dpbl

dp's blog

dp in tutorial February 12, 2017February 23, 2018 2,414 Words

A tutorial on python-daemon – or – Why doesn't python-daemon have any documentation?

> *Quick note: this tutorial implies you are using one of the Unix-like (https://upload.wikimedia.org/wikipedia/commons/7/77/Unix_history-simple.svg) operating systems, since daemons are specific to Unix.*

Introduction

A few weeks ago I needed to create a daemon for a school project (https://github.com/dpdani/SUS). I had never really dealt with daemons before so I browsed on the Internet for what they were and how they worked.

After reading some pages, I found out how hard it seemed to be working with daemons: you need to deal with the correct forking (https://en.wikipedia.org/wiki/Fork_(system_call)) of a process, prevent core dump (https://en.wikipedia.org/wiki/Core_dump) generation, change root and working directories, change process and file umasks (https://en.wikipedia.org/wiki/Umask) and ownership, and quite a lot of other OS-related stuff you can find listed here (https://www.python.org/dev/peps/pep-3143/#correct-daemon-behaviour). By the way, at this time I'm still not quite sure of how these conditions came to be.

Now, I could have gone coding through all of that stuff. ~~And that's exactly what I did.~~

But nope.

So my inner sloth got back on Google.

Wait. What is a daemon?

> *If you're only interested in the tutorial part of this article, that's ok* 🙂

A little side note. Although the word "daemon" might sound like something diabolical, it has nothing to do with Satan, in fact (https://www.freebsd.org/copyright/daemon.html) it comes from the Greek word δαίμων, which refers to the spirits people have inside and which eventually define them.

A daemon is a process (https://en.wikipedia.org/wiki/Process_(computing)). No more and no less then your browser's process is. The key difference is, though, that a daemon is a process that doesn't need user input to work. Think, for instance, about a web server which isn't waiting for its own user to perform some action, but rather it's waiting for some other host on the network to perform a request. Such request needs to be processed without any human taking part in it.

So a daemon is a process that performs what you could think of as a background task. May it be a server like a web server or an ssh server, or something more complicated like systemd (https://github.com/systemd/systemd).

OS-wise, daemons are specific to the world of Unix. If you're running Windows or Mac OS X, you should keep in mind that Unix daemons do have their respective counter-part in other OSes: Windows (https://msdn.microsoft.com/en-us/library/windows/desktop/ms685141(v=vs.85).aspx) has its so-called services; OSX (https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/Introduction.html) sometimes calls them "daemons", sometimes "agents" and they still pretty much work just as Unix daemons, but it sort of expects you to make them compatible with launchd (https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingLaunchdJobs.html).

How can I create a daemon?

There are a lot of ways you can create a daemon in Unix, since nobody enforces or supports one in favor of the others.

This is a (~~probably~~ *definitely* incomplete) list of ways you can create a daemon:

- Within the shell of your choice, type in the command you were trying to daemonize and add '&' at the end of it.

```
$ python spam.py &
```

And that's it. You have a daemon.

Now, while this is a very fast way to spawn a daemon, it might not be the most reasonable choice for a number of reasons: the process will output anything to your current shell (using '&' doesn't close the stdout and stderr file descriptors), you can't assign the daemon any PID lock file so multiple daemons might be running at the same time (I'll come back to the lock file later) and often that makes it generally harder to control the daemon.

Nonetheless this remains the fastest way to create a daemon in Unix. Just a character away. And sometimes speed and simplicity are exactly what you need.

- There are a couple recipes online that will do the job for you.

I won't go through the details of all of them, I'll just link a couple of them here:

Creating a daemon the python way — by Chad J. Schroeder (http://code.activestate.com/recipes/278731-creating-a-daemon-the-python-way/)

A simple unix/linux daemon in Python — by Sander Marechal (http://web.archive.org/web/20131017130434/http://www.jejik.com/articles/2007/02/a_simple_unix_linux_daemon_in_python/)

Daemon with start/stop/restart behavior — by Clark Evans (http://code.activestate.com/recipes/66012/#c9)

The problem these recipes share is that they are simple recipes. Simple text files somebody put on the Internet.

This does have some consequences: you cannot pip install them and you cannot have a simple way to know if there is going to be an update for any of them (crucial in a security-aware context), but still these recipes will get you the job done.

- Use "daemon (http://www.libslack.org/daemon/)". Here's a nice article

(https://blog.terminal.com/using-daemon-to-daemonize-your-programs/) about it.

This is a pretty nice utility, which will do most of the job for you, similarly to python-daemon, but if you're relying on python code for running your daemon, it may not be the best choice. Anyway, it's simple enough that you might want to look at it if you need to spawn deamons in C or any other language.

EDIT: As Thibault pointed out in the comment section, there are also other supervisor utilities available one popular of which is supervirord (http://supervisord.org/).

- Hack it yourself.

While quite a lot of people (like me) find it tedious to re-invent the wheel, others sometimes feel like they need to re-invent the whole TCP/IP stack from the ground up. I won't judge you for this. You're going to have a lot of fun with that 🙂

Just a couple of modules I found a lot of people were using while writing their homemade daemons: os.fork (https://docs.python.org/3/library/os.html#os.fork), os.setsid (https://docs.python.org/3/library/os.html#os.setsid), signal (https://docs.python.org/3/library/signal.html), subprocess (https://docs.python.org/3/library/subprocess.html).

- Use python-daemon.

Ok, here we are. This is the way I create daemons and the way I would recommend to most people.

It doesn't require a lot of knowledge about the underlying machinery that gets the daemon to run, its source code (https://pagure.io/python-daemon/tree/master) is rather readable, its APIs are very very simple (we'll go through these later), you can pip install it, it is still maintained (https://pagure.io/python-daemon/commits/) and it was going to be the standard (https://www.python.org/dev/peps/pep-3143/) way to create daemons in Python. Its main missing feature is its lack of documentation: the documentation you'll find online is sparse and you'll often need to look at its source code if you encounter any bugs.

So what is python-daemon?

Back in the first weeks of 2009 PEP 3143 (https://www.python.org/dev/peps/pep-3143/) was created. Its aim was to create "a package [in] the Python standard library that provides a simple interface to the task of becoming a daemon process."

While the goal was not an impossible task and quite some people were interested in seeing this project succeed, it didn't make it. The guy that was in charge of doing it simply didn't have enough time anymore and no one stepped in to save the project. Such a sad death for such a nice project 😟

This tragedy didn't affect the functionality of python-daemon too much (it does include basically anything you need for a daemon), but rather its documentation. As I said earlier, its weakest point is documentation: you'll find some inside the PEP (https://www.python.org/dev/peps/pep-3143/#daemoncontext-objects) and some within the code (https://pagure.io/python-daemon/blob/master/f/daemon/daemon.py#_295) itself.

And how do you make it work?

Ok, I guess I got you interested in python-daemon since you're still reading.

First, let me start off by telling you what you shouldn't use in this library: DaemonRunner. Googling python-daemon will find some (http://stackoverflow.com/a/9047339) pages (http://nanvel.name/2013/02/python-unix-daemon) that will point you to the DaemonRunner object to handle your daemon, but it is a deprecated (https://pagure.io/python-daemon/blob/master/f/daemon/runner.py#_49) part of the library.

Instead, you want to use the DaemonContext (https://www.python.org/dev/peps/pep-3143/#daemoncontext-objects) API which is used inside of DaemonRunner. It's true that DaemonRunner extends DaemonContext functionality, but it does so in a very old-fashioned way (doesn't use *argparse* for instance). This is probably the reason why it ended up being deprecated.

Without further ado, DaemonContext makes it super simple to start your daemon with just a context manager:

```
with daemon.DaemonContext():
    main()
```

This is the most basic configuration you can pass to DaemonContext, and it will actually create a well-behaving daemon with just one line of code and four spaces of indentation.

I'll give you an overview of what you can set in order to have a more complex and detailed configuration for the daemon you need.

Dealing with the file system.

A daemon is a fairly peculiar process: since it is unbound from human interaction, a daemon will have its own keys to be identified user-wise. This means that, regardless of the user that started a daemon, the daemon will have its own UID, GID (**U**ser/**G**roup **ID**), its own root and working directories, and its own umask (https://en.wikipedia.org/wiki/Umask).

Don't be afraid, DaemonContext will take care of this stuff for you, even with just the default configuration, but let's say that you need to customize this stuff. To change the root directory, useful for confining your daemon, simply set the *chroot_directory* argument to a valid directory on your file system. The same goes for the working directory, which is a more usual thing to do, under the argument *working_directory*. By default, DaemonContext will set your working directory to root "/".

An example*:

```
with daemon.DaemonContext(
        chroot_directory=None,
        working_directory='/var/lib/myprettylittledaemon'):
    print(os.getcwd())
```

*: in case you don't see on-screen the result of print, that's because you need to keep the stdout stream open. Such configuration is explained in the "Preserve files" paragraph below.

For the UID and GID, DaemonContext by default "will relinquish any effective privilege elevation inherited by the process" which is usually the reason why you need to change them. In case you don't find this satisfactory, the process is still pretty straight-forward: set them to what you need, provided that your user is granted permission to do so. In case your user doesn't have root permissions, DaemonContext will raise a *DaemonOSEnvironmentError* exception.

```
with daemon.DaemonContext(
        uid=1001,
        gid=777):
    print(os.getuid())
    print(os.getgid())
```

Additionally, you might want to set the daemon umask, which will set the mode the daemon will create files with:

```
with daemon.DaemonContext(
        umask=0o002):
    your_mask = os.umask(0)  # i'm doing this weird three lines trick
    print(your_mask)         # to print the umask set by DaemonContex
t
    os.umask(your_mask)      # due to the behaviour of os.umask (http
s://docs.python.org/3/library/os.html#os.umask).
```

Preserve files.

One thing to take into account when creating a daemon is that on start DaemonContext will close any open files you have around. This is normal and it's what it is supposed to do (https://www.python.org/dev/peps/pep-3143/#correct-daemon-behaviour).

Now, even though this is the behavior we should expect from the library, you might still need some files to be opened in your program. You can do this by declaring what files you won't need to be closed through the *files_preserve* argument. For instance:

```
some_important_file = open('AVERYBIGDATABASE', 'r')

with daemon.DaemonContext(
        files_preserve=[some_important_file]):
    print(some_important_file.readlines())
```

Along with your open files, DaemonContext will also close the standard streams file descriptors, namely *stdin*, *stdout* and *stderr*. By default it will redirect them to *os.devnull (https://docs.python.org/3/library/os.html#os.devnull)*. If you need to keep them open, simply set the *stdin*, *stdout* and *stderr* arguments according to your needs.

```
with daemon.DaemonContext(
        stdout=sys.stdout,
        stderr=sys.stderr):
    print("Hello World! Daemon here.")
```

Handling OS signals.

Signals (https://docs.python.org/3/library/signal.html) coming from the OS are important, regardless of whether you're switching your program to be daemonized. Furthermore, this makes it even more important for you to take care of such signals, since it might become the only way a human interacts with your process.

DaemonContext will conveniently let you define a dictionary in the *signal_map* argument that will be linked to the signals you might want to configure. Some popular ones are: SIGINT, SIGKILL, SIGTERM, SIGTSTP. You can find further details here (https://people.cs.pitt.edu/~alanjawi/cs449/code/shell/UnixSignals.htm).

```
import signal

def shutdown(signum, frame):  # signum and frame are mandatory
    sys.exit(0)

with daemon.DaemonContext(
        signal_map={
            signal.SIGTERM: shutdown,
            signal.SIGTSTP: shutdown
        }):
    main()
```

One at a time.

More than often daemons will use resources, such as a TCP port for a listening server or some files on disk. You'll probably want to make sure that there aren't multiple daemons conflicting for these resources.

To make sure that only one of your daemons is running at the same time, you can use a PID lock file, which is a file containing the PID of a process that will prevent the same program from running on more than one instance. Please note that it is the duty of the newly spawned process (handled within DaemonContext) to check the lock file and abort the start procedure. If you're already familiar with threading.Lock (https://docs.python.org/3/library/threading.html#threading.Lock) the concept is basically the same.

You can set a lock file like this:

```
import lockfile

with daemon.DaemonContext(
        pidfile=lockfile.FileLock('/var/run/spam.pid')):
    main()
```

Start/stop/reload.

A common pattern for a daemon to interact with its administrator is to provide a start/stop/reload behavior which is usually implemented as a set of command line arguments. This is particularly useful if you're planning to support initd.

DaemonContext, though, will not take care of this for you. DaemonRunner (https://pagure.io/python-daemon/blob/master/f/daemon/runner.py#_82) does have code in regard to this behavior, but I wouldn't advise you to use it directly since it is deprecated. Anyway you can still use its source code as a reference, for further details take a look at the *_start* (https://pagure.io/python-daemon/blob/master/f/daemon/runner.py#_174) and *_stop* (https://pagure.io/python-daemon/blob/master/f/daemon/runner.py#_216) methods.

Conclusions

The package python-daemon is absolutely not the only way you can create a daemon for a Python program, you should carefully consider every possibility you have.

If your choice is python-daemon, we have gone through pretty much all of the configurations of DaemonContext. I haven't covered all of them though, if you're still looking for more options you should look in the PEP; if you can't find enough information there, have a look at DaemonContext's source code (https://pagure.io/python-daemon/blob/master/f/daemon/daemon.py#_63).

References

Linux documentation on processes: http://www.tldp.org/LDP/tlk/kernel/processes.html (http://www.tldp.org/LDP/tlk/kernel/processes.html)

Windows services: https://msdn.microsoft.com/en-us/library/windows/desktop/ms685141(v=vs.85).aspx (https://msdn.microsoft.com/en-us/library/windows/desktop/ms685141(v=vs.85).aspx)

OSX
agents/daemons: https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingLaunchdJobs.html (https://developer.apple.com/library/content/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingLaunchdJobs.html)

Process forking in Unix: https://en.wikipedia.org/wiki/Fork_(system_call) (https://en.wikipedia.org/wiki/Fork_(system_call))

Core dump: https://en.wikipedia.org/wiki/Core_dump (https://en.wikipedia.org/wiki/Core_dump)

Unix umasks: https://en.wikipedia.org/wiki/Umask (https://en.wikipedia.org/wiki/Umask)

python-daemon is hosted at: https://pagure.io/python-daemon/ (https://pagure.io/python-daemon/)

python-daemon on PyPI: https://pypi.org/project/python-daemon/ (https://pypi.org/project/python-daemon/)

Begin of DaemonContext documentation: https://pagure.io/python-daemon/blob/master/f/daemon/daemon.py#_64 (https://pagure.io/python-daemon/blob/master/f/daemon/daemon.py#_64)

Wikipedia page on daemons: https://en.wikipedia.org/wiki/Daemon_(computing) (https://en.wikipedia.org/wiki/Daemon_(computing))

What a daemon is: https://www.freebsd.org/copyright/daemon.html (https://www.freebsd.org/copyright/daemon.html)

How is a daemon is different from common processes: https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/basics-processes.html (https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/basics-processes.html)

PEP 3143 on "Standard daemon process library": https://www.python.org/dev/peps/pep-3143/ (https://www.python.org/dev/peps/pep-3143/)

Death reasoning for the python-daemon library: https://www.python.org/dev/peps/pep-3143/#pep-deferral (https://www.python.org/dev/peps/pep-3143/#pep-deferral)

Doctor Says Don't Cover Up Your Dark Spots - (Try This Instead)

REPORT THIS AD

Tagged:
api,
daemon,
pep3143,
python,
python-daemon,
tutorial,
unix

10 thoughts on "A tutorial on python-daemon – or – Why doesn't python-daemon have any documentation?"

thibault ketterer says:
February 1, 2018 at 3:07 pm
you should also consider supervisor

 Reply

dp says:
February 23, 2018 at 6:15 pm
As I stated, the list truly is incomplete, but I'll add it since it is indeed quite popular

 Reply

HoudaM says:
April 1, 2018 at 7:46 pm
Thank you was very useful !

 Reply

kot says:
April 24, 2018 at 10:09 pm
Thank you!

 Reply

chris says:
August 1, 2018 at 3:53 pm
Great article, thx!

 Reply

Zheng Liu says:
October 17, 2018 at 5:10 pm
Great article. What I don't understand is that it seems the entire blog only contains this article? Did you build a blog for posting this daemon article? lol. Anyway, I really learned a lot from your article. Cheers!

 Reply

dp says:
October 17, 2018 at 6:07 pm

You're actually right Zheng this is indeed the only article I've posted here. To be fair I have been busy with my Bachelor CS course in this last year, but I've been thinking lately about writing a follow-up on this post. Would you recommend me some parts that you found missing, confusing or that would need more details?

Reply

Sebastian says:
October 27, 2018 at 1:42 pm
Your article is great, thanks for that. But, I think I figured out why there is no documentation. Pretty simple IMHO: it became obsolete with the wide support of systemd. You just don't have to care about it anymore, systemd does it all for you. Unfortunately I've only learned that after implementing it fully. 😉 Thanks anyway! Cheers!

Reply

Alex Rodenberg says:
October 31, 2018 at 9:51 am
well I need initd start stop functionality, so elaborating on that with examples would be nice.. if we can't use DaemonRunner

Reply

dp says:
October 31, 2018 at 5:14 pm
Will sure do 👍

Reply