pyconcurrent

Release 2.11.0

Gene C

Jan 03, 2026

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

- switch python packaging from hatch to uv.
- New function run_prog() to run external command. Strictly speaking, this has nothing to do with concurrency, but doing this robustly can be a little tricky. So it is included here.
- PEP 561: Mark module as typed. Now *mypy* run on code using this module will have the type hints.
- Asyncio now uses the recommended TaskGroup class together with the timeout() context manager. These were introduced in python 3.11. This newer approach is cleaner, more robust and ensures all tasks are appropriately cancelled in the event one task fails. It also offers superior timeout capabilities.
- Timeout now works when using a caller provided function in addition to subprocesses.

CHAPTER

TWO

GETTING STARTED

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

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.

2.1.1 Example 1a: Asyncio

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 caller, one per task. *arg* is an additional argument for each task; typically *arg* provides for whatever work that task is responsible for. Each *result* returned contains both the *key* and the *arg* used by that task, information about the success of the task as well as any outputs produced by the task. See *ProcResult* class for more detail.

This example has 5 tasks to be run concurrently, at most 4 at a time. 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*.

Listing 1: Example 1a

```
#!/usr/bin/python
"""
Example 1
"""
import asyncio
from pyconcurrent import ProcRunAsyncio


async def main():
    """
    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()
```

(continues on next page)

(continued from previous page)

```
    proc_run.print_results()


if __name__ == '__main__':
    asyncio.run(main())
```

2.1.2 Example 1b: Multiprocessing

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

Listing 2: Example 1b

```
#!/usr/bin/python
"""
Example 1b
"""
from pyconcurrent import ProcRunMp


def main():
    """
    Multiprocessing
    """
    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()


if __name__ == '__main__':
    main()
```

2.1.3 Example 2: Asnycio

The next example uses a caller supplied function together with asyncio. As in the first example, there are 5 tasks to do and the number of workers is 4, so that 4 tasks are permitted to be run simultaneously.

Listing 3: Example 2

```
#!/usr/bin/python
"""
Example 2
"""
import asyncio
from typing import Any
from pyconcurrent import ProcRunAsyncio


async def test_func_async(key, args) -> tuple[bool, dict[str, Any]]:
```

(continues on next page)

(continued from previous page)

```
    """
    return 2-tuple (success, result).
    """
    success = True
    nap_time = args[-1]     # pull off the last argument

    await asyncio.sleep(nap_time)
    answer: dict[str, Any] = {
            'key': key,
            'args': args,
            'success': success,
            'result': 'test_func done',
            }
    return (success, answer)


async def main():
    """
    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()


if __name__ == '__main__':
    asyncio.run(main())
```

For equivalent multiprocessor version for this one, same as above, simply replace *ProcRunAsyncio* with *ProcRunMp* and drop any references to **async/await**.

The caller supplied function here, *test_func_async()*, must return a 2-tuple of *(success:bool, answer:Any)* where success should be *True* if function succeeded.

The function may optionally raise a *RuntimeError* exception, but typically setting *success* is sufficient. If you are using exceptions then please use this one.

2.1.4 Example 3: Non-concurrent

This one shows a non-concurrent external program being executed.

Listing 4: Example 3

```
#!/usr/bin/python
"""
Non-Concurrent
"""
from pyconcurrent import run_prog
```

(continues on next page)

(continued from previous page)

```python
def main():
    """
    Run external program with arguments
    """
    pargs_good = ['/usr/bin/sleep', '1']
    pargs_bad = ['/usr/bin/false']

    for pargs in [pargs_good] + [pargs_bad]:
        print(f'Testing: {pargs}:')
        (ret, stdout, stderr) = run_prog(pargs)

        if ret == 0:
            print('\tAll well')
            print(stdout)
        else:
            print('\tFailed')
            print(stderr)


if __name__ == '__main__':
    main()
```

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
- uv
- uv_build (aka python-uv-build)
- 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 as recommended by Google's Abseil team[1]. This means we recommend using the latest commit on git master branch.

3.4 License

Created by Gene C. and licensed under the terms of the GPL-2.0-or-later license.

- SPDX-License-Identifier: GPL-2.0-or-later
- SPDX-FileCopyrightText: © 2025-present Gene C <arch@sapience.com>

[1] https://abseil.io/about/philosophy#upgrade-support

CHAPTER

FOUR

LICENSE

pyconcurrent is a python module that provides ability to easily take advantage of running tasks concurrently.

CHAPTER

FIVE

API REFERENCE

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

5.1 pyconcurrent

Public Methods. pyconcurrent.

5.1.1 Submodules

pyconcurrent.proc_asyncio

Concurrent tasks using asyncio.

Module Contents

`class ProcRunAsyncio`(*pargs: list[Any]*, *tasks_todo: list[tuple[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_todo ([(Any, Any)]):
list of task items to be run concurrently. Each task is a 2-tuple, *(key, arg)*.

- key is a unique identifier, converable to string via str(key)
- 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):
The maximum number of seconds allotted to each process. If not complete within "timeout", then process/function will be cancelled/killed and the "result" instance will include:

- res.success set to *False*

[1] Created with sphinx-autoapi

- res.timeout set to *True*.

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

Attributes:

result (*[ProcResult]*):
list of results, one per task. See ProcResult for more detail.

Methods:

`print_results()`

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

`async run_all()`

Start running all the provided commands/functions concurrently.

Awaitable, so caller is responsible for calling asyncio.run(). See run_all_start_asyncio() for non-awaitable version.

pyconcurrent.proc_mp

Concurrent tasks using multiprocessing.

Module Contents

`class ProcRunMp`(*pargs: list[Any]*, *tasks_todo: list[tuple[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

Result class which ProcRunMp/ProcRunAsyncio use.

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 failed to complete 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.run_prog

External program execution

Module Contents

`run_prog`(*pargs: list[str]*, *input_str: str | None = None*, *stdout: int = subprocess.PIPE*, *stderr: int = subprocess.PIPE*, *env: dict[str, str] | None = None*, *test: bool = False*, *verb: bool = False*) → tuple[int, str, str]

Run external program using subprocess.

Take care to handle large outputs (default buffer size is 8k). This avoids possible hangs should IO buffer fill up.

non-blocking IO together with select() loop provides a robust methodology.

Args:

pargs (list[str]):
The command + arguments to be run in standard list format. e.g. ['/usr/bin/sleep', '22'].

input_str (str | None):
Optional input to be fed to subprocess stdin. Defaults to None.

stdout (int):
Subprocess stdout. Defaults to subprocess.PIPE

stderr (int):
Subprocess stderr. Defaults to subprocess.PIPE

env (None | dict[str, str]):
Optional to specify environment for subprocess to use. If not set, inherits from calling process as usual.

test (bool):
Flag - if true dont actually run anything.

verb (bool):
Flag - only used with test == True - prints pargs.

Returns:

tuple[retc: int, stdout: str, stderr: str]:
retc is 0 when all is well. stdout is what the subprocess returns on it's stdout and stderr is what it's stderr return.

Note that any input string is written in it's entirety in one shot to the subprocess. This should not be a problem.

pyconcurrent.version

Project pyconcurrent.

Module Contents

`version()` → str
report version and release date

PYTHON MODULE INDEX

INDEX