Ptyprocess Documentation

Release 0.7

Thomas Kluyver

Sep 12, 2023

Contents

1 Ptyprocess API 3

2 What is a pty? 7

3 Indices and tables 9

Python Module Index 11

Index 13

Launch a subprocess in a pseudo terminal (pty), and interact with both the process and its pty.

Sometimes, piping stdin and stdout is not enough. There might be a password prompt that doesn't read from stdin, output that changes when it's going to a pipe rather than a terminal, or curses-style interfaces that rely on a terminal. If you need to automate these things, running the process in a pseudo terminal (pty) is the answer.

Interface:

```
from ptyprocess import PtyProcessUnicode
p = PtyProcessUnicode.spawn(['python'])
p.read(20)
p.write('6+6\n')
p.read(20)
```

Contents:

CHAPTER 1

Ptyprocess API

class ptyprocess.**PtyProcess**(*pid*, *fd*)

This class represents a process running in a pseudoterminal.

The main constructor is the *spawn()* classmethod.

classmethod spawn(*argv*, *cwd=None*, *env=None*, *echo=True*, *preexec_fn=None*, *dimensions=(24, 80)*, *pass_fds=()*)

Start the given command in a child process in a pseudo terminal.

This does all the fork/exec type of stuff for a pty, and returns an instance of PtyProcess.

If preexec_fn is supplied, it will be called with no arguments in the child process before exec-ing the specified command. It may, for instance, set signal handlers to SIG_DFL or SIG_IGN.

Dimensions of the psuedoterminal used for the subprocess can be specified as a tuple (rows, cols), or the default (24, 80) will be used.

By default, all file descriptors except 0, 1 and 2 are closed. This behavior can be overridden with pass_fds, a list of file descriptors to keep open between the parent and the child.

read(*size=1024*)

Read and return at most `size` bytes from the pty.

Can block if there is nothing to read. Raises `EOFError` if the terminal was closed.

Unlike Pexpect's `read_nonblocking` method, this doesn't try to deal with the vagaries of EOF on platforms that do strange things, like IRIX or older Solaris systems. It handles the errno=EIO pattern used on Linux, and the empty-string return used on BSD platforms and (seemingly) on recent Solaris.

readline()

Read one line from the pseudoterminal, and return it as unicode.

Can block if there is nothing to read. Raises `EOFError` if the terminal was closed.

write(*s*, *flush=True*)

Write bytes to the pseudoterminal.

Returns the number of bytes written.

sendcontrol(*char*)
Helper method for sending control characters to the terminal.

For example, to send Ctrl-G (ASCII 7, bell, `'\a'`):

```
child.sendcontrol('g')
```

See also, *sendintr()* and *sendeof()*.

sendeof()
Sends an EOF (typically Ctrl-D) through the terminal.

This sends a character which causes the pending parent output buffer to be sent to the waiting child program without waiting for end-of-line. If it is the first character of the line, the read() in the user program returns 0, which signifies end-of-file. This means to work as expected a sendeof() has to be called at the beginning of a line. This method does not send a newline. It is the responsibility of the caller to ensure the eof is sent at the beginning of a line.

sendintr()
Send an interrupt character (typically Ctrl-C) through the terminal.

This will normally trigger the kernel to send SIGINT to the current foreground process group. Processes can turn off this translation, in which case they can read the raw data sent, e.g. `b'\x03'` for Ctrl-C.

See also the *kill()* method, which sends a signal directly to the immediate child process in the terminal (which is not necessarily the foreground process).

getecho()
Returns True if terminal echo is on, or False if echo is off.

Child applications that are expecting you to enter a password often disable echo. See also *waitnoecho()*.

Not supported on platforms where `isatty()` returns False.

waitnoecho(*timeout=None*)
Wait until the terminal ECHO flag is set False.

This returns True if the echo mode is off, or False if echo was not disabled before the timeout. This can be used to detect when the child is waiting for a password. Usually a child application will turn off echo mode when it is waiting for the user to enter a password. For example, instead of expecting the "password:" prompt you can wait for the child to turn echo off:

```
p = pexpect.spawn('ssh user@example.com')
p.waitnoecho()
p.sendline(mypassword)
```

If `timeout=None` then this method to block until ECHO flag is False.

setecho(*state*)
Enable or disable terminal echo.

Anything the child sent before the echo will be lost, so you should be sure that your input buffer is empty before you call setecho(). For example, the following will work as expected:

```
p = pexpect.spawn('cat') # Echo is on by default.
p.sendline('1234') # We expect see this twice from the child...
p.expect(['1234']) # ... once from the tty echo...
p.expect(['1234']) # ... and again from cat itself.
p.setecho(False) # Turn off tty echo
p.sendline('abcd') # We will set this only once (echoed by cat).
```

(continues on next page)

(continued from previous page)

```
p.sendline('wxyz') # We will set this only once (echoed by cat)
p.expect(['abcd'])
p.expect(['wxyz'])
```

The following WILL NOT WORK because the lines sent before the setecho will be lost:

```
p = pexpect.spawn('cat')
p.sendline('1234')
p.setecho(False) # Turn off tty echo
p.sendline('abcd') # We will set this only once (echoed by cat).
p.sendline('wxyz') # We will set this only once (echoed by cat)
p.expect(['1234'])
p.expect(['1234'])
p.expect(['abcd'])
p.expect(['wxyz'])
```

Not supported on platforms where `isatty()` returns False.

getwinsize()
Return the window size of the pseudoterminal as a tuple (rows, cols).

setwinsize(*rows*, *cols*)
Set the terminal window size of the child tty.

This will cause a SIGWINCH signal to be sent to the child. This does not change the physical window size. It changes the size reported to TTY-aware applications like vi or curses – applications that respond to the SIGWINCH signal.

eof()
This returns True if the EOF exception was ever raised.

isalive()
This tests if the child process is running or not. This is non-blocking. If the child was terminated then this will read the exitstatus or signalstatus of the child. This returns True if the child process appears to be running or False if not. It can take literally SECONDS for Solaris to return the right status.

wait()
This waits until the child exits. This is a blocking call. This will not read any data from the child, so this will block forever if the child has unread output and has terminated. In other words, the child may have printed output then called exit(), but, the child is technically still alive until its output is read by the parent.

kill(*sig*)
Send the given signal to the child application.

In keeping with UNIX tradition it has a misleading name. It does not necessarily kill the child unless you send the right signal. See the `signal` module for constants representing signal numbers.

terminate(*force=False*)
This forces a child process to terminate. It starts nicely with SIGHUP and SIGINT. If "force" is True then moves onto SIGKILL. This returns True if the child was terminated. This returns False if the child could not be terminated.

close(*force=True*)
This closes the connection with the child application. Note that calling close() more than once is valid. This emulates standard Python behavior with files. Set force to True if you want to make sure that the child is terminated (SIGKILL is sent if the child ignores SIGHUP and SIGINT).

class ptyprocess.**PtyProcessUnicode**(*pid*, *fd*, *encoding='utf-8'*, *codec_errors='strict'*)
Unicode wrapper around a process running in a pseudoterminal.

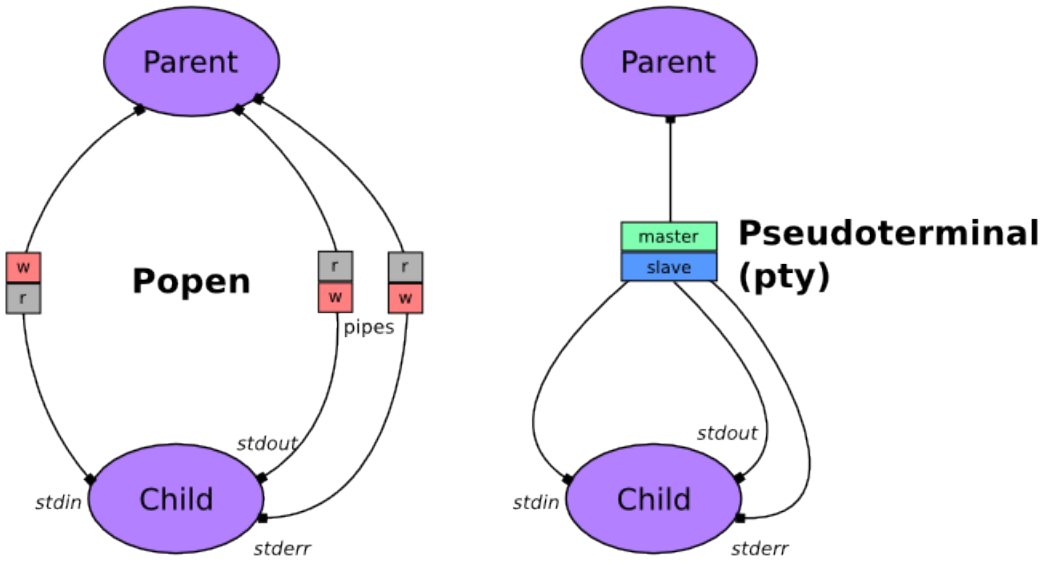
This class exposes a similar interface to *PtyProcess*, but its read methods return unicode, and its `write()` accepts unicode.

CHAPTER 2

What is a pty?

A pty is a kernel-level object which processes can write data to and read data from, a bit like a pipe.

Unlike a pipe, data moves through a single pty in both directions. When you use a program in a shell pipeline, or with `subprocess.Popen` in Python, up to three pipes are created for the process's standard streams (stdin, stdout and stderr). When you run a program using ptyprocess, all three of its standard streams are connected to a single pty:



A pty also does more than a pipe. It keeps track of the window size (rows and columns of characters) and notifies child processes (with a SIGWINCH signal) when it changes. In *cooked mode*, it does some processing of data sent from the parent process, so for instance the byte `03` (entered as Ctrl-C) will cause SIGINT to be sent to the child process.

Many command line programs behave differently if they detect that stdin or stdout is connected to a terminal instead

of a pipe (using isatty()), because this normally means that they're being used interactively by a human user. They may format output differently (e.g. `ls` lists files in columns) or prompt the user to confirm actions. When you run these programs in ptyprocess, they will exhibit their 'interactive' behaviour, instead of the 'pipe' behaviour you'll see using `Popen()`.

See also:

The TTY demystified Detailed article by Linus Akesson

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

ptyprocess, 3

Index

C

close() (*ptyprocess.PtyProcess method*), 5

E

eof() (*ptyprocess.PtyProcess method*), 5

G

getecho() (*ptyprocess.PtyProcess method*), 4
getwinsize() (*ptyprocess.PtyProcess method*), 5

I

isalive() (*ptyprocess.PtyProcess method*), 5

K

kill() (*ptyprocess.PtyProcess method*), 5

P

PtyProcess (*class in ptyprocess*), 3
ptyprocess (*module*), 3
PtyProcessUnicode (*class in ptyprocess*), 5

R

read() (*ptyprocess.PtyProcess method*), 3
readline() (*ptyprocess.PtyProcess method*), 3

S

sendcontrol() (*ptyprocess.PtyProcess method*), 3
sendeof() (*ptyprocess.PtyProcess method*), 4
sendintr() (*ptyprocess.PtyProcess method*), 4
setecho() (*ptyprocess.PtyProcess method*), 4
setwinsize() (*ptyprocess.PtyProcess method*), 5
spawn() (*ptyprocess.PtyProcess class method*), 3

T

terminate() (*ptyprocess.PtyProcess method*), 5

W

wait() (*ptyprocess.PtyProcess method*), 5
waitnoecho() (*ptyprocess.PtyProcess method*), 4
write() (*ptyprocess.PtyProcess method*), 3