18.4. `signal` — Set handlers for asynchronous events¶

This module provides mechanisms to use signal handlers in Python. Some general rules for working with signals and their handlers:

- A handler for a particular signal, once set, remains installed until it is explicitly reset (Python emulates the BSD style interface regardless of the underlying implementation), with the exception of the handler for `SIGCHLD`, which follows the underlying implementation.
- There is no way to “block” signals temporarily from critical sections (since this is not supported by all Unix flavors).
- Although Python signal handlers are called asynchronously as far as the Python user is concerned, they can only occur between the “atomic” instructions of the Python interpreter. This means that signals arriving during long calculations implemented purely in C (such as regular expression matches on large bodies of text) may be delayed for an arbitrary amount of time.
- When a signal arrives during an I/O operation, it is possible that the I/O operation raises an exception after the signal handler returns. This is dependent on the underlying Unix system’s semantics regarding interrupted system calls.
- Because the C signal handler always returns, it makes little sense to catch synchronous errors like `SIGFPE` or `SIGSEGV`.
- Python installs a small number of signal handlers by default: `SIGPIPE` is ignored (so write errors on pipes and sockets can be reported as ordinary Python exceptions) and `SIGINT` is translated into a `KeyboardInterrupt` exception. All of these can be overridden.
- Some care must be taken if both signals and threads are used in the same program. The fundamental thing to remember in using signals and threads simultaneously is: always perform `signal()` operations in the main thread of execution. Any thread can perform an `alarm()`, `getsignal()`, `pause()`, `setitimer()` or `getitimer()`; only the main thread can set a new signal handler, and the main thread will be the only one to receive signals (this is enforced by the Python `signal` module, even if the underlying thread implementation supports sending signals to individual threads). This means that signals can’t be used as a means of inter-thread communication. Use locks instead.

The variables defined in the `signal` module are:

`signal.SIG_DFL`¶

This is one of two standard signal handling options; it will simply perform the default function for the signal. For example, on most systems the default action for `SIGQUIT` is to dump core and exit, while the default action for `SIGCHLD` is to simply ignore it.

`signal.SIG_IGN`¶

This is another standard signal handler, which will simply ignore the given signal.

`SIG*`

All the signal numbers are defined symbolically. For example, the hangup signal is defined as `signal.SIGHUP`; the variable names are identical to the names used in C programs, as found in <`signal.h`>. The Unix man page for ‘`signal()`‘ lists the existing signals (on some systems this is *signal(2)*, on others the list is in *signal(7)*). Note that not all systems define the same set of signal names; only those names defined by the system are defined by this module.

`signal.NSIG`¶

One more than the number of the highest signal number.

`signal.ITIMER_REAL`¶

Decrements interval timer in real time, and delivers `SIGALRM` upon expiration.

`signal.ITIMER_VIRTUAL`¶

Decrements interval timer only when the process is executing, and delivers SIGVTALRM upon expiration.

`signal.ITIMER_PROF`¶

Decrements interval timer both when the process executes and when the system is executing on behalf of the process. Coupled with ITIMER_VIRTUAL, this timer is usually used to profile the time spent by the application in user and kernel space. SIGPROF is delivered upon expiration.

The `signal` module defines one exception:

exception `signal.ItimerError`¶

Raised to signal an error from the underlying `setitimer()` or `getitimer()` implementation. Expect this error if an invalid interval timer or a negative time is passed to `setitimer()`. This error is a subtype of `IOError`.

The `signal` module defines the following functions:

`signal.alarm`(*time*)¶

If *time* is non-zero, this function requests that a `SIGALRM` signal be sent to the process in *time* seconds. Any previously scheduled alarm is canceled (only one alarm can be scheduled at any time). The returned value is then the number of seconds before any previously set alarm was to have been delivered. If *time* is zero, no alarm is scheduled, and any scheduled alarm is canceled. If the return value is zero, no alarm is currently scheduled. (See the Unix man page *alarm(2)*.) Availability: Unix.

`signal.getsignal`(*signalnum*)¶
Return the current signal handler for the signal *signalnum*. The returned value may be a callable Python object, or one of the special values `signal.SIG_IGN`, `signal.SIG_DFL` or `None`. Here, `signal.SIG_IGN` means that the signal was previously ignored, `signal.SIG_DFL` means that the default way of handling the signal was previously in use, and `None` means that the previous signal handler was not installed from Python.

`signal.pause`()¶
Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing. Not on Windows. (See the Unix man page *signal(2)*.)

`signal.setitimer`(*which*, *seconds*[, *interval*])¶

Sets given interval timer (one of `signal.ITIMER_REAL`, `signal.ITIMER_VIRTUAL` or `signal.ITIMER_PROF`) specified by *which* to fire after *seconds* (float is accepted, different from `alarm()`) and after that every *interval* seconds. The interval timer specified by *which* can be cleared by setting seconds to zero.

When an interval timer fires, a signal is sent to the process. The signal sent is dependent on the timer being used; `signal.ITIMER_REAL` will deliver `SIGALRM`, `signal.ITIMER_VIRTUAL` sends `SIGVTALRM`, and `signal.ITIMER_PROF` will deliver `SIGPROF`.

The old values are returned as a tuple: (delay, interval).

Attempting to pass an invalid interval timer will cause an `ItimerError`. Availability: Unix.

New in version 2.6.

`signal.getitimer`(*which*)¶

Returns current value of a given interval timer specified by *which*. Availability: Unix.

New in version 2.6.

`signal.set_wakeup_fd`(*fd*)¶

Set the wakeup fd to *fd*. When a signal is received, a `'\0'` byte is written to the fd. This can be used by a library to wakeup a poll or select call, allowing the signal to be fully processed.

The old wakeup fd is returned. *fd* must be non-blocking. It is up to the library to remove any bytes before calling poll or select again.

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a `ValueError` exception to be raised.

`signal.siginterrupt`(*signalnum*, *flag*)¶

Change system call restart behaviour: if *flag* is `False`, system calls will be restarted when interrupted by signal *signalnum*, otherwise system calls will be interrupted. Returns nothing. Availability: Unix (see the man page *siginterrupt(3)* for further information).

Note that installing a signal handler with `signal()` will reset the restart behaviour to interruptible by implicitly calling `siginterrupt()` with a true *flag* value for the given signal.

New in version 2.6.

`signal.signal`(*signalnum*, *handler*)¶

Set the handler for signal *signalnum* to the function *handler*. *handler* can be a callable Python object taking two arguments (see below), or one of the special values `signal.SIG_IGN` or `signal.SIG_DFL`. The previous signal handler will be returned (see the description of `getsignal()` above). (See the Unix man page *signal(2)*.)

When threads are enabled, this function can only be called from the main thread; attempting to call it from other threads will cause a `ValueError` exception to be raised.

The *handler* is called with two arguments: the signal number and the current stack frame (`None` or a frame object; for a description of frame objects, see the *description in the type hierarchy* or see the attribute descriptions in the `inspect` module).

18.4.1. Example¶

Here is a minimal example program. It uses the `alarm()` function to limit the time spent waiting to open a file; this is useful if the file is for a serial device that may not be turned on, which would normally cause the `os.open()` to hang indefinitely. The solution is to set a 5-second alarm before opening the file; if the operation takes too long, the alarm signal will be sent, and the handler raises an exception.

```
import signal, os

def handler(signum, frame):
   print 'Signal handler called with signal', signum
   raise IOError("Couldn't open device!")
```

```
# Set the signal handler and a 5-second alarm
signal.signal(signal.SIGALRM, handler)
signal.alarm(5)

# This open() may hang indefinitely
fd = os.open('/dev/ttyS0', os.O_RDWR)

signal.alarm(0)          # Disable the alarm
```


Previous topic

18.3. `ssl` — SSL wrapper for socket objects

Next topic

18.5. `popen2` — Subprocesses with accessible I/O streams

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 18. Interprocess Communication and Networking »