atexit — Program Shutdown Callbacks

Purpose: Register function(s) to be called when a program is closing down.

The `atexit` module provides an interface to register functions to be called when a program closes down normally.

Registering Exit Callbacks

This is an example of registering a function explicitly by calling `register()`.

```
# atexit_simple.py

import atexit


def all_done():
    print('all_done()')


print('Registering')
atexit.register(all_done)
print('Registered')
```

Because the program does not do anything else, `all_done()` is called right away.

```
$ python3 atexit_simple.py

Registering
Registered
all_done()
```

It is also possible to register more than one function and to pass arguments to the registered functions. That can be useful to cleanly disconnect from databases, remove temporary files, etc. Instead of keeping a list of resources that need to be freed, a separate clean-up function can be registered for each resource.

```
# atexit_multiple.py

import atexit


def my_cleanup(name):
    print('my_cleanup({})'.format(name))


atexit.register(my_cleanup, 'first')
atexit.register(my_cleanup, 'second')
atexit.register(my_cleanup, 'third')
```

The exit functions are called in the reverse of the order in which they are registered. This method allows modules to be cleaned up in the reverse order from which they are imported (and therefore register their `atexit` functions), which should reduce dependency conflicts.

```
$ python3 atexit_multiple.py

my_cleanup(third)
my_cleanup(second)
my_cleanup(first)
```

Decorator Syntax

Functions that require no arguments can be registered by using `register()` as a decorator. This alternative syntax is

convenient for cleanup functions that operate on module-level global data.

```
# atexit_decorator.py

import atexit


@atexit.register
def all_done():
    print('all_done()')


print('starting main program')
```

Because the function is registered as it is defined, it is also important to ensure that it works properly even if no other work is performed by the module. If the resources it is supposed to clean up were never initialized, calling the exit callback should not produce an error.

```
$ python3 atexit_decorator.py

starting main program
all_done()
```

Canceling Callbacks

To cancel an exit callback, remove it from the registry using `unregister()`.

```
# atexit_unregister.py

import atexit


def my_cleanup(name):
    print('my_cleanup({})'.format(name))


atexit.register(my_cleanup, 'first')
atexit.register(my_cleanup, 'second')
atexit.register(my_cleanup, 'third')

atexit.unregister(my_cleanup)
```

All calls to the same callback are canceled, regardless of how many times it has been registered.

```
$ python3 atexit_unregister.py
```

Removing a callback that was not previously registered is not considered an error.

```
# atexit_unregister_not_registered.py

import atexit


def my_cleanup(name):
    print('my_cleanup({})'.format(name))


if False:
    atexit.register(my_cleanup, 'never registered')

atexit.unregister(my_cleanup)
```

Because it silently ignores unknown callbacks, `unregister()` can be used even when the sequence of registrations might not be known.

```
$ python3 atexit_unregister_not_registered.py
```

When Are atexit Callbacks Not Called?

The callbacks registered with `atexit` are not invoked if any of these conditions is met:

- The program dies because of a signal.
- `os._exit()` is invoked directly.
- A fatal error is detected in the interpreter.

An example from the subprocess section can be updated to show what happens when a program is killed by a signal. Two files are involved, the parent and the child programs. The parent starts the child, pauses, then kills it.

```
# atexit_signal_parent.py

import os
import signal
import subprocess
import time

proc = subprocess.Popen('./atexit_signal_child.py')
print('PARENT: Pausing before sending signal...')
time.sleep(1)
print('PARENT: Signaling child')
os.kill(proc.pid, signal.SIGTERM)
```

The child sets up an `atexit` callback, and then sleeps until the signal arrives.

```
# atexit_signal_child.py

import atexit
import time
import sys


def not_called():
    print('CHILD: atexit handler should not have been called')


print('CHILD: Registering atexit handler')
sys.stdout.flush()
atexit.register(not_called)

print('CHILD: Pausing to wait for signal')
sys.stdout.flush()
time.sleep(5)
```

When run, this is the output.

```
$ python3 atexit_signal_parent.py

CHILD: Registering atexit handler
CHILD: Pausing to wait for signal
PARENT: Pausing before sending signal...
PARENT: Signaling child
```

The child does not print the message embedded in `not_called()`.

If a program uses `os._exit()`, it can avoid having the `atexit` callbacks invoked.

```
# atexit_os_exit.py

import atexit
import os


def not_called():
    print('This should not be called')


print('Registering')
```

```
print('Registering')
atexit.register(not_called)
print('Registered')

print('Exiting...')
os._exit(0)
```

Because this example bypasses the normal exit path, the callback is not run. The print output is also not flushed, so the example is run with the `-u` option to enable unbuffered I/O.

```
$ python3 -u atexit_os_exit.py

Registering
Registered
Exiting...
```

To ensure that the callbacks are run, allow the program to terminate by running out of statements to execute or by calling `sys.exit()`.

```
# atexit_sys_exit.py

import atexit
import sys


def all_done():
    print('all_done()')


print('Registering')
atexit.register(all_done)
print('Registered')

print('Exiting...')
sys.exit()
```

This example calls `sys.exit()`, so the registered callbacks are invoked.

```
$ python3 atexit_sys_exit.py

Registering
Registered
Exiting...
all_done()
```

Handling Exceptions

Tracebacks for exceptions raised in `atexit` callbacks are printed to the console and the last exception raised is re-raised to be the final error message of the program.

```
# atexit_exception.py

import atexit


def exit_with_exception(message):
    raise RuntimeError(message)


atexit.register(exit_with_exception, 'Registered first')
atexit.register(exit_with_exception, 'Registered second')
```

The registration order controls the execution order. If an error in one callback introduces an error in another (registered earlier, but called later), the final error message might not be the most useful error message to show the user.

```
$ python3 atexit_exception.py

Error in atexit._run_exitfuncs:
```

```
Traceback (most recent call last):
  File "atexit_exception.py", line 11, in exit_with_exception
    raise RuntimeError(message)
RuntimeError: Registered second
Error in atexit._run_exitfuncs:
Traceback (most recent call last):
  File "atexit_exception.py", line 11, in exit_with_exception
    raise RuntimeError(message)
RuntimeError: Registered first
```

It is usually best to handle and quietly log all exceptions in cleanup functions, since it is messy to have a program dump errors on exit.

See also

- Standard library documentation for atexit
- Exception Handling – Global handling for uncaught exceptions.
- Python 2 to 3 porting notes for atexit

fileinput — Command-Line Filter Framework

sched — Timed Event Scheduler

Quick Links

Registering Exit Callbacks
Decorator Syntax
Canceling Callbacks
When Are atexit Callbacks Not Called?
Handling Exceptions

This page was last updated 2016-12-31.

Navigation

fileinput — Command-Line Filter Framework
sched — Timed Event Scheduler

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?

Other Writing

- Blog
- The Python Standard Library By Example