the core codebase.

Write Documentation

Cookiecutter could always use more documentation, whether as part of the official Cookiecutter docs, in docstrings, or even on the web in blog posts, articles, and such.

If you want to review your changes on the documentation locally, you can do:

```
pip install --group dev
just servedocs
```

This will compile the documentation, open it in your browser and start watching the files for changes, recompiling as you save.

Submit Feedback

The best way to send feedback is to file an issue at https://github.com/cookiecutter/cookiecutter/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

3.1.2 Setting Up the Code for Local Development

Here's how to set up `cookiecutter` for local development.

1. Fork the `cookiecutter` repo on GitHub.
2. Clone your fork locally:

   ```
   git clone git@github.com:your_name_here/cookiecutter.git
   ```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

   ```
   cd cookiecutter/
   pip install -e .
   pip install --group dev
   ```

4. Create a branch for local development:

   ```
   git checkout -b name-of-your-bugfix-or-feature
   ```

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass the tests and lint check:

   ```
   just lint
   just test-all
   ```

6. Ensure that your feature or commit is fully covered by tests. Check report after regular `pytest` run. You can also run coverage only report and get html report with statement by statement highlighting:

   ```
   just coverage
   ```

   You report will be placed to `htmlcov` directory. Please do not include this directory to your commits. By default this directory in our `.gitignore` file.

7. Commit your changes and push your branch to GitHub:

```
git add .
git commit -m "Your detailed description of your changes."
git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

3.1.3 Contributor Guidelines

Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. The pull request should be contained: if it's too big consider splitting it into smaller pull requests.
3. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.md.
4. The pull request must pass all CI/CD jobs before being ready for review.
5. If one CI/CD job is failing for unrelated reasons you may want to create another PR to fix that first.

Coding Standards

- PEP8
- Functions over classes except in tests
- Quotes via http://stackoverflow.com/a/56190/5549
  - Use double quotes around strings that are used for interpolation or that are natural language messages
  - Use single quotes for small symbol-like strings (but break the rules if the strings contain quotes)
  - Use triple double quotes for docstrings and raw string literals for regular expressions even if they aren't needed.
  - Example:

```
LIGHT_MESSAGES = {
    'English': "There are %(number_of_lights)s lights.",
    'Pirate':  "Arr! Thar be %(number_of_lights)s lights."
}
def lights_message(language, number_of_lights):
    """Return a language-appropriate string reporting the light count."""
    return LIGHT_MESSAGES[language] % locals()
def is_pirate(message):
    """Return True if the given message sounds piratical."""
    return re.search(r"(?i)(arr|avast|yohoho)!", message) is not None
```

3.1.4 Testing

The project uses `pytest` as test runner.

For further information please consult the pytest usage docs.

To run a particular test class with `pytest`:

```
pytest -k TestFindHooks
```

To run some tests with names matching a string expression:

```
pytest -k generate
```

Will run all tests matching "generate", test_generate_files for example.

To run just one method:

```
pytest -k "TestFindHooks and test_find_hook"
```

To run all tests using various versions of Python, just run `just test-all`:

```
just test-all
```

This configuration file setup the pytest-cov plugin and it is an additional dependency. It generate a coverage report after the tests.

It is possible to test with specific versions of Python. To do this, the command is:

```
uv run --python=3.13 --isolated --group test -- pytest
```

This will run `pytest` with the `python3.13` interpreters.

3.1.5 Core Committer Guide

Vision and Scope

Core committers, use this section to:

- Guide your instinct and decisions as a core committer
- Limit the codebase from growing infinitely

Command-Line Accessible

- Provides a command-line utility that creates projects from cookiecutters
- Extremely easy to use without having to think too hard
- Flexible for more complex use via optional arguments

API Accessible

- Entirely function-based and stateless (Class-free by intentional design)
- Usable in pieces for developers of template generation tools

Being Jinja2-specific

- Sets a standard baseline for project template creators, facilitating reuse
- Minimizes the learning curve for those who already use Flask or Django
- Minimizes scope of Cookiecutter codebase

Extensible

Being extendable by people with different ideas for Jinja2-based project template tools.

- Entirely function-based
- Aim for statelessness
- Lets anyone write more opinionated tools

Freedom for Cookiecutter users to build and extend.

- No officially-maintained cookiecutter templates, only ones by individuals
- Commercial project-friendly licensing, allowing for private cookiecutters and private Cookiecutter-based tools

Fast and Focused

Cookiecutter is designed to do one thing, and do that one thing very well.

- Cover the use cases that the core committers need, and as little as possible beyond that :)
- Generates project templates from the command-line or API, nothing more
- Minimize internal line of code (LOC) count
- Ultra-fast project generation for high performance downstream tools

Inclusive

- Cross-platform and cross-version support are more important than features/functionality
- Fixing Windows bugs even if it's a pain, to allow for use by more beginner coders

Stable

- Aim for 100% test coverage and covering corner cases
- No pull requests will be accepted that drop test coverage on any platform, including Windows
- Conservative decisions patterned after CPython's conservative decisions with stability in mind
- Stable APIs that tool builders can rely on
- New features require a +1 from 3 core committers

VCS-Hosted Templates

Cookiecutter project templates are intentionally hosted VCS repos as-is.

- They are easily forkable
- It's easy for users to browse forks and files
- They are searchable via standard Github/Bitbucket/other search interface
- Minimizes the need for packaging-related cruft files
- Easy to create a public project template and host it for free
- Easy to collaborate

Process: Pull Requests

How to prioritize pull requests, from most to least important:

- Fixes for broken tests. Broken means broken on any supported platform or Python version.
- Extra tests to cover corner cases.
- Minor edits to docs.
- Bug fixes.
- Major edits to docs.
- Features.

Pull Requests Review Guidelines

- Think carefully about the long-term implications of the change. How will it affect existing projects that are dependent on this? If this is complicated, do we really want to maintain it forever?
- Take the time to get things right, PRs almost always require additional improvements to meet the bar for quality. **Be very strict about quality.**
- When you merge a pull request take care of closing/updating every related issue explaining how they were affected by those changes. Also, remember to add the author to `AUTHORS.md`.

Process: Issues

If an issue is a bug that needs an urgent fix, mark it for the next patch release. Then either fix it or mark as please-help.

For other issues: encourage friendly discussion, moderate debate, offer your thoughts.

New features require a +1 from 2 other core committers (besides yourself).

Process: Releasing a New Version

1. **Bump the version** and **write the changelog:**

   ```
   uv version <version>        # or: uv version --bump minor
   ```

   Then write `CHANGELOG/<version>.md`. See previous entries for the format.

2. **Commit:**

   ```
   git add pyproject.toml uv.lock CHANGELOG/
   git commit -m "Release <version>"
   ```

3. **Tag and push:**

   ```
   just tag
   ```

   This verifies you're on `main` with a clean working tree and a changelog file, creates an annotated `v*` tag from the version in `pyproject.toml`, and pushes the commit and tag to GitHub.

4. **Approve the deployment.** The tag triggers the publish workflow (`.github/workflows/publish.yml`), which builds the package, generates SLSA provenance attestations, and publishes to PyPI via trusted publishing. The `pypi` environment requires reviewer approval before the publish job runs.

5. **Create the GitHub Release.** Use the tag and paste the changelog entry as the release body.

Process: Your own code changes

All code changes, regardless of who does them, need to be reviewed and merged by someone else. This rule applies to all the core committers.

Exceptions:

- Minor corrections and fixes to pull requests submitted by others.
- While making a formal release, the release manager can make necessary, appropriate changes.
- Small documentation changes that reinforce existing subject matter. Most commonly being, but not limited to spelling and grammar corrections.

Responsibilities

- Ensure cross-platform compatibility for every change that's accepted. Windows, macOS and Linux.
- Create issues for any major changes and enhancements that you wish to make. Discuss things transparently and get community feedback.
- Don't add any classes to the codebase unless absolutely needed. Err on the side of using functions.
- Keep feature versions as small as possible, preferably one new feature per version.
- Be welcoming to newcomers and encourage diverse new contributors from all backgrounds. Look at *Code of Conduct*.

Becoming a Core Committer

Contributors may be given core commit privileges. Preference will be given to those with:

1. Past contributions to Cookiecutter and other open-source projects. Contributions to Cookiecutter include both code (both accepted and pending) and friendly participation in the issue tracker. Quantity and quality are considered.
2. A coding style that the other core committers find simple, minimal, and clean.
3. Access to resources for cross-platform development and testing.
4. Time to devote to the project regularly.

3.2 Credits

3.2.1 Development Lead

- Audrey M. Roy Greenfeld (@audreyfeldroy)

3.2.2 Past Core Committers

Huge gratitude to all who have served as a Cookiecutter core committer over the years:

- Daniel Roy Greenfeld (@pydanny)
- Raphael Pierzina (@hackebrot)
- Michael Joseph (@michaeljoseph)
- Paul Moore (@pfmoore)
- Andrey Shpak (@insspb)
- Sorin Sbarnea (@ssbarnea)

- Fábio C. Barrionuevo da Luz (@luzfcb)
- Simone Basso (@simobasso)
- Jens Klein (@jensens)
- Érico Andrei (@ericof)

3.2.3 Contributors

- Steven Loria (@sloria)
- Goran Peretin (@gperetin)
- Hamish Downer (@foobacca)
- Thomas Orozco (@krallin)
- Jindrich Smitka (@s-m-i-t-a)
- Benjamin Schwarze (@benjixx)
- Raphi (@raphigaziano)
- Thomas Chiroux (@ThomasChiroux)
- Sergi Almacellas Abellana (@pokoli)
- Alex Gaynor (@alex)
- Rolo (@rolo)
- Pablo (@oubiga)
- Bruno Rocha (@rochacbruno)
- Alexander Artemenko (@svetlyak40wt)
- Mahmoud Abdelkader (@mahmoudimus)
- Leonardo Borges Avelino (@lborgav)
- Chris Trotman (@solarnz)
- Rolf (@relekang)
- Noah Kantrowitz (@coderanger)
- Vincent Bernat (@vincentbernat)
- Germán Moya (@pbacterio)
- Ned Batchelder (@nedbat)
- Dave Dash (@davedash)
- Johan Charpentier (@cyberj)
- Éric Araujo (@merwok)
- saxix (@saxix)
- Tzu-ping Chung (@uranusjr)
- Caleb Hattingh (@cjrh)
- Flavio Curella (@fcurella)
- Adam Venturella (@aventurella)
- Monty Taylor (@emonty)

- schacki (@schacki)
- Ryan Olson (@ryanolson)
- Trey Hunner (@treyhunner)
- Russell Keith-Magee (@freakboy3742)
- Mishbah Razzaque (@mishbahr)
- Robin Andeer (@robinandeer)
- Rachel Sanders (@trustrachel)
- Rémy Hubscher (@Natim)
- Dino Petron3 (@dinopetrone)
- Peter Inglesby (@inglesp)
- Ramiro Batista da Luz (@ramiroluz)
- Omer Katz (@thedrow)
- lord63 (@lord63)
- Randy Syring (@rsyring)
- Mark Jones (@mark0978)
- Marc Abramowitz (@msabramo)
- Lucian Ursu (@LucianU)
- Osvaldo Santana Neto (@osantana)
- Matthias84 (@Matthias84)
- Simeon Visser (@svisser)
- Guruprasad (@lgp171188)
- Charles-Axel Dein (@charlax)
- Diego Garcia (@drgarcia1986)
- maiksensi (@maiksensi)
- Andrew Conti (@agconti)
- Valentin Lab (@vaab)
- Ilja Bauer (@iljabauer)
- Elias Dorneles (@eliasdorneles)
- Matias Saguir (@mativs)
- Johannes (@johtso)
- macrotim (@macrotim)
- Will McGinnis (@wdm0006)
- Cédric Krier (@cedk)
- Tim Osborn (@ptim)
- Aaron Gallagher (@habnabit)
- mozillazg (@mozillazg)

- Joachim Jablon (@ewjoachim)
- Andrew Ittner (@tephyr)
- Diane DeMers Chen (@purplediane)
- zzzirk (@zzzirk)
- Carol Willing (@willingc)
- phoebebauer (@phoebebauer)
- Adam Chainz (@adamchainz)
- Sulé (@suledev)
- Evan Palmer (@palmerev)
- Bruce Eckel (@BruceEckel)
- Robert Lyon (@ivanlyon)
- Terry Bates (@terryjbates)
- Brett Cannon (@brettcannon)
- Michael Warkentin (@mwarkentin)
- Bartłomiej Kurzeja (@B3QL)
- Thomas O'Donnell (@andytom)
- Jeremy Carbaugh (@jcarbaugh)
- Nathan Cheung (@cheungnj)
- Abdó Roig-Maranges (@aroig)
- Steve Piercy (@stevepiercy)
- Corey (@coreysnyder04)
- Dmitry Evstratov (@devstrat)
- Eyal Levin (@eyalev)
- mathagician (@mathagician)
- Guillaume Gelin (@ramnes)
- @delirious-lettuce (@delirious-lettuce)
- Gasper Vozel (@karantan)
- Joshua Carp (@jmcarp)
- @meahow (@meahow)
- Andrea Grandi (@andreagrandi)
- Issa Jubril (@jubrilissa)
- Nytiennzo Madooray (@Nythiennzo)
- Erik Bachorski (@dornheimer)
- cclauss (@cclauss)
- Andy Craze (@accraze)
- Anthony Sottile (@asottile)

- Jonathan Sick (@jonathansick)
- Hugo (@hugovk)
- Min ho Kim (@minho42)
- Ryan Ly (@rly)
- Akintola Rahmat (@mihrab34)
- Jai Ram Rideout (@jairideout)
- Diego Carrasco Gubernatis (@dacog)
- Wagner Negrão (@wagnernegrao)
- Josh Barnes (@jcb91)
- Nikita Sobolev (@sobolevn)
- Matt Stibbs (@mattstibbs)
- MinchinWeb (@MinchinWeb)
- kishan (@kishan)
- tonytheleg (@tonytheleg)
- Roman Hartmann (@RomHartmann)
- DSEnvel (@DSEnvel)
- kishan (@kishan)
- Bruno Alla (@browniebroke)
- nicain (@nicain)
- Carsten Rösnick-Neugebauer (@croesnick)
- igorbasko01 (@igorbasko01)
- Dan Booth Dev (@DanBoothDev)
- Pablo Panero (@ppanero)
- Chuan-Heng Hsiao (@chhsiao1981)
- Mohammad Hossein Sekhavat (@mhsekhavat)
- Amey Joshi (@amey589)
- Paul Harrison (@smoothml)
- Fabio Todaro (@SharpEdgeMarshall)
- Nicholas Bollweg (@bollwyvl)
- Jace Browning (@jacebrowning)
- Ionel Cristian Mărieș (@ionelmc)
- Kishan Mehta (@kishan3)
- Wieland Hoffmann (@mineo)
- Antony Lee (@anntzer)
- Aurélien Gâteau (@agateau)
- Axel H. (@noirbizarre)

- Chris (@chrisbrake)
- Chris Streeter (@streeter)
- Gábor Lipták (@gliptak)
- Javier Sánchez Portero (@javiersanp)
- Nimrod Milo (@milonimrod)
- Philipp Kats (@Casyfill)
- Reinout van Rees (@reinout)
- Rémy Greinhofer (@rgreinho)
- Sebastian (@sebix)
- Stuart Mumford (@Cadair)
- Tom Forbes (@orf)
- Xie Yanbo (@xyb)
- Maxim Ivanov (@ivanovmg)

3.2.4 Backers

We would like to thank the following people for supporting us in our efforts to maintain and improve Cookiecutter:

- Alex DeBrie
- Alexandre Y. Harano
- Bruno Alla
- Carol Willing
- Russell Keith-Magee

3.2.5 Sprint Contributors

PyCon 2016 Sprint

The following people made contributions to the cookiecutter project at the PyCon sprints in Portland, OR from June 2-5 2016. Contributions include user testing, debugging, improving documentation, reviewing issues, writing tutorials, creating and updating project templates, and teaching each other.

- Adam Chainz (@adamchainz)
- Andrew Ittner (@tephyr)
- Audrey M. Roy Greenfeld (@audreyfeldroy)
- Carol Willing (@willingc)
- Christopher Clarke (@chrisdev)
- Citlalli Murillo (@citmusa)
- Daniel Roy Greenfeld (@pydanny)
- Diane DeMers Chen (@purplediane)
- Elaine Wong (@elainewong)
- Elias Dorneles (@eliasdorneles)

- Emily Cain (@emcain)
- John Roa (@jhonjairoroa87)
- Jonan Scheffler (@1337807)
- Phoebe Bauer (@phoebebauer)
- Kartik Sundararajan (@skarbot)
- Katia Lira (@katialira)
- Leonardo Jimenez (@xpostudio4)
- Lindsay Slazakowski (@lslaz1)
- Meghan Heintz (@dot2dotseurat)
- Raphael Pierzina (@hackebrot)
- Umair Ashraf (@umrashrf)
- Valdir Stumm Junior (@stummjr)
- Vivian Guillen (@viviangb)
- Zaro (@zaro0508)

3.3 Case Studies

This showcase is where organizations can describe how they are using Cookiecutter.

3.3.1 BeeWare

Building Python tools for platforms like mobile phones and set top boxes requires a lot of boilerplate code just to get the project running. Cookiecutter has enabled us to very quickly stub out a starter project in which running Python code can be placed, and makes maintaining those templates very easy. With Cookiecutter we've been able to deliver support Android devices, iOS devices, tvOS boxes, and we're planning to add native support for iOS and Windows devices in the future.

BeeWare is an organization building open source libraries for Python support on all platforms.

3.3.2 ChrisDev

Anytime we start a new project we begin with a Cookiecutter template that generates a Django/Wagtail project Our developers like it for maintainability and our designers enjoy being able to spin up new sites using our tool chain very quickly. Cookiecutter is very useful for because it supports both Mac OSX and Windows users.

ChrisDev is a Trinidad-based consulting agency.

3.3.3 OpenStack

OpenStack uses several Cookiecutter templates to generate:

- Openstack compliant puppet-modules
- Install guides
- New tempest plugins

OpenStack is open source software for creating private and public clouds.

3.4 Code of Conduct

Everyone interacting in the Cookiecutter project's codebases and documentation is expected to follow the PyPA Code of Conduct. This includes, but is not limited to, issue trackers, chat rooms, mailing lists, and other virtual or in real life communication.

CHAPTER

FOUR

INDEX

- genindex
- modindex

PYTHON MODULE INDEX

C

INDEX

Symbols

A

C

D

E

F

G

I

J

L

M

N

O

P

R

S

T

U

V

W

Y