mattdaemon Documentation

Release 1.1.1

plausibility

January 10, 2017

mattdaemon is a simple and effective way to daemonize any and all Python scripts you have.

CHAPTER 1

Documentation

1.1 Usage

1.1.1 Install & setup

What's the point in an awesome module (which this is!) if you can't use it? Nothing, that's what!

mattdaemon is on pypi, so you just have to `pip install mattdaemon`, and you're good to `import mattdaemon`.

Setup

mattdaemon is fairly simple to setup, all you have to do is subclass the *`mattdaemon.daemon`* class, and override the *`mattdaemon.daemon.run()`* method.

Here is how you'd make a simple daemon instance.

```
import mattdaemon

class MyDaemon(mattdaemon.daemon):
    def run(self, *args, **kwargs):
        # ... do something ...

# /tmp/my-daemon.pid is our PID file.
daem = MyDaemon("/tmp/my-daemon.pid")
```

1.1.2 Working with the daemon

Note: In these examples, `daem` refers to the instance of the MyDaemon class, demonstrated above.

Note: Most of these are fairly simple to grasp; there isn't a dying need to document them, it's just nice to have.

start

Use this to start and daemonize your app.

```
daem.start()
```

stop

```
if daem.status():
    daem.stop()
```

restart

Note: This literally just calls *mattdaemon.daemon.stop()* and *mattdaemon.daemon.start()*, in that order.

```
if daem.status():
    daem.restart()
```

status

This will return a `True` or `False` value based on whether or not the daemon process is currently running.

```
if daem.status():
    print 'daemon currently running!'
else:
    print 'daemon not running!'
```

run

Note: You don't actually call this; it's called by the *mattdaemon.daemon.start()* method.

Warning: If you don't add the `*args` and `**kwargs` variables to the function, all sorts of hell might break loose.

This is the method you override to get your daemon calling what you want it to call.

```
import mattdaemon

class MyDaemon(mattdaemon.daemon):
    def run(self, *args, **kwargs):
        # ...
```

1.1.3 Daemonizing your app

Note: These are just some good ideas to keep in mind when creating your app with mattdaemon.

We don't need no daemonization

Changed in version 1.1.0.

Note: Previous to `1.1.0`, you just pass `daemonize=False` to *`mattdaemon.daemon.start()`* if you don't want to daemonize. This was changed to make it easier to pass through variables.

If you'd like to debug your app, you'll have to stop the daemonization from happening temporarily. If you'd like to do that, you can simply pass `daemonize=False` through to the creation of your daemon instance, like so:

```
daem = MyDaemon("/tmp/my-daemon.pid", daemonize=False, **kw)
```

Pass through variables

New in version 1.1.0.

Note: This makes use of the argument lists you can use in Python. If you know how many args are passed in, you can simply expand the `*args` into variables.

If you want to pass through variables from the creation of your daemon to the *`mattdaemon.daemon.run()`* method, you can easily do so by passing them into *`mattdaemon.daemon.start()`*!

For example, say we want to pass some settings in from `sys.argv`:

```
#...
class MyDaemon(mattdaemon.daemon):
    def run(self, *args, **kwargs):
        for count, thing in enumerate(args):
            print '{0}. {1}'.format(count, thing)

        for k, v in kwargs.items():
            print '{0} = {1}'.format(k, v)

#...
daem.start(sys.argv[0], sys.argv[1], foo=sys.argv[2], bar=sys.argv[3])
```

If we call the script like this: `python example.py gibson tree 50`, we'll get the output:

```
$ python example.py gibson tree 50
0. example.py
1. gibson
foo = tree
bar = 50
```

There is no std*

Why would there be? Your daemon will be running in the background. If you need the user to enter information, perhaps in a loop, you'll have to look for other ways of doing that.

By default, `sys.stdin`, `sys.stdout` and `sys.stderr` are all redirected to `/dev/null`, since they're useless in a deamonized app.

What does this mean? If you want to log anything, you'll need to use a dedicated logger, or provide a log file to your daemon instance (example below).

```
kw = {
    "pidfile": "/tmp/my-daemon.pid",
    "stdin": "/dev/null", # since we don't need it
    "stdout": "/tmp/my-daemon.log",
    "stderr": "/tmp/my-daemon.log"
}
daem = MyDaemon(**kw)
# daem.start(), whathaveyou
```

Don’t assume your working dir

Due to how daemonization works, the working directory is changed to `/`, the root of the file system. Because of this, you can’t assume any required files are relative, since they might not be.

```
# Relative, bad!
with open('relative/file', 'rb') as f:
    # ...

# Absolute, good!
with open('/some/absolute/file', 'rb') as f:
    # ...
```

To root or not to root

Note: The default for the root check is `False`; as in, no, we don’t require root.

mattdaemon understands you, as well as your app. Careless users often run things with a higher privilege than they need. Do you need root to write some files to temp, or serve data on a high port? **Nope**. Do people do it anyway? They sure do!

Due to this, there is a built in root check. You can either tell the user that they **do** require root, or that they **don’t** (which, to be fair, you should be aiming for anyway). Why yes or no? If you don’t need root, you shouldn’t allow it. If you **do** need root, you should require it, since it will be needed at one point or another.

Root checks are simple, and can be controlled like such (with the `root` keyword):

```
# Yes, we need root!
daem = MyDaemon("/tmp/my-daemon.pid", root=True)
```

By default, the check also requires `--requires-root` to be in the arguments passed, so that the user acknowledges the use of root. This might not be what you want, so you can easily disable that, with the `root_chk_argv` keyword.

```
# Yes, we need root!
# No, we don't care about --requires-root
daem = MyDaemon("/tmp/my-daemon.pid", root=True, root_chk_argv=False)
```

Handling SIGTERM

Warning: This only works in POSIX compliant operating systems, since it uses signals, and Windows doesn’t.

If you do anything with resources, be it an open port, file, network request, you should handle SIGTERM.

```
import signal

def my_handler(signum, frame):
    print 'Received signal', signum, 'cleaning up resources to exit'
    my_resource.close()
    my_socket.close()
    print 'done..'

# Register the handler.
signal.signal(signal.SIGTERM, my_handler)
```

1.2 Automated Documentation

This documentation is automatically built from the latest version of chr, located at the git repo.

class `mattdaemon.`**`daemon`**(*pidfile*, *daemonize=True*, *root=False*, *root_chk_argv=True*, *stdin='/dev/null'*, *stdout='/dev/null'*, *stderr='/dev/null'*)

A generic daemon class.

Usage: subclass the Daemon class and override the run() method

`__init__`(*pidfile*, *daemonize=True*, *root=False*, *root_chk_argv=True*, *stdin='/dev/null'*, *stdout='/dev/null'*, *stderr='/dev/null'*)

Make our daemon instance. pidfile: the file we're going to store the process id in. ex: /tmp/matt-daemon.pid root: does this script require root? True if it does, False if it doesn't. Will be enforced. root_chk_argv: does the script require '–requires-root' in sys.argv to run as root? (usage is good) stdin: where the script gets stdin from. "/dev/null", "/dev/stdin", etc. stdout: where the script writes stdout. "/dev/null", "/dev/stdout", etc. stderr: where the script writes stderr. "/dev/null", "/dev/stderr", etc.

`daemonize`()

do the UNIX double-fork magic, see Stevens' "Advanced Programming in the UNIX Environment" for details (ISBN 0201563177) http://www.erlenstar.demon.co.uk/unix/faq_2.html#SEC16

`restart`()

Restart the daemon

`run`(**args*, ***kwargs*)

You should override this method when you subclass Daemon. It will be called after the process has been daemonized by start() or restart().

`start`(**args*, ***kwargs*)

Start the daemon

`status`()

Check if the daemon is currently running. Requires procfs, so it will only work on POSIX compliant OS'.

`stop`()

Stop the daemon