18.8. `signal` — Set handlers for asynchronous events

This module provides mechanisms to use signal handlers in Python.

18.8.1. General rules

The `signal.signal()` function allows defining custom handlers to be executed when a signal is received. A small number of default handlers are installed: `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.

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.

18.8.1.1. Execution of Python signal handlers

A Python signal handler does not get executed inside the low-level (C) signal handler. Instead, the low-level signal handler sets a flag which tells the virtual machine to execute the corresponding Python signal handler at a later point(for example at the next bytecode instruction). This has consequences:

- It makes little sense to catch synchronous errors like `SIGFPE` or `SIGSEGV` that are caused by an invalid operation in C code. Python will return from the signal handler to the C code, which is likely to raise the same signal again, causing Python to apparently hang. From Python 3.3 onwards, you can use the `faulthandler` module to report on synchronous errors.
- A long-running calculation implemented purely in C (such as regular expression matching on a large body of text) may run uninterrupted for an arbitrary amount of time, regardless of any signals received. The Python signal handlers will be called when the calculation finishes.

18.8.1.2. Signals and threads

Python signal handlers are always executed in the main Python thread, even if the signal was received in another thread. This means that signals can’t be used as a means of inter-thread communication. You can use the synchronization primitives from the `threading` module instead.

Besides, only the main thread is allowed to set a new signal handler.

18.8.2. Module contents

Changed in version 3.5: signal (SIG*), handler (`SIG_DFL`, `SIG_IGN`) and sigmask (`SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`) related constants listed below were turned into `enums`. `getsignal()`, `pthread_sigmask()`, `sigpending()` and `sigwait()` functions return human-readable `enums`.

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.`**`CTRL_C_EVENT`**

The signal corresponding to the `Ctrl+C` keystroke event. This signal can only be used with `os.kill()`.

Availability: Windows.

New in version 3.2.

`signal.`**`CTRL_BREAK_EVENT`**

The signal corresponding to the `Ctrl+Break` keystroke event. This signal can only be used with `os.kill()`.

Availability: Windows.

New in version 3.2.

`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.

`signal.`**`SIG_BLOCK`**

A possible value for the *how* parameter to `pthread_sigmask()` indicating that signals are to be blocked.

New in version 3.3.

`signal.`**`SIG_UNBLOCK`**

A possible value for the *how* parameter to `pthread_sigmask()` indicating that signals are to be unblocked.

New in version 3.3.

`signal.`**`SIG_SETMASK`**

A possible value for the *how* parameter to `pthread_sigmask()` indicating that the signal mask is to be replaced.

New in version 3.3.

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 `OSError`.

New in version 3.3: This error used to be a subtype of `IOError`, which is now an alias of `OSError`.

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)*.)

See also `sigwait()`, `sigwaitinfo()`, `sigtimedwait()` and `sigpending()`.

`signal.`**`pthread_kill`**(*thread_id*, *signalnum*)

Send the signal *signalnum* to the thread *thread_id*, another thread in the same process as the caller. The target thread can be executing any code (Python or not). However, if the target thread is executing the Python interpreter, the Python signal handlers will be executed by the main thread. Therefore, the only point of sending a signal to a particular Python thread would be to force a running system call to fail with `InterruptedError`.

Use `threading.get_ident()` or the `ident` attribute of `threading.Thread` objects to get a suitable value for *thread_id*.

If *signalnum* is 0, then no signal is sent, but error checking is still performed; this can be used to check if the target thread is still running.

Availability: Unix (see the man page *pthread_kill(3)* for further information).

See also `os.kill()`.

New in version 3.3.

`signal.`**`pthread_sigmask`**(*how*, *mask*)

Fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller. Return the old signal mask as a set of signals.

The behavior of the call is dependent on the value of *how*, as follows.

- `SIG_BLOCK`: The set of blocked signals is the union of the current set and the *mask* argument.
- `SIG_UNBLOCK`: The signals in *mask* are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
- `SIG_SETMASK`: The set of blocked signals is set to the *mask* argument.

mask is a set of signal numbers (e.g. {`signal.SIGINT`, `signal.SIGTERM`}). Use `range(1, signal.NSIG)` for a full mask including all signals.

For example, `signal.pthread_sigmask(signal.SIG_BLOCK, [])` reads the signal mask of the calling thread.

Availability: Unix. See the man page *sigprocmask(3)* and *pthread_sigmask(3)* for further information.

See also `pause()`, `sigpending()` and `sigwait()`.

New in version 3.3.

`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.

`signal.`**`getitimer`**(*which*)

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

signal.**set_wakeup_fd**(*fd*)

Set the wakeup file descriptor to *fd*. When a signal is received, the signal number is written as a single byte into 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 (or -1 if file descriptor wakeup was not enabled). If *fd* is -1, file descriptor wakeup is disabled. If not -1, *fd* must be non-blocking. It is up to the library to remove any bytes from *fd* before calling poll or select again.

Use for example `struct.unpack('%uB' % len(data), data)` to decode the signal numbers list.

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.

Changed in version 3.5: On Windows, the function now also supports socket handles.

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.

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).

On Windows, `signal()` can only be called with `SIGABRT`, `SIGFPE`, `SIGILL`, `SIGINT`, `SIGSEGV`, `SIGTERM`, or `SIGBREAK`. A `ValueError` will be raised in any other case. Note that not all systems define the same set of signal names; an `AttributeError` will be raised if a signal name is not defined as `SIG*` module level constant.

`signal.`**`sigpending`**()

Examine the set of signals that are pending for delivery to the calling thread (i.e., the signals which have been raised while blocked). Return the set of the pending signals.

Availability: Unix (see the man page *sigpending(2)* for further information).

See also `pause()`, `pthread_sigmask()` and `sigwait()`.

New in version 3.3.

`signal.`**`sigwait`**(*sigset*)

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal (removes it from the pending list of signals), and returns the signal number.

Availability: Unix (see the man page *sigwait(3)* for further information).

See also `pause()`, `pthread_sigmask()`, `sigpending()`, `sigwaitinfo()` and `sigtimedwait()`.

New in version 3.3.

`signal.`**`sigwaitinfo`**(*sigset*)

Suspend execution of the calling thread until the delivery of one of the signals specified in the signal set *sigset*. The function accepts the signal and removes it from the pending list of signals. If one of the signals in *sigset* is already pending for the calling thread, the function will return immediately with information about that signal. The signal handler is not called for the delivered signal. The function raises an `InterruptedError` if it is interrupted by a signal that is not in *sigset*.

The return value is an object representing the data contained in the `siginfo_t` structure, namely: `si_signo`, `si_code`, `si_errno`, `si_pid`, `si_uid`, `si_status`, `si_band`.

Availability: Unix (see the man page *sigwaitinfo(2)* for further information).

See also `pause()`, `sigwait()` and `sigtimedwait()`.

New in version 3.3.

Changed in version 3.5: The function is now retried if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

`signal.`**`sigtimedwait`**(*sigset*, *timeout*)

Like `sigwaitinfo()`, but takes an additional *timeout* argument specifying a timeout. If *timeout* is specified as `0`, a poll is performed. Returns `None` if a timeout occurs.

Availability: Unix (see the man page *sigtimedwait(2)* for further information).

See also `pause()`, `sigwait()` and `sigwaitinfo()`.

New in version 3.3.

Changed in version 3.5: The function is now retried with the recomputed *timeout* if interrupted by a signal not in *sigset* and the signal handler does not raise an exception (see **PEP 475** for the rationale).

18.8.3. 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 OSError("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
```