signal — Set handlers for asynchronous events

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.

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.

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.

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.

signal.SIGABRT

Abort signal from abort(3).

signal.SIGALRM

Timer signal from alarm(2).

signal.SIGBREAK

Interrupt from keyboard (CTRL + BREAK).

signal.SIGBUS

Bus error (bad memory access).

signal.SIGCHLD

Child process stopped or terminated.

signal.SIGCLD

Alias to SIGCHLD.

signal.SIGCONT

Continue the process if it is currently stopped

signal.SIGFPE

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

signal.SIGHUP

Hangup detected on controlling terminal or death of controlling process.

signal.SIGILL

Illegal instruction.

signal.SIGINT

Interrupt from keyboard (CTRL + C).

Default action is to raise KeyboardInterrupt.

signal.SIGKILL

Kill signal.

It cannot be caught, blocked, or ignored.

signal.SIGPIPE

Broken pipe: write to pipe with no readers.

Default action is to ignore the signal.

signal.SIGSEGV

Segmentation fault: invalid memory reference.

signal.SIGTERM

Termination signal.

signal.SIGUSR1

User-defined signal 1.

signal.SIGUSR2

User-defined signal 2.

signal.SIGWINCH

Window resize 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().

signal.CTRL_BREAK_EVENT

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

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.

signal.SIG_UNBLOCK

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

signal.SIG_SETMASK

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

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.

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.

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.strsignal(signalnum)

Return the system description of the signal signalnum, such as “Interrupt”, “Segmentation fault”, etc. Returns None if the signal is not recognized.

signal.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.

signal.pause()

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

signal.raise_signal(signum)

Sends a signal to the calling process. Returns nothing.

signal.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.

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 of the main interpreter. 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.

Raises an auditing event signal.pthread_kill with arguments thread_id, signalnum.

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 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.

signal.setitimer(which, seconds, interval=0.0)

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 (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 tuple: (delay, interval).

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

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
```

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()
```