```python
"""
COSC364 2022-S1 Assignment: RIP routing
Authors: MENG ZHANG (71682325), ZHENG CHAO (21671773)
File: rip_main.py
"""


# Import Modules
import sys
import time
import threading
from rip_init import rip_router_init


# Program Entry Point
if __name__ == "__main__":
    print("Starts RIP Daemon...")
    # get config file name
    try:
        if len(sys.argv) != 2:
            raise ValueError("Invalid argument for rip_main\n" +
                             "Rip router requires ONE config file")
        config_file_name = sys.argv[1]
    except ValueError as error:
        print(error)

    # Initialise a new Router object
    ROUTER = rip_router_init(config_file_name)

    # First advertise ROUTER itself immediately
    ROUTER.advertise_routes('all')
    ROUTER.print_routing_table()
    ROUTER.random_offset_period()

    # Start loop
    while True:
        ROUTER.receive_routes()
        ROUTER.advertise_all_routes_periodically()
        ROUTER.check_timeout_entries_periodically()
```

```python
"""
COSC364 2022-S1 Assignment: RIP routing
Authors: MENG ZHANG (71682325), ZHENG CHAO (21671773)
File: rip_init.py
"""


# Import Modules
from IO_parser import router_config
from rip_router import Router


# Initialise router
def rip_router_init(config_file_name):
    """
    Parameter:
    config_file_name: the name of the config file to initialise a new
    router object.

    Return: a new Router object
    """
    config = router_config(config_file_name)
    router = Router(config['router_id'],
                    config['input_ports'],
                    config['output_ports_metric_id'],
                    config['period'],
                    config['timeout'])
    print(f"Created Router {router.get_router_id()}")
    return router
```

```python
"""
COSC364 2022-S1 Assignment: RIP routing
Authors: MENG ZHANG (71682325), ZHENG CHAO (21671773)
File: IO_parser.py
"""

def router_config(file_name):
    """
    Parameter:
    file_name: string
    file format:
    i.e.
    ------------------------------
    router-id 2
    input-ports 6020, 6021
    output-ports 6010-1-1, 6030-2-3
    period 3
    timeout 18
    ------------------------------

    Return: config_data
    a dictionary with 4 keys of router_id, input_ports, output_ports,
    timers
    i.e. {'router_id': 2, 'input_ports': [6020, 6021],
    'output_ports_metric_id': {6010: {'metric': 1, 'router_id': 1},
                               6020: {...}}, 'period': 3, 'timeout': 18}
    """
    raw_config = read_config(file_name)
    config_data = parse_config(raw_config)
    return config_data


def read_config(file_name):
    """
    Parameter:
    file_name: string.
    file format:
    i.e.
    ------------------------------
    router-id 2
    input-ports 6020, 6021
    output-ports 6010-1-1, 6030-2-3
    period 3
    timeout 18
    ------------------------------

    Return: a list of strings with 4 elements.
    i.e. ['router-id 2', 'input-ports 6020, 6021', 'output-ports 6010-1-1,
          6030-2-3', 'period 3', 'timeout 18']
    """
    try:
        with open(file_name) as config_file:
            raw_config = config_file.read().splitlines()
            return raw_config
    except FileNotFoundError:
        print("Error: the config file name is invalid")


def parse_config(raw_config):
    """
    Parameter:
    raw_config: a list of strings with 4 elements.
    i.e. ['router-id 2', 'input-ports 6020, 6021', 'output-ports 6010-1-1,
          6030-2-3', 'period 3', 'timeout 18']

    Return: config_data
    a dictionary with 4 keys of router_id, input_ports, output_ports,
    timers
    i.e. {'router_id': 2, 'input_ports': [6020, 6021],
          'output_ports_metric_id': {6010: {'metric': 1, 'router_id': 1},
                                     6020: {...}}, 'period': 3, 'timeout': 18}
    """
    try:
        # get router id
        router_id = parse_id(raw_config[0])
        # get input ports
        input_ports = parse_input_ports(raw_config[1])
        # check if input ports contains duplicate ports
        if contains_duplicates(input_ports):
            raise ValueError("The input ports contains duplicate ports")
        # get output ports
        output_ports, output_ports_metric_id = parse_output_ports(raw_config[2])
        # check if input ports and output ports contain duplicate ports
        if duplicate_lists(input_ports, output_ports):
            raise ValueError("The input ports and output ports contain duplicate ports")
        # get period
        period = parse_period(raw_config[3])
        # get timeout
        timeout = parse_timeout(raw_config[4])
        # check timeout vs period ratio
        if not is_valid_timer_ratio(period, timeout):
            raise ValueError("The ratio timeout vs period should be 6")
        # create coinfig_data dictionary
        config_data = {"router_id": router_id, "input_ports": input_ports,
                       "output_ports_metric_id": output_ports_metric_id,
                       "period": period, "timeout": timeout}
        return config_data
    except IndexError as ie:
        print(ie)
        print("Some value of the config file is not available")
    except ValueError as ve:
        print(ve)
        print("Some value of the config file is invalid")


def parse_id(raw_id):
    """
    Parameter:
    raw_id: a string
    i.e. 'router-id 2'

    Return: router_id
    an interger between 1 and 64000 i.e. 1
    """
    try:
        router_id = int(raw_id.split()[1])
        if (router_id < 1 or router_id > 64000):
            raise ValueError("Router ID value is out of bounds")
        return router_id
    except IndexError as e:
        print(e)
        print("The config router ID value is not available")
    except ValueError as e:
        print(e)
        print("The config router ID value must be an integer between 1 and 64000")


def parse_input_ports(raw_input_ports):
    """
    Parameter:
    raw_input_ports: a string
    i.e 'input-ports 6020, 6021'

    Return: input_ports
    a list of integers which are between 1024 and 64000
    i.e. [6020, 6021]
    """
    try:
        input_ports_temp = raw_input_ports.split()[1:]
        input_ports = []
        for port_str in input_ports_temp:
            port_int = int(port_str.strip(','))
            if (port_int < 1024 or port_int > 64000):
                raise ValueError("Input port value is out of bounds")
            input_ports.append(port_int)
        return input_ports
    except IndexError as e:
        print(e)
        print("The config input port value is not available")
    except ValueError as e:
        print(e)
        print("The config input port value must be an integer between 1024 and 64000")


def parse_output_ports(raw_output_ports):
    """
    Parameter:
    raw_input_ports: a string
    i.e 'output-ports 6010-1-1, 6030-2-3'

    Return: output_ports, output_ports_metric_id
    output_ports: a list of integers which are between 1024 and 64000
    i.e. [6010, 6030]
    output_ports_metric_id: a dict of dicts in which key is port number
    and each sub dict contains key(port)'s metric and id.
    Metric > 0, 1 <= ID <= 64000
    i.e. {6010: {'metric': 1, 'router_id': 1}, 6020: {...}}
    """
    try:
        output_ports_combo_temp = raw_output_ports.split()[1:]
        output_ports = []
        output_ports_metric_id = {}
        for port_combo_str in output_ports_combo_temp:
            port_combo_temp = port_combo_str.strip(',').split('-')
            port_int = int(port_combo_temp[0])
            metric_int = int(port_combo_temp[1])
            id_int = int(port_combo_temp[2])
            if (port_int < 1024 or port_int > 64000):
                raise ValueError("Ouput port value is out of bounds")
            if metric_int < 1:
                raise ValueError("Output port metric is out of bounds")
            if id_int < 1 or id_int > 64000:
                raise ValueError("Output id is out of bounds")
            output_ports.append(port_int)
            # output_ports_metric_id.append([port_int, metric_int, id_int])
            output_ports_metric_id[port_int] = {'metric': metric_int,
                                                'router_id': id_int}
        return output_ports, output_ports_metric_id
    except IndexError as e:
        print(e)
        print("The config output port value is not available")
    except ValueError as e:
        print(e)
        print("The config output ports must be fomatted as port-metric-id")
        print("The config output port value must be an integer between 1024 and 64000")
        print("The config output port metric must be an integer greater than 0")
        print("The config output port id must be an integer between 1 and 64000")

```

```python

def parse_period(raw_period):
    """
    Parameter:
    raw_period: a string
    i.e. 'period 3'

    Return: period
    period: a positive integer
    i.e. 3
    """
    try:
        period = int(raw_period.split()[1])
        if period < 1:
            raise ValueError("Router period value is out of bounds")
        return period
    except IndexError as e:
        print(e)
        print("The config router period value is not available")
    except ValueError as e:
        print(e)
        print("The config router timeout value must be a positive integer")

def parse_timeout(raw_timeout):
    """
    Parameter:
    raw_timeout: a string
    i.e. 'timeout 18'

    Return: timeout
    timeout: a positive integer
    i.e. 18
    """
    try:
        timeout = int(raw_timeout.split()[1])
        if timeout < 1:
            raise ValueError("Router timeout value is out of bounds")
        return timeout
    except IndexError as e:
        print(e)
        print("The config router timeout value is not available")
    except ValueError as e:
        print(e)
        print("The config router timeout value must be a positive integer")


def contains_duplicates(lst):
    """
    Parameter:
    lst: a list

    Return: boolean
    if the lst contains duplicates, return true, otherwise false
    """
    return len(set(lst)) != len(lst)

def duplicate_lists(lst1, lst2):
    """
    Parameters:
    lst1: a list
    lst2: a list

    Return: boolean
    if the two lists contains duplicate items, return true, otherwise false
    """
    return len(set(lst1).union(set(lst2))) != len(lst1) + len(lst2)

```

```python

def is_valid_timer_ratio(period, timeout):
    """
    Parameters:
    period: a positive integer
    period: a positive integer

    Return: boolean
    if timeout / period = 6, return true, otherwise false
    """
    return timeout / period == 6
```

```python
"""
COSC364 2022-S1 Assignment: RIP routing
Authors: MENG ZHANG (71682325), ZHENG CHAO (21671773)
File: rip_router.py
"""

# Import Modules
import time
import random
from datetime import datetime
from network_interface import Interface
from forwarding_route import Route
from rip_packet import RipPacket, RipEntry
from IO_formatter import routing_table_formatter


# Router Class
class Router:
    """
    An object that simulates a router with rip protocol
    """

    # Class attributes
    INFINITY = 16
    REGULAR_TIMER_OFFSET = 1.0

    def __init__(self, router_id,
                 inputs, outputs,
                 period, timeout):
        """
        the __* attributes are private attributes which can only be
        accessed by getter outside of class.

        Parameters:
        router_id: an integer, i.e. 1, 2, 3, etc
        inputs: a list of integers, i.e. [5001, 5002, 5003]
        outputs: a dictionary of dictionaries, i.e.
        {6010(port): {'metric': 1, 'router_id': 1},
         6030(port): {'metric': 2, 'router_id': 3},
             ... : {...}
        }
        period: an integer
        timout: an integer
        """
        # Instance attributes
        self.__router_id = router_id
        self.__split_horizon_poison_reverse = True
        self.__input_ports = inputs
        self.__output_ports = outputs
        self.__regular_advertise_timer = time.time()
        self.__default_period = period
        self.__period = period
        self.__trigger_advertise_timer = time.time()
        self.__default_triggered_updates_period = period / 6
        self.__triggered_updates_period = period / 6
        self.__timeout_check_timer = time.time()
        self.__timeout = timeout
        self.__garbage_collection_time = period * 4
        self.__interface = None
        self.__routing_table = {}
        # Initialisation
        self.init_interface(inputs)
        self.init_routing_table()
        self.random_offset_period()


    def get_router_id(self):
        """
        router_id getter
        """
        return self.__router_id


    def set_router_id(self, new_id):
        """
        router_id setter
        """
        self.__router_id = new_id


    def get_input_ports(self):
        """
        router input_ports getter
        """
        return self.__input_ports


    def set_input_ports(self, new_inputs):
        """
        router input_ports setter
        """
        self.__input_ports = new_inputs
        self.init_interface(new_inputs)


    def get_output_ports(self):
        """
        router output_ports getter
        """
        return self.__output_ports


    def set_output_ports(self, new_outputs):
        """
        router output_ports setter
        """
        self.__output_ports = new_outputs


    def get_period(self):
        """
        router period getter
        """
        return self.__period


    def set_period(self, new_period):
        """
        router period setter
        """
        self.__period = new_period


    def get_timeout(self):
        """
        router timeout getter
        """
        return self.__timeout


    def set_timeout(self, new_timeout):
        """
        router timout setter
        """
        self.__timeout = new_timeout


    def get_interface(self):
        """
        router interface getter
        """
        return self.__interface


    def get_routing_table(self):
        """
        router routing_table getter
        """
        return self.__routing_table


    def print_routing_table(self):
        """
        Print the current self.__routing_table
        """
        print(routing_table_formatter(self.__router_id,
                                      self.__routing_table))


    def random_offset_period(self):
        """
        randomize self.__period +- TIMER_OFFSET
        """
        self.__period = self.__default_period +\
            random.uniform(-self.REGULAR_TIMER_OFFSET, \
                           +self.REGULAR_TIMER_OFFSET)
        print("Set Router regular update period to " + \
              f"{self.__period:.2f}")


    def random_triggered_updates_period(self):
        """
        randomize self.__triggered_updates_period
        """
        self.__triggered_updates_period = \
            self.__default_triggered_updates_period -\
            random.uniform(0, 0.4)
        print("Set Router triggered update period to " + \
              f"{self.__triggered_updates_period:.2f}")


    def init_interface(self, ports):
        """
        Create a new Interface object and set it as the default
        interface for the current Router object
        """
        self.__interface = Interface(ports)


    def init_routing_table(self):
        """
        Initialise the __routing_table attribute

        Route object format:
        route.next_hop: 2,
        route.metric: 1,
        route.timeout: 1234,
        route.garbage_collect_time: None(default)
        state: 'active'(default)
        """
        # Create a new Route object to router itself
        self_route = Route('-', 0, None)
        self.__routing_table[self.__router_id] = self_route


#-----------------------------------------------------
# Above is the init implementation
#-----------------------------------------------------


    def advertise_all_routes_periodically(self):
        """
        Call advertise_all_routes() periodcally by self.__period

        Use random.random() to calculate offset for self.__period
        in order to avoid synchronized update messages which can lead
        to unnecessary collisions on broadcast networks.
        """
        now = time.time()
        if now - self.__regular_advertise_timer >= self.__period:
            self.advertise_routes('all')
            self.print_routing_table()
            self.__regular_advertise_timer = now
            self.random_offset_period()


    def advertise_updated_routes(self):
        """
        advertise the updated routes to all neighbours
        """
        now = time.time()
        if now - self.__trigger_advertise_timer >= \
           self.__triggered_updates_period:
            self.advertise_routes('update')
            self.print_routing_table()
            self.__trigger_advertise_timer = now
            self.random_triggered_updates_period()


    def advertise_routes(self, mode):
        """
        parameter:
        mode: a string 'all' / 'update'
        get the latest advertising rip packet from
        update_packet() & triggered_packet() methods and
        advertise the packet to all the neighbours (ouput ports)

        need to add a parameter for updata_packet/triggered_packet
        """
        try:
            ports_num = len(self.__output_ports)
            if ports_num < 1:
                raise ValueError("No output port/socket available")
            for dest_port, metric_id in self.__output_ports.items():
                packet = self.update_packet(metric_id['router_id'], mode)
                if packet is None:
                    print("A packet without entry. Stop Sending")
                    return
                self.__interface.send(packet, dest_port)
                current_time = datetime.now().strftime('%H:%M:%S.%f')[:-4]
                if mode == 'all':
                    message = "Sends all routes to Router"
                else:
                    message = 'Sends triggred update to Router'
                print(message +
                     f"{metric_id['router_id']} " +
                     f"[{dest_port}] at {current_time}")
            # clear flags of "update"
            for route in self.__routing_table.values():
                if mode == 'update' and route.state == 'updated':
                    route.state = 'active'
        except ValueError as error:
            print(error)


    def update_packet(self, receiver_id, mode):
        """
        parameter:
        receiver_port

        Process the current routing table data and convert it into
        a rip format packet for advertise_all_routes() method
        """
        # Create RipEntries for all the routes
        entries = []
        for dest, route in self.__routing_table.items():

            if mode == "update" and route.state == "active":
                continue
            metric = route.metric
            # split_horizon_poison_reverse
            if self.__split_horizon_poison_reverse and\
               route.next_hop == receiver_id:
                metric = self.INFINITY
            entry = RipEntry(dest, metric)
            entries.append(entry)

        # Create RipPacket
        packet = RipPacket(entries, self.__router_id)
        packet_bytes = packet.packet_bytes()
        return packet_bytes


#----------------------------------------
# Above is sender implementation
#----------------------------------------


    def receive_routes(self):
        """
        Receive the routes update from neighbours (input ports)

        The implementation is in a while loop and should be called with
        a separate thread from the main thread
        """
        # The __interface only listen to the input ports
        # print(f"Listening to ports at {time.ctime()}")
        packets_list = self.__interface.receive()
        for raw_packet in packets_list:
            self.process_received_packet(raw_packet)


    def process_received_packet(self, raw_packet):
        """
        Process the received packet and call update_routing_table()
        if necessary

        Parameter: packet
        an array of bytes
        """
        # Check if raw_packet valid in RipPacket and RipEntry classes
        # Process the raw_packet if valid,
        # and return (True, RipPacket object)
        # otherwise, return (False, router_id)
        is_valid, rip_packet = RipPacket.decode_packet(raw_packet)
        if is_valid:
            # update routing_table if incoming packet is valid
            print(f'Received update from Router {rip_packet.router_id}')
            self.update_routing_table(rip_packet)
        else:
            # drop the packet if incoming packet is invalid
            print(f'Drop invalid packet from Router {rip_packet}')


    def update_routing_table(self, rip_packet):
        """
        check all the entries in rip_packet object, and update current
        routing table if necessary

        Parameter:
        rip_packet: a valid rip_packet object

        Reture: boolean
        return True if new route added, otherwise False
        """
        # get metric from sender
        sender_id = rip_packet.router_id
        metric_to_sender = None
        for neighbour in self.__output_ports.values():
            if neighbour['router_id'] == sender_id:
                metric_to_sender = neighbour['metric']
        for entry in rip_packet.entries:
            # update the metric for each entry
            # by adding the metric to sender
            # metric = min(metric + metric_to_sender, 16(infinity))
            updated_metric = min(entry.metric + metric_to_sender,
                                 self.INFINITY)
            #if route to dest is unavailable in __routing_table
            if updated_metric != self.INFINITY and\
               not entry.dest in self.__routing_table.keys():
                self.__routing_table[entry.dest] = \
                    Route(sender_id, updated_metric, time.time())
                # Triggered update for new route
                # self.__routing_table[entry.dest].state = 'updated'
                # print("Triggerd update for new route")
                # self.advertise_updated_routes()
            elif entry.dest in self.__routing_table:
                self.update_availabe_route(entry,
                                           updated_metric,
                                           sender_id)


    def update_availabe_route(self, entry, updated_metric, sender_id):
        """
        Parameters:
        entry: a RipEntry object
        sender_id: the router id from which the entry is sent
        """
        # if route to dest is available in __routing_table

        # 1. if packet is from the same router as
        # existing router, reinitialize the timeout anyway
        from_same_router = sender_id == \
            self.__routing_table[entry.dest].next_hop
        is_timeout = not \
            self.__routing_table[entry.dest].garbage_collect_time is \
            None
        if from_same_router:
            self.__routing_table[entry.dest].timeout = time.time()


```

```python
        # 2. compare metrics
        new_metric = updated_metric
        old_metric = self.__routing_table[entry.dest].metric
        have_differnt_metrics = new_metric != old_metric
        is_lower_new_metric = new_metric < old_metric
        is_almost_timeout = \
            not self.__routing_table[entry.dest].timeout is None and \
            not is_timeout and \
            (time.time() - self.__routing_table[entry.dest].timeout) \
            >= self.__timeout / 2

        if from_same_router and have_differnt_metrics:
            self.__routing_table[entry.dest].metric = new_metric
            if not is_timeout and new_metric == self.INFINITY:
                self.__routing_table[entry.dest].garbage_collect_time \
                    = time.time()
                # Triggered update for invalid route
                self.__routing_table[entry.dest].state = 'dying'
                print("triggered update for invalid route")
                self.advertise_updated_routes()
            elif is_timeout:
                self.__routing_table[entry.dest].garbage_collect_time \
                    = None
                self.__routing_table[entry.dest].state = 'active'

        elif is_lower_new_metric:
            self.__routing_table[entry.dest].metric = new_metric
            self.__routing_table[entry.dest].next_hop = sender_id
            self.__routing_table[entry.dest].timeout = time.time()
            if is_timeout:
                self.__routing_table[entry.dest].garbage_collect_time \
                    = None
                self.__routing_table[entry.dest].state = 'active'
            # Triggered update
            # self.__routing_table[entry.dest].state = 'updated'
            # print("triggered updated route from different router with lower metric")
            # self.advertise_updated_routes()
        elif not from_same_router and \
             not have_differnt_metrics and \
             not is_timeout and is_almost_timeout:
            self.__routing_table[entry.dest].next_hop = sender_id
            self.__routing_table[entry.dest].timeout = time.time()


#----------------------------------------
# Above is receiver implementation
#----------------------------------------

    def check_timeout_entries_periodically(self):
        """
        call check_timeout_entries() every default_period
        """
        now = time.time()
        if now - self.__timeout_check_timer >= self.__default_period:
            self.check_timeout_entries()
            self.__timeout_check_timer = now


    def check_timeout_entries(self):
        """
        Check the timeout of each entry in __routing_table

        if an entry is timeout, start its garbage_collect_time
        """
        current_time = datetime.now().strftime('%H:%M:%S.%f')[:-4]
        print(f"Checking timeout entries at {current_time}")

        entries_to_remove = []
        for dest_id, entry in self.__routing_table.items():
            if not entry.timeout is None and \
               entry.garbage_collect_time is None and \
               time.time() - entry.timeout >= self.__timeout:
                entry.garbage_collect_time = time.time()
                entry.metric = self.INFINITY
                entry.state = 'dying'
                # Triggered update
                print("Triggered update for invalid route")
                self.advertise_updated_routes()

            if not entry.garbage_collect_time is None and \
                (time.time() - entry.garbage_collect_time) \
                >= self.__garbage_collection_time:
                entries_to_remove.append(dest_id)

        for dest_id in entries_to_remove:
            self.__routing_table.pop(dest_id)
            print(f"Removed timeout route to {dest_id}")
            self.print_routing_table()


#-------------------------------------------------------
# Above is timeout and garbage_collection implementation
#-------------------------------------------------------

    def __str__(self):
        return ("Router: {0}\n"
                "Input Ports: {1}\n"
                "Output Ports: {2}\n"
                "Period: {3}\n"
                "Timeout: {4}").format(self.__router_id,
                                       self.__input_ports,
                                       self.__output_ports,
                                       self.__period,
                                       self.__timeout)
```

```python
"""
COSC364 2022-S1 Assignment: RIP routing
Authors: MENG ZHANG (71682325), ZHENG CHAO (21671773)
File: router_interface.py
"""

# Import Modules
import socket
import select


# Router Network Interface Class
class Interface:
    """
    A router interface object which includes:
    * Multiple sockets with corresponding ports as instance attributes
    * A series of methods for socket operations:
    - send(port),
    - receive(port)
    """
    def __init__(self, ports):
        """
        Parameters: ports
        ports: a list of integers of port number
        """
        self.host = "127.0.0.1" # local host
        self.select_timeout = 0.5 # default 0.5
        self.ports = ports # input ports
        self.sending_port = ports[0] # set 1st port as the sending port
        self.ports_sockets = {} # input ports and sockets
        self.init_sockets()

    def init_sockets(self):
        """
        Parameter: ports
        ports: a list of integers of ports

        Return: port_socket
        port_socket: a list of
        """
        try:
            for port in self.ports:
                udp_socket = socket.socket(socket.AF_INET,
                                           socket.SOCK_DGRAM)
                udp_socket.bind((self.host, port))
                # udp_socket.setblocking(0)  # blocking switch

                self.ports_sockets[port] = udp_socket
        except socket.error as error:
            print("Failed to initialise sockets for ports\n", error)

    def get_ports_sockets(self):
        """
        ports_sockets getter
        """
        return self.ports_sockets

    def receive(self):
        """
        Using select() to monitor a list of ports and receive the port
        with readable data

        Parameter: sockets
        ports: a list of socket objects

        Return: (data, port)
        """
        sockets = []
        for input_socket in self.ports_sockets.values():
            sockets.append(input_socket)
        sockets_to_read = (select.select(sockets, [], [], \
                            self.select_timeout))[0]
        data_list = []
        for socket_to_read in sockets_to_read:
            # get the receiving port number which the socket binds
            # port = socket_to_read.getsockname()
            # get data from socket
            data = socket_to_read.recv(1024)
            data_list.append(data)
        return data_list

    def send(self, data_bytes, dest_port):
        """
        Parameter: data_bytes
        data_bytes: data in bytes format
        i.e. data can be the update packet from router
        """
        try:
            sending_socket = self.ports_sockets[self.sending_port]
            dest = (self.host, dest_port)
            sending_socket.sendto(data_bytes, dest)
        except KeyError:
            print("The port for sending packet does not exist")
        except socket.error as error:
            print("Can't send packet with the socket\n" + error)

    def __str__(self):
        return ("Host: {0}\n"
              "Ports: {1}\n"
              "Ports_Sockets: {2}").format(self.host,
                                    self.ports,
                                    self.ports_sockets)
```

```python
"""
COSC364 2022-S1 Assignment: RIP routing
Authors: MENG ZHANG (71682325), ZHENG CHAO (21671773)
File: forwarding_route.py
"""

# Route Class
class Route:
    """
    A Route class for creating entries of RIP routing table

    Why we use a class instead of a dictionary/list for route?
    * Compared to list we can quickly get a route value by name
    instead of number index. i.e. route.nexthop
    * Compared to dict/list, a Route class avoid accidental
    modification.
    i.e. What if we accidentally do: route[error_key] = error
    """
    def __init__(self, next_hop, metric,
                 timeout, garbage_collect_time=None,
                 state = 'active'):
        """
        parameters:
        next_hop: an integer of router ID, i.e. 2, 3
        metric: an integer, i.e. 1, 5, 7
        timeout: the current time obtained by time.time()
        garbage_collect_time: None or the current time
        state: a string, i.e. 'active', 'dying', 'updated'
        """
        self.next_hop = next_hop
        self.metric = metric
        self.timeout = timeout
        self.garbage_collect_time = garbage_collect_time
        self.state = state
```

```python
"""
COSC364 2022-S1 Assignment: RIP routing
Authors: MENG ZHANG (71682325), ZHENG CHAO (21671773)
File: rip_packet.py
"""
# RipPacket Class
class RipPacket:
    """
    A class for creating RIP update packet and provide methods to
    encode/decode outputing/incoming packets
    """
    # class attributes
    HEADER_LEN = 4
    ENTRY_LEN = 20


    def __init__(self, entries, router_id, command=2, version=2):
        """
        Parameters:
        entries: a list of rip entry objects
        router_id: the sender ID,  an integer between 1 and 64000
        (use the 16-bit wide all-zero field)
        command: an integer,
        i.e. 2 represents 'response'; 1 represents 'request'
        version: an integer, i.e. 1, 2(default)
        """
        # instance attributes
        self.command = command
        self.version = version
        self.router_id = router_id
        self.entries = entries


    @classmethod
    def decode_packet(cls, raw_packet):
        """
        Parameter:
        raw_packet: a packet of bytes
        i.e.
        HEADER:
        [command(1 byte), version(1), sender_id(2)]
        ENTRY:
        [afi(2 bytes), padding(2)
        dest(4)
        padding(4)
        padding(4)
        metric(4)]

        Return (True, RipPacket object) if raw_packet is valid,
        otherwise return (False, sender_id)
        """
        # Header: 4 bytes [0:4]
        command = raw_packet[0]
        version = raw_packet[1]
        sender_id = (raw_packet[2] << 8) + raw_packet[3]
        entries_num = int(len(raw_packet[4:]) / cls.ENTRY_LEN)
        # check header validity
        if not cls.is_valid_header(command, version,
                                   sender_id, entries_num):
            print("Broken packet:", "invalid header")
            return (False, sender_id)
        # Entries: n * 20 bytes [4:]
        # decode each entry
        entries = []
        for i in range(4, len(raw_packet), cls.ENTRY_LEN):
            raw_entry = raw_packet[i:i+cls.ENTRY_LEN]
            entry = RipEntry.decode_enty(raw_entry)
            # check entry validity
            # invalid entry is represented as None
            if entry is None:
                print("Broken packet:", "invalid entry")
                return (False, sender_id)
            entries.append(entry)
        rip_packet = RipPacket(entries, sender_id)
        return (True, rip_packet)


    def packet_bytes(self):
        """
        Return a rip packet in array of bytes
        """
        return self.header_bytes() + self.entries_bytes()


    def header_bytes(self):
        """
        Common header: 4 bytes in total
        [command(1 byte), version(1), sender_id(2)]

        command:2 (1 byte)
        version: 2 (1 byte)
        rouer_id: 1-64000 (2 bytes)

        Return a 4-byte rip header.
        """
        command_byte = self.command.to_bytes(1, byteorder='big')
        version_byte = self.version.to_bytes(1, byteorder='big')
        sender_id_bytes = self.router_id.to_bytes(2, byteorder='big')
        header = command_byte + version_byte + sender_id_bytes
        return header


    def entries_bytes(self):
        """
        Return a list of 20-byte rip entry
        """
        entries = bytes()
        for entry in self.entries:
            entries += entry.entry_bytes()
        return entries

    @classmethod
    def is_valid_header(self, command, version, router_id, entries_num):
        """
        check if a packet is valid
        """
        is_valid_command = command == 2
        if (not is_valid_command):
            print(f"invalid header command: {command}")
        is_valid_version = version == 2
        if (not is_valid_version):
            print(f"invalid header version: {version}")
        is_valid_id = 1 <= router_id <= 64000
        if (not is_valid_id):
            print(f"invalid header id: {router_id}")
        is_valid_entries_num = 1 <= entries_num <= 25
        if (not is_valid_entries_num):
            print(f"invalid header entries num: {entries_num}")
        return is_valid_command and\
               is_valid_version and\
               is_valid_id and\
               is_valid_entries_num

```

```python


# RipEntry Class
class RipEntry:
    """
    A class for creating entry objects in a rip packet
    """

    # class attributes
    PADDING_2BYTES = (0).to_bytes(2, byteorder='big')
    PADDING_4BYTES = (0).to_bytes(4, byteorder='big')


    def __init__(self, dest, metric, afi=2):
        """
        Parameters:
        dest: an integer, router_id of destination
        metric: an integer between 1 and 16 (inclusive)
        AFI: Address FAmily Identifier
        """
        self.dest = dest
        self.metric = metric
        self.afi = afi


    @classmethod
    def decode_enty(cls, raw_entry):
        """
        Parameter:
        raw_entry: an entry of bytes
        i.e.
        ENTRY:
        [afi(2 bytes), padding(2)
        dest(4)
        padding(4)
        padding(4)
        metric(4)]

        Return RipEntry object if raw_entry is valid,
        otherwise return None
        """
        # afi: 2 bytes [0:3]
        afi = (raw_entry[0] << 8) + raw_entry[1]
        # dest: 4 bytes but practically take 2 bytes [4:8]
        if (raw_entry[4] != 0 or
            raw_entry[5] != 0):
            print("Invalid dest of entry")
            return None
        dest = (raw_entry[6] << 8) + raw_entry[7]
        # metric 4 bytes but practically take 1 byte [16:]
        if (raw_entry[16] != 0 or
            raw_entry[17] != 0 or
            raw_entry[18] != 0):
            print("Invalid metric of entry")
            return None
        metric = raw_entry[19]
        entry = RipEntry(dest, metric, afi)
        if not entry.is_valid_entry():
            return None
        return entry


    def entry_bytes(self):
        """
        Rip entry: 20 bytes each
        [afi(2 bytes), padding(2)
        dest(4)
        padding(4)
        padding(4)
        metric(4)]

        afi: 2 (2 bytes)
        dest: 1-64000 (4 bytes)
        metric: 1-16 (4 bytes)
        padding: 0 (2 or 4 bytes)
        """
        afi_bytes = self.afi.to_bytes(2, byteorder='big')
        dest_bytes = self.dest.to_bytes(4, byteorder='big')
        metric_bytes = self.metric.to_bytes(4, byteorder='big')
        entry = afi_bytes + self.PADDING_2BYTES +\
                          dest_bytes +\
                          self.PADDING_4BYTES +\
                          self.PADDING_4BYTES +\
                          metric_bytes
        return entry

    def is_valid_entry(self):
        """
        check if an entry is valid
        """
        is_valid_dest = 1 <= self.dest <= 64000
        is_valid_metric = 0 <= self.metric <= 16
        is_valid_afi = self.afi == 2
        return is_valid_dest and is_valid_metric and is_valid_afi

    def set_metric_infinite(self):
        """
        set the metric to be infinite(16)
        """
        self.metric = 16

    def increment_metric(self):
        """
        add 1 to metric
        """
        self.metric += 1
```

```python
"""
COSC364 2022-S1 Assignment: RIP routing
Authors: MENG ZHANG (71682325), ZHENG CHAO (21671773)
File: IO_formatter.py
"""
import time

def routing_table_formatter(router_id, table):
    """
    Parameters:
    table: a dictionary of routes
    {id1: Route object, id2: Route object, ...}

    Return:
    table: a formatted string which contains data of the table
    """
    # Get header
    header = table_header_formatter(router_id)
    # Get content
    content = table_content_formatter(table)
    return header + content


def talbe_border_formatter(length):
    """
    return two formatted routing table borders
    """
    border = length * '-'
    double_border = length * '='
    return border, double_border


def table_header_formatter(router_id):
    """
    return a formatted routing table header
    """
    border, double_border = talbe_border_formatter(72)
    title = f'Router {router_id:02} RIP ROUTING TABLE'
    padded_title = '|' + 21 * ' ' + title + 22 * ' ' + '|'
    labels = "|   Dest   |   Next   |   Metric   |  Timeout  |  Garbage  |   State   |"
    header = '\n' + double_border + '\n' +\
        padded_title + '\n' +\
        double_border + '\n' +\
        labels + '\n' +\
        border + '\n'
    return header


def table_content_formatter(table):
    """
    retrun formatted routing table content
    """
    border = talbe_border_formatter(72)[0]
    content = ""
    for dest, rip_route in table.items():
        next_hop = rip_route.next_hop
        metric = rip_route.metric

        timeout = rip_route.timeout
        if not timeout is None:
            timeout = round(time.time() - rip_route.timeout, 2)
            timeout_str = f'{timeout:^11.1f}'
        else:
            timeout_str = 5 * ' ' +  '-' + 5 * ' '

        garbage_collect_time = rip_route.garbage_collect_time
        if not garbage_collect_time is None:
            garbage_collect_time = int(time.time() - garbage_collect_time)
            gc_str = f'{garbage_collect_time:^11.0f}'
        else:
            gc_str = 5 * ' ' +  '-' + 5 * ' '
        state = rip_route.state
        content += f'|{dest:^10}|{next_hop:^10}|{metric:^12}|'+\
            f'{timeout_str}|{gc_str}|{state:^11}|' + '\n' +\
            border + '\n'
    return content
```