paco

Release 0.1.7

December 23, 2016

Contents

Small and idiomatic utility library for coroutine-driven, asynchronous-oriented generic programming in Python +3.4.

Built on top of asyncio, paco provides missing capabilities from Python *stdlib* to write asynchronous cooperative multitasking in a nice-ish way, plus higher-order function goodness.

paco can be your utility belt to deal with I/O-bound non-blocking concurrent code in a bette and elegant way.

CHAPTER 1

Features

- Simple and idiomatic API, extending Python `stdlib` with async coroutines gotchas.
- Built-in configurable control-flow concurrency support.
- Useful iterables, decorators, functors and convenient helpers.
- Coroutine-based functional helpers: compose, throttle, partial, timeout, times, until, race...
- Asynchronous coroutine port of Python built-in functions: *filter*, *map*, *dropwhile*, *filterfalse*, *reduce*...
- Concurrent iterables and higher-order functions.
- Better `asyncio.gather()` and `asyncio.wait()` with optional concurrency control and ordered results.
- Works with both async/await and yield from coroutines syntax.
- Reliable coroutine timeout handler context manager.
- Designed for intensive I/O-bound concurrent non-blocking tasks.
- Good interoperability with `asyncio` and Python `stdlib` functions.
- Composable pipelines of functors for iterables via `|` operator overloading.
- Small and dependency free.
- Compatible with Python +3.4.

CHAPTER 2

Installation

Using `pip` package manager:

```
pip install --upgrade paco
```

Or install the latest sources from Github:

```
pip install -e git+git://github.com/h2non/paco.git#egg=paco
```

CHAPTER 3

API

- paco.times
- paco.until
- paco.wait
- paco.whilst
- paco.wraps

3.1 Examples

Asynchronously and concurrently execute multiple HTTP requests.

```
import paco
import aiohttp

async def fetch(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as res:
            return res

async def fetch_urls():
    urls = [
        'https://www.google.com',
        'https://www.yahoo.com',
        'https://www.bing.com',
        'https://www.baidu.com',
        'https://duckduckgo.com',
    ]

    # Map concurrent executor with concurrent limit of 3
    responses = await paco.map(fetch, urls, limit=3)

    for res in responses:
        print('Status:', res.status)

# Run in event loop
paco.run(fetch_urls())
```

Concurrent pipeline-style composition of functors over an iterable object.

```
import paco

async def filterer(x):
    return x < 8

async def mapper(x):
    return x * 2

async def drop(x):
    return x < 10

async def reducer(acc, x):
    return acc + x

async def task(numbers):
    return await (numbers
```

```
                    | paco.filter(filterer)
                    | paco.map(mapper)
                    | paco.dropwhile(drop)
                    | paco.reduce(reducer, initializer=0))

# Run in event loop
number = paco.run(task((1, 2, 3, 4, 5, 6, 7, 8, 9, 10)))
print('Number:', number) # => 36
```

CHAPTER 4

License

MIT - Tomas Aparicio

CHAPTER 5

Contents

5.1 Examples

5.1.1 Asynchronously and concurrently execute multiple HTTP requests.

```
import paco
import aiohttp

async def fetch(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as res:
            return res

async def fetch_urls():
    urls = [
        'https://www.google.com',
        'https://www.yahoo.com',
        'https://www.bing.com',
        'https://www.baidu.com',
        'https://duckduckgo.com',
    ]

    # Map concurrent executor with concurrent limit of 3
    responses = await paco.map(fetch, urls, limit=3)

    for res in responses:
        print('Status:', res.status)

# Run in event loop
paco.run(fetch_urls())
```

5.1.2 Concurrent pipeline-style chain composition of functors over any iterable object.

```
import paco

async def filterer(x):
    return x < 8

async def mapper(x):
```

```
    return x * 2

async def drop(x):
    return x < 10

async def reducer(acc, x):
    return acc + x

async def task(numbers):
    return await (numbers
                  | paco.filter(filterer)
                  | paco.map(mapper)
                  | paco.dropwhile(drop)
                  | paco.reduce(reducer, initializer=0))

# Run in event loop
number = paco.run(task((1, 2, 3, 4, 5, 6, 7, 8, 9, 10)))
print('Number:', number) # => 36
```

5.2 API documentation

class paco.**ConcurrentExecutor**(*limit=10*, *loop=None*, *coros=None*, *ignore_empty=False*)

Bases: object

Concurrent executes a set of asynchronous coroutines with a simple throttle concurrency configurable concurrency limit.

Provides an observer pub/sub interface, allowing API consumers to subscribe normal functions or coroutines to certain events that happen internally.

ConcurrentExecutor is a low-level implementation that powers most of the utility functions provided in *paco*.

For most cases you won't need to rely on it, instead you can use the high-level API functions that provides a simpler abstraction for the majority of the use cases.

This class is not thread safe.

Events:

- start (executor): triggered before executor cycle starts.
- finish (executor): triggered when all the coroutine finished.
- task.start (task): triggered before coroutine starts.
- task.finish (task, result): triggered when the coroutine finished.

Parameters

- **limit** (*int*) – concurrency limit. Defaults to 10.
- **coros** (*list[coroutine], optional*) – list of coroutines to schedule.
- **loop** (*asyncio.BaseEventLoop, optional*) – loop to run. Defaults to asyncio.get_event_loop().
- **ignore_empty** (*bool, optional*) – do not raise an exception if there are no coroutines to schedule are empty.

Returns ConcurrentExecutor

Usage:

```
async def sum(x, y):
    return x + y

pool = paco.ConcurrentExecutor(limit=2)
pool.add(sum, 1, 2)
pool.add(sum, None, 'str')

done, pending = await pool.run(return_exceptions=True)
[task.result() for task in done]
# => [3, TypeError("unsupported operand type(s) for +: 'NoneType' and 'str'")]  # noqa
```

__len__()
Returns the current length of the coroutines pool queue.

Returns current coroutines pool length.

Return type int

add(*coro*, **args*, ***kw*)
Adds a new coroutine function with optional variadic argumetns.

Parameters

- **coro** (*coroutine function*) – coroutine to execute.
- ***args** (*mixed*) – optional variadic arguments

Raises TypeError – if the coro object is not a valid coroutine

Returns coroutine wrapped future

Return type future

cancel()
Tries to gracefully cancel the pending coroutine scheduled coroutine tasks.

extend(**coros*)
Add multiple coroutines to the executor pool.

Raises TypeError – if the coro object is not a valid coroutine

is_running()
Checks the executor running state.

Returns True if the executur is running, otherwise False.

Return type bool

off(*event*)
Removes event subscribers.

Parameters **event** (*str*) – event name to remove observers.

on(*event*, *fn*)
Subscribes to a specific event.

Parameters

- **event** (*str*) – event name to subcribe.
- **fn** (*function*) – function to trigger.

reset()
Resets the executer scheduler internal state.

Raises `RuntimeError` – is the executor is still running.

run(*timeout=None*, *return_exceptions=None*, *return_when='ALL_COMPLETED'*, *ignore_empty=None*)
Executes the registered coroutines in the executor queue.

Parameters

- **timeout** (*int/float*) – max execution timeout. No limit by default.
- **return_exceptions** (*bool*) – in case of coroutine exception.
- **return_when** (*str*) – sets when coroutine should be resolved. See asyncio.wait for supported values.
- **ignore_empty** (*bool, optional*) – do not raise an exception if there are no coroutines to schedule are empty.

Returns **asyncio.Future** – two sets of Futures: `(done, pending)`

Return type tuple

Raises

- `ValueError` – if there is no coroutines to schedule.
- `RuntimeError` – if executor is still running.
- `TimeoutError` – if execution takes more than expected.

submit(*coro*, **args*, ***kw*)
Adds a new coroutine function with optional variadic argumetns.

Parameters

- **coro** (*coroutine function*) – coroutine to execute.
- ***args** (*mixed*) – optional variadic arguments

Raises `TypeError` – if the coro object is not a valid coroutine

Returns coroutine wrapped future

Return type future

wait(*timeout=None*, *return_exceptions=None*, *return_when='ALL_COMPLETED'*, *ignore_empty=None*)
Executes the registered coroutines in the executor queue.

Parameters

- **timeout** (*int/float*) – max execution timeout. No limit by default.
- **return_exceptions** (*bool*) – in case of coroutine exception.
- **return_when** (*str*) – sets when coroutine should be resolved. See asyncio.wait for supported values.
- **ignore_empty** (*bool, optional*) – do not raise an exception if there are no coroutines to schedule are empty.

Returns **asyncio.Future** – two sets of Futures: `(done, pending)`

Return type tuple

Raises

- `ValueError` – if there is no coroutines to schedule.

- `RuntimeError` – if executor is still running.
- `TimeoutError` – if execution takes more than expected.

`paco.`**`apply`**(*coro*, **args*, ***kw*)

Creates a continuation coroutine function with some arguments already applied.

Useful as a shorthand when combined with other control flow functions. Any arguments passed to the returned function are added to the arguments originally passed to apply.

This is similar to *paco.partial()*.

This function can be used as decorator.

Parameters

- **`coro`** (*`coroutinefunction`*) – coroutine function to wrap.
- **`*args`** (*`mixed`*) – mixed variadic arguments for partial application.
- **`*kwargs`** (*`mixed`*) – mixed variadic keyword arguments for partial application.

Raises `TypeError` – if coro argument is not a coroutine function.

Returns wrapped coroutine function.

Return type coroutinefunction

Usage:

```
async def hello(name, mark='!'):
    print('Hello, {name}{mark}'.format(name=name, mark=mark))

hello_mike = paco.apply(hello, 'Mike')
await hello_mike()
# => Hello, Mike!

hello_mike = paco.apply(hello, 'Mike', mark='?')
await hello_mike()
# => Hello, Mike?
```

`paco.`**`compose`**(**coros*)

Creates a coroutine function based on the composition of the passed coroutine functions.

Each function consumes the yielded result of the coroutine that follows.

Composing coroutine functions f(), g(), and h() would produce the result of f(g(h())).

Parameters **`*coros`** (*`coroutinefunction`*) – variadic coroutine functions to compose.

Raises `RuntimeError` – if cannot execute a coroutine function.

Returns coroutinefunction

Usage:

```
async def sum_1(num):
    return num + 1

async def mul_2(num):
    return num * 2

coro = paco.compose(sum_1, mul_2, sum_1)
await coro(2)
# => 7
```

paco.**concurrent**
alias of *ConcurrentExecutor*

paco.**constant**(*value*, *delay=None*)
Returns a coroutine function that when called, always returns the provided value.

Parameters

- **value** (*mixed*) – value to constantly return when coroutine is called.
- **delay** (*int/float*) – optional return value delay in seconds.

Returns coroutinefunction

Usage:

```
coro = paco.constant('foo')

await coro()
# => 'foo'
await coro()
# => 'foo'
```

paco.**curry**(*arity_or_fn=None*, *ignore_kwargs=False*, *evaluator=None*, **args*, ***kw*)
Creates a function that accepts one or more arguments of a function and either invokes func returning its result if at least arity number of arguments have been provided, or returns a function that accepts the remaining function arguments until the function arity is satisfied.

This function is overloaded: you can pass a function or coroutine function as first argument or an *int* indicating the explicit function arity.

Function arity can be inferred via function signature or explicitly passed via *arity_or_fn* param.

You can optionally ignore keyword based arguments as well passsing the *ignore_kwargs* param with *True* value.

This function can be used as decorator.

Parameters

- **arity_or_fn** (*int|function|coroutinefunction*) – function arity to curry or function to curry.
- **ignore_kwargs** (*bool*) – ignore keyword arguments as arity to satisfy during curry.
- **evaluator** (*function*) – use a custom arity evaluator function.
- ***args** (*mixed*) – mixed variadic arguments for partial function application.
- ***kwargs** (*mixed*) – keyword variadic arguments for partial function application.

Raises TypeError – if function is not a function or a coroutine function.

Returns function will be returned until all the function arity is satisfied, where a coroutine function will be returned instead.

Return type function or coroutinefunction

Usage:

```
# Function signature inferred function arity
@paco.curry
async def task(x, y, z=0):
    return x * y + z

await task(4)(4)(z=8)
# => 24
```

```
    # User defined function arity
    @paco.curry(4)
    async def task(x, y, *args, **kw):
        return x * y + args[0] * args[1]

    await task(4)(4)(8)(8)
    # => 80

    # Ignore keyword arguments from arity
    @paco.curry(ignore_kwargs=True)
    async def task(x, y, z=0):
        return x * y

    await task(4)(4)
    # => 16
```

paco.**defer**(*coro*, *delay=1*)

Returns a coroutine function wrapper that will defer the given coroutine execution for a certain amount of seconds in a non-blocking way.

This function can be used as decorator.

Parameters

- **coro** (*coroutinefunction*) – coroutine function to defer.
- **delay** (*int/float*) – number of seconds to defer execution.

Raises TypeError – if coro argument is not a coroutine function.

Returns **filtered values** – ordered list of resultant values.

Return type list

Usage:

```
    # Usage as function
    await paco.defer(coro, delay=1)
    await paco.defer(coro, delay=0.5)

    # Usage as decorator
    @paco.defer(delay=1)
    async def mul_2(num):
        return num * 2

    await mul_2(2)
    # => 4
```

paco.**dropwhile**(*coro*, *iterable*, *loop=None*)

Make an iterator that drops elements from the iterable as long as the predicate is true; afterwards, returns every element.

Note, the iterator does not produce any output until the predicate first becomes false, so it may have a lengthy start-up time.

This function is pretty much equivalent to Python standard *itertools.dropwhile()*, but designed to be used with async coroutines.

This function is a coroutine.

This function can be composed in a pipeline chain with | operator.

Parameters

- **coro** (*coroutine function*) – coroutine function to call with values to reduce.
- **iterable** (*iterable*) – an iterable collection yielding coroutines functions.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.

Raises TypeError – if coro argument is not a coroutine function.

Returns **filtered values** – ordered list of resultant values.

Return type list

Usage:

```
async def filter(num):
    return num < 4

await paco.dropwhile(filter, [1, 2, 3, 4, 5, 1])
# => [4, 5, 1]
```

paco.**each**(*coro*, *iterable*, *limit=0*, *loop=None*, *collect=False*, *timeout=None*, *return_exceptions=False*, **args*, ***kw*)

Concurrently iterates values yielded from an iterable, passing them to an asynchronous coroutine.

You can optionally collect yielded values passing collect=True param, which would be equivalent to *paco.map()*‘.

Mapped values will be returned as an ordered list. Items order is preserved based on origin iterable order.

Concurrency level can be configurable via *limit* param.

All coroutines will be executed in the same loop.

This function is a coroutine.

This function can be composed in a pipeline chain with | operator.

Parameters

- **coro** (*coroutinefunction*) – coroutine iterator function that accepts iterable values.
- **iterable** (*iter*) – an iterable collection yielding coroutines functions. Asynchronous iterables are not supported.
- **limit** (*int*) – max iteration concurrency limit. Use 0 for no limit.
- **collect** (*bool*) – return yielded values from coroutines. Default False.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.
- **return_exceptions** (*bool*) – enable/disable returning exceptions in case of error. *collect* param must be True.
- **timeout** (*int* | *float*) – timeout can be used to control the maximum number of seconds to wait before returning. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.
- ***args** (*mixed*) – optional variadic arguments to pass to the coroutine iterable function.

Returns **results** – ordered list of values yielded by coroutines

Return type list

Raises TypeError – in case of invalid input arguments.

Usage:

```
    async def mul_2(num):
        return num * 2

    await paco.each(mul_2, [1, 2, 3, 4, 5])
    # => None

    await paco.each(mul_2, [1, 2, 3, 4, 5], collect=True)
    # => [2, 4, 6, 8, 10]
```

paco.**every**(*coro*, *iterable*, *limit=1*, *loop=None*)

Returns *True* if every element in a given iterable satisfies the coroutine asynchronous test.

If any iteratee coroutine call returns *False*, the process is inmediately stopped, and *False* will be returned.

You can increase the concurrency limit for a fast race condition scenario.

This function is a coroutine.

This function can be composed in a pipeline chain with `|` operator.

Parameters

- **coro** (*coroutine function*) – coroutine function to call with values to reduce.
- **iterable** (*iterable*) – an iterable collection yielding coroutines functions.
- **limit** (*int*) – max concurrency execution limit. Use `0` for no limit.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.

Raises `TypeError` – if input arguments are not valid.

Returns *True* if all the values passes the test, otherwise *False*.

Return type bool

Usage:

```
    async def gt_10(num):
        return num > 10

    await paco.every(gt_10, [1, 2, 3, 11])
    # => False

    await paco.every(gt_10, [11, 12, 13])
    # => True
```

paco.**filter**(*coro*, *iterable*, *assert_fn=None*, *limit=0*, *loop=None*)

Returns a list of all the values in coll which pass an asynchronous truth test coroutine.

Operations are executed concurrently by default, but results will be in order.

You can configure the concurrency via *limit* param.

This function is the asynchronous equivalent port Python built-in *filter()* function.

This function is a coroutine.

This function can be composed in a pipeline chain with `|` operator.

Parameters

- **coro** (*coroutine function*) – coroutine filter function to call accepting iterable values.
- **iterable** (*iterable*) – an iterable collection yielding coroutines functions.

- **assert_fn** (*coroutinefunction*) – optional assertion function.
- **limit** (*int*) – max filtering concurrency limit. Use `0` for no limit.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.

Raises `TypeError` – if coro argument is not a coroutine function.

Returns ordered list containing values that passed the filter.

Return type list

Usage:

```
async def iseven(num):
    return num % 2 == 0

async def assert_false(el):
    return not el

await paco.filter(iseven, [1, 2, 3, 4, 5])
# => [2, 4]

await paco.filter(iseven, [1, 2, 3, 4, 5], assert_fn=assert_false)
# => [1, 3, 5]
```

`paco.`**filterfalse**(*coro*, *iterable*, *limit=0*, *loop=None*)

Returns a list of all the values in coll which pass an asynchronous truth test coroutine.

Operations are executed concurrently by default, but results will be in order.

You can configure the concurrency via *limit* param.

This function is the asynchronous equivalent port Python built-in *filterfalse()* function.

This function is a coroutine.

This function can be composed in a pipeline chain with `|` operator.

Parameters

- **coro** (*coroutine function*) – coroutine filter function to call accepting iterable values.
- **iterable** (*iterable*) – an iterable collection yielding coroutines functions.
- **assert_fn** (*coroutinefunction*) – optional assertion function.
- **limit** (*int*) – max filtering concurrency limit. Use `0` for no limit.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.

Raises `TypeError` – if coro argument is not a coroutine function.

Returns

filtered values – ordered list containing values that do not

passed the filter.

Return type list

Usage:

```
async def iseven(num):
    return num % 2 == 0
```

```
    await paco.filterfalse(coro, [1, 2, 3, 4, 5])
    # => [1, 3, 5]
```

paco.**flat_map**(*coro*, *iterable*, *limit=0*, *loop=None*, *timeout=None*, *return_exceptions=False*, *initializer=None*, **args*, ***kw*)

Concurrently iterates values yielded from an iterable, passing them to an asynchronous coroutine.

This function is the flatten version of to `paco.map()`.

Mapped values will be returned as an ordered list. Items order is preserved based on origin iterable order.

Concurrency level can be configurable via `limit` param.

All coroutines will be executed in the same loop.

This function is a coroutine.

This function can be composed in a pipeline chain with `|` operator.

Parameters

- **coro** (*coroutinefunction*) – coroutine iterator function that accepts iterable values.
- **iterable** (*iter*) – an iterable collection yielding coroutines functions. Asynchronous iterables are not supported.
- **limit** (*int*) – max iteration concurrency limit. Use `0` for no limit.
- **collect** (*bool*) – return yielded values from coroutines. Default False.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.
- **return_exceptions** (*bool*) – enable/disable returning exceptions in case of error. *collect* param must be True.
- **timeout** (*int|float*) – timeout can be used to control the maximum number of seconds to wait before returning. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.
- ***args** (*mixed*) – optional variadic arguments to pass to the coroutine iterable function.

Returns **results** – ordered list of values yielded by coroutines

Return type list

Raises `TypeError` – in case of invalid input arguments.

Usage:

```
    async def mul_2(num):
        return num * 2

    await paco.flat_map(mul_2, [1, [2], [3, [4]], [(5,)]])
    # => [2, 4, 6, 8, 10]

    # Pipeline style
    await [1, [2], [3, [4]], [(5,)]] | paco.flat_map(mul_2)
    # => [2, 4, 6, 8, 10]
```

paco.**gather**(**coros_or_futures*, *limit=0*, *loop=None*, *timeout=None*, *preserve_order=False*, *return_exceptions=False*)

Return a future aggregating results from the given coroutine objects with a concurrency execution limit.

If all the tasks are done successfully, the returned future's result is the list of results (in the order of the original sequence, not necessarily the order of results arrival).

If return_exceptions is *True*, exceptions in the tasks are treated the same as successful results, and gathered in the result list; otherwise, the first raised exception will be immediately propagated to the returned future.

All futures must share the same event loop.

This functions is mostly compatible with Python standard `asyncio.gather`, but providing ordered results and concurrency control flow.

This function is a coroutine.

Parameters

- ***coros_or_futures** (*coroutines|list*) – an iterable collection yielding coroutines functions or futures.
- **limit** (*int*) – max concurrency limit. Use `0` for no limit.
- **can be used to control the maximum number** (`timeout`) – of seconds to wait before returning. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.
- **preserve_order** (*bool*) – preserves results order.
- **return_exceptions** (*bool*) – returns exceptions as valid results.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.

Returns coroutines returned results.

Return type list

Usage:

```
async def sum(x, y):
    return x + y

await paco.gather(
    sum(1, 2),
    sum(None, 'str'),
    return_exceptions=True)
# => [3, TypeError("unsupported operand type(s) for +: 'NoneType' and 'str'")]  # noqa
```

`paco.`**`map`**(*coro*, *iterable*, *limit=0*, *loop=None*, *timeout=None*, *return_exceptions=False*, **args*, ***kw*)

Concurrently maps values yielded from an iterable, passing then into an asynchronous coroutine function.

Mapped values will be returned as list. Items order will be preserved based on origin iterable order.

Concurrency level can be configurable via `limit` param.

This function is the asynchronous equivalent port Python built-in *map()* function.

This function is a coroutine.

This function can be composed in a pipeline chain with `|` operator.

Parameters

- **coro** (*coroutinefunction*) – map coroutine function to use.
- **iterable** (*iter*) – an iterable collection yielding coroutines functions. Asynchronous iterables are not supported.
- **limit** (*int*) – max concurrency limit. Use `0` for no limit.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.

- **timeout** (*int | float*) – timeout can be used to control the maximum number of seconds to wait before returning. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.
- **return_exceptions** (*bool*) – returns exceptions as valid results.
- ***args** (*mixed*) – optional variadic arguments to be passed to the coroutine map function.

Returns ordered list of values yielded by coroutines

Return type list

Usage:

```
async def mul_2(num):
    return num * 2

await paco.map(mul_2, [1, 2, 3, 4, 5])
# => [2, 4, 6, 8, 10]
```

paco.**once**(*coro*, *raise_exception=False*, *return_value=None*)

Wrap a given coroutine function that is restricted to one execution.

Repeated calls to the coroutine function will return the value of the first invocation.

This function can be used as decorator.

Parameters

- **coro** (*coroutinefunction*) – coroutine function to wrap.
- **raise_exception** (*bool*) – raise exception if execution times exceeded.
- **return_value** (*mixed*) – value to return when execution times exceeded, instead of the memoized one from last invocation.

Raises TypeError – if coro argument is not a coroutine function.

Returns coroutinefunction

Usage:

```
async def mul_2(num):
    return num * 2

once = paco.once(mul_2)
await once(2)
# => 4
await once(3)
# => 4

once = paco.once(mul_2, return_value='exceeded')
await once(2)
# => 4
await once(3)
# => 'exceeded'
```

paco.**partial**(*coro*, **args*, ***kw*)

Partial function implementation designed for coroutines, allowing variadic input arguments.

This function can be used as decorator.

Parameters

- **coro** (*coroutinefunction*) – coroutine function to wrap.

- ***args** (*mixed*) – mixed variadic arguments for partial application.

Raises TypeError – if coro is not a coroutine function.

Returns coroutinefunction

Usage:

```
async def pow(x, y):
    return x ** y

pow_2 = paco.partial(pow, 2)
await pow_2(4)
# => 16
```

paco.**race**(*iterable*, *loop=None*, *timeout=None*, **args*, ***kw*)

Runs coroutines from a given iterable concurrently without waiting until the previous one has completed.

Once any of the tasks completes, the main coroutine is immediately resolved, yielding the first resolved value.

All coroutines will be executed in the same loop.

This function is a coroutine.

Parameters

- **iterable** (*iterable*) – an iterable collection yielding coroutines functions or coroutine objects.
- ***args** (*mixed*) – mixed variadic arguments to pass to coroutines.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.
- **timeout** (*int|float*) – timeout can be used to control the maximum number of seconds to wait before returning. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.
- ***args** – optional variadic argument to pass to coroutine function, if provided.

Raises

- TypeError – if iterable argument is not iterable.
- asyncio.TimoutError – if wait timeout is exceeded.

Returns **filtered values** – ordered list of resultant values.

Return type list

Usage:

```
async def coro1():
    await asyncio.sleep(2)
    return 1

async def coro2():
    return 2

async def coro3():
    await asyncio.sleep(1)
    return 3

await paco.race([coro1, coro2, coro3])
# => 2
```

paco.**reduce**(*coro*, *iterable*, *right=False*, *initializer=None*, *limit=1*, *loop=None*)

Apply function of two arguments cumulatively to the items of sequence, from left to right, so as to reduce the sequence to a single value.

Reduction will be executed sequentially without concurrency, so passed values would be in order.

This function is the asynchronous coroutine equivalent to Python standard *functools.reduce()* function.

This function is a coroutine.

This function can be composed in a pipeline chain with `|` operator.

Parameters

- **coro** (*coroutine function*) – coroutine function to call with values to reduce.
- **iterable** (*iterable*) – an iterable collection yielding coroutines functions.
- **right** (*bool*) – reduce iterable from right to left.
- **initializer** (*mixed*) – initial accumulator value used in the first reduction call.
- **limit** (*int*) – max iteration concurrency limit. Use `0` for no limit.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.

Raises `TypeError` – if input arguments are not valid.

Returns accumulated final reduced value.

Return type mixed

Usage:

```
async def reducer(acc, num):
   return acc + num

await paco.reduce(reducer, [1, 2, 3, 4, 5], initializer=0)
# => 15
```

paco.**repeat**(*coro*, *times=1*, *step=1*, *limit=1*, *loop=None*)

Executes the coroutine function `x` number of times, and accumulates results in order as you would use with `map`.

Execution concurrency is configurable using `limit` param.

This function is a coroutine.

Parameters

- **coro** (*coroutinefunction*) – coroutine function to schedule.
- **times** (*int*) – number of times to execute the coroutine.
- **step** (*int*) – increment iteration step, as with `range()`.
- **limit** (*int*) – concurrency execution limit. Defaults to 10.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.

Raises `TypeError` – if coro is not a coroutine function.

Returns accumulated yielded values returned by coroutine.

Return type list

Usage:

```
async def mul_2(num):
    return num * 2

await paco.repeat(mul_2, times=5)
# => [2, 4, 6, 8, 10]
```

paco.**run**(*coro*, *loop=None*)

Convenient shortcut alias to `loop.run_until_complete`.

Parameters

- **coro** (*coroutine*) – coroutine object to schedule.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use. Defaults to: `asyncio.get_event_loop()`.

Returns returned value by coroutine.

Return type mixed

Usage:

```
async def mul_2(num):
    return num * 2

paco.run(mul_2(4))
# => 8
```

paco.**series**(**coros_or_futures*, *timeout=None*, *loop=None*, *return_exceptions=False*)

Run the given coroutine functions in series, each one running once the previous execution has completed.

If any coroutines raises an exception, no more coroutines are executed. Otherwise, the coroutines returned values will be returned as *list*.

`timeout` can be used to control the maximum number of seconds to wait before returning. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.

If `return_exceptions` is True, exceptions in the tasks are treated the same as successful results, and gathered in the result list; otherwise, the first raised exception will be immediately propagated to the returned future.

All futures must share the same event loop.

This functions is basically the sequential execution version of `asyncio.gather()`. Interface compatible with `asyncio.gather()`.

This function is a coroutine.

Parameters

- ***coros_or_futures** (*iter|list*) – an iterable collection yielding coroutines functions.
- **timeout** (*int/float*) – maximum number of seconds to wait before returning.
- **return_exceptions** (*bool*) – exceptions in the tasks are treated the same as successful results, instead of raising them.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.
- ***args** (*mixed*) – optional variadic argument to pass to the coroutines function.

Returns coroutines returned results.

Return type list

Raises

- `TypeError` – in case of invalid coroutine object.
- `ValueError` – in case of empty set of coroutines or futures.
- `TimeoutError` – if execution takes more than expected.

Usage:

```
async def sum(x, y):
    return x + y

await paco.series(
    sum(1, 2),
    sum(2, 3),
    sum(3, 4))
# => [3, 5, 7]
```

paco.**some**(*coro*, *iterable*, *limit=0*, *timeout=None*, *loop=None*)

Returns *True* if at least one element in the iterable satisfies the asynchronous coroutine test. If any iteratee call returns *True*, iteration stops and *True* will be returned.

This function is a coroutine.

This function can be composed in a pipeline chain with `|` operator.

Parameters

- **coro** (*coroutine function*) – coroutine function for test values.
- **iterable** (*iterable*) – an iterable.
- **limit** (*int*) – max concurrency limit. Use `0` for no limit.
- **can be used to control the maximum number** (`timeout`) – of seconds to wait before returning. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.

Raises `TypeError` – if input arguments are not valid.

Returns *True* if at least on value passes the test, otherwise *False*.

Return type bool

Usage:

```
async def gt_3(num):
    return num > 3

await paco.some(test, [1, 2, 3, 4, 5])
# => True
```

paco.**throttle**(*coro*, *limit=1*, *timeframe=1*, *return_value=None*, *raise_exception=False*)

Creates a throttled coroutine function that only invokes `coro` at most once per every time frame of seconds or milliseconds.

Provide options to indicate whether func should be invoked on the leading and/or trailing edge of the wait timeout.

Subsequent calls to the throttled coroutine return the result of the last coroutine invocation.

This function can be used as decorator.

Parameters

- **coro** (*coroutinefunction*) – coroutine function to wrap with throttle strategy.
- **limit** (*int*) – number of coroutine allowed execution in the given time frame.
- **timeframe** (*int|float*) – throttle limit time frame in seconds.
- **return_value** (*mixed*) – optional return if the throttle limit is reached. Returns the latest returned value by default.
- **raise_exception** (*bool*) – raise exception if throttle limit is reached.

Raises RuntimeError – if cannot throttle limit reached (optional).

Returns coroutinefunction

Usage:

```
async def mul_2(num):
    return num * 2

# Use as simple wrapper
throttled = paco.throttle(mul_2, limit=1, timeframe=2)
await throttled(2)
# => 4
await throttled(3)  # ignored!
# => 4
await asyncio.sleep(2)
await throttled(3)  # executed!
# => 6

# Use as decorator
@paco.throttle(limit=1, timeframe=2)
async def mul_2(num):
    return num * 2

await mul_2(2)
# => 4
await mul_2(3)  # ignored!
# => 4
await asyncio.sleep(2)
await mul_2(3)  # executed!
# => 6
```

paco.**timeout**(*coro*, *timeout=None*, *loop=None*)

Wraps a given coroutine function, that when executed, if it takes more than the given timeout in seconds to execute, it will be canceled and raise an *asyncio.TimeoutError*.

This function is equivalent to Python standard *asyncio.wait_for()* function.

This function can be used as decorator.

Parameters

- **coro** (*coroutinefunction|coroutine*) – coroutine to wrap.
- **timeout** (*int|float*) – max wait timeout in seconds.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.

Raises TypeError – if coro argument is not a coroutine function.

Returns wrapper coroutine function.

Return type coroutinefunction

Usage:

```
await paco.timeout(coro, timeout=10)
```

class paco.**TimeoutLimit**(*timeout*, *loop=None*)
Bases: object

Timeout limit context manager.

Useful in cases when you want to apply timeout logic around block of code or in cases when asyncio.wait_for is not suitable.

Originally based on: https://github.com/aio-libs/async-timeout

Parameters

- **timeout** (*int*) – value in seconds or None to disable timeout logic.
- **loop** (*asyncio.BaseEventLoop*) – asyncio compatible event loop.

Usage:

```
with paco.TimeoutLimit(0.1):
    await paco.wait(task1, task2)
```

cancel()
Cancels current task running task in the context manager.

paco.**times**(*coro*, *limit=1*, *raise_exception=False*, *return_value=None*)
Wraps a given coroutine function to be executed only a certain amount of times.

If the execution limit is exceeded, the last execution return value will be returned as result.

You can optionally define a custom return value on exceeded via *return_value* param.

This function can be used as decorator.

Parameters

- **coro** (*coroutinefunction*) – coroutine function to wrap.
- **limit** (*int*) – max limit of coroutine executions.
- **raise_exception** (*bool*) – raise exception if execution times exceeded.
- **return_value** (*mixed*) – value to return when execution times exceeded.

Raises

- TypeError – if coro argument is not a coroutine function.
- RuntimeError – if max execution excedeed (optional).

Returns coroutinefunction

Usage:

```
async def mul_2(num):
    return num * 2

timed = paco.times(mul_2, 3)
await timed(2)
# => 4
await timed(3)
# => 6
await timed(4)
# => 8
```

```
    await timed(5)  # ignored!
    # => 8
```

paco.**until**(*coro*, *coro_test*, *assert_coro=None*, **args*, ***kw*)

Repeatedly call *coro* coroutine function until *coro_test* returns *True*.

This function is the inverse of *paco.whilst()*.

This function is a coroutine.

Parameters

- **coro** (*coroutinefunction*) – coroutine function to execute.
- **coro_test** (*coroutinefunction*) – coroutine function to test.
- **assert_coro** (*coroutinefunction*) – optional assertion coroutine used to determine if the test passed or not.
- ***args** (*mixed*) – optional variadic arguments to pass to *coro* function.

Raises TypeError – if input arguments are invalid.

Returns result values returned by *coro*.

Return type list

Usage:

```
    calls = 0

    async def task():
        nonlocal calls
        calls += 1
        return calls

    async def calls_gt_4():
        return calls > 4

    await paco.until(task, calls_gt_4)
    # => [1, 2, 3, 4, 5]
```

paco.**wait**(**coros_or_futures*, *limit=0*, *timeout=None*, *loop=None*, *return_exceptions=False*, *return_when='ALL_COMPLETED'*)

Wait for the Futures and coroutine objects given by the sequence futures to complete, with optional concurrency limit. Coroutines will be wrapped in Tasks.

timeout can be used to control the maximum number of seconds to wait before returning. timeout can be an int or float. If timeout is not specified or None, there is no limit to the wait time.

If return_exceptions is True, exceptions in the tasks are treated the same as successful results, and gathered in the result list; otherwise, the first raised exception will be immediately propagated to the returned future.

return_when indicates when this function should return. It must be one of the following constants of the concurrent.futures module.

All futures must share the same event loop.

This functions is mostly compatible with Python standard asyncio.wait().

Parameters

- ***coros_or_futures** (*iter|list*) – an iterable collection yielding coroutines functions.

- **limit** (*int*) – optional concurrency execution limit. Use `0` for no limit.
- **timeout** (*int/float*) – maximum number of seconds to wait before returning.
- **return_exceptions** (*bool*) – exceptions in the tasks are treated the same as successful results, instead of raising them.
- **return_when** (*str*) – indicates when this function should return.
- **loop** (*asyncio.BaseEventLoop*) – optional event loop to use.
- ***args** (*mixed*) – optional variadic argument to pass to the coroutines function.

Returns Returns two sets of Future: (done, pending).

Return type tuple

Raises

- `TypeError` – in case of invalid coroutine object.
- `ValueError` – in case of empty set of coroutines or futures.
- `TimeoutError` – if execution takes more than expected.

Usage:

```
async def sum(x, y):
    return x + y

done, pending = await paco.wait(
    sum(1, 2),
    sum(3, 4))
[task.result() for task in done]
# => [3, 7]
```

paco.**whilst**(*coro*, *coro_test*, *assert_coro=None*, **args*, ***kw*)

Repeatedly call *coro* coroutine function while *coro_test* returns *True*.

This function is the inverse of *paco.until()*.

This function is a coroutine.

Parameters

- **coro** (*coroutinefunction*) – coroutine function to execute.
- **coro_test** (*coroutinefunction*) – coroutine function to test.
- **assert_coro** (*coroutinefunction*) – optional assertion coroutine used to determine if the test passed or not.
- ***args** (*mixed*) – optional variadic arguments to pass to *coro* function.

Raises `TypeError` – if input arguments are invalid.

Returns result values returned by *coro*.

Return type list

Usage:

```
calls = 0

async def task():
    nonlocal calls
    calls += 1
```

```
        return calls

    async def calls_lt_4():
        return calls > 4

    await paco.until(task, calls_lt_4)
    # => [1, 2, 3, 4, 5]
```

paco.**wraps**(*fn*)

Wraps a given function as coroutine function.

This function can be used as decorator.

Parameters **fn** (*function*) – function object to wrap.

Returns wrapped function as coroutine.

Return type coroutinefunction

Usage:

```
    def mul_2(num):
        return num * 2

    # Use as function wrapper
    coro = paco.wraps(mul_2)
    await coro(2)
    # => 4

    # Use as decorator
    @paco.wraps
    def mul_2(num):
        return num * 2

    await mul_2(2)
    # => 4
```

5.3 History

5.3.1 0.1.6 / 2016-12-11

- feat(pipe): isolate pipe operator overload code
- refactor: decorator and util functions
- feat(#11): timeout limit context manager.
- refactor(core): several minor refactors
- fix(docs): comment out latex sphinx settings
- fix(docs): use current package version
- Documentation examples improvements (#27)
- feat(history): update
- feat: add pool length magic method

5.3.2 0.1.5 (2016-12-04)

- fix(#25): allow empty iterables in iterators functions, such as `map`, `filter`, `reduce`.

5.3.3 0.1.4 (2016-11-28)

- fix(#24): explicitly pass loop instance to `asyncio.wait`.

5.3.4 0.1.3 (2016-10-27)

- feat(#17): add `flat_map` function.
- feat(#18): add pipeline-style operator overloading composition.

5.3.5 0.1.2 (2016-10-25)

- fix(setup.py): fix pip installation.
- refactor(api): minor refactors in several functions and tests.

5.3.6 0.1.1 (2016-10-24)

- refactor(name): use new project name.

5.3.7 0.1.0 (2016-10-23)

- First version (beta)

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

p

Symbols

A

C

D

E

F

G

I

M

O

P

R

S

T

U

W