:mod:`pty` --- Pseudo-terminal utilities

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]pty.rst, line 1); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]pty.rst, line 4)

Unknown directive type "module".

```
.. module:: pty
   :platform: Unix
   :synopsis: Pseudo-Terminal Handling for Unix.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]pty.rst, line 8)

Unknown directive type "moduleauthor".

```
.. moduleauthor:: Steen Lumholt
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]pty.rst, line 9)

Unknown directive type "sectionauthor".

```
.. sectionauthor:: Moshe Zadka <moshez@zadka.site.co.il>
```

Source code: :source:`Lib/pty.py`

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]pty.rst, line 11); *backlink*

Unknown interpreted text role "source".

The :mod:`pty` module defines operations for handling the pseudo-terminal concept: starting another process and being able to write to and read from its controlling terminal programmatically.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]pty.rst, line 15); *backlink*

Unknown interpreted text role "mod".

Pseudo-terminal handling is highly platform dependent. This code is mainly tested on Linux, FreeBSD, and macOS (it is supposed to work on other POSIX platforms but it's not been thoroughly tested).

The :mod:`pty` module defines the following functions:

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]pty.rst, line 23); *backlink*

Unknown interpreted text role "mod".

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]pty.rst, line 26)

Unknown directive type "function".

```
.. function:: fork()

   Fork. Connect the child's controlling terminal to a pseudo-terminal. Return
   value is ``(pid, fd)``. Note that the child  gets *pid* 0, and the *fd* is
```

```
*invalid*. The parent's return value is the *pid* of the child, and *fd* is a
file descriptor connected to the child's controlling terminal (and also to the
child's standard input and output).
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]pty.rst, line 35)

Unknown directive type "function".

```
.. function:: openpty()

   Open a new pseudo-terminal pair, using :func:`os.openpty` if possible, or
   emulation code for generic Unix systems. Return a pair of file descriptors
   ``(master, slave)``, for the master and the slave end, respectively.
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]pty.rst, line 42)

Unknown directive type "function".

```
.. function:: spawn(argv[, master_read[, stdin_read]])

   Spawn a process, and connect its controlling terminal with the current
   process's standard io. This is often used to baffle programs which insist on
   reading from the controlling terminal. It is expected that the process
   spawned behind the pty will eventually terminate, and when it does *spawn*
   will return.

   A loop copies STDIN of the current process to the child and data received
   from the child to STDOUT of the current process. It is not signaled to the
   child if STDIN of the current process closes down.

   The functions *master_read* and *stdin_read* are passed a file descriptor
   which they should read from, and they should always return a byte string. In
   order to force spawn to return before the child process exits an
   empty byte array should be returned to signal end of file.

   The default implementation for both functions will read and return up to 1024
   bytes each time the function is called. The *master_read* callback is passed
   the pseudoterminalâ€™s master file descriptor to read output from the child
   process, and *stdin_read* is passed file descriptor 0, to read from the
   parent process's standard input.

   Returning an empty byte string from either callback is interpreted as an
   end-of-file (EOF) condition, and that callback will not be called after
   that. If *stdin_read* signals EOF the controlling terminal can no longer
   communicate with the parent process OR the child process. Unless the child
   process will quit without any input, *spawn* will then loop forever. If
   *master_read* signals EOF the same behavior results (on linux at least).

   Return the exit status value from :func:`os.waitpid` on the child process.

   :func:`waitstatus_to_exitcode` can be used to convert the exit status into
   an exit code.

   .. audit-event:: pty.spawn argv pty.spawn

   .. versionchanged:: 3.4
      :func:`spawn` now returns the status value from :func:`os.waitpid`
      on the child process.
```

Example

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]pty.rst, line 86)

Unknown directive type "sectionauthor".

```
.. sectionauthor:: Steen Lumholt
```

The following program acts like the Unix command :manpage:`script(1)`, using a pseudo-terminal to record all input and output of a

terminal session in a "typescript".

> **System Message: ERROR/3** (`D:\onboarding-resources\sample-onboarding-resources\cpython-main\Doc\library\[cpython-main][Doc][library]pty.rst`, line 88); *backlink*
>
> Unknown interpreted text role "manpage".

```
import argparse
import os
import pty
import sys
import time

parser = argparse.ArgumentParser()
parser.add_argument('-a', dest='append', action='store_true')
parser.add_argument('-p', dest='use_python', action='store_true')
parser.add_argument('filename', nargs='?', default='typescript')
options = parser.parse_args()

shell = sys.executable if options.use_python else os.environ.get('SHELL', 'sh')
filename = options.filename
mode = 'ab' if options.append else 'wb'

with open(filename, mode) as script:
    def read(fd):
        data = os.read(fd, 1024)
        script.write(data)
        return data

    print('Script started, file is', filename)
    script.write(('Script started on %s\n' % time.asctime()).encode())

    pty.spawn(shell, read)

    script.write(('Script done on %s\n' % time.asctime()).encode())
    print('Script done, file is', filename)
```