pydistcheck

James Lamb

Feb 18, 2024

CONTENTS:

`pydistcheck` is a command-line interface (CLI) used to perform the following activities on Python package distributions.

inspect the distribution's contents during development

- *how large is the package, compressed and uncompressed?*
- *how many files does it contain?*
- *what % of the package is Python files? compiled objects?*
- *what's the difference between the compressed and uncompressed size?*

enforce constraints in continuous integration

- *should not be larger than* `n` *MB uncompressed*
- *should not contain more than* `x` *files*
- *should not contain non-portable filepaths*
- *should not contain compiled code with debugging symbols*

```
# install
pipx install pydistcheck

# run checks
pydistcheck dist/*

# run checks and view diagnostic information
pydistcheck --inspect dist/*
```

CHAPTER

ONE

INSTALLATION

`pydistcheck` is a command-line interface (CLI) written in Python.

Because of this, the preferred way to install it from PyPI is with `pipx` (docs).

```
pipx install pydistcheck
```

Checking `.conda`-format `conda` packages requires some additional dependencies. To install those, run the following.

```
pipx install 'pydistcheck[conda]'
```

If that doesn't work for you, see the sections below for other options.

1.1 PyPI

If you do not want to use `pipx` but want to install from PyPI, install with `pip`.

```
pip install pydistcheck
```

Checking `.conda`-format `conda` packages requires some additional dependencies. To install those, run the following.

```
pip install 'pydistcheck[conda]'
```

1.2 conda-forge

If you use tools like `conda` or `mamba` to manage environments, install `pydistcheck` from conda-forge.

```
conda install -c conda-forge pydistcheck
```

or

```
mamba install pydistcheck
```

1.3 development version

To install the latest development (not released) version of `pydistcheck`, clone the repo.

```
git clone https://github.com/jameslamb/pydistcheck.git
cd pydistcheck
pipx install '.[conda]'
```

CHAPTER

TWO

CONFIGURATION

This page describes how to configure `pydistcheck`.

`pydistscheck` resolves different sources of configuration in the following order.

1. default values
2. *pyproject.toml*
3. *CLI arguments*

2.1 CLI arguments

2.1.1 pydistcheck

Run the contents of a distribution through a set of checks, and warn about any problematic characteristics that are detected.

Exit codes:

0 = no issues detected

1 = issues detected

```
pydistcheck [OPTIONS] [FILEPATHS]...
```

Options

--version

Print the version of pydistcheck and exit.

--config <config>

Path to a TOML file containing a [tool.pydistcheck] table. If provided, pyproject.toml will be ignored.

--ignore <ignore>

comma-separated list of checks to skip, e.g. `distro-too-large-compressed,path-contains-spaces`.

--inspect

print diagnostic information about the distribution

--max-allowed-files <max_allowed_files>

maximum number of files allowed in the distribution

Default

2000

--max-allowed-size-compressed <max_allowed_size_compressed>

maximum allowed compressed size, a string like '1.5M' indicating '1.5 megabytes'. Supported units: - B = bytes - K = kilobytes - M = megabytes - G = gigabytes

Default

50M

--max-allowed-size-uncompressed <max_allowed_size_uncompressed>

maximum allowed uncompressed size, a string like '1.5M' indicating '1.5 megabytes'. Supported units: - B = bytes - K = kilobytes - M = megabytes - G = gigabytes

Default

75M

--unexpected-directory-patterns <unexpected_directory_patterns>

comma-delimited list of patterns matching directories that are not expected to be found in the distribution. Patterns should be in the format understood by `fnmatch.fnmatchcase()`. See https://docs.python.org/3/library/fnmatch.html.

Default

```
*/.appveyor,*/.binder,*/.circleci,*/.git,*/.github,*/.idea,*/.
pytest_cache,*/.mypy_cache
```

--unexpected-file-patterns <unexpected_file_patterns>

comma-delimited list of patterns matching files that are not expected to be found in the distribution. Patterns should be in the format understood by `fnmatch.fnmatchcase()`. See https://docs.python.org/3/library/fnmatch.html.

Default

```
*/appveyor.yml,*/.appveyor.yml,*/azure-pipelines.yml,*/.azure-pipelines.
yml,*/.cirrus.star,*/.cirrus.yml,*/codecov.yml,*/.codecov.yml,*/.
DS_Store,*/.gitignore,*/.gitpod.yml,*/.hadolint.yaml,*/.readthedocs.
yaml,*/.travis.yml,*/vsts-ci.yml,*/.vsts-ci.yml
```

Arguments

FILEPATHS

Optional argument(s)

2.2 pyproject.toml

If a file `pyproject.toml` exists in the working directory `pydistcheck` is run from, `pydistcheck` will look there for configuration.

Alternatively, a path to a TOML file can be provided via CLI argument `--config`.

Put configuration in a `[tool.pydistcheck]` section.

The example below contains all of the configuration options for `pydistcheck`, set to their default values.

```toml
[tool.pydistcheck]

ignore = []
inspect = false
max_allowed_files = 2000
max_allowed_size_compressed = '50M'
max_allowed_size_uncompressed = '75M'
unexpected_directory_patterns = [
    '*/.appveyor',
    '*/.binder',
    '*/.circleci',
    '*/.git',
    '*/.github',
    '*/.idea',
    '*/.pytest_cache',
    '*/.mypy_cache'
]
unexpected_file_patterns = [
    '*/appveyor.yml',
    '*/.appveyor.yml',
    '*/azure-pipelines.yml',
    '*/.azure-pipelines.yml',
    '*/.cirrus.star',
    '*/.cirrus.yml',
    '*/codecov.yml',
    '*/.codecov.yml',
    '*/.DS_Store',
    '*/.gitignore',
    '*/.gitpod.yml',
    '*/.hadolint.yaml',
    '*/.readthedocs.yaml',
    '*/.travis.yml',
    '*/vsts-ci.yml',
    '*/.vsts-ci.yml'
]
```

CHAPTER

THREE

CHECK REFERENCE

This page describes each of the checks that `pydistcheck` performs. The section headings correspond to the error codes printed in `pydistcheck`'s output.

3.1 compiled-objects-have-debug-symbols

The distribution contains compiled objects, like C/C++ shared libraries, with debug symbols.

Compilers for languages like C, C++, Fortran, and Rust can optionally include additional information like source code file names and line numbers, and other information useful for printing stack traces or enabling interactive debugging.

The inclusion of such information can increase the size of built objects substantially. It's `pydistcheck`'s position that the inclusion of such debug symbols in a shared library distributed as part of Python wheel is rarely desirable, and that by default wheels shouldn't include that type of information.

This check attempts to run the following tools with `subprocess.run()`.

- `dsymutil`
- `llvm-nm`
- `llvm-objdump`
- `nm`
- `objdump`
- `readelf`

Installing more of these in the environment where you run `pydistcheck` improves its ability to detect debug symbols.

> **Warning:** If `pydistcheck` invoking these other tools with `subprocess.run()` is a concern for you (for example, if it causes permissions-related issues), turn this check off by passing it to `--ignore`.

For a LOT more information about this topic, see these discussions in other open source projects.

- "(auditwheel) Add –strip option to 'repair'"
- "(cibuildwheel) Strip debug symbols of wheels"
- "(numpy) ENH:Distutils Remove debugging symbols by default"
- "(psycopg2) Excessive size of wheel packages"
- "(scikit-image) Add linker flags to strip debug symbols during wheel building "
- "(scylladb/python-driver) scylla-driver is 100 times larger than cassandra-driver"

And these other resources.

- "Adding debugging information to your native extension" (memray docs)
- "How can I tell if a binary was compiled with debug symbols?" (vscode-lldb docs)

3.2 distro-too-large-compressed

The package distribution is larger (compressed) than the allowed size.

Change that limit using configuration option `max-distro-size-compressed`.

3.3 distro-too-large-uncompressed

The package distribution is larger (uncompressed) than the allowed size.

Change that limit using configuration option `max-distro-size-uncompressed`.

3.4 files-only-differ-by-case

The package distribution contains filepaths which are identical after lowercasing.

Such paths are not portable, as some filesystems (notably macOS), are case-insensitive.

3.5 mixed-file-extensions

Filepaths in the package distribution use a mix of file extensions for the same type of file.

For example, `some_file.yaml` and `other_file.yml`.

Some programs may use file extensions, instead of more reliable mechanisms like magic bytes to detect file types, like this:

```
if filepath.endswith(".yaml"):
```

In such cases, having a mix of file extensions can lead to only a subset of relevant files being matched.

Standardizing on a single extension for files of the same type improves the probability of either catching or completely avoiding such bugs… either all intended files will be matched or none will.

3.6 path-contains-non-ascii-characters

At least one filepath in the package distribution contains non-ASCII characters.

Non-ASCII characters are not portable, and their inclusion in filepaths can lead to installation and usage issues on different platforms.

For more information, see:

- "Archives Containing Non-ASCII Filenames" (Oracle docs)
- example issue from pillow/PIL

- "Unix and non-ASCII file names, a summary of issues"
- `jqlang/jq#811`: "File names with non ASCII characters"

3.7 path-contains-spaces

At least one filepath in the package distribution contains spaces.

Filepaths with spaces require special treatment, like quoting in some settings. Avoiding paths with spaces eliminates a whole class of potential issues related to software that doesn't handle such paths well.

For more information, see:

- "Long filenames or paths with spaces require quotation marks" (Windows docs)
- "Don't use spaces or underscores in file paths" (blog post)
- "What technical reasons exist for not using space characters in file names?" (Stack Overflow)

3.8 too-many-files

The package distribution contains more than the allowed number of files.

This is a very very rough way to detect that unexpected files have been included in a new release of a project.

`pydistcheck` defaults to raising this error when a distribution has more than 2000 files...a totally arbitrary number chosen by the author.

To change that limit, use configuration option `max-allowed-files`.

3.9 unexpected-files

Files were found in the distribution which weren't expected to be included.

With `pydistcheck`'s default settings, this check raises errors for the inclusion of files that are commonly found in source control during development but are not useful in distributions, like `.gitignore`.

Which files are "expected" is highly project-specific. See *Configuration* for a list of the files `pydistcheck` complains about by default, and for information about how to customize that list.

CHAPTER

FOUR

HOW TO TEST A PYTHON DISTRIBUTION

4.1 Introduction

Your code is good. Really good.

You enforced consistent style with `black`, `isort`, and `pydocstyle`.

You checked for a wide range of performance, correctness, and readability issues with `flake8`, `mypy`, `pylint`, and `ruff`.

You ran extensive tests with `pytest` and `hypothesis`.

You even scanned for legal and security issues with tools like `pip-audit`, `pip-licenses`, and `safety`.

So finally, FINALLY, it's time to package it up into a tarball and upload it to PyPI.

```
python -m build .
```

... right?

Hopefully! But let's check.

- *Are those distributions valid zip or tar files?*
- *Will their READMEs look pretty when rendered on the PyPI homepage?*
- *Do they have correctly-formatted platform tags?*
- *Are they as small as possible, to be kind to package repositories and users with weak internet connections?*
- *Are they free from filepaths and file names that some operating systems will struggle with?*

You checked those things, right? And in continuous integration, with open-source tools, not with manual steps and random `tar` incantations copied from Stack Overflow?

If yes, great! I'm proud of you.

If you're feeling like you could use some help with that… read on.

4.2 Quickstart

After building at least one wheel and sdist…

```
python -m build -o dist .
```

Run the following on those distributions to catch a wide range of packaging issues.

```
# are all the sdists valid gzipped tar files?
gunzip -tv dist/*.tar.gz

# are all the wheels valid zip files?
zip -T dist/*.whl

# is the package metadata well-formatted?
pyroma --min=10 dist/*.tar.gz
twine check --strict dist/*

# is the distribution properly structured and portable?
check-wheel-contents dist/*.whl
pydistcheck --inspect dist/*
```

4.3 List of Tools

The following open-source tools can be used to detect (and in some cases repair) a wide range of Python packaging issues.

- `abi3audit` (link) = detect ABI incompatibilities in wheels with CPython extensions
- `auditwheel` (link) = detect and repair issues in Linux wheels that link to external shared libraries
- `auditwheel-emscripten` (link) = like `auditwheel`, but focused on Python-in-a-web-browser applications (e.g. pyodide auditwheel)
- `auditwheel-symbols` (link) = detect which symbols in a Linux wheel's shared library are causing `auditwheel` to sugggest a more recent `manylinux` tag
- `check-manifest` (link) = check that sdists contain all the files you expect them to, based on what you've checked into version control
- `check-wheel-contents` (link) = detect unnecessary files, import issues, portability problems in wheels
- `delocate` (link) = detect and repair issues in macOS wheels that link to external shared libraries
- `delvewheel` (link) = detect and repair issues in Windows wheels that link to external shared libraries
- `pydistcheck` (link) = detect portability problems in wheels and sdists
- `pyroma` (link) = detect incomplete or malformed metadata in sdists
- `twine` (link) = detect issues in package metadata (via `twine check`)
- `wheel-inspect` (link) = dump summary information about wheels into machine-readable formats

INDEX

Symbols

F

P