Sargedocumentation

Release 0.1.2

Vinay Sajip

December 17, 2013

Contents

Welcome to the documentation for `sarge`, a wrapper for `subprocess` which aims to make life easier for anyone who needs to interact with external applications from their Python code.

Please note: this documentation is *work in progress*.

CHAPTER 1

Overview

Start here for all things sarge.

1.1 What is Sarge for?

If you want to interact with external programs from your Python applications, Sarge is a library which is intended to make your life easier than using the subprocess module in Python's standard library.

Sarge is, of course, short for sergeant – and like any good non-commissioned officer, sarge works to issue commands on your behalf and to inform you about the results of running those commands.

The acronym lovers among you might be amused to learn that sarge can also stand for "Subprocess Allegedly Rewards Good Encapsulation" :-)

Here's a taster (example suggested by Kenneth Reitz's Envoy documentation):

```
>>> from sarge import capture_stdout
>>> p = capture_stdout('fortune|cowthink')
>>> p.returncode
0
>>> p.commands
[Command('fortune'), Command('cowthink')]
>>> p.returncodes
[0, 0]
>>> print(p.stdout.text)
 ____________________________________
( The last thing one knows in        )
( constructing a work is what to put )
( first.                             )
(                                    )
( -- Blaise Pascal                   )
 ------------------------------------
        o   ^__^
         o  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

The `capture_stdout()` function is a convenient form of an underlying function, `run()`. You can also use conditionals:

```
>>> from sarge import run
>>> p = run('false && echo foo')
>>> p.commands
[Command('false')]
>>> p.returncodes
[1]
>>> p.returncode
1
>>> p = run('false || echo foo')
foo
>>> p.commands
[Command('false'), Command('echo foo')]
>>> p.returncodes
[1, 0]
>>> p.returncode
0
```

The conditional logic is being done by sarge and not the shell – which means you can use the identical code on Windows. Here's an example of some more involved use of pipes, which also works identically on Posix and Windows:

```
>>> cmd = 'echo foo | tee stdout.log 3>&1 1>&2 2>&3 | tee stderr.log > %s' % os.devnull
>>> p = run(cmd)
>>> p.commands
[Command('echo foo'), Command('tee stdout.log'), Command('tee stderr.log')]
>>> p.returncodes
[0, 0, 0]
>>>
vinay@eta-oneiric64:~/projects/sarge$ cat stdout.log
foo
vinay@eta-oneiric64:~/projects/sarge$ cat stderr.log
foo
```

In the above example, the first tee invocation swaps its `stderr` and `stdout` – see this post for a longer explanation of this somewhat esoteric usage.

1.2 Why not just use `subprocess`?

The `subprocess` module in the standard library contains some very powerful functionality. It encapsulates the nitty-gritty details of subprocess creation and communication on Posix and Windows platforms, and presents the application programmer with a uniform interface to the OS-level facilities. However, `subprocess` does not do much more than this, and is difficult to use in some scenarios. For example:

- You want to use command pipelines, but using `subprocess` out of the box often leads to deadlocks because pipe buffers get filled up.
- You want to use bash-style pipe syntax on Windows, but Windows shells don't support some of the syntax you want to use, like `&&`, `||`, `|&` and so on.
- You want to process output from commands in a flexible way, and `communicate()` is not flexible enough for your needs – for example, you need to process output a line at a time.
- You want to avoid shell injection problems by having the ability to quote your command arguments safely.
- `subprocess` allows you to let `stderr` be the same as `stdout`, but not the other way around – and you need to do that.

1.3 Main features

Sarge offers the following features:

- A simple run command which allows a rich subset of Bash-style shell command syntax, but parsed and run by sarge so that you can run on Windows without `cygwin`.
- The ability to format shell commands with placeholders, such that variables are quoted to prevent shell injection attacks:

```
>>> from sarge import shell_format
>>> shell_format('ls {0}', '*.py')
"ls '*.py'"
>>> shell_format('cat {0}', 'a file name with spaces')
"cat 'a file name with spaces'"
```

- The ability to capture output streams without requiring you to program your own threads. You just use a `Capture` object and then you can read from it as and when you want:

```
>>> from sarge import Capture, run
>>> with Capture() as out:
...     run('echo foobarbaz', stdout=out)
...
<sarge.Pipeline object at 0x175ed10>
>>> out.read(3)
'foo'
>>> out.read(3)
'bar'
>>> out.read(3)
'baz'
>>> out.read(3)
'\n'
>>> out.read(3)
''
```

  A `Capture` object can capture the output from multiple commands:

```
>>> from sarge import run, Capture
>>> p = run('echo foo; echo bar; echo baz', stdout=Capture())
>>> p.stdout.readline()
'foo\n'
>>> p.stdout.readline()
'bar\n'
>>> p.stdout.readline()
'baz\n'
>>> p.stdout.readline()
''
```

  Delays in commands are honoured in asynchronous calls:

```
>>> from sarge import run, Capture
>>> cmd = 'echo foo & (sleep 2; echo bar) & (sleep 1; echo baz)'
>>> p = run(cmd, stdout=Capture(), async=True) # returns immediately
>>> p.close() # wait for completion
>>> p.stdout.readline()
'foo\n'
>>> p.stdout.readline()
'baz\n'
>>> p.stdout.readline()
```

```
'bar\n'
>>>
```

Here, the `sleep` commands ensure that the asynchronous `echo` calls occur in the order `foo` (no delay), `baz` (after a delay of one second) and `bar` (after a delay of two seconds); the capturing works as expected.

1.4 Python version and platform compatibility

Sarge is intended to be used on any Python version >= 2.6 and is tested on Python versions 2.6, 2.7, 3.1, 3.2 and 3.3 on Linux, Windows, and Mac OS X (not all versions are tested on all platforms, but are expected to work correctly).

1.5 Project status

The project has reached alpha status in its development: there is a test suite and it has been exercised on Windows, Ubuntu and Mac OS X. However, because of the timing sensitivity of the functionality, testing needs to be performed on as wide a range of hardware and platforms as possible.

The source repository for the project is on BitBucket:

https://bitbucket.org/vinay.sajip/sarge/

You can leave feedback by raising a new issue on the issue tracker (BitBucket registration not necessary, but recommended).

Note: For testing under Windows, you need to install the GnuWin32 coreutils package, and copy the relevant executables (currently `libiconv2.dll`, `libintl3.dll`, `cat.exe`, `echo.exe`, `tee.exe`, `false.exe`, `true.exe`, `sleep.exe` and `touch.exe`) to the directory from which you run the test harness (`test_sarge.py`).

1.6 API stability

Although every attempt will be made to keep API changes to the absolute minimum, it should be borne in mind that the software is in its very early stages. For example, the asynchronous feature (where commands are run in separate threads when you specify `&` in a command pipeline) can be considered experimental, and there may be changes in this area. However, you aren't forced to use this feature, and `sarge` should be useful without it.

1.7 Change log

1.7.1 0.1.3

Released: Not yet.

1.7.2 0.1.2

Released: 2013-12-17

- Fixed issue #13: Removed module globals to improve thread safety.

- Fixed issue #12: Fixed a hang which occurred when a redirection failed.
- Fixed issue #11: Added + to the characters allowed in parameters.
- Fixed issue #10: Removed a spurious debugger breakpoint.
- Fixed issue #9: Relative pathnames in redirections are now relative to the current working directory for the redirected process.
- Added the ability to pass objects with `fileno()` methods as values to the `input` argument of `run()`, and a `Feeder` class which facilitates passing data to child processes dynamically over time (rather than just an initial string, byte-string or file).
- Added functionality under Windows to use PATH, PATHEXT and the registry to find appropriate commands. This can e.g. convert a command `'foo bar'`, if `'foo.py'` is a Python script in the `c:\Tools` directory which is on the path, to the equivalent `'c:\Python26\Python.exe c:\Tools\foo.py bar'`. This is done internally when a command is parsed, before it is passed to `subprocess`.
- Fixed issue #7: Corrected handling of whitespace and redirections.
- Fixed issue #8: Added a missing import.
- Added Travis integration.
- Added encoding parameter to the `Capture` initializer.
- Fixed issue #6: addressed bugs in Capture logic so that iterating over captures is closer to `subprocess` behaviour.
- Tests added to cover added functionality and reported issues.
- Numerous documentation updates.

1.7.3 0.1.1

Released: 2013-06-04

- `expect` method added to `Capture` class, to allow searching for specific patterns in subprocess output streams.
- added `terminate`, `kill` and `poll` methods to `Command` class to operate on the wrapped subprocess.
- `Command.run` now propagates exceptions which occur while spawning subprocesses.
- Fixed issue #4: `shell_shlex` does not split on `@`.
- Fixed issue #3: `run` et al now accept commands as lists, just as `subprocess.Popen` does.
- Fixed issue #2: `shell_quote` implementation improved.
- Improved `shell_shlex` resilience by handling Unicode on 2.x (where `shlex` breaks if passed Unicode).
- Added `get_stdout`, `get_stderr` and `get_both` for when subprocess output is not expected to be voluminous.
- Added an internal lock to serialise access to shared data.
- Tests added to cover added functionality and reported issues.
- Numerous documentation updates.

1.7.4 0.1

Released: 2012-02-10

- Initial release.

1.8 Next steps

You might find it helpful to look at the *Tutorial*, or the *API Reference*.

CHAPTER 2

Tutorial

This is the place to start your practical exploration of `sarge`.

2.1 Installation and testing

sarge is a pure-Python library. You should be able to install it using:

```
pip install sarge
```

for installing `sarge` into a virtualenv or other directory where you have write permissions. On Posix platforms, you may need to invoke using `sudo` if you need to install `sarge` in a protected location such as your system Python's `site-packages` directory.

A full test suite is included with `sarge`. To run it, you'll need to unpack a source tarball and run `python setup.py test` in the top-level directory of the unpack location. You can of course also run `python setup.py install` to install from the source tarball (perhaps invoking with `sudo` if you need to install to a protected location).

2.2 Common usage patterns

In the simplest cases, sarge doesn't provide any major advantage over `subprocess`:

```
>>> from sarge import run
>>> run('echo "Hello, world!"')
Hello, world!
<sarge.Pipeline object at 0x1057110>
```

The `echo` command got run, as expected, and printed its output on the console. In addition, a `Pipeline` object got returned. Don't worry too much about what this is for now – it's more useful when more complex combinations of commands are run.

By comparison, the analogous case with `subprocess` would be:

```
>>> from subprocess import call
>>> call('echo "Hello, world!"'.split())
"Hello, world!"
0
```

We had to call `split()` on the command (or we could have passed `shell=True`), and as well as running the command, the `call()` method returned the exit code of the subprocess. To get the same effect with `sarge` you have to do:

```
>>> from sarge import run
>>> run('echo "Hello, world!"').returncode
Hello, world!
0
```

If that's as simple as you want to get, then of course you don't need `sarge`. Let's look at more demanding uses next.

2.2.1 Finding commands under Windows

In versions 0.1.1 and earlier, `sarge`, like `subprocess`, did not do anything special to find the actual executable to run – it was expected to be found in the current directory or the path. Specifically, `PATHEXT` was not supported: where you might type `yada` in a command shell and have it run `python yada.py` because `.py` is in the `PATHEXT` environment variable and Python is registered to handle files with that extension, neither `subprocess` (with `shell=False`) nor `sarge` did this. You needed to specify the executable name explicitly in the command passed to `sarge`.

In 0.1.2 and later versions, `sarge` has improved command-line handling. The "which" functionality has been back-ported from Python 3.3, which takes care of using `PATHEXT` to resolve a command `yada` as `c:\Tools\yada.py` where `c:\Tools` is on the PATH and `yada.py` is in there. In addition, `sarge` queries the registry to see which programs are associated with the extension, and updates the command line accordingly. Thus, a command line `foo bar` passed to `sarge` may actually result in `c:\Windows\py.exe c:\Tools\foo.py bar` being passed to `subprocess` (assuming the Python Launcher for Windows, `py.exe`, is associated with `.py` files).

This new functionality is not limited to Python scripts - it should work for any extensions which are in `PATHEXT` and have an ftype/assoc binding them to an executable through `shell`, `open` and `command` subkeys in the registry, and where the command line is of the form `"<path_to_executable>" "%1" %*` (this is the standard form used by several languages).

2.2.2 Chaining commands

It's easy to chain commands together with `sarge`. For example:

```
>>> run('echo "Hello,"; echo "world!"')
Hello,
world!
<sarge.Pipeline object at 0x247ed50>
```

whereas this would have been more involved if you were just using `subprocess`:

```
>>> call('echo "Hello,"'.split()); call('echo "world!"'.split())
"Hello,"
0
"world!"
0
```

You get two return codes, one for each command. The same information is available from `sarge`, in one place – the `Pipeline` instance that's returned from a `run()` call:

```
>>> run('echo "Hello,"; echo "world!"').returncodes
Hello,
world!
[0, 0]
```

The `returncodes` property of a `Pipeline` instance returns a list of the return codes of all the commands that were run, whereas the `returncode` property just returns the last element of this list. The `Pipeline` class defines a number of useful properties - see the reference for full details.

2.2.3 Handling user input safely

By default, `sarge` does not run commands via the shell. This means that wildcard characters in user input do not have potentially dangerous consequences:

```
>>> run('ls *.py')
ls: cannot access *.py: No such file or directory
<sarge.Pipeline object at 0x20f3dd0>
```

This behaviour helps to avoid shell injection attacks.

There might be circumstances where you need to use `shell=True`, in which case you should consider formatting your commands with placeholders and quoting any variable parts that you get from external sources (such as user input). Which brings us on to ...

2.2.4 Formatting commands with placeholders for safe usage

If you need to merge commands with external inputs (e.g. user inputs) and you want to prevent shell injection attacks, you can use the `shell_format()` function. This takes a format string, positional and keyword arguments and uses the new formatting (`str.format()`) to produce the result:

```
>>> from sarge import shell_format
>>> shell_format('ls {0}', '*.py')
"ls '*.py'"
```

Note how the potentially unsafe input has been quoted. With a safe input, no quoting is done:

```
>>> shell_format('ls {0}', 'test.py')
'ls test.py'
```

If you really want to prevent quoting, even for potentially unsafe inputs, just use the `s` conversion:

```
>>> shell_format('ls {0!s}', '*.py')
'ls *.py'
```

There is also a `shell_quote()` function which quotes potentially unsafe input:

```
>>> from sarge import shell_quote
>>> shell_quote('abc')
'abc'
>>> shell_quote('ab?')
"'ab?'"
>>> shell_quote('"ab?"')
'\'"ab?"\''
>>> shell_quote("'ab?'")
'"\'ab?\'"'
```

This function is used internally by `shell_format()`, so you shouldn't need to call it directly except in unusual cases.

2.3 Passing input data to commands

You can pass input to a command pipeline using the `input` keyword parameter to `run()`:

```
>>> from sarge import run
>>> p = run('cat|cat', input='foo')
foo>>>
```

Here's how the value passed as `input` is processed:

- Text is encoded to bytes using UTF-8, which is then wrapped in a `BytesIO` object.
- Bytes are wrapped in a `BytesIO` object.
- Starting with 0.1.2, if you pass an object with a `fileno` attribute, that will be called as a method and the resulting value will be passed to the `subprocess` layer. This would normally be a readable file descriptor.
- Other values (such as integers representing OS-level file descriptors, or special values like `subprocess.PIPE`) are passed to the `subprocess` layer as-is.

If the result of the above process is a `BytesIO` instance (or if you passed in a `BytesIO` instance), then `sarge` will spin up an internal thread to write the data to the child process when it is spawned. The reason for a separate thread is that if the child process consumes data slowly, or the size of data is large, then the calling thread would block for potentially long periods of time.

2.3.1 Passing input data to commands dynamically

Sometimes, you may want to pass quite a lot of data to a child process which is not conveniently available as a string, byte-string or a file, but which is generated in the parent process (the one using `sarge`) by some other means. Starting with 0.1.2, `sarge` facilitates this by supporting objects with `fileno()` attributes as described above, and includes a `Feeder` class which has a suitable `fileno()` implementation.

Creating and using a feeder is simple:

```
import sys
from sarge import Feeder, run

feeder = Feeder()
run([sys.executable, 'echoer.py'], input=feeder, async=True)
```

After this, you can feed data to the child process' `stdin` by calling the `feed()` method of the `Feeder` instance:

```
feeder.feed('Hello')
feeder.feed(b'Goodbye')
```

If you pass in text, it will be encoded to bytes using UTF-8.

Once you've finished with the feeder, you can close it:

```
feeder.close()
```

Depending on how quickly the child process consumes data, the thread calling `feed()` might block on I/O. If this is a problem, you can spawn a separate thread which does the feeding.

Here's a complete working example:

```
import os
import subprocess
import sys
import time
import sarge

try:
    text_type = unicode
except NameError:
    text_type = str

def main(args=None):
    feeder = sarge.Feeder()
    p = sarge.run([sys.executable, 'echoer.py'], input=feeder, async=True)
    try:
        lines = ('hello', 'goodbye')
        gen = iter(lines)
        while p.commands[0].returncode is None:
            try:
                data = next(gen)
            except StopIteration:
                break
            feeder.feed(data + '\n')
            p.commands[0].poll()
            time.sleep(0.05)    # wait for child to return echo
    finally:
        p.commands[0].terminate()
        feeder.close()

if __name__ == '__main__':
    try:
        rc = main()
    except Exception as e:
        print(e)
        rc = 9
    sys.exit(rc)
```

In the above example, the `echoer.py` script (included in the `sarge` source distribution, as it's part of the test suite) just reads lines from its `stdin`, duplicates and prints to its `stdout`. Since we passed in the strings `hello` and `goodbye`, the output from the script should be:

```
hello hello
goodbye goodbye
```

2.4 Chaining commands conditionally

You can use `&&` and `||` to chain commands conditionally using short-circuit Boolean semantics. For example:

```
>>> from sarge import run
>>> run('false && echo foo')
<sarge.Pipeline object at 0xb8dd50>
```

Here, `echo foo` wasn't called, because the `false` command evaluates to `False` in the shell sense (by returning an exit code other than zero). Conversely:

```
>>> run('false || echo foo')
foo
<sarge.Pipeline object at 0xa11d50>
```

Here, `foo` is output because we used the `||` condition; because the left- hand operand evaluates to `False`, the right-hand operand is evaluated (i.e. run, in this context). Similarly, using the `true` command:

```
>>> run('true && echo foo')
foo
<sarge.Pipeline object at 0xb8dd50>
>>> run('true || echo foo')
<sarge.Pipeline object at 0xa11d50>
```

2.5 Creating command pipelines

It's just as easy to construct command pipelines:

```
>>> run('echo foo | cat')
foo
<sarge.Pipeline object at 0xb8dd50>
>>> run('echo foo; echo bar | cat')
foo
bar
<sarge.Pipeline object at 0xa96c50>
```

2.6 Using redirection

You can also use redirection to files as you might expect. For example:

```
>>> run('echo foo | cat > /tmp/junk')
<sarge.Pipeline object at 0x24b3190>
^D (to exit Python)
$ cat /tmp/junk
foo
```

You can use `>`, `>>`, `2>`, `2>>` which all work as on Posix systems. However, you can't use `<` or `<<`.

To send things to the bit-bucket in a cross-platform way, you can do something like:

```
>>> run('echo foo | cat > %s' % os.devnull)
<sarge.Pipeline object at 0x2765b10>
```

2.7 Capturing `stdout` and `stderr` from commands

To capture output for commands, just pass a `Capture` instance for the relevant stream:

```
>>> from sarge import run, Capture
>>> p = run('echo foo; echo bar | cat', stdout=Capture())
>>> p.stdout.text
u'foo\nbar\n'
```

The `Capture` instance acts like a stream you can read from: it has `read()`, `readline()` and `readlines()` methods which you can call just like on any file-like object, except that they offer additional options through `block` and `timeout` keyword parameters.

As in the above example, you can use the `bytes` or `text` property of a `Capture` instance to read all the bytes or text captured. The latter just decodes the former using UTF-8 (the default encoding isn't used, because on Python 2.x, the default encoding isn't UTF-8 – it's ASCII).

There are some convenience functions – `capture_stdout()`, `capture_stderr()` and `capture_both()` – which work just like `run()` but capture the relevant streams to `Capture` instances, which can be accessed using the appropriate attribute on the `Pipeline` instance returned from the functions.

There are more convenience functions, `get_stdout()`, `get_stderr()` and `get_both()`, which work just like `capture_stdout()`, `capture_stderr()` and `capture_both()` respectively, but return the captured text. For example:

```
>>> from sarge import get_stdout
>>> get_stdout('echo foo; echo bar')
u'foo\nbar\n'
```

New in version 0.1.1: The `get_stdout()`, `get_stderr()` and `get_both()` functions were added. A `Capture` instance can capture output from one or more sub-process streams, and will create a thread for each such stream so that it can read all sub-process output without causing the sub-processes to block on their output I/O. However, if you use a `Capture`, you should be prepared either to consume what it's read from the sub-processes, or else be prepared for it all to be buffered in memory (which may be problematic if the sub-processes generate a *lot* of output).

2.8 Iterating over captures

You can iterate over `Capture` instances. By default you will get successive lines from the captured data, as bytes; if you want text, you can wrap with `io.TextIOWrapper`. Here's an example using Python 3.2:

```
>>> from sarge import capture_stdout
>>> p = capture_stdout('echo foo; echo bar')
>>> for line in p.stdout: print(repr(line))
...
b'foo\n'
b'bar\n'
>>> p = capture_stdout('echo bar; echo baz')
>>> from io import TextIOWrapper
>>> for line in TextIOWrapper(p.stdout): print(repr(line))
...
'bar\n'
'baz\n'
```

This works the same way in Python 2.x. Using Python 2.7:

```
>>> from sarge import capture_stdout
>>> p = capture_stdout('echo foo; echo bar')
>>> for line in p.stdout: print(repr(line))
...
'foo\n'
'bar\n'
>>> p = capture_stdout('echo bar; echo baz')
>>> from io import TextIOWrapper
>>> for line in TextIOWrapper(p.stdout): print(repr(line))
...
u'bar\n'
u'baz\n'
```

2.9 Interacting with child processes

Sometimes you need to interact with a child process in an interactive manner. To illustrate how to do this, consider the following simple program, named `receiver`, which will be used as the child process:

```
#!/usr/bin/env python
import sys

def main(args=None):
    while True:
        user_input = sys.stdin.readline().strip()
        if not user_input:
            break
        s = 'Hi, %s!\n' % user_input
        sys.stdout.write(s)
        sys.stdout.flush() # need this when run as a subprocess

if __name__ == '__main__':
    sys.exit(main())
```

This just reads lines from the input and echoes them back as a greeting. If we run it interactively:

```
$ ./receiver
Fred
Hi, Fred!
Jim
Hi, Jim!
Sheila
Hi, Sheila!
```

The program exits on seeing an empty line.

We can now show how to interact with this program from a parent process:

```
>>> from sarge import Command, Capture
>>> from subprocess import PIPE
>>> p = Command('./receiver', stdout=Capture(buffer_size=1))
>>> p.run(input=PIPE, async=True)
Command('./receiver')
>>> p.stdin.write('Fred\n')
>>> p.stdout.readline()
'Hi, Fred!\n'
>>> p.stdin.write('Jim\n')
>>> p.stdout.readline()
'Hi, Jim!\n'
>>> p.stdin.write('Sheila\n')
>>> p.stdout.readline()
'Hi, Sheila!\n'
>>> p.stdin.write('\n')
>>> p.stdout.readline()
''
>>> p.returncode
>>> p.wait()
0
```

The `p.returncode` didn't print anything, indicating that the return code was `None`. This means that although the child process has exited, it's still a zombie because we haven't "reaped" it by making a call to `wait()`. Once that's done, the zombie disappears and we get the return code.

2.9.1 Buffering issues

From the point of view of buffering, note that two elements are needed for the above example to work:

- We specify `buffer_size=1` in the Capture constructor. Without this, data would only be read into the Capture's queue after an I/O completes – which would depend on how many bytes the Capture reads at a time. You can also pass a `buffer_size=-1` to indicate that you want to use line- buffering, i.e. read a line at a time from the child process. (This may only work as expected if the child process flushes its outbut buffers after every line.)
- We make a `flush` call in the `receiver` script, to ensure that the pipe is flushed to the capture queue. You could avoid the `flush` call in the above example if you used `python -u receiver` as the command (which runs the script unbuffered).

This example illustrates that in order for this sort of interaction to work, you need cooperation from the child process. If the child process has large output buffers and doesn't flush them, you could be kept waiting for input until the buffers fill up or a flush occurs.

If a third party package you're trying to interact with gives you buffering problems, you may or may not have luck (on Posix, at least) using the `unbuffer` utility from the `expect-dev` package (do a Web search to find it). This invokes a program directing its output to a pseudo-tty device which gives line buffering behaviour. This doesn't always work, though :-(

2.9.2 Looking for specific patterns in child process output

You can look for specific patterns in the output of a child process, by using the `expect()` method of the `Capture` class. This takes a string, bytestring or regular expression pattern object and a timeout, and either returns a regular expression match object (if a match was found in the specified timeout) or `None` (if no match was found in the specified timeout). If you pass in a bytestring, it will be converted to a regular expression pattern. If you pass in text, it will be encoded to bytes using the `utf-8` codec and then to a regular expression pattern. This pattern will be used to look for a match (using `search`). If you pass in a regular expression pattern, make sure it is meant for bytes rather than text (to avoid `TypeError` on Python 3.x). You may also find it useful to specify `re.MULTILINE` in the pattern flags, so that you can match using `^` and `$` at line boundaries. Note that on Windows, you may need to use `\r?$` to match ends of lines, as `$` matches Unix newlines (LF) and not Windows newlines (CRLF). New in version 0.1.1: The `expect` method was added. To illustrate usage of `Capture.expect()`, consider the program `lister.py` (which is provided as part of the source distribution, as it's used in the tests). This prints `line 1`, `line 2` etc. indefinitely with a configurable delay, flushing its output stream after each line. We can capture the output from a run of `lister.py`, ensuring that we use line-buffering in the parent process:

```
>>> from sarge import Capture, run
>>> c = Capture(buffer_size=-1)     # line-buffering
>>> p = run('python lister.py -d 0.01', async=True, stdout=c)
>>> m = c.expect('^line 1$')
>>> m.span()
(0, 6)
>>> m = c.expect('^line 5$')
>>> m.span()
(28, 34)
>>> m = c.expect('^line 1.*$')
>>> m.span()
(63, 70)
>>> c.close(True)           # close immediately, discard any unread input
>>> p.commands[0].kill()    # kill the subprocess
>>> c.bytes[63:70]
'line 10'
>>> m = c.expect(r'^line 1\d\d$')
>>> m.span()
```

```
(783, 791)
>>> c.bytes[783:791]
'line 100'
```

2.9.3 Displaying progress as a child process runs

You can display progress as a child process runs, assuming that its output allows you to track that progress. Consider the following script, `progress.py`:

```
import optparse # because of 2.6 support
import sys
import threading
import time

from sarge import capture_stdout

def progress(capture, options):
    lines_seen = 0
    messages = {
        'line 25\n': 'Getting going ...\n',
        'line 50\n': 'Well on the way ...\n',
        'line 75\n': 'Almost there ...\n',
    }
    while True:
        s = capture.readline()
        if not s and lines_seen:
            break
        if options.dots:
            sys.stderr.write('.')
        else:
            msg = messages.get(s)
            if msg:
                sys.stderr.write(msg)
        lines_seen += 1
    if options.dots:
        sys.stderr.write('\n')
    sys.stderr.write('Done - %d lines seen.\n' % lines_seen)

def main():
    parser = optparse.OptionParser()
    parser.add_option('-n', '--no-dots', dest='dots', default=True,
                      action='store_false', help='Show dots for progress')
    options, args = parser.parse_args()
    p = capture_stdout('python lister.py -d 0.1 -c 100', async=True)
    t = threading.Thread(target=progress, args=(p.stdout, options))
    t.start()
    while(p.returncodes[0] is None):
        # We could do other useful work here. If we have no useful
        # work to do here, we can call readline() and process it
        # directly in this loop, instead of creating a thread to do it in.
        p.commands[0].poll()
        time.sleep(0.05)
    t.join()

if __name__ == '__main__':
    sys.exit(main())
```

When this is run without the `--no-dots` argument, you should see the following:

```
$ python progress.py
.......................................................  (100 dots printed)
Done - 100 lines seen.
```

If run with the `--no-dots` argument, you should see:

```
$ python progress.py --no-dots
Getting going ...
Well on the way ...
Almost there ...
Done - 100 lines seen.
```

with short pauses between the output lines.

2.9.4 Direct terminal usage

Some programs don't work through their `stdin`/`stdout`/`stderr` streams, instead opting to work directly with their controlling terminal. In such cases, you can't work with these programs using `sarge`; you need to use a pseudo-terminal approach, such as is provided by (for example) pexpect. `Sarge` works within the limits of the `subprocess` module, which means sticking to `stdin`, `stdout` and `stderr` as ordinary streams or pipes (but not pseudo-terminals).

Examples of programs which work directly through their controlling terminal are `ftp` and `ssh` - the password prompts for these programs are generally always printed to the controlling terminal rather than `stdout` or `stderr`.

2.10 Environments

In the `subprocess.Popen` constructor, the `env` keyword argument, if supplied, is expected to be the *complete* environment passed to the child process. This can lead to problems on Windows, where if you don't pass the `SYSTEMROOT` environment variable, things can break. With `sarge`, it's assumed that anything you pass in `env` is *added* to the contents of `os.environ`. This is almost always what you want – after all, in a Posix shell, the environment is generally inherited with certain additions for a specific command invocation.

Note: On Python 2.x on Windows, environment keys and values must be of type `str` - Unicode values will cause a `TypeError`. Be careful of this if you use `from __future__ import unicode_literals`. For example, the test harness for sarge uses Unicode literals on 2.x, necessitating the use of different logic for 2.x and 3.x:

```
if PY3:
    env = {'FOO': 'BAR'}
else:
    # Python 2.x wants native strings, at least on Windows
    env = { b'FOO': b'BAR' }
```

2.11 Working directory and other options

You can set the working directory for a `Command` or `Pipeline` using the `cwd` keyword argument to the constructor, which is passed through to the subprocess when it's created. Likewise, you can use the other keyword arguments which are accepted by the `subprocess.Popen` constructor.

Avoid using the `stdin` keyword argument – instead, use the `input` keyword argument to the `Command.run()` and `Pipeline.run()` methods, or the `run()`, `capture_stdout()`, `capture_stderr()`, and `capture_both()` functions. The `input` keyword makes it easier for you to pass literal text or byte data.

2.12 Unicode and bytes

All data between your process and sub-processes is communicated as bytes. Any text passed as input to `run()` or a `run()` method will be converted to bytes using UTF-8 (the default encoding isn't used, because on Python 2.x, the default encoding isn't UTF-8 – it's ASCII).

As `sarge` requires Python 2.6 or later, you can use `from __future__ import unicode_literals` and byte literals like `b'foo'` so that your code looks and behaves the same under Python 2.x and Python 3.x. (See the note on using native string keys and values in *Environments*.)

As mentioned above, `Capture` instances return bytes, but you can wrap with `io.TextIOWrapper` if you want text.

2.13 Use as context managers

The `Capture` and `Pipeline` classes can be used as context managers:

```
>>> with Capture() as out:
...     with Pipeline('cat; echo bar | cat', stdout=out) as p:
...         p.run(input='foo\n')
...
<sarge.Pipeline object at 0x7f3320e94310>
>>> out.read().split()
['foo', 'bar']
```

2.14 Synchronous and asynchronous execution of commands

By default. commands passed to `run()` run synchronously, i.e. all commands run to completion before the call returns. However, you can pass `async=True` to run, in which case the call returns a `Pipeline` instance before all the commands in it have run. You will need to call `wait()` or `close()` on this instance when you are ready to synchronise with it; this is needed so that the sub processes can be properly disposed of (otherwise, you will leave zombie processes hanging around, which show up, for example, as `<defunct>` on Linux systems when you run `ps -ef`). Here's an example:

```
>>> p = run('echo foo|cat|cat|cat|cat', async=True)
>>> foo
```

Here, `foo` is printed to the terminal by the last `cat` command, but all the sub-processes are zombies. (The `run` function returned immediately, so the interpreter got to issue the `>>>` ` prompt *before* the ``foo` output was printed.)

In another terminal, you can see the zombies:

```
$ ps -ef | grep defunct | grep -v grep
vinay     4219  4217  0 19:27 pts/0    00:00:00 [echo] <defunct>
vinay     4220  4217  0 19:27 pts/0    00:00:00 [cat] <defunct>
vinay     4221  4217  0 19:27 pts/0    00:00:00 [cat] <defunct>
vinay     4222  4217  0 19:27 pts/0    00:00:00 [cat] <defunct>
vinay     4223  4217  0 19:27 pts/0    00:00:00 [cat] <defunct>
```

Now back in the interactive Python session, we call `close()` on the pipeline:

```
>>> p.close()
```

and now, in the other terminal, look for defunct processes again:

```
$ ps -ef | grep defunct | grep -v grep
$
```

No zombies found :-)

2.15 About threading and forking on Posix

If you run commands asynchronously by using `&` in a command pipeline, then a thread is spawned to run each such command asynchronously. Remember that thread scheduling behaviour can be unexpected – things may not always run in the order you expect. For example, the command line:

```
echo foo & echo bar & echo baz
```

should run all of the `echo` commands concurrently as far as possible, but you can't be sure of the exact sequence in which these commands complete – it may vary from machine to machine and even from one run to the next. This has nothing to do with `sarge` – there are no guarantees with just plain Bash, either.

On Posix, `subprocess` uses `os.fork()` to create the child process, and you may see dire warnings on the Internet about mixing threads, processes and `fork()`. It *is* a heady mix, to be sure: you need to understand what's going on in order to avoid nasty surprises. If you run into any such, it may be hard to get help because others can't reproduce the problems. However, that's no reason to shy away from providing the functionality altogether. Such issues do not occur on Windows, for example: because Windows doesn't have a `fork()` system call, child processes are created in a different way which doesn't give rise to the issues which sometimes crop up in a Posix environment.

For an exposition of the sort of things which might bite you if you are using locks, threading and `fork()` on Posix, see this post.

Other resources on this topic:

- http://bugs.python.org/issue6721

Please report any problems you find in this area (or any other) either via the mailing list or the issue tracker.

2.16 Next steps

You might find it helpful to look at information about how `sarge` works internally – *Under the hood* – or peruse the *API Reference*.

CHAPTER 3

Under the hood

This is the section where some description of how `sarge` works internally will be provided, as and when time permits.

3.1 How capturing works

This section describes how `Capture` is implemented.

3.1.1 Basic approach

A `Capture` consists of a queue, some output streams from sub-processes, and some threads to read from those streams into the queue. One thread is created for each stream, and the thread exits when its stream has been completely read. When you read from a `Capture` instance using methods like `read()`, `readline()` and `readlines()`, you are effectively reading from the queue.

3.1.2 Blocking and timeouts

Each of the `read()`, `readline()` and `readlines()` methods has optional `block` and `timeout` keyword arguments. These default to `True` and `None` respectively, which means block indefinitely until there's some data – the standard behaviour for file-like objects. However, these can be overridden internally in a couple of ways:

- The `Capture` constructor takes an optional `timeout` keyword argument. This defaults to `None`, but if specified, that's the timeout used by the `readXXX` methods unless you specify values in the method calls. If `None` is specified in the constructor, the module attribute `default_capture_timeout` is used, which is currently set to 0.02 seconds. If you need to change this default, you can do so before any `Capture` instances are created (or just provide an alternative default in every `Capture` creation).
- If all streams feeding into the capture have been completely read, then `block` is always set to `False`.

3.1.3 Implications when handling large amounts of data

There shouldn't be any special implications of handling large amounts of data, other than buffering, buffer sizes and memory usage (which you would have to think about anyway). Here's an example of piping a 20MB file into a capture across several process boundaries:

```
$ ls -l random.bin
-rw-rw-r-- 1 vinay vinay 20971520 2012-01-17 17:57 random.bin
$ python
[snip]
>>> from sarge import run, Capture
>>> p = run('cat random.bin|cat|cat|cat|cat|cat', stdout=Capture(), async=True)
>>> for i in range(8):
...     data = p.stdout.read(2621440)
...     print('Read chunk %d: %d bytes' % (i, len(data)))
...
Read chunk 0: 2621440 bytes
Read chunk 1: 2621440 bytes
Read chunk 2: 2621440 bytes
Read chunk 3: 2621440 bytes
Read chunk 4: 2621440 bytes
Read chunk 5: 2621440 bytes
Read chunk 6: 2621440 bytes
Read chunk 7: 2621440 bytes
>>> p.stdout.read()
''
```

3.2 Swapping output streams

A new constant, `STDERR`, is defined by `sarge`. If you specify `stdout=STDERR`, this means that you want the child process `stdout` to be the same as its `stderr`. This is analogous to the core functionality in `subprocess.Popen` where you can specify `stderr=STDOUT` to have the child process `stderr` be the same as its `stdout`. The use of this constant also allows you to swap the child's `stdout` and `stderr`, which can be useful in some cases.

This functionality works through a class `sarge.Popen` which subclasses `subprocess.Popen` and overrides the internal `_get_handles` method to work the necessary magic – which is to duplicate, close and swap handles as needed.

3.3 How shell quoting works

The `shell_quote()` function works as follows. Firstly, an empty string is converted to `''`. Next, a check is made to see if the string has already been quoted (i.e. it begins and ends with the `'` character), and if so, it is returned enclosed in `"` and with any contained " characters escaped with a backslash. Otherwise, it's bracketed with the `'` character and every internal instance of `'` is replaced with `'"'"'`.

3.4 How shell command formatting works

This is inspired by Nick Coghlan's shell_command project. An internal `ShellFormatter` class is derived from `string.Formatter` and overrides the `string.Formatter.convert_field()` method to provide quoting for placeholder values. This formatter is simpler than Nick's in that it forces you to explicitly provide the indices of positional arguments: You have to use e.g. `'cp {0} {1}` instead of `cp {} {}`. This avoids the need to keep an internal counter in the formatter, which would make its implementation be not thread-safe without additional work.

3.5 How command parsing works

Internally sarge uses a simple recursive descent parser to parse commands. A simple BNF grammar for the parser would be:

```
<list> ::= <pipeline> ((";" | "&") <pipeline>)*
<pipeline> ::= <logical> (("&&" | "||") <logical>)*
<logical> ::= (<command> (("|" | "|&") <command>)*) | "(" <list> ")"
<command> ::= <command-part>+
<command-part> ::= WORD ((<NUM>)? (">" | ">>") (<WORD> | ("&" <NUM>)))*
```

where WORD and NUM are terminal tokens with the meanings you would expect.

The parser constructs a parse tree, which is used internally by the Pipeline class to manage the running of the pipeline.

The standard library's shlex module contains a class which is used for lexical scanning. Since the shlex.shlex class is not able to provide the needed functionality, sarge includes a module, shlext, which defines a subclass, shell_shlex, which provides the necessary functionality. This is not part of the public API of sarge, though it has been submitted as an enhancement on the Python issue tracker.

3.6 Thread debugging

Sometimes, you can get deadlocks even though you think you've taken sufficient measures to avoid them. To help identify where deadlocks are occurring, the sarge source distribution includes a module, stack_tracer, which is based on MIT-licensed code by László Nagy in an ActiveState recipe. To see how it's invoked, you can look at the sarge test harness test_sarge.py – this is set to invoke the tracer if the TRACE_THREADS variable is set (which it is, by default). If the unit tests hang on your system, then the threads-X.Y.log file will show where the deadlock is (just look and see what all the threads are waiting for).

3.7 Future changes

At the moment, if a Capture is used, it will read from its sub-process output streams into a queue, which can then be read by your code. If you don't read from the Capture in a timely fashion, a lot of data could potentially be buffered in memory – the same thing that happens when you use subprocess.Popen.communicate(). There might be added some means of "turning the tap off", i.e. pausing the reader threads so that the capturing threads stop reading from the sub-process streams. This will, of course, cause those sub-processes to block on their I/O, so at some point the tap would need to be turned back on. However, such a facility would afford better sub-process control in some scenarios.

3.8 Next steps

You might find it helpful to look at the *API Reference*.

CHAPTER 4

API Reference

This is the place where the functions and classes in `sarge's` public API are described.

4.1 Attributes

`default_capture_timeout`
This is the default timeout which will be used by `Capture` instances when you don't specify one in the `Capture` constructor. This is currently set to **0.02 seconds**.

4.2 Functions

`run`(*command*, *input=None*, *async=False*, ***kwargs*)
This function is a convenience wrapper which constructs a `Pipeline` instance from the passed parameters, and then invokes `run()` and `close()` on that instance.

Parameters

- **command** (*str*) – The command(s) to run.
- **input** (*Text, bytes or a file-like object containing bytes (not text).*) – Input data to be passed to the command(s). If text is passed, it's converted to `bytes` using the default encoding. The bytes are converted to a file-like object (a `BytesIO` instance). If a value such as a file-like object, integer file descriptor or special value like `subprocess.PIPE` is passed, it is passed through unchanged to `subprocess.Popen`.
- **kwargs** – Any keyword parameters which you might want to pass to the wrapped `Pipeline` instance.

Returns The created `Pipeline` instance.

`capture_stdout`(*command*, *input=None*, *async=False*, ***kwargs*)
This function is a convenience wrapper which does the same as `run()` while capturing the `stdout` of the subprocess(es). This captured output is available through the `stdout` attribute of the return value from this function.

Parameters

- **command** – As for `run()`.

- **input** – As for `run()`.
- **kwargs** – As for `run()`.

Returns As for `run()`.

`get_stdout`(*command*, *input=None*, *async=False*, ***kwargs*)

This function is a convenience wrapper which does the same as `capture_stdout()` but also returns the text captured. Use this when you know the output is not voluminous, so it doesn't matter that it's buffered in memory.

Parameters

- **command** – As for `run()`.
- **input** – As for `run()`.
- **kwargs** – As for `run()`.

Returns The captured text.

New in version 0.1.1.

`capture_stderr`(*command*, *input=None*, *async=False*, ***kwargs*)

This function is a convenience wrapper which does the same as `run()` while capturing the `stderr` of the subprocess(es). This captured output is available through the `stderr` attribute of the return value from this function.

Parameters

- **command** – As for `run()`.
- **input** – As for `run()`.
- **kwargs** – As for `run()`.

Returns As for `run()`.

`get_stderr`(*command*, *input=None*, *async=False*, ***kwargs*)

This function is a convenience wrapper which does the same as `capture_stderr()` but also returns the text captured. Use this when you know the output is not voluminous, so it doesn't matter that it's buffered in memory.

Parameters

- **command** – As for `run()`.
- **input** – As for `run()`.
- **kwargs** – As for `run()`.

Returns The captured text.

New in version 0.1.1.

`capture_both`(*command*, *input=None*, *async=False*, ***kwargs*)

This function is a convenience wrapper which does the same as `run()` while capturing the `stdout` and the `stderr` of the subprocess(es). This captured output is available through the `stdout` and `stderr` attributes of the return value from this function.

Parameters

- **command** – As for `run()`.
- **input** – As for `run()`.
- **kwargs** – As for `run()`.

Returns As for `run()`.

`get_both`(*command*, *input=None*, *async=False*, ***kwargs*)
This function is a convenience wrapper which does the same as `capture_both()` but also returns the text captured. Use this when you know the output is not voluminous, so it doesn't matter that it's buffered in memory.

Parameters

- **command** – As for `run()`.
- **input** – As for `run()`.
- **kwargs** – As for `run()`.

Returns The captured text as a 2-element tuple, with the `stdout` text in the first element and the `stderr` text in the second.

New in version 0.1.1.

`shell_quote`(*s*)
Quote text so that it is safe for Posix command shells.

For example, "*.py" would be converted to "'*.py'". If the text is considered safe it is returned unquoted.

Parameters **s** (*str, or unicode on 2.x*) – The value to quote

Returns A safe version of the input, from the point of view of Posix command shells

Return type The passed-in type

`shell_format`(*fmt*, **args*, ***kwargs*)
Format a shell command with format placeholders and variables to fill those placeholders.

Note: you must specify positional parameters explicitly, i.e. as {0}, {1} instead of {}, {}. Requiring the formatter to maintain its own counter can lead to thread safety issues unless a thread local is used to maintain the counter. It's not that hard to specify the values explicitly yourself :-)

Parameters

- **fmt** (*str, or unicode on 2.x*) – The shell command as a format string. Note that you will need to double up braces you want in the result, i.e. { -> {{ and } -> }}, due to the way `str.format()` works.
- **args** – Positional arguments for use with `fmt`.
- **kwargs** – Keyword arguments for use with `fmt`.

Returns The formatted shell command, which should be safe for use in shells from the point of view of shell injection.

Return type The type of `fmt`.

4.3 Classes

class `Command`(*args*, ***kwargs*)
This represents a single command to be spawned as a subprocess.

Parameters

- **args** (str if `shell=True`, or an array of str) – The command to run.
- **kwargs** – Any keyword parameters you might pass to `Popen`, other than `stdin` (for which, you need to see the `input` argument of `run()`).

`run`(*input=None*, *async=False*)
Run the command.

Parameters

- **input** (*Text, bytes or a file-like object containing bytes.*) – Input data to be passed to the command. If text is passed, it's converted to `bytes` using the default encoding. The bytes are converted to a file-like object (a `BytesIO` instance). The contents of the file-like object are written to the `stdin` stream of the sub-process.
- **async** (*bool*) – If `True`, the command is run asynchronously – that is to say, `wait()` is not called on the underlying `Popen` instance.

`wait`()
Wait for the command's underlying sub-process to complete.

`terminate`()
Terminate the command's underlying sub-process by calling `subprocess.Popen.terminate()` on it. New in version 0.1.1.

`kill`()
Kill the command's underlying sub-process by calling `subprocess.Popen.kill()` on it. New in version 0.1.1.

`poll`()
Poll the command's underlying sub-process by calling `subprocess.Popen.poll()` on it. Returns the result of that call. New in version 0.1.1.

class `Pipeline`(*source*, *posix=True*, ***kwargs*)
This represents a set of commands which need to be run as a unit.

Parameters

- **source** (*str*) – The source text with the command(s) to run.
- **posix** (*bool*) – Whether the source will be parsed using Posix conventions.
- **kwargs** – Any keyword parameters you would pass to `subprocess.Popen`, other than `stdin` (for which, you need to use the `input` parameter of the `run()` method instead). You can pass `Capture` instances for `stdout` and `stderr` keyword arguments, which will cause those streams to be captured to those instances.

`run`(*input=None*, *async=False*)
Run the pipeline.

Parameters

- **input** – The same as for the `Command.run()` method.
- **async** – The same as for the `Command.run()` method. Note that parts of the pipeline may specify synchronous or asynchronous running – this flag refers to the pipeline as a whole.

`wait`()
Wait for all command sub-processes to finish.

`close`()
Wait for all command sub-processes to finish, and close all opened streams.

`returncodes`
A list of the return codes of all sub-processes which were actually run.

`returncode`
The return code of the last sub-process which was actually run.

commands
The `Command` instances which were actually created.

class `Capture`(*timeout=None*, *buffer_size=0*)
A class which allows an output stream from a sub-process to be captured.

Parameters

- **timeout** (*float*) – The default timeout, in seconds. Note that you can override this in particular calls to read input. If `None` is specified, the value of the module attribute `default_capture_timeout` is used instead.
- **buffer_size** (*int*) – The buffer size to use when reading from the underlying streams. If not specified or specified as zero, a 4K buffer is used. For interactive applications, use a value of 1.

read(*size=-1*, *block=True*, *timeout=None*)
Like the `read` method of any file-like object.

Parameters

- **size** (*int*) – The number of bytes to read. If not specified, the intent is to read the stream until it is exhausted.
- **block** (*bool*) – Whether to block waiting for input to be available,
- **timeout** (*float*) – How long to wait for input. If `None`, use the default timeout that this instance was initialised with. If the result is `None`, wait indefinitely.

readline(*size=-1*, *block=True*, *timeout=None*)
Like the `readline` method of any file-like object.

Parameters

- **size** – As for the `read()` method.
- **block** – As for the `read()` method.
- **timeout** – As for the `read()` method.

readlines(*sizehint=-1*, *block=True*, *timeout=None*)
Like the `readlines` method of any file-like object.

Parameters

- **sizehint** – As for the `read()` method's `size`.
- **block** – As for the `read()` method.
- **timeout** – As for the `read()` method.

expect(*string_or_pattern*, *timeout=None*)
This looks for a pattern in the captured output stream. If found, it returns immediately; otherwise, it will block until the timeout expires, waiting for a match as bytes from the captured stream continue to be read.

Parameters

- **string_or_pattern** – A string or pattern representing a regular expression to match. Note that this needs to be a bytestring pattern if you pass a pattern in; if you pass in text, it is converted to bytes using the `utf-8` codec and then to a pattern used for matching (using `search`). If you pass in a pattern, you may want to ensure that its flags include `re/MULTILINE` so that you can make use of `^` and `$` in matching line boundaries. Note that on Windows, you may need to use `\r?$` to match ends of lines, as `$` matches Unix newlines (LF) and not Windows newlines (CRLF).

- **timeout** – If not specified, the module's `default_expect_timeout` is used.

Returns A regular expression match instance, if a match was found within the specified timeout, or `None` if no match was found.

close(stop_threads=False):
Close the capture object. By default, this waits for the threads which read the captured streams to terminate (which may not happen unless the child process is killed, and the streams read to exhaustion). To ensure that the threads are stopped immediately, specify `True` for the `stop_threads` parameter, which will asks the threads to terminate immediately. This may lead to losing data from the captured streams which has not yet been read.

class **Popen**
This is a subclass of `subprocess.Popen` which is provided mainly to allow a process' `stdout` to be mapped to its `stderr`. The standard library version allows you to specify `stderr=STDOUT` to indicate that the standard error stream of the sub-process be the same as its standard output stream. However. there's no facility in the standard library to do `stdout=STDERR` – but it *is* provided in this subclass.

In fact, the two streams can be swapped by doing `stdout=STDERR, stderr=STDOUT` in a call. The `STDERR` value is defined in `sarge` as an integer constant which is understood by `sarge` (much as `STDOUT` is an integer constant which is understood by `subprocess`).

4.4 Shell syntax understood by `sarge`

Shell commands are parsed by `sarge` using a simple parser.

4.4.1 Command syntax

The `sarge` parser looks for commands which are separated by `;` and `&`:

```
echo foo; echo bar & echo baz
```

which means to run *echo foo*, wait for its completion, and then run `echo bar` and then `echo baz` without waiting for `echo bar` to complete.

The commands which are separated by `&` and `;` are *conditional* commands, of the form:

```
a && b
```

or:

```
c || d
```

Here, command `b` is executed only if `a` returns success (i.e. a return code of 0), whereas `d` is only executed if `c` returns failure, i.e. a return code other than 0. Of course, in practice all of `a`, `b`, `c` and `d` could have arguments, not shown above for simplicity's sake.

Each operand on either side of `&&` or `||` could also consist of a pipeline – a set of commands connected such that the output streams of one feed into the input stream of another. For example:

```
echo foo | cat
```

or:

```
command-a |& command-b
```

where the use of `|` indicates that the standard output of `echo foo` is piped to the input of `cat`, whereas the standard error of `command-a` is piped to the input of `command-b`.

4.4.2 Redirections

The sarge parser also understands redirections such as are shown in the following examples:

```
command arg-1 arg-2 > stdout.txt
command arg-1 arg-2 2> stderr.txt
command arg-1 arg-2 2>&1
command arg-1 arg-2 >&2
```

In general, file descriptors other than 1 and 2 are not allowed, as the functionality needed to provided them (dup2) is not properly supported on Windows. However, an esoteric special case *is* recognised:

```
echo foo | tee stdout.log 3>&1 1>&2 2>&3 | tee stderr.log > /dev/null
```

This redirection construct will put foo in both stdout.log *and* stderr.log. The effect of this construct is to swap the standard output and standard error streams, using file descriptor 3 as a temporary as in the code analogue for swapping variables a and b using temporary variable c:

```
c = a
a = b
b = c
```

This is recognised by sarge and used to swap the two streams, though it doesn't literally use file descriptor 3, instead using a cross-platform mechanism to fulfill the requirement.

You can see this post for a longer explanation of this somewhat esoteric usage of redirection.

4.5 Next steps

You might find it helpful to look at the mailing list.