pyconcurrent

Release 1.1.3

Gene C

Apr 24, 2025

CONTENTS:

CHAPTER

ONE

PYCONCURRENT

1.1 Overview

pyconcurrent is a python class that provides a simple way to do concurrent processing. It supports both asyncio and multiprocessing. The tasks to be run concurrently can either be an executable which is run as a subprocess or a python function to be called.

1.2 Key features

- Provides two classes to do the work: *ProcRunAsyncio* and *ProcRunMp*
- Results are provided by the *results* attribute in each class. This is a list of *ProcResults*; one per run.
- Documentation includes the API reference.
- pytest classes validate that all functionality works as it should.

1.3 New / Interesting

New release.

CHAPTER

TWO

GETTING STARTED

2.1 pyconcurrent module

Please see the API reference for additional details.

Here are a couple of simple examples illustrating how the module can be used.

This example uses asyncio and subprocesses to call an executable. *tasks* must be a list of *(key, arg)* pairs, 1 per task.

key is a unique identifier, used by calle, per task. arg is an additional argument for each task. Each result returned contains both the *key* and the *arg* used for that task.

This has 5 items to be run concurrently. The results are available in the *proc_run.result*, which is a list of *ProcResult* items; one per task. Since the result order is not pre-defined, each task is identifiable by it's key available in the : *result.key*.

```
# pargs can have additional arguments
pargs = ['/usr/bin/sleep']
tasks = [(1, 1), (2,7), (3,2), (4, 2), (5, 1)]

proc_run = ProcRunAsyncio(pargs, tasks, num_workers=4, timeout=30)
await proc_run.run_all()
proc_run.print_results()
```

To switch to *multiprocessing* simply replace *ProcRunAsyncio* with *ProcRunMp*, and drop *await* since MP is not *async*. i.e.

```
pargs = ['/usr/bin/sleep']
tasks = [(1, 1), (2,7), (3,2), (4, 2), (5, 1)]

proc_run = ProcRunMp(pargs, tasks, num_workers=4, timeout=30)
proc_run.run_all()
proc_run.print_results()
```

This example uses a caller supplied function with asyncio:

```
async def test_func_async(key, args) -> (bool, []):
    ''' return 2-tuple (success, result) '''
    success = True
    # pull off the last argument
    nap = args[-1]
    await asyncio.sleep(nap)
    answer = {
```

(continues on next page)

(continued from previous page)

```
            'key' : key,
            'args' : args,
            'success' : success,
            'result' : 'test_func done',
          }
    return (success, answer)

    pargs = [test_func_async, 'dummy-arg']]
    tasks = [(1, 1), (2,7), (3,2), (4, 2), (5, 1)]

    proc_run = ProcRunAsyncio(pargs, tasks, num_workers=4, timeout=30)
    await proc_run.run_all()
    proc_run.print_results()
```

For equivalent multiprocessor version then, as above, simply replace *ProcRunAsyncio* with *ProcRunMp* and drop any references to **async/await**. The user supplied function must return a 2-tuple of *(success:bool, answer:Any)* where success should be *True* if function succeeded.

The function may optionally raise an *RuntimeError* exception, but typically setting *success* is sufficient.

CHAPTER

THREE

APPENDIX

3.1 Installation

Available on

- Github
- Archlinux AUR

On Arch you can build using the provided PKGBUILD in the packaging directory or from the AUR. All git tags are signed with arch@sapience.com key which is available via WKD or download from https://www.sapience.com/tech. Add the key to your package builder gpg keyring. The key is included in the Arch package and the source= line with *?signed* at the end can be used to verify the git tag. You can also manually verify the signature

```
git tag -v <tag-name>
```

To build manually, clone the repo and :

```
rm -f dist/*
/usr/bin/python -m build --wheel --no-isolation
root_dest="/"
./scripts/do-install $root_dest
```

When running as non-root then root_dest must be a user writable directory

3.2 Dependencies

Run Time :

- python (3.13 or later)

Building Package :

- git
- hatch (aka python-hatch)
- wheel (aka python-wheel)
- build (aka python-build)
- installer (aka python-installer)
- rsync
- pytest (aka python-pytest)
- pytest-asyncio (aka python-pytest-asyncio)

Optional for building docs :

- sphinx
- myst-parser (aka python-myst-parser)
- sphinx-autoapi (aka python-sphinx-autoapi)
- texlive-latexextra (archlinux packaging of texlive tools)

3.3 Philosophy

We follow the *live at head commit* philosophy. This means we recommend using the latest commit on git master branch. We also provide git tags.

This approach is also taken by Google[1][2].

3.4 License

[1] https://github.com/google/googletest
[2] https://abseil.io/about/philosophy#upgrade-support

CHAPTER

FOUR

CHANGELOG

4.1 Tags

```
1.1.2, origin/master (2025-04-24) -> 1.1.3 (2025-04-24)
2 commits.
```

4.2 Commits

- 2025-04-24 : **1.1.3**

```
Add dateutil dep to PKGBUILD
```

- 2025-04-24 : **1.1.2, origin/master**

```
Initial Commit
```

CHAPTER

FIVE

MIT LICENSE

CHAPTER

SIX

HOW TO HELP WITH THIS PROJECT

Thank you for your interest in improving this project. This project is open-source under the MIT license.

6.1 Important resources

- Git Repo

6.2 Reporting Bugs or feature requests

Please report bugs on the issue tracker in the git repo. To make the report as useful as possible, please include

- operating system used
- version of python
- explanation of the problem or enhancement request.

6.3 Code Changes

If you make code changes, please update the documentation if it's appropriate.

CHAPTER

SEVEN

CONTRIBUTOR COVENANT CODE OF CONDUCT

7.1 Our Pledge

In the interest of fostering an open and welcoming environment, we as contributors and maintainers pledge to making participation in our project and our community a harassment-free experience for everyone, regardless of age, body size, disability, ethnicity, sex characteristics, gender identity and expression, level of experience, education, socio-economic status, nationality, personal appearance, race, religion, or sexual identity and orientation.

7.2 Our Standards

Examples of behavior that contributes to creating a positive environment include:

- Using welcoming and inclusive language
- Being respectful of differing viewpoints and experiences
- Gracefully accepting constructive criticism
- Focusing on what is best for the community
- Showing empathy towards other community members

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery and unwelcome sexual attention or advances
- Trolling, insulting/derogatory comments, and personal or political attacks
- Public or private harassment
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Other conduct which could reasonably be considered inappropriate in a professional setting

7.3 Our Responsibilities

Maintainers are responsible for clarifying the standards of acceptable behavior and are expected to take appropriate and fair corrective action in response to any instances of unacceptable behavior.

Maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

7.4 Scope

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community. Examples of representing a project or community include using an official project e-mail address, posting via an official social media account, or acting as an appointed representative at an online or offline event. Representation of a project may be further defined and clarified by project maintainers.

7.5 Enforcement

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting the project team at <arch@sapience.com>. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. The Code of Conduct Committee is obligated to maintain confidentiality with regard to the reporter of an incident. Further details of specific enforcement policies may be posted separately.

7.6 Attribution

This Code of Conduct is adapted from the Contributor Covenant, version 1.4, available at https://www.contributor-covenant.org/version/1/4/code-of-conduct.html

7.7 Interpretation

The interpretation of this document is at the discretion of the project team.

CHAPTER

EIGHT

API REFERENCE

This page contains auto-generated API reference documentation[1].

8.1 test_mp

Test :
ProcRunMp class using subprocesses

8.1.1 Module Contents

`class TestMp`

Tests ProcRunMp with and without a timeout case

`test_mp_func()`

Function test without timeout being hit

`test_mp_func_timeout()`

Function test with 1 timeout being hit

`test_mp_subprocess()`

Subprocess test without timeout being hit

`test_mp_subprocess_timeout()`

Subprocess test with timeout being hit once

8.2 pyconcurrent

Public Methods pyconcurrent

8.2.1 Submodules

pyconcurrent.proc_asyncio

Concurrent tasks using asyncio.

Module Contents

[1] Created with sphinx-autoapi

class ProcRunAsyncio(*pargs: [Any], tasks: [Any, Any], num_workers: int = 4, timeout: int = 0, verb: bool = False*)

Run concurrent processes using asyncio.

Asynio concurrent process runs. Supports program to be run as a subprocess or a function to be called. The result of each run is returned as in ProcResult class instance.

Args:

pargs ([Any]):
The first element is the command/function to be run and remainder are any additional arguments.

tasks ([(Any, Any)]):
List of task items to be run concurrently. Each task is a 2-tuple, *(key, arg)*. Key is a unique identifier for this run. arg is an additional argument to the routine when it is called. Both key and arg are saved into the result class instance returned.

num_workers (int):
Max number of processes to use. Value of 0 is unlimited and 1 will mean each is run serially one at a time.

timeout (int):
Applies to commands run as subprocesses. The maximum number of seconds allotted to each process. If not complete, then process will be killed and the result will have res.success set to *False* and res.timeout set to *True*.

verb (bool):
If set to true, some additional information is sent to stdout.

Attributes:

result (*[ProcResult]*):
A list of results, one per item run. See ProcResult for details what is provided.

Methods:

print_results()

Test tool : prints each result using the ProcResul::print()

async run_all()

Start running all the provided commands/functions concurrently.

pyconcurrent.proc_mp

Concurrent tasks using multiprocessing.

Module Contents

class ProcRunMp(*pargs: [Any], tasks: [Any, Any], num_workers: int = 4, timeout: int = 0, verb: bool = False*)

Run concurrent processes using multiprocessing.

Same calling convention as ProcRunAsyncio.

Note: func cannot be async func() - conflicts with mp starmap using async

print_results()

Test tool : prints each result using the ProcResul::print()

run_all()

Do the work

pyconcurrent.proc_result

Run process with timeout

Module Contents

`class ProcResult`(*key*, *arg*)

Result of running one of the concurrent processes.

Args:

key (Any):
Caller provided unique identifier.

arg (Any):
The additional argument used for this run.

Attributes:

time_start (float):
Unix time in seconds.

time_run (float):
Seconds taken for this item to complete.

success (bool):
True if completed successfully.

timeout (bool):
True if process failed to completed in less than timeout restriction.

key (Any):
The caller provided unique identifier.

arg (Any):
The called provided argument for this run.

returncode (int):
Return value of subprocess. Typically 0 for success.

stdout (str):
Returned stdout of subprocess.

stderr (str):
Returned stderr of subprocess.

answer (Any):
Return provided by the function.

`print`()

Testing: simple print attributes

pyconcurrent.version

Project pyconcurrent

Module Contents

`version`() → str

report version and release date

8.3 test_asyncio

Test :
ProcRunAsyncio class using subprocesses

8.3.1 Module Contents

class TestAsyncio

Tests ProcRunAsyncio with and without a timeout case

async test_asyncio_func()

Function test without timeout being hit

async test_asyncio_func_timeout()

Function test with 1 timeout being hit

async test_asyncio_subprocess()

Subprocess test without timeout being hit

async test_asyncio_subprocess_timeout()

Subprocess test with timeout being hit once

CHAPTER

NINE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

t

INDEX