`!signal` --- Set handlers for asynchronous events

- General rules
  - Execution of Python signal handlers
  - Signals and threads
- Module contents
- Examples
- Note on SIGPIPE
- Note on Signal Handlers and Exceptions

signal

Source code: `Lib/signal.py`

This module provides mechanisms to use signal handlers in Python.

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 if the parent process has not changed it.

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.

On WebAssembly platforms, signals are emulated and therefore behave differently. Several functions and signals are not available on these platforms.

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.
- If the handler raises an exception, it will be raised "out of thin air" in the main thread. See the `note below <handlers-and-exceptions>` for a discussion.

Signals and threads

Python signal handlers are always executed in the main Python thread of the main interpreter, 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 of the main interpreter is allowed to set a new signal handler.

> Synchronization primitives such as `threading.Lock` should not be used within signal handlers. Doing so can lead to unexpected deadlocks.

Module contents

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 <enum.IntEnum>` (`Signals`, `Handlers` and `Sigmasks` respectively). `getsignal`, `pthread_sigmask`, `sigpending` and `sigwait` functions return human-readable `enums <enum.IntEnum>` as `Signals` objects.

The signal module defines three enums:

`enum.IntEnum` collection of SIG* constants and the CTRL* constants.

3.5

`enum.IntEnum` collection of the constants `SIG_DFL` and `SIG_IGN`.

3.5

`enum.IntEnum` collection of the constants `SIG_BLOCK`, `SIG_UNBLOCK` and `SIG_SETMASK`.

Unix.

See the man page `sigprocmask(2)` and `pthread_sigmask(3)` for further information.

3.5

The variables defined in the `!signal` module are:

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.

SIG_IGN

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

SIGABRT

Abort signal from `abort(3)`.

SIGALRM

Timer signal from `alarm(2)`.

Unix.

SIGBREAK

Interrupt from keyboard (CTRL + BREAK).

Windows.

SIGBUS

Bus error (bad memory access).

Unix.

SIGCHLD

Child process stopped or terminated.

Unix.

SIGCLD

Alias to `SIGCHLD`.

not macOS.

SIGCONT

Continue the process if it is currently stopped

Unix.

SIGFPE

Floating-point exception. For example, division by zero.

`ZeroDivisionError` is raised when the second argument of a division or modulo operation is zero.

SIGHUP

Hangup detected on controlling terminal or death of controlling process.

Unix.

SIGILL

Illegal instruction.

SIGINT

Interrupt from keyboard (CTRL + C).

Default action is to raise `KeyboardInterrupt`.

SIGKILL

Kill signal.

It cannot be caught, blocked, or ignored.

Unix.

SIGPIPE

Broken pipe: write to pipe with no readers.

Default action is to ignore the signal.

Unix.

SIGPROF

Profiling timer expired.

Unix.

SIGQUIT

Terminal quit signal.

Unix.

SIGSEGV

Segmentation fault: invalid memory reference.

SIGSTOP

Stop executing (cannot be caught or ignored).

Unix.

SIGSTKFLT

Stack fault on coprocessor. The Linux kernel does not raise this signal: it can only be raised in user space.

Linux.

On architectures where the signal is available. See the man page `signal(7)` for further information.

3.11

SIGTERM

Termination signal.

SIGUSR1

User-defined signal 1.

Unix.

SIGUSR2

User-defined signal 2.

Unix.

SIGVTALRM

Virtual timer expired.

Unix.

SIGWINCH

Window resize signal.

Unix.

SIGXCPU

CPU time limit exceeded.

Unix.

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.

CTRL_C_EVENT

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

Windows.

3.2

CTRL_BREAK_EVENT

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

Windows.

3.2

NSIG

One more than the number of the highest signal number. Use `valid_signals` to get valid signal numbers.

ITIMER_REAL

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

ITIMER_VIRTUAL

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

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.

SIG_BLOCK

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

3.3

SIG_UNBLOCK

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

3.3

SIG_SETMASK

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

3.3

The `!signal` module defines one exception:

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`.

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:

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.

Unix.

See the man page `alarm(2)` for further information.

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.

strsignal(signalnum)

Returns the description of signal *signalnum*, such as "Interrupt" for `SIGINT`. Returns `None` if *signalnum* has no description. Raises `ValueError` if *signalnum* is invalid.

3.8

valid_signals()

Return the set of valid signal numbers on this platform. This can be less than `range(1, NSIG)` if some signals are reserved by the system for internal use.

3.8

pause()

Cause the process to sleep until a signal is received; the appropriate handler will then be called. Returns nothing.

Unix.

See the man page `signal(2)` for further information.

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

raise_signal(signum)

Sends a signal to the calling process. Returns nothing.

3.8

pidfd_send_signal(pidfd, sig, siginfo=None, flags=0)

Send signal *sig* to the process referred to by file descriptor *pidfd*. Python does not currently support the *siginfo* parameter; it must be `None`. The *flags* argument is provided for future extensions; no flag values are currently defined.

See the `pidfd_send_signal(2)` man page for more information.

Linux >= 5.1, Android >= `build-time <sys.getandroidapilevel>` API level 31

3.9

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`

`of the main interpreter <signals-and-threads>`. 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 `~threading.Thread.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.

signal.pthread_kill thread_id,signalnum signal.pthread_kill

Unix.

See the man page `pthread_kill(3)` for further information.

See also `os.kill`.

3.3

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 `~signal.valid_signals` for a full mask including all signals.

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

`SIGKILL` and `SIGSTOP` cannot be blocked.

Unix.

See the man page `sigprocmask(2)` and `pthread_sigmask(3)` for further information.

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

3.3

setitimer(which, seconds, interval=0)

Sets given interval timer (one of `signal.ITIMER_REAL`, `signal.ITIMER_VIRTUAL` or `signal.ITIMER_PROF`) specified by *which* to fire after *seconds* (rounded up to microseconds, different from `alarm`) and after that every *interval* seconds (if *interval* is non-zero). 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 two-tuple of floats: (`delay`, `interval`).

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

Unix.

3.15 Accepts any real numbers as *seconds* and *interval*, not only integers or floats.

getitimer(which)

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

Unix.

set_wakeup_fd(fd, *, warn_on_full_buffer=True)

Set the wakeup file descriptor to *fd*. When a signal your program has registered a signal handler for is received, the signal number is written as a single byte into the fd. If you haven't registered a signal handler for the signals you care about, then nothing will be written to the wakeup 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.

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

There are two common ways to use this function. In both approaches, you use the fd to wake up when a signal arrives, but then they differ in how they determine *which* signal or signals have arrived.

In the first approach, we read the data out of the fd's buffer, and the byte values give you the signal numbers. This is simple, but in rare cases it can run into a problem: generally the fd will

have a limited amount of buffer space, and if too many signals arrive too quickly, then the buffer may become full, and some signals may be lost. If you use this approach, then you should set `warn_on_full_buffer=True`, which will at least cause a warning to be printed to stderr when signals are lost.

In the second approach, we use the wakeup fd *only* for wakeups, and ignore the actual byte values. In this case, all we care about is whether the fd's buffer is empty or non-empty; a full buffer doesn't indicate a problem at all. If you use this approach, then you should set `warn_on_full_buffer=False`, so that your users are not confused by spurious warning messages.

3.5 On Windows, the function now also supports socket handles.

3.7 Added `warn_on_full_buffer` parameter.

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.

Unix.

See the man page `siginterrupt(3)` for further information.

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

When threads are enabled, this function can only be called from `the main thread of the main interpreter <signals-and-threads>`; 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 <frame-objects>` or see the attribute descriptions in the `inspect` module).

On Windows, `~signal.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.

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.

Unix.

See the man page `sigpending(2)` for further information.

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

3.3

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.

Unix.

See the man page `sigwait(3)` for further information.

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

3.3

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`.

Unix.

See the man page `sigwaitinfo(2)` for further information.

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

3.3

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 `475` for the rationale).

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.

Unix.

See the man page `sigtimedwait(2)` for further information.

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

3.3

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 `475` for the rationale).

3.15 Accepts any real number as *timeout*, not only integer or float.

Examples

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):
    signame = signal.Signals(signum).name
    print(f'Signal handler called with signal {signame} ({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
```

Note on SIGPIPE

Piping output of your program to tools like `head(1)` will cause a `SIGPIPE` signal to be sent to your process when the receiver of its standard output closes early. This results in an exception like `BrokenPipeError: [Errno 32] Broken pipe`. To handle this case, wrap your entry point to catch this exception as follows:

```
import os
import sys

def main():
    try:
        # simulate large output (your code replaces this loop)
        for x in range(10000):
            print("y")
        # flush output here to force SIGPIPE to be triggered
        # while inside this try block.
        sys.stdout.flush()
    except BrokenPipeError:
        # Python flushes standard streams on exit; redirect remaining output
        # to devnull to avoid another BrokenPipeError at shutdown
        devnull = os.open(os.devnull, os.O_WRONLY)
        os.dup2(devnull, sys.stdout.fileno())
        sys.exit(1)  # Python exits with error code 1 on EPIPE

if __name__ == '__main__':
    main()
```

Do not set `SIGPIPE`'s disposition to `SIG_DFL` in order to avoid `BrokenPipeError`. Doing that would cause your program to exit unexpectedly whenever any socket connection is interrupted while your program is still writing to it.

Note on Signal Handlers and Exceptions

If a signal handler raises an exception, the exception will be propagated to the main thread and may be raised after any `bytecode` instruction. Most notably, a `KeyboardInterrupt` may appear at any point during execution. Most Python code, including the standard library, cannot be made robust against this, and so a `KeyboardInterrupt` (or any other exception resulting from a signal handler) may on rare occasions put the program in an unexpected state.

To illustrate this issue, consider the following code:

```
class SpamContext:
    def __init__(self):
        self.lock = threading.Lock()

    def __enter__(self):
        # If KeyboardInterrupt occurs here, everything is fine
        self.lock.acquire()
        # If KeyboardInterrupt occurs here, __exit__ will not be called
        ...
        # KeyboardInterrupt could occur just before the function returns

    def __exit__(self, exc_type, exc_val, exc_tb):
        ...
        self.lock.release()
```

For many programs, especially those that merely want to exit on `KeyboardInterrupt`, this is not a problem, but applications that are complex or require high reliability should avoid raising exceptions from signal handlers. They should also avoid catching `KeyboardInterrupt` as a means of gracefully shutting down. Instead, they should install their own `SIGINT` handler. Below is an example of an HTTP server that avoids `KeyboardInterrupt`:

```
import signal
import socket
from selectors import DefaultSelector, EVENT_READ
from http.server import HTTPServer, SimpleHTTPRequestHandler

interrupt_read, interrupt_write = socket.socketpair()

def handler(signum, frame):
```

```
    print('Signal handler called with signal', signum)
    interrupt_write.send(b'\0')
signal.signal(signal.SIGINT, handler)

def serve_forever(httpd):
    sel = DefaultSelector()
    sel.register(interrupt_read, EVENT_READ)
    sel.register(httpd, EVENT_READ)

    while True:
        for key, _ in sel.select():
            if key.fileobj == interrupt_read:
                interrupt_read.recv(1)
                return
            if key.fileobj == httpd:
                httpd.handle_request()

print("Serving on port 8000")
httpd = HTTPServer(('', 8000), SimpleHTTPRequestHandler)
serve_forever(httpd)
print("Shutdown...")
```

□□□ #2
□ □□□ □□□ 2026-10-31 17:18:55 UTC
□ □□□ □□□ 2026-10-31 17:18:58 UTC