yabs Documentation

Release 0.4.0

Martin Wendt

Apr 01, 2021

Contents

1 Installation **3**
- 1.1 Install Into the System Python . . . 3
- 1.2 Run From a Virtual Environment . . . 3

2 User Guide **5**
- 2.1 Quickstart . . . 5
- 2.2 Tutorial . . . 5
- 2.3 Writing Scripts . . . 8
- 2.4 yabs.yaml . . . 13
- 2.5 Script Reference . . . 16
- 2.6 Command Line Interface . . . 21
- 2.7 Writing Plugins . . . 23

3 Reference Guide **25**
- 3.1 Architecture . . . 25
- 3.2 API Reference . . . 25
- 3.3 Index . . . 30

4 Development **31**
- 4.1 Install for Development . . . 31
- 4.2 Run Tests . . . 32
- 4.3 Code . . . 32
- 4.4 Create a Pull Request . . . 32

5 Release Info **33**

6 Features **35**

7 Quickstart **37**

Python Module Index **39**

Index **41**

Test, Build, Deliver!

0.4, Date: Apr 01, 2021

Warning: Yabs has currently *beta* status.

CHAPTER 1

Installation

Requirements: Python 3.5+ is required. Releases are hosted on PyPI and can be installed using pip or pipenv.

1.1 Install Into the System Python

Installing *yabs* as part of your system's Python will make the available from the command line. You may need administrator permissions, like `sudo`. Also make sure to use Python 3 if the system installation uses Python 2 (as on macOS).

For example:

```
$ sudo python3 -m pip install -U yabs
$ yabs --version -v
yabs/1.0.0 Python/3.6.1 Darwin-17.6.0-x86_64-i386-64bit
$ yabs --help
...
```

1.2 Run From a Virtual Environment

Installing *yabs* and its dependencies into a 'sandbox' will help to keep your system Python clean, but requires to activate the virtual environment:

```
$ cd /path/to/yabs
$ pipenv shell
(yabs) $ pipenv install yabs --upgrade
(yabs) $ yabs --version -v
yabs/0.0.1 Python/3.6.1 Darwin-17.6.0-x86_64-i386-64bit
(yabs) $ yabs --help
...
```

See also:

See *Development* for directions for contributors.

Now the `yabs` command is available:

```
$ yabs --help
```

and the `yabs` package can be used in Python code:

```
$ python
>>> from yabs import __version__
>>> __version__
'0.0.1'
```

CHAPTER 2

User Guide

2.1 Quickstart

1. Install *yabs* (*details*)
2. Create and edit the workflow definition script (*yabs.yaml*) (*details*)
3. Run the script:

```
$ yabs run --inc patch
```

Use the `--dry-run` or `-n` argument test the workflow without really releasing. Use the `--verbose` or `-v` to increase verbosity. Use the `--workflow` argument to specify the location of the configuration file if it is not at the default location *./yabs.yaml*.

```
$ yabs run --inc minor --workflow /path/to/yabs.yaml --dry-run
```

2.2 Tutorial

2.2.1 Overview

Yabs is a command line tool, that runs a sequence of tasks in order to test, build, and deliver a Python software project.

The workflow is defined in a configuration file, using a simple YAML format and can be executed like

```
$ yabs run --inc minor
```

The example above assumes that the config file is found at the default location `./yabs.yaml`. The workflow refers to the `--inc` argument for the 'bump' task (in this case a minor version increment).

A typical release workflow may look like this:

1. Check preconditions: *Is the workspace clean, anything to commit?*, *Is GitHub reachable?*, *Are we on the correct branch?*, . . .
2. Make sure that static code linters and unit tests pass, run tox.
3. Bump the project's version number (major, minor, or patch, according to Semantic Versioning). Then patch the version string into the respective Python module or text file.
4. Build *sdist* and *wheel* assets.
5. Tag the version, commit, and push.
6. Upload distribution to PyPI.
7. Create a new release on GitHub and upload assets.
8. Bump, tag, commit, and push for post-release.

Some **preconditions** are assumed:

- We use git, PyPI and GitHub.
- Version numbers follow roughly the Semantic Versioning pattern.
- The project's version number is maintained in *one of the supported locations*.

Note: Yabs can be extended using the plugin API.

2.2.2 Workflow Definition

A **workflow definition** is a YAML file that defines some general settings and a sequence of *tasks*.

Tasks are the building blocks of a workflow. They have a type name and additonal parameters.

The internal **Task Runner** executes the task sequence and passes a **Task Context** along. This allows an upstream task to pass information downstream. For examle a *bump* task will set a new version number that may be used in a commit message *template*.

Some string parameters are evaluated as **template**, i.e. included *macros*, like `"{version}"`, are expanded.

`yabs.yaml`:

```yaml
file_version: yabs#1
config:
  repo: 'mar10/test-release-tool'
  version:
    # Location of the project's version:
    - type: __version__
      file: src/test_release_tool/__init__.py

tasks:
  - task: check
    branches: master
    github: true
    clean: true

  - task: exec
    args: ["tox", "-e", "lint"]
    always: true

  - task: bump  # bump version according to `--inc` argument
```

(continues on next page)

(continued from previous page)

```
  - task: commit
    message: |
      Bump version to {version}

  - ...
```

See Writing Scripts for details.

2.2.3 Versions

Python projects should have a version number that is stored at *one* central location. This version number will appear in the about box, when a CLI is called with a `--version` argument, when `setup.py --version` is called, etc. Most importantly, it is used to generate tag names, that uniquely identify PyPI releases.

Especially when our project is a kind of a library that other projects may depend on, incrementing ('bumping') the version number is an important step in the release process. Installation tools like pip and Pipenv rely heavily on consistent version number schemes, when defining requirements:

```
[dev-packages]
black = "==19.10b0"
tox = "~=3.2"
[packages]
PyYAML = "~=5.2"
...
```

See also PEP 440. Yabs assumes that a version number consists of three parts and optional extension, as described in Semantic Versioning, e.g. `"1.2.3"`, `"1.2.4-a1"`.

After-Release Versions

(**TODO:** verify this section.)

After a new release was published, we should do another bump. This will make sure that any following code change is not accidently associated with the public tag name.

This *after-release* version number should

- have a higher sort order than the previous release, i.e. compare *greater than* (>) our current release
- however the increment should be small, since we should never *de*crement a version number, and we don't know by now if the next release will contain major-, minor-, or patch-level changes
- have a format that indicates a *preliminary* status (i.e. not be installed, unless `--pre` is passed to `pip`)

SemVer does support pre-releases, but not post-releases (only build metadata, which is not sortable). Pre-releases are considered 'unstable', which is what we want to signal until we make the next release.

PEP 440 supports pre-, post-, and developmental releases.

Yabs suggests this pattern:

> After a release, bump & commit a patch-incremented version with pre-release appendix, for example: `v1.2.3` `v1.2.4a1` The next release will be a patch, minor, or major increment, which brings us to `v1.2.4`, `v1.3.0`, or `v2.0.0`.

Version Locations

Although there seems to be consent that Python projects should have a version number that is stored at *one* central location, the community has not agreed upon that location yet.

Yabs supports some common approaches, that you can configure under `config.version`, for example:

`yabs.yaml`:

```
file_version: yabs#1
config:
  ...
   version:
     - type: __version__
       file: src/test_release_tool/__init__.py
 ...
```

See Writing Scripts for details.

2.3 Writing Scripts

2.3.1 Overview

Workflow deinitions are text files in YAML format, that are read, compiled, and executed by the *Task Runner*.

Note: Unless you are a Python programmer, you may have to get used to the fact that **whitespace matters** in YAML files: Make sure you indent uniformly. Don't mix tabs and spaces. We recommend to use an editor that supports the YAML syntax (e.g. VS Code).

A simple confuguration script may look like this: `yabs.yaml`:

```
# Yabs Workflow Definition
# See https://github.com/mar10/yabs
file_version: yabs#1

config:
  repo: 'mar10/test-release-tool'
  version:
    - type: __version__
      file: src/test_release_tool/__init__.py
  branches:
    - master


tasks:
  # The following tools are available. They are executed in the order
  # listed here

  # 'check': Assert preconditons and fail otherwise
  - task: check
    branch: master          # Current branch must be in this list
    can_push: true          # Test if 'git push' would/would not succeed
    clean: true             # Repo must/must not contain modifications
    python: ">=3.5"         # SemVer specifier
```

(continues on next page)

(continued from previous page)

```
    twine: true             # `twine` is available
    up_to_date: true        # everything pulled from remote
    venv: true              # running inside a virtual environment
    version: true           # `setup.py --version` returns the configured version

  # 'exec': Run arbitrary shell command
  - task: exec
    args: ["tox", "-e", "lint"]     # shell command and optional arguments
    always: true            # `true`: run even in dry-run mode

  # 'bump': increment manifest.version and synchronize other JSON files
  - task: bump
    inc: null               # Use value passed as 'yabs run --inc INC'

  # 'commit': Commit modified files
  - task: commit
    add_known: true         # Commit with -a flag
    message: |
      Bump version to {version}

  # 'tag': Create an annotated tag
  - task: tag
    name: v{version}
    message: |
      Version {version}

  # 'push': Push changes and tags
  - task: push
    tags: true

  # 'pypi_release': Create a release on PyPI using `twine`
  - task: pypi_release
    build:
      - sdist
      - bdist_wheel
    upload: true

  # 'github_release': Create a release on GitHub
  - task: github_release
    draft: false

  # Bump 'v1.2.3' => 'v1.2.4-a0'
  - task: bump
    inc: "postrelease"

  # Commit using '[ci skip]' as part of the message to prevent CI testing
  - task: commit
    add_known: true
    message: |
      Bump prerelease ({version}) [ci skip]

  # Push to GitHub
  - task: push
```

See *yabs.yaml* for a complete configuration with all available options and defaults.

2.3.2 Task Types

See also:

See *Script Reference* for a list of all tasks and options and *yabs.yaml* for a complete configuration file with all available tasks.

2.3.3 Template Macros

Some tasks have string options such as tag names, commit messges, etc. These strings may contain inline *macros* that will be expanded.

Typical macros are *version*, *tag_name*, *repo*, ... Macro names must be embedded in curly braces, for example:

```
- task: github_release
  name: 'v{version}'
  message: |
    Released {version}
    [Commit details](https://github.com/{repo}/compare/{org_tag_name}...{tag_name}).
```

All attributes of the task context are available as macros:

{inc} Value of the `--inc` argument.

{org_tag_name} The repo's latest tag name (before 'bump').

{org_version} Latest version (before 'bump').

{repo} GitHub repo name, e.g. 'USER/PROJECT'.

{tag_name} The current tag name (after 'bump').

{version} Current version (after 'bump').

See *TaskContext* for a complete list.

2.3.4 Version Locations

Note: Currently only a small subset is implemented. Please open an issue if you need another one and are ready to help with testing.

(**TODO:** verify this section.)

Although there seems to be consent that Python projects should have a version number that is stored at *one* central location, the community has not agreed upon that location yet.

In order to find and bump this versions, we need to pass a hint in the configuration *yabs.yaml* like so:

```
file_version: yabs#1
config:
  ...
  version:
    - type: __version__  # Example!
      file: src/my_project/__init__.py
...
```

Yabs supports some common approaches. Following some typical patterns how Python projects store version numbers.

Note: Currently we would recommend this variant (unless Poetry is used): Store the version in `__init__.py` of the project's root folder:

```
__version__ = "1.2.3"
```

Then reference this in `setup.cfg`:

```
[metadata]
name = my_package
version = attr: my_project.__version__
```

This would then configured in `yabs.yaml` like so:

```
config:
  version:
    - type: __version__
      file: my_project/__init__.py
```

See below for details about the different use cases.

Poetry

Todo: Not yet implemented.

Poetry stores the version number in its own section in `pyproject.toml` (defined in PEP-518):

`pyproject.toml`:

```
[project]
...
[tool.poetry]
name = "my_project"
version =  "1.2.3"
```

`yabs.yaml`:

```
config:
  version:
    - type: poetry
```

flit

Todo: Not yet implemented.

__init__.py of the project's root package

`__init__.py`:

```
__version__ = "1.2.3"
```

`yabs.yaml`:

```
config:
  version:
    - type: __version__
      file: src/my_project/__init__.py
```

Or a variant the mimics Python's *sys.version_info* style:

`__init__.py`:

```
version_info = (1, 2, 3)
version = ".".join(str(c) for c in version_info)
```

`yabs.yaml`:

```
config:
  version:
    # TODO
```

Plain Text File

For example a *_version.txt* file in the project's *src* folder containing:

`_version.txt`:

```
1.2.3
```

`yabs.yaml`:

```
config:
  version:
    # TODO
```

setup.cfg

See also PEP-396 and setuptools .

`setup.cfg` in the project's root folder:

```
[metadata]
name = my_package
version = 1.2.3
```

`yabs.yaml`:

```
config:
  version:
    # TODO
```

The follwing two examples for setup.cfg use the special `attr:` and `file:` directives that where introduced with setuptools v39.2).

Note: This assumes that the version is stored in a separate text- or Python file, which is covered in the examples above.

```
[metadata]
name = my_package
version = attr: my_project.__version__
```

```
[metadata]
name = my_package
version = file: path/to/file
```

The follwing two examples for setup.cfg use the special `version-file` and `version-from-file` options that where proposed for distutils2.

Note: This assumes that the version is stored in a separate text- or Python file, which is covered in the examples above.

```
[metadata]
# The entire contents of the file contains the version number
version-file = version.txt
```

```
[metadata]
# The version number is contained within a larger file, e.g. of Python code,
# such that the file must be parsed to extract the version
version-from-file = elle.py
```

2.3.5 Debugging

Use the `--verbose` (short `-v`) option to generate more console logging. Use the `--dry-run` (short `-n`) option to run all tasks in a simulation mode:

```
$ yabs run --inc patch -vn
```

2.4 yabs.yaml

The following file lists all available tasks with all available options and respective defaults.

Note: This is not a meaningful or realistic workflow definition, but rather a demonstration of what's available. A realistic workflow would omit default options and execute tasks in a more useful order. See *Writing Scripts* for an example.

2.4.1 Annotated Sample Configuration

```
# Release-Tool Workflow Definition
# See https://github.com/mar10/yabs
file_version: yabs#1
config:
  # Options used as default for all tools in this workflow
  project: ''                # Mandatory:
  repo: 'mar10/test-release-tool'
  # GitHub access token
  gh_auth:
    oauth_token_var: GITHUB_OAUTH_TOKEN
```

(continues on next page)

(continued from previous page)

```
  version:
    - type: __version__         # First entry is master for synchronizing
      file: src/test_release_tool/__init__.py
      # match: '__version__\s*=\s*[''\"](\d+\.\d+\.\d+).*[''\"]'
      # template: '__version__ = "{version}"'
    # - type: setup_cfg         # First entry is master for synchronizing
    #   entry: metadata.version
    #   template:
  max_increment: minor
  branches:                   # Allowed git branches
    - master


tasks:
  # The following tools are available. They are executed in the order
  # listed here

  # 'check': Assert preconditons and fail otherwise
  - task: check
    branches: master          # Current branch must be in this list
    can_push: true            # Test if 'git push' would/would not succeed
    # connected: null           # TODO: internet available
    github: true              # GitHub repo name valid and online accessible
    clean: true               # Repo must/must not contain modifications
    # cmp_version: null         # E.g. set to 'gt' to assert that the current
    #                           # version is higher than the latest tag (gt,
    #                           # gte, lt, lte, eq, neq)
    os: null                  # (str, list)
    python: ">=3.5"           # SemVer specifier
    twine: true               # `twine` is available
    up_to_date: true          # everything pulled from remote
    venv: true                # running inside a virtual environment
    version: true             # `setup.py --version` returns the configured version

  # 'run': Run arbitrary shell command
  - task: exec
    args: ["tox", "-e", "lint"]     # shell command and optional arguments
    # dry_run_args: ["pwd"]   #
    always: true              # `true`: run even in dry-run mode
    silent: true              # `true`: suppress output
    ignore_errors: false      # `true`: show warning, but proceed on errors (exit code !
↪= 0)

  - task: exec
    args: ["tox"]             # shell command and optional arguments
    # dry_run_args: ["pwd"]   #
    always: true              # `true`: run even in dry-run mode
    silent: true              # `true`: suppress output
    ignore_errors: false      # `true`: show warning, but proceed on errors (exit code !
↪= 0)

  # 'bump': increment manifest.version and synchronize other JSON files
  - task: bump
    # bump also requires a mode argument (yabs:target:MODE)
    inc: null                 # Use value passed as 'yabs run INC'

  # # 'replace': In-place string replacements
```

(continues on next page)

(continued from previous page)

```yaml
  # # (Uses https://github.com/outaTiME/applause)
  # - task: replace
  #  files: null               # minimatch globbing pattern
  #  patterns: []              # See https://github.com/outaTiME/applause
  #  # Shortcut patterns (pass false to disable):
  #  setTimestamp: "{%= grunt.template.today('isoUtcDateTime') %}"
  #                            # Replace '@@timestamp' with current datetime
  #  setVersion: '{version}'   # Replace '@@version' with current version

    # 'commit': Commit modified files
  - task: commit
    add: []                 # Also `git add` these files ('.' for all)
    add_known: true         # Commit with -a flag
    message: |
      Bump version to {version}

  # 'tag': Create an annotated tag
  - task: tag               #
    name: v{version}        #
    message: |              #
      Version {version}

  # 'push': Push changes and tags
  - task: push
    tags: true              # Use `--follow-tags`

  # 'pypi_release': Create a release on PyPI
  - task: pypi_release
    build:
      - sdist
      - bdist_wheel
    upload: true
    # revert_bump_on_error: true

  # 'github_release': Create a release on GitHub
  - task: github_release
    # Override `config.gh_gh_auth`:
    gh_auth: null
    name: 'v{version}'
    message: |
      Released {version}
      [Commit details](https://github.com/{repo}/compare/{org_tag_name}...{tag_name}).
    draft: true
    prerelease: null  # null: guess from version number format
    upload:
      - sdist
      - bdist_wheel

  - task: bump
    inc: "postrelease"
    prerelease_prefix: "a"

  - task: commit
    add_known: true
    message: |
      Bump prerelease ({version}) [ci skip]

```

(continues on next page)

(continued from previous page)

```
123   - task: push
```

2.5 Script Reference

2.5.1 Workflow Configuration

At the beginning of the configuration file, we define general options for all following tasks in this workflow:

```
file_version: yabs#1

config:
  repo: 'mar10/test-release-tool'
  gh_auth:
    oauth_token_var: GITHUB_OAUTH_TOKEN
  version:
    - type: __version__
      file: src/test_release_tool/__init__.py
  max_increment: minor
  branches:  # not yet implemented
    - master
```

branches (list) *Not yet implemented.* See the *branches* option of the check-task instead.

gh_auth (dict | str), *mandatory* Name of the environment variable that contains your GitHub OAuth token :

oauth_token_var (str) oauth_token_var: GITHUB_OAUTH_TOKEN

max_increment (str) Restrict the maximum bump-increment. The supported increments have this order: 'postrelease' < 'prerelease' < 'patch' < 'minor' < 'major'. For an example, a value of 'patch' will prevent 'minor' or 'major' bumps, which may be handy to prevent accidental releases from maintenance branches. Passing `--force` on the command line will allow to ignore this setting.

repo (str), *mandatory* GitHub repository name in the form 'USER/REPO', for example 'mar10/test-release-tool'.

version (dict) Define the location of the project's version number. See *Version Locations* for details.

2.5.2 Tasks

Common Task Options

All tasks share these common arguments (see also *`WorkflowTask`*):

dry_run (bool), default: *false* Run this task without really writing changes. This task-flag overrides the global mode, which is enabled using the `--dry-run` (or `-n`) argument.

verbose (int), default: *3* Set log verbosity level in the range of quiet (0) to very verbose (5). This task-flag overrides the global mode, which is incremented/decremented using the `--verbose`/`--quiet` (or `-n`/`-q`) arguments.

'build' Task

This task calls `python setup.py TARGET` to create Python builds. The artifacts are then typically used by following tasks like *pypi_release*. Technically, the files are first created in a temporary folder and then moved to the project's `/dist` folder:

```
- task: build
  targets:
    - sdist
    - bdist_wheel
```

clean (bool), default: *true* Run `python setup.py clean --all` after the builds were created on order to cleanup the build/ folder.

revert_bump_on_error (bool), default: *true* Un-patch a previously bumped version number if an error occurred while running this build task. This may make it a bit easier to recover and cleanup manually.

targets (list), default: *['sdist', 'bdist_wheel']* Valid targets are "sdist", "bdist_wheel", and "bdist_msi".

Command Line Arguments:

`--dry-run` Build artifacts to temp folder, but do not copy them to dist/.

'bump' Task

This task increments the project's version number according to SemVer by patching the respective text file. Please read *Version Locations* and After-Release Versions for details. Example: bump version according to the `--inc` command line argument:

```
- task: bump
  inc: null
```

Bump version for after-release status:

```
- task: bump
  inc: postrelease
  prerelease_prefix: "a"
  prerelease_start_idx: 1
```

check (bool), default: *true* If *true*, `python setup.py --version` is called after bumping the version and an error is raised if it does not match the expected value.

inc (str|null), default: *null* If *null*, the value that was passed as `--inc` argument on the command line is used. Otherwise the value must be one of *major*, *minor*, *patch*, *postrelease*, or *prerelease*.

prerelease_prefix (str), default: *"a"* This value is used to prefix pre- or post-release version numbers. For example if `"a"` (the default) is passed, the pre-release version for `1.2.3` could be `1.2.3-a1`.

prerelease_start_idx (int), default: *1* This value is used to prefix pre- or post-release version numbers. For example if `0` is passed, the pre-release version for `1.2.3` would be `1.2.3-a0`.

Command Line Arguments:

`--dry-run` Calculate, but do not write the new version to the target file.

`--inc` Define the SemVer increment ('postrelease', 'prerelease', 'patch', 'minor', or 'major'). This arguemnt is only considered if the task defines the `inc: null` option.

`--force` Bump version even if the max_increment rule would be violated.

`--no-bump` Skip all *bump* tasks by forcing them to dry-run mode.

'check' Task

This task will test a bunch of preconditons and stop the workflow if one or more checks fail.

```
- task: check
  branches: master          # Current branch must be in this list
  can_push: true            # Test if 'git push' would succeed
  clean: true               # Repo must/must not contain modifications
  github: true              # GitHub repo name valid and online accessible
  os: null                  # (str, list)
  python: ">=3.7"           # SemVer specifier
  twine: true               # `twine` is available
  up_to_date: true          # everything pulled from remote
  venv: true                # running inside a virtual environment
  version: true             # `setup.py --version` returns the configured version
```

branches (str | list), default: *null* Git branch name (or list of such) that are allowed. This check is typically used to prevent creating accidental releases from feature or maintenance branches.

can_push (bool), default: *null* Test if `git push --dry-run` would succeed.

clean (bool), default: *null* Test if the index or the working copy is clean, i.e. has no changes.

github (bool), default: *null* Test if the GitHub repository is accessible. This implies that

- An internet connection is up
- GitHub is reachable
- The GitHub OAuth token (*config.gh_auth.oauth_token_var* option) is valid
- The repository name (*config.repo* option) exists and is accessible

os (str | list), default: *null* Test if the return value of `platform.system()` is in the provided list. Typical values are 'Linux', 'Darwin', 'Java', 'Windows'.

python (str), default: *null* Test if the the current Python version matches the provided specification. Example `python: '>=3.7'`

repo (str), default: *(value from config.repo)* Allows to override the global setting.

twine (bool), default: *null* Test if twine is available, which is required by the *pypi_release* task.

up_to_date (bool), default: *null* Test if the remote branch contains unpulled changes, by calling `git status -uno`.

venv (bool), default: *null* Test if yabs is running inside a virtual environment.

version (bool), default: *null* Test if the result of `python setup.py --version` matches the version that yabs read from the configured version location.

Command Line Arguments:

`--no-check` Print warnings but continue workflow even if one or more checks failed.

'commit' Task

Commit modified files using `git commit`:

```
- task: commit
  add_known: true
  message: |
    Bump version to {version}
```

add (list), default: *[]* Optional list of files and patterns to add to the index.

add_known (bool), default: *true* Commit with –all option (commit all changed files).

message (str), default: ***'Bump version to {version}'*** Commit message. Context macros are expanded, e.g. '{version}', ... See *Template Macros* for details. Tip: when using Travis, a '[ci skip]' substring tells travis to ignore this commit.

Command Line Arguments:

--dry-run Pass `--dry-run` to git commands.

'exec' Task

Run a shell command using subprocess.run(), for example `tox -e lint`:

```
- task: exec
  args: ["tox", "-e", "lint"]
  always: true             # `true`: run even in dry-run mode
  silent: true             # `true`: suppress final printing of process output
  ignore_errors: false     # `true`: show warning, but proceed on errors (exit code !=␣
↪0)
  timeout: 60.0            # Kill process after <n> seconds
```

args (list), mandatory List of command line parts.

always (bool), default: ***false*** If true, this command will also be run in dry-run mode.

dry_run_args (list), default: ***null*** List of command line parts that will be used instead of the *exec.args* option when dry-run mode is active. Otherwise in dry-run mode only the command line args are printed.

ignore_errors (bool), default: ***false*** If true, error code != 0 will be ignored (yabs would stop otherwise).

log_start (bool), default: ***true*** If true, 'Running xxx...' is printed before calling the actual script.

silent (bool), default: ***false*** Controls whether the process output will be printed to the console *after* the command finished. *false*: Always print output after the command finished. *true*: Print output only when errors occured (return code != 0). NOTE: A summary line is always printed. NOTE: For long-running tasks, *streamed: true* may be a better option.

streamed (bool), default: ***null*** Poll and log output *while* the process is running. *true* enable polling (mutually exclusive with *silent: false*). *false* disable polling. *null* assume *true* if verbose mode is on.

timeout (float), default: ***null*** Kill the subprocess after *timeout* seconds.

Command Line Arguments:

--dry-run Do not execute the shell command (see also *always* and *dry_run_args* above).

'github_release' Task

Use the GitHub API to create a release from the tag and artifacts that yabs created in previous tasks:

```
- task: github_release
  name: 'v{version}'
  message: |
    Released {version}
    [Commit details](https://github.com/{repo}/compare/{org_tag_name}...{tag_name}).
  prerelease: null  # null: guess from version number format
  upload:
    - sdist
    - bdist_wheel
```

gh_auth (dict), default: ***null*** Optionally override the global *config.gh_auth* setting.

draft (bool), default: *false* *true*: create a draft (unpublished) release *false*: to create a published one. Use the `--gh-draft` argument to override.

message (str), default: *'(see example above)'* Description of the release. See also *Template Macros*.

name (str), default: *'v{version}'* The name of the new release. See also *Template Macros*.

prerelease (bool), default: *null* *false*: mark as full release. *true*: mark as pre-release, i.e. not ready for production and may be unstable. *null*: guess from version number, i.e. post-release numbers containing '-' are considered pre-releases. Use the `--gh-pre` to argument to override.

repo (str), default: *null* Optionally override the global *config.repo* setting.

target_commitish (str), default: *null* Specifies the commitish value that determines where the Git tag is created from. Can be any branch or commit SHA. Unused if the Git tag already exists. Default: the repository's default branch (usually master).

upload (list), default: *null* List of artifact names ('sdist', 'bdist_wheel', and 'bdist_msi'). Default *null*: upload all artifacts that were created in the previous build-task.

Command Line Arguments:

`--dry-run` Do not actually call the GitHub API request.

`--gh-draft` Force *github_release.draft: true*.

`--gh-pre` Force *github_release.prerelease: true*.

`--no-release` Skip this task.

Preconditions

A *tag* and *build* task must be run first.

'push' Task

Call `git push` to push changes and tags:

```
- task: push
  tags: true
```

tags (bool), default: *false* Use `--follow-tags` to push annotated tags as well.

target (str), default: '' Defines the push target. By default, the 'branch.*.remote' configuration for the current branch is consulted. If the configuration is missing, it defaults to 'origin'.

Command Line Arguments:

`--dry-run` Pass '–dry-run' option to 'git push' command.

'pypi_release' Task

Call `twine upload` create a release on PyPI from the artifacts that yabs created in previous tasks:

```
- task: pypi_release
```

upload (list), default: *null* List of artifact names ('sdist', 'bdist_wheel', and 'bdist_msi'). Default *null*: upload all artifacts that were created in the previous build-task.

Command Line Arguments:

`--dry-run` description.

--no-release Skip this task.

Preconditions

- A *tag* and *build* task must be run first.
- twine must be available.

'tag' Task

Call `git tag` to create an annotated tag:

```
- task: tag
  name: v{version}
  message: |
    Version {version}
```

message (str), default: ***'Version {version}'*** The description of the new tag. See also *Template Macros*.

name (str), default: ***'v{version}'*** The name of the new tag. See also *Template Macros*.

Command Line Arguments:

--dry-run description.

2.6 Command Line Interface

2.6.1 Basic Command

Use the `--help` or `-h` argument to get help:

```
$ yabs --help
usage: yabs [-h] [-v | -q] [-n] [--no-color] [-V]
            {run,bump,check,commit,exec,gh-release,push,pypi-release,tag} ...

Release workflow automation tools.

positional arguments:
{run,bump,check,commit,exec,gh-release,push,pypi-release,tag}
                        sub-command help
    run                 run a workflow definition
    bump                increment current project version
    check               check preconditions
    commit              increment current 'patch' version (add `--minor` or
                        `--major`)
    exec                execute shell command
    gh-release          create a release on GitHub
    push                increment current 'patch' version (add `--minor` or
                        `--major`)
    pypi-release        Make sdist, wheel, and upload on PyPI
    tag                 increment current 'patch' version (add `--minor` or
                        `--major`)

optional arguments:
-h, --help            show this help message and exit
-v, --verbose         increment verbosity by one (default: 3, range: 0..5)
```

(continues on next page)

(continued from previous page)

```
-q, --quiet           decrement verbosity by one
-n, --dry-run         just simulate and log results, but don't change
                        anything
--no-color            prevent use of ansi terminal color codes
-V, --version         display version info and exit (combine with -v for
                        more information)

See also https://github.com/mar10/yabs
$
```

2.6.2 *run* command

The main purpose of the yabs command line tool is to execute a test scenario:

```
$ yabs run --inc patch
```

See also the help:

```
$ yabs run --help
usage: yabs run [-h] [-v | -q] [-n] [--no-color]
            [--inc {major,minor,patch,postrelease}] [--no-bump]
            [--no-check] [--no-release]
            [workflow]

positional arguments:
workflow              run a workflow definition

optional arguments:
-h, --help            show this help message and exit
-v, --verbose         increment verbosity by one (default: 3, range: 0..5)
-q, --quiet           decrement verbosity by one
-n, --dry-run         just simulate and log results, but don't change
                        anything
--no-color            prevent use of ansi terminal color codes
--inc {major,minor,patch,postrelease}
                        bump semantic version (used as default for `bump`
                        tasks)
--no-bump             skip all 'bump' tasks
--no-check            don't let the 'check' task stop the workflow (log
                        warnings instead)
--no-release          skip all 'gh-release' and 'pypi-release' tasks
$
```

See the *User Guide* example for details.

2.6.3 Verbosity Level

The verbosity level can have a value from 0 to 6:

Verbosity	Option	Log level	Remarks
0	-qqq	CRITICAL	quiet
1	-qq	ERROR	
2	-q	WARN	show less info
3		INFO	show write operations
4	-v	DEBUG	show more info
5	-vv	DEBUG	
6	-vvv	DEBUG	

2.6.4 Exit Codes

The CLI returns those exit codes:

```
0: OK
1: Error (network, internal, ...)
2: CLI syntax error
3: Aborted by user
```

2.7 Writing Plugins

> **Warning:** The plugin API is still preliminary: expect changes!

Additional task types can be added to *Yabs* by the way of *plugins*.

For example let's assume we need a new task *cowsay* that is used like so:

```
- task: cowsay
  message: |
    Dear fellow cattle,
    We just released version {version}.
    (This message was brought to you by the 'yabs-cowsay' extension.)
```

and produces this output:

```
 ________________________________________
/ Dear fellow cattle,                    \
| We just released version 0.0.19-a2.    |
| (This message was brought to you by the |
\ 'yabs-cowsay' extension.)              /
 ----------------------------------------
    \   ^__^
     \  (oo)\_______
        (__)\       )\/\
            ||----w |
            ||     ||
```

This can be implemented by a separate installable Python module, that exposes a special entry point:

```
[options.entry_points]
# Plugins are found by the 'yabs.tasks' namespace.
```

(continues on next page)

(continued from previous page)

```
# The 'register()' function is then called by the plugin loader.
# The 'cowsay' name is used as yabs task type name.
yabs.tasks =
    cowsay = yabs_cowsay:register
```

See the sample implementation for implemntation details and the sample project for a usage example.

Note: Please let's reserve the namespace `yabs-TASKNAME` for 'official' extensions. If you publish your own custom extension on PyPI, choose a name like `yabs-USER-TASKNAME` or similar.

Also add 'yabs-plugin' to the keywords, to make it more discoverable.

CHAPTER 3

Reference Guide

3.1 Architecture

3.1.1 Class Overview

General Classes

Workflow Tasks

3.1.2 Context Variables

`TaskContext`

inc (str) ...

repo (str) The GitHub repository name, e.g. *"mar10/wsgidav"*.

3.2 API Reference

3.2.1 yabs

yabs package

yabs.task_runner

yabs.util module

exception yabs.util.**CheckError**
Bases: *yabs.util.ExitingYabsError*

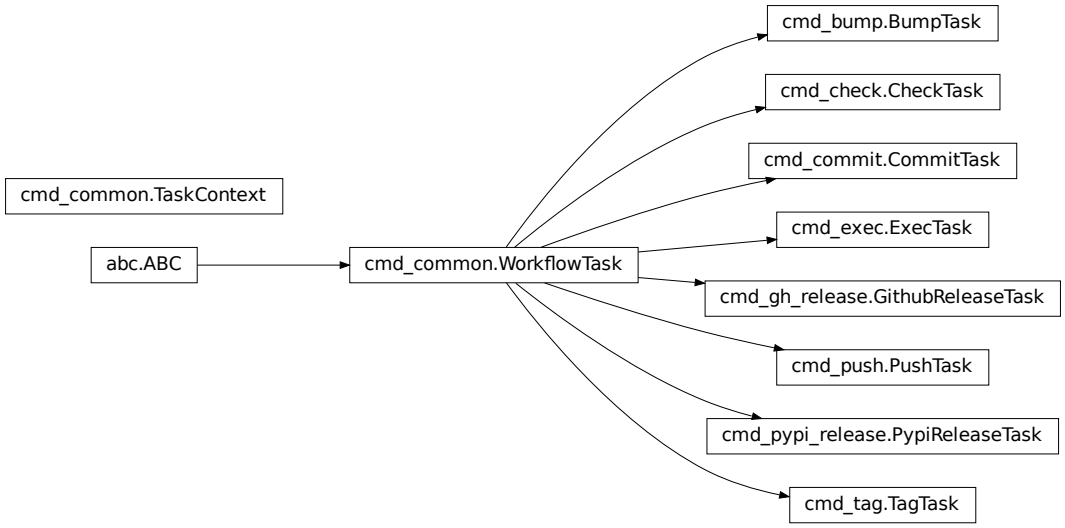


Fig. 1: Workflow Tasks

–check condition failed.

code
exception code

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception yabs.util.**ConfigError**
Bases: *yabs.util.ExitingYabsError*

Invalid yabs.yaml or command line (terninates without stacktrace).

code
exception code

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception yabs.util.**ExitingYabsError**
Bases: SystemExit, *yabs.util.YabsError*

YabsError the will cause sys.exit(). Raised for errors that don't need a stack trace.

code
exception code

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

class yabs.util.**NO_DEFAULT**
Bases: object

Used as default parameter to distinguish from *None*.

exception yabs.util.**YabsError**
Bases: RuntimeError

Base class for all exception that we deliberatly throw.

with_traceback()
Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

yabs.util.**assert_always**(*condition*, *msg=None*)
assert even in production code.

yabs.util.**check_arg**(*argument*, *allowed_types*, *condition=<class 'yabs.util.NO_DEFAULT'>*, *or_none=False*)
Check if *argument* has the expected type and value.

Note: the exception's traceback is manipulated, so that the back frame points to the `check_arg()` line, instead of the actual `raise`.

Parameters

- **argument** (*any*) – value of the argument to check
- **allowed_types** (*type or tuple of types*)
- **condition** (*bool, optional*) – additional condition that must be true
- **or_none** (*bool, optional*) – defaults to false

Returns None

Raises

- `TypeError` – if *argument* is of an unexpected type
- `ValueError` – if *argument* does not fulfill the optional condition
- `AssertionError` – if *check_arg* was called with a wrong syntax, i.e. *allowed_types* does not contain types, e.g. if *1* was passed instead of *int*.

Examples:

```
def foo(name, amount, options=None):
    check_arg(name, str)
    check_arg(amount, (int, float), amount > 0)
    check_arg(options, dict, or_none=True)
```

yabs.util.**check_cli_verbose**(*default=3*)
Check for presence of *–verbose/–quiet* or *-v/-q* without using argparse.

yabs.util.**datetime_to_iso**(*dt=None*, *microseconds=False*)
Return current UTC datetime as ISO formatted string.

yabs.util.**format_elap**(*seconds*, *count=None*, *unit='items'*, *high_prec=False*)
Return elapsed time as H:M:S.h string with reasonable precision.

yabs.util.**format_rate**(*count*, *time*, *unit=None*, *high_prec=False*)
Return count / time with reasonable precision.

yabs.util.**get_dict_attr**(*d*, *key_path*, *default=<class 'yabs.util.NO_DEFAULT'>*)
Return the value of a nested dict using dot-notation path.

Parameters

- **d** (*dict*)
- **key_path** (*str*)

Raises

- `KeyError`
- `ValueError`
- `IndexError`

Examples:

```
...
```

Todo:
- k[1] instead of k.[1]
- default arg

`yabs.util.`**`init_logging`**(*verbose=3*, *path=None*)
CLI calls this.

`yabs.util.`**`lstrip_string`**(*s*, *prefix*, *ignore_case=False*)
Remove leading string from s.

Note: This is different than *s.lstrip('bar')* which would remove all leading 'a', 'b', and 'r' chars.

`yabs.util.`**`resolve_path`**(*root*, *path*, *must_exist=True*, *check_root=False*)
Return an absolute path, assuming relative to the root file or folder.

`yabs.util.`**`run_process_streamed`**(*process*, *name=None*, *on_output=None*, *log_alive=5.0*, *timeout=None*, *poll_interval=0.1*, *flush_min_interval=0.2*, *prefix_chunks=False*)
Read and print output of a running process.

Parameters
- **process** (*subprocess.Popen*) – A running process instance
- **name** (*str*) – Descriptive name of the process
- **on_output** (*stream*) – Call this method for output (default: *logger.info*)
- **log_alive** (*float*) – Print a simple message if not new output was received for X seconds (default: 5.0)
- **timeout** (*float*) – Kill process after X seconds (default: None)
- **poll_interval** (*float*) – Poll output buffer every X seconds (default: 0.1)
- **flush_min_interval** (*float*) – Minimal log interval: Wait X seconds for more lines (default: 0.2)
- **prefix_chunks** (*bool*) – Prefix output chunks with <name> (default: False)

Returns tuple (ret_code, output)

`yabs.util.`**`set_console_ctrl_handler`**(*ctrl_handler*, *do_ctrl_c=True*, *do_ctrl_break=False*, *do_ctrl_close=False*)
The do_ctrl_* functions could simply be sys.exit(1), which will ensure that atexit handlers get called. See https://bugs.python.org/issue35935

Raises `ctypes.WinError`

Returns False if not on Windows or could not register the handler.

`yabs.util.`**`shorten_string`**(*long_string*, *max_chars*, *min_tail_chars=0*, *place_holder='[...]'*)
Return string, shortened to max_chars characters.

long_string = "This is a long string, that will be truncated." trunacated_string = truncate_string(long_string, max_chars=26, min_tail_chars=11, place_holder="[. . .]") print trunacated_string >> This is a [. . .] truncated.

@param long_string: string to be truncated @param max_chars: max chars of returned string @param min_tail_chars: minimum of tailing chars @param place_holder: place holder for striped content @return: truncated string

yabs.util.**timetag**(*seconds=True*, *ms=False*)
Return a time stamp string that can be used as (part of a) filename (also sorts well).

yabs.util.**to_list**(*obj*)
Convert a single object to a list.

yabs.log module

yabs.cmd package

yabs.cmd_common module

class yabs.cmd_common.**TaskContext**(*args*, *task_runner*)
Bases: object

Context information that is passed by the task runner to all tasks. This instance is used by tasks to pass information to downstream tasks.

args = None
CLI arguments namespace object

artifacts = None
(dict) all files that 'pypi_release' created, e.g. {"sdist": <path>, "bdist_msi": <path>}

dry_run = None
(bool) true if --dry-run was passed

gh_auth_token = None
(str) GitHub authentication token

inc = None
(str) value of --inc argument ('major', 'minor', 'patch', 'postrelease')

org_tag_name = None
(str) the repo's latest tag name (before 'bump')

org_version = None
(semantic_version.Version) latest version (before 'bump')

repo = None
(str) GitHub repo name, e.g. 'USER/PROJECT'

repo_obj = None
(git.repo.base.Repo)

repo_path = None
(str) Root folder

tag_name = None
(str) the current tag name (after 'bump')

task_runner = None
(TaskRunner)

version = None
(semantic_version.Version) current version (after 'bump')

version_manager = None
(VersionManager)

class yabs.cmd_common.**WorkflowTask**(*opts*)
Bases: abc.ABC

Common base class for all yabs tasks.

DEFAULT_OPTS = None
(dict) define all supported arguments and their default values. This attribute must be defined by derived classes.

classmethod check_task_def(*task_def*, *parser*, *args*, *yaml*)
Check task definition for errors.

This allows static pre-checks before the actual workflow starts.

Returns (str|list|bool) Error message(s)

dry_run = None
(bool) true if *–dry-run* was passed to the CLI

classmethod handle_cli_command(*parser*, *args*)
Default implementation, when run as stand-alone CLI command.

opts = None
(dict) The actual arguments, i.e. the default values merged with passed options

classmethod register_cli_command(*subparsers*, *parents*, *run_parser*)
Let tasks add a sub-command and/or arguments to the 'run' command.

verbose = None
(int, default=3) 0..5

3.3 Index

CHAPTER 4

Development

4.1 Install for Development

First off, thanks for taking the time to contribute!

This small guideline may help taking the first steps.

Happy hacking :)

4.1.1 Fork the Repository

Clone yabs to a local folder and checkout the branch you want to work on:

```
$ git clone git@github.com:mar10/yabs.git
$ cd yabs
$ git checkout my_branch
```

4.1.2 Work in a Virtual Environment

Install Python

We need Python 3.5+, and pipenv on our system.

If you want to run tests on *all* supported platforms, install Python 3.5, 3.6, 3.7, and 3.8.

Create and Activate the Virtual Environment

Install dependencies for debugging:

```
$ cd /path/to/yabs
$ pipenv shell
(yabs) $ pipenv install --dev
(yabs) $
```

The development requirements already contain the yabs source folder, so `pipenv install -e .` is not required.

The code should now run:

```
$ yabs --version
2.0.0
```

The test suite should run as well:

```
$ tox
```

Build Sphinx documentation to target: *<yabs>/docs/sphinx-build/index.html*)

```
$ tox -e docs
```

4.2 Run Tests

Run all tests with coverage report. Results are written to <yabs>/htmlcov/index.html:

```
$ tox
```

Run selective tests:

```
$ tox -e py37
$ tox -e py37 -- -k test_context_manager
```

4.3 Code

The tests also check for eslint, flake8, black, and isort standards.

Format code using the editor's formatting options or like so:

```
$ tox -e format
```

Note: Follow the Style Guide, basically PEP 8.

Failing tests or not follwing PEP 8 will break builds on travis, so run `$ tox` and `$ tox -e format` frequently and before you commit!

4.4 Create a Pull Request

Todo: TODO

CHAPTER 5

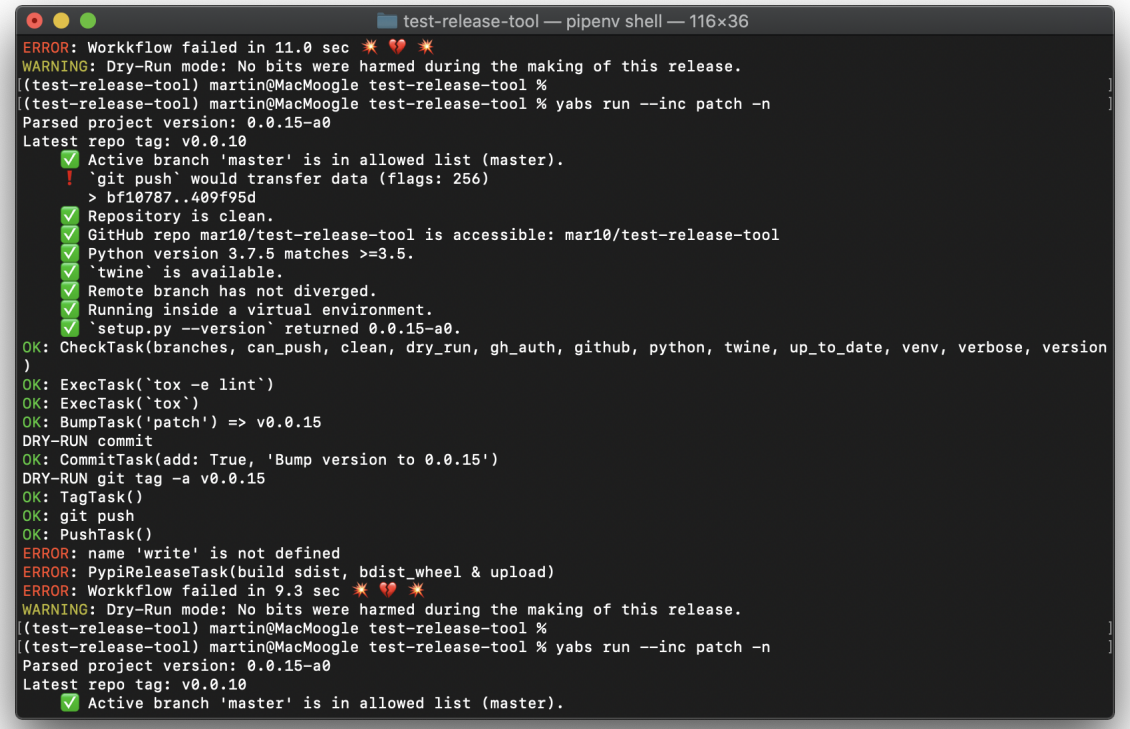
Release Info

```
# Changelog

## 1.0.0 (unreleased)

Initial release.
```

Yabs is a command line tool, that runs a sequence of tasks in order to test, build, and deliver a Python software project.


test-release-tool — pipenv shell — 116×36
ERROR: Workkflow failed in 11.0 sec 💥 💔 💥
WARNING: Dry-Run mode: No bits were harmed during the making of this release.
(test-release-tool) martin@MacMoogle test-release-tool %
(test-release-tool) martin@MacMoogle test-release-tool % yabs run --inc patch -n
Parsed project version: 0.0.15-a0
Latest repo tag: v0.0.10
✅ Active branch 'master' is in allowed list (master).
❗ `git push` would transfer data (flags: 256)
> bf10787..409f95d
✅ Repository is clean.
✅ GitHub repo mar10/test-release-tool is accessible: mar10/test-release-tool
✅ Python version 3.7.5 matches >=3.5.
✅ `twine` is available.
✅ Remote branch has not diverged.
✅ Running inside a virtual environment.
✅ `setup.py --version` returned 0.0.15-a0.
OK: CheckTask(branches, can_push, clean, dry_run, gh_auth, github, python, twine, up_to_date, venv, verbose, version
)
OK: ExecTask(`tox -e lint`)
OK: ExecTask(`tox`)
OK: BumpTask('patch') => v0.0.15
DRY-RUN commit
OK: CommitTask(add: True, 'Bump version to 0.0.15')
DRY-RUN git tag -a v0.0.15
OK: TagTask()
OK: git push
OK: PushTask()
ERROR: name 'write' is not defined
ERROR: PypiReleaseTask(build sdist, bdist_wheel & upload)
ERROR: Workkflow failed in 9.3 sec 💥 💔 💥
WARNING: Dry-Run mode: No bits were harmed during the making of this release.
(test-release-tool) martin@MacMoogle test-release-tool %
(test-release-tool) martin@MacMoogle test-release-tool % yabs run --inc patch -n
Parsed project version: 0.0.15-a0
Latest repo tag: v0.0.10
✅ Active branch 'master' is in allowed list (master).

CHAPTER 6

Features

- Define a custom workflow definition using YAML syntax.
- Check preconditions to prevent unwanted or failing releases.
- Run external tools like linters and check their return values.
- Bump version number in Python or text files.
- Create source and built distributions by running setup.py.
- Tag, commit, and push to git repository.
- Publish releases on PyPI or GitHub.
- Comes with prebuilt activities, but can be extended by custom task-plugins.
- This is a command line tool that runs on Linux, macOS, and Windows. . .
- . . . and a library for use in custom Python projects.

CHAPTER 7

Quickstart

Releases are hosted on PyPI and can be installed using pipenv (Python 3.5+ is required)

```
$ pipenv shell
(yabs) $ pipenv install yabs --upgrade
(yabs) $ yabs --version -v
(yabs) $ yabs --help
(yabs) $ yabs run --inc patch
```

Python Module Index

y

yabs.cmd_common, 29
yabs.task_runner, 25
yabs.util, 25

Index

A

args (*yabs.cmd_common.TaskContext attribute*), 29
artifacts (*yabs.cmd_common.TaskContext attribute*), 29
assert_always() (*in module yabs.util*), 27

C

check_arg() (*in module yabs.util*), 27
check_cli_verbose() (*in module yabs.util*), 27
check_task_def() (*yabs.cmd_common.WorkflowTask class method*), 30
CheckError, 25
code (*yabs.util.CheckError attribute*), 26
code (*yabs.util.ConfigError attribute*), 26
code (*yabs.util.ExitingYabsError attribute*), 26
ConfigError, 26

D

datetime_to_iso() (*in module yabs.util*), 27
DEFAULT_OPTS (*yabs.cmd_common.WorkflowTask attribute*), 30
dry_run (*yabs.cmd_common.TaskContext attribute*), 29
dry_run (*yabs.cmd_common.WorkflowTask attribute*), 30

E

ExitingYabsError, 26

F

format_elap() (*in module yabs.util*), 27
format_rate() (*in module yabs.util*), 27

G

get_dict_attr() (*in module yabs.util*), 27
gh_auth_token (*yabs.cmd_common.TaskContext attribute*), 29

H

handle_cli_command() (*yabs.cmd_common.WorkflowTask class method*), 30

I

inc (*yabs.cmd_common.TaskContext attribute*), 29
init_logging() (*in module yabs.util*), 28

L

lstrip_string() (*in module yabs.util*), 28

N

NO_DEFAULT (*class in yabs.util*), 26

O

opts (*yabs.cmd_common.WorkflowTask attribute*), 30
org_tag_name (*yabs.cmd_common.TaskContext attribute*), 29
org_version (*yabs.cmd_common.TaskContext attribute*), 29

R

register_cli_command() (*yabs.cmd_common.WorkflowTask class method*), 30
repo (*yabs.cmd_common.TaskContext attribute*), 29
repo_obj (*yabs.cmd_common.TaskContext attribute*), 29
repo_path (*yabs.cmd_common.TaskContext attribute*), 29
resolve_path() (*in module yabs.util*), 28
run_process_streamed() (*in module yabs.util*), 28

S

set_console_ctrl_handler() (*in module yabs.util*), 28
shorten_string() (*in module yabs.util*), 28

T

tag_name (*yabs.cmd_common.TaskContext attribute*), 29
task_runner (*yabs.cmd_common.TaskContext attribute*), 29
TaskContext (*class in yabs.cmd_common*), 29
timetag() (*in module yabs.util*), 29
to_list() (*in module yabs.util*), 29

V

verbose (*yabs.cmd_common.WorkflowTask attribute*), 30
version (*yabs.cmd_common.TaskContext attribute*), 29
version_manager (*yabs.cmd_common.TaskContext attribute*), 30

W

with_traceback() (*yabs.util.CheckError method*), 26
with_traceback() (*yabs.util.ConfigError method*), 26
with_traceback() (*yabs.util.ExitingYabsError method*), 26
with_traceback() (*yabs.util.YabsError method*), 27
WorkflowTask (*class in yabs.cmd_common*), 30

Y

yabs.cmd_common (*module*), 29
yabs.task_runner (*module*), 25
yabs.util (*module*), 25
YabsError, 26