```
"""
classes.py
Defines three classes:
    Router, which holds information about the router
    RipEntry, which hold the data for each individual rip entry
    RipPacket, which holds the entire packet (including the rip entries)
"""
import datetime
from struct import *
from prettytable import PrettyTable


class Router:
    """
    Router Class - This is the main router object and contains the configuration data
    and manages the router's Routing Table(RT)
    """

    def __init__(self, rid, inputs, outputs):
        """
        Initialisation of the Router class which holds all information regarding routing tables, timeouts and more
        Router Constructor
        :param rid: Router object ID
        :param inputs: List of the input ports of the router
        :param outputs: List containing input ports of peer routers, metric and the peer router id
        """
        self.router_id, self.next_hop = int(rid), int(rid)
        self.input_ports = inputs
        self.output_ports = outputs
        self.metric = 0
        self.garbage = False
        self.timeout, self.route_timeout = None, None
        self.connected_networks = {}
        self.routing_table = {}  # Dictionary with key as RID, metric and next-hop, and deletion flag
        self.routing_table = {self.router_id: [self.metric, self.next_hop, self.garbage, self.timeout]}
        for entry in self.output_ports:
            self.connected_networks.update({entry[2]: [entry[0], entry[1]]})

    def print_routing_table(self):
        """
        Prints the router's current routing table using the PrettyTable formatter
        """
        table = PrettyTable(['Router ID', 'Metric', 'Next Hop', 'Garbage', 'Timeout'])
        for key, value in self.routing_table.items():
            now = datetime.datetime.now().timestamp()
            table.add_row([key, value[0], value[1], value[2],
                           int(self.route_timeout - (now - value[3])) if value[3] is not None else None])
        print("+-------------------------------------------------------+")
        print("|                    Router ID: {0}                     |".format(self.router_id))
        print(table)

    def print_router_config(self):
        """
        Prints the router configuration data if a table form
        """
        line_count = 40
        line_special = "+" + ("-" * line_count) + "+"
        table = PrettyTable(["Parameter", "Value"])
        table.add_row(["Router ID", self.router_id])
        table.add_row(["Input Ports", ", ".join(map(str, self.input_ports))])
        title = "|          ROUTER CONFIGURATION          |\n"
        output_table = PrettyTable(['Output Port', 'Metric', 'Peer ID'])
        for value in self.output_ports:
            output_table.add_row([value[0], value[1], value[2]])
        print(line_special + '\n' + title + line_special)
        print(table)
        print(output_table)


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


# Sub-Class
class RipEntry:
    """
    Class to deal with packing and unpacking RIP Entries into the RIP Packet
    Contains functions to create, pack (into byte stream), unpack (from byte stream), printing and indexing
    """
```

```
    def __init__(self, input_bytes=None):
        """
        Initialisation of our RipEntry class can receive a byte stream to convert into an object or raw input
        RipEntry is a sub-class of RipPacket and is used within the creation of RipPackets
        :param input_bytes: A byte stream to be converted from raw

        [Address Family ID (2 Bytes)] [Zero (2 Bytes)]
        [           Router ID (4 Bytes)              ]
        [           Zero (4 Bytes)                   ]
        [           Next Hop (4 Bytes)               ]
        [           Metric (4 Bytes)                 ]
        """
        if input_bytes:
            self.unpack_entry(input_bytes)
        else:
            self.address_family_id = None
            self.rid = None
            self.next_hop = None
            self.metric = None
        self.padding_1 = 0
        self.padding_2 = 0

    def __str__(self):
        """
        Prints the contents of the RipEntry class
        :return string: Representation of the RipEntry in list form
        """
        return "[{0}, {1}, {2}, {3}]".format(self.address_family_id, self.rid, self.next_hop, self.metric)

    def print_entry(self):
        """
        Prints the representation of the RipEntry class
        :return repr: Returns a representation of the RipEntry
        """
        return "\tAddress ID: {0}\n" \
               "\tZero: {1}\n" \
               "\tRouter ID: {2}\n" \
               "\tZero: {3}\n" \
               "\tNext Hop: {4}\n" \
               "\tMetric: {5}".format(self.address_family_id, self.padding_1, self.rid, self.padding_2,
                                      self.next_hop, self.metric)

    def __len__(self):
        """
        Redefines the length command for the RipEntry class
        :return length: Returns the length of the Entry packet
        """
        return len(self.entry_payload())

    def __getitem__(self, index):
        """
        Redefines the __getitem__ function to allow for indexing
        :param index: Index of item in 'self'
        :return items[index]: Returns an element of the 'self' class at given index
        """
        items = [self.address_family_id, self.rid, self.next_hop, self.metric]
        return items[index]

    def create_entry(self, data):
        """
        Updates/creates the RipEntry class fields when new information is present
        :param data: An input list of size 4 to be converted to the object
        """
        self.address_family_id, self.rid, self.next_hop, self.metric = data[0], data[1], data[2], data[3]

    def unpack_entry(self, byte_string):
        """
        Updates the RipEntry class fields when new information is present
        :param byte_string: Unpacks a byte string and converts it to a RipEntry object
        """
        data = unpack(">hhiiii", byte_string)
        self.address_family_id, self.rid, self.next_hop, self.metric = data[0], data[2], data[4], data[5]

    def entry_payload(self):
        """
        Converts the RipEntry class object into a byte stream
        :return byte_stream: Returns a byte stream made from the RipEntry object
```

```
        """
        return pack(">hhiiii", self.address_family_id, self.padding_1, self.rid, self.padding_2, self.next_hop,
                    self.metric)


# Super-Class
class RipPacket(RipEntry):
    """
    Class to deal with packing and unpacking RIP Packets
    Contains functions to create, pack (into byte stream), unpack (from byte stream),
    checking, printing and indexing
    """

    def __init__(self, input_bytes=None):
        """
        Initialisation of our RipPacket class can receive a byte stream to convert into an object or raw input
        RipPacket is a super-class of RipEntry which is used when creating packets
        :param input_bytes: A byte stream to be converted from raw

        [Command (1 Byte)] [Version (1 Byte)] [Router ID (2 Bytes)]
        [                  RIP Entry (20 Bytes)                   ]
        """
        super().__init__()  # Superclass / Parent of RipEntry
        self.drop_packet = False
        self.rip_entries = []
        if input_bytes:
            self.unpack_packet(input_bytes)
        else:
            self.command = None
            self.version = None
            self.router_id = None

    def __str__(self):
        """Prints the contents of the RipPacket class
        :return repr: Representation of the RipPacket
        """
        return "\nCommand: {0}\n" \
               "Version: {1}\n" \
               "Router ID: {2}\n" \
               "RIP Entries: {3}".format(self.command, self.version, self.router_id, self.str_rip_entries())

    def str_rip_entries(self):
        """
        Prints the representation of all the RipEntries
        :return string: Returns a representation of each RipEntry contained in the packet
        """
        string = ""
        for item in self.rip_entries:
            string += "\n" + item.print_entry() + "\n"
        return string

    def __len__(self):
        """
        Redefines the length command for the RipPacket class
        :return length: Returns the length of the entire RipPacket
        """
        return len(self.pack_packet())

    def __getitem__(self, index):
        """
        Redefines the __getitem__ function to allow for indexing
        :param index: Index of item in 'self'
        :return items[index]: Returns indexed RipEntry from the list of RipEntries contained in 'self'
        """
        return self.rip_entries[index]

    def create_packet(self, command, version, router_id, entry_table):
        """
        Creates a packet from the RipPacket class, including all the RipEntries also contained within 'self'
        :param command: Command field to update
        :param version: Version field to update
        :param router_id: Router ID field to update
        :param entry_table: List of RipEntries to update
        """
        if None in (command, version, router_id) or None in entry_table:
            self.drop_packet = True
            print("create_packet has received 'None' arguments.\n"
                  "Dropping packet.")
```

```
        self.command, self.version, self.router_id = command, version, router_id
        for item in entry_table:
            entry = RipEntry()
            entry.create_entry(item)
            self.rip_entries.append(entry)

    def pack_packet(self):
        """
        Creates a byte stream from the RipPacket class, including all the RipEntries also contained within 'self'
        :return packet: A complete representation of the RipPacket's byte stream
        """
        packet = pack(">bbh", self.command, self.version, self.router_id)
        for item in self.rip_entries:
            packet += item.entry_payload()
        return packet

    def unpack_packet(self, input_bytes):
        """
        Updates the RipPacket object fields from a stream of bytes
        :param input_bytes: Unpacks a byte string and converts it to a RipPacket object
        """
        count, packet_len = 4, len(input_bytes)
        # Check that entry packets are 20 bytes each
        if (packet_len - 4) % 20 != 0:
            self.drop_packet = True
            print("One of the entry packets are not of size 20 bytes.\n"
                  "Dropping packet.")
        else:
            self.command, self.version, self.router_id = unpack(">bbh", input_bytes[:4])
            # For each 20 bytes, split into a list of RipEntries
            while count < packet_len:
                entry = RipEntry(input_bytes[count:count + 20])
                self.rip_entries.append(entry)
                count += 20
            if self.check_packet() is not None:
                self.drop_packet = True
                print(self.check_packet())

    def check_packet(self):
        """
        Checks the converted message for errors when converting into the RipPacket class
        Checks for None fields, header length, command fields, version fields
        and RipEntries (address_family_id & metrics)
        :return string: Returns a string representation of th error that occurred if failing any checks
        """
        # Checking within the RipPacket Header
        if None in [self.command, self.version, self.router_id]:
            return "Error creating packet 'None' fields exist.\n" \
                   "Dropping packet."
        if self.__len__() < 4:
            return "RIP header is an incorrect size.\n" \
                   "Dropping packet."
        elif 2 < self.command < 0:  # 1 -> Request, 2 -> Response
            return "Command field ({0}) not within 1 - 2.\n" \
                   "Dropping packet.".format(self.command)
        elif self.version != 2:  # 2 -> RIP Format
            return "Version field ({0}) does not equal 2.\n" \
                   "Dropping packet.".format(self.version)
        # Checking all RipEntries
        if len(self.rip_entries) > 0:
            for item in self.rip_entries:
                if item.address_family_id != 2:
                    return "RipEntry version field ({0}) does not equal 2.\n" \
                           "Dropping packet.".format(item.address_family_id)
                elif item.metric < 0:
                    return "Metric field ({0}) is less than 0.\n" \
                           "Dropping packet.".format(item.metric)
        return None

# ---------------------------------------------------------------------- #
```

```
"""
config_parse.py
This file deals with unpacking a .txt file and creating routers based on the file arguments
"""
import sys
from classes import *


def open_config():
    """
    Opens config file specified in the command line arguments and reads each line
    :return config_parser: String containing all lines in the config file
    :except FileNotFoundError: Raises file not found error if the parser cannot locate the given file
    """
    if len(sys.argv) != 2:
        print("USAGE: config_parse.py <input-file>")
        sys.exit()
    try:
        with open(sys.argv[1]) as config_file:
            config_data = config_file.read().splitlines()
    except FileNotFoundError:
        print("Error! Config file not found.")
        sys.exit()
    return config_parser(config_data)


def config_parser(file_data):
    """
    Parses information from the data extracted from the config file and returns
    a router Object constructed with the parsed data
    :return Router: Constructed Router object
    """
    inputs, outputs = [], []
    rid = -1
    config_params = ["router-id", "input-ports", "outputs"]
    for lines in file_data:
        if lines.strip():  # Remove empty lines in config file
            lines = lines.split(" ")
            if lines[0] == config_params[0]:
                rid = lines[1]
            elif lines[0] == config_params[1]:
                for each in lines[1:]:
                    inputs.append(int(each.strip(",")))
            elif lines[0] == config_params[2]:
                for each in lines[1:]:
                    outputs.append([int(i) for i in each.strip(",").split("-")])
            else:  # Ignore comments and other artifacts in file
                pass
        else:
            pass
    return Router(rid, inputs, outputs)


def check_config(router):
    """
    Checks Router configuration for integrity, as per assignment specification
    :param router: This is the main router object which contains the configuration and routing tables
    :except ValueError: Raises value error if duplicate ports are found in the config
    """
    for in_port in router.input_ports:
        if router.input_ports.count(in_port) > 1:
            raise ValueError("Found duplicate port ({0}) in the routers inputs.".format(in_port))
        elif in_port not in range(1024, 64000):
            raise ValueError("Input port {0} not in range of 1024 - 64000".format(in_port))
    out_ports = [port[0] for port in router.output_ports]
    for port in out_ports:
        if out_ports.count(port) > 1:
            raise ValueError("Found duplicate port ({0}) in the routers outputs.".format(port))
        if port not in range(1024, 64000):
            raise ValueError("Output port {0} not in range of 1024 - 64000".format(port))
```

```python
#!/usr/bin/env python3
"""
controller.py
This file will deal with threading operations and act as a core controller for the router
"""
import os
import threading
import config_parse
import receive_controller
import send_controller
import timers

DEBUG = False
DIVISOR = 12  # Used for testing to speed up network convergence
TIMER = 30 // DIVISOR
TIMEOUT = 180 // DIVISOR
DELETE_TIMER = 120 // DIVISOR
CLI_REFRESH_RATE = 1

# Main function for which the program initially starts from
if __name__ == '__main__':
    os.system('cls' if os.name == 'nt' else 'clear')
    router = config_parse.open_config()
    config_parse.check_config(router)
    router.route_timeout = TIMEOUT
    router.print_router_config()
    router.print_routing_table()
    receive = threading.Thread(target=receive_controller.receive_packet, args=(router, TIMEOUT))
    periodic_send = threading.Thread(target=send_controller.periodic_send_control, args=(router, TIMER))
    timeout_timer = threading.Thread(target=timers.timeout, args=(router, TIMEOUT))
    cli_update = threading.Thread(target=timers.cli_refresh, args=(router, CLI_REFRESH_RATE))
    cli_update.start()
    periodic_send.start()
    receive.start()
    timeout_timer.start()
```

```
"""
receive_controller.py
Deals with the receiving functionality of the router
"""
import select
import socket
import controller
import send_controller
import timers
from classes import *


def receive_packet(router, timeout_len):
    """
    Function that puts the router into a continuous search for incoming packets
    :param router: This is the main router object which contains the configuration and routing tables
    :param timeout_len: Length of time it takes for a route to timeout as specified in the main controller
    :except OSError: Raises an OS error if ports could not be bound
    """
    server_ip = "127.0.0.1"
    sock_list = [socket.socket(socket.AF_INET, socket.SOCK_DGRAM) for _ in router.input_ports]
    try:
        counter = 0
        for sock in sock_list:
            sock.bind((server_ip, router.input_ports[counter]))
            sock.setblocking(False)
            counter += 1
    except OSError:
        print("Could not open and/or bind specified server ports.\n---Exiting---")
    print("Sockets connected.\nListening on ports {0} for incoming transmissions.\n...".format(router.input_ports))
    while True:
        readable, writable, exceptional = select.select(sock_list, [], [])
        for connection in readable:
            accept_connection(connection, router)


def accept_connection(connection, router):
    """
    Receives the packet from the socket and calls the packet verification function
    If verification fails, an error is thrown
    :param connection: Ports which the router is listening on
    :param router: This is the main router object which contains the configuration and routing tables
    """
    data, address = connection.recvfrom(1024)
    received_packet = RipPacket()
    received_packet.unpack_packet(data)
    received_packet.check_packet()
    if not received_packet.drop_packet:
        update_routing_table(router, received_packet)


def update_routing_table(router, received_packet):
    """
    Updates Router routing table entries with that of received packet
    :param received_packet: Received byte stream to convert
    :param router: This is the main router object which contains the configuration and routing tables
    """
    trigger_send = False
    working_dict = dict(router.routing_table)  # Thread Safety
    received_rid = received_packet.router_id
    for (_, destination_rid, next_hop, metric) in received_packet:
        timestamp = datetime.datetime.now().timestamp()
        new_metric = min((metric + router.connected_networks.get(received_rid)[1]), 16)
        existing_metric = 16 if working_dict.get(destination_rid) is None else working_dict.get(destination_rid)[0]
        # If destination_rid not in router.connected_networks
        if destination_rid not in working_dict:
            if new_metric < 16:
                working_dict.update({destination_rid: [new_metric, received_rid, False, timestamp]})
        elif (working_dict.get(destination_rid)[1] == received_rid or
              new_metric < existing_metric):
            if new_metric <= existing_metric and new_metric < 16:
                working_dict.update({destination_rid: [new_metric, received_rid, False, timestamp]})
            elif new_metric == 16:
                prev_time = working_dict.get(destination_rid)[3]
                working_dict.update({destination_rid: [16, received_rid, True, prev_time]})
                trigger_send = True
    router.routing_table = dict(working_dict)  # Copy dictionary back into place
    if trigger_send:
```

```
        send_controller.triggered_send_control(router)
        timers.garbage_collection(router, controller.DELETE_TIMER)
```

```
"""
send_controller.py
Deals with the sending functionality of the router
"""
import socket
import time
from random import randint
from classes import *

PAUSE = False  # Pause Send Thread
HOST = socket.gethostbyname("localhost")
COMMAND = 1
VERSION = 2


def create_routing_table(router, port):
    """
    Copies dictionary to make thread-safe
    :param router: This is the main router object which contains the configuration and routing tables
    :param port: Port which the router should send the table to
    :return route_list: Routing table for the routers to build a route database of
    """
    neighbour_router_id = None
    route_list = []
    for key, value in router.connected_networks.items():
        if value[0] == port:
            neighbour_router_id = key
    address_family_id = 2
    working_dict = dict(router.routing_table)
    for peerRouterId, route in working_dict.items():
        # If the route in the table has a next hop of the router we are sending to, don't add it to table to send
        if route[1] != neighbour_router_id:  # Split Horizon
            route_list.append([address_family_id, peerRouterId, route[1], route[0]])
    return route_list


def periodic_send_control(router, base_period):
    """
    Function to send packets periodically except when the global PAUSE variable is True
    (This is to prevent ports clashing)
    Function loops after waiting for the specified time period
    :param router: This is the main router object which contains the configuration and routing tables
    :param base_period: The time between periodic sends as specified by the main controller
    """
    time.sleep(randint(0, 4))
    while True:
        if not PAUSE:
            for address in router.output_ports:
                packet = RipPacket()
                packet.create_packet(COMMAND, VERSION, int(router.router_id),
                                     create_routing_table(router, address[0]))
                packet = packet.pack_packet()
                send_packet(packet, (HOST, address[0]))
        time.sleep(base_period)


def triggered_send_control(router):
    """
    Function to send packets when triggered by the receive controller or the garbage collector
    Pauses the periodic send thread while in operation
    :param router: This is the main router object which contains the configuration and routing tables
    :except "ErrorType": FIXME: Error type?
    """
    try:
        PAUSE = True
        for address in router.output_ports:
            packet = RipPacket()
            packet.create_packet(COMMAND, VERSION, int(router.router_id), create_routing_table(router, address[0]))
            packet = packet.pack_packet()
            send_packet(packet, (HOST, address[0]))
    except Exception:  # FIXME: Proper Exception type?
        pass
    PAUSE = False


def send_packet(packet, address):
    """
    Function that sends the packet the the ports specified in the config file
```

```
    Raises an exception on error
    :param packet: The byte stream to be sent
    :param address: A tuple of the IP address to send to and the corresponding port
    :except ConnectionRefusedError: Raises a connection error if the router is unable to establish the ports
    :except socket.error: Raises a socket error if the router is unable to send the packet
    """
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        sock.connect(address)
    except ConnectionRefusedError:
        raise ConnectionRefusedError("[!] ERROR: Unable to establish sockets")
    try:
        sock.sendto(packet, address)
        sock.close()
    except socket.error:
        print("[!] ERROR: Unable to send request")
```

```
"""
timers.py
This file just defines a timeout period used by the routers for timeouts and other timed events
"""
import datetime
import os
import threading
import time
import controller
import send_controller


def timeout(router, timeout_len):
    """
    Handles timeout thread and triggers an update and garbage collection if a link times out
    :param router: This is the main router object which contains the configuration and routing tables
    :param timeout_len: Length of time it takes for a route to timeout as specified in the main controller
    """
    while True:
        trigger_send = False
        now = datetime.datetime.now().timestamp()
        key_list = []
        working_dict = dict(router.routing_table)
        for peerId, (_, _, garbage, timeout) in working_dict.items():
            if peerId != router.router_id:  # Prevent deleting self from table.
                if int(now - timeout) >= timeout_len:
                    garbage = True
                    key_list.append(peerId)
                    trigger_send = True
        for each in key_list:
            working_dict[each][2] = True
            working_dict[each][0] = 16
        router.routing_table = dict(working_dict)  # Copy dictionary back into place
        if trigger_send:
            send_controller.triggered_send_control(router)
            garbage_thread = threading.Thread(target=garbage_collection, args=(router, controller.DELETE_TIMER))
            garbage_thread.start()
        time.sleep(1)


def cli_refresh(router, refresh):
    """
    Refreshes the display every second as long as the Global DEBUG is not set
    :param router: This is the main router object which contains the configuration and routing tables
    :param refresh: Length of time before the display refreshes as specified in the main controller
    """
    while True:
        time.sleep(refresh)
        if not controller.DEBUG:
            os.system('cls' if os.name == 'nt' else 'clear')
        router.print_routing_table()


def garbage_collection(router, period):
    """
    Cleans the routing table of expired routes after the timer has expired
    :param router: This is the main router object which contains the configuration and routing tables
    :param period: Length of time after timeout until the garbage collection starts as specified
    in the main controller
    """
    time.sleep(period)
    key_list = []
    working_dict = dict(router.routing_table)
    for key, value in working_dict.items():
        if key != router.router_id and value[2]:
            key_list.append(key)
    for key in key_list:
        del working_dict[key]
    router.routing_table = dict(working_dict)  # Copy dictionary back into place
```