```python
"""This is the py_athletics athlete module."""

from activity.activity import Activity
from activity.activity import Cycle, Run, Tennis, Walk, Workout
from goal.goal import Goal, YearGoal, CumulativeGoal, MonthGoal
from helpers.helpers import td_cvt, is_date, parse_date, none_factory
from helpers.garmin_helpers import garmin_to_decimal, garmin_to_int, garmin_to_time

from collections import defaultdict
from datetime import datetime, timedelta, date
from csv import DictReader
from pickle import dump, load
from typing import Union
import unicodedata


class Athlete:

    """py_athletics Athlete class."""

    def __init__(self):
        """Create an Athlete."""

        # The activities attribute is a set of nested defaultdicts.
        # Activities are partitioned by Activity class objects
        # and then by datetime start objects.
        #
        # Since data collected from Garmin is typically static and there is a
        # possibility that the activity object could be updated inside
        # py_athletics, we ignore subsequent attempts to add the activity.
        # A more sophisticated approach regarding changes could be added in the
        # future.
        #
        # The goals attribute is a set nested defaultdicts.
        # Goals are partitioned by Activity class objects and then by the goal
        # timeframe.
        #
        # New goals supersede any prior goals for the specified timeframe.
        #
        # The activities and goals attributes are hidden and should be
        # accessed with add_activity, add_goal, get_activities and
        # get_goals methods.

        self.__activities = defaultdict(none_factory)

        for activity_subclass in Activity.subclasses():
            self.__activities[activity_subclass] = defaultdict(none_factory)

        self.__goals = defaultdict(none_factory)

        for activity_subclass in Activity.subclasses():
            self.__goals[activity_subclass] = defaultdict(none_factory)

    def __repr__(self) -> str:
        activity_count = 0
        for class_dict in self.__activities.values():
            activity_count += len(class_dict)

        goal_count = 0
        for goal_dict in self.__goals.values():
            goal_count += len(goal_dict)

        return f"(Athlete with {activity_count} activities and {goal_count} goals)"

    def save(self, filename: str = "py_athletics.pickle") -> None:
        """Save Athlete data to a file.
        The default filename is py_athletics.pickle, a different name can be
        specified with the filename keyword argument.

        Optional Parameters
        -------------------
        filename: string
        """

        with open(filename, "wb") as pickle_out:
            dump(self, pickle_out)

    @staticmethod
    def load(filename: str = "py_athletics.pickle"):
        """Load Athlete data from a file.

        The default filename is py_athletics.pickle, a different name can be
        specified with the filename keyword argument.

        Optional Parameters
        -------------------
        filename: string
        """

        with open(filename, "rb") as pickle_in:
            athlete = load(pickle_in)
        return athlete

    def add_activity(self, activity: Activity) -> None:
        """Add an Activity if the Athlete does not already have an Activity
        of the same type and with the same start datetime.
        """

        if not isinstance(activity, Activity):
            raise TypeError("activity must be an Activity")

        activity_type = type(activity)

        # Grab the relevant activity type dictionary

        subclass_activities = self.__activities[activity_type]

        # Then add activity if it is not already in that dictionary

        if subclass_activities[activity.start] is None:
            subclass_activities[activity.start] = activity

    def get_activities(self, activity_subclass=None) -> list:
        """Return a list containing an Athlete's activities.  The activities_subclass
        parameter is used to limit the results to the specified subclass.
        """

        if activity_subclass and activity_subclass not in Activity.subclasses():
            raise ValueError("invalid activity subclass")

        # If the activity_subclass parameter is specified, return a list
        # of activities for that activity subclass, otherwise return a list
        # of all activities.

        if activity_subclass:
            activities = self.__activities[activity_subclass].values()
            result = [activity for activity in activities]

        else:
            result = [
                activity
                for sub_dict in self.__activities.values()
                for activity in sub_dict.values()
            ]

        return result

    def add_goal(self, exercise: str, metric: str, timeframe: str, target: int) -> None:
        """Add a Goal.

        A Goal replaces existing Goals for the same activity class and metric.
        The distance metric is only valid for Cycle, Run and Walk activities.

        Parameters
        ----------
        exercise: string = {Cycle|Run|Tennis|Walk|Workout}
        metric: string = {count|distance|duration}
        timeframe: string = {month|year|cumulative}
        target: a positive integer
        """

        if not isinstance(exercise, str):
            raise TypeError("exercise must be a string")

        if exercise not in Activity.subclass_names():
            raise ValueError("invalid exercise")

        if not isinstance(timeframe, str):
            raise TypeError("timeframe must be a string")

        if timeframe not in Goal.GOAL_TIMEFRAMES:
            raise ValueError("invalid timeframe")

        if not isinstance(metric, str):
            raise TypeError("metric must be a string")

        if metric not in Goal.GOAL_METRICS:
            raise ValueError("invalid metric")

        target_class = Activity.activity_dictionary()[exercise]

        if timeframe == "month":
            goal = MonthGoal(target_class, metric, int(target))
        elif timeframe == "year":
            goal = YearGoal(target_class, metric, int(target))
        else:
            goal = CumulativeGoal(target_class, metric, int(target))

        # Grab the relevant goal dictionary

        subclass_goals = self.__goals[target_class]

        # The class specific goal dictionary uses (metric, timeframe)
        # tuples as keys.  New goals supersede prior goals.

        subclass_goals[(metric, timeframe)] = goal

    def get_goals(self, activity_subclass=None) -> list:
        """Return a list containing an Athlete’s goals.  If the optional
        activity_subclass parameter is specified, the result is limited to
        goals for that subclass.
        """

        if activity_subclass and activity_subclass not in Activity.subclasses():
            raise ValueError("invalid activity subclass")

        # If the activity_subclass parameter is specified, return a list
        # of goals for that activity subclass, otherwise return a list
        # of all goals.

        if activity_subclass:
            goals = self.__goals[activity_subclass].values()
            result = [goal for goal in goals]

        else:
            result = [
                goal for sub_dict in self.__goals.values() for goal in sub_dict.values()
            ]

        return result

    def delete_goal(self, exercise: str, metric: str, timeframe: str) -> None:
        """Delete a Goal.

        Parameters
        ----------
        exercise: string = {Cycle|Run|Tennis|Walk|Workout}
        metric: string = {count|distance|duration}
        timeframe: string = {month|year|cumulative}
        """

        if not isinstance(exercise, str):
            raise TypeError("exercise must be a string")

        if exercise not in Activity.subclass_names():
            raise ValueError("invalid exercise")

        target_class = Activity.activity_dictionary()[exercise]

        # Grab the relevant goal dictionary

        subclass_goals = self.__goals[target_class]

        # The class specific goal dictionary uses (metric, timeframe)
        # tuples as keys.  Delete the key if it exists, otherwise
        # return None.

        subclass_goals.pop((metric, timeframe), None)

        return None

    def show_goals(self, exercise: str = None) -> None:
        """Display a list of Goals.

        If exercise is specified, the listing is limited to that exercise.

        Optional Parameters
        -------------------
        exercise: string = {Cycle|Run|Tennis|Walk|Workout}
        """

        # If exercise is not specified, make recursive calls
        # over every Activity subclass.

        if exercise is None:
            for name in Activity.subclass_names():
                self.show_goals(exercise=name)
            return

        # The class was specified, so handle it.

        if not isinstance(exercise, str):
            raise TypeError("class name must be a string")
        if exercise not in Activity.subclass_names():
            raise ValueError("invalid class name")

        target_class = Activity.activity_dictionary()[exercise]

        for goal in self.get_goals(target_class):
            print(repr(goal))

        return

    def summarize_goals(self, exercise=None) -> None:
        """Display a summary of Goals.

        If exercise is specified, the listing is limited to that exercise.

        Optional Parameters
        -------------------
        exercise: string = {Cycle|Run|Tennis|Walk|Workout}
        """

        # If exercise is not specified, make recursive calls
        # over every Activity subclass.

        if exercise is None:
            for name in Activity.subclass_names():
                self.summarize_goals(exercise=name)
            return

        # The class was specified, so handle it.

        if not isinstance(exercise, str):
            raise TypeError("class name must be a string")

        if exercise not in Activity.subclass_names():
            raise ValueError("invalid class name")

        for goal in self.get_goals(Activity.activity_dictionary()[exercise]):
            goal.report(athlete=self)

        return

    def read_garmin_activity_file(self, filename="Activities.csv"):
        """Read a Garmin activity file and create Activity objects.

        Garmin fitness data is stored at http://connect.garmin.com.
        Subscribers can download comprehensive activity data into a CSV file.
        This method reads a Garmin activity file, creates py_athletics
        Activity objects and adds them to the Athlete’s activity collection.

        The  default filename is Activities.csv, a different name can be
        specified with the filename keyword argument.

        Optional Parameters
        -------------------
        filename: string
        """

        # A Garmin activity file is a CSV file with activity information.
        # The first row is a header row.
        # We open the file and read it with csv.Dictreader
        #
        # We transform the start field into a datetime object
        # We transform the duration field into a timedelta object, we look at
        # the first 8 characters only because sometimes Garmin includes
        # fractional seconds which we will ignore.
```

```
# Garmin includes the registered sign character in some fields.
CIRCLE_R = unicodedata.lookup("REGISTERED SIGN")
NORMALIZED_POWER_KEY = f"Normalized Power{CIRCLE_R} (NP{CIRCLE_R})"

with open(filename, "rt") as garmin_activities_csv_file:
    activity_reader = DictReader(garmin_activities_csv_file)
    for activity_row in activity_reader:

        # We examine the Activity Type column in the CSV file to
        # determine the Activity subclass we will use.  We bind
        # a variable to the correct function to call to create the
        # Activity subclass instance.

        garmin_activity_type = activity_row["Activity Type"]

        if "Cycling" in garmin_activity_type:
            instantiator = Cycle
        elif "Gym" in garmin_activity_type:
            instantiator = Workout
        elif "Running" in garmin_activity_type:
            instantiator = Run
        elif "Tennis" in garmin_activity_type:
            instantiator = Tennis
        elif "Walking" in garmin_activity_type:
            instantiator = Walk
        else:
            instantiator = Activity

        # These fields are always applicable.

        start_string = activity_row["Date"]
        start = datetime.fromisoformat(start_string)

        duration_string = activity_row["Time"][0:8]
        duration_dt = datetime.strptime(duration_string, "%H:%M:%S")
        duration = timedelta(
            hours=duration_dt.hour,
            minutes=duration_dt.minute,
            seconds=duration_dt.second,
        )

        description = activity_row["Title"]

        calories_string = activity_row["Calories"]
        calories = garmin_to_int(calories_string)

        max_HR_string = activity_row["Max HR"]
        maximum_heart_rate = garmin_to_int(max_HR_string)

        avg_HR_string = activity_row["Avg HR"]
        average_heart_rate = garmin_to_int(avg_HR_string)

        # Distance is only meaningful for Cycling, Running and
        # Walking, so we will ignore distance data from Garmin
        # for other Activity subclasses.

        distance = None

        if (
            "Cycling" in garmin_activity_type
            or "Running" in garmin_activity_type
            or "Walking" in garmin_activity_type
        ):
            distance_string = activity_row["Distance"]
            distance = garmin_to_decimal(distance_string)
```

```
                # Sadly, Garmin uses MPH for cycling speed and minutes:seconds
                # for runnning and walking speed.  As with distance these
                # fields are not relevant for other Activity subclasses.

                max_speed_string = activity_row["Max Speed"]
                avg_speed_string = activity_row["Avg Speed"]
                maximum_speed = None
                average_speed = None

                if "Cycling" in garmin_activity_type:
                    maximum_speed = garmin_to_decimal(max_speed_string)
                    average_speed = garmin_to_decimal(avg_speed_string)

                if (
                    "Running" in garmin_activity_type
                    or "Walking" in garmin_activity_type
                ):
                    maximum_speed = garmin_to_time(max_speed_string)
                    average_speed = garmin_to_time(avg_speed_string)

                # Normalized power is only meaningful for Cycling, so we will
                # ignore normalized power data from Garmin for other Activity
                # subclasses.

                normalized_power = None

                if "Cycling" in garmin_activity_type:
                    np_string = activity_row[NORMALIZED_POWER_KEY]
                    normalized_power = garmin_to_int(np_string)

                activity = instantiator(
                    start=start,
                    duration=duration,
                    garmin_activity_type=garmin_activity_type,
                    description=description,
                    calories=calories,
                    maximum_heart_rate=maximum_heart_rate,
                    average_heart_rate=average_heart_rate,
                    distance=distance,
                    maximum_speed=maximum_speed,
                    average_speed=average_speed,
                    normalized_power=normalized_power,
                )

                self.add_activity(activity)

    def show_activities(
        self, exercise: str = None, start: str = None, end: str = None
    ) -> None:
        """Display a list of Activities.

        If exercise is specified the listing is limited to that exercise.
        A timeframe for the listing can be established with one or both of the
        start and end keywords.

        Optional Parameters
        -------------------
        exercise: string = {Cycle|Run|Tennis|Walk|Workout}
        start: string in the form YYYY-MM-DD
        end: string in the form YYYY-MM-DD
        """

        # If exercise is not specified, make recursive calls
        # over every Activity subclass.
        if exercise is None:
            for name in Activity.subclass_names():
                self.show_activities(exercise=name, start=start, end=end)
            return

        # The class was specified, so handle it.

        if not isinstance(exercise, str):
            raise TypeError("class name must be a string")

        if exercise not in Activity.subclass_names():
            raise ValueError("invalid class name")

        if start and not isinstance(start, str):
            raise TypeError("start must be a string")

        if start is None:
            start_date = parse_date("1970-01-01")
        else:
            if is_date(start):
                start_date = parse_date(start)
            else:
                raise ValueError("invalid start")

        if end and not isinstance(end, str):
            raise TypeError("end must be a string")

        if end is None:
            end_date = date.today()
        else:
            if is_date(end):
                end_date = parse_date(end)
            else:
                raise ValueError("invalid end")

        target_class = Activity.activity_dictionary()[exercise]

        for activity in self.get_activities(target_class):
            if start_date <= activity.start.date() <= end_date:
                print(repr(activity))

    def summarize_activities(self, exercise=None, start=None, end=None) -> None:
        """Display a summary of Activities.

        If exercise is specified the listing is limited to that exercise.
        A timeframe for the listing can be established with one or both of the
        start and end keywords.

        Optional Parameters
        -------------------
        exercise: string = {Cycle|Run|Tennis|Walk|Workout}
        start: string in the form YYYY-MM-DD
        end: string in the form YYYY-MM-DD
        """

        # If exercise is not specified, make recursive calls
        # over every Activity subclass.

        if exercise is None:
            for name in Activity.subclass_names():
                self.summarize_activities(exercise=name, start=start, end=end)
            return
        # The class was specified, so handle it.

        if not isinstance(exercise, str):
            raise TypeError("class name must be a string")
        if exercise not in Activity.subclass_names():
            raise ValueError("invalid class name")

        tally = Activity.tally(self, exercise, start=start, end=end)

        # If there were no activities, return.
        if tally["count"] == 0:
            return

        hr, min, sec = td_cvt(tally["duration"])
        f_1 = f"{exercise:7} Summary: "
        f_2 = f"Activity Count: {tally[’count’]:2,} "
        f_3 = f"Exercise Time (h:m:s): {hr:3}:{min:02}:{sec:02} "
        f_4 = f"Calories Burned: {tally[’calories’]:6,}"

        if exercise in ("Cycle", "Run", "Walk"):
            f_5 = f" Distance (miles): {tally[’distance’]:>8,}"
        else:
            f_5 = ""
        print(f_1 + f_2 + f_3 + f_4 + f_5)

        return

    def earliest_activity(self, exercise: str) -> Union[datetime, None]:
        """Return a datetime object for the earliest exercise instance."""

        target_class = Activity.activity_dictionary()[exercise]
        activity_list = self.get_activities(target_class)
        date_list = [activity.start for activity in activity_list]
        if date_list:
            return min(date_list)
        else:
            return None
```