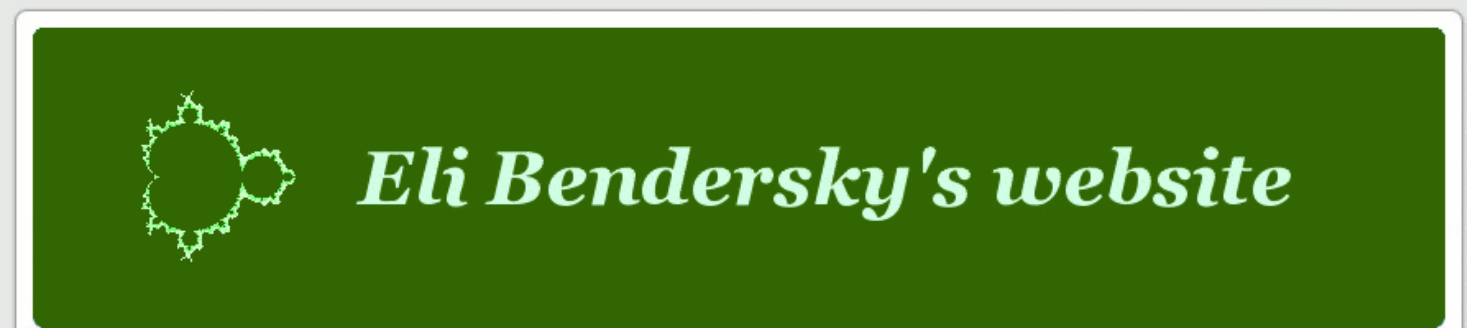

<<< Back to blog Archives

How (not) to set a timeout on a computation in Python

August 22nd, 2011 at 5:50 am

A common question that comes up in mailing lists and Stack Overflow is how to set a timeout on some function call or computation in Python. When people ask this question, they usually imagine the following scenario: some function their code is calling can run for too long, and they want to make sure this doesn't happen, so after some pre-set timeout the computation should terminate and the program is free to do something else. Oh, and this should work on all platforms, of course.

It turns out this seemingly simple task is hard to do in Python. Here I want to discuss some solutions commonly proposed, with their drawbacks.

One of the solutions is to use `signal.SIGALRM`. Apart from the trickiness of using the `signal` module within multi-threaded applications (read this for more details), there's a big problem – `SIGARLM` is only supported on Unix platforms. If you need this code to run on Windows, you're out of luck.

Another common "solution" I've seen is the following [1]. I've simplified the code to make the point clearer, ignoring exceptions and other special conditions:

```
class TimeLimitExpired(Exception): pass

def timelimit(timeout, func, args=(), kwargs={}):
    """ Run func with the given timeout. If func didn't finish running
        within the timeout, raise TimeLimitExpired
    """
    import threading
    class FuncThread(threading.Thread):
        def __init__(self):
            threading.Thread.__init__(self)
            self.result = None

        def run(self):
            self.result = func(*args, **kwargs)

    it = FuncThread()
    it.start()
    it.join(timeout)
    if it.isAlive():
        raise TimeLimitExpired()
    else:
        return it.result
```

The trick here is to run the function inside a thread and use the `timeout` argument of `Thread.join` to implement the time limit. `join` will return after `timeout` whether the thread (i.e. the function) stopped running or not. If it's still running (`isAlive()` returns `True`) then the time limit exception is raised.

I hope the problem here is obvious. Think about it for a moment – suppose the function didn't finish within the given timeout, what happens to the thread after the exception is raised? Nothing – it just keeps on happily running. If the function call never returns for some reason, we've just made ourselves a "zombie" thread that will continue executing, consuming CPU resources.

What we really need to do is to somehow kill the thread if the timeout expires. Whoops, we're in trouble. **Threads can't be killed in Python**, and for a good reason.

This is why I was very surprised to find [2] an "improved" version of the approach presented above. Again, the code is simplified to keep only the relevant parts:

```
class TimeLimitExpired(Exception): pass
```

```
def timelimit(timeout, func, args=(), kwargs={}):
    """ Run func with the given timeout. If func didn't finish running
        within the timeout, raise TimeLimitExpired
    """
    import threading
    class FuncThread(threading.Thread):
        def __init__(self):
            threading.Thread.__init__(self)
            self.result = None

        def run(self):
            self.result = func(*args, **kwargs)

        def _stop(self):
            if self.isAlive():
                Thread._Thread__stop(self)

    it = FuncThread()
    it.start()
    it.join(timeout)
    if it.isAlive():
        it._stop()
        raise TimeLimitExpired()
    else:
        return it.result
```

Whoa – what is that? `Thread._Thread__stop` is a call to the private, *name mangled* method `__stop` of the `Thread` class, with the apparent hope that this method actually causes a thread to stop. But it doesn't! All it does is set the internal `Thread.__stopped` flag that allows `join` to return earlier. *You can't kill threads in Python, remember?* So this approach is just a fallacy based on the misunderstanding of the internals of `Thread` [3].

So even more sophisticated "solutions" propose to ditch the Python-level-API and just brutally kill a thread with `pthread_kill` (on Unix) or `TerminateThread` (on Windows). This is a **very bad idea**. Even low-level APIs like `pthread`, which do provide a means to kill threads, recommend avoiding it. In Python it's even more problematic because of the way the interpreter works. If the thread you kill happened to hold the GIL, you're most likely going to have a deadlock.

Other non-solutions include using `sys.settrace` in the thread. In addition to making the thread code horribly slow, this will also fail to work when the thread calls into C functions. The same is true for approaches attempting to raise an exception in another thread – the exception will get ignored if the thread is busy inside some C call.

This is where many people give up on threads and suggest using sub-processes instead. However, a process is not as lightweight as a thread, and if you need to run many functions with a timeout (or run a single function often) you have to be aware of the costs of creating and destroying a child process each time. Besides, if the sub-process has access to some shared resources, many of the troubles of threads surface here too.

In general, though, with some care sub-processes can be made to work. The `multiprocessing` package can even make processes as simple to use as threads, exposing similar APIs to `threading`. Additionally, it provides the `multiprocessing.Pool` class that can help lower the costs of process creation and destruction – assuming that the function we want to timeout does terminate most of the time before the timeout is reached.

Another reasonable solution is to make the computation cooperative, i.e. call back on the invoking code occasionally asking if it's time to finish. This is a technique well familiar to GUI programmers, where a function invoked from the GUI main loop should not run for too long, and should break its work to chunks.

An additional aspect to consider is that often long-running computations involve IO such as sockets. In this case, if a timeout is required, it's recommended to use asynchronous IO which naturally supports interruptions. Unfortunately, asynchronous IO also makes code more convoluted and difficult to write. Frameworks exist to alleviate this burden – the best known for Python is probably Twisted. Take a look at it – it's a bag full of solutions for your IO problems.

So what we've seen here is a relatively simple problem, which unfortunately has no really simple solution in Python. The blame here is on the problem, not the language, however. Even in languages that do allow killing threads (for example, C with native OS APIs), this is a discouraged practice – fickle and hard to get exactly right.

---

[1] http://code.activestate.com/recipes/473878-timeout-function-using-threading/
[2] http://code.activestate.com/recipes/576780-timeout-for-nearly-any-callable/
[3] If you're skeptical, make `func` a simple endless loop that prints something out every once in a while. You'll note that this keeps getting printed even after the timeout has expired and the thread was "stopped". Checking `threading.active_count()` is another telling clue.

Related posts:

1. Python threads: communication and stopping

2. Code sample – socket client thread in Python
3. Python – parallelizing CPU-bound tasks with multiprocessing
4. Shared counter with Python's multiprocessing
5. Python impressions

This entry was posted on Monday, August 22nd, 2011 at 05:50 and is filed under Python. You can follow any responses to this entry through the RSS 2.0 feed. You can skip to the end and leave a response. Pinging is currently not allowed.

16 Responses to "How (not) to set a timeout on a computation in Python"

1. *David Fraser* Says:
August 22nd, 2011 at 11:17

Good summary. As you say, the best approach is to write the code so it knows points at which it can be interrupted. I've found setting `threading.Event()` objects to signal the desire to abort is helpful here

I've done the naughty TerminateThread thing, but only on shutdown – it seems to work better than having to wait for the remaining threads in some cases

Another option is to call into the C API ThreadAsyncRaise through ctypes – this isn't always successful but can sometimes work.

http://trac.sjsoft.com/browser/trunk/j5/src/j5/OS/ThreadRaise.py and
http://trac.sjsoft.com/browser/trunk/j5/src/j5/OS/ThreadControl.py for some example code

2. *eliben* Says:
August 22nd, 2011 at 11:28

David Fraser,

> Good summary. As you say, the best approach is to write the code so it knows points at which it can be interrupted. I've found setting threading.Event() objects to signal the desire to abort is helpful here

I also like `threading.Event()` for cooperative signaling between threads.

> I've done the naughty TerminateThread thing, but only on shutdown – it seems to work better than having to wait for the remaining threads in some cases
> Indeed, when you're shutting down and going to exit anyway, forcibly killing the thread is OK.

Yes, when you're exiting anyway, forcibly killing the thread is usually fine.

> Another option is to call into the C API ThreadAsyncRaise through ctypes – this isn't always successful but can sometimes work.

I mentioned this approach in the post – the problem is that it won't really raise the exception if the thread is currently executing C code.

3. *Sycren* Says:
August 22nd, 2011 at 18:32

Would it not be possible to break up a function into smaller pieces which each check against the timeout?
So instead of trying to kill the thread (which is not possible), by adding the checks more often, you could divert the thread easing up computational power..

4. *eliben* Says:
August 22nd, 2011 at 19:49

Sycren,

Definitely. This is exactly what I meant in the post in the paragraph starting with *"Another reasonable solution is to make the computation cooperative, i.e. call back on the invoking code occasionally asking if it's time to finish"*

5. *Steve* Says:
August 22nd, 2011 at 20:10

Heads-up: setting a default arg to a mutable object in python is a big no-no.

For an example of why, run this:

```
def foo(a, b=[]):
  b.append(a)
  print b

foo('test')
foo('test2')
```

The output will be:

```
['test']
```

```
['test', 'test2']
```

This is because the defaults are defined at the function level, not within the function.

6. *Joe* Says:
August 22nd, 2011 at 20:16

I use multiprocessing (backticks doesn't work, at least in the preview)

```
import time
from multiprocessing import Process

class TestClass():
.   def test_f(self, name):
..      ctr = 0
..     while True:
...         ctr += 1
...          print name, ctr
...          time.sleep(1.0)

if __name__ == '__main__':
.CT=TestClass()
.p = Process(target=CT.test_f, args=('P1',))
.p.start()

.## sleep for 5 seconds and terminate
.time.sleep(5.0)
.p.terminate()
```

7. *eliben* Says:
August 22nd, 2011 at 20:27

Steve,

Default values for keyword arguments are a no-no if the function is going to modify the arguments. If they're read only, it's OK.

Anyhow, this is not the part of the sample code I wanted to highlight 😃 But your observation is correct.

9. *Travis Cline* Says:
August 22nd, 2011 at 21:19

I just wanted to note that with coroutine libraries in python asynchronous io is not really convoluted. See how one utilizes gevent for example. Here are how you approach timeouts in gevent: http://www.gevent.org/gevent.html#timeouts

Paul Says:
August 23rd, 2011 at 00:18

Generally a good roundup until you dismissed the subprocess method as being too heavyweight. Too heavy compared to what other sensible way of handling this? As though making the computation co-operative was always simple or even possible. And on what evidence is process start and teardown too resource intensive? You've just glossed over the best solution aside for no good reason.

10. *eliben* Says:
August 23rd, 2011 at 05:30

Paul,

I don't see how I have "dismissed" the subprocess method. I've just said *"you have to be aware of the costs of creating and destroying a child process each time."*

11. *matt harrison* Says:
August 23rd, 2011 at 08:11

I've implemented the multiprocessing timeout that you refer to: http://panela.blog-city.com/how_to_timeout_in_python_with_multiprocessing.htm

12. *Steve* Says:
August 23rd, 2011 at 09:31

I am aware that it doesn't cause an issue in this specific code chunk (although I must admit, I had to check to see what would happen if **kwargs was taken in and modified by the function pointed to by func – it turns out that it makes a copy of the keyword args at some point, so even if you modify it, the original is not affected).

My point is that bad practices should probably still be considered bad practices even if they don't cause a bug in a given example. It's not immediately obvious to somebody who has not encountered this issue that modifying kwargs will result in this issue.

13. *eliben* Says:
August 23rd, 2011 at 09:37

Matt,

Well done.

`multiprocessing.Process.terminate` is the key ingredient that's lacking from thread APIs

14. *Nick Coghlan* Says:
August 23rd, 2011 at 13:21

As of Python 3.3, the stdlib will have a faulthandler module which can be used to detect and report hung threads, although it won't try to interrupt them.

http://docs.python.org/dev/library/faulthandler

15. *eliben* Says:
August 23rd, 2011 at 13:42

Nick,

`faulthandler` is neat and I've been following its development with interest. However, will it really be applicable to this problem? AFAIU it can only be used to force dump of all threads, or give some thread the ability to dump its own traceback.

16. *Thomas Fanslau* Says:
August 24th, 2011 at 11:05

First thing learned about Threads nearly 30 years ago: Never stop them from the outside. Even thinking how to do that is wrong. Just imagine the thread as a black box that you have no idea what it is doing right now when you "pull the plug" and you see all this resouce allocation problems creeping up.

In planning I always treat Threads as external far away teflon-coated Black-Box-Processes that by "accident" share memory with me. Find a way to talk with them. Tell them to go away ... Accept if they don't want to ... Like In-Laws that you can't stand

Once you accept that you will learn that your view on Threads and how to use them will take a twist into a formerly-absurd territory. And from there on everything will be fine

Leave a Reply

Name (required)

Mail (will not be published) (required)

Website

Write the number 4 here (required)

To post code with preserved formatting, enclose it in `backticks` (even multiple lines)

Submit Comment

Preview

A preview will appear here

Eli Bendersky's website is powered by WordPress
Entries (RSS) and Comments (RSS).