aioshell Documentation

Release 1.0a1

Carlos Eduardo Moreira dos Santos

December 08, 2015

Contents

Run shell and SSH commands concurrently inside Python code with few keystrokes and the new powerful Python's `asyncio` module (since version 3.4). Quick example:

```
from aioshell import Executor, Shell
exe = Executor()
exe.add(Shell('date >/tmp/aioshell; sleep 1'))
exe.add(Shell('sleep 1; date >>/tmp/aioshell'))
exe.finish()
# Check /tmp/aioshell file to see that only 1 sec has passed.
```

Contents:

CHAPTER 1

aioshell module

Run single-threaded concurrent shell and ssh commands with few keystrokes.

A simpler way to use Python's new `asyncio` module, making it easier and faster to run shell and ssh commands.

1.1 Executor

class `aioshell.`**`Executor`**

Less boilerplate to run asynchronous tasks.

Basic usage: call *`add()`* once for every *`Runnable`* object (*`Shell`*, *`SSH`*) and, in the end, call *`finish()`* to wait for them to finish and clean resources. Example:

```
from aioshell import Executor, Shell
exe = Executor()
exe.add(Shell('date >/tmp/aioshell; sleep 1'))
exe.add(Shell('sleep 1; date >>/tmp/aioshell'))
exe.finish()
```

In the */tmp/test* file, you'll notice that it took only 1 second instead of 2.

`add`(*runnable_coro_future*)

Run a Runnable (basic usage), coroutine or Future.

Note for advanced users: each call will append a correspondent `Task` or `Future` to *`futures`* in case you want more information about the execution than provided by this class.

Parameters **`runnable_coro_future`** (*`Runnable`* (e.g. *`Shell`*, *`SSH`*), coroutine or `Future` objects) – action to be performed asynchronously.

`finish`()

Wait for all tasks, close the event loop and clean resources.

You should call this method in the end of your program. It performs 3 actions:

1. Wait for all added tasks (by *`add()`*) to be finished (like *`wait()`* does);
2. Clear *`futures`* list;
3. Close the main event loop:

- Allow clean exit, without warnings;
- If you need to run anything else, use a new object of this class.

futures = None
(*Advanced usage*) accumulated results of *add()*.

Useful for further information about executions. If *add()* is called with a *Runnable* or coroutine object, a correpondent Task will be appended to this list. If a Future is added, the Future itself will be appended.

Type list of Task or Future

loop = None
(*Advanced usage*) event loop.

This event loop is shared between all objects until it is closed. If the loop is closed, automatically create a new one (it will also be shared between new objects).

Type BaseEventLoop

run_wait(*runnable_coro_future*)
Block until *runnable_coro_future* is finished.

It will wait for only this *runnable_coro_future*. Useful if you want the traditional behaviour of sequential programming for some reason. Otherwise, use the concurrent and faster version *add()*.

Parameters **runnable_coro_future** (*Runnable* (e.g. *Shell*, *SSH*), coroutine or Future objects) – action to be performed synchronously.

wait()
Block until all added tasks by *add()* are done.

You can add more tasks later. When you are finished, call *finish()*.

1.2 Shell

class aioshell.**Shell**(*cmd*, *title=None*, *stdout=None*, *stderr=None*)
Run shell commands asynchronously with few keystrokes.

Examples:

```
from aioshell import Executor, Shell
exe = Executor()

# If you don't care about shell's output
exe.add(Shell('date >/tmp/aioshell'))

# Output can be read later from Shell object
shell = Shell('date', stdout=Shell.TRUE)
exe.add(shell)

# We won't add any other task, so let's finish:
exe.finish()

# All the tasks are done after finish(), so stdout is now available:
print(shell.stdout)
```

The constructor has all the information to run a shell command. It will be run after being passed as an argument to *Executor.add()*. The default behavior is to capture only stderr output (for error debugging).

Parameters

- **cmd** (*str*) – shell command to be executed.
- **title** (*str*) – a meaninful name for your task for debugging purposes.

- **stdout** (*Shell.DEVNULL* or *Shell.TRUE*) – whether or not to capture stdout. Default: don't capture.
- **stderr** (*Shell.TRUE*, *Shell.DEVNULL* or *Shell.ERR2OUT*) – whether or not to capture stderr. Default: capture.

Variables

- **cmd** (*str*) – shell command (constructor's argument).
- **title** (*str*) – title as in the constructor.
- **returncode** (*int*) – shell exit code.
- **stdout** (*str*) – shell standard output. None if not requested or not executed.
- **stderr** (*str*) – shell standard error. None if not requested or not executed.

DEVNULL = -3
Ignore stdout or stderr output.

ERR2OUT = -2
Mix stderr and stdout outputs in stdout.

TRUE = -1
Capture stdout or stderr output.

run()
(coroutine) Execute shell command asynchronously.

You should not call this method directly. Instead, pass this object as an argument to *Executor.add()*.

Returns coroutine for the shell stdout (also found as an attribute).

Return type coroutine, str

Raises CalledProcessError

1.3 SSH

class aioshell.**SSH**(*params*, *cmd*, *title=None*, *stdout=None*, *stderr=None*)
Run SSH asynchronously.

The difference between using this class and *Shell* is that the remote stdout and stderr are also managed. For example, if a remote command prints long and useless output, it would be transfered throught the network and then discarded locally by *Shell*. To solve this problem, SSH class manages also the remote stdout and stderr, so no bandwidth is wasted.

SSH behaves like *Shell* (subclass). The only difference is:

Parameters **params** (*string*) – all ssh options. Requires hostname or IP address.

1.4 Runnable

class aioshell.**Runnable**
Interface used by *Executor*.

Implement this interface to easily run asynchronous code with *Executor.add()*.

Current implementations: *Shell*, *SSH*.

run()
Coroutine called by *Executor.add()*, without arguments.

If you need arguments, keep them as attributes instead.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

a

A

D

E

F

L

R

S

T

W