Ori

Neocrym Records Inc.

Feb 18, 2022

CONTENTS:

CHAPTER

ONE

ORI.ASYNCIO

Utility code for working with Python's asyncio.

ori.asyncio.**sync**(*awaitable*, *event_loop=None*)

Run an awaitable synchronously. Good for calling asyncio code from sync Python.

The usage is as follows:

```
from ori.asyncio import sync

async def your_function(arg1, arg2):
    # Additional asyncio code can go here
    return arg1 + arg2

# If you call your_function() directly, you will get a coroutine
# object. You need to either use the await keyword...
# or the sync() wrapper as below.

value = sync(your_function(1, 2))
assert value == 3
```

Parameters

- **awaitable** (Awaitable) – This is a function declared with *async def* or another type of awaitable object in Python.
- **event_loop** (Optional[AbstractEventLoop]) – An asyncio event loop. This defaults to *None*, in which case this function will try to pull the running event loop. If no event loop exists, this function will try to create one.

Returns This returns the same value that *awaitable* would if you used the *await* keyword instead of *sync()*.

Raises TypeError – If *awaitable* is not actually an awaitable object, then we raise a TypeError.

CHAPTER

TWO

ORI.CONCURRENCY

Utilities for concurrency of Python functions with multithreading or multiprocessing.

ori.concurrency.**run_in_background**(*func*, ***, *executor_class*, *max_workers=None*)

Runs the function in a background thread or process, depending on the executor that you provided.

Most users will want to use *ori.concurrency.run_in_background_thread()* or *ori.concurrency.run_in_background_process()* instead.

Parameters

- **func** (Callable) – The function to run in the background.
- **executor_class** (type) – A *concurrent.futures.Executor* subclass that the function will be submitted to. Two popular choices are concurrent.futures.ThreadPoolExecutor or concurrent.futures.ProcessPoolExecutor, but you can pass your own subclass if if it follows the same API.
- **max_workers** (Optional[int]) – Specify this to set the maximum number of parallel threads or processes to run at once. If this is set to None, then the executor's default maximum number of workers are used.

Returns This function returns a wrapper of the *func* function that you passed. The wrapper takes the exact same arguments, but instead returns a concurrent.futures.Future object. To wait for the return value of *func*, call *.result()* on the future that this function returns.

Raises *OriValidationError* – When you provide a value for *func* that is not a callable function.

ori.concurrency.**run_in_background_process**(*func*, ***, *max_workers=None*)

Runs the function in a background process.

Parameters

- **func** (Callable) – The function to run in the background.
- **max_workers** (Optional[int]) – Specify this to set the maximum number of parallel threads or processes to run at once. If this is set to None, then the executor's default maximum number of workers are used.

Returns This function returns a wrapper of the *func* function that you passed. The wrapper takes the exact same arguments, but instead returns a concurrent.futures.Future object. To wait for the return value of *func*, call *.result()* on the future that this function returns.

Raises *OriValidationError* – When you provide a value for *func* that is not a callable function.

ori.concurrency.**run_in_background_thread**(*func*, ***, *max_workers=None*)

Runs the function in a background thread.

Parameters

- **func** (Callable) – The function to run in the background.
- **max_workers** (Optional[int]) – Specify this to set the maximum number of parallel threads or processes to run at once. If this is set to None, then the executor's default maximum number of workers are used.

Returns This function returns a wrapper of the *func* function that you passed. The wrapper takes the exact same arguments, but instead returns a concurrent.futures.Future object. To wait for the return value of *func*, call *.result()* on the future that this function returns.

Raises *OriValidationError* – When you provide a value for *func* that is not a callable function.

CHAPTER

THREE

ORI.POOLCHAIN

Module for the poolchain.

The "poolchain" is a high-level abstraction over the Python `concurrent.futures` concurrency module. A poolchain used to run a single iterable (e.g. a list) through a chain of functions and collect an iterable of results at the end.

For example, here is one way to use a poolchain.

```
from ori.poolchain import PoolChain

results = (
    PoolChain()
    .add_threadpool(lambda num: num * 10)
    .add_processpool(str, chunksize=2, max_workers=3)
    .add_threadpool(lambda s: s + " is a number", timeout=5)
).execute_eager([1, 2, 3, 4, 5])

for row in results:
    print(row)
# `results` is an iterator, so we go through it with a for loop.
# we could have run `.execute_eager()` to automatically
# turn the iterator into a list.
```

The general workflow is to create a `PoolChain` object and then make a series of chained calls that add functions to execute over every item in the iterable in sequence.

After the chain of functions has been specified, call one of the *.execute_*()* functions like *PoolChain.execute_lazy()* to get an iterator of the final return value or *PoolChain.execute_eager()* to get an entire list upfront.

class ori.poolchain.**PoolChain**

Bases: `object`

Creates a poolchain.

add(*function*, *, *executor_class*, *max_workers=None*, *timeout=None*, *chunksize=1*)

Add a new element to the poolchain.

Note that if your *executor_class* is a `concurrent.futures.ProcessPoolExecutor`, you will not be able to pass *lambda* functions as *func*. You will need to use named functions defined with the *def* keyword.

Parameters

- **function** (`Callable`) – This is the function that you want to run at the end of the current chain.

- **executor_class** (type) – the concurrent.futures.Executor subclass type to use to execute this part of the poolchain.
- **max_workers** (Optional[int]) – The maximum number of workers that the concurrent.futures.ProcessPoolExecutor can start. If this is *None*, then then the executor will create the same number of workers as you have processors on your machine. This is usually a sensible default.
- **timeout** (Optional[int]) – The amount of time (in seconds) to wait for a concurrent.futures.Future before throwing an error.
- **chunksize** (int) – Sometimes the executor runs faster when providing chunks of the iterable to workers rather than just one iterable element at a time (the default). Try setting chunksize to a value greater than one to more efficiently use interprocess communication.

Returns The PoolChain object itself, allowing you to chain subsequent *add_*()* calls and at the end an *execute_*()* call.

Raises *OriValidationError* – Thrown if you don't send a real function, or a real Executor, a non-integer for one of the integer fields, etc.

add_processpool(*function*, *, *max_workers=None*, *timeout=None*, *chunksize=1*)

Add a function to run with a "process pool" to the poolchain.

You cannot pass a *lambda* function to a process pool. You need to make up your poolchain entirely with named functions defined with the *def* keyword.

Parameters

- **function** (Callable) – This is the function that you want to run at the end of the current chain.
- **max_workers** (Optional[int]) – The maximum number of workers that the concurrent.futures.ProcessPoolExecutor can start. If this is *None*, then then the executor will create the same number of workers as you have processors on your machine. This is usually a sensible default.
- **timeout** (Optional[int]) – The amount of time (in seconds) to wait for a concurrent.futures.Future before throwing an error.
- **chunksize** (int) – Sometimes the executor runs faster when providing chunks of the iterable to workers rather than just one iterable element at a time (the default). Try setting chunksize to a value greater than one to more efficiently use interprocess communication.

Returns The PoolChain object itself, allowing you to chain subsequent *add_*()* calls and at the end an *execute_*()* call.

Raises *OriValidationError* – Thrown if you don't send a real function, or a real Executor, a non-integer for one of the integer fields, etc.

add_threadpool(*function*, *, *max_workers=None*, *timeout=None*)

Add a function to run with threadpool to the poolchain.

Parameters

- **function** (Callable) – This is the function that you want to run at the end of the current chain.
- **max_workers** (Optional[int]) – The maximum number of workers that the concurrent.futures.ProcessPoolExecutor can start. If this is *None*, then then the executor will create the same number of workers as you have processors on your machine. This is usually a sensible default.

- **timeout** (Optional[int]) – The amount of time (in seconds) to wait for a concurrent.futures.Future before throwing an error.

Returns The PoolChain object itself, allowing you to chain subsequent *add_*()* calls and at the end an *execute_*()* call.

Raises *OriValidationError* – Thrown if you don't send a real function, or a real Executor, a non-integer for one of the integer fields, etc.

execute_eager(*iterable*)

Run the given iterable through the poolchain, eagerly returning a list.

This method returns the results upfront in a list, which lets you look at the entire output without having to iterate through it yourself.

Parameters **iterable** (Iterable[Any]) – This is an iterable where every element in the iterable is processed through the chain.

Returns This function returns a complete Python list containing the return value of every element in the list.

execute_eager_single_threaded(*iterable*)

Execute the chain in the current thread, returning a Python list.

This method is typically used when you want to test your chain without creating threads or processes. When running in one thread, PoolChain would never be faster than just running the functions yourself in sequence.

This method also returns the results upfront in a list, which lets you look at the entire output without having to iterate through it yourself.

Parameters **iterable** (Iterable[Any]) – This is an iterable where every element in the iterable is processed through the chain.

Returns This function returns a complete Python list containing the return value of every element in the list.

execute_lazy(*iterable*)

Process the given iterable through the poolchain.

When you call this function, the poolchain begins executing immediately

Returns This function returns an iterator that lazily executes the poolchain.

execute_lazy_single_threaded(*iterable*)

Execute the chain in the current thread, returning a generator.

This function is typically used when you want to test your chain without creating threads or processes. When running in one thread, PoolChain would never be faster than just running the functions yourself in sequence.

This function returns a generator that you have to iterate through yourself for results.

Parameters **iterable** (Iterable[Any]) – This is an iterable where every element in the iterable is processed through the chain.

Returns This function returns an iterator that lazily executes the poolchain in the current thread. Every time the next output is fetched, the corresponding input is sent through every function in the chain.

CHAPTER

FOUR

ORI.SUBPROCESS

Utility function for running external commands as subprocesses.

ori.subprocess.**run_process_in_background**(*, *command*, *stdout_function=None*, *stderr_function=None*, *exception_function=None*, *exit_code_on_killed=0*)

Runs *command*, writing stdout and stderr to the logger in *logger*.

This function is based on this answer on StackOverflow.

Parameters

- **command** (Sequence[Union[bytes, str, PathLike]]) – This is an iterable–like a list–of strings. This iterable describes a command line program to run. For example, the command *ls -l /home* would be broken up into *["ls", "-l", "/home"]*. You can use the function `shlex.split()` to turn any string command into an iterable.
- **stdout_function** (Optional[Callable]) – This is a function that your subprocess will call for every line of standard output that your program emits. You can choose functions like `print()` or, say, `logging.debug()`.
- **stderr_function** (Optional[Callable]) – This is a function that your subprocess will call for every line of standard error that your program emits. The same rules apply as for *stdout_function*.
- **exception_function** (Optional[Callable]) – This is a function to call if we catch an exception while logging the program.
- **exit_code_on_killed** (int) – This is the exit code we return for the subprocess when we catch an exception while logging the program.

Returns This returns an already-started `multiprocessing.Process` instance, which you can use to monitor or kill the process as time goes on.

Warning: The parameters **stdout_function**, **stderr_function** and **exception_function** are used to log the standard output, standard error, and any exceptions that your command might raise. These functions are run in a Python subprocess, which means that they must be *importable* at the module level and *pickleable*. Closures, which are functions enclosed in other functions, are not pickleable.

If you feel like you need to add additional parameters or state to your logging functions, you can write your functions as instances of Python classes with the __call__() magic method.

CHAPTER

FIVE

ORI.ERRORS

All exceptions for the Ori module.

exception ori.errors.**OriError**
Bases: Exception

The base class efor all Ori errors.

exception ori.errors.**OriNoInteractiveProcessPools**
Bases: *ori.errors.OriError*

Raises when the user tries to use process pools in interactive mode.

The concurrent.futures.ProcessPoolExecutor does not work when the user is running in an interactive shell. Code must be saved to disk and importable in order to be used.

exception ori.errors.**OriValidationError**
Bases: *ori.errors.OriError*

Raise when Ori receives bad input from the user.

Ori is a high-level wrapper around Python's *concurrent.futures* module, designed to make multithreading and multiprocessing as easy as possible.

CHAPTER

SIX

ORI MODULES

The tools that Ori provides are divided into several modules.

ori.asyncio – Tools to integrate Python asyncio code into a synchronous codebase, and vice-versa.

ori.concurrency – Tools to run Python functions in the background using multithreading or multiprocessing.

ori.poolchain – A way to chain function calls for parallel processing over any list or other iterable.

ori.subprocess – Tools for running external commands as subprocesses and efficiently collecting the standard output and standard error.

CHAPTER

SEVEN

FREQUENTLY ASKED QUESTIONS (FAQS)

Who made Ori?

Ori was written by James Mishra and incubated at Neocrym, a record label that uses artificial intelligence to find and promote musicians. Neocrym heavily relies on Ori to make their I/O-bound Python code run faster.

The source code for Ori is owned by Neocrym Records Inc., but licensed to Ori under the MIT License.

Why should I use Ori over directly interfacing with concurrent.futures?

The Python module concurrent.futures was introduced as a high-level abstraction over lower-level interfaces like *threading.Thread* and *multiprocessing.Process*. However, *concurrent.futures* merely moves the problem away from managing threads or processes to managing *executors*. Ori has the ambitious goal of also abstracting away the executors–making multithreading or multiprocessing no harder than writing single-threaded code.

Is Ori a good replacement for Python's asyncio?

For the hardcore asyncio user, probably not. Ori is focused on providing high-level abstractions over Python's concurrent.futures module that provides speed boosts for synchronous, I/O-bound Python.

What do I need to know to contribute to Ori?

Ori manages itself with the Python packaging tool Poetry. You can install Poetry on your system with:

```
pip3 install poetry
poetry install
```

To check that your changes to Ori's codebase match our coding standards, and to reformat any errant code to meet our standards, run this command:

```
poetry run make lint
```

To run Ori's unit tests in the Python virtualenv created by Poetry, just run:

```
poetry run make test
```

We can also run tests across multiple versions of Python with Tox, but it requres your system has Docker and Docker Compose installed. If so, just run:

```
poetry run make tox
```

Where did the name "Ori" come from?

The name "Ori" is a reference to the god-like villains in the Stargate TV shows. There is no meaningful connection between the villains or concurrency.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

O

INDEX