GPSD socket connection and decoding JSON into Python dictionaries

https://codereview.stackexchange.com/questions/120951/gpsd-socket-connection-and-decoding-json-into-python-dictionaries

---

GPS3 is a python 2.7-3.5 interface to GPSD.

I've stripped back everything to two classes.

```
#!/usr/bin/env python3
# coding=utf-8
"""
GPS3 (gps3.py) is a Python 2.7-3.5 GPSD interface
(http://www.catb.org/gpsd)
Defaults host='127.0.0.1', port=2947, gpsd_protocol='json'

GPS3 has two classes.
1) 'GPSDSocket' to create a socket connection and retreive the output from
GPSD.
2) 'Fix' unpacks the streamed gpsd data into python dictionaries.

These dictionaries are populated from the JSON data packet sent from the
GPSD.

Import           import gps3
Instantiate      gps_connection = gps3.GPSDSocket()
                 gps_fix = gps3.Fix()
 Use             print('Altitude = 'gps_fix.TPV['alt'])
                 print('Latitude = 'gps_fix.TPV['lat'])

Consult Lines 150-ff for Attribute/Key possibilities.
or http://www.catb.org/gpsd/gpsd_json.html

Run human.py; python[X] human.py [arguments] for a human experience.
"""
from __future__ import print_function

import json
import select
import socket
import sys

__author__ = 'Moe'
__copyright__ = "Copyright 2015-2016  Moe"
__license__ = "MIT"
__version__ = "0.11a"

HOST = "127.0.0.1"  # gpsd defaults
GPSD_PORT = 2947  # "
PROTOCOL = 'json'  # "
```

```
class GPSDSocket(object):
    """Establish a socket with gpsd, by which to send commands and receive
data.
    """

    def __init__(self, host=HOST, port=GPSD_PORT, gpsd_protocol=PROTOCOL,
devicepath=None):
        self.devicepath_alternate = devicepath
        self.response = None
        self.protocol = gpsd_protocol
        self.streamSock = None

        if host:
            self.connect(host, port)

    def connect(self, host, port):
        """Connect to a host on a given port.
        :param port:
        :param host:
        """
        for alotta_stuff in socket.getaddrinfo(host, port, 0,
socket.SOCK_STREAM):
            family, socktype, proto, _canonname, host_port = alotta_stuff
            try:
                self.streamSock = socket.socket(family, socktype, proto)
                self.streamSock.connect(host_port)
                self.streamSock.setblocking(False)

            except OSError as error:
                sys.stderr.write('\nGPSDSocket.connect OSError is-->',
error)
                sys.stderr.write('\nAttempt to connect to a gpsd at {0} on
port \'{1}\' failed:\n'.format(host, port))
                sys.stderr.write('Please, check your number and dial
again.\n')
                self.close()
                sys.exit(1)  # TODO: gpsd existence check and start

            finally:
                self.watch(gpsd_protocol=self.protocol)

    def watch(self, enable=True, gpsd_protocol='json', devicepath=None):
        """watch gpsd in various gpsd_protocols or devices.
        Arguments:
            self:
            enable: (bool) stream data to socket
            gpsd_protocol: (str) 'json', 'nmea', 'rare', 'raw', 'scaled',
'split24', or 'pps'
            devicepath: option for non-default device path
        Returns:
            command: (str) e.g., '?WATCH={{"enable":true,"json":true}}'
        """
        # TODO: 'timing' requires special attention, as it is undocumented
and lives with dragons
        command =
'?WATCH={{"enable":true,"{0}":true}}'.format(gpsd_protocol)

        if gpsd_protocol == 'rare':  # 1 for a channel, gpsd reports the
unprocessed NMEA or AIVDM data stream
```

```
            command = command.replace('"rare":true', '"raw":1')
        if gpsd_protocol == 'raw':  # 2 channel that processes binary data,
received data verbatim without hex-dumping.
            command = command.replace('"raw":true', '"raw",2')
        if not enable:
            command = command.replace('true', 'false')  # sets -all-
command values false .
        if devicepath:
            command = command.replace('}', ',"device":"') + devicepath +
'"}'

        return self.send(command)

    def send(self, commands):
        """Ship commands to the daemon
        :param commands:
        """
        # session.send("?POLL;")  # TODO: Figure a way to work this in.
        # The POLL command requests data from the last-seen fixes on all
active GPS devices.
        # Devices must previously have been activated by ?WATCH to be
pollable.
        if sys.version_info[0] < 3:  # Not less than 3, but 'broken
hearted' because
            self.streamSock.send(commands)  # 2.7 chokes on 'bytes' and
'encoding='
        else:
            self.streamSock.send(bytes(commands, encoding='utf-8'))  # It
craps out here when there is no gpsd running
            # TODO: Add recovery, check gpsd existence, re/start, etc..

    def __iter__(self):
        """banana"""  # <------- for scale
        return self

    def next(self, timeout=0):
        """Return empty unless new data is ready for the client.  Will sit
and wait for timeout seconds
        :param timeout:
        """
        try:
            (waitin, _waitout, _waiterror) =
select.select((self.streamSock,), (), (), timeout)
            if not waitin:
                return
            else:
                gpsd_response = self.streamSock.makefile()  # was
'.makefile(buffering=4096)' In strictly Python3
                self.response = gpsd_response.readline()
            return self.response

        except OSError as error:
            sys.stderr.write('The readline OSError in GPSDSocket.next is
this: ', error)
            return

    __next__ = next  # Workaround for changes in iterating between Python
2.7 and 3.5

    def close(self):
        """turn off stream and close socket"""
```

```
        if self.streamSock:
            self.watch(enable=False)
            self.streamSock.close()
        self.streamSock = None
        return


class Fix(object):
    """Retrieve JSON Object(s) from GPSDSocket and unpack it into
respective
    gpsd 'class' dictionaries, TPV, SKY, etc. yielding hours of fun and
entertainment.
    """

    def __init__(self):
        """Sets of potential data packages from a device through gpsd, as a
generator of class attribute dictionaries"""

        version = {"release", "proto_major", "proto_minor", "remote",
"rev"}

        tpv = {"alt", "climb", "device", "epc", "epd", "eps", "ept", "epv",
"epx", "epy", "lat", "lon", "mode", "speed", "tag", "time", "track"}

        sky = {"satellites", "gdop", "hdop", "pdop", "tdop", "vdop",
"xdop", "ydop"}

        gst = {"alt", "device", "lat", "lon", "major", "minor", "orient",
"rms", "time"}

        att = {"acc_len", "acc_x", "acc_y", "acc_z", "depth", "device",
"dip", "gyro_x", "gyro_y", "heading", "mag_len", "mag_st", "mag_x",
"mag_y", "mag_z",
               "pitch", "pitch_st", "roll", "roll_st", "temperature",
"time", "yaw", "yaw_st"}  # TODO: Check Device flags

        pps = {"device", "clock_sec", "clock_nsec", "real_sec",
"real_nsec"}

        device = {"activated", "bps", "cycle", "mincycle", "driver",
"flags", "native", "parity", "path", "stopbits", "subtype"}  # TODO: Check
Device flags

        poll = {"active", "fixes", "skyviews", "time"}

        devices = {"devices", "remote"}

        # ais = {}  # see: http://catb.org/gpsd/AIVDM.html

        error = {"message"}

        # 'repository' of dictionaries possible, and possibly 'not
applicable'
        packages = {"VERSION": version,
                    "TPV": tpv,
                    "SKY": sky, "GST": gst, "ATT": att, "PPS": pps,
                    "DEVICE": device, "POLL": poll,
                    "DEVICES": devices,
                    "ERROR": error}  # etc.
        # TODO: Create the full suite of possible JSON objects and a better
way for deal with subsets
```

```
        for package_name, datalist in packages.items():
            _emptydict = {key: 'n/a' for (key) in datalist}  # There is a
case for using None instead of 'n/a'
            setattr(self, package_name, _emptydict)
        self.SKY['satellites'] = [{'PRN': 'n/a', 'ss': 'n/a', 'el': 'n/a',
'az': 'n/a', 'used': 'n/a'}]
        self.DEVICES['devices'] = [{"class": 'n/a', "path": 'n/a',
"activated": 'n/a', "flags": 'n/a', "driver": 'n/a',
                                    "native": 'n/a', "bps": 'n/a',
"parity": 'n/a', "stopbits": 'n/a', "cycle": 'n/a'}]

    def refresh(self, gpsd_data_package):
        """Sets new socket data as Fix attributes
        Arguments:
            self (class):
            gpsd_data_package (json object):
        Returns:
        self attribute dictionaries, e.g., self.TPV['lat']
        Raises:
        AttributeError: 'str' object has no attribute 'keys' when the
device falls out of the system
        ValueError, KeyError: stray data, should not happen
        """
        try:
            fresh_data = json.loads(gpsd_data_package)  # The reserved word
'class' is popped from JSON object class
            package_name = fresh_data.pop('class', 'ERROR')  # gpsd data
package errors are also 'ERROR'.
            package = getattr(self, package_name, package_name)  # packages
are named for JSON object class
            for key in package.keys():  # TODO: Rollover and retry.  It
fails here when device disappears
                package[key] = fresh_data.get(key, 'n/a')  # Updates and
restores 'n/a' if key is absent in the socket
                # response, present --> "key: 'n/a'" instead.'
        except AttributeError:  # 'str' object has no attribute 'keys'
TODO: if returning 'None' is a good idea
            print("No Data")
            return None

        except (ValueError, KeyError) as error:
            sys.stderr.write(str(error))  # Look for extra data in stream
            return None


if __name__ == '__main__':
    print('\n', __doc__)

#
# Someday a cleaner Python interface will live here
#
# End
```

While 'refreshing' the data from the GPSD socket read, the JSON object is loaded into a JSON decoder module.

This fresh data output has 'class' popped and it's value becomes an attribute of the instance.

Remaining data goes into a dictionary with the new values, such as `gps_fix.TPV['lat'] = -33.123456789`.

If data is missing from the socket, key or value, persistently or sporadically, the key has its value replaced with 'n/a', the initialised value.

In general looks good and well documented.

- In the `close` method the `return` statement is unnecessary.
- Quotes are inconsistently used.
- Most of the `:param` annotations in the docstrings are unused. If you're not going to document them, just leave them out. The `watch` method is also not using the syntax at all, where it would make a lot of sense to use it.
- The `finally` block in `connect` seems weird. If I'm not mistaken it *will* be executed even if `sys.exit` is called (since that's implemented using a `SystemExit` exception) - is that intentional? I'd put a comment on it if so.
- Also, is the `watch` method intended to be called from outside the class? If not, then the default arguments are moot. Possibly also prefix it to avoid calling it from outside the class.
- In `next` the `else` block can be put inline as the `if` already `return`s from the method. Again, the `return` in the `except` handler is not necessary.
- Also, `return None` is the same as `return`, but I imagine that's done for clarity.
- In the `_emptydict` creation, the parens around `key` aren't needed:

  ```
  _emptydict = {key: 'n/a' for key in datalist}
  ```

  If possible I'd use the same construction for `SKY` and `DEVICES` btw.

- The documentation for `refresh` is wrong, there's nothing returned from that method (well `None`, but that doesn't count).