Working with Subprocesses

It is frequently necessary to work with other programs and processes, to take advantage of existing code without rewriting it or to access libraries or features not available from within Python. As with network I/O, `asyncio` includes two abstractions for starting another program and then interacting with it.

Using the Protocol Abstraction with Subprocesses

This example uses a coroutine to launch a process to run the Unix command `df` to find the free space on local disks. It uses `subprocess_exec()` to launch the process and tie it to a protocol class that knows how to read the `df` command output and parse it. The methods of the protocol class are called automatically based on I/O events for the subprocess. Because both the `stdin` and `stderr` arguments are set to None, those communication channels are not connected to the new process.

```
# asyncio_subprocess_protocol.py

import asyncio
import functools


async def run_df(loop):
    print('in run_df')

    cmd_done = asyncio.Future(loop=loop)
    factory = functools.partial(DFProtocol, cmd_done)
    proc = loop.subprocess_exec(
        factory,
        'df', '-hl',
        stdin=None,
        stderr=None,
    )
    try:
        print('launching process')
        transport, protocol = await proc
        print('waiting for process to complete')
        await cmd_done
    finally:
        transport.close()

    return cmd_done.result()
```

The class `DFProtocol` is derived from `SubprocessProtocol`, which defines the API for a class to communicate with another process through pipes. The done argument is expected to be a `Future` that the caller will use to watch for the process to finish.

```
class DFProtocol(asyncio.SubprocessProtocol):

    FD_NAMES = ['stdin', 'stdout', 'stderr']

    def __init__(self, done_future):
        self.done = done_future
        self.buffer = bytearray()
        super().__init__()
```

As with socket communication, `connection_made()` is invoked when the input channels to the new process are set up. The `transport` argument is an instance of a subclass of `BaseSubprocessTransport`. It can read data output by the process and write data to the input stream for the process, if the process was configured to receive input.

```
    def connection_made(self, transport):
        print('process started {}'.format(transport.get_pid()))
        self.transport = transport
```

When the process has generated output, `pipe_data_received()` is invoked with the file descriptor where the data was emitted and the actual data read from the pipe. The protocol class saves the output from the standard output channel of the process in a buffer for later processing.

```
    def pipe_data_received(self, fd, data):
        print('read {} bytes from {}'.format(len(data),
                                             self.FD_NAMES[fd]))
        if fd == 1:
            self.buffer.extend(data)
```

When the process terminates, `process_exited()` is called. The exit code of the process is available from the transport object by calling `get_returncode()`. In this case, if there is no error reported the available output is decoded and parsed before being returned through the `Future` instance. If there is an error, the results are assumed to be empty. Setting the result of the future tells `run_df()` that the process has exited, so it cleans up and then returns the results.

```
    def process_exited(self):
        print('process exited')
        return_code = self.transport.get_returncode()
        print('return code {}'.format(return_code))
        if not return_code:
            cmd_output = bytes(self.buffer).decode()
            results = self._parse_results(cmd_output)
        else:
            results = []
        self.done.set_result((return_code, results))
```

The command output is parsed into a sequence of dictionaries mapping the header names to their values for each line of output, and the resulting list is returned.

```
    def _parse_results(self, output):
        print('parsing results')
        # Output has one row of headers, all single words.  The
        # remaining rows are one per filesystem, with columns
        # matching the headers (assuming that none of the
        # mount points have whitespace in the names).
        if not output:
            return []
        lines = output.splitlines()
        headers = lines[0].split()
        devices = lines[1:]
        results = [
            dict(zip(headers, line.split()))
            for line in devices
        ]
        return results
```

The `run_df()` coroutine is run using `run_until_complete()`, then the results are examined and the free space on each device is printed.

```
event_loop = asyncio.get_event_loop()
try:
    return_code, results = event_loop.run_until_complete(
        run_df(event_loop)
    )
finally:
    event_loop.close()

if return_code:
    print('error exit {}'.format(return_code))
else:
    print('\nFree space:')
    for r in results:
        print('{Mounted:25}: {Avail}'.format(**r))
```

The output below shows the sequence of steps taken, and the free space on three drives on the system where it was run.

```
$ python3 asyncio_subprocess_protocol.py

in run_df
```

```
launching process
process started 49675
waiting for process to complete
read 332 bytes from stdout
process exited
return code 0
parsing results

Free space:
/                        : 233Gi
/Volumes/hubertinternal  : 157Gi
/Volumes/hubert-tm       : 2.3Ti
```

Calling Subprocesses with Coroutines and Streams

To use coroutines to run a process directly, instead of accessing it through a `Protocol` subclass, call `create_subprocess_exec()` and specify which of `stdout`, `stderr`, and `stdin` to connect to a pipe. The result of the coroutine to spawn the subprocess is a `Process` instance that can be used to manipulate the subprocess or communicate with it.

```
# asyncio_subprocess_coroutine.py

import asyncio
import asyncio.subprocess


async def run_df():
    print('in run_df')

    buffer = bytearray()

    create = asyncio.create_subprocess_exec(
        'df', '-hl',
        stdout=asyncio.subprocess.PIPE,
    )
    print('launching process')
    proc = await create
    print('process started {}'.format(proc.pid))
```

In this example, `df` does not need any input other than its command line arguments, so the next step is to read all of the output. With the `Protocol` there is no control over how much data is read at a time. This example uses `readline()` but it could also call `read()` directly to read data that is not line-oriented. The output of the command is buffered, as with the protocol example, so it can be parsed later.

```
    while True:
        line = await proc.stdout.readline()
        print('read {!r}'.format(line))
        if not line:
            print('no more output from command')
            break
        buffer.extend(line)
```

The `readline()` method returns an empty byte string when there is no more output because the program has finished. To ensure the process is cleaned up properly, the next step is to wait for the process to exit fully.

```
    print('waiting for process to complete')
    await proc.wait()
```

At that point the exit status can be examined to determine whether to parse the output or treat the error as it produced no output. The parsing logic is the same as in the previous example, but is in a stand-alone function (not shown here) because there is no protocol class to hide it in. After the data is parsed, the results and exit code are then returned to the caller.

```
    return_code = proc.returncode
    print('return code {}'.format(return_code))
    if not return_code:
        cmd_output = bytes(buffer).decode()
        results = _parse_results(cmd_output)
    else:
        results = []
```

```
    return (return_code, results)
```

The main program looks similar to the protocol-based example, because the implementation changes are isolated in `run_df()`.

```
event_loop = asyncio.get_event_loop()
try:
    return_code, results = event_loop.run_until_complete(
        run_df()
    )
finally:
    event_loop.close()

if return_code:
    print('error exit {}'.format(return_code))
else:
    print('\nFree space:')
    for r in results:
        print('{Mounted:25}: {Avail}'.format(**r))
```

Since the output from `df` can be read one line at a time, it is echoed to show the progress of the program. Otherwise, the output looks similar to the previous example.

```
$ python3 asyncio_subprocess_coroutine.py

in run_df
launching process
process started 49678
read b'Filesystem     Size   Used  Avail Capacity   iused
ifree %iused  Mounted on\n'
read b'/dev/disk2s2  446Gi  213Gi  233Gi    48%  55955082
61015132   48%   /\n'
read b'/dev/disk1    465Gi  307Gi  157Gi    67%  80514922
41281172   66%   /Volumes/hubertinternal\n'
read b'/dev/disk3s2  3.6Ti  1.4Ti  2.3Ti    38% 181837749
306480579   37%   /Volumes/hubert-tm\n'
read b''
no more output from command
waiting for process to complete
return code 0
parsing results

Free space:
/                        : 233Gi
/Volumes/hubertinternal  : 157Gi
/Volumes/hubert-tm       : 2.3Ti
```

Sending Data to a Subprocess

Both of the previous examples used only a single communication channel to read data from a second process. It is often necessary to send data into a command for processing. This example defines a coroutine to execute the Unix command `tr` for translating characters in its input stream. In this case, `tr` is used to convert lower-case letters to upper-case letters.

The `to_upper()` coroutine takes as argument an event loop and an input string. It spawns a second process running "`tr [:lower:] [:upper:]`".

```
# asyncio_subprocess_coroutine_write.py
```

```
import asyncio
import asyncio.subprocess


async def to_upper(input):
    print('in to_upper')

    create = asyncio.create_subprocess_exec(
        'tr', '[:lower:]', '[:upper:]',
        stdout=asyncio.subprocess.PIPE,
        stdin=asyncio.subprocess.PIPE,
```

```
        stdin=asyncio.subprocess.PIPE,
    )
    print('launching process')
    proc = await create
    print('pid {}'.format(proc.pid))
```

Next `to_upper()` uses the `communicate()` method of the `Process` to send the input string to the command and read all of the resulting output, asynchronously. As with the `subprocess.Popen` version of the same method, `communicate()` returns the complete output byte strings. If a command is likely to produce more data than can fit comfortably into memory, the input cannot be produced all at once, or the output must be processed incrementally, it is possible to use the stdin, stdout, and stderr handles of the `Process` directly instead of calling `communicate()`.

```
    print('communicating with process')
    stdout, stderr = await proc.communicate(input.encode())
```

After the I/O is done, waiting for the process to completely exit ensures it is cleaned up properly.

```
    print('waiting for process to complete')
    await proc.wait()
```

The return code can then be examined, and the output byte string decoded, to prepare the return value from the coroutine.

```
    return_code = proc.returncode
    print('return code {}'.format(return_code))
    if not return_code:
        results = bytes(stdout).decode()
    else:
        results = ''

    return (return_code, results)
```

The main part of the program establishes a message string to be transformed, and then sets up the event loop to run `to_upper()` and prints the results.

```
MESSAGE = """
This message will be converted
to all caps.
"""

event_loop = asyncio.get_event_loop()
try:
    return_code, results = event_loop.run_until_complete(
        to_upper(MESSAGE)
    )
finally:
    event_loop.close()

if return_code:
    print('error exit {}'.format(return_code))
else:
    print('Original: {!r}'.format(MESSAGE))
    print('Changed : {!r}'.format(results))
```

The output shows the sequence of operations and then how the simple text message is transformed.

```
$ python3 asyncio_subprocess_coroutine_write.py

in to_upper
launching process
pid 49684
communicating with process
waiting for process to complete
return code 0
Original: '\nThis message will be converted\nto all caps.\n'
Changed : '\nTHIS MESSAGE WILL BE CONVERTED\nTO ALL CAPS.\n'
```

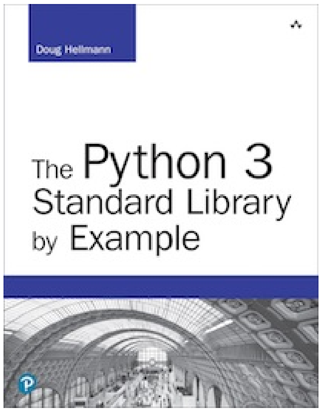
Quick Links

Using the Protocol Abstraction with Subprocesses
Calling Subprocesses with Coroutines and Streams
Sending Data to a Subprocess

This page was last updated 2016-12-18.

Navigation

Interacting with Domain Name Services
Receiving Unix Signals



Get the book

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

© Copyright 2019, Doug Hellmann

Other Writing

Blog
The Python Standard Library By Example