
cysignals Documentation

Release 1.11.4

Martin Albrecht, Jeroen Demeyer

Mar 17, 2024

CONTENTS

1	Interrupt/Signal Handling	3
1.1	Interrupt handling	3
1.2	Handling other signals	8
1.3	Further topics in interrupt/signal handling	9
1.4	Debugging Python crashes	10
2	Error handling	13
2.1	Error handling in C libraries	13
3	Signal-related interfaces for Python code	15
4	Links	17
	Index	19

This is the documentation for **cysignals**, a package to deal with interrupts and signal handling in Cython code.

When writing [Cython](#) code, special care must be taken to ensure that the code can be interrupted with CTRL-C. Since Cython optimizes for speed, Cython normally does not check for interrupts. For example, code like the following cannot be interrupted in Cython:

```
while True:
    pass
```

While this is running, pressing CTRL-C has no effect. The only way out is to kill the Python process. On certain systems, you can still quit Python by typing CTRL-\ (sending a Quit signal) instead of CTRL-C. The package cysignals provides functionality to deal with this, see [Interrupt handling](#).

Besides this, cysignals also provides Python functions/classes to deal with signals. These are not directly related to interrupts in Cython, but provide some supporting functionality beyond what Python provides, see [Signal-related interfaces for Python code](#).

INTERRUPT/SIGNAL HANDLING

Dealing with interrupts and other signals using `sig_check` and `sig_on`:

1.1 Interrupt handling

`cysignals` provides two related mechanisms to deal with interrupts:

- Use `sig_check()` if you are writing mixed Cython/Python code. Typically this is code with (nested) loops where every individual statement takes little time.
- Use `sig_on()` and `sig_off()` if you are calling external C libraries or inside pure Cython code (without any Python functions) where even an individual statement, like a library call, can take a long time.

The functions `sig_check()`, `sig_on()` and `sig_off()` can be put in all kinds of Cython functions: `def`, `cdef` or `cpdef`. You cannot put them in pure Python code (files with extension `.py`).

1.1.1 Basic example

The `sig_check()` in the loop below ensures that the loop can be interrupted by CTRL-C:

```
from cysignals.signals cimport sig_check
from libc.math cimport sin

def sine_sum(double x, long count):
    cdef double s = 0
    for i in range(count):
        sig_check()
        s += sin(i*x)
    return s
```

See the `example` directory for this complete working example.

Note: Cython `cdef` or `cpdef` functions with a return type (like `cdef int myfunc():`) need to have an `except value` to propagate exceptions. Remember this whenever you write `sig_check()` or `sig_on()` inside such a function, otherwise you will see a message like `Exception KeyboardInterrupt: KeyboardInterrupt() in <function name> ignored`.

1.1.2 Using sig_check()

`sig_check()` can be used to check for pending interrupts. If an interrupt happens during the execution of C or Cython code, it will be caught by the next `sig_check()`, the next `sig_on()` or possibly the next Python statement. With the latter we mean that certain Python statements also check for interrupts, an example of this is the `print` statement. The following loop *can* be interrupted:

```
>>> while True:
...     print("Hello")
```

The typical use case for `sig_check()` is within tight loops doing complicated stuff (mixed Python and Cython code, potentially raising exceptions). It is reasonably safe to use and gives a lot of control, because in your Cython code, a `KeyboardInterrupt` can *only* be raised during `sig_check()`:

```
from cysignals.signals cimport sig_check
def sig_check_example():
    for x in foo:
        # (one loop iteration which does not take a long time)
        sig_check()
```

This `KeyboardInterrupt` is treated like any other Python exception and can be handled as usual:

```
from cysignals.signals cimport sig_check
def catch_interrupts():
    try:
        while some_condition():
            sig_check()
            do_something()
    except KeyboardInterrupt:
        # (handle interrupt)
```

Of course, you can also put the `try/except` inside the loop in the example above.

The function `sig_check()` is an extremely fast inline function which should have no measurable effect on performance.

1.1.3 Using sig_on() and sig_off()

Another mechanism for interrupt handling is the pair of functions `sig_on()` and `sig_off()`. It is more powerful than `sig_check()` but also a lot more dangerous. You should put `sig_on()` *before* and `sig_off()` *after* any Cython code which could potentially take a long time. These two *must always* be called in **pairs**, i.e. every `sig_on()` must be matched by a closing `sig_off()`.

In practice your function will probably look like:

```
from cysignals.signals cimport sig_on, sig_off
def sig_example():
    # (some harmless initialization)
    sig_on()
    # (a long computation here, potentially calling a C library)
    sig_off()
    # (some harmless post-processing)
    return something
```


It is possible to put `sig_on()` and `sig_off()` in different functions, provided that `sig_off()` is called before the function which calls `sig_on()` returns. The reason is that `sig_on()` is implemented using `setjmp()`, which requires that the stack frame is kept alive. Therefore, the following code is *invalid*:

```
# INVALID code because we return from function foo()
# without calling sig_off() first.
cdef foo():
    sig_on()

def f1():
    foo()
    sig_off()
```

But the following is valid since you cannot call `foo` interactively:

```
from cysignals.signals cimport sig_on, sig_off

cdef int foo():
    sig_off()
    return 2+2

def f1():
    sig_on()
    return foo()
```

For clarity however, it is best to avoid this.

A common mistake is to put `sig_off()` towards the end of a function (before the `return`) when the function has multiple `return` statements. So make sure there is a `sig_off()` before *every* `return` (and also before every `raise`).

Warning: The code inside `sig_on()` should be pure C or Cython code. If you call any Python code or manipulate any Python object (even something trivial like `x = []`), an interrupt can mess up Python's internal state. When in doubt, try to use `sig_check()` instead.

Also, when an interrupt occurs inside `sig_on()`, code execution immediately stops without cleaning up. For example, any memory allocated inside `sig_on()` is lost. See [Signal handling without exceptions](#) for ways to deal with this.

When the user presses CTRL-C inside `sig_on()`, execution will jump back to `sig_on()` (the first one if there is a stack) and `sig_on()` will raise `KeyboardInterrupt`. As with `sig_check()`, this exception can be handled in the usual way:

```
from cysignals.signals cimport sig_on, sig_off
def catch_interrupts():
    try:
        sig_on() # This must be INSIDE the try
                 # (some long computation)
        sig_off()
    except KeyboardInterrupt:
        # (handle interrupt)
```

It is possible to stack `sig_on()` and `sig_off()`. If you do this, the effect is exactly the same as if only the outer `sig_on()/sig_off()` was there. The inner ones will just change a reference counter and otherwise do nothing. Make sure that the number of `sig_on()` calls equal the number of `sig_off()` calls:

```
from cysignals.signals cimport sig_on, sig_off

def f1():
    sig_on()
    x = f2()
    sig_off()

cdef f2():
    sig_on()
    # ...
    sig_off()
    return ans
```

Extra care must be taken with exceptions raised inside `sig_on()`. The problem is that, if you do not do anything special, the `sig_off()` will never be called if there is an exception. If you need to *raise* an exception yourself, call a `sig_off()` before it:

```
from cysignals.signals cimport sig_on, sig_off
def raising_an_exception():
    sig_on()
    # (some long computation)
    if (something_failed):
        sig_off()
        raise RuntimeError("something failed")
    # (some more computation)
    sig_off()
    return something
```

Alternatively, you can use `try/finally` which will also catch exceptions raised by subroutines inside the `try`:

```
from cysignals.signals cimport sig_on, sig_off
def try_finally_example():
    sig_on() # This must be OUTSIDE the try
    try:
        # (some long computation, potentially raising exceptions)
        return something
    finally:
        sig_off()
```

If you also want to catch this exception, you need a nested `try`:

```
from cysignals.signals cimport sig_on, sig_off
def try_finally_and_catch_example():
    try:
        sig_on()
        try:
            # (some long computation, potentially raising exceptions)
        finally:
            sig_off()
    except Exception:
        print("Trouble! Trouble!")
```

`sig_on()` is implemented using the C library call `setjmp()` which takes a very small but still measurable amount of time. In very time-critical code, one can conditionally call `sig_on()` and `sig_off()`:

```

from cysignals.signals cimport sig_on, sig_off
def conditional_sig_on_example(long n):
    if n > 100:
        sig_on()
    # (do something depending on n)
    if n > 100:
        sig_off()

```

This should only be needed if both the check (`n > 100` in the example) and the code inside the `sig_on()` block take very little time.

1.1.4 Using custom blocking and signal handlers

The following illustrates how signals can be held back similar to `sig_block` and `sig_unblock`. The number theory library PARI/GP defines a variable, which indicates that the execution should not currently be interrupted. Another variable is used to indicate a pending signal, so that PARI/GP can treat it.

Other external libraries might use a similar scheme. Here we indicate this might work:

```

from cysignals.signals cimport sig_on, sig_off, add_custom_signals

cdef extern from "stdio.h":
    void sleep(int)

cdef int SIGINT_block = 0
cdef int SIGINT_pending = 0

cdef int signal_is_blocked():
    return SIGINT_block

cdef void signal_unblock():
    global SIGINT_block
    SIGINT_block = 0

cdef void set_pending_signal(int sig):
    global SIGINT_pending
    SIGINT_pending = sig

# Use the hook provided by cysignals.
add_custom_signals(&signal_is_blocked, &signal_unblock, &set_pending_signal)

def foo(size_t b, int blocked):
    global SIGINT_block, SIGINT_pending
    sig_on()
    SIGINT_block = blocked
    for i in range(b):
        sleep(1)
        if SIGINT_pending:
            SIGINT_block = 0
            SIGINT_pending = 0
            raise KeyboardInterrupt("interrupt was held back")
    SIGINT_block = 0

```

(continues on next page)

(continued from previous page)

```
sig_off()
return
```

In the above scenario `foo(10, 0)` would just wait for 10 seconds, while allowing interrupts. `foo(10, 1)` blocks the interrupt until the end of the second. The pending signal is then treated with a custom message.

1.2 Handling other signals

Apart from handling interrupts, `sig_on()` provides more general signal handling. For example, it handles `alarm()` time-outs by raising an `AlarmInterrupt` (inherited from `KeyboardInterrupt`) exception.

If the code inside `sig_on()` would generate a segmentation fault or call the C function `abort()` (or more generally, raise any of `SIGSEGV`, `SIGILL`, `SIGABRT`, `SIGFPE`, `SIGBUS`), this is caught by the interrupt framework and an exception is raised (`RuntimeError` for `SIGABRT`, `FloatingPointError` for `SIGFPE` and the custom exception `SignalError`, based on `BaseException`, otherwise):

```
from libc.stdlib cimport abort
from cysignals.signals cimport sig_on, sig_off

def abort_example():
    sig_on()
    abort()
    sig_off()
```

```
>>> abort_example()
Traceback (most recent call last):
...
RuntimeError: Aborted
```

This exception can be handled by a `try/except` block as explained above. A segmentation fault or `abort()` unguarded by `sig_on()` would simply terminate the Python Interpreter. This applies only to `sig_on()`, the function `sig_check()` only deals with interrupts and alarms.

Instead of `sig_on()`, there is also a function `sig_str(s)`, which takes a C string `s` as argument. It behaves the same as `sig_on()`, except that the string `s` will be used as a string for the exception. `sig_str(s)` should still be closed by `sig_off()`. Example Cython code:

```
from libc.stdlib cimport abort
from cysignals.signals cimport sig_str, sig_off

def abort_example_with_sig_str():
    sig_str("custom error message")
    abort()
    sig_off()
```

Executing this gives:

```
>>> abort_example_with_sig_str()
Traceback (most recent call last):
...
RuntimeError: custom error message
```

With regard to ordinary interrupts (i.e. SIGINT), `sig_str(s)` behaves the same as `sig_on()`: a simple `KeyboardInterrupt` is raised.

1.3 Further topics in interrupt/signal handling

1.3.1 Testing interrupts

When writing documentation, one sometimes wants to check that certain code can be interrupted in a clean way. The best way to do this is to use `cysignals.alarm()`.

The following is an example of a doctest demonstrating that the SageMath function `factor()` can be interrupted:

```
>>> from cysignals.alarm import alarm, AlarmInterrupt
>>> try:
...     alarm(0.5)
...     factor(10**1000 + 3)
... except AlarmInterrupt:
...     print("alarm!")
alarm!
```

If you use the SageMath doctesting framework, you can instead doctest the exception in the usual way (the Python `doctest` module exits whenever a `KeyboardInterrupt` is raised in a doctest). To avoid race conditions, make sure that the calls to `alarm()` and the function you want to test are in the same doctest:

```
>>> alarm(0.5); factor(10**1000 + 3)
Traceback (most recent call last):
...
AlarmInterrupt
```

1.3.2 Signal handling without exceptions

There are several more specialized functions for dealing with interrupts. As mentioned above, `sig_on()` makes no attempt to clean anything up (restore state or freeing memory) when an interrupt occurs. In fact, it would be impossible for `sig_on()` to do that. If you want to add some cleanup code, use `sig_on_no_except()` for this. This function behaves *exactly* like `sig_on()`, except that any exception raised (like `KeyboardInterrupt` or `RuntimeError`) is not yet passed to Python. Essentially, the exception is there, but we prevent Cython from looking for the exception. Then `cython_check_exception()` can be used to make Cython look for the exception.

Normally, `sig_on_no_except()` returns 1. If a signal was caught and an exception raised, `sig_on_no_except()` instead returns 0. The following example shows how to use `sig_on_no_except()`:

```
def no_except_example():
    if not sig_on_no_except():
        # (clean up messed up internal state)

        # Make Cython realize that there is an exception.
        # It will look like the exception was actually raised
        # by cython_check_exception().
        cython_check_exception()
    # (some long computation, messing up internal state of objects)
    sig_off()
```

There is also a function `sig_str_no_except(s)` which is analogous to `sig_str(s)`.

Note: See the file `src/cysignals/tests.pyx` for more examples of how to use the various `sig_*`() functions.

1.3.3 Releasing the Global Interpreter Lock (GIL)

All the functions related to interrupt and signal handling do not require the [Python GIL](#) (if you don't know what this means, you can safely ignore this section), they are declared `nogil`. This means that they can be used in Cython code inside `with nogil` blocks. If `sig_on()` needs to raise an exception, the GIL is temporarily acquired internally.

If you use C libraries without the GIL and you want to raise an exception before calling `sig_error()`, remember to acquire the GIL while raising the exception. Within Cython, you can use a `with gil context`.

Warning: The GIL should never be released or acquired inside a `sig_on()` block. If you want to use a `with nogil` block, put both `sig_on()` and `sig_off()` inside that block. When in doubt, choose to use `sig_check()` instead, which is always safe to use.

1.4 Debugging Python crashes

If `cysignals` is imported, it sets up a hook which triggers when Python crashes. For example, it would be triggered on a segmentation fault outside a `sig_on()` block.

When a crash happens, first a simple C backtrace is printed if supported by the C library on the system. Then GDB is run to print a much more complete backtrace (except on OS X, where running a debugger requires special privileges). For your convenience, these GDB backtraces are also saved to a logfile.

Finally, this familiar message is shown:

This probably occurred because a **compiled** module has a bug **in it and is not** properly wrapped **with** `sig_on()`, `sig_off()`.
Python will now terminate.

1.4.1 Environment variables

There are several environment variables which influence this:

CYSIGNALS_CRASH_QUIET

If set, be completely quiet whenever a crash happens. No backtrace or other message is shown and GDB is not run.

CYSIGNALS_CRASH_NDEBUG

If set, disable the GDB backtrace. The simple backtrace is still shown.

CYSIGNALS_CRASH_LOGS

The directory where the logs of the crashes are stored. If this is empty, disable storing of crash logs. The default is `cysignals_crash_logs` in the current directory.

CYSIGNALS_CRASH_DAYS

Automatically delete crash logs older than this many days in the directory where crash logs are stored. A negative value means that logs are never deleted. The default is 7 days if `CYSIGNALS_CRASH_LOGS` is unset and -1 days (never delete) otherwise.

ERROR HANDLING

Defining error callbacks for external libraries using `sig_error`:

2.1 Error handling in C libraries

Some C libraries can produce errors and use some sort of callback mechanism to report errors: an external error handling function needs to be set up which will be called by the C library if an error occurs.

The function `sig_error()` can be used to deal with these errors. This function may only be called within a `sig_on()` block (otherwise the Python interpreter will crash hard) after raising a Python exception. You need to use the [Python/C API](#) for this and call `sig_error()` after calling some variant of `PyErr_SetObject()`. Even within Cython, you cannot use the `raise` statement, because then the `sig_error()` will never be executed. The call to `sig_error()` will use the `sig_on()` machinery such that the exception will be seen by `sig_on()`.

A typical error handler implemented in Cython would look as follows:

```
from csignals.signals cimport sig_error
from cpython.exc cimport PyErr_SetString

cdef void error_handler(char *msg):
    PyErr_SetString(RuntimeError, msg)
    sig_error()
```

Exceptions which are raised this way can be handled as usual by putting the `sig_on()` in a `try/except` block. For example, the package `cypari2` provides a wrapper around the number theory library PARI/GP. The `error handler` has a callback which turns errors from PARI/GP into Python exceptions of type `PariError`. This can be handled as follows:

```
from csignals.signals cimport sig_on, sig_off
def handle_pari_error():
    try:
        sig_on() # This must be INSIDE the try
        # (call to PARI)
        sig_off()
    except PariError:
        # (handle error)
```

SageMath uses this mechanism for libGAP, GLPK, NTL and PARI.

SIGNAL-RELATED INTERFACES FOR PYTHON CODE

`cysignals` provides further support for system calls related to signals:

CHAPTER FOUR

LINKS

- cysignals on the Python package index: <https://pypi.org/project/cysignals/>
- cysignals code repository and issue tracker on GitHub: <https://github.com/sagemath/cysignals>
- cysignals documentation on Read the Docs: <https://cysignals.readthedocs.io>

INDEX

E

environment variable

- CYSIGNALS_CRASH_DAYS, [10](#)
- CYSIGNALS_CRASH_LOGS, [10](#)
- CYSIGNALS_CRASH_NDEBUG, [10](#)
- CYSIGNALS_CRASH_QUIET, [10](#)