shelmet Documentation

Release 0.6.0

Derrick Gilland

Mar 29, 2021

CONTENTS

A shell power-up for working with the file system and running subprocess commands.

CHAPTER

ONE

LINKS

- Project: https://github.com/dgilland/shelmet
- Documentation: https://shelmet.readthedocs.io
- PyPI: https://pypi.python.org/pypi/shelmet/
- Github Actions: https://github.com/dgilland/shelmet/actions

CHAPTER

TWO

FEATURES

- Run and define subprocess commands
  - `run`
  - `cmd`
- Interact with files
  - `atomicdfile`, `atomicdir`
  - `read`, `readchunks`, `readlines`, `readtext`, `readbytes`
  - `write`, `writechunks`, `writelines`, `writetext`, `writebytes`
  - `fsync`, `dirsync`
- Execute core shell operations
  - `cp`, `mv`, `mkdir`, `touch`
  - `rm`, `rmfile`, `rmdir`
  - `ls`, `lsfiles`, `lsdirs`
  - `walk`, `walkfiles`, `walkdirs`
- Archive and backup files
  - `archive`, `unarchive`, `lsarchive`
  - `backup`
- Other utilities
  - `cd`
  - `environ`
  - `cwd`, `homedir`
  - and more!
- 100% test coverage
- Fully type-annotated
- Python 3.6+

CHAPTER

THREE

QUICKSTART

Install using pip:

```
pip3 install shelmet
```

Import the `sh` module:

```
import shelmet as sh
```

Run system commands:

```
# sh.run() is a wrapper around subprocess.run() that defaults to output capture, text-
↪mode,
# exception raising on non-zero exit codes, environment variable extension instead of
# replacement, and support for passing command arguments as a variable number of␣
↪strings
# instead of just a list of strings.
result = sh.run("ps", "aux")
print(result.stdout)
print(result.stderr)

# stdout and stderr can be combined with...
result = sh.run("some", "command", combine_output=True)

# or not captured at all...
sh.run("...", capture_output=False)
```

Create reusable run commands that support chained commands like "pipe" `|`, "and" `&&`, "or" `||`, and "after" `;`:

```
# sh.cmd() returns a sh.Command object that can be used to execute a fixed command.
ps_aux = sh.cmd("ps", "aux")

# And has the option to pipe it's output into another command automatically.
grep_ps = ps_aux.pipe("grep", "-i", check=False)
print(grep_ps.shell_cmd)
# ps aux | grep -i

search_result_1 = grep_ps.run("search term 1")
print(search_result_1.stdout)

search_result_2 = grep_ps.run("search term 2")
print(search_result_2.stdout)

# Equivalent to: mkdir foo && echo 'success' || echo 'failure'
sh.cmd("mkdir", "foo").and_("echo", "success").or_("echo", "failure").run()
```

Perform file system operations:

```
# Make directories and sub-directories. Behaves like "$ mkdir -p"
sh.mkdir("a", "b", "c", "d/e/f/g")

# Context-manager to change working directory temporarily. Behaves like "$ cd".
with sh.cd("d/e/f/g"):
    sh.touch("1.txt", "2.txt", "3.txt")

    # Move files or directories. Works across file-systems. Behaves like "$ mv".
    sh.mv("1.txt", "11.txt")

    # Copy files or directories. Behaves like "$ cp -r"
    sh.cp("2.txt", "22.txt")

    # List top-level directory contents.
    # NOTE: sh.ls() and its siblings return iterables.
    list(sh.ls())

    # Limit to files.
    list(sh.lsfiles())

    # Limit to directories.
    list(sh.lsdirs())

    # Remove files.
    sh.rmfile("11.txt", "22.txt", "3.txt")
    # Or use sh.rm which handles both files and directories.
    sh.rm("11.txt", "22.txt", "3.txt")

# Recursively walk current directory.
# NOTE: sh.walk() and its siblings return iterables.
list(sh.walk())

# Or just a specified directory.
list(sh.walk("d"))

# Or just it's files or directories.
list(sh.walkfiles())
list(sh.walkdirs())

# Remove directories.
sh.rmdir("a", "b", "c", "d")
# Or use sh.rm which handles both files and directories.
sh.rm("a", "b", "c", "d")
```

Perform file IO:

```
sh.write("test.txt", "some text\n")
sh.write("test.txt", " some more text\n", "a")

sh.write("test.bin", b"some bytes")
sh.write("test.bin", b" some more bytes", "ab")

sh.writelines("output.txt", ["1", "2", "3"])               # -> "1\n2\n3\n"
sh.writelines("output.txt", (str(i) for i in range(5)))   # -> "0\n1\n2\n3\n4\n"

# Write to a file atomically. See sh.atomicfile for more details.
```

(continues on next page)

(continued from previous page)

```
sh.write("test.txt", "content", atomic=True)
sh.writelines("test.txt", ["content"], atomic=True)

text = sh.read("test.txt")         # -> "some text\nsome more text\n"
data = sh.read("text.bin", "rb")   # -> b"some bytes some more bytes"

for line in sh.readlines("test.txt"):
    print(line)

for chunk in sh.readchunks("test.txt", size=1024):
    print(chunk)

sh.write("test.txt", "a|b|c|d")
items = list(sh.readchunks("test.txt", sep="|"))
print(items)  # -> ["a", "b", "c", "d"]

sh.write("test.txt", b"a|b|c|d", "wb")
assert "".join(sh.readchunks("test.txt", "rb", sep=b"|")) == b"a|b|c|d"
```

Backup files:

```
# Create backup as copy of file.
backup_file = sh.backup("a.txt")
print(backup_file)                                  # a.txt.2021-02-24T16:19:20.
↪276491~
sh.backup("a.txt", utc=True)                        # a.txt.2021-02-24T11:19:20.
↪276491Z~
sh.backup("a.txt", epoch=True)                      # a.txt.1614878783.56201
sh.backup("a.txt", suffix=".bak")                   # a.txt.2021-02-24T16:19:20.
↪276491.bak
sh.backup("a.txt", suffix=".bak", timestamp=False)  # a.txt.bak
sh.backup("a.txt", prefix="BACKUP_", suffix="")     # BACKUP_a.txt.2021-02-
↪24T16:19:20.276491

# Create backup as copy of directory.
sh.backup("path/to/dir")                            # path/to/dir.2021-02-24T16:19:20.
↪276491~

# Create backup as archive of file or directory.
sh.backup("b/c", ext=".tar.gz")                    # b/c.2021-02-24T16:19:20.276491.
↪tar.gz
sh.backup("b/c", ext=".tar.bz2")                   # b/c.2021-02-24T16:19:20.276491.
↪tar.bz2
sh.backup("b/c", ext=".tar.xz")                    # b/c.2021-02-24T16:19:20.276491.
↪tar.xz
sh.backup("b/c", ext=".zip")                       # b/c.2021-02-24T16:19:20.276491.
↪zip

from functools import partial
import itertools

counter = itertools.count(1)
backup = partial(sh.backup, namer=lambda src: f"{src.name}-{next(counter)}~")
backup("test.txt")  # test.txt-1~
backup("test.txt")  # test.txt-2~
backup("test.txt")  # test.txt-3~
```

Archive files:

```
# Create tar, tar-gz, tar-bz2, tar-xz, or zip archives.
sh.archive("archive.tar.gz", "/path/to/foo", "/path/to/bar")

# Archive type is inferred from extension in filename but can be explicitly set.
sh.archive("archive", "path", ext=".tbz")

# Files can be filtered with ls, lsfiles, lsdirs, walk, walkfiles, and walkdirs␣
↪functions.
sh.archive(
    "archive.tgz",
    sh.walk("path", include="*.py"),
     sh.walk("other/path", exclude="*.log"),
 )

# Archive paths can be customized with root and repath arguments.
# root changes the base path for archive members.
sh.archive("archive.txz", "/a/b/c/1", "/a/b/d/2", root="/a/b")
# -> archive members will be "c/1/*" and "d/2/*"
# -> without root, they would be "b/c/1/*" and "b/d/2/*"

# repath renames paths.
sh.archive("archive.zip", "/a/b/c", "/a/b/d", repath={"/a/b/c": "foo/bar"})
# -> archive members: "foo/bar/*" and "b/d/*"

# repath also works with ls* and walk* by matching on the base path.
sh.archive(
    "log-dump.taz",
    sh.walk("path/to/logs", include="*.log*"),
    repath={"path/to/logs": "logs"},
)
```

Get list of archive contents:

```
# Get list of archive contents as PurePath objects.
listing = sh.lsarchive("archive.tgz")

# Use an explicit extension when archive doesn't have one but is supported.
listing = sh.lsarchive("archive", ext=".tgz")
```

Unarchive tar and zip based archives:

```
# Extract tar, tar-gz, tar-bz2, tar-xz, or zip archives to directory.
sh.unarchive("archive.tgz", "out/dir")

# Potentially unsafe archives will raise an exception if the extraction path falls␣
↪outside
# the destination, e.g., when the archive contains absolute paths.
try:
    sh.unarchive("unsafe-archive.tz2", "out")
except sh.ArchiveError:
    pass

# But if an archive can be trusted...
sh.unarchive("unsafe-archive.tz2", "out")
```

Write to a new file atomically where content is written to a temporary file and then moved once finished:

```
import os

with sh.atomicfile("path/to/atomic.txt") as fp:
    # Writes are sent to a temporary file in the same directory as the destination.
    print(fp.name) # will be something like "path/to/.atomic.txt_XZKVqrlk.tmp"
    fp.write("some text")
    fp.write("some more text")

    # File doesn't exist yet.
    assert not os.path.exists("path/to/atomic.txt")

# Exiting context manager will result in the temporary file being atomically moved to
# destination. This will also result in a lower-level fsync on the destination file␣
↪and
# directory.
assert os.path.exists("path/to/atomic.txt")

# File mode, sync skipping, and overwrite flag can be specified to change the default
# behavior which is...
with sh.atomicfile("file.txt", "w", skip_sync=False, overwrite=True) as fp:
    pass

# Additional parameters to open() can be passed as keyword arguments.
with sh.atomicfile("file.txt", "w", **open_kwargs) as fp:
    pass

# To writie to a file atomically without a context manager
sh.write("file.txt", "content", atomic=True)
```

Create a new directory atomically where its contents are written to a temporary directory and then moved once finished:

```
with sh.atomicdir("path/to/atomic_dir") as atomic_dir:
    # Yielded path is temporary directory within the same parent directory as the␣
↪destination.
    # path will be something like "path/to/.atomic_dir_QGLDfPwz_tmp"
    some_file = atomic_dir / "file.txt"

    # file written to "path/to/.atomic_dir_QGLDfPwz_tmp/file.txt"
    some_file.write_text("contents")

    some_dir = atomic_dir / "dir"
    some_dir.mkdir()  # directory created at "path/to/.atomic_dir_QGLDfPwz_tmp/dir/"

    # Directory doesn't exist yet.
    assert not os.path.exists("path/to/atomic_dir")

# Exiting context manager will atomically move the the temporary directory to the␣
↪destination.
assert os.path.exists("path/to/atomic_dir")

# Sync skipping and overwrite flag can be specified to change the default behavior␣
↪which is...
with sh.atomicdir("atomic_dir", skip_sync=False, overwrite=True) as atomic_dir:
    pass
```

Temporarily change environment variables:

```
# Extend existing environment.
with sh.environ({"KEY1": "val1", "KEY2": "val2"}) as new_environ:
    # Do something while environment changed.
    # Environment variables include all previous ones and {"KEY1": "val1", "KEY2":
↪"val2"}.
    pass

# Replace the entire environment with a new one.
with sh.environ({"KEY": "val"}, replace=True):
    # Environment variables are replaced and are now just {"KEY": "val"}.
    pass
```

For more details, please see the full documentation at https://shelmet.readthedocs.io.

CHAPTER

FOUR

GUIDE

4.1 Installation

shelmet requires Python >= 3.6.

To install from PyPI:

```
pip install shelmet
```

4.2 API Reference

The shelmet package.

A shell power-up for working with the file system and running subprocess commands.

Exceptions:

ArchiveError	General archive error.
UnsafeArchiveError	Unsafe archive exception raised when an untrusted archive would extract contents outside of the destination directory.

Classes:

Command	A system command that can be executed multiple times and used to create piped commands.
Ls	Directory listing iterable that iterates over its contents and returns them as `Path` objects.

Functions:

archive	Create an archive from the given source paths.
atomicdir	Context-manager that is used to atomically create a directory and its contents.

continues on next page

Table 3 – continued from previous page

atomicfile	Context-manager similar to `open()` that is used to perform an atomic file write operation by first writing to a temporary location in the same directory as the destination and then renaming the file to the destination after all write operations are finished.
backup	Create a backup of a file or directory as either a direct copy or an archive file.
cd	Context manager that changes the working directory on enter and restores it on exit.
chmod	Change file or directory permissions using numeric or symbolic modes.
chown	Change ownership of file or directory to user and/or group.
cmd	Factory that returns an instance of *Command* initialized with the given arguments.
cp	Copy file or directory to destination.
cwd	Return current working directory as `Path` object.
dirsync	Force sync on directory.
environ	Context manager that updates environment variables with *env* on enter and restores the original environment on exit.
fsync	Force write of file to disk.
getdirsize	Return total size of directory's contents.
homedir	Return current user's home directory as `Path` object.
ls	Return iterable that lists directory contents as `Path` objects.
lsarchive	Return list of member paths contained in archive file.
lsdirs	Return iterable that only lists directories in directory as `Path` objects.
lsfiles	Return iterable that only lists files in directory as `Path` objects.
mkdir	Recursively create directories in *paths* along with any parent directories that don't already exists.
mv	Move source file or directory to destination.
read	Return contents of file.
readbytes	Return binary contents of file.
readchunks	Yield contents of file as chunks.
readlines	Yield each line of a file.
readtext	Return text contents of file.
reljoin	Like `os.path.join` except that all paths are treated as relative to the previous one so that an absolute path in the middle will extend the existing path instead of becoming the new root path.
rm	Delete files and directories.
rmdir	Delete directories.
rmfile	Delete files.
run	Convenience function-wrapper around *Command.run()*.
touch	Touch files.
umask	Context manager that sets the umask to *mask* and restores it on exit.

continues on next page

Table 3 – continued from previous page

unarchive	Extract an archive to the given destination path.
walk	Return iterable that recursively lists all directory contents as `Path` objects.
walkdirs	Return iterable that recursively lists only directories in directory as `Path` objects.
walkfiles	Return iterable that recursively lists only files in directory as `Path` objects.
write	Write contents to file.
writebytes	Write binary contents to file.
writelines	Write lines to file.
writetext	Write text contents to file.

exception shelmet.**ArchiveError**(**args*, *orig_exc=None*)

General archive error.

class shelmet.**Command**(**args*, *stdin=None*, *input=None*, *stdout=- 1*, *stderr=- 1*, *capture_output=True*, *combine_output=False*, *cwd=None*, *timeout=None*, *check=True*, *encoding=None*, *errors=None*, *text=True*, *env=None*, *replace_env=False*, *parent=None*, ***popen_kwargs*)

A system command that can be executed multiple times and used to create piped commands.

Executing the command is done using *run()* which is a wrapper around `subprocess.run`. However, the default arguments for a *Command* enable different default behavior than `subprocess.run`:

- Output is captured
- Text-mode is enabled
- Environment variables extend `os.environ` instead of replacing them.
- Exceptions are raised by default when the completed process returns a non-zero exit code.
- System command arguments can be passed as a var-args instead of just a list.

To disable output capture completely, use `capture_output=False`. To disable output capture for just one of them, set `stdout` or `stderr` to `None`.

To disable `os.environ` extension, use `replace_env=True`.

To disable exception raising, use `check=False`.

Therefore, to use the default behavior of `subprocess.run`, set the following keyword arguments:

```
ls = Command(["ls", "-la"], capture_output=False, text=False, check=False,␣
↪replace_env=True)
ls.run()
```

Parameters

- ***args** – System command arguments to execute. If `None` is given as an argument value, it will be discarded.
- **stdin** (`Union[int, IO[Any], None]`) – Specify the executed command's standard input.
- **input** (`Union[str, bytes, None]`) – If given it will be passed to the underlying process as stdin. When used, stdin will be set to `PIPE` automatically and cannot be used. The value will be encoded or decoded automatically if it does not match the expected type based on whether text-mode is enabled or not.

- **stdout** (Union[int, IO[Any], None]) – Specify the executed command's standard output.
- **stderr** (Union[int, IO[Any], None]) – Specify the executed command's standard error.
- **capture_output** (bool) – Whether to capture stdout and stderr and include in the returned completed process result.
- **combine_output** (bool) – Whether to combine stdout and stderr. Equilvalent to setting stderr=subprocess.STDOUT.
- **cwd** (Union[str, Path, None]) – Set the current working directory when executing the command.
- **timeout** (Union[float, int, None]) – If the timeout expires, the child process will be killed and waited for.
- **check** (bool) – Whether to check return code and raise if it is non-zero.
- **encoding** (Optional[str]) – Set encoding to use for text-mode.
- **errors** (Optional[str]) – Specify how encoding and decoding errors should be handled. Must be one of "strict", "ignore", "replace", "backslashreplace", "xmlcharrefreplace", "namereplace"
- **text** (bool) – Set text-mode.
- **env** (Optional[dict]) – Environment variables for the new process. Unlike in subprocess.run, the default behavior is to extend the existing environment. Use replace_env=True to replace the environment variables instead.
- **replace_env** (bool) – Whether to replace the current environment when *env* given.

Keyword Arguments

- **other keyword arguments are passed to subprocess.run which subsequently passes them** (*All*) –
- **subprocess.Popen.** (*to*) –

Methods:

after	Return a new command that will be executed after this command regardless of this command's return code.
*and_*	Return a new command that will be AND'd with this command.
*or_*	Return a new command that will be OR'd with this command.
pipe	Return a new command whose input will be piped from the output of this command.
run	Wrapper around subprocess.run that uses this class' arguments as defaults.

Attributes:

parents	Return list of parent *Command* objects that pipe output into this command.

continues on next page

Table 5 – continued from previous page

shell_cmd	Return string version of command that would be used when executing from a shell.

after(**args*, *stdin=None*, *input=None*, *stdout=- 1*, *stderr=- 1*, *capture_output=True*, *combine_output=False*, *cwd=None*, *timeout=None*, *check=True*, *encoding=None*, *errors=None*, *text=True*, *env=None*, *replace_env=False*, ***popen_kwargs*)
Return a new command that will be executed after this command regardless of this command's return code.

This is like running "<this-command> ; <next-command>".

Return type Command

and_(**args*, *stdin=None*, *input=None*, *stdout=- 1*, *stderr=- 1*, *capture_output=True*, *combine_output=False*, *cwd=None*, *timeout=None*, *check=True*, *encoding=None*, *errors=None*, *text=True*, *env=None*, *replace_env=False*, ***popen_kwargs*)
Return a new command that will be AND'd with this command.

This is like running "<this-command> && <next-command>".

Return type Command

or_(**args*, *stdin=None*, *input=None*, *stdout=- 1*, *stderr=- 1*, *capture_output=True*, *combine_output=False*, *cwd=None*, *timeout=None*, *check=True*, *encoding=None*, *errors=None*, *text=True*, *env=None*, *replace_env=False*, ***popen_kwargs*)
Return a new command that will be OR'd with this command.

This is like running "<this-command> || <next-command>".

Return type Command

property parents
Return list of parent *Command* objects that pipe output into this command.

Return type List[ChainCommand]

pipe(**args*, *stdin=None*, *input=None*, *stdout=- 1*, *stderr=- 1*, *capture_output=True*, *combine_output=False*, *cwd=None*, *timeout=None*, *check=True*, *encoding=None*, *errors=None*, *text=True*, *env=None*, *replace_env=False*, ***popen_kwargs*)
Return a new command whose input will be piped from the output of this command.

This is like running "<this-command> | <next-command>".

Return type Command

run(**extra_args*, ***override_kwargs*)
Wrapper around `subprocess.run` that uses this class' arguments as defaults.

To add additional command args to `args`, pass them as var-args.

To override default keyword arguments, pass them as keyword-args.

If `parent` is set (e.g. if this command was created with *pipe()*, *after()*, *and_()*, or *or_()*), then the parent command will be called first and then chained with this command.

Parameters

- ***extra_args** – Extend `args` with extra command arguments.
- ****override_kwargs** – Override this command's keyword arguments.

Return type CompletedProcess

property shell_cmd
Return string version of command that would be used when executing from a shell.

Return type str

class shelmet.**Ls**(*path='.'*, ***, *recursive=False*, *only_files=False*, *only_dirs=False*, *include=None*, *exclude=None*)

Directory listing iterable that iterates over its contents and returns them as Path objects.

Parameters

- **path** (Union[str, Path]) – Directory to list.
- **recursive** (bool) – Whether to recurse into subdirectories. Defaults to False.
- **only_files** (bool) – Limit results to files only. Mutually exclusive with only_dirs.
- **only_dirs** (bool) – Limit results to directories only. Mutually exclusive with only_files.
- **include** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Include paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is included if any of the filters return True and path matches only_files or only_dirs (if set). If path is a directory and is not included, its contents are still eligible for inclusion if they match one of the include filters.
- **exclude** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Exclude paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is not yielded if any of the filters return True. If the path is a directory and is excluded, then all of its contents will be excluded.

exception shelmet.**UnsafeArchiveError**(**args*, *orig_exc=None*)

Unsafe archive exception raised when an untrusted archive would extract contents outside of the destination directory.

shelmet.**archive**(*file*, **paths*, *root=None*, *repath=None*, *ext=''*)

Create an archive from the given source paths.

The source paths can be relative or absolute but the path names inside the archive will always be relative. By default, the paths within the archive will be determined by taking the common path of all the sources and removing it from each source path so that the archive paths are all relative to the shared parent path of all sources. If *root* is given, it will be used in place of the dynamic common path determination, but it must be a parent path common to all sources.

The archive member names of the source paths can be customized using the *repath* argument. The *repath* argument is a mapping of source paths to their custom archive name. If a source path is given as relative, then its repath key must also be relative. If a source path is given as absolute, then its repath key must also be absolute. The repath keys/values should be either strings or Path objects but they don't have to match the corresponding source path. Both the keys and values will have their path separators normalized.

Archives can be created in either the tar or zip format. A tar archive can use the same compressions that are available from tarfile which are gzipped, bzip2, and lzma. A zip archive will use deflate compression if the zlib library is available. Otherwise, it will fallback to being uncompressed.

The archive format is interfered from the file extension of *file* by default, but can be overridden using the *ext* argument (e.g. ext=".tgz" for a gzipped tarball).

The supported tar-based extensions are:

- .tar
- .tar.gz, .tgz, .taz
- .tar.bz2, .tb2, .tbz, .tbz2, .tz2

- `.tar.xz`, `.txz`

The supported zip-based extensions are:

- `.zip`,
- `.egg`, `.jar`
- `.docx`, `pptx`, `xlsx`
- `.odg`, `.odp`, `.ods`, `.odt`

Parameters

- **file** (`Union[str, Path]`) – Archive file path to create.
- ***paths** – Source paths (files and/or directories) to archive. Directories will be recursively added.
- **root** (`Union[str, Path, None]`) – Archive member paths will be relative to this root directory. The root path must be a parent directory of all source paths, otherwise, an exception will be raised.
- **repath** (`Union[str, Mapping[Union[str, Path], Union[str, Path]], None]`) – A mapping of source paths to archive names that will rename the source path to the mapped value within the archive. A string representing the archive member name can only be used when a single source path is being added to the archive.
- **ext** (`str`) – Specify the archive format to use by referencing the corresponding file extension (starting with a leading ".") instead of interfering the format from the *file* extension.

Return type `None`

shelmet.**atomicdir**(*dir*, *, *skip_sync=False*, *overwrite=True*)

Context-manager that is used to atomically create a directory and its contents.

This context-manager will create a temporary directory in the same directory as the destination and yield the temporary directory as a `pathblib.Path` object. All atomic file system updates to the directory should then be done within the context-manager. Once the context-manager exits, the temporary directory will be passed to *dirsync()* (unless `skip_sync=True`) and then moved to the destination followed by *dirsync()* on the parent directory. If the destination directory exists, it will be overwritten unless `overwrite=False`.

Parameters

- **dir** (`Union[str, Path]`) – Directory path to create.
- **skip_sync** (`bool`) – Whether to skip calling *dirsync()* on the directory. Skipping this can help with performance at the cost of durability.
- **overwrite** (`bool`) – Whether to raise an exception if the destination exists once the directory is to be moved to its destination.

Return type `Iterator[Path]`

shelmet.**atomicfile**(*file*, *mode='w'*, *, *skip_sync=False*, *overwrite=True*, ***open_kwargs*)

Context-manager similar to `open()` that is used to perform an atomic file write operation by first writing to a temporary location in the same directory as the destination and then renaming the file to the destination after all write operations are finished.

This context-manager will open a temporary file for writing in the same directory as the destination and yield a file object just like `open()` does. All file operations while the context-manager is opened will be performed on the temporary file. Once the context-manager exits, the temporary file will flushed and fsync'd (unless `skip_sync=True`). If the destination file exists, it will be overwritten unless `overwrite=False`.

Parameters

- **file** (Union[str, Path]) – File path to write to.
- **mode** (str) – File open mode.
- **skip_sync** (bool) – Whether to skip calling fsync on file. Skipping this can help with performance at the cost of durability.
- **overwrite** (bool) – Whether to raise an exception if the destination file exists once the file is to be written to its destination.
- ****open_kwargs** – Additional keyword arguments to open() when creating the temporary write file.

Return type Iterator[IO]

shelmet.**backup**(*src*, ***, *timestamp='%Y-%m-%dT%H:%M:%S.%f%z'*, *utc=False*, *epoch=False*, *prefix=''*, *suffix='~'*, *ext=None*, *hidden=False*, *overwrite=False*, *dir=None*, *namer=None*)

Create a backup of a file or directory as either a direct copy or an archive file.

The format of the backup name is {prefix}{src}.{timestamp}{suffix|ext}.

By default, the backup will be created in the same parent directory as the source and be named like "src.YYYY-MM-DDThh:mm:ss.ffffff~", where the timestamp is the current local time.

If *utc* is True, then the timestamp will be in the UTC timezone.

If *epoch* is True, then the timestamp will be the Unix time as returned by time.time() instead of the strftime format.

If *ext* is given, the backup created will be an archive file. The extension must be one that *archive()* supports. The *suffix* value will be ignored and *ext* used in its place.

If *hidden* is True, then a "." will be prepended to the *prefix*. It won't be added if *prefix* already starts with a ".".

If *dir* is given, it will be used as the parent directory of the backup instead of the source's parent directory.

If *overwrite* is True and the backup location already exists, then it will be overwritten.

If *namer* is given, it will be called with namer(src) and it should return the full destination path of the backup. All other arguments to this function will be ignored except for *overwrite*.

Parameters

- **src** (Union[str, Path]) – Source file or directory to backup.
- **timestamp** (Optional[str]) – Timestamp strftime-format string or None to exclude timestamp from backup name. Defaults to ISO-8601 format.
- **utc** (bool) – Whether to use UTC time instead of local time for the timestamp.
- **epoch** (bool) – Whether to use the Unix time for the timestamp instead of the strftime format in *timestamp*.
- **prefix** (str) – Name prefix to prepend to the backup.
- **suffix** (str) – Name suffix to append to the backup.
- **ext** (Optional[str]) – Create an archive of *src* as the backup instead of a direct copy using the given archive extension. The extension must be supported by *archive()* or an exception will be raised. When given the *suffix* value is ignored and *ext* will be used in its place.
- **hidden** (bool) – Whether to ensure that the backup location is a hidden file or directory.

- **overwrite** (bool) – Whether to overwrite an existing file or directory when backing up.
- **dir** (Union[str, Path, None]) – Set the parent directory of the backup. Defaults to None which will use the parent directory of the *src*.
- **namer** (Optional[Callable[[Path], Union[str, Path]]]) – Naming function that can be used to return the full path of the backup location. It will be passed the *src* value as a pathlib.Path object as a positional argument. It should return the destination path of the backup as a str or pathlib.Path.

Return type Path

Returns Backup location.

shelmet.**cd**(*path*)

Context manager that changes the working directory on enter and restores it on exit.

Parameters **path** (Union[str, Path]) – Directory to change to.

Return type Iterator[None]

shelmet.**chmod**(*path*, *mode*, *, *follow_symlinks=True*, *recursive=False*)

Change file or directory permissions using numeric or symbolic modes.

The mode can either be an integer, an octal number (e.g. 0o600), an octal string (e.g. "600"), or a symbolic permissions string (e.g. "u+rw,g=r,o-rwx").

The symbolic permissions string format is similar to what is accepted by the UNIX command chmod:

- Symbolic format: [ugoa...][-+=][rwxstugo...][,...]
- [ugoa...]: Optional zero or more characters that set the user class parameter.
  - u: user
  - g: group
  - o: other
  - a: all
  - Defaults to a when none given
- [-+=]: Required operation that modifies the permissions.
  - -: removes the given permissions
  - +: adds the given permissions
  - =: sets the given permissions to what was specified
  - If = is used without permissions, then the user class will have all of its permissions removed
- [rwxstugo...]: Permissions to modify for the given user classes.
  - r: Read
  - w: Write
  - x: Execute
  - s: User or Group ID bit
  - t: Sticky bit
  - u: User permission bits of the original path mode
  - g: Group permission bits of the original path mode

  - `o`: Other permission bits of the original path mode

- Multiple permission clauses are separated with `,`.

Examples:

```
# Set permissions to 600 using octal number.
chmod(path, 0o600)

# Set permissions to 600 using octal string.
chmod(path, "600")

# Set user to read-write, group to read, and remove read-write-execute from other
chmod(path, "u=rw,g=r,o-rwx")

# Set user, group, and other to read-write
chmod(path, "a=rw")

# Add execute permission for user, group, and other
chmod(path, "+x")

# Add user id bit, group id bit, and set sticky bit
chmod(path, "u+s,g+s,+t")

# Set group permission to same as user
chmod(path, "g=u")
```

Parameters

- **path** (`Union[str, Path, int]`) – File, directory, or file-descriptor.
- **mode** (`Union[str, int]`) – Permission mode to set.
- **follow_symlinks** (`bool`) – Whether to follow symlinks.
- **recursive** (`bool`) – Whether to recursively apply permissions to subdirectories and their files.

Return type `None`

`shelmet.`**`chown`**(*path*, *user=None*, *group=None*, ***, *follow_symlinks=True*, *recursive=False*)

Change ownership of file or directory to user and/or group.

User and group can be a string name or a numeric id. Leave as `None` to not change the respective user or group ownership.

Parameters

- **path** (`Union[str, Path, int]`) – File, directory, or file-descriptor.
- **user** (`Union[str, int, None]`) – User name or uid to set as owner. Use `None` or `-1` to not change.
- **group** (`Union[str, int, None]`) – Group name or gid to set as owner. Use `None` or `-1` to not change.
- **follow_symlinks** (`bool`) – Whether to follow symlinks.
- **recursive** (`bool`) – Whether to recursively apply ownership to subdirectories and their files.

Return type `None`

shelmet.**cmd**(**args*, *stdin=None*, *input=None*, *stdout=- 1*, *stderr=- 1*, *capture_output=True*, *combine_output=False*, *cwd=None*, *timeout=None*, *check=True*, *encoding=None*, *errors=None*, *text=True*, *env=None*, *replace_env=False*, ***popen_kwargs*)

Factory that returns an instance of *Command* initialized with the given arguments.

See also:

Command for description of arguments.

Return type Command

shelmet.**cp**(*src*, *dst*, ***, *follow_symlinks=True*)

Copy file or directory to destination.

Files are copied atomically by first copying to a temporary file in the same target directory and then renaming the temporary file to its actual filename.

Parameters

- **src** (Union[str, Path]) – Source file or directory to copy from.
- **dst** (Union[str, Path]) – Destination file or directory to copy to.
- **follow_symlinks** (bool) – When true (the default), symlinks in the source will be dereferenced into the destination. When false, symlinks in the source will be preserved as symlinks in the destination.

Return type None

shelmet.**cwd**()

Return current working directory as Path object.

Return type Path

shelmet.**dirsync**(*path*)

Force sync on directory.

Parameters **path** (Union[str, Path]) – Directory to sync.

Return type None

shelmet.**environ**(*env=None*, ***, *replace=False*)

Context manager that updates environment variables with *env* on enter and restores the original environment on exit.

Parameters

- **env** (Optional[Dict[str, str]]) – Environment variables to set.
- **replace** (bool) – Whether to clear existing environment variables before setting new ones. This fully replaces the existing environment variables so that only *env* are set.

Yields The current environment variables.

Return type Iterator[Dict[str, str]]

shelmet.**fsync**(*fd*)

Force write of file to disk.

The file descriptor will have os.fsync() (or fcntl.fcntl() with fcntl.F_FULLFSYNC if available) called on it. If a file object is passed it, then it will first be flushed before synced.

Parameters **fd** (Union[IO, int]) – Either file descriptor integer or file object.

Return type None

shelmet.**getdirsize**(*path*, *pattern='**/*'*)
Return total size of directory's contents.

Parameters

- **path** (Union[str, Path]) – Directory to calculate total size of.
- **pattern** (str) – Only count files if they match this glob-pattern.

Return type int

Returns Total size of directory in bytes.

shelmet.**homedir**()
Return current user's home directory as Path object.

shelmet.**ls**(*path='.'*, ***, *recursive=False*, *only_files=False*, *only_dirs=False*, *include=None*, *exclude=None*)
Return iterable that lists directory contents as Path objects.

Parameters

- **path** (Union[str, Path]) – Directory to list.
- **recursive** (bool) – Whether to recurse into subdirectories. Defaults to False.
- **only_files** (bool) – Limit results to files only. Mutually exclusive with only_dirs.
- **only_dirs** (bool) – Limit results to directories only. Mutually exclusive with only_files.
- **include** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Include paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is included if any of the filters return True and path matches only_files or only_dirs (if set). If path is a directory and is not included, its contents are still eligible for inclusion if they match one of the include filters.
- **exclude** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Exclude paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is not yielded if any of the filters return True. If the path is a directory and is excluded, then all of its contents will be excluded.

Return type Ls

shelmet.**lsarchive**(*file*, *ext=''*)
Return list of member paths contained in archive file.

Parameters

- **file** (Union[str, Path]) – Archive file to list.
- **ext** (str) – Specify the archive format to use by referencing the corresponding file extension (starting with a leading ".") instead of interfering the format from the *file* extension.

Return type List[PurePath]

shelmet.**lsdirs**(*path='.'*, ***, *include=None*, *exclude=None*)
Return iterable that only lists directories in directory as Path objects.

See also:

This function is not recursive and will only yield the top-level contents of a directory. Use *walkdirs()* to recursively yield all directories from a directory.

Parameters

- **path** (Union[str, Path]) – Directory to list.
- **include** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Include paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is included if any of the filters return True. If path is a directory and is not included, its contents are still eligible for inclusion if they match one of the include filters.
- **exclude** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Exclude paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is not yielded if any of the filters return True. If the path is a directory and is excluded, then all of its contents will be excluded.

Return type Ls

shelmet.**lsfiles**(*path='.'*, ***, *include=None*, *exclude=None*)

Return iterable that only lists files in directory as Path objects.

See also:

This function is not recursive and will only yield the top-level contents of a directory. Use *walkfiles()* to recursively yield all files from a directory.

Parameters

- **path** (Union[str, Path]) – Directory to list.
- **include** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Include paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is included if any of the filters return True. If path is a directory and is not included, its contents are still eligible for inclusion if they match one of the include filters.
- **exclude** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Exclude paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is not yielded if any of the filters return True. If the path is a directory and is excluded, then all of its contents will be excluded.

Return type Ls

shelmet.**mkdir**(**paths*, *mode=511*, *exist_ok=True*)

Recursively create directories in *paths* along with any parent directories that don't already exists.

This is like the Unix command `mkdir -p <path1> <path2> ...`.

Parameters

- ***paths** – Directories to create.
- **mode** (int) – Access mode for directories.
- **exist_ok** (bool) – Whether it's ok or not if the path already exists. When True, a FileExistsError will be raised.

Return type None

shelmet.**mv**(*src*, *dst*)
Move source file or directory to destination.

The move semantics are as follows:

- If src and dst are files, then src will be renamed to dst and overwrite dst if it exists.
- If src is a file and dst is a directory, then src will be moved under dst.
- If src is a directory and dst does not exist, then src will be renamed to dst and any parent directories that don't exist in the dst path will be created.
- If src is a directory and dst is a directory and the src's basename does not exist under dst or if it is an empty directory, then src will be moved under dst.
- If src is directory and dst is a directory and the src's basename is a non-empty directory under dst, then an `OSError` will be raised.
- If src and dst reference two difference file-systems, then src will be copied to dst using *cp()* and then deleted at src.

Parameters

- **src** (Union[str, Path]) – Source file or directory to move.
- **dst** (Union[str, Path]) – Destination file or directory to move source to.

Return type None

shelmet.**read**(*file*, *mode='r'*, ***open_kwargs*)
Return contents of file.

Parameters

- **file** (Union[str, Path]) – File to read.
- **mode** (str) – File open mode.
- ****open_kwargs** – Additional keyword arguments to pass to `open`.

Return type Union[str, bytes]

shelmet.**readbytes**(*file*, ***open_kwargs*)
Return binary contents of file.

Equivalent to calling *read()* with `mode="rb"`.

Parameters

- **file** (Union[str, Path]) – File to read.
- ****open_kwargs** – Additional keyword arguments to pass to `open`.

Return type bytes

shelmet.**readchunks**(*file*, *mode='r'*, ***, *size=8192*, *sep=None*, ***open_kwargs*)
Yield contents of file as chunks.

If separator, *sep*, is not given, chunks will be yielded by *size*.

If separator, *sep*, is given, chunks will be yielded from as if from `contents.split(sep)`. The *size* argument will still be used for each file read operation, but the contents will be buffered until a separator is encountered.

Parameters

- **file** (Union[str, Path]) – File to read.

- **mode** (str) – File open mode.
- **size** (int) – Size of chunks to read from file at a time and chunk size to yield when *sep* not given.
- **sep** (Union[str, bytes, None]) – Separator to split chunks by in lieu of splitting by size.
- ****open_kwargs** – Additional keyword arguments to pass to open.

Return type Generator[Union[str, bytes], None, None]

shelmet.**readlines**(*file*, *mode='r'*, ***, *limit=- 1*, ***open_kwargs*)

Yield each line of a file.

Note: Line-endings are included in the yielded values.

Parameters

- **file** (Union[str, Path]) – File to read.
- **mode** (str) – File open mode.
- **limit** (int) – Maximum length of each line to yield. For example, limit=10 will yield the first 10 characters of each line.
- ****open_kwargs** – Additional keyword arguments to pass to open.

Return type Generator[Union[str, bytes], None, None]

shelmet.**readtext**(*file*, ***open_kwargs*)

Return text contents of file.

Equivalent to calling *read()* with mode="r" (the default behavior of *read()*).

Parameters

- **file** (Union[str, Path]) – File to read.
- ****open_kwargs** – Additional keyword arguments to pass to open.

Return type str

shelmet.**reljoin**(**paths*)

Like os.path.join except that all paths are treated as relative to the previous one so that an absolute path in the middle will extend the existing path instead of becoming the new root path.

Parameters ***paths** – Paths to join together.

Return type str

shelmet.**rm**(**paths*)

Delete files and directories.

Note: Deleting non-existent files or directories does not raise an error.

Warning: This function is like $ rm -rf so be careful. To limit the scope of the removal to just files or just directories, use *rmfile()* or *rmdir()* respectively.

Parameters ***paths** – Files and/or directories to delete.

Return type None

shelmet.**rmdir**(**dirs*)
Delete directories.

Note: Deleting non-existent directories does not raise an error.

Warning: This function is like calling `$ rm -rf` on a directory. To limit the scope of the removal to just files, use *rmfile()*.

Parameters ***dirs** – Directories to delete.

Raises **NotADirectoryError** – When given path is not a directory.

Return type None

shelmet.**rmfile**(**files*)
Delete files.

Note: Deleting non-existent files does not raise an error.

Parameters ***files** – Files to delete.

Raises **IsADirectoryError** – When given path is a directory.

Return type None

shelmet.**run**(**args*, *stdin=None*, *input=None*, *stdout=- 1*, *stderr=- 1*, *capture_output=True*, *combine_output=False*, *cwd=None*, *timeout=None*, *check=True*, *encoding=None*, *errors=None*, *text=True*, *env=None*, *replace_env=False*, ***popen_kwargs*)
Convenience function-wrapper around *Command.run()*.

Using this function is equivalent to:

```
result = sh.cmd(*args, **kwargs).run()
```

See also:

Command for description of arguments.

Return type CompletedProcess

shelmet.**touch**(**paths*)
Touch files.

Parameters ***paths** – File paths to create.

Return type None

shelmet.**umask**(*mask=0*)
Context manager that sets the umask to *mask* and restores it on exit.

Parameters **mask** (int) – Numeric umask to set.

Yields None

Return type Iterator[None]

shelmet.**unarchive**(*file*, *dst='.'*, ***, *ext=''*, *trusted=False*)

Extract an archive to the given destination path.

If the archive contains any paths that would be extracted outside the destination path, an *ArchiveError* will be raised to prevent untrusted archives from extracting contents to locations that may pose a security risk. To allow a trusted archive to extract contents outside the destination, use the argument `trusted=True`.

Archives can be extracted from either zip or tar formats with compression. The tar compressions available are the same as what is supported by `tarfile` which are gzipped, bzip2, and lzma.

The archive format is interfered from the file extension of *file* by default, but can be overridden using the *ext* argument (e.g. `ext=".tgz"` for a gzipped tarball).

The supported tar extensions are:

- `.tar`
- `.tar.gz`, `.tgz`, `.taz`
- `.tar.bz2`, `.tb2`, `.tbz`, `.tbz2`, `.tz2`
- `.tar.xz`, `.txz`
- `.zip`,
- `.egg`, `.jar`
- `.docx`, `pptx`, `xlsx`
- `.odg`, `.odp`, `.ods`, `.odt`

Parameters

- **file** (Union[str, Path]) – Archive file to unarchive.
- **dst** (Union[str, Path]) – Destination directory to unarchive contents to.
- **ext** (str) – Specify the archive format to use by referencing the corresponding file extension (starting with a leading ".") instead of interfering the format from the *file* extension.
- **trusted** (bool) – Whether the archive is safe and can be trusted to allow it to extract contents outside of the destination path. Only enable this for archives that have been verified as originating from a trusted source.

Return type None

shelmet.**walk**(*path='.'*, ***, *only_files=False*, *only_dirs=False*, *include=None*, *exclude=None*)

Return iterable that recursively lists all directory contents as `Path` objects.

See also:

This function is recursive and will list all contents of a directory. Use *ls()* to list only the top-level contents of a directory.

Parameters

- **path** (Union[str, Path]) – Directory to walk.
- **only_files** (bool) – Limit results to files only. Mutually exclusive with `only_dirs`.
- **only_dirs** (bool) – Limit results to directories only. Mutually exclusive with `only_files`.

- **include** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Include paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is included if any of the filters return `True` and path matches `only_files` or `only_dirs` (if set). If path is a directory and is not included, its contents are still eligible for inclusion if they match one of the include filters.
- **exclude** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Exclude paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is not yielded if any of the filters return `True`. If the path is a directory and is excluded, then all of its contents will be excluded.

Return type `Ls`

shelmet.**walkdirs**(*path='.'*, ***, *include=None*, *exclude=None*)

Return iterable that recursively lists only directories in directory as `Path` objects.

See also:

This function is recursive and will list all directories in a directory. Use *`lsfiles()`* to list only the top-level directories in a directory.

Parameters

- **path** (Union[str, Path]) – Directory to walk.
- **include** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Include paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is included if any of the filters return `True`. If path is a directory and is not included, its contents are still eligible for inclusion if they match one of the include filters.
- **exclude** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Exclude paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is not yielded if any of the filters return `True`. If the path is a directory and is excluded, then all of its contents will be excluded.

Return type `Ls`

shelmet.**walkfiles**(*path='.'*, ***, *include=None*, *exclude=None*)

Return iterable that recursively lists only files in directory as `Path` objects.

See also:

This function is recursive and will list all files in a directory. Use *`lsfiles()`* to list only the top-level files in a directory.

Parameters

- **path** (Union[str, Path]) – Directory to walk.
- **include** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Include paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is included if any of the filters return `True`. If path is a directory and is not included, its contents are still eligible for inclusion if they match one of the include filters.

- **exclude** (Union[str, Pattern, Callable[[Path], bool], Iterable[Union[str, Pattern, Callable[[Path], bool]]], None]) – Exclude paths by filtering on a glob-pattern string, compiled regex, callable, or iterable containing any of those types. Path is not yielded if any of the filters return True. If the path is a directory and is excluded, then all of its contents will be excluded.

Return type Ls

shelmet.**write**(*file*, *contents*, *mode='w'*, ***, *atomic=False*, ***open_kwargs*)
Write contents to file.

Parameters

- **file** (Union[str, Path]) – File to write.
- **contents** (Union[str, bytes]) – Contents to write.
- **mode** (str) – File open mode.
- **atomic** (bool) – Whether to write the file to a temporary location in the same directory before moving it to the destination.
- ****open_kwargs** – Additional keyword arguments to pass to open.

Return type None

shelmet.**writebytes**(*file*, *contents*, *mode='wb'*, ***, *atomic=False*, ***open_kwargs*)
Write binary contents to file.

Parameters

- **file** (Union[str, Path]) – File to write.
- **contents** (bytes) – Contents to write.
- **mode** (str) – File open mode.
- **atomic** (bool) – Whether to write the file to a temporary location in the same directory before moving it to the destination.
- ****open_kwargs** – Additional keyword arguments to pass to open.

Return type None

shelmet.**writelines**(*file*, *items*, *mode='w'*, ***, *ending=None*, *atomic=False*, ***open_kwargs*)
Write lines to file.

Parameters

- **file** (Union[str, Path]) – File to write.
- **items** (Union[Iterable[str], Iterable[bytes]]) – Items to write.
- **mode** (str) – File open mode.
- **ending** (Union[str, bytes, None]) – Line ending to use. Defaults to newline.
- **atomic** (bool) – Whether to write the file to a temporary location in the same directory before moving it to the destination.
- ****open_kwargs** – Additional keyword arguments to pass to open.

Return type None

shelmet.**writetext**(*file*, *contents*, *mode='w'*, ***, *atomic=False*, ***open_kwargs*)
Write text contents to file.

Parameters

- **file** (Union[str, Path]) – File to write.
- **contents** (str) – Contents to write.
- **mode** (str) – File open mode.
- **atomic** (bool) – Whether to write the file to a temporary location in the same directory before moving it to the destination.
- ****open_kwargs** – Additional keyword arguments to pass to open.

Return type None

4.3 Developer Guide

This guide provides an overview of the tooling this project uses and how to execute developer workflows using the developer CLI.

4.3.1 Python Environments

This Python project is tested against different Python versions. For local development, it is a good idea to have those versions installed so that tests can be run against each.

There are libraries that can help with this. Which tools to use is largely a matter of preference, but below are a few recommendations.

For managing multiple Python versions:

- pyenv
- OS package manager (e.g. apt, yum, homebrew, etc)
- Build from source

For managing Python virtualenvs:

- pyenv-virtualenv
- pew
- python-venv

4.3.2 Tooling

The following tools are used by this project:

Tool	Description	Configuration
black	Code formatter	pyproject.toml
isort	Import statement formatter	setup.cfg
docformatter	Docstring formatter	setup.cfg
flake8	Code linter	setup.cfg
pylint	Code linter	pylintrc
pytest	Test framework	setup.cfg
tox	Test environment manager	tox.ini
invoke	CLI task execution library	tasks.py

4.3.3 Workflows

The following workflows use developer CLI commands via invoke and are defined in `tasks.py`.

Autoformat Code

To run all autoformatters:

```
inv fmt
```

This is the same as running each autoformatter individually:

```
inv black
inv isort
inv docformatter
```

Lint

To run all linters:

```
inv lint
```

This is the same as running each linter individually:

```
inv flake8
inv pylint
```

Test

To run all unit tests:

```
inv unit
```

To run unit tests and builds:

```
inv test
```

Test on All Supported Python Versions

To run tests on all supported Python versions:

```
tox
```

This requires that the supported versions are available on the PATH.

Build Package

To build the package:

```
inv build
```

This will output the source and binary distributions under `dist/`.

Build Docs

To build documentation:

```
inv docs
```

This will output the documentation under `docs/_build/`.

Serve Docs

To serve docs over HTTP:

```
inv docs -s|--server [-b|--bind 127.0.0.1] [-p|--port 8000]

inv docs -s
inv docs -s -p 8080
inv docs -s -b 0.0.0.0 -p 8080
```

Delete Build Files

To remove all build and temporary files:

```
inv clean
```

This will remove Python bytecode files, egg files, build output folders, caches, and tox folders.

Release Package

To release a new version of the package to https://pypi.org:

```
inv release
```

4.3.4 CI/CD

This project uses Github Actions for CI/CD:

- https://github.com/dgilland/shelmet/actions

CHAPTER

FIVE

PROJECT INFO

5.1 License

5.2 Versioning

This project follows Semantic Versioning.

5.3 Changelog

5.3.1 v0.6.0 (2021-03-29)

- Change return type for `ls`, `lsfiles`, `lsdirs`, `walk`, `walkfiles`, and `walkdirs` to an iterable class, `Ls`. Previously, these functions were generators.
- Add option to backup to an archive file in `backup`.
- Add functions:
  - `archive`
  - `chmod`
  - `chown`

  - `lsarchive`
  - `unarchive`

5.3.2 v0.5.0 (2021-03-04)

- Import all utility functions into `shelmet` namespace.
- Remove `shelmet.sh` catch-all submodule in favor of splitting it into smaller submodules, `shelmet.filesystem` and `shelmet.path`. Recommend using `import shelmet as sh` as primary usage pattern instead of importing submodules. **breaking change**
- Add functions:
  - `backup`
  - `read`
  - `readbytes`
  - `readchunks`
  - `readlines`
  - `readtext`
  - `write`
  - `writebytes`
  - `writelines`
  - `writetext`

5.3.3 v0.4.0 (2021-01-26)

- Rename `sh.command` to `sh.cmd`. **breaking change**
- Add methods to `sh.Command` / `sh.command`:
  - `Command.and_`
  - `Command.or_`
  - `Command.after`

5.3.4 v0.3.0 (2020-12-24)

- Add to `sh` module:
  - `Command`
  - `command`
  - `cwd`
  - `homedir`
  - `run`

5.3.5 v0.2.0 (2020-11-30)

- Add to `sh` module:
  - `atomicdir`
- Rename `atomic_write` to `atomicfile`. **breaking change**

5.3.6 v0.1.0 (2020-11-16)

- First release.
- Add `sh` module:
  - `atomic_write`
  - `cd`
  - `cp`
  - `dirsync`
  - `environ`
  - `fsync`
  - `getdirsize`
  - `ls`
  - `lsdirs`
  - `lsfiles`
  - `mkdir`
  - `mv`
  - `reljoin`
  - `rm`
  - `rmdir`
  - `rmfile`
  - `touch`
  - `umask`
  - `walk`
  - `walkdirs`
  - `walkfiles`

5.4 Authors

5.4.1 Lead

- Derrick Gilland, dgilland@gmail.com, dgilland@github

5.4.2 Contributors

None

5.5 Contributing

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

You can contribute in many ways:

5.5.1 Types of Contributions

Report Bugs

Report bugs at https://github.com/dgilland/shelmet.

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.
- Detailed steps to reproduce the bug.

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" or "help wanted" is open to whoever wants to implement it.

Write Documentation

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

Submit Feedback

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

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 :)

5.5.2 Get Started!

Ready to contribute? Here's how to set up `shelmet` for local development.

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

```
$ git clone git@github.com:your_username_here/shelmet.git
```

3. Install Python dependencies into a virtualenv:

```
$ cd shelmet
$ pip install -r requirements.txt
```

4. Create a branch for local development:

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

   Now you can make your changes locally.

5. Autoformat code:

```
$ inv fmt
```

6. When you're done making changes, check that your changes pass all unit tests by testing with `tox` across all supported Python versions:

```
$ tox
```

7. Add yourself to `AUTHORS.rst`.
8. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "<Detailed description of your changes>"
$ git push origin name-of-your-bugfix-or-feature-branch
```

9. Submit a pull request through GitHub.

5.5.3 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 work for all versions Python that this project supports.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX