wexpect

Release 4.0.1

Sep 01, 2023

Contents

Wexpect is a Windows variant of Pexpect Wexpect and Pexpect makes Python a better tool for controlling other applications.

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

Wexpect 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. Wexpect highly depends on Mark Hammond's pywin32 which provides access to many of the Windows APIs from Python.

CHAPTER 1

Install

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

```
pip install wexpect
```

CHAPTER 2

Hello Wexpect

To interract with a child process use `spawn` method:

```
import wexpect
child = wexpect.spawn('cmd.exe')
child.expect('>')
child.sendline('ls')
child.expect('>')
print(child.before)
child.sendline('exit')
```

For more information see examples folder.

Contents:

2.1 API documentation

2.1.1 Wexpect symbols

Wexpect package has the following symbols. (Exported by `__all__` in code:*__init__.py*)

spawn

This is the main class interface for Wexpect. Use this class to start and control child applications. There are two implementation: *`wexpect.host.SpawnPipe`* uses Windows-Pipe for communicate child. `wexpect.SpawnSocket` uses TCP socket. Choose the default implementation with `WEXPECT_SPAWN_CLASS` environment variable, or the *`wexpect.host.SpawnPipe`* will be chosen by default.

SpawnPipe

`wexpect.host.SpawnPipe` is the default spawn class, but you can access it directly with its exact name.

SpawnSocket

`wexpect.host.SpawnSocket` is the secondary spawn class, you can access it directly with its exact name or by setting the `WEXPECT_SPAWN_CLASS` environment variable to `SpawnSocket`

run

wexpect.host.run() runs the given command; waits for it to finish; then returns all output as a string. This function is similar to `os.system()`.

EOF

wexpect.wexpect_util.EOF is an exception. This usually means the child has exited.

TIMEOUT

wexpect.wexpect_util.TIMEOUT raised when a read time exceeds the timeout.

__version__

This gives back the version of the wexpect release. Versioning is handled by the pbr package, which derives it from Git tags.

spawn_class_name

Contains the default spawn class' name even if the user has not specified it. The value can be `SpawnPipe` or `SpawnSocket`

ConsoleReaderSocket

For advanced users only! *wexpect.console_reader.ConsoleReaderSocket*

ConsoleReaderPipe

For advanced users only! *wexpect.console_reader.ConsoleReaderPipe*

2.1.2 Wexpect modules

Host

Host module contains classes and functions for the host application. These will spawn the child application. These host classes (and some util classes) are the interface for the user. Handle other modules as protected.

Functions

`host.`**`run`**(*timeout=-1*, *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 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:

```
child = spawn('scp foo myname@host.example.com:.')
child.expect ('(?i)password')
child.sendline (mypassword)
```

The previous code can be replace with the following:

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

Examples

Start the apache daemon on the local machine:

```
run ("/usr/local/apache/bin/apachectl start")
```

Check in a file using SVN:

```
run ("svn ci -m 'automatic commit' my_file.py")
```

Run a command and capture exit status:

```
(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 wexpect 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 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.

SpawnPipe

class wexpect.host.**SpawnPipe**(*command*, *args=[]*, *timeout=30*, *maxread=60000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*, *codepage=None*, *echo=True*, *interact=False*, ***kwargs*)

__init__(*command*, *args=[]*, *timeout=30*, *maxread=60000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*, *codepage=None*, *echo=True*, *interact=False*, ***kwargs*)

This starts the given command in a child process. This does all the fork/exec type of stuff for a pty. This is called by __init__. If args is empty then command will be parsed (split on spaces) and args will be set to parsed arguments.

The pid and child_fd of this object get set by this method. Note that it is difficult for this method to fail. You cannot detect if the child process cannot start. So the only way you can tell if the child process started or not is to try to read from the file descriptor. If you get EOF immediately then it means that the child is already dead. That may not necessarily be bad because you may haved spawned a child that performs some task; creates no stdout output; and then dies.

expect (*pattern*, *timeout=-1*, *searchwindowsize=None*)

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', wexpect.EOF, wexpect.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 = wexpect.spawn('/bin/ls')
p.expect (wexpect.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.

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)
   ...
```

send (*s*, *delaybeforesend=None*)
Virtual definition

sendline (*s=''*)
This is like send(), but it adds a line feed (os.linesep). This returns the 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.

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.

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. 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.

read_nonblocking (*size=1*)
This reads at most size characters from the child application. If the end of file is read then an EOF exception will be raised.

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 Wtty.read().

SpawnSocket

class wexpect.host.**SpawnSocket** (*command*, *args=[]*, *timeout=30*, *maxread=60000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*, *codepage=None*, *echo=True*, *port=4321*, *host='127.0.0.1'*, *interact=False*, ***kwargs*)

__init__ (*command*, *args=[]*, *timeout=30*, *maxread=60000*, *searchwindowsize=None*, *logfile=None*, *cwd=None*, *env=None*, *codepage=None*, *echo=True*, *port=4321*, *host='127.0.0.1'*, *interact=False*, ***kwargs*)
This starts the given command in a child process. This does all the fork/exec type of stuff for a pty. This is called by __init__. If args is empty then command will be parsed (split on spaces) and args will be set to parsed arguments.

The pid and child_fd of this object get set by this method. Note that it is difficult for this method to fail. You cannot detect if the child process cannot start. So the only way you can tell if the child process started or not is to try to read from the file descriptor. If you get EOF immediately then it means that the child is already dead. That may not necessarily be bad because you may haved spawned a child that performs some task; creates no stdout output; and then dies.

expect (*pattern*, *timeout=-1*, *searchwindowsize=None*)
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', wexpect.EOF, wexpect.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 = wexpect.spawn('/bin/ls')
p.expect (wexpect.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.

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)
   ...
```

send (*s*, *delaybeforesend=None*)
Virtual definition

sendline (*s=''*)
This is like send(), but it adds a line feed (os.linesep). This returns the 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.

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.

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. 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.

read_nonblocking(*size=1*)
This reads at most size characters from the child application. If the end of file is read then an EOF exception will be raised.

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 Wtty.read().

Wexpect util

Wexpect is a Windows variant of pexpect https://pexpect.readthedocs.io.

Wexpect is a Python module for spawning child applications and controlling them automatically.

wexpect util contains small functions, and classes, which are used in multiple classes. The command line argument parsers, and the Exceptions placed here.

Functions

wexpect_util.**str2bool**()

wexpect_util.**spam**(*message*, **args*, ***kws*)
Very verbose debug dunction.

wexpect_util.**init_logger**()
Initializes the logger. I wont measure coverage for this debug method.

wexpect_util.**split_command_line**(*escape_char='^'*)
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.

wexpect_util.**join_args**()
Joins arguments a command line. It quotes all arguments that contain spaces or any of the characters ^!$%&()[]{}=;'+,'~

ExceptionPexpect

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

EOF

class wexpect.wexpect_util.**EOF**(*value*)
Raised when EOF is read from a child. This usually means the child has exited. The user can wait to EOF, which means he waits the end of the execution of the child process.

TIMEOUT

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

Console reader

Wexpect is a Windows variant of pexpect https://pexpect.readthedocs.io.

Wexpect is a Python module for spawning child applications and controlling them automatically.

console_reader Implements a virtual terminal, and starts the child program. The main wexpect.spawn class connect to this class to reach the child's terminal.

ConsoleReaderPipe

class wexpect.console_reader.**ConsoleReaderPipe**(*path*, *host_pid*, *codepage=None*, *window_size_x=80*, *window_size_y=25*, *buffer_size_x=80*, *buffer_size_y=16000*, *local_echo=True*, *interact=False*, ***kwargs*)

__init__(*path*, *host_pid*, *codepage=None*, *window_size_x=80*, *window_size_y=25*, *buffer_size_x=80*, *buffer_size_y=16000*, *local_echo=True*, *interact=False*, ***kwargs*)
Initialize the console starts the child in it and reads the console periodically.

Args: path (str): Child's executable with arguments. parent_pid (int): Parent (aka. host) process process-ID codepage (:obj:, optional): Output console code page.

read_loop()

suspend_child()
Pauses the main thread of the child process.

resume_child()
Un-pauses the main thread of the child process.

refresh_console()
Clears the console after pausing the child and reading all the data currently on the console.

terminate_child()

isalive(*process*)
True if the child is still alive, false otherwise

write(*s*)
Writes input into the child consoles input buffer.

createKeyEvent(*char*)
Creates a single key record corrosponding to the ascii character char.

initConsole(*consout=None*, *window_size_x=80*, *window_size_y=25*, *buffer_size_x=80*, *buffer_size_y=16000*)

parseData(*s*)
Ensures that special characters are interpretted as newlines or blanks, depending on if there written over characters or screen-buffer-fill characters.

getConsoleOut()

getCoord (*offset*)
Converts an offset to a point represented as a tuple.

getOffset (*coord*)
Converts a tuple-point to an offset.

readConsole (*startCo*, *endCo*)
Reads the console area from startCo to endCo and returns it as a string.

readConsoleToCursor ()
Reads from the current read position to the current cursor position and inserts the string into self.__buffer.

interact ()
Displays the child console for interaction.

sendeof ()
This sends an EOF to the host. This sends a character which inform the host that child has been finished, and all of it's output has been send to host.

create_connection (*timeout=-1*, ***kwargs*)

close_connection ()

send_to_host (*msg*)

get_from_host ()

ConsoleReaderSocket

class wexpect.console_reader.**ConsoleReaderSocket** (*path*, *host_pid*, *codepage=None*, *window_size_x=80*, *window_size_y=25*, *buffer_size_x=80*, *buffer_size_y=16000*, *local_echo=True*, *interact=False*, ***kwargs*)

__init__ (*path*, *host_pid*, *codepage=None*, *window_size_x=80*, *window_size_y=25*, *buffer_size_x=80*, *buffer_size_y=16000*, *local_echo=True*, *interact=False*, ***kwargs*)
Initialize the console starts the child in it and reads the console periodically.

Args: path (str): Child's executable with arguments. parent_pid (int): Parent (aka. host) process process-ID codepage (:obj:, optional): Output console code page.

read_loop ()

suspend_child ()
Pauses the main thread of the child process.

resume_child ()
Un-pauses the main thread of the child process.

refresh_console ()
Clears the console after pausing the child and reading all the data currently on the console.

terminate_child ()

isalive (*process*)
True if the child is still alive, false otherwise

write (*s*)
Writes input into the child consoles input buffer.

createKeyEvent(*char*)
Creates a single key record corrosponding to the ascii character char.

initConsole(*consout=None, window_size_x=80, window_size_y=25, buffer_size_x=80, buffer_size_y=16000*)

parseData(*s*)
Ensures that special characters are interpretted as newlines or blanks, depending on if there written over characters or screen-buffer-fill characters.

getConsoleOut()

getCoord(*offset*)
Converts an offset to a point represented as a tuple.

getOffset(*coord*)
Converts a tuple-point to an offset.

readConsole(*startCo, endCo*)
Reads the console area from startCo to endCo and returns it as a string.

readConsoleToCursor()
Reads from the current read position to the current cursor position and inserts the string into self.__buffer.

interact()
Displays the child console for interaction.

sendeof()
This sends an EOF to the host. This sends a character which inform the host that child has been finished, and all of it's output has been send to host.

create_connection(***kwargs*)

close_connection()

send_to_host(*msg*)

get_from_host()

2.2 History

Wexpect was a one-file code developed at University of Washington. There were several copy and reference to this code with very few (almost none) documentation nor integration.

This project fixes these limitations, with example codes, tests, and pypi integration.

2.2.1 Refactor

The original wexpect was a monolith, one-file code, with several structural weaknesses. This leads me to rewrite the whole code. The first variant of the new structure is delivered with v3.2.0. (The default is the old variant (`legacy_wexpect`) in v3.2.0. `WEXPECT_SPAWN_CLASS` environment variable can choose the new-structured implementation.) Now `SpawnPipe` is the default spawn class.

2.2.2 Old vs new

But what is the difference between the old and new and what was the problem with the old?

Generally, wexpect (both old and new) has three processes:

- *host* is our original python script/program, which want to launch the child.
- *console* is a process which started by the host, and launches the child. (This is a python script)
- *child* is the process which want to be launched.

The child and the console has a common Windows console, distict from the host.

The `legacy_wexpect`'s console is a thin script, almost do nothing. It initializes the Windows's console, and monitors the host and child processes. The magic is done by the host process, which has the `switchTo()` and `switchBack()` functions, which (de-) attaches the *child-console* Windows-console. The host manipulates the child's console directly. This direct manipulation is the main structural weakness. The following task/use-cases are hard/impossible:

- thread-safe multiprocessing of the host.
- logging (both console and host)
- using in graphical IDE or with pytest
- This variant is highly depends on the pywin32 package.

The new structure's console is a thick script. The console process do the major console manipulation, which is controlled by the host via socket (see SpawnSocket) or named-pipe (SpawnPipe). The host only process the except-loops.

2.3 Examples

hello_wexpect.py

This is the simplest example. It starts a windows command interpreter (aka. cmd) lists the current directory and exits.

```
import wexpect

# Start cmd as child process
child = wexpect.spawn('cmd.exe')

# Wait for prompt when cmd becomes ready.
child.expect('>')

# Prints the cmd's start message
print(child.before, end='')
print(child.after, end='')

# run list directory command
child.sendline('ls')

# Waiting for prompt
child.expect('>')

# Prints content of the directory
print(child.before, end='')
print(child.after, end='')

# Exit from cmd
child.sendline('exit')
```

(continues on next page)

(continued from previous page)

```
# Waiting for cmd termination.
child.wait()
```

terminaton.py

This script shows three method to terminate your application. *terminate_programmatically()* shows the recommended way, which kills the child by sending the application specific exit command. After that waiting for the terminaton is recommended. *terminate_eof()* shows how to terminate child program by sending EOF character. Some program can be terminated by sending EOF character. Waiting for the terminaton is recommended in this case too. *terminate_terminate()* shows how to terminate child program by sending kill signal. Some application requires sending SIGTERM to kill the child process. *terminate()* call *kill()* function, which sends SIGTERM to child process. Waiting for the terminaton is not required explicitly in this case. The wait is included in *terminate()* function.

```
import wexpect

def terminate_programmatically():
    '''Terminate child program by command. This is the recommended method.␣
↪Send your application's
    exit command to quit the child's process. After that wait for the␣
↪terminaton.
    '''
    print('terminate_programmatically')

    # Start cmd as child process
    child = wexpect.spawn('cmd.exe')

    # Wait for prompt when cmd becomes ready.
    child.expect('>')

    # Exit from cmd
    child.sendline('exit')

    # Waiting for cmd termination.
    child.wait()

def terminate_eof():
    '''Terminate child program by sending EOF character. Some program can be␣
↪terminated by sending
    an EOF character. Waiting for the terminaton is recommended in this case␣
↪too.
    '''
    print('terminate_eof')

    # Start cat as child process
    child = wexpect.spawn('cat')

    # Exit by sending EOF
    child.sendeof()

    # Waiting for cmd termination.
    child.wait()


def terminate_terminate():
    '''Terminate child program by sending kill signal. Some application␣
↪requires sending SIGTERM to kill
```

(continues on next page)

(continued from previous page)

```
    the child process. `terminate()` call `kill()` function, which sends␣
↪SIGTERM to child process.
    Waiting for the terminaton is not required explicitly in this case. The␣
↪wait is included in
    `terminate()` function.
    '''
    print('terminate_terminate')

    # Start cmd as child process
    child = wexpect.spawn('cmd.exe')

    # Wait for prompt when cmd becomes ready.
    child.expect('>')

    # Exit from cmd
    child.terminate()


terminate_programmatically()
terminate_eof()
terminate_terminate()
```

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

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

W

Index

Symbols

C

E

G

I