Invocations

Release

Aug 13, 2019

Contents

Invocations is a collection of reusable Invoke tasks, task collections and helper functions. Originally sourced from the Invoke project's own project-management tasks file, they are now highly configurable and used across a number of projects, with the intent to become a clearinghouse for implementing common best practices.

Currently implemented topics include (but are not limited to):

- management of Sphinx documentation trees
- Python project release lifecycles
- dependency vendoring
- running test suites (unit, integration, coverage-oriented, etc)
- console utilities such as confirmation prompts

and more.

CHAPTER 1

Roadmap

While Invocations has been released with a major version number to signal adherence to semantic versioning, it's somewhat early in development and has not fully achieved its design vision yet.

We expect it to gain maturity in tandem with the adoption and development of Invoke post-1.x. It's also highly likely that Invocations will see a few major releases as its API (and those of its sister library, patchwork) matures.

CHAPTER 2

Contents

2.1 Changelog

- : Remove some apparently non-functional `setup.py` logic around conditionally requiring `enum34`; it was never getting selected and thus breaking a couple modules that relied on it.

  `enum34` is now a hard requirement like the other semi-optional-but-not-really requirements.

- : Was missing a 'hide output' flag on a subprocess shell call, the result of which was mystery git branch names appearing in the output of `inv release` and friends. Fixed now.

- : Remove some apparently non-functional `setup.py` logic around conditionally requiring `enum34`; it was never getting selected and thus breaking a couple modules that relied on it.

  `enum34` is now a hard requirement like the other semi-optional-but-not-really requirements.

- : Was missing a 'hide output' flag on a subprocess shell call, the result of which was mystery git branch names appearing in the output of `inv release` and friends. Fixed now.

- : Remove some apparently non-functional `setup.py` logic around conditionally requiring `enum34`; it was never getting selected and thus breaking a couple modules that relied on it.

  `enum34` is now a hard requirement like the other semi-optional-but-not-really requirements.

- : Split out the body of the (sadly incomplete) `packaging.release.all` task into the better-named `packaging.release.prepare`. (`all` continues to behave as it did, it just now calls `prepare` explicitly.)

- : Pre-history / code primarily for internal consumption

CHAPTER 3

API/task docs

3.1 autodoc

Sphinx autodoc hooks for documenting Invoke-level objects such as tasks.

Unlike most of the rest of Invocations, this module isn't for reuse in the "import and call functions" sense, but instead acts as a Sphinx extension which allows Sphinx's autodoc functionality to see and document Invoke tasks and similar Invoke objects.

Note: This functionality is mostly useful for redistributable/reusable tasks which have been defined as importable members of some Python package or module, as opposed to "local-only" tasks that live in a single project's `tasks.py`.

However, it will work for any tasks that Sphinx autodoc can import, so in a pinch you could for example tweak `sys.path` in your Sphinx `conf.py` to get it loading up a "local" tasks file for import.

To use:

- Add `"sphinx.ext.autodoc"` and `"invocations.autodoc"` to your Sphinx `conf.py`'s `extensions` list.
- Use Sphinx autodoc's `automodule` directive normally, aiming it at your tasks module(s), e.g. `.. automodule:: myproject.tasks` in some `.rst` document of your choosing.
  - As noted above, this only works for modules that are importable, like any other Sphinx autodoc use case.
  - Unless you want to opt-in which module members get documented, use `:members:` or add `"members"` to your `conf.py`'s `autodoc_default_flags`.
  - By default, only tasks with docstrings will be picked up, unless you also give the `:undoc-members:` flag or add `:undoc-members:` / add `"undoc-members"` to `autodoc_default_flags`.
  - Please see the autodoc docs for details on these settings and more!
- Build your docs, and you should see your tasks showing up as documented functions in the result.

3.2 console

Text console UI helpers and patterns, e.g. 'Y/n' prompts and the like.

invocations.console.**confirm**(*question*, *assume_yes=True*)

Ask user a yes/no question and return their response as a boolean.

`question` should be a simple, grammatically complete question such as "Do you wish to continue?", and will have a string similar to `" [Y/n] "` appended automatically. This function will *not* append a question mark for you.

By default, when the user presses Enter without typing anything, "yes" is assumed. This can be changed by specifying `assume_yes=False`.

Note: If the user does not supplies input that is (case-insensitively) equal to "y", "yes", "n" or "no", they will be re-prompted until they do.

Parameters

- **question** (*str*) – The question part of the prompt.
- **assume_yes** (*bool*) – Whether to assume the affirmative answer by default. Default value: `True`.

Returns A `bool`.

3.3 docs

Tasks for managing Sphinx documentation trees.

invocations.docs.**build**(*c*, *clean=False*, *browse=False*, *nitpick=False*, *opts=None*, *source=None*, *target=None*)

Build the project's Sphinx docs.

invocations.docs.**doctest**(*c*)

Run Sphinx' doctest builder.

This will act like a test run, displaying test results & exiting nonzero if all tests did not pass.

A temporary directory is used for the build target, as the only output is the text file which is automatically printed.

invocations.docs.**sites**(*c*)

Build both doc sites w/ maxed nitpicking.

invocations.docs.**tree**(*c*)

Display documentation contents with the 'tree' program.

invocations.docs.**watch_docs**(*c*)

Watch both doc trees & rebuild them if files change.

This includes e.g. rebuilding the API docs if the source code changes; rebuilding the WWW docs if the README changes; etc.

Reuses the configuration values `packaging.package` or `tests.package` (the former winning over the latter if both defined) when determining which source directory to scan for API doc updates.

3.4 packaging

3.4.1 packaging.release

Python package release tasks.

This module assumes:

- you're using semantic versioning for your releases
- you maintain a file called `$package/_version.py` containing normal version conventions (`__version_info__` tuple and `__version__` string).

`invocations.packaging.release.`**`all_`**`(c)`
Catchall version-bump/tag/changelog/PyPI upload task.

`invocations.packaging.release.`**`build`**`(c, sdist=True, wheel=False, directory=None, python=None, clean=True)`
Build sdist and/or wheel archives, optionally in a temp base directory.

All parameters save `directory` honor config settings of the same name, under the `packaging` tree. E.g. say `.configure({'packaging': {'wheel': True}})` to force building wheel archives by default.

Parameters

- **`sdist`** (*bool*) – Whether to build sdists/tgzs.
- **`wheel`** (*bool*) – Whether to build wheels (requires the `wheel` package from PyPI).
- **`directory`** (*str*) – Allows specifying a specific directory in which to perform builds and dist creation. Useful when running as a subroutine from `publish` which sets up a temporary directory.

  Two subdirectories will be created within this directory: one for builds, and one for the dist archives.

  When `None` or another false-y value, the current working directory is used (and thus, local `dist/` and `build/` subdirectories).
- **`python`** (*str*) – Which Python binary to use when invoking `setup.py`.

  Defaults to just `python`.

  If `wheel=True`, then this Python must have `wheel` installed in its default `site-packages` (or similar) location.
- **`clean`** (*bool*) – Whether to clean out the local `build/` folder before building.

`invocations.packaging.release.`**`prepare`**`(c)`
Edit changelog & version, git commit, and git tag, to set up for release.

`invocations.packaging.release.`**`publish`**`(c, sdist=True, wheel=False, index=None, sign=False, dry_run=False, directory=None, dual_wheels=False, alt_python=None, check_desc=False)`
Publish code to PyPI or index of choice.

All parameters save `dry_run` and `directory` honor config settings of the same name, under the `packaging` tree. E.g. say `.configure({'packaging': {'wheel': True}})` to force building wheel archives by default.

Parameters

- **`sdist`** (*bool*) – Whether to upload sdists/tgzs.

- **wheel** (*bool*) – Whether to upload wheels (requires the `wheel` package from PyPI).
- **index** (*str*) – Custom upload index/repository name. See `upload` help for details.
- **sign** (*bool*) – Whether to sign the built archive(s) via GPG.
- **dry_run** (*bool*) – Skip actual publication step if `True`.

  This also prevents cleanup of the temporary build/dist directories, so you can examine the build artifacts.
- **directory** (*str*) – Base directory within which will live the `dist/` and `build/` directories.

  Defaults to a temporary directory which is cleaned up after the run finishes.
- **dual_wheels** (*bool*) – When `True`, builds individual wheels for Python 2 and Python 3.

  Useful for situations where you can't build universal wheels, but still want to distribute for both interpreter versions.

  Requires that you have a useful `python3` (or `python2`, if you're on Python 3 already) binary in your `$PATH`. Also requires that this other python have the `wheel` package installed in its `site-packages`; usually this will mean the global site-packages for that interpreter.

  See also the `alt_python` argument.
- **alt_python** (*str*) – Path to the 'alternate' Python interpreter to use when `dual_wheels=True`.

  When `None` (the default) will be `python3` or `python2`, depending on the currently active interpreter.
- **check_desc** (*bool*) – Whether to run `setup.py check -r -s` (uses `readme_renderer`) before trying to publish - catches long_description bugs. Default: `False`.

`invocations.packaging.release.`**`status`**(*c*)

Print current release (version, changelog, tag, etc) status.

Doubles as a subroutine, returning the return values from its inner call to `_converge` (an `(actions, state)` two-tuple of Lexicons).

`invocations.packaging.release.`**`upload`**(*c*, *directory*, *index=None*, *sign=False*, *dry_run=False*)

Upload (potentially also signing) all artifacts in `directory`.

Parameters

- **index** (*str*) – Custom upload index/repository name.

  By default, uses whatever the invoked `pip` is configured to use. Modify your `pypirc` file to add new named repositories.
- **sign** (*bool*) – Whether to sign the built archive(s) via GPG.
- **dry_run** (*bool*) – Skip actual publication step if `True`.

  This also prevents cleanup of the temporary build/dist directories, so you can examine the build artifacts.

3.4.2 packaging.vendorize

Tasks for importing external code into a vendor subdirectory.

invocations.packaging.vendorize.**vendorize**(*c*, *distribution*, *version*, *vendor_dir*, *package=None*, *git_url=None*, *license=None*)

Vendorize Python package `distribution` at version/SHA `version`.

Specify the vendor folder (e.g. `<mypackage>/vendor`) as `vendor_dir`.

For Crate/PyPI releases, `package` should be the name of the software entry on those sites, and `version` should be a specific version number. E.g. `vendorize('lexicon', '0.1.2')`.

For Git releases, `package` should be the name of the package folder within the checkout that needs to be vendorized and `version` should be a Git identifier (branch, tag, SHA etc.) `git_url` must also be given, something suitable for `git clone <git_url>`.

For SVN releases: xxx.

For packages where the distribution name is not the same as the package directory name, give `package='name'`.

By default, no explicit license seeking is done – we assume the license info is in file headers or otherwise within the Python package vendorized. This is not always true; specify `license=/path/to/license/file` to trigger copying of a license into the vendored folder from the checkout/download (relative to its root.)

3.5 pytest

Pytest-using variant of testing.py. Will eventually replace the latter.

invocations.pytest.**coverage**(*c*, *report='term'*, *opts=''*, *tester=None*)

Run pytest with coverage enabled.

Assumes the `pytest-cov` pytest plugin is installed.

Parameters

- **report** (*str*) – Coverage report style to use. If 'html', will also open in browser.
- **opts** (*str*) – Extra runtime opts to pass to pytest.
- **tester** – Specific test task object to invoke. If `None` (default), uses this module's local *test*.

invocations.pytest.**integration**(*c*, *opts=None*, *pty=True*)

Run the integration test suite. May be slow!

invocations.pytest.**test**(*c*, *verbose=True*, *color=True*, *capture='sys'*, *module=None*, *k=None*, *x=False*, *opts=''*, *pty=True*)

Run pytest with given options.

Parameters

- **verbose** (*bool*) – Whether to run tests in verbose mode.
- **color** (*bool*) – Whether to request colorized output (typically only works when `verbose=True`.)
- **capture** (*str*) – What type of stdout/err capturing pytest should use. Defaults to `sys` since pytest's own default, `fd`, tends to trip up subprocesses trying to detect PTY status. Can be set to `no` for no capturing / useful print-debugging / etc.
- **module** (*str*) – Select a specific test module to focus on, e.g. `main` to only run `tests/main.py`. (Note that this is a specific idiom aside from the use of `-o '-k pattern'`.) Default: `None`.
- **k** (*str*) – Convenience passthrough for `pytest -k`, i.e. test selection. Default: `None`.

- **x** (*bool*) – Convenience passthrough for `pytest -x`, i.e. fail-fast. Default: `False`.
- **opts** (*str*) – Extra runtime options to hand to `pytest`.
- **pty** (*bool*) – Whether to use a pty when executing pytest. Default: `True`.

Python Module Index

i

Index