We created sshlogin.py here, which tries to ssh-login into a target IP. On most basic linux systems, if you ssh into a target IP from another, it will ask you for info such as "do you want to continue", "password", etc. PEXPECT is a method that allows us to expect a sys response, and then respond to that response.

# `pexpect` — Spawn child applications and control them automatically.

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 Pexpect – the function, run() and the class, spawn. You can call the run() function to execute a command and return the output. This is a handy replacement for os.system().

For example:

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

The more powerful interface is the spawn class. You can use this to spawn an external child command and then interact with the child by sending lines and expecting responses.

For example:

```
child = pexpect.spawn('scp foo myname@host.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.

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 (Let me know if I forgot anyone.)

Free, open source, and all that good stuff.



$Id: pexpect.py 516 2008-05-23 20:46:01Z noah $

*exception* `pexpect.`**`ExceptionPexpect`**(*value*)

Base class for all exceptions raised by this module.

**`get_trace`**()

This returns an abbreviated stack trace with lines that only concern the caller. In other words, the stack trace inside the Pexpect module is not included.

*exception* `pexpect.`**`EOF`**(*value*)

Raised when EOF is read from a child. This usually means the child has exited.

*exception* `pexpect.`**`TIMEOUT`**(*value*)

Raised when a read time exceeds the timeout.

*class* pexpect.**spawn**(*command*, *args=[]*, *timeout=30*, *maxread=2000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*)

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

**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).

**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(clp, timeout)
   ...
```

**eof**()

This returns True if the EOF exception was ever raised.

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

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
```

After a match is found the instance attributes ‘before’, ‘after’ and ‘match’ will be set. You can see all the data read before the match in ‘before’. You can see the data that was matched in ‘after’. The re.MatchObject used in the re match will be in ‘match’. If an error occurred then ‘before’ will be set to all the data read so far and ‘after’ and ‘match’ will be None.

If timeout is -1 then timeout will be set to the self.timeout value.

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().

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

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.

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

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(). If timeout==-1 then the self.timeout value is used. If searchwindowsize==-1 then the self.searchwindowsize value is used.

**expect_loop**(*searcher*, *timeout=-1*, *searchwindowsize=-1*)

This is the common loop used inside expect. The 'searcher' should be an instance of searcher_re or searcher_string, which describes how and what to search for in the input.

See expect() for other arguments, return value and exceptions.

**fileno**()

This returns the file descriptor of the pty for the child.

**flush**()

This does nothing. It is here to support the interface for a File-like object.

**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().

**getwinsize**()

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

**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 stop. The default for escape_character is ^]. This should not be confused with ASCII 27 – the ESC character. ASCII 29 was chosen for historical merit because this is the character used by 'telnet' as the escape character. The escape_character will not be sent to the child process.

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.TIOC
    global p
    p.setwinsize(a[0],a[1])
p = pexpect.spawn('/bin/bash') # Note this is global and used in sigwinch_p
signal.signal(signal.SIGWINCH, sigwinch_passthrough)
p.interact()
```

**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.

**isatty**()

This returns True if the file descriptor is open and connected to a tty(-like) device, else False.

**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.

**next**()

This is to support iterators over a file-like object.

**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.

**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 log file was set using setlog() then all data will also be written to the log file.

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 was 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 effected 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.

**`readline`**(*size=-1*)

This reads and returns one entire line. A trailing newline is kept in the string, but may be absent when a file ends with an incomplete line. Note: This readline() looks for a rn pair even on UNIX because this is what the pseudo tty device returns. So contrary to what you may expect you will receive the newline as rn. An empty string is returned when EOF is hit immediately. Currently, the size argument is mostly ignored, so this behavior is not standard for a file-like object. If size is 0 then an empty string is returned.

**`readlines`**(*sizehint=-1*)

This reads until EOF using readline() and returns a list containing the lines thus read. The optional "sizehint" argument is ignored.

**`send`**(*s*)

This sends a string to the child process. This returns the number of bytes written. If a log file was set then the data is also written to the log.

**`sendcontrol`**(*char*)

This sends a control character to the child such as Ctrl-C or Ctrl-D. For

example, to send a Ctrl-G (ASCII 7):

```
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.

**sendline**(*s=''*)

This is like send(), but it adds a line feed (os.linesep). This returns the number of bytes written.

**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')
p.sendline ('1234') # We will see this twice (once from tty echo and again
p.expect (['1234'])
p.expect (['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 (['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') # We will see this twice (once from tty echo and again
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'])
```

**setlog**(*fileobject*)

This method is no longer supported or allowed.

**setmaxread**(*maxread*)

This method is no longer supported or allowed. I don't like getters and setters without a good reason.

**setwinsize**(*r*, *c*)

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.

**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.

**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, technically, the child is still alive until its output is read.

**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 is None then this method to block forever until ECHO flag is False.

**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.

pexpect.**run**(*command*, *timeout=-1*, *withexitstatus=False*, *events=None*, *extra_args=None*, *logfile=None*, *cwd=None*, *env=None*)

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 pseudo ttys. 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 myname@host.example.com:.')
child.expect ('(?i)password')
child.sendline (mypassword)
```

The previous code can be replace with the following:

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

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
```

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
```

The 'events' argument should be a dictionary of patterns and responses. Whenever one of the patterns is seen in the command out run() will send the associated response string. Note that you should put newlines in your string if Enter is necessary. The responses may also contain callback functions. Any callback is function that takes a dictionary 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.

pexpect.**which**(*filename*)

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.