signal — Asynchronous System Events

Purpose: Asynchronous system events

Signals are an operating system feature that provide a means of notifying a program of an event, and having it handled asynchronously. They can be generated by the system itself, or sent from one process to another. Since signals interrupt the regular flow of the program, it is possible that some operations (especially I/O) may produce errors if a signal is received in the middle.

Signals are identified by integers and are defined in the operating system C headers. Python exposes the signals appropriate for the platform as symbols in the `signal` module. The examples in this section use `SIGINT` and `SIGUSR1`. Both are typically defined for all Unix and Unix-like systems.

> **Note**
>
> Programming with Unix signal handlers is a non-trivial endeavor. This is an introduction, and does not include all of the details needed to use signals successfully on every platform. There is some degree of standardization across versions of Unix, but there is also some variation, so consult the operating system documentation if you run into trouble.

Receiving Signals

As with other forms of event-based programming, signals are received by establishing a callback function, called a *signal handler*, that is invoked when the signal occurs. The arguments to the signal handler are the signal number and the stack frame from the point in the program that was interrupted by the signal.

```
# signal_signal.py

import signal
import os
import time


def receive_signal(signum, stack):
    print('Received:', signum)


# Register signal handlers
signal.signal(signal.SIGUSR1, receive_signal)
signal.signal(signal.SIGUSR2, receive_signal)

# Print the process ID so it can be used with 'kill'
# to send this program signals.
print('My PID is:', os.getpid())

while True:
    print('Waiting...')
    time.sleep(3)
```

This example script loops indefinitely, pausing for a few seconds each time. When a signal comes in, the `sleep()` call is interrupted and the signal handler `receive_signal` prints the signal number. After the signal handler returns, the loop continues.

Send signals to the running program using `os.kill()` or the Unix command line program `kill`.

```
$ python3 signal_signal.py

My PID is: 71387
Waiting...
Waiting...
Waiting...
Received: 30
Waiting...
Waiting...
Received: 31
Waiting...
Waiting...
Traceback (most recent call last):
  File "signal_signal.py", line 28, in <module>
    time.sleep(3)
KeyboardInterrupt
```

The previous output was produced by running `signal_signal.py` in one window, then in another window running:

```
$ kill -USR1 $pid
$ kill -USR2 $pid
$ kill -INT $pid
```

Retrieving Registered Handlers

To see what signal handlers are registered for a signal, use `getsignal()`. Pass the signal number as argument. The return value is the registered handler, or one of the special values `SIG_IGN` (if the signal is being ignored), `SIG_DFL` (if the default behavior is being used), or `None` (if the existing signal handler was registered from C, rather than Python).

```python
# signal_getsignal.py

import signal


def alarm_received(n, stack):
    return


signal.signal(signal.SIGALRM, alarm_received)

signals_to_names = {
    getattr(signal, n): n
    for n in dir(signal)
    if n.startswith('SIG') and '_' not in n
}

for s, name in sorted(signals_to_names.items()):
    handler = signal.getsignal(s)
    if handler is signal.SIG_DFL:
        handler = 'SIG_DFL'
    elif handler is signal.SIG_IGN:
        handler = 'SIG_IGN'
    print('{:<10} ({:2d}):'.format(name, s), handler)
```

Again, since each OS may have different signals defined, the output on other systems may vary. This is from OS X:

```
$ python3 signal_getsignal.py

SIGHUP     ( 1): SIG_DFL
SIGINT     ( 2): <built-in function default_int_handler>
SIGQUIT    ( 3): SIG_DFL
SIGILL     ( 4): SIG_DFL
SIGTRAP    ( 5): SIG_DFL
SIGIOT     ( 6): SIG_DFL
SIGEMT     ( 7): SIG_DFL
SIGFPE     ( 8): SIG_DFL
SIGKILL    ( 9): None
SIGBUS     (10): SIG_DFL
SIGSEGV    (11): SIG_DFL
SIGSYS     (12): SIG_DFL
SIGPIPE    (13): SIG_IGN
SIGALRM    (14): <function alarm_received at 0x1019a6a60>
SIGTERM    (15): SIG_DFL
SIGURG     (16): SIG_DFL
SIGSTOP    (17): None
SIGTSTP    (18): SIG_DFL
SIGCONT    (19): SIG_DFL
```

```
SIGCHLD    (20): SIG_DFL
SIGTTIN    (21): SIG_DFL
SIGTTOU    (22): SIG_DFL
SIGIO      (23): SIG_DFL
SIGXCPU    (24): SIG_DFL
SIGXFSZ    (25): SIG_IGN
SIGVTALRM  (26): SIG_DFL
SIGPROF    (27): SIG_DFL
SIGWINCH   (28): SIG_DFL
SIGINFO    (29): SIG_DFL
SIGUSR1    (30): SIG_DFL
SIGUSR2    (31): SIG_DFL
```

Sending Signals

The function for sending signals from within Python is `os.kill()`. Its use is covered in the section on the os module, Creating Processes with os.fork().

Alarms

Alarms are a special sort of signal, where the program asks the OS to notify it after some period of time has elapsed. As the standard module documentation for os points out, this is useful for avoiding blocking indefinitely on an I/O operation or other system call.

```
# signal_alarm.py

import signal
import time


def receive_alarm(signum, stack):
    print('Alarm :', time.ctime())


# Call receive_alarm in 2 seconds
signal.signal(signal.SIGALRM, receive_alarm)
signal.alarm(2)

print('Before:', time.ctime())
time.sleep(4)
print('After :', time.ctime())
```

In this example, the call to `sleep()` is interrupted, but then continues after the signal is processed so the message printed after `sleep()` returns shows that the program was paused for at least as long as the sleep duration.

```
$ python3 signal_alarm.py

Before: Sat Apr 22 14:48:57 2017
Alarm : Sat Apr 22 14:48:59 2017
After : Sat Apr 22 14:49:01 2017
```

Ignoring Signals

To ignore a signal, register `SIG_IGN` as the handler. This script replaces the default handler for `SIGINT` with `SIG_IGN`, and registers a handler for `SIGUSR1`. Then it uses `signal.pause()` to wait for a signal to be received.

```
# signal_ignore.py

import signal
import os
import time


def do_exit(sig, stack):
    raise SystemExit('Exiting')


signal.signal(signal.SIGINT, signal.SIG_IGN)
signal.signal(signal.SIGINT, signal.SIG_IGN)
signal.signal(signal.SIGUSR1, do_exit)

print('My PID:', os.getpid())

signal.pause()
```

Normally `SIGINT` (the signal sent by the shell to a program when the user presses `Ctrl-C`) raises a `KeyboardInterrupt`. This example ignores `SIGINT` and raises `SystemExit` when it sees `SIGUSR1`. Each `^C` in the output represents an attempt to use `Ctrl-C` to kill the script from the terminal. Using `kill -USR1 72598` from another terminal eventually causes the script to exit.

```
$ python3 signal_ignore.py

My PID: 72598
^C^C^C^CExiting
```

Signals and Threads

Signals and threads do not generally mix well because only the main thread of a process will receive signals. The following example sets up a signal handler, waits for the signal in one thread, and sends the signal from another.

```
# signal_threads.py

import signal
import threading
import os
import time


def signal_handler(num, stack):
    print('Received signal {} in {}'.format(
        num, threading.currentThread().name))


signal.signal(signal.SIGUSR1, signal_handler)


def wait_for_signal():
    print('Waiting for signal in',
          threading.currentThread().name)
    signal.pause()
    print('Done waiting')


# Start a thread that will not receive the signal
receiver = threading.Thread(
    target=wait_for_signal,
    name='receiver',
)
receiver.start()
time.sleep(0.1)


def send_signal():
    print('Sending signal in', threading.currentThread().name)
    os.kill(os.getpid(), signal.SIGUSR1)


sender = threading.Thread(target=send_signal, name='sender')
sender.start()
sender.join()

# Wait for the thread to see the signal (not going to happen!)
print('Waiting for', receiver.name)
signal.alarm(2)
receiver.join()
```

The signal handlers were all registered in the main thread because this is a requirement of the `signal` module implementation for Python, regardless of underlying platform support for mixing threads and signals. Although the receiver thread calls `signal.pause()`, it does not receive the signal. The `signal.alarm(2)` call near the end of the example prevents an infinite signal.pause(), it does not receive the signal. The signal.alarm(2) call near the end of the example prevents an infinite block, since the receiver thread will never exit.

```
$ python3 signal_threads.py

Waiting for signal in receiver
Sending signal in sender
Received signal 30 in MainThread
Waiting for receiver
Alarm clock
```

Although alarms can be set in any thread, they are always received by the main thread.

```
# signal_threads_alarm.py

import signal
import time
import threading


def signal_handler(num, stack):
    print(time.ctime(), 'Alarm in',
          threading.currentThread().name)


signal.signal(signal.SIGALRM, signal_handler)


def use_alarm():
    t_name = threading.currentThread().name
    print(time.ctime(), 'Setting alarm in', t_name)
    signal.alarm(1)
    print(time.ctime(), 'Sleeping in', t_name)
    time.sleep(3)
    print(time.ctime(), 'Done with sleep in', t_name)


# Start a thread that will not receive the signal
alarm_thread = threading.Thread(
    target=use_alarm,
    name='alarm_thread',
)
alarm_thread.start()
time.sleep(0.1)

# Wait for the thread to see the signal (not going to happen!)
print(time.ctime(), 'Waiting for', alarm_thread.name)
alarm_thread.join()

print(time.ctime(), 'Exiting normally')
```

The alarm does not abort the `sleep()` call in `use_alarm()`.

```
$ python3 signal_threads_alarm.py

Sat Apr 22 14:49:01 2017 Setting alarm in alarm_thread
Sat Apr 22 14:49:01 2017 Sleeping in alarm_thread
Sat Apr 22 14:49:01 2017 Waiting for alarm_thread
Sat Apr 22 14:49:02 2017 Alarm in MainThread
Sat Apr 22 14:49:04 2017 Done with sleep in alarm_thread
Sat Apr 22 14:49:04 2017 Exiting normally
```

See also

- Standard library documentation for signal
- **PEP 475** – Retry system calls failing with EINTR
- subprocess – More examples of sending signals to processes.
- Creating Processes with os.fork() – The `kill()` function can be used to send signals between processes.

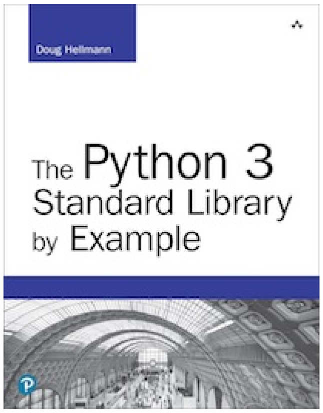
subprocess — Spawning Additional Processes
threading — Manage Concurrent Operations Within a Process

Quick Links

This page was last updated 2017-04-22.

Navigation

subprocess — Spawning Additional Processes
threading — Manage Concurrent Operations Within a Process



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?

This Site

Module Index
Index

Other Writing

Blog
The Python Standard Library By Example