[Display the output of the program on GUI with tkinter?](https://stackoverflow.com/questions/48977473/display-the-output-of-the-program-on-gui-with-tkinter)

I would like to display my program's "live" output on GUI (all what printed in it). how can i access to my output? and what the right way to display it for example in text box?

edited: where am i wrong? (I would like that the "hello world" to appear inside the text box. (Test2 is the running program))

from tkinter import \*

from subprocess import \*

print("Hello world")

def func():

proc = Popen("Test2.py", stdout=PIPE, shell=True)

proc = proc.communicate()

output.insert(END, proc)

Master = Tk()

Check = Button(Master, text="Display output", command=func)

Quit = Button(Master, text="Exit", fg="red", command=Master.quit)

output = Text(Master, width=40, height=8)

Check.pack(padx=20, pady=8)

Quit.pack(padx=20, pady=18)

output.pack()

Master.mainloop()

I took the time to debug and modify the errorwindow.py module in my answer to [another question](https://stackoverflow.com/a/18091356/355230) so it will work in both Python 2 and 3—the code in the linked answer was written for Python 2.x. Note I only did the minimum necessary to get it functioning under the two versions. The modified version of the script has been named errorwindow3k.py (despite that fact it also works in Python 2).

The majority of the issues were simply due to module renaming, however there was a harder one to figure-out that turned-out was due to the switch to Unicode strings being the default string-type in version 3—apparently (on Windows anyway), pipes between processes are byte-streams, not Unicode characters. Fortunately the "fix" of decoding and then encoding the data on the other side also doesn't hurt in Python 2 which made correcting the problem fairly easy.

This nice thing is that using it is very easy. Just import it and from that point on any output sent to either sys.stderr or sys.stdout will cause tkinter-based output windows to appear as needed to display the information. In your sample code just insert import errorwindow3k somewhere before the print("Hello world").

File **errorwindow3k.py:**

# Code derived from Bryan Olson's source posted in this related Usenet discussion:

# https://groups.google.com/d/msg/comp.lang.python/HWPhLhXKUos/TpFeWxEE9nsJ

# https://groups.google.com/d/msg/comp.lang.python/HWPhLhXKUos/eEHYAl4dH9YJ

#

# See the comments and doc string below.

#

# Here's a module to show stderr output from console-less Python

# apps, and stay out of the way otherwise. I plan to make a ASPN

# recipe of it, but I thought I'd run it by this group first.

#

# To use it, import the module. That's it. Upon import it will

# assign sys.stderr.

#

# In the normal case, your code is perfect so nothing ever gets

# written to stderr, and the module won't do much of anything.

# Upon the first write to stderr, if any, the module will launch a

# new process, and that process will show the stderr output in a

# window. The window will live until dismissed; I hate, hate, hate

# those vanishing-consoles-with-critical-information.

#

# The code shows some arguably-cool tricks. To fit everthing in

# one file, the module runs the Python interpreter on itself; it

# uses the "if \_\_name\_\_ == '\_\_main\_\_'" idiom to behave radically

# differently upon import versus direct execution. It uses TkInter

# for the window, but that's in a new process; it does not import

# TkInter into your application.

#

# To try it out, save it to a file -- I call it "errorwindow.py" -

# - and import it into some subsequently-incorrect code. For

# example:

#

# import errorwindow

#

# a = 3 + 1 + nonesuchdefined

#

# should cause a window to appear, showing the traceback of a

# Python NameError.

#

# --

# --Bryan

# ----------------------------------------------------------------

#

# martineau - Modified to use subprocess.Popen instead of the os.popen

# which has been deprecated since Py 2.6. Changed so it

# redirects both stdout and stderr. Added numerous

# comments, and also inserted double quotes around paths

# in case they have embedded space characters in them, as

# they did on my Windows system.

#

# Recently updated it to work in both Python 2 and Python 3.

"""

Import this module into graphical Python apps to provide a

sys.stderr. No functions to call, just import it. It uses

only facilities in the Python standard distribution.

If nothing is ever written to stderr, then the module just

sits there and stays out of your face. Upon write to stderr,

it launches a new process, piping it error stream. The new

process throws up a window showing the error messages.

"""

import subprocess

import sys

try:

import thread

except ModuleNotFoundError: # Python 3

import \_thread as thread

import os

EXC\_INFO\_FILENAME = 'exc\_info.txt'

if \_\_name\_\_ == '\_\_main\_\_': # When spawned as separate process.

# create window in which to display output

# then copy stdin to the window until EOF

# will happen when output is sent to each OutputPipe created

try:

from Tkinter import BOTH, END, Frame, Text, TOP, YES

import tkFont

import Queue

except ModuleNotFoundError: # Python 3

from tkinter import BOTH, END, Frame, Text, TOP, YES

import tkinter.font as tkFont

import queue as Queue

Q\_EMPTY = Queue.Empty # An exception class.

queue = Queue.Queue(1000) # FIFO

def read\_stdin(app, bufsize=4096):

fd = sys.stdin.fileno() # File descriptor for os.read() calls.

read = os.read

put = queue.put

while True:

put(read(fd, bufsize))

class Application(Frame):

def \_\_init\_\_(self, master=None, font\_size=8, text\_color='#0000AA', rows=25, cols=100):

Frame.\_\_init\_\_(self, master)

# Create title based on the arguments passed to the spawned script:

# argv[0]: name of this script (ignored)

# argv[1]: name of script that imported this module

# argv[2]: name of redirected stream (optional)

if len(sys.argv) < 2:

title = "Output stream from unknown source"

elif len(sys.argv) < 3:

title = "Output stream from %s" % (sys.argv[1],)

else: # Assume it's a least 3.

title = "Output stream '%s' from %s" % (sys.argv[2], sys.argv[1])

self.master.title(title)

self.pack(fill=BOTH, expand=YES)

font = tkFont.Font(family='Courier', size=font\_size)

width = font.measure(' ' \* (cols+1))

height = font.metrics('linespace') \* (rows+1)

self.configure(width=width, height=height)

self.pack\_propagate(0) # Force frame to be configured size.

self.logwidget = Text(self, font=font)

self.logwidget.pack(side=TOP, fill=BOTH, expand=YES)

# Disallow key entry, but allow text copying with <Control-c>

self.logwidget.bind('<Key>', lambda x: 'break')

self.logwidget.bind('<Control-c>', lambda x: None)

self.logwidget.configure(foreground=text\_color)

self.logwidget.insert(END, '==== Start of Output Stream ====\n\n')

self.logwidget.see(END)

self.after(200, self.start\_thread, ()) # Start polling thread.

def start\_thread(self, \_):

thread.start\_new\_thread(read\_stdin, (self,))

self.after(200, self.check\_q, ())

def check\_q(self, \_):

log = self.logwidget

log\_insert = log.insert

log\_see = log.see

queue\_get\_nowait = queue.get\_nowait

go = True

while go:

try:

data = queue\_get\_nowait().decode() # Must decode for Python 3.

if not data:

data = '[EOF]'

go = False

log\_insert(END, data)

log\_see(END)

except Q\_EMPTY:

self.after(200, self.check\_q, ())

go = False

app = Application()

app.mainloop()

else: # when module is first imported

import traceback

class OutputPipe(object):

def \_\_init\_\_(self, name=''):

self.lock = thread.allocate\_lock()

self.name = name

def flush(self): # NO-OP.

pass

def \_\_getattr\_\_(self, attr):

if attr == 'pipe': # Attribute doesn't exist, so create it.

# Launch this module as a separate process to display any output

# it receives.

# Note: It's important to put double quotes around everything just in

# case any have embedded space characters.

command = '"%s" "%s" "%s" "%s"' % (sys.executable, # executable

\_\_file\_\_, # argv[0]

os.path.basename(sys.argv[0]), # argv[1]

self.name) # argv[2]

#

# Typical command and arg values on receiving end:

# C:\Python3\python[w].exe # executable

# C:\vols\Files\PythonLib\Stack Overflow\errorwindow3k.py # argv[0]

# errorwindow3k\_test.py # argv[1]

# stderr # argv[2]

# Execute this script directly as \_\_main\_\_ with a stdin PIPE for sending

# output to it.

try:

# Had to also make stdout and stderr PIPEs too, to work with pythonw.exe

self.pipe = subprocess.Popen(command, bufsize=0,

stdin=subprocess.PIPE,

stdout=subprocess.PIPE,

stderr=subprocess.PIPE).stdin

except Exception:

# Output exception info to a file since this module isn't working.

exc\_type, exc\_value, exc\_traceback = sys.exc\_info()

msg = ('%r exception in %s\n' %

(exc\_type.\_\_name\_\_, os.path.basename(\_\_file\_\_)))

with open(EXC\_INFO\_FILENAME, 'wt') as info:

info.write('fatal error occurred spawning output process')

info.write('exeception info:' + msg)

traceback.print\_exc(file=info)

sys.exit('fatal error occurred')

return super(OutputPipe, self).\_\_getattribute\_\_(attr)

def write(self, data):

with self.lock:

data = data.encode() # Must encode for Python 3.

self.pipe.write(data) # First reference to pipe attr will cause an

# OutputPipe process for the stream to be created.

# Clean-up any left-over debugging files.

try:

os.remove(DEBUG\_FILENAME) # Delete previous file, if any.

except Exception:

pass

try:

os.remove(EXC\_INFO\_FILENAME) # Delete previous file, if any.

except Exception:

pass

# Redirect standard output streams in the process that imported this module.

sys.stderr = OutputPipe('stderr')

sys.stdout = OutputPipe('stdout')