raise SystemExit(0)

DAVID BEAZLEY

David Beazley is an open source developer and author of the *Python Essential Reference* (4th Edition, Addison-Wesley, 2009). He is also known as the creator of Swig (http://www.swig.org) and Python Lex-Yacc (http://www.dabeaz.com/ply.html). Beazley is based in Chicago, where he also teaches a variety of Python courses. dave@dabeaz.com

It's with some regret that this is the final installment of my regular Python column. I suppose that I should offer some final words of wisdom, a historical retrospective, or maybe even a forward-looking "Python of the future" vision—however, I'm simply not that clever. Truth be told, I'm simply stretched a bit thin these days and think it would be a good time to step aside to make room for a fresh new voice and insights. So, in the spirit of leaving on a useful note, I thought I'd end on a practical trifle of a matter of some importance—the problem of getting a Python program to quit.

Bailing Out

Suppose your program has reached the final limit of what it can tolerate and you want it to die. To make it happen, raise a `SystemExit` exception and be done with it. For example:

```
raise SystemExit(1)
```

It is standard practice to include some kind of numeric exit code, which indicates success (zero) or failure (non-zero) back to the process that launched Python. Alternatively, you can include a diagnostic message.

```
raise SystemExit('Goodbye cruel world')
```

When you give a message, it is printed to `sys.stderr` and Python exits with a status code of 1. Problem solved—your program gracefully cleans up after itself and quits. Of course, that's naturally not the end of the story or else this would be a pretty short article. Let's continue.

Catching Exceptions

The `SystemExit` exception is not grouped with other exceptions. For example, it's somewhat common to encounter code that catches all errors like this:

```
try:
    something_complicated()
except Exception as e:
    print("It didn't work. Reason: %s", e)
```

This code will catch most errors, but not `SystemExit`. If you wanted to catch that, you'd have to add an extra clause for it. For example,

```
try:
    something_complicated()
except Exception as e:
    print("It didn't work. Reason: %s", e)
except SystemExit as e:
    print("I see you're going away")
    raise
```

In practice, it's rarely the case that you would ever catch `SystemExit` yourself. Even if you did, the most sensible action is perhaps to simply log the event and re-raise the exception as shown.

A common confusion is that sometimes programmers catch SystemExit by accident by writing sloppy exception handling code:

```
try:
   something_complicated()
except:
   print("It didn't work.")
```

This actually catches all possible errors, including SystemExit. However, this behavior is often unexpected and a potential source of obscure bugs. It's usually best to catch just the exceptions you need as opposed to casting such a wide net.

Keeping the Interpreter Alive

For the purpose of debugging, sometimes it's nice to keep the interpreter alive so that you can go poking around. Use python -i for that. It works even in programs that intentionally raise a SystemExit exception. You might see a message printed, but otherwise, you'll be dropped into the interactive Python shell afterwards. From there, you can mess around. For example:

```
% python3 -i spam.py
Traceback (most recent call last):
  File "spam.py", line 5, in
    spam()
  File "spam.py", line 3, in spam
    raise SystemExit("I'm dead")
SystemExit: I'm dead
>>>
```

Cleanup Actions

Sometimes you might want extra actions to take place upon program termination. For this, you can use the atexit module [1]. For example:

```
import atexit

def goodbye():
   print("So long and thanks for all of the fish")

atexit.register(goodbye)
```

atexit allows you to register an arbitrary number of zero-argument functions that get fired upon termination of the Python interpreter. The functions execute in reverse order of registration. If you need to carry extra information, it is standard practice to use a lambda or functools.partial to do it. For example:

```
def spam(name):
   atexit.register(lambda: print('Goodbye', name))
   ...
```

If needed, you can also unregister a previously registered function using atexit.unregister().

Cleanup with Context Managers

On the subject of cleanup, when working with objects, it's usually best to make use of context managers and Python's with statement. For example, suppose you had some object that involved closing a resource such as a file or connection. A good way to clean it up is to give it __enter__() and __exit__() methods like this:

```
class Spam(object):
   def __init__(self):
      self.resource = SomeResource()
      ...
   def __enter__(self):
      return self
   def __exit__(self, *args):
       # Cleanup
      self.resource.close()
```

With this object, you can now write code like this:

```
with Spam() as s:
   ...
    # Use s
   ...
 # Resources released here
```

When control-flow leaves the indented block, the __exit__() method will run. This happens regardless of what happens in the block—including SystemExit.

The __del__ Puzzle

Sometimes user-defined classes will define a __del__() method for the purposes of cleanup. For example:

```
 # spam.py
import datetime

class Spam(object):
   def __del__(self):
     print("%s destroyed at %s" % (self, datetime.datetime.now()))
```

__del__() is a particularly troublesome method to be defining in general. The main problem is that you simply don't know when it's actually going to fire. This is especially true on program exit. When Python shuts down, all active objects get garbage collected—this includes functions, classes, and modules. In the above example, it's entirely possible you could get a warning message like this printed to standard error:

```
Exception AttributeError: "'NoneType' object has no attribute
'datetime'" in <bound method Spam.__del__ of <spam.Spam
object at 0x1007d9f90>> ignored
```

What's happened here is that the datetime module reference has already been garbage-collected and is no longer defined in global scope. The __del__() method blows up because datetime

is gone. This sort of thing can often be fixed by playing weird games with default arguments. For example:

```
 # spam.py
import datetime

class Spam(object):
    def __del__(self, now=datetime.datetime.now):
        print("%s destroyed at %s" % (self, now()))
```

Ugh. Explaining something like that to your coworkers is going to be hard and even then, it's no guarantee that it's going to work (what if the `now()` function itself needs other functionality that's already been garbage-collected?). The bottom line is don't rely on `__del__()` to perform cleanup actions properly when it comes to program exit. You're better off considering a context manager or a more explicit approach.

Threads

Program exit becomes much more interesting when you start programming with threads [2]. Consider the following code:

```
import threading
import time

def countdown(n):
    while n > 0:
        print('T-minus', n)
        time.sleep(5)
        n -= 1

threading.Thread(target=countdown, args=(5,)).start()
raise SystemExit("Goodbye")
```

If you run this, program termination is delayed until the thread runs to completion. In fact, if you had a lot of threads, program exit won't occur until all of them terminate.

This situation is made even more unfortunate given that there is no mechanism for terminating or signaling a thread once started. Your only sane recourse is to build in some kind of periodic polling or check.

```
import threading
import time

main_thread = threading.current_thread()

def countdown(n):
    while n > 0 and main_thread.is_alive():
        print('T-minus', n)
        time.sleep(5)
        n -= 1

threading.Thread(target=countdown, args=(5,)).start()
raise SystemExit('Goodbye cruel world')
```

Alternatively, you could create the thread as "daemonic" like this:

```
import threading
import time

def countdown(n):
    while n > 0:
        print('T-minus', n)
        time.sleep(5)
        n -= 1

threading.Thread(target=countdown, args=(5,), daemon=True).
start()
raise SystemExit("Goodbye")
```

Daemonic threads are killed immediately once the main thread exits. This is not without its own set of concerns, however. Daemonic threads killed in this way do not exit gracefully. For example, they don't garbage-collect remaining objects (they don't execute `__del__()` methods), and they don't run the `__exit__()` method of context managers. So if you were expecting some kind of graceful cleanup from this, don't.

Curiously, functions registered with `atexit` will still run upon termination of a threaded program—even if registered by daemonic threads. So you could write code like this:

```
import threading
import time
import atexit

def countdown(n):
    onexit = lambda: print('Thread dead. Final value', n)
    atexit.register(onexit)
    while n > 0:
        print('T-minus', n)
        time.sleep(5)
        n -= 1
    atexit.unregister(onexit)

threading.Thread(target=countdown, args=(5,), daemon=True).
start()
time.sleep(12)
raise SystemExit('Goodbye cruel world')
```

When you run this, you'll see a message about a final value of 3.

Keyboard Interrupts and Signals

One especially nasty problem with program termination is the handling of keyboard interrupts (Control-C) and signals [3]. These events often ultimately result in program termination, but unlike a typical `SystemExit` exception, they occur asynchronously. This means that they could potentially occur on any statement in your program.

Perhaps the most important thing to note about signals is that they are only handled by Python's main execution thread. There

are situations where it is impossible to receive signals and it will appear as if it is impossible to kill your program. The most common scenario is if the main program gets tied up on a lock or becomes busy with some CPU-intensive task. Here is a simple example you can try:

```
>>> 'a' in range(1000000000)    # Use xrange on Python2
<Ctrl-C>
```

In this example, you'll find that the code becomes totally unresponsive to the keyboard interrupt until the operation completes. Under the covers the interpreter is tied up with a big calculation taking place in C. There's just no opportunity for it to be interrupted.

More diabolical situations can arise with combinations of locking and signal handling. Consider this interesting bit of code involving the logging module:

```
import logging
import time
import signal

log = logging.getLogger(__name__)

def goodbye(signo, frame):
   log.debug('Goodbye')
   raise SystemExit()

def spin():
   while True:
      log.debug('Hey %f' % time.time())

signal.signal(signal.SIGINT, goodbye)
logging.basicConfig(level=logging.DEBUG)
spin()
```

In this code, a constant stream of log messages is quickly emitted until terminated by a Control-C (SIGINT). It might look innocent enough and it might even seem to work when you try it. However, there are hidden dangers. It turns out that the logging module internally uses thread locks. If you run this program repeatedly, killing it with Control-C, you might find that just every so often, instead of dying, the whole program freezes. What happened? The main program was in the middle of logging a message with the lock held when a signal arrived. The signal handler then tried to log a message, but is now deadlocked due to the logging lock being in use. Your only recourse here—open up another terminal and kill Python using `kill -9`.

Some general advice concerning threads, signals, and program exit. If you want your program to terminate, a sensible strategy is often one that keeps the main-thread free for nothing other than signal handling. Use it to catch keyboard interrupts and other signals and have it arrange to have the rest of the program exit in the most graceful manner that you can devise. This is only a simple template:

```
import threading
import time

terminated = False

def countdown(n):
   while n > 0 and not terminated:
      print('T-minus', n)
      time.sleep(5)
      n -= 1

threading.Thread(target=countdown, args=(5,)).start()

 # Main-thread. Spin and wait for termination
try:
   while True:
     time.sleep(1)
finally:
   terminated = True
```

There are many variations on this theme, but if you've got a very complicated application and it involves concurrency, directing all asynchronous signals and/or the keyboard interrupt to a single well-defined place is probably a good strategy.

The Nuclear Option

Finally, if all else fails, there is always `os._exit()`. For example:

```
import os; os._exit(1)
```

This is a direct line to the underlying `exit()` system call. It will terminate Python immediately, with no cleanup of any kind. As a general rule, though, you'd probably want to avoid this except as a last resort.

Final Words

As noted, this is my last installment of the regular Python column. I'd just like to thank Rik Farrow and everyone else at USENIX for their support over the last six years and hope that you've enjoyed it. I intend to stay active in the Python community, so say hello if you ever see me at a conference, or if you happen to be in the Chicago area, please feel free to look me up. Until then, happy Python hacking!

References

[1] Atexit module: https://docs.python.org/3/library/atexit.html.

[2] Threading module: https://docs.python.org/3/library/threading.html.

[3] Signal module: https://docs.python.org/3/library/signal.html.