Pexpect Documentation

Release 4.2.1

Noah Spurrier and contributors

Aug 14, 2017

Contents

Pexpect makes Python a better tool for controlling other applications.

Pexpect is a pure Python module for spawning child applications; controlling them; and responding to expected patterns in their output. Pexpect works like Don Libes' Expect. Pexpect allows your script to spawn a child application and control it as if a human were typing commands.

Pexpect can be used for automating interactive applications such as ssh, ftp, passwd, telnet, etc. It can be used to a automate setup scripts for duplicating software package installations on different servers. It can be used for automated software testing. Pexpect is in the spirit of Don Libes' Expect, but Pexpect is pure Python. Unlike other Expect-like modules for Python, Pexpect does not require TCL or Expect nor does it require C extensions to be compiled. It should work on any platform that supports the standard Python pty module. The Pexpect interface was designed to be easy to use.

Contents:

CHAPTER 1

Installation

Pexpect is on PyPI, and can be installed with standard tools:

```
pip install pexpect
```

Or:

```
easy_install pexpect
```

Requirements

This version of Pexpect requires Python 3.3 or above, or Python 2.7.

As of version 4.0, Pexpect can be used on Windows and POSIX systems. However, *pexpect.spawn* and *pexpect.run()* are only available on POSIX, where the `pty` module is present in the standard library. See *Pexpect on Windows* for more information.

CHAPTER 2

API Overview

Pexpect can be used for automating interactive applications such as ssh, ftp, mencoder, passwd, etc. The Pexpect interface was designed to be easy to use.

Here is an example of Pexpect in action:

```
# This connects to the openbsd ftp site and
# downloads the recursive directory listing.
import pexpect
child = pexpect.spawn('ftp ftp.openbsd.org')
child.expect('Name .*: ')
child.sendline('anonymous')
child.expect('Password:')
child.sendline('noah@example.com')
child.expect('ftp> ')
child.sendline('lcd /tmp')
child.expect('ftp> ')
child.sendline('cd pub/OpenBSD')
child.expect('ftp> ')
child.sendline('get README')
child.expect('ftp> ')
child.sendline('bye')
```

Obviously you could write an ftp client using Python's own `ftplib` module, but this is just a demonstration. You can use this technique with any application. This is especially handy if you are writing automated test tools.

There are two important methods in Pexpect – *`expect()`* and *`send()`* (or *`sendline()`* which is like *`send()`* with a linefeed). The *`expect()`* method waits for the child application to return a given string. The string you specify is a regular expression, so you can match complicated patterns. The *`send()`* method writes a string to the child application. From the child's point of view it looks just like someone typed the text from a terminal. After each call to *`expect()`* the `before` and `after` properties will be set to the text printed by child application. The `before` property will contain all text up to the expected string pattern. The `after` string will contain the text that was matched by the expected pattern. The match property is set to the re match object.

An example of Pexpect in action may make things more clear. This example uses ftp to login to the OpenBSD site; list files in a directory; and then pass interactive control of the ftp session to the human user:

```
import pexpect
child = pexpect.spawn ('ftp ftp.openbsd.org')
child.expect ('Name .*: ')
child.sendline ('anonymous')
child.expect ('Password:')
child.sendline ('noah@example.com')
child.expect ('ftp> ')
child.sendline ('ls /pub/OpenBSD/')
child.expect ('ftp> ')
print child.before   # Print the result of the ls command.
child.interact()     # Give control of the child to the user.
```

Special EOF and TIMEOUT patterns

There are two special patterns to match the End Of File (*EOF*) or a Timeout condition (*TIMEOUT*). You can pass these patterns to *expect()*. These patterns are not regular expressions. Use them like predefined constants.

If the child has died and you have read all the child's output then ordinarily *expect()* will raise an *EOF* exception. You can read everything up to the EOF without generating an exception by using the EOF pattern expect. In this case everything the child has output will be available in the `before` property.

The pattern given to *expect()* may be a regular expression or it may also be a list of regular expressions. This allows you to match multiple optional responses. The *expect()* method returns the index of the pattern that was matched. For example, say you wanted to login to a server. After entering a password you could get various responses from the server – your password could be rejected; or you could be allowed in and asked for your terminal type; or you could be let right in and given a command prompt. The following code fragment gives an example of this:

```
child.expect('password:')
child.sendline(my_secret_password)
# We expect any of these three patterns...
i = child.expect (['Permission denied', 'Terminal type', '[#\$] '])
if i==0:
    print('Permission denied on host. Can\'t login')
    child.kill(0)
elif i==1:
    print('Login OK... need to send terminal type.')
    child.sendline('vt100')
    child.expect('[#\$] ')
elif i==2:
    print('Login OK.')
    print('Shell command prompt', child.after)
```

If nothing matches an expected pattern then *expect()* will eventually raise a *TIMEOUT* exception. The default time is 30 seconds, but you can change this by passing a timeout argument to *expect()*:

```
# Wait no more than 2 minutes (120 seconds) for password prompt.
child.expect('password:', timeout=120)
```

Find the end of line – CR/LF conventions

Pexpect matches regular expressions a little differently than what you might be used to.

The $ pattern for end of line match is useless. The $ matches the end of string, but Pexpect reads from the child one character at a time, so each character looks like the end of a line. Pexpect can't do a look-ahead into the child's output stream. In general you would have this situation when using regular expressions with any stream.

Note: Pexpect does have an internal buffer, so reads are faster than one character at a time, but from the user's perspective the regex patterns test happens one character at a time.

The best way to match the end of a line is to look for the newline: "\r\n" (CR/LF). Yes, that does appear to be DOS-style. It may surprise some UNIX people to learn that terminal TTY device drivers (dumb, vt100, ANSI, xterm, etc.) all use the CR/LF combination to signify the end of line. Pexpect uses a Pseudo-TTY device to talk to the child application, so when the child app prints "\n" you actually see "\r\n".

UNIX uses just linefeeds to end lines of text, but not when it comes to TTY devices! TTY devices are more like the Windows world. Each line of text ends with a CR/LF combination. When you intercept data from a UNIX command from a TTY device you will find that the TTY device outputs a CR/LF combination. A UNIX command may only write a linefeed (\n), but the TTY device driver converts it to CR/LF. This means that your terminal will see lines end with CR/LF (hex 0D 0A). Since Pexpect emulates a terminal, to match ends of lines you have to expect the CR/LF combination:

```
child.expect('\r\n')
```

If you just need to skip past a new line then expect('\n') by itself will work, but if you are expecting a specific pattern before the end of line then you need to explicitly look for the \r. For example the following expects a word at the end of a line:

```
child.expect('\w+\r\n')
```

But the following would both fail:

```
child.expect('\w+\n')
```

And as explained before, trying to use $ to match the end of line would not work either:

```
child.expect ('\w+$')
```

So if you need to explicitly look for the END OF LINE, you want to look for the CR/LF combination – not just the LF and not the $ pattern.

This problem is not limited to Pexpect. This problem happens any time you try to perform a regular expression match on a stream. Regular expressions need to look ahead. With a stream it is hard to look ahead because the process generating the stream may not be finished. There is no way to know if the process has paused momentarily or is finished and waiting for you. Pexpect must implicitly always do a NON greedy match (minimal) at the end of a input.

Pexpect compiles all regular expressions with the re.DOTALL flag. With the DOTALL flag, a "." will match a newline.

Beware of + and * at the end of patterns

Remember that any time you try to match a pattern that needs look-ahead that you will always get a minimal match (non greedy). For example, the following will always return just one character:

```
child.expect ('.+')
```

This example will match successfully, but will always return no characters:

```
child.expect ('.*')
```

Generally any star * expression will match as little as possible.

One thing you can do is to try to force a non-ambiguous character at the end of your \d+ pattern. Expect that character to delimit the string. For example, you might try making the end of your pattern be \D+ instead of \D*. Number digits alone would not satisfy the (\d+)\D+ pattern. You would need some numbers and at least one non-number at the end.

Debugging

If you get the string value of a *pexpect.spawn* object you will get lots of useful debugging information. For debugging it's very useful to use the following pattern:

```
try:
    i = child.expect ([pattern1, pattern2, pattern3, etc])
except:
    print("Exception was thrown")
    print("debug information:")
    print(str(child))
```

It is also useful to log the child's input and out to a file or the screen. The following will turn on logging and send output to stdout (the screen):

```
child = pexpect.spawn(foo)
child.logfile = sys.stdout
```

Exceptions

EOF

Note that two flavors of EOF Exception may be thrown. They are virtually identical except for the message string. For practical purposes you should have no need to distinguish between them, but they do give a little extra information about what type of platform you are running. The two messages are:

- "End Of File (EOF) in read(). Exception style platform."
- "End Of File (EOF) in read(). Empty string style platform."

Some UNIX platforms will throw an exception when you try to read from a file descriptor in the EOF state. Other UNIX platforms instead quietly return an empty string to indicate that the EOF state has been reached.

If you wish to read up to the end of the child's output without generating an *EOF* exception then use the expect(pexpect.EOF) method.

TIMEOUT

The *expect()* and *read()* methods will also timeout if the child does not generate any output for a given amount of time. If this happens they will raise a *TIMEOUT* exception. You can have these methods ignore timeout and block indefinitely by passing None for the timeout parameter:

```
child.expect(pexpect.EOF, timeout=None)
```

Pexpect on Windows

New in version 4.0: Windows support

Pexpect can be used on Windows to wait for a pattern to be produced by a child process, using *pexpect.popen_spawn.PopenSpawn*, or a file descriptor, using *pexpect.fdpexpect.fdspawn*. This should be considered experimental for now.

pexpect.spawn and *pexpect.run()* are *not* available on Windows, as they rely on Unix pseudoterminals (ptys). Cross platform code must not use these.

CHAPTER 3

API documentation

Core pexpect components

Pexpect is a Python module for spawning child applications and controlling them automatically. Pexpect can be used for automating interactive applications such as ssh, ftp, passwd, telnet, etc. It can be used to a automate setup scripts for duplicating software package installations on different servers. It can be used for automated software testing. Pexpect is in the spirit of Don Libes' Expect, but Pexpect is pure Python. Other Expect-like modules for Python require TCL and Expect or require C extensions to be compiled. Pexpect does not use C, Expect, or TCL extensions. It should work on any platform that supports the standard Python pty module. The Pexpect interface focuses on ease of use so that simple tasks are easy.

There are two main interfaces to the Pexpect system; these are the function, run() and the class, spawn. The spawn class is more powerful. The run() function is simpler than spawn, and is good for quickly calling program. When you call the run() function it executes a given program and then returns the output. This is a handy replacement for os.system().

For example:

```
pexpect.run('ls -la')
```

The spawn class is the more powerful interface to the Pexpect system. You can use this to spawn a child program then interact with it by sending input and expecting responses (waiting for patterns in the child's output).

For example:

```
child = pexpect.spawn('scp foo user@example.com:.')
child.expect('Password:')
child.sendline(mypassword)
```

This works even for commands that ask for passwords or other input outside of the normal stdio streams. For example, ssh reads input directly from the TTY device which bypasses stdin.

Credits: Noah Spurrier, Richard Holden, Marco Molteni, Kimberley Burchett, Robert Stone, Hartmut Goebel, Chad Schroeder, Erick Tryzelaar, Dave Kirby, Ids vander Molen, George Todd, Noel Taylor, Nicolas D. Cesar, Alexander Gattin, Jacques-Etienne Baudoux, Geoffrey Marshall, Francisco Lourenco, Glen Mabey, Karthik Gurusamy, Fernando

Perez, Corey Minyard, Jon Cohen, Guillaume Chazarain, Andrew Ryan, Nick Craig-Wood, Andrew Stone, Jorgen Grahn, John Spiegel, Jan Grant, and Shane Kerr. Let me know if I forgot anyone.

Pexpect is free, open source, and all that good stuff. http://pexpect.sourceforge.net/

PEXPECT LICENSE

spawn class

class pexpect.**spawn**(*command*, *args=[]*, *timeout=30*, *maxread=2000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*, *ignore_sighup=False*, *echo=True*, *preexec_fn=None*, *encoding=None*, *codec_errors='strict'*, *dimensions=None*)

This is the main class interface for Pexpect. Use this class to start and control child applications.

__init__(*command*, *args=[]*, *timeout=30*, *maxread=2000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*, *ignore_sighup=False*, *echo=True*, *preexec_fn=None*, *encoding=None*, *codec_errors='strict'*, *dimensions=None*)

This is the constructor. The command parameter may be a string that includes a command and any arguments to the command. For example:

```
child = pexpect.spawn('/usr/bin/ftp')
child = pexpect.spawn('/usr/bin/ssh user@example.com')
child = pexpect.spawn('ls -latr /tmp')
```

You may also construct it with a list of arguments like so:

```
child = pexpect.spawn('/usr/bin/ftp', [])
child = pexpect.spawn('/usr/bin/ssh', ['user@example.com'])
child = pexpect.spawn('ls', ['-latr', '/tmp'])
```

After this the child application will be created and will be ready to talk to. For normal use, see expect() and send() and sendline().

Remember that Pexpect does NOT interpret shell meta characters such as redirect, pipe, or wild cards (>, |, or *). This is a common mistake. If you want to run a command and pipe it through another command then you must also start a shell. For example:

```
child = pexpect.spawn('/bin/bash -c "ls -l | grep LOG > logs.txt"')
child.expect(pexpect.EOF)
```

The second form of spawn (where you pass a list of arguments) is useful in situations where you wish to spawn a command and pass it its own argument list. This can make syntax more clear. For example, the following is equivalent to the previous example:

```
shell_cmd = 'ls -l | grep LOG > logs.txt'
child = pexpect.spawn('/bin/bash', ['-c', shell_cmd])
child.expect(pexpect.EOF)
```

The maxread attribute sets the read buffer size. This is maximum number of bytes that Pexpect will try to read from a TTY at one time. Setting the maxread size to 1 will turn off buffering. Setting the maxread value higher may help performance in cases where large amounts of output are read back from the child. This feature is useful in conjunction with searchwindowsize.

When the keyword argument *searchwindowsize* is None (default), the full buffer is searched at each iteration of receiving incoming data. The default number of bytes scanned at each iteration is very large and may be reduced to collaterally reduce search cost. After *expect()* returns, the full buffer attribute remains up to size *maxread* irrespective of *searchwindowsize* value.

When the keyword argument `timeout` is specified as a number, (default: *30*), then *TIMEOUT* will be raised after the value specified has elapsed, in seconds, for any of the *expect()* family of method calls. When None, TIMEOUT will not be raised, and *expect()* may block indefinitely until match.

The logfile member turns on or off logging. All input and output will be copied to the given file object. Set logfile to None to stop logging. This is the default. Set logfile to sys.stdout to echo everything to standard output. The logfile is flushed after each write.

Example log input and output to a file:

```
child = pexpect.spawn('some_command')
fout = open('mylog.txt','wb')
child.logfile = fout
```

Example log to stdout:

```
# In Python 2:
child = pexpect.spawn('some_command')
child.logfile = sys.stdout

# In Python 3, we'll use the ``encoding`` argument to decode data
# from the subprocess and handle it as unicode:
child = pexpect.spawn('some_command', encoding='utf-8')
child.logfile = sys.stdout
```

The logfile_read and logfile_send members can be used to separately log the input from the child and output sent to the child. Sometimes you don't want to see everything you write to the child. You only want to log what the child sends back. For example:

```
child = pexpect.spawn('some_command')
child.logfile_read = sys.stdout
```

You will need to pass an encoding to spawn in the above code if you are using Python 3.

To separately log output sent to the child use logfile_send:

```
child.logfile_send = fout
```

If `ignore_sighup` is True, the child process will ignore SIGHUP signals. The default is False from Pexpect 4.0, meaning that SIGHUP will be handled normally by the child.

The delaybeforesend helps overcome a weird behavior that many users were experiencing. The typical problem was that a user would expect() a "Password:" prompt and then immediately call sendline() to send the password. The user would then see that their password was echoed back to them. Passwords don't normally echo. The problem is caused by the fact that most applications print out the "Password"

prompt and then turn off stdin echo, but if you send your password before the application turned off echo, then you get your password echoed. Normally this wouldn't be a problem when interacting with a human at a real keyboard. If you introduce a slight delay just before writing then this seems to clear up the problem. This was such a common problem for many users that I decided that the default pexpect behavior should be to sleep just before writing to the child application. 1/20th of a second (50 ms) seems to be enough to clear up the problem. You can set delaybeforesend to None to return to the old behavior.

Note that spawn is clever about finding commands on your path. It uses the same logic that "which" uses to find executables.

If you wish to get the exit status of the child you must call the close() method. The exit or signal status of the child will be stored in self.exitstatus or self.signalstatus. If the child exited normally then exitstatus will store the exit return code and signalstatus will be None. If the child was terminated abnormally with a signal then signalstatus will store the signal value and exitstatus will be None:

```
child = pexpect.spawn('some_command')
child.close()
print(child.exitstatus, child.signalstatus)
```

If you need more detail you can also read the self.status member which stores the status returned by os.waitpid. You can interpret this using os.WIFEXITED/os.WEXITSTATUS or os.WIFSIGNALED/os.TERMSIG.

The echo attribute may be set to False to disable echoing of input. As a pseudo-terminal, all input echoed by the "keyboard" (send() or sendline()) will be repeated to output. For many cases, it is not desirable to have echo enabled, and it may be later disabled using setecho(False) followed by waitnoecho(). However, for some platforms such as Solaris, this is not possible, and should be disabled immediately on spawn.

If preexec_fn is given, it will be called in the child process before launching the given command. This is useful to e.g. reset inherited signal handlers.

The dimensions attribute specifies the size of the pseudo-terminal as seen by the subprocess, and is specified as a two-entry tuple (rows, columns). If this is unspecified, the defaults in ptyprocess will apply.

expect (*pattern*, *timeout=-1*, *searchwindowsize=-1*, *async=False*)

This seeks through the stream until a pattern is matched. The pattern is overloaded and may take several types. The pattern can be a StringType, EOF, a compiled re, or a list of any of those types. Strings will be compiled to re types. This returns the index into the pattern list. If the pattern was not a list this returns index 0 on a successful match. This may raise exceptions for EOF or TIMEOUT. To avoid the EOF or TIMEOUT exceptions add EOF or TIMEOUT to the pattern list. That will cause expect to match an EOF or TIMEOUT condition instead of raising an exception.

If you pass a list of patterns and more than one matches, the first match in the stream is chosen. If more than one pattern matches at that point, the leftmost in the pattern list is chosen. For example:

```
# the input is 'foobar'
index = p.expect(['bar', 'foo', 'foobar'])
# returns 1('foo') even though 'foobar' is a "better" match
```

Please note, however, that buffering can affect this behavior, since input arrives in unpredictable chunks. For example:

```
# the input is 'foobar'
index = p.expect(['foobar', 'foo'])
# returns 0('foobar') if all input is available at once,
# but returs 1('foo') if parts of the final 'bar' arrive late
```

When a match is found for the given pattern, the class instance attribute *match* becomes an re.MatchObject result. Should an EOF or TIMEOUT pattern match, then the match attribute will be an instance of that

exception class. The pairing before and after class instance attributes are views of the data preceding and following the matching pattern. On general exception, class attribute *before* is all data received up to the exception, while *match* and *after* attributes are value None.

When the keyword argument timeout is -1 (default), then TIMEOUT will raise after the default value specified by the class timeout attribute. When None, TIMEOUT will not be raised and may block indefinitely until match.

When the keyword argument searchwindowsize is -1 (default), then the value specified by the class maxread attribute is used.

A list entry may be EOF or TIMEOUT instead of a string. This will catch these exceptions and return the index of the list entry instead of raising the exception. The attribute 'after' will be set to the exception type. The attribute 'match' will be None. This allows you to write code like this:

```
index = p.expect(['good', 'bad', pexpect.EOF, pexpect.TIMEOUT])
if index == 0:
    do_something()
elif index == 1:
    do_something_else()
elif index == 2:
    do_some_other_thing()
elif index == 3:
    do_something_completely_different()
```

instead of code like this:

```
try:
    index = p.expect(['good', 'bad'])
    if index == 0:
        do_something()
    elif index == 1:
        do_something_else()
except EOF:
    do_some_other_thing()
except TIMEOUT:
    do_something_completely_different()
```

These two forms are equivalent. It all depends on what you want. You can also just expect the EOF if you are waiting for all output of a child to finish. For example:

```
p = pexpect.spawn('/bin/ls')
p.expect(pexpect.EOF)
print p.before
```

If you are trying to optimize for speed then see expect_list().

On Python 3.4, or Python 3.3 with asyncio installed, passing `async=True` will make this return an `asyncio` coroutine, which you can yield from to get the same result that this method would normally give directly. So, inside a coroutine, you can replace this code:

```
index = p.expect(patterns)
```

With this non-blocking form:

```
index = yield from p.expect(patterns, async=True)
```

expect_exact (*pattern_list*, *timeout=-1*, *searchwindowsize=-1*, *async=False*)

This is similar to expect(), but uses plain string matching instead of compiled regular expressions in 'pattern_list'. The 'pattern_list' may be a string; a list or other sequence of strings; or TIMEOUT and EOF.

This call might be faster than expect() for two reasons: string searching is faster than RE matching and it is possible to limit the search to just the end of the input buffer.

This method is also useful when you don't want to have to worry about escaping regular expression characters that you want to match.

Like *expect()*, passing `async=True` will make this return an asyncio coroutine.

expect_list (*pattern_list*, *timeout=-1*, *searchwindowsize=-1*, *async=False*)

This takes a list of compiled regular expressions and returns the index into the pattern_list that matched the child output. The list may also contain EOF or TIMEOUT(which are not compiled regular expressions). This method is similar to the expect() method except that expect_list() does not recompile the pattern list on every call. This may help if you are trying to optimize for speed, otherwise just use the expect() method. This is called by expect().

Like *expect()*, passing `async=True` will make this return an asyncio coroutine.

compile_pattern_list (*patterns*)

This compiles a pattern-string or a list of pattern-strings. Patterns must be a StringType, EOF, TIMEOUT, SRE_Pattern, or a list of those. Patterns may also be None which results in an empty list (you might do this if waiting for an EOF or TIMEOUT condition without expecting any pattern).

This is used by expect() when calling expect_list(). Thus expect() is nothing more than:

```
cpl = self.compile_pattern_list(pl)
return self.expect_list(cpl, timeout)
```

If you are using expect() within a loop it may be more efficient to compile the patterns first and then call expect_list(). This avoid calls in a loop to compile_pattern_list():

```
cpl = self.compile_pattern_list(my_pattern)
while some_condition:
   ...
   i = self.expect_list(cpl, timeout)
   ...
```

send (*s*)

Sends string `s` to the child process, returning the number of bytes written. If a logfile is specified, a copy is written to that log.

The default terminal input mode is canonical processing unless set otherwise by the child process. This allows backspace and other line processing to be performed prior to transmitting to the receiving program. As this is buffered, there is a limited size of such buffer.

On Linux systems, this is 4096 (defined by N_TTY_BUF_SIZE). All other systems honor the POSIX.1 definition PC_MAX_CANON – 1024 on OSX, 256 on OpenSolaris, and 1920 on FreeBSD.

This value may be discovered using fpathconf(3):

```
>>> from os import fpathconf
>>> print(fpathconf(0, 'PC_MAX_CANON'))
256
```

On such a system, only 256 bytes may be received per line. Any subsequent bytes received will be discarded. BEL (`''`) is then sent to output if IMAXBEL (termios.h) is set by the tty driver. This is usually enabled by default. Linux does not honor this as an option – it behaves as though it is always set on.

Canonical input processing may be disabled altogether by executing a shell, then stty(1), before executing the final program:

```
>>> bash = pexpect.spawn('/bin/bash', echo=False)
>>> bash.sendline('stty -icanon')
>>> bash.sendline('base64')
>>> bash.sendline('x' * 5000)
```

sendline(*s=''*)
Wraps send(), sending string `s` to child process, with `os.linesep` automatically appended. Returns number of bytes written. Only a limited number of bytes may be sent for each line in the default terminal mode, see docstring of *send()*.

write(*s*)
This is similar to send() except that there is no return value.

writelines(*sequence*)
This calls write() for each element in the sequence. The sequence can be any iterable object producing strings, typically a list of strings. This does not add line separators. There is no return value.

sendcontrol(*char*)
Helper method that wraps send() with mnemonic access for sending control character to the child (such as Ctrl-C or Ctrl-D). For example, to send Ctrl-G (ASCII 7, bell, ''):

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

See also, sendintr() and sendeof().

sendeof()
This sends an EOF to the child. 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()
This sends a SIGINT to the child. It does not require the SIGINT to be the first character on a line.

read(*size=-1*)
This reads at most "size" bytes from the file (less if the read hits EOF before obtaining size bytes). If the size argument is negative or omitted, read all data until EOF is reached. The bytes are returned as a string object. An empty string is returned when EOF is encountered immediately.

readline(*size=-1*)
This reads and returns one entire line. The newline at the end of line is returned as part of the string, unless the file ends without a newline. An empty string is returned if EOF is encountered immediately. This looks for a newline as a CR/LF pair (rn) even on UNIX because this is what the pseudotty device returns. So contrary to what you may expect you will receive newlines as rn.

If the size argument is 0 then an empty string is returned. In all other cases the size argument is ignored, which is not standard behavior for a file-like object.

read_nonblocking(*size=1*, *timeout=-1*)
This reads at most size characters from the child application. It includes a timeout. If the read does not complete within the timeout period then a TIMEOUT exception is raised. If the end of file is read then an EOF exception will be raised. If a logfile is specified, a copy is written to that log.

If timeout is None then the read may block indefinitely. If timeout is -1 then the self.timeout value is used. If timeout is 0 then the child is polled and if there is no data immediately ready then this will raise a TIMEOUT exception.

The timeout refers only to the amount of time to read at least one character. This is not affected by the 'size' parameter, so if you call read_nonblocking(size=100, timeout=30) and only one character is available right away then one character will be returned immediately. It will not wait for 30 seconds for another 99 characters to come in.

This is a wrapper around os.read(). It uses select.select() to implement the timeout.

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

interact(*escape_character='\x1d'*, *input_filter=None*, *output_filter=None*)
This gives control of the child process to the interactive user (the human at the keyboard). Keystrokes are sent to the child process, and the stdout and stderr output of the child process is printed. This simply echos the child stdout and child stderr to the real stdout and it echos the real stdin to the child stdin. When the user types the escape_character this method will return None. The escape_character will not be transmitted. The default for escape_character is entered as `Ctrl - ]`, the very same as BSD telnet. To prevent escaping, escape_character may be set to None.

If a logfile is specified, then the data sent and received from the child process in interact mode is duplicated to the given log.

You may pass in optional input and output filter functions. These functions should take a string and return a string. The output_filter will be passed all the output from the child process. The input_filter will be passed all the keyboard input from the user. The input_filter is run BEFORE the check for the escape_character.

Note that if you change the window size of the parent the SIGWINCH signal will not be passed through to the child. If you want the child window size to change when the parent's window size changes then do something like the following example:

```
import pexpect, struct, fcntl, termios, signal, sys
def sigwinch_passthrough (sig, data):
    s = struct.pack("HHHH", 0, 0, 0, 0)
    a = struct.unpack('hhhh', fcntl.ioctl(sys.stdout.fileno(),
        termios.TIOCGWINSZ , s))
    global p
    p.setwinsize(a[0],a[1])
# Note this 'p' global and used in sigwinch_passthrough.
p = pexpect.spawn('/bin/bash')
signal.signal(signal.SIGWINCH, sigwinch_passthrough)
p.interact()
```

logfile
logfile_read
logfile_send
Set these to a Python file object (or `sys.stdout`) to log all communication, data read from the child process, or data sent to the child process.

Note: With *spawn* in bytes mode, the log files should be open for writing binary data. In unicode mode, they should be open for writing unicode text. See *Handling unicode*.

Controlling the child process

class pexpect.**spawn**

kill(*sig*)
This sends 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.

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.

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.

This method is non-blocking if *wait()* has already been called previously or *isalive()* method returns False. It simply returns the previously determined exit status.

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).

getwinsize()
This returns the terminal window size of the child tty. The return value is a tuple of (rows, cols).

setwinsize(*rows*, *cols*)
This sets 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.

getecho()
This returns the terminal echo mode. This returns True if echo is on or False if echo is off. Child applications that are expecting you to enter a password often set ECHO False. See waitnoecho().

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

setecho(*state*)
This sets the terminal echo mode on or off. Note that 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).
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.

waitnoecho(*timeout=-1*)
This waits until the terminal ECHO flag is set False. This returns True if the echo mode is off. This returns False if the ECHO flag was not set False 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 set ECHO off:

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

If timeout==-1 then this method will use the value in self.timeout. If timeout==None then this method to block until ECHO flag is False.

pid
The process ID of the child process.

child_fd
The file descriptor used to communicate with the child process.

Handling unicode

By default, *spawn* is a bytes interface: its read methods return bytes, and its write/send and expect methods expect bytes. If you pass the *encoding* parameter to the constructor, it will instead act as a unicode interface: strings you send will be encoded using that encoding, and bytes received will be decoded before returning them to you. In this mode, patterns for *expect()* and *expect_exact()* should also be unicode.

Changed in version 4.0: *spawn* provides both the bytes and unicode interfaces. In Pexpect 3.x, the unicode interface was provided by a separate `spawnu` class.

For backwards compatibility, some Unicode is allowed in bytes mode: the send methods will encode arbitrary unicode as UTF-8 before sending it to the child process, and its expect methods can accept ascii-only unicode strings.

Note: Unicode handling with pexpect works the same way on Python 2 and 3, despite the difference in names. I.e.:

- Bytes mode works with `str` on Python 2, and `bytes` on Python 3,
- Unicode mode works with `unicode` on Python 2, and `str` on Python 3.

---

run function

`pexpect`.**run**(*command*, *timeout=30*, *withexitstatus=False*, *events=None*, *extra_args=None*, *logfile=None*, *cwd=None*, *env=None*, ***kwargs*)
This function runs the given command; waits for it to finish; then returns all output as a string. STDERR is included in output. If the full path to the command is not given then the path is searched.

Note that lines are terminated by CR/LF (rn) combination even on UNIX-like systems because this is the standard for pseudottys. If you set 'withexitstatus' to true, then run will return a tuple of (command_output, exitstatus). If 'withexitstatus' is false then this returns just command_output.

The run() function can often be used instead of creating a spawn instance. For example, the following code uses spawn:

```
from pexpect import *
child = spawn('scp foo user@example.com:.')
child.expect('(?i)password')
child.sendline(mypassword)
```

The previous code can be replace with the following:

```
from pexpect import *
run('scp foo user@example.com:.', events={'(?i)password': mypassword})
```

Examples

Start the apache daemon on the local machine:

```
from pexpect import *
run("/usr/local/apache/bin/apachectl start")
```

Check in a file using SVN:

```
from pexpect import *
run("svn ci -m 'automatic commit' my_file.py")
```

Run a command and capture exit status:

```
from pexpect import *
(command_output, exitstatus) = run('ls -l /bin', withexitstatus=1)
```

The following will run SSH and execute 'ls -l' on the remote machine. The password 'secret' will be sent if the '(?i)password' pattern is ever seen:

```
run("ssh username@machine.example.com 'ls -l'",
    events={'(?i)password':'secret\n'})
```

This will start mencoder to rip a video from DVD. This will also display progress ticks every 5 seconds as it runs. For example:

```
from pexpect import *
def print_ticks(d):
    print d['event_count'],
run("mencoder dvd://1 -o video.avi -oac copy -ovc copy",
    events={TIMEOUT:print_ticks}, timeout=5)
```

The 'events' argument should be either a dictionary or a tuple list that contains patterns and responses. Whenever one of the patterns is seen in the command output, run() will send the associated response string. So, run() in the above example can be also written as:

run("mencoder dvd://1 -o video.avi -oac copy -ovc copy", events=[(TIMEOUT,print_ticks)], timeout=5)

Use a tuple list for events if the command output requires a delicate control over what pattern should be matched, since the tuple list is passed to pexpect() as its pattern list, with the order of patterns preserved.

Note that you should put newlines in your string if Enter is necessary.

Like the example above, the responses may also contain a callback, either a function or method. It should accept a dictionary value as an argument. The dictionary contains all the locals from the run() function, so you can access the child spawn object or any other variable defined in run() (event_count, child, and extra_args are the most useful). A callback may return True to stop the current run process. Otherwise run() continues until the next event. A callback may also return a string which will be sent to the child. 'extra_args' is not used by directly run(). It provides a way to pass data to a callback function through run() through the locals dictionary passed to a callback.

Like *spawn*, passing *encoding* will make it work with unicode instead of bytes. You can pass *codec_errors* to control how errors in encoding and decoding are handled.

Exceptions

class pexpect.**EOF**(*value*)
Raised when EOF is read from a child. This usually means the child has exited.

class pexpect.**TIMEOUT**(*value*)
Raised when a read time exceeds the timeout.

class pexpect.**ExceptionPexpect**(*value*)
Base class for all exceptions raised by this module.

Utility functions

pexpect.**which**(*filename*, *env=None*)
This takes a given filename; tries to find it in the environment path; then checks if it is executable. This returns the full path to the filename if found and executable. Otherwise this returns None.

pexpect.**split_command_line**(*command_line*)
This splits a command line into a list of arguments. It splits arguments on spaces, but handles embedded quotes, doublequotes, and escaped characters. It's impossible to do this with a regular expression, so I wrote a little state machine to parse the command line.

fdpexpect - use pexpect with a file descriptor

This is like pexpect, but it will work with any file descriptor that you pass it. You are reponsible for opening and close the file descriptor. This allows you to use Pexpect with sockets and named pipes (FIFOs).

PEXPECT LICENSE

This license is approved by the OSI and FSF as GPL-compatible. http://opensource.org/licenses/isc-license.txt

Copyright (c) 2012, Noah Spurrier <noah@noah.org> PERMISSION TO USE, COPY, MODIFY, AND/OR DISTRIBUTE THIS SOFTWARE FOR ANY PURPOSE WITH OR WITHOUT FEE IS HEREBY GRANTED, PROVIDED THAT THE ABOVE COPYRIGHT NOTICE AND THIS PERMISSION NOTICE APPEAR IN ALL COPIES. THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

fdspawn class

class pexpect.fdpexpect.**fdspawn**(*fd*, *args=None*, *timeout=30*, *maxread=2000*, *searchwindowsize=None*, *logfile=None*, *encoding=None*, *codec_errors='strict'*)

Bases: pexpect.spawnbase.SpawnBase

This is like pexpect.spawn but allows you to supply your own open file descriptor. For example, you could use it to read through a file looking for patterns, or to control a modem or serial device.

__init__(*fd*, *args=None*, *timeout=30*, *maxread=2000*, *searchwindowsize=None*, *logfile=None*, *encoding=None*, *codec_errors='strict'*)

This takes a file descriptor (an int) or an object that support the fileno() method (returning an int). All Python file-like objects support fileno().

isalive()

This checks if the file descriptor is still valid. If os.fstat() does not raise an exception then we assume it is alive.

close()

Close the file descriptor.

Calling this method a second time does nothing, but if the file descriptor was closed elsewhere, OSError will be raised.

expect()
expect_exact()
expect_list()

As *pexpect.spawn*.

popen_spawn - use pexpect with a piped subprocess

Provides an interface like pexpect.spawn interface using subprocess.Popen

PopenSpawn class

class pexpect.popen_spawn.**PopenSpawn**(*cmd*, *timeout=30*, *maxread=2000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*, *encoding=None*, *codec_errors='strict'*)

__init__(*cmd*, *timeout=30*, *maxread=2000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*, *encoding=None*, *codec_errors='strict'*)

send(*s*)

Send data to the subprocess' stdin.

Returns the number of bytes written.

sendline(*s=''*)

Wraps send(), sending string s to child process, with os.linesep automatically appended. Returns number of bytes written.

write(*s*)

This is similar to send() except that there is no return value.

writelines(*sequence*)

This calls write() for each element in the sequence.

The sequence can be any iterable object producing strings, typically a list of strings. This does not add line separators. There is no return value.

kill(*sig*)
Sends a Unix signal to the subprocess.

Use constants from the `signal` module to specify which signal.

sendeof()
Closes the stdin pipe from the writing end.

wait()
Wait for the subprocess to finish.

Returns the exit code.

expect()
expect_exact()
expect_list()
As *pexpect.spawn*.

replwrap - Control read-eval-print-loops

Generic wrapper for read-eval-print-loops, a.k.a. interactive shells

New in version 3.3.

class pexpect.replwrap.**REPLWrapper**(*cmd_or_spawn*, *orig_prompt*, *prompt_change*, *new_prompt='[PEXPECT_PROMPT>'*, *continuation_prompt='[PEXPECT_PROMPT+'*, *extra_init_cmd=None*)
Wrapper for a REPL.

Parameters

- **cmd_or_spawn** – This can either be an instance of *pexpect.spawn* in which a REPL has already been started, or a str command to start a new REPL process.
- **orig_prompt** (*str*) – The prompt to expect at first.
- **prompt_change** (*str*) – A command to change the prompt to something more unique. If this is `None`, the prompt will not be changed. This will be formatted with the new and continuation prompts as positional parameters, so you can use `{}` style formatting to insert them into the command.
- **new_prompt** (*str*) – The more unique prompt to expect after the change.
- **extra_init_cmd** (*str*) – Commands to do extra initialisation, such as disabling pagers.

run_command(*command*, *timeout=-1*)
Send a command to the REPL, wait for and return output.

Parameters

- **command** (*str*) – The command to send. Trailing newlines are not needed. This should be a complete block of input that will trigger execution; if a continuation prompt is found after sending input, `ValueError` will be raised.
- **timeout** (*int*) – How long to wait for the next prompt. -1 means the default from the *pexpect.spawn* object (default 30 seconds). None means to wait indefinitely.

pexpect.replwrap.**PEXPECT_PROMPT**
A string that can be used as a prompt, and is unlikely to be found in output.

Using the objects above, it is easy to wrap a REPL. For instance, to use a Python shell:

```
py = REPLWrapper("python", ">>> ", "import sys; sys.ps1={!r}; sys.ps2={!r}")
py.run_command("4+7")
```

Convenience functions are provided for Python and bash shells:

pexpect.replwrap.**python**(*command='python'*)
Start a Python shell and return a *REPLWrapper* object.

pexpect.replwrap.**bash**(*command='bash'*)
Start a bash shell and return a *REPLWrapper* object.

pxssh - control an SSH session

This class extends pexpect.spawn to specialize setting up SSH connections. This adds methods for login, logout, and expecting the shell prompt.

PEXPECT LICENSE

> **This license is approved by the OSI and FSF as GPL-compatible.** http://opensource.org/licenses/isc-license.txt
>
> Copyright (c) 2012, Noah Spurrier <noah@noah.org> PERMISSION TO USE, COPY, MODIFY, AND/OR DISTRIBUTE THIS SOFTWARE FOR ANY PURPOSE WITH OR WITHOUT FEE IS HEREBY GRANTED, PROVIDED THAT THE ABOVE COPYRIGHT NOTICE AND THIS PERMISSION NOTICE APPEAR IN ALL COPIES. THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

class pexpect.pxssh.**ExceptionPxssh**(*value*)
Raised for pxssh exceptions.

pxssh class

class pexpect.pxssh.**pxssh**(*timeout=30*, *maxread=2000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*, *ignore_sighup=True*, *echo=True*, *options={}*, *encoding=None*, *codec_errors='strict'*)
This class extends pexpect.spawn to specialize setting up SSH connections. This adds methods for login, logout, and expecting the shell prompt. It does various tricky things to handle many situations in the SSH login process. For example, if the session is your first login, then pxssh automatically accepts the remote certificate; or if you have public key authentication setup then pxssh won't wait for the password prompt.

pxssh uses the shell prompt to synchronize output from the remote host. In order to make this more robust it sets the shell prompt to something more unique than just $ or #. This should work on most Borne/Bash or Csh style shells.

Example that runs a few commands on a remote server and prints the result:

```
from pexpect import pxssh
import getpass
try:
    s = pxssh.pxssh()
    hostname = raw_input('hostname: ')
    username = raw_input('username: ')
    password = getpass.getpass('password: ')
    s.login(hostname, username, password)
    s.sendline('uptime')    # run a command
    s.prompt()              # match the prompt
    print(s.before)         # print everything before the prompt.
    s.sendline('ls -l')
    s.prompt()
    print(s.before)
    s.sendline('df')
    s.prompt()
    print(s.before)
    s.logout()
except pxssh.ExceptionPxssh as e:
    print("pxssh failed on login.")
    print(e)
```

Example showing how to specify SSH options:

```
from pexpect import pxssh
s = pxssh.pxssh(options={
                    "StrictHostKeyChecking": "no",
                    "UserKnownHostsFile": "/dev/null"})
...
```

Note that if you have ssh-agent running while doing development with pxssh then this can lead to a lot of confusion. Many X display managers (xdm, gdm, kdm, etc.) will automatically start a GUI agent. You may see a GUI dialog box popup asking for a password during development. You should turn off any key agents during testing. The 'force_password' attribute will turn off public key authentication. This will only work if the remote SSH server is configured to allow password logins. Example of using 'force_password' attribute:

```
s = pxssh.pxssh()
s.force_password = True
hostname = raw_input('hostname: ')
username = raw_input('username: ')
password = getpass.getpass('password: ')
s.login (hostname, username, password)
```

__init__(*timeout=30*, *maxread=2000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*, *ignore_sighup=True*, *echo=True*, *options={}*, *encoding=None*, *codec_errors='strict'*)

PROMPT

The regex pattern to search for to find the prompt. If you call *login()* with `auto_prompt_reset=False`, you must set this attribute manually.

force_password

If this is set to True, public key authentication is disabled, forcing the server to ask for a password. Note that the sysadmin can disable password logins, in which case this won't work.

options

The dictionary of user specified SSH options, eg, `options = dict(StrictHostKeyChecking="no", UserKnownHostsFile="/dev/null")`

login(*server*, *username*, *password=''*, *terminal_type='ansi'*, *original_prompt='[#$]'*, *login_timeout=10*, *port=None*, *auto_prompt_reset=True*, *ssh_key=None*, *quiet=True*, *sync_multiplier=1*, *check_local_ip=True*)
This logs the user into the given server.

It uses 'original_prompt' to try to find the prompt right after login. When it finds the prompt it immediately tries to reset the prompt to something more easily matched. The default 'original_prompt' is very optimistic and is easily fooled. It's more reliable to try to match the original prompt as exactly as possible to prevent false matches by server strings such as the "Message Of The Day". On many systems you can disable the MOTD on the remote server by creating a zero-length file called `~/.hushlogin` on the remote server. If a prompt cannot be found then this will not necessarily cause the login to fail. In the case of a timeout when looking for the prompt we assume that the original prompt was so weird that we could not match it, so we use a few tricks to guess when we have reached the prompt. Then we hope for the best and blindly try to reset the prompt to something more unique. If that fails then login() raises an *ExceptionPxssh* exception.

In some situations it is not possible or desirable to reset the original prompt. In this case, pass `auto_prompt_reset=False` to inhibit setting the prompt to the UNIQUE_PROMPT. Remember that pxssh uses a unique prompt in the *prompt()* method. If the original prompt is not reset then this will disable the *prompt()* method unless you manually set the *PROMPT* attribute.

logout()
Sends exit to the remote shell.

If there are stopped jobs then this automatically sends exit twice.

prompt(*timeout=-1*)
Match the next shell prompt.

This is little more than a short-cut to the *expect()* method. Note that if you called *login()* with `auto_prompt_reset=False`, then before calling *prompt()* you must set the *PROMPT* attribute to a regex that it will use for matching the prompt.

Calling *prompt()* will erase the contents of the `before` attribute even if no prompt is ever matched. If timeout is not given or it is set to -1 then self.timeout is used.

Returns True if the shell prompt was matched, False if the timeout was reached.

sync_original_prompt(*sync_multiplier=1.0*)
This attempts to find the prompt. Basically, press enter and record the response; press enter again and record the response; if the two responses are similar then assume we are at the original prompt. This can be a slow function. Worst case with the default sync_multiplier can take 12 seconds. Low latency connections are more likely to fail with a low sync_multiplier. Best case sync time gets worse with a high sync multiplier (500 ms with default).

set_unique_prompt()
This sets the remote prompt to something more unique than `#` or `$`. This makes it easier for the *prompt()* method to match the shell prompt unambiguously. This method is called automatically by the *login()* method, but you may want to call it manually if you somehow reset the shell prompt. For example, if you 'su' to a different user then you will need to manually reset the prompt. This sends shell commands to the remote host to set the prompt, so this assumes the remote host is ready to receive commands.

Alternatively, you may use your own prompt pattern. In this case you should call *login()* with `auto_prompt_reset=False`; then set the *PROMPT* attribute to a regular expression. After that, the *prompt()* method will try to match your prompt pattern.

The modules `pexpect.screen` and `pexpect.ANSI` have been deprecated in Pexpect version 4. They were separate from the main use cases for Pexpect, and there are better maintained Python terminal emulator packages, such as pyte. These modules are still present for now, but we don't advise using them in new code.

CHAPTER 4

Examples

Under the distribution tarball directory you should find an "examples" directory. This is the best way to learn to use Pexpect. See the descriptions of Pexpect Examples.

topip.py This runs *netstat* on a local or remote server. It calculates some simple statistical information on the number of external inet connections. This can be used to detect if one IP address is taking up an excessive number of connections. It can also send an email alert if a given IP address exceeds a threshold between runs of the script. This script can be used as a drop-in Munin plugin or it can be used stand-alone from cron. I used this on a busy web server that would sometimes get hit with denial of service attacks. This made it easy to see if a script was opening many multiple connections. A typical browser would open fewer than 10 connections at once. A script might open over 100 simultaneous connections.

hive.py This script creates SSH connections to a list of hosts that you provide. Then you are given a command line prompt. Each shell command that you enter is sent to all the hosts. The response from each host is collected and printed. For example, you could connect to a dozen different machines and reboot them all at once.

script.py This implements a command similar to the classic BSD "script" command. This will start a subshell and log all input and output to a file. This demonstrates the `interact()` method of Pexpect.

ftp.py This demonstrates an FTP "bookmark". This connects to an ftp site; does a few ftp tasks; and then gives the user interactive control over the session. In this case the "bookmark" is to a directory on the OpenBSD ftp server. It puts you in the i386 packages directory. You can easily modify this for other sites. This demonstrates the `interact()` method of Pexpect.

monitor.py This runs a sequence of commands on a remote host using SSH. It runs a simple system checks such as uptime and free to monitor the state of the remote host.

passmass.py This will login to each given server and change the password of the given user. This demonstrates scripting logins and passwords.

python.py This starts the python interpreter and prints the greeting message backwards. It then gives the user iteractive control of Python. It's pretty useless!

ssh_tunnel.py This starts an SSH tunnel to a remote machine. It monitors the connection and restarts the tunnel if it goes down.

uptime.py This will run the uptime command and parse the output into variables. This demonstrates using a single regular expression to match the output of a command and capturing different variable in match groups. The grouping regular expression handles a wide variety of different uptime formats.

CHAPTER 5

FAQ

Q: Where can I get help with pexpect? Is there a mailing list?

A: You can use the pexpect tag on Stackoverflow to ask questions specifically related to Pexpect. For more general Python support, there's the python-list mailing list, and the #python IRC channel. Please refrain from using github for general python or systems scripting support.

Q: Why don't shell pipe and redirect (| and >) work when I spawn a command?

A: Remember that Pexpect does NOT interpret shell meta characters such as redirect, pipe, or wild cards (>, |, or *). That's done by a shell not the command you are spawning. This is a common mistake. If you want to run a command and pipe it through another command then you must also start a shell. For example:

```
child = pexpect.spawn('/bin/bash -c "ls -l | grep LOG > log_list.txt"')
child.expect(pexpect.EOF)
```

The second form of spawn (where you pass a list of arguments) is useful in situations where you wish to spawn a command and pass it its own argument list. This can make syntax more clear. For example, the following is equivalent to the previous example:

```
shell_cmd = 'ls -l | grep LOG > log_list.txt'
child = pexpect.spawn('/bin/bash', ['-c', shell_cmd])
child.expect(pexpect.EOF)
```

Q: The 'before' and 'after' properties sound weird.

A: This is how the -B and -A options in grep works, so that made it easier for me to remember. Whatever makes my life easier is what's best. Originally I was going to model Pexpect after Expect, but then I found that I didn't actually like the way Expect did some things. It was more confusing. The *after* property can be a little confusing at first, because it will actually include the matched string. The *after* means after the point of match, not after the matched string.

Q: Why not just use Expect?

A: I love it. It's great. I has bailed me out of some real jams, but I wanted something that would do 90% of what I need from Expect; be 10% of the size; and allow me to write my code in Python instead of TCL. Pexpect is not nearly as big as Expect, but Pexpect does everything I have ever used Expect for. **Q: Why not just use a pipe (popen())?**

A: A pipe works fine for getting the output to non-interactive programs. If you just want to get the output from ls, uname, or ping then this works. Pipes do not work very well for interactive programs and pipes will almost certainly fail for most applications that ask for passwords such as telnet, ftp, or ssh.

There are two reasons for this.

- First an application may bypass stdout and print directly to its controlling TTY. Something like SSH will do this when it asks you for a password. This is why you cannot redirect the password prompt because it does not go through stdout or stderr.
- The second reason is because most applications are built using the C Standard IO Library (anything that uses `#include <stdio.h>`). One of the features of the stdio library is that it buffers all input and output. Normally output is line buffered when a program is printing to a TTY (your terminal screen). Everytime the program prints a line-feed the currently buffered data will get printed to your screen. The problem comes when you connect a pipe. The stdio library is smart and can tell that it is printing to a pipe instead of a TTY. In that case it switches from line buffer mode to block buffered. In this mode the currently buffered data is flushed when the buffer is full. This causes most interactive programs to deadlock. Block buffering is more efficient when writing to disks and pipes. Take the situation where a program prints a message `"Enter your user name:\n"` and then waits for you type type something. In block buffered mode, the stdio library will not put the message into the pipe even though a linefeed is printed. The result is that you never receive the message, yet the child application will sit and wait for you to type a response. Don't confuse the stdio lib's buffer with the pipe's buffer. The pipe buffer is another area that can cause problems. You could flush the input side of a pipe, whereas you have no control over the stdio library buffer.

More information: the Standard IO library has three states for a `FILE *`. These are: _IOFBF for block buffered; _IOLBF for line buffered; and _IONBF for unbuffered. The STDIO lib will use block buffering when talking to a block file descriptor such as a pipe. This is usually not helpful for interactive programs. Short of recompiling your program to include fflush() everywhere or recompiling a custom stdio library there is not much a controlling application can do about this if talking over a pipe.

The program may have put data in its output that remains unflushed because the output buffer is not full; then the program will go and deadlock while waiting for input – because you never send it any because you are still waiting for its output (still stuck in the STDIO's output buffer).

The answer is to use a pseudo-tty. A TTY device will force line buffering (as opposed to block buffering). Line buffering means that you will get each line when the child program sends a line feed. This corresponds to the way most interactive programs operate – send a line of output then wait for a line of input.

I put "answer" in quotes because it's ugly solution and because there is no POSIX standard for pseudo-TTY devices (even though they have a TTY standard...). What would make more sense to me would be to have some way to set a mode on a file descriptor so that it will tell the STDIO to be line-buffered. I have investigated, and I don't think there is a way to set the buffered state of a child process. The STDIO Library does not maintain any external state in the kernel or whatnot, so I don't think there is any way for you to alter it. I'm not quite sure how this line-buffered/block-buffered state change happens internally in the STDIO library. I think the STDIO lib looks at the file descriptor and decides to change behavior based on whether it's a TTY or a block file (see isatty()).

I hope that this qualifies as helpful. Don't use a pipe to control another application.

Q: Can I do screen scraping with this thing?

A: That depends. If your application just does line-oriented output then this is easy. If a program emits many terminal sequences, from video attributes to screen addressing, such as programs using curses, then it may become very difficult to ascertain what text is displayed on a screen.

We suggest using the pyte library to screen-scrape. The module `pexpect.ANSI` released with previous versions of pexpect is now marked deprecated and may be removed in the future.

Q: I get strange behavior with pexect and gevent

A: Pexpect uses fork(2), exec(2), select(2), waitpid(2), and implements its own selector in expect family of calls.

pexpect has been known to misbehave when paired with gevent. A solution might be to isolate your pexpect dependent code from any frameworks that manipulate event selection behavior by running it in an another process entirely.

CHAPTER 6

Common problems

Threads

On Linux (RH 8) you cannot spawn a child from a different thread and pass the handle back to a worker thread. The child is successfully spawned but you can't interact with it. The only way to make it work is to spawn and interact with the child all in the same thread. [Adam Kerrison]

Timing issue with send() and sendline()

This problem has been addressed and should not affect most users.

It is sometimes possible to read an echo of the string sent with *send()* and *sendline()*. If you call *send()* and then immediately call *readline()*, you may get part of your output echoed back. You may read back what you just wrote even if the child application does not explicitly echo it. Timing is critical. This could be a security issue when talking to an application that asks for a password; otherwise, this does not seem like a big deal. But why do TTYs do this?

People usually report this when they are trying to control SSH or some other login. For example, if your code looks something like this:

```
child.expect ('[pP]assword:')
child.sendline (my_password)
```

1. SSH prints "password:" prompt to the user.
2. SSH turns off echo on the TTY device.
3. SSH waits for user to enter a password.

When scripting with Pexpect what can happen is that Pexpect will respond to the "password:" prompt before SSH has had time to turn off TTY echo. In other words, Pexpect sends the password between steps 1. and 2., so the password gets echoed back to the TTY. I would call this an SSH bug.

Pexpect now automatically adds a short delay before sending data to a child process. This more closely mimics what happens in the usual human-to-app interaction. The delay can be tuned with the `delaybeforesend` attribute of the spawn class. In general, this fixes the problem for everyone and so this should not be an issue for most users. For some applications you might with to turn it off:

```
child = pexpect.spawn ("ssh user@example.com")
child.delaybeforesend = None
```

Truncated output just before child exits

So far I have seen this only on older versions of Apple's MacOS X. If the child application quits it may not flush its output buffer. This means that your Pexpect application will receive an EOF even though it should have received a little more data before the child died. This is not generally a problem when talking to interactive child applications. One example where it is a problem is when trying to read output from a program like *ls*. You may receive most of the directory listing, but the last few lines will get lost before you receive an EOF. The reason for this is that *ls* runs; completes its task; and then exits. The buffer is not flushed before exit so the last few lines are lost. The following example demonstrates the problem:

```
child = pexpect.spawn('ls -l')
child.expect(pexpect.EOF)
print child.before
```

Controlling SSH on Solaris

Pexpect does not yet work perfectly on Solaris. One common problem is that SSH sometimes will not allow TTY password authentication. For example, you may expect SSH to ask you for a password using code like this:

```
child = pexpect.spawn('ssh user@example.com')
child.expect('password')
child.sendline('mypassword')
```

You may see the following error come back from a spawned child SSH:

```
Permission denied (publickey,keyboard-interactive).
```

This means that SSH thinks it can't access the TTY to ask you for your password. The only solution I have found is to use public key authentication with SSH. This bypasses the need for a password. I'm not happy with this solution. The problem is due to poor support for Solaris Pseudo TTYs in the Python Standard Library.

child does not receive full input, emits BEL

You may notice when running for example cat(1) or base64(1), when sending a very long input line, that it is not fully received, and the BEL ('a') may be found in output.

By default the child terminal matches the parent, which is often in "canonical mode processing". You may wish to disable this mode. The exact limit of a line varies by operating system, and details of disabling canonical mode may be found in the docstring of *send()*.

CHAPTER 7

History

Releases

Version 4.2.1

- Fix to allow running `env` in replwrap-ed bash.
- Raise more informative exception from pxssh if it fails to connect.
- Change `passmass` example to not log passwords entered.

Version 4.2

- Change: When an `env` parameter is specified to the *`spawn`* or `run` family of calls containing a value for `PATH`, its value is used to discover the target executable from a relative path, rather than the current process's environment `PATH`. This mirrors the behavior of `subprocess.Popen()` in the standard library (#348).
- Regression: Re-introduce capability for `read_nonblocking()` in class `fdspawn` as previously supported in version 3.3 (#359).

Version 4.0

- Integration with `asyncio`: passing `async=True` to *`expect()`*, *`expect_exact()`* or *`expect_list()`* will make them return a coroutine. You can get the result using `yield from`, or wrap it in an `asyncio.Task`. This allows the event loop to do other things while waiting for output that matches a pattern.
- Experimental support for Windows (with some caveats)—see *Pexpect on Windows*.
- Enhancement: allow method as callbacks of argument `events` for *`pexpect.run()`* (#176).
- It is now possible to call *`wait()`* multiple times, or after a process is already determined to be terminated without raising an exception (PR #211).

- New *pexpect.spawn* keyword argument, `dimensions=(rows, columns)` allows setting terminal screen dimensions before launching a program (#122).
- Fix regression that prevented executable, but unreadable files from being found when not specified by absolute path – such as /usr/bin/sudo (#104).
- Fixed regression when executing pexpect with some prior releases of the multiprocessing module where stdin has been closed (#86).

Backwards incompatible changes

- Deprecated `pexpect.screen` and `pexpect.ANSI`. Please use other packages such as pyte to emulate a terminal.
- Removed the independent top-level modules (`pxssh fdpexpect FSM screen ANSI`) which were installed alongside Pexpect. These were moved into the Pexpect package in 3.0, but the old names were left as aliases.
- Child processes created by Pexpect no longer ignore SIGHUP by default: the `ignore_sighup` parameter of *pexpect.spawn* defaults to False. To get the old behaviour, pass `ignore_sighup=True`.

Version 3.3

- Added a mechanism to wrap REPLs, or shells, in an object which can conveniently be used to send commands and wait for the output (*pexpect.replwrap*).
- Fixed issue where pexpect would attempt to execute a directory because it has the 'execute' bit set (#37).
- Removed the `pexpect.psh` module. This was never documented, and we found no evidence that people use it. The new *pexpect.replwrap* module provides a more flexible alternative.
- Fixed `TypeError: got <type 'str'> ('\r\n') as pattern` in `spawnu.readline()` method (#67).
- Fixed issue where EOF was not correctly detected in *interact()*, causing a repeating loop of output on Linux, and blocking before EOF on BSD and Solaris (#49).
- Several Solaris (SmartOS) bugfixes, preventing `IOError` exceptions, especially when used with cron(1) (#44).
- Added new keyword argument `echo=True` for `spawn`. On SVR4-like systems, the method `isatty()` will always return *False*: the child pty does not appear as a terminal. Therefore, *setecho()*, *getwinsize()*, *setwinsize()*, and *waitnoecho()* are not supported on those platforms.

After this, we intend to start working on a bigger refactoring of the code, to be released as Pexpect 4. There may be more bugfix 3.x releases, however.

Version 3.2

- Fix exception handling from `select.select()` on Python 2 (PR #38). This was accidentally broken in the previous release when it was fixed for Python 3.
- Removed a workaround for `TIOCSWINSZ` on very old systems, which was causing issues on some BSD systems (PR #40).
- Fixed an issue with exception handling in *pxssh* (PR #43)

The documentation for *pxssh* was improved.

Version 3.1

- Fix an issue that prevented importing pexpect on Python 3 when `sys.stdout` was reassigned (#30).
- Improve prompt synchronisation in *pxssh* (PR #28).
- Fix pickling exception instances (PR #34).
- Fix handling exceptions from `select.select()` on Python 3 (PR #33).

The examples have also been cleaned up somewhat - this will continue in future releases.

Version 3.0

The new major version number doesn't indicate any deliberate API incompatibility. We have endeavoured to avoid breaking existing APIs. However, pexpect is under new maintenance after a long dormancy, so some caution is warranted.

- A new *unicode API* was introduced.
- Python 3 is now supported, using a single codebase.
- Pexpect now requires at least Python 2.6 or 3.2.
- The modules other than pexpect, such as *pexpect.fdpexpect* and *pexpect.pxssh*, were moved into the pexpect package. For now, wrapper modules are installed to the old locations for backwards compatibility (e.g. `import pxssh` will still work), but these will be removed at some point in the future.
- Ignoring `SIGHUP` is now optional - thanks to Kimmo Parviainen-Jalanko for the patch.

We also now have docs on ReadTheDocs, and continuous integration on Travis CI.

Version 2.4

- Fix a bug regarding making the pty the controlling terminal when the process spawning it is not, actually, a terminal (such as from cron)

Version 2.3

- Fixed OSError exception when a pexpect object is cleaned up. Previously, you might have seen this exception:

```
Exception exceptions.OSError: (10, 'No child processes')
in <bound method spawn.__del__ of <pexpect.spawn instance at 0xd248c>> ignored
```

  You should not see that anymore. Thanks to Michael Surette.

- Added support for buffering reads. This greatly improves speed when trying to match long output from a child process. When you create an instance of the spawn object you can then set a buffer size. For now you MUST do the following to turn on buffering – it may be on by default in future version:

```
child = pexpect.spawn ('my_command')
child.maxread=1000 # Sets buffer to 1000 characters.
```

- I made a subtle change to the way TIMEOUT and EOF exceptions behave. Previously you could either expect these states in which case pexpect will not raise an exception, or you could just let pexpect raise an exception when these states were encountered. If you expected the states then the `before` property was set to everything before the state was encountered, but if you let pexpect raise the exception then `before` was not set. Now, the `before` property will get set either way you choose to handle these states.

- The spawn object now provides iterators for a *file-like interface*. This makes Pexpect a more complete file-like object. You can now write code like this:

```
child = pexpect.spawn ('ls -l')
for line in child:
    print line
```

- write and writelines() no longer return a value. Use send() if you need that functionality. I did this to make the Spawn object more closely match a file-like object.
- Added the attribute `exitstatus`. This will give the exit code returned by the child process. This will be set to `None` while the child is still alive. When `isalive()` returns 0 then `exitstatus` will be set.
- Made a few more tweaks to `isalive()` so that it will operate more consistently on different platforms. Solaris is the most difficult to support.
- You can now put `TIMEOUT` in a list of expected patterns. This is just like putting `EOF` in the pattern list. Expecting for a `TIMEOUT` may not be used as often as `EOF`, but this makes Pexpect more consistent.
- Thanks to a suggestion and sample code from Chad J. Schroeder I added the ability for Pexpect to operate on a file descriptor that is already open. This means that Pexpect can be used to control streams such as those from serial port devices. Now, you just pass the integer file descriptor as the "command" when constructing a spawn open. For example on a Linux box with a modem on ttyS1:

```
fd = os.open("/dev/ttyS1", os.O_RDWR|os.O_NONBLOCK|os.O_NOCTTY)
m = pexpect.spawn(fd) # Note integer fd is used instead of usual string.
m.send("+++") # Escape sequence
m.send("ATZ0\r") # Reset modem to profile 0
rval = m.expect(["OK", "ERROR"])
```

- `read()` was renamed to `read_nonblocking()`. Added new `read()` method that matches file-like object interface. In general, you should not notice the difference except that `read()` no longer allows you to directly set the timeout value. I hope this will not effect any existing code. Switching to `read_nonblocking()` should fix existing code.
- Changed the name of `set_echo()` to `setecho()`.
- Changed the name of `send_eof()` to `sendeof()`.
- Modified `kill()` so that it checks to make sure the pid `isalive()`.
- modified `spawn()` (really called from `__spawn()`) so that it does not raise an expection if `setwinsize()` fails. Some platforms such as Cygwin do not like setwinsize. This was a constant problem and since it is not a critical feature I decided to just silence the error. Normally I don't like to do that, but in this case I'm making an exception.
- Added a method `close()` that does what you think. It closes the file descriptor of the child application. It makes no attempt to actually kill the child or wait for its status.
- Add variables `__version__` and `__revision__` (from cvs) to the pexpect modules. This is mainly helpful to me so that I can make sure that I'm testing with the right version instead of one already installed.
- `log_open()` and `log_close(` have been removed. Now use `setlog()`. The `setlog()` method takes a file object. This is far more flexible than the previous log method. Each time data is written to the file object it will be flushed. To turn logging off simply call `setlog()` with None.
- renamed the `isAlive()` method to `isalive()` to match the more typical naming style in Python. Also the technique used to detect child process status has been drastically modified. Previously I did some funky stuff with signals which caused indigestion in other Python modules on some platforms. It was a big headache. It still is, but I think it works better now.

- attribute `matched` renamed to `after`
- new attribute `match`
- The `expect_eof()` method is gone. You can now simply use the `expect()` method to look for EOF.
- **Pexpect works on OS X**, but the nature of the quirks cause many of the tests to fail. See bugs. (Incomplete Child Output). The problem is more than minor, but Pexpect is still more than useful for most tasks.
- **Solaris**: For some reason, the *second* time a pty file descriptor is created and deleted it never gets returned for use. It does not effect the first time or the third time or any time after that. It's only the second time. This is weird... This could be a file descriptor leak, or it could be some peculiarity of how Solaris recycles them. I thought it was a UNIX requirement for the OS to give you the lowest available filedescriptor number. In any case, this should not be a problem unless you create hundreds of pexpect instances... It may also be a pty module bug.

Moves and forks

- Pexpect development used to be hosted on Sourceforge.
- In 2011, Thomas Kluyver forked pexpect as 'pexpect-u', to support Python 3. He later decided he had taken the wrong approach with this.
- In 2012, Noah Spurrier, the original author of Pexpect, moved the project to Github, but was still too busy to develop it much.
- In 2013, Thomas Kluyver and Jeff Quast forked Pexpect again, intending to call the new fork Pexpected. Noah Spurrier agreed to let them use the name Pexpect, so Pexpect versions 3 and above are based on this fork, which now lives here on Github.

Pexpect is developed on Github. Please report issues there as well.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

B

C

E

F

G

I

K

L

O

P

R

S

T

W