

Lucretiel / tcl_async_demo.py

Last active 4 months ago • Report abuse


☆ Star

<> Code

Revisions 6

☆ Stars 17

Forks 1

 tcl_async_demo.py

```
1  from tkinter import *
2  import asyncio
3  from functools import wraps
4  import websockets
5
6  def runloop(func):
7      '''
8      This decorator converts a coroutine into a function which, when called,
9      runs the underlying coroutine to completion in the asyncio event loop.
10     '''
11     func = asyncio.coroutine(func)
12     @wraps(func)
13     def wrapper(*args, **kwargs):
14         return asyncio.get_event_loop().run_until_complete(func(*args, **kwargs))
15     return wrapper
16
17 @asyncio.coroutine
18 def run_tk(root, interval=0.05):
19     '''
20     Run a tkinter app in an asyncio event loop.
21     '''
22     try:
23         while True:
24             root.update()
25             yield from asyncio.sleep(interval)
26     except TclError as e:
27         if "application has been destroyed" not in e.args[0]:
28             raise
29
30 @asyncio.coroutine
31 def listen_websocket(url):
32     '''
33     Connect to a websocket url, then print messages received on the connection
34     until closed by the server.
35     '''
36     ws = yield from websockets.connect(url)
```

```

37     while True:
38         msg = yield from ws.recv()
39         if msg is None:
40             break
41         print(msg)
42
43 @runloop
44 def main():
45     root = Tk()
46     entry = Entry(root)
47     entry.grid()
48
49     def spawn_ws_listener():
50         return asyncio.async(listen_websocket(entry.get()))
51
52     Button(root, text='Print', command=spawn_ws_listener).grid()
53
54     yield from run_tk(root)
55
56 if __name__ == "__main__":
57     main()

```



nameoftherose commented on 17 Nov 2016 • edited ▼

Thank you for sharing this.

I have transliterated your script to use the async/await syntax. I have also replaced the websocket server with a simple tcp server to ease testing.

[embedding tkinter in asyncio](#)



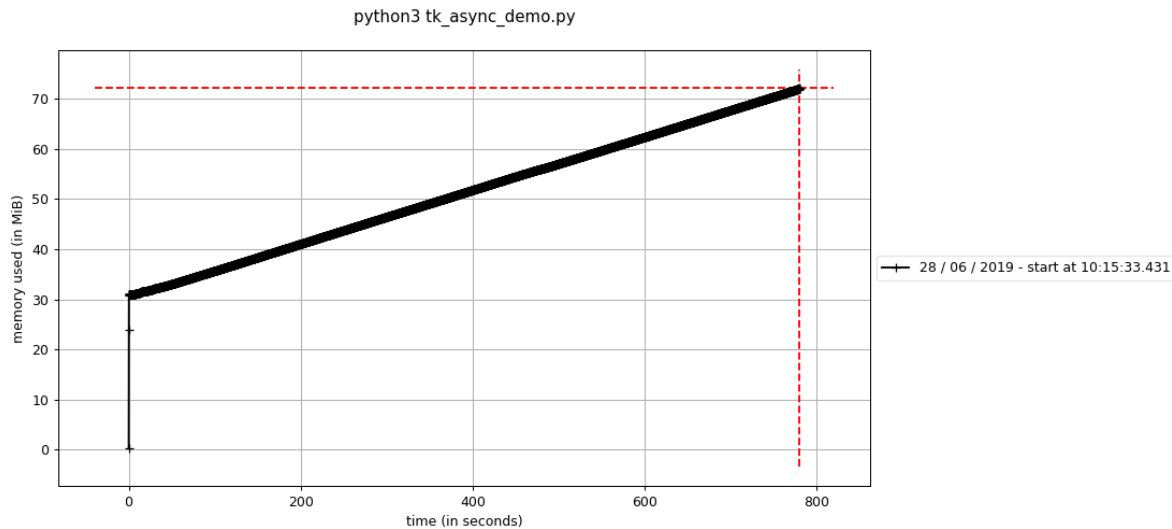
Restioson commented on 23 Apr 2017

What license is this? Can I use the `run_tk` method in my code?



chmedly commented on 28 Jun 2019

Please note that this technique (`tk.update()` driven by an async loop) causes increasing memory consumption as the program runs. I ran `nameoftherose`'s version of this in python 3.7 with `mprof` and attached an image of the plot results below. This is the same behavior I've observed in every implementation of this that I've tried. I would assume that it's initializing all the tk objects every time it updates and the old objects are not being garbage collected.



 **rudasoftware** commented on 2 Jul 2019

@chmedly, I think this behavior may be more due to the specifics of the rest of the async code rather than something inherent to coroutine-ating `tk.update()`. Also, may or may not be something introduced after python 3.6, since that's the version I'm testing with. TL;DR - I profiled a couple approaches to async `tkinter` and did *not* see the same continuously increasing memory consumption.

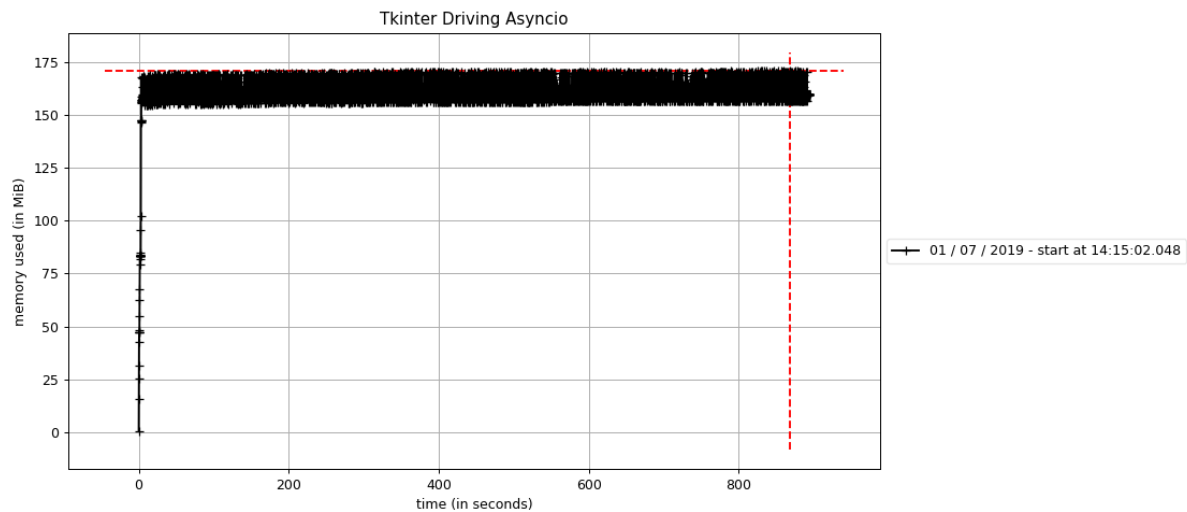
My app *doesn't* communicate over the network, but it does do I/O with locally connected cameras. It captures uncompressed 1928x1448 frames at ~10Hz and draws them to a `matplotlib` canvas, so the data throughput is relatively high. The heavy lifting is being done by a thread-pool executor, because the camera API is blocking. Your mileage may vary, obviously.

I came across this gist while researching the same question (dang `tkinter` and its notorious thread-unsafeness and poor concurrency), and your memory graph scared me. :)

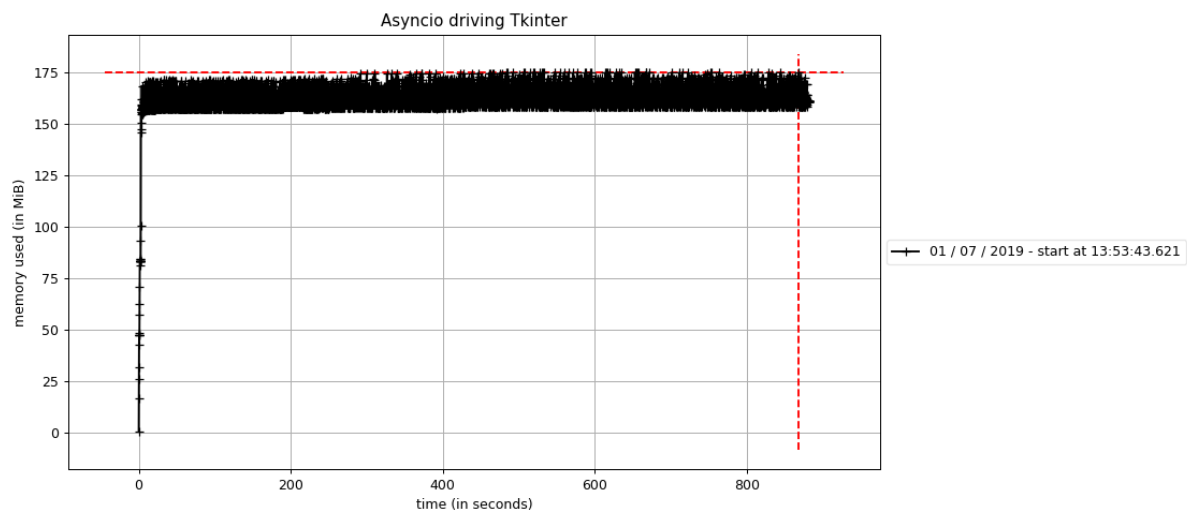
I tried an alternative approach (suggested [here](#)), essentially the inverse of this technique: instead of driving `tkinter` event loop from `asyncio`, a custom `asyncio` event loop driven by `tkinter` after `method`. It's "old" `asyncio` from mid-2015, so I had to make a few tweaks to get it up to ~python 3.6 API level. It worked, but performance of dragging/resizing was abysmal. More importantly, it wouldn't exit properly if I had a sync function operating inside a threaded executor. Techniques for graceful shutdown didn't work well, I believe because the lifetime of the loop was incorrectly coupled with the lifetime of the tcl loop: the loop was told it "finished" when the tcl app was destroyed, but the tcl loop is needed to call the correct cleanup code for incomplete futures.

I'm not extremely savvy with `asyncio` OR `tkinter` internals, so it's possible there may be a better way to tweak their example to be more complete/performant/correct (e.g. they inherit from `asyncio.base_events.BaseEventLoop`, which is not recommended). But, its characteristics were unacceptable and I didn't have time to dig deeper (especially into the opaque black-box of Tcl). So, I figured I'd give this gist a spin.

Other repo strategy



This gist strategy (default interval of 50ms, limiting to 20 fps)

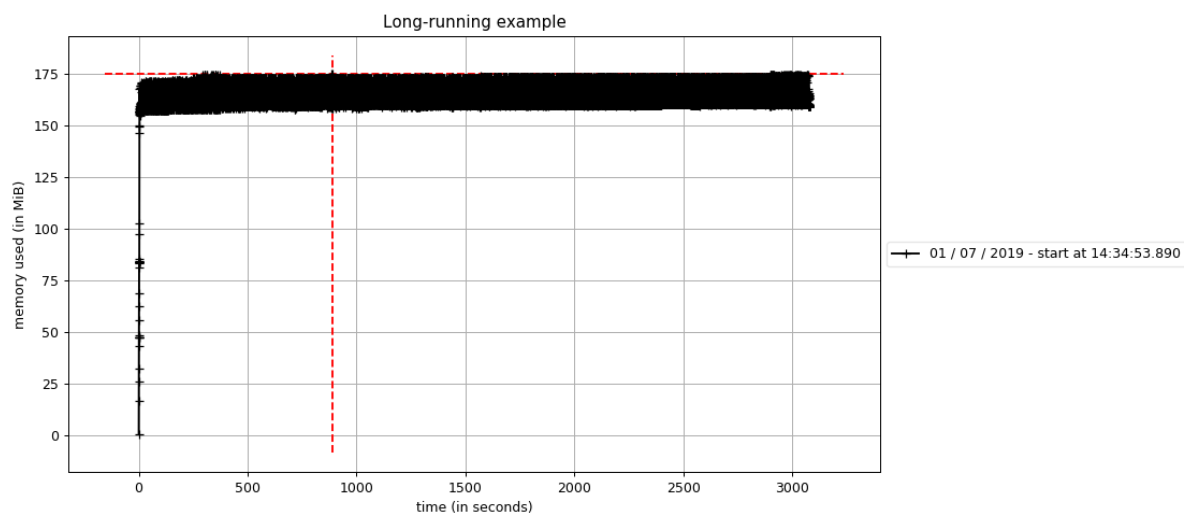


I'm actually surprised how similar the usage patterns seem to be. I see spikes which likely correspond to old frames being gc'd, but no linear, monotonic increase of overall memory consumption.

Update: I realized I could still see a slight upward trend over time in my graphs. I decided to run a longer profile (while asynchronously watching an `asyncio` talk) to get some additional data.

The data in your graph show a memory consumption of $\sim 54\text{KiB/s}$ which, while not "huge", will definitely soon spell disaster if that trend continues unabated.

This gist strategy (shorter interval of 10ms, limiting to 100 fps)



The beginning of the graph looks almost identical to the graph I posted previously. There's a gradual rise from ~ 170 to $\sim 175\text{MiB}$ that appears to level off, with peak usage just before ending profile at $\sim 15\text{min}$: a consumption rate of $\sim 0.006\text{KiB/s}$. It then stays right around that peak for the rest of the 50min duration. The red dotted lines indicating peak usage also appear at almost exactly the same place! To me, that looks like the point where the garbage collector reached equilibrium, because the program essentially does exactly the same thing over and over again across its running time--a rather decent benchmark.

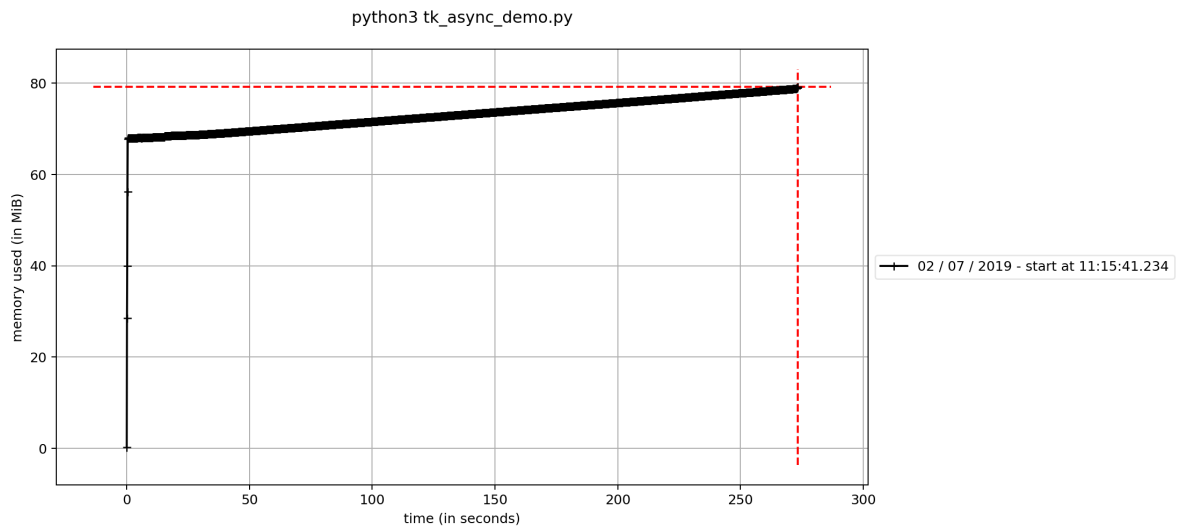
I have to conclude that whatever is causing your (admittedly alarming) memory graph is *not* due to continuous calls to `update`. It's probably something surrounding the TCP client stuff, another area where I'll have to admit insufficient experience.

 chmedly commented on 2 Jul 2019

Hmm. Perhaps Python3.7 is doing something differently. I don't have 3.6 installed on any machines at this point to test for a difference. Like I said, I ran `nameoftherose`'s script because I didn't want to spend too much time adapting the code on this page. I've also run other examples of this technique using dummy tasks and my own implementation in a websocket server/client scenario and see the same rise in memory use as it runs.

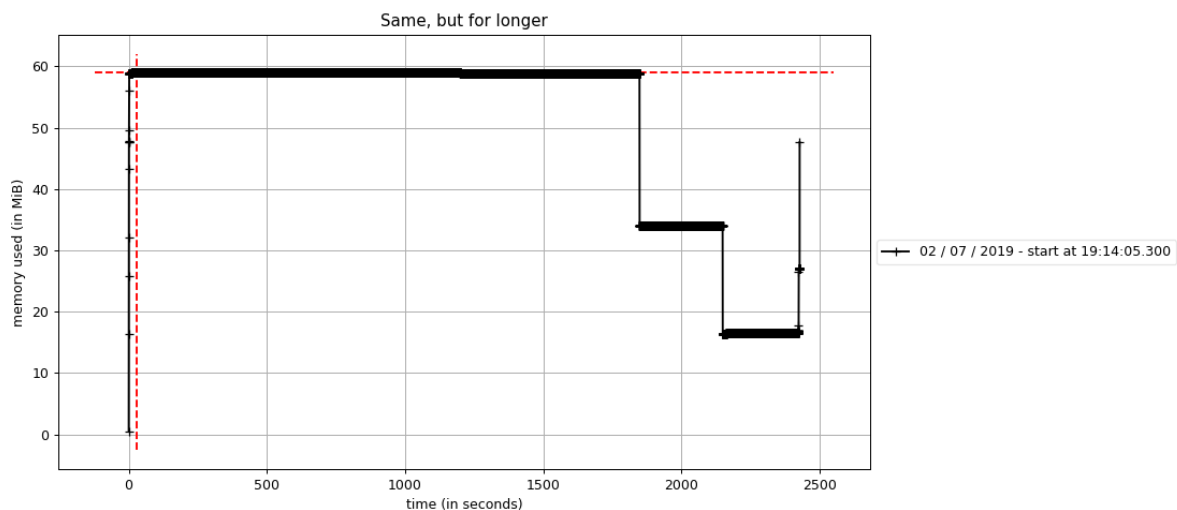
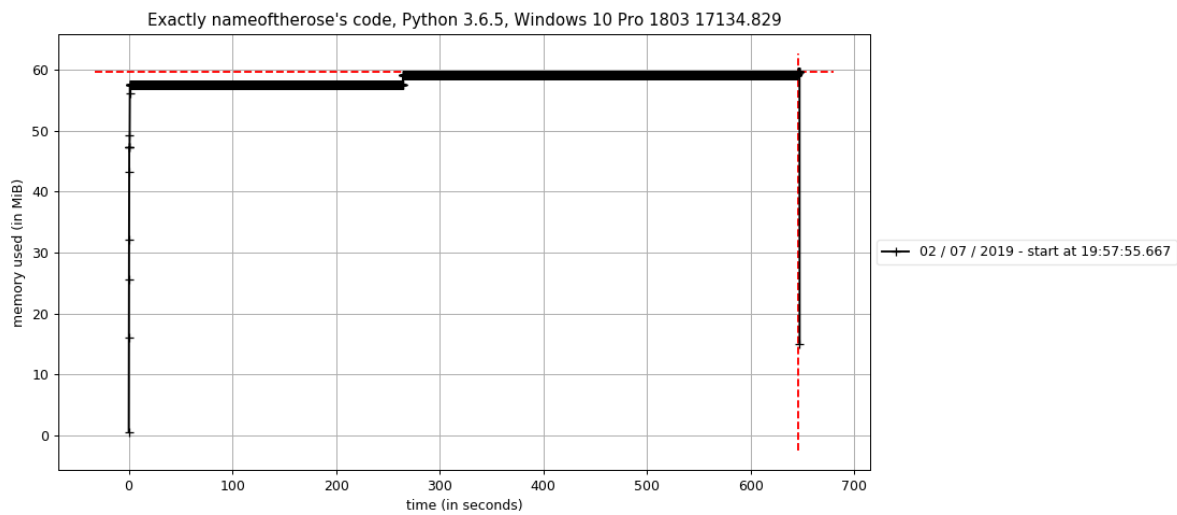
 chmedly commented on 2 Jul 2019

I found that I do have python3.6.1 installed on another Mac. So, I ran the same nameoftherose code there. I find the same increasing memory use. Interestingly, 3.6.1 starts off with double the memory usage of 3.7.3! But that's neither here nor there. Mind you, I'm not clicking on the button. I'm just starting the app and then ending by closing the tk window. So, the socket stuff doesn't get run.



 rudasoftware commented on 3 Jul 2019

Hmm, that's very intriguing. I decided to run nameoftherose code, exactly as written, on my system to see what happen.

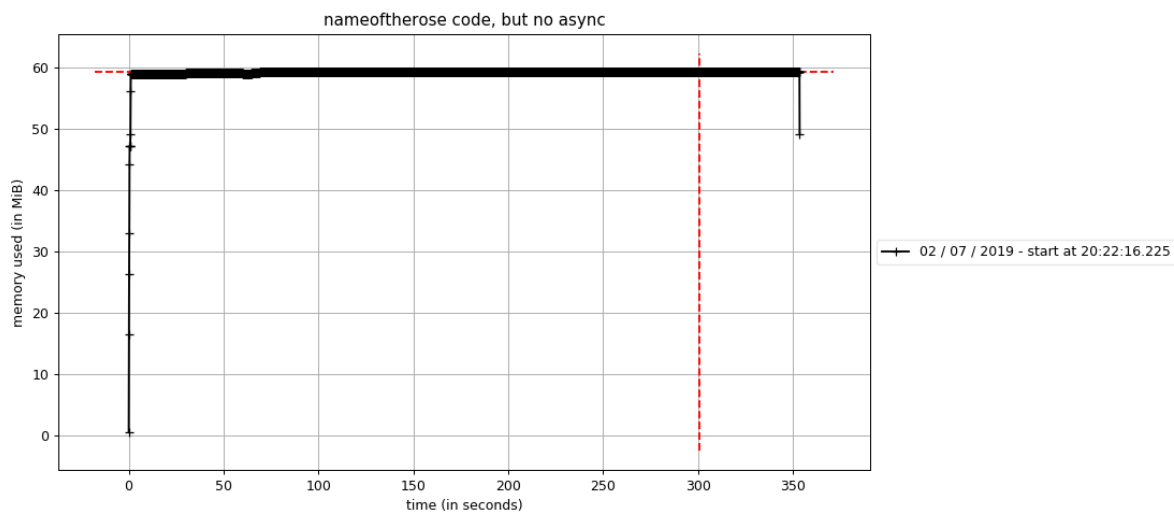


It's possible that the memory consumption baseline of 3.6 vs 3.7 is down to some of the under-the-hood optimizations made to `asyncio` specifically and the data model generally.

What's weird is... my graphs don't show that linearly increasing behavior at all. I start right around 60 and never really go above it. I can't explain the weird sudden step-changes at all. Maybe the scheduler realizes that the event loop isn't really doing all that much and pages unused stuff out of memory? Would that even be opaque to `mprof`? Maybe some reference to some bootstrapping object finally gets gc'd, maybe also because of the loop doing basically nothing? Is the consumption you experience caused by MacOS somehow? This really raises more questions than answers...

I'd be inclined to maybe write to the python mailing list, if for nothing else than to satisfy curiosity.

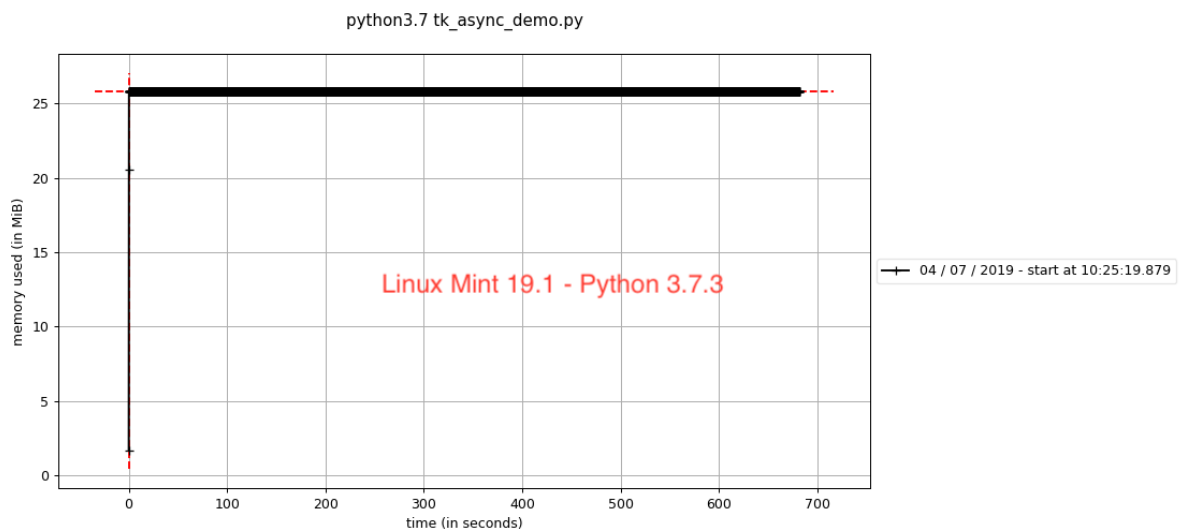
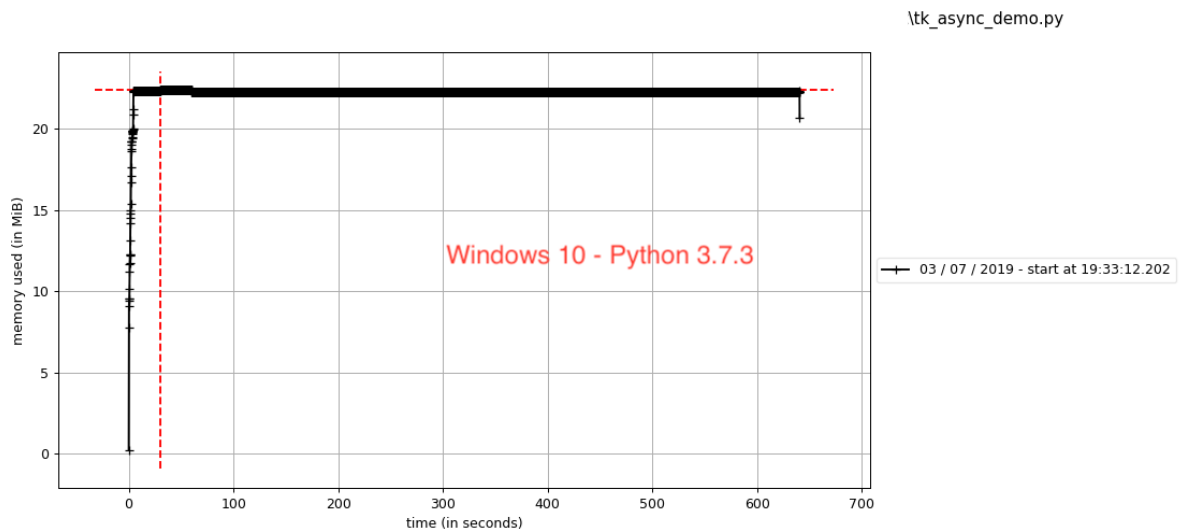
Oooh OOOOH, what happens if you profile the same `tkinter` code, but without the `asyncio` stuff? What happens if it just spins on `mainloop`?



Except for the weird discontinuities in the previous graphs, I'd be hard pressed to distinguish the sync from the async runs.

 **chmedly** commented on 4 Jul 2019

So, I tried this on Windows 10 with Python 3.7.3 AND Linux Mint 19.1 with Python 3.7.3 and neither of them appear to have memory issues. It looks like this is isolated to MacOS. And High Sierra (10.13.6) is the only version that I've tried this with. I haven't tried removing the `async` portion yet. Can you post your code showing how you set that up?



 rudasoftware commented on 8 Jul 2019

The only changes necessary are at the very end of the listing.

Original

```
async def main():
    root = Tk()
    entry = Entry(root)
    entry.grid()

    def spawn_ws_listener():
        addr=entry.get().split(':')
        print('spawn',addr)
        return asyncio.ensure_future( tclient( addr[0],int(addr[1])) ) )
```

```

Button(root, text='Connect', command=spawn_ws_listener).grid()

await run_tk(root)

if __name__ == "__main__":
    asyncio.get_event_loop().run_until_complete(main())

```

Synchronous Equivalent

```

def main():
    root = Tk()
    entry = Entry(root)
    entry.grid()

    # NOTE: Just don't click the button, as it won't work.
    #       Makes no difference for testing, because we weren't clicking the button anyway.
    def spawn_ws_listener():
        addr=entry.get().split(':')
        print('spawn',addr)
        return asyncio.ensure_future( tclient( addr[0],int(addr[1])) ) )

    Button(root, text='Connect', command=spawn_ws_listener).grid()

    root.mainloop()

if __name__ == "__main__":
    main()

```

 **rudasoftware** commented on 8 Jul 2019

What happens if you run it on MacOS for longer? Try profiling for ~1h and see if it ever reaches equilibrium. Maybe it just takes longer to get to a steady state? The extra memory usage compared to Windows is pretty egregious in any case, but I'd be curious to see how bad the problem could get for a long-running app...

 **chmedly** commented on 9 Jul 2019

Hmm. With those modifications you never call `run_tk()` and therefore don't ever call `root.update()`. Instead what you're showing is just a standard tk app with `root.mainloop()` handling the refreshing. How about this code?

```

from tkinter import *
import asyncio
import time

def run_tk(root, interval=0.05):
    try:

```

```

        while True:
            root.update()
            time.sleep(interval)
except TclError as e:
    if "application has been destroyed" not in e.args[0]:
        raise

async def tclient(addr,port):
    print('tclient',addr,port)
    try:
        sock ,_= await asyncio.open_connection(host=addr,port=port)
        #print(sock)
        #f=sock.as_stream()
        while True:
            #data = yield from f.readline()
            data = await sock.readline()
            if not data:break
            data=data.decode()
            print(data,end='\n' if data[-1]!='\r' else'')
    except:
        pass

def main():
    root = Tk()
    entry = Entry(root)
    entry.grid()

    def spawn_ws_listener():
        addr=entry.get().split(':')
        print('spawn',addr)
        return asyncio.ensure_future( tclient( addr[0],int(addr[1])) ) )

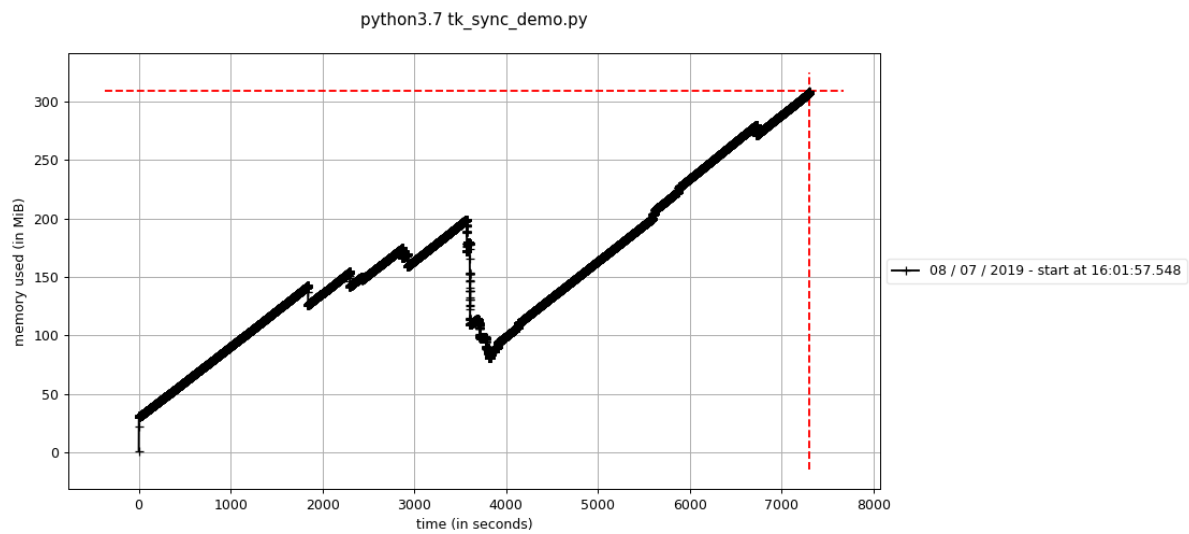
    Button(root, text='Connect', command=spawn_ws_listener).grid()

    runTk(root)

if __name__ == "__main__":
    main()

```


I ran it for about 2 hrs on MacOS with Python 3.7.3 and here's the Mprof Plot result. Some interesting memory releases but overall it still seems to be a constantly increasing bag of goodies.



 **rudasoftware** commented on 9 Jul 2019 • edited ▼

Right. I just wanted to see how `tkinter` runs "as intended" in the MacOS environment.

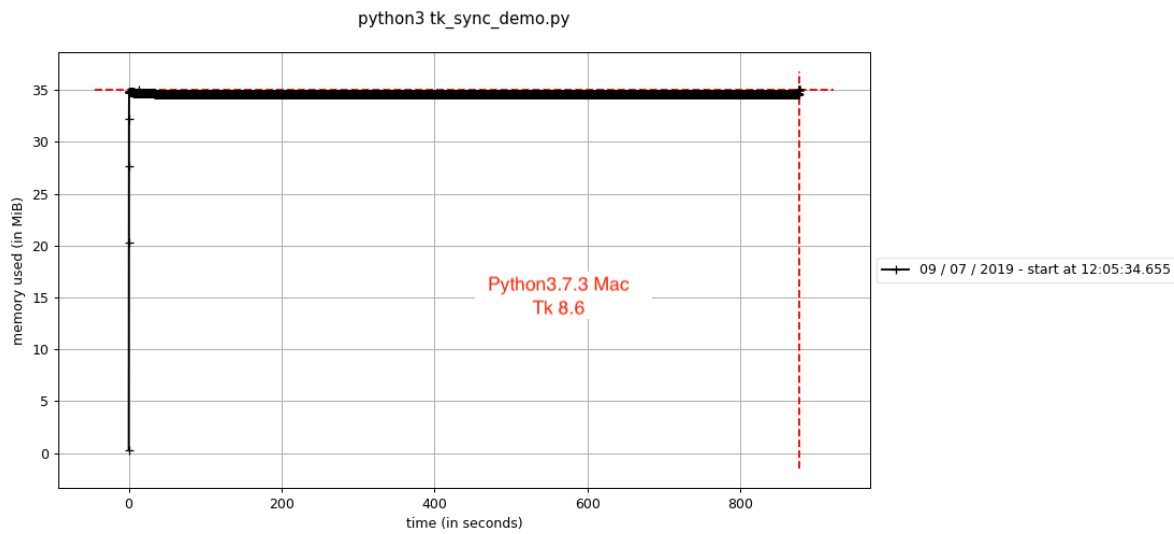
Yep, that's a goofy-looking unfortunate graph, haha.

 **chmedly** commented on 9 Jul 2019

I think I've figured it out. This page at tkdocs tells of bad things that happen with the elderly version of Tcl/Tk included with MacOs.

<https://tkdocs.com/tutorial/install.html>

They say to not use this version (typically 8.5) but instead to install the latest 8.6 (8.6.9 now). But, if you install Python with homebrew, there is currently no easy way to point it to a newly installed version of Tk. At this point in my research, the only way to get the updated Tcl/Tk to work with Python3 is to install Python without homebrew. Anyway, I've run the tk_sync_demo.py code (that I posted previously) on a Mac with Python 3.7.3 AND Tk version 8.6. Memory runs flatline. So, it looks like my intention of simplifying cross platform support by using a built-in library for the GUI (instead of PyQt5) is not as flawless as I had hoped.



 rudasoftware commented on 11 Jul 2019

Aww man, that's a bummer. Glad you found the culprit, though!