

---

# **cysignals Documentation**

***Release 1.3.2***

**Martin Albrecht, Jeroen Demeyer**

**Nov 09, 2016**



## CONTENTS

<b>1</b>	<b>Basic example</b>	<b>3</b>
<b>2</b>	<b>Using <code>sig_check()</code></b>	<b>5</b>
<b>3</b>	<b>Using <code>sig_on()</code> and <code>sig_off()</code></b>	<b>7</b>
<b>4</b>	<b>Other Signals</b>	<b>11</b>
<b>5</b>	<b>Error Handling in C Libraries</b>	<b>13</b>
<b>6</b>	<b>Advanced Functions</b>	<b>15</b>
<b>7</b>	<b>Testing Interrupts</b>	<b>17</b>
<b>8</b>	<b>Releasing the Global Interpreter Lock (GIL)</b>	<b>19</b>



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.

This module 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`).



## BASIC EXAMPLE

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

```
include "cysignals/signals.pxi"

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
```

The line `include "cysignals/signals.pxi"` must be put in every `.pyx` file using `cysignals`. You must not put this in a `.pxd` file; a `.pxi` file included only in `.pyx` files also works.

Because of [cython/cython#483](#), you should add `include_path=sys.path` to your `cythonize()` call in `setup.py` (otherwise Cython will not find `cysignals/signals.pxi`). 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`.

---





## 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()`:

```
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:

```
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.



## 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:

```
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 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:

```
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 [Advanced Functions](#) 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:

```
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:

```
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:

```
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`:

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

```
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()`:

```
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.



## 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):

```
cdef extern from 'stdlib.h':
    void abort()

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:

```
cdef extern from 'stdlib.h':
    void abort()

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.





## 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:

```
include "cysignals/signals.pxi"
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, in SageMath, the PARI interface can raise a custom `PariError` exception. This can be handled as follows:

```
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, NTL and PARI.



## ADVANCED FUNCTIONS

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.

---



## 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. 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
```



## 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.