```
"""This is the py_athletics activity module."""

import datetime
from decimal import Decimal
from collections import Counter
from helpers.helpers import parse_date


class Activity:
    """py_athletics Activity class."""

    # Activities can either be indoor or outdoor.  While the venue attribute
    # can be set manually, it will more often be derived as part of a subclass
    # instantiation process.  Venues are not yet recognized.

    venue_type_set = {"indoor", "outdoor"}

    def __init__(
        self, start: datetime.datetime, duration: datetime.timedelta, **kwargs
    ):
        """Create an Activity.

        This method requires start and duration arguments and will accept
        keyword arguments for all other attributes.  All Activity subclasses
        call it after they have removed subclass specific keyword arguments.
        Surplus keyword arguments are tolerated and ignored.  Venues are
        currently accepted by not used.
        """

        # The overwhelming bulk of the init method is to do type and value
        # checking.  Because of the rich set of optional attributes this code
        # relies on **kwargs.  It looks for specific keywords and ignores
        # everything else.  This is done, in part, to permit subclasses to add
        # attributes and not require them to strip related keywords in advance
        # of calling this methods, and in part, for convenience.

        if not isinstance(start, datetime.datetime):
            raise TypeError("start must be a datetime")

        if not isinstance(duration, datetime.timedelta):
            raise TypeError("duration must be a timedelta object")

        # Handle optional arguments from argument dictionary.

        description = kwargs.pop("description", None)
        if description and not isinstance(description, str):
            raise TypeError("description must be a string")

        calories = kwargs.pop("calories", None)
        if calories and not isinstance(calories, int):
            raise TypeError("calories must be an integer")
        if calories and calories <= 0:
            raise ValueError("calories must be positive")

        maximum_heart_rate = kwargs.pop("maximum_heart_rate", None)
        if maximum_heart_rate and not isinstance(maximum_heart_rate, int):
            raise TypeError("maximum heart rate must be an integer")
        if maximum_heart_rate and maximum_heart_rate <= 0:
            raise ValueError("maximum heart rate must be positive")

        average_heart_rate = kwargs.pop("average_heart_rate", None)
        if average_heart_rate and not isinstance(average_heart_rate, int):
            raise TypeError("average heart rate must be an integer")
        if average_heart_rate and average_heart_rate <= 0:
            raise ValueError("average heart rate must be positive")
```

```
        venue = kwargs.pop("venue", None)
        if venue and not isinstance(venue, str):
            raise TypeError("venue must be a string")

        venue_type = kwargs.pop("venue_type", None)
        if venue_type and not isinstance(venue_type, str):
            raise TypeError("venue must be a string")
        if venue_type and venue_type not in Activity.venue_type_set:
            raise ValueError("invalid venue type")

        garmin_activity_type = kwargs.pop("garmin_activity_type", None)
        if garmin_activity_type and not isinstance(garmin_activity_type, str):
            raise TypeError("garmin activity type must be a string")

        self.start = start
        self.duration = duration
        self.description = description
        self.calories = calories
        self.maximum_heart_rate = maximum_heart_rate
        self.average_heart_rate = average_heart_rate
        self.venue = venue
        self.venue_type = venue_type
        self.__garmin_activity_type = garmin_activity_type

    @property
    def garmin_activity_type(self) -> str:
        """Get the Garmin activity type string used to derive the Activity
        subclass for Activities read from a Garmin activity file."""

        return self.__garmin_activity_type

    def __str__(self) -> str:

        # The __name__ attribute of the class object is a convenient
        # was to get informtion about an instance.  We use it here to
        # add the class or subclass name, as the case may be,
        # in the return string.

        class_str = type(self).__name__
        start_str = self.start.strftime("%Y-%m-%d at %H:%M")
        duration_str = str(self.duration)
        return f"[{class_str} on {start_str} for {duration_str}]"

    def __repr__(self) -> str:

        # The __repr__ method serves as the foundation for the py_athletics
        # activity reporting system.  Subclasses that add attributes to be
        # reported can call this method for the bulk of the string and modify
        # it as need be for subclass purposes.

        class_str = type(self).__name__
        start_str = self.start.strftime("%Y-%m-%d at %H:%M")
        duration_str = str(self.duration)

        # Don't bother with a description if it is the same text as an obvious
        # gerund.  When we think we have a useful description, will annotate
        # the class reference in the return string with the description. Any
        # occurences of None are disregarded as well.

        if (
            (self.description == class_str)
            or (self.description is None)
            or (self.description == "Running" and class_str == "Run")
            or (self.description == "Cycling" and class_str == "Cycle")
        ):
            caption = ""
```

```
        else:
            caption = f" ({self.description})"

        # Transform None to "--" for optional fields other than
        # description, which is handled above.  We will do the same
        # for 0 because that typically means data wasn't captured.
        # Since "if x" will not be true if x is None or x is 0, we
        # can set the field to "--" then test on x and adjust.

        calories_token = "--"
        if self.calories:
            calories_token = str(self.calories)

        max_hr_token = "--"
        if self.maximum_heart_rate:
            max_hr_token = str(self.maximum_heart_rate)

        avg_hr_token = "--"
        if self.average_heart_rate:
            avg_hr_token = str(self.average_heart_rate)

        f_1 = f"[{class_str}{caption} on {start_str} "
        f_2 = f"for {duration_str} Calories: {calories_token} "
        f_3 = f"Max HR: {max_hr_token} Avg HR: {avg_hr_token}]"

        return f_1 + f_2 + f_3

    @staticmethod
    def subclasses() -> tuple:
        """Return a tuple of Activity subclasses."""
        return tuple(Activity.__subclasses__())

    @staticmethod
    def subclass_names() -> tuple:
        """Return a tuple of Activity subclass names."""
        return tuple([cls.__name__ for cls in Activity.subclasses()])

    @staticmethod
    def activity_dictionary() -> dict:
        """Return a dictionary of Activity subclass names to subclasses."""
        return {cls.__name__: cls for cls in Activity.subclasses()}

    @staticmethod
    def tally(athlete, class_name: str, start=None, end=None):
        """Return a Counter with athlete's aggregated activity data for the specified
        Activity class.  All counters include activity count, calories and
        duration.  Counters for Walk, Cycle and Run include distance.
        """

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

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

        if start is None:
            start_date = parse_date("1970-01-01")
        else:
            start_date = parse_date(start)

        if end is None:
            end_date = datetime.date.today()
        else:
            end_date = parse_date(end)

        target_class = Activity.activity_dictionary()[class_name]
```

```
        tally = Counter({"count": 0, "calories": 0, "duration": datetime.timedelta()})

        if class_name in ("Cycle", "Run", "Walk"):
            tally.update({"distance": Decimal(0)})

        activities = athlete.get_activities(target_class)
        for activity in activities:
            if start_date <= activity.start.date() <= end_date:
                tally.update({"count": 1, "duration": activity.duration})
                if activity.calories:
                    tally.update({"calories": activity.calories})
                if class_name in ("Cycle", "Run", "Walk") and activity.distance:
                    tally.update({"distance": activity.distance})

        return tally


class Cycle(Activity):
    """py_athletics Cycle Activity subclass. A Cycle object may include all
    Activity attributes as well as distance, type, maximum_speed,
    Average_speed, average_power and maximum_average_power attributes.
    Cycle type is a string and can be one of 'commute', 'road', 'trail',or
    'stationary'.  The maximum_speed and average_speed attributes are
    expressed in miles per hour.  The normalized_power attribute is in watts.
    Cycle type is accepted but not yet used.
    """

    cycle_type_set = {"commute", "road", "trail", "stationary"}

    def __init__(
        self, start: datetime.datetime, duration: datetime.timedelta, **kwargs
    ):
        """Create a Cycle Activity. This method requires start and duration
        arguments and will accept keyword arguments for all other Activity and
        Cycle attributes."""

        # Remove distance, type, maximum_speed, average_speed, average_power,
        # and maximum_average_power from the argument dictionary and pass the
        # remainder to Activity for handling.

        distance = kwargs.pop("distance", None)
        type = kwargs.pop("type", None)
        maximum_speed = kwargs.pop("maximum_speed", None)
        average_speed = kwargs.pop("average_speed", None)
        normalized_power = kwargs.pop("normalized_power", None)

        super().__init__(start, duration, **kwargs)

        if distance and not isinstance(distance, Decimal):
            raise TypeError("distance must be a Decimal")
        if distance and distance <= 0:
            raise ValueError("distance must be positive")

        if type and not isinstance(type, str):
            raise TypeError("type must be a string")
        if type and type not in Cycle.cycle_type_set:
            raise ValueError("invalid Cycle type")

        if maximum_speed and not isinstance(maximum_speed, Decimal):
            raise TypeError("maximum speed must be a Decimal")
        if maximum_speed and maximum_speed <= 0:
            raise ValueError("maximum must be positive")

        if average_speed and not isinstance(average_speed, Decimal):
            raise TypeError("average  speed must be a Decimal")
```

```
        if average_speed and average_speed <= 0:
            raise ValueError("average speed must be positive")

        if normalized_power and not isinstance(normalized_power, int):
            raise TypeError("normalized power must be an integer")
        if normalized_power and normalized_power <= 0:
            raise ValueError("normalized power must be positive")

        self.distance = distance
        self.type = type
        self.maximum_speed = maximum_speed
        self.average_speed = average_speed
        self.normalized_power = normalized_power

    def __repr__(self):

        # The Cycle class adds distance, maximum_speed, average_speed and
        # normalized power to the detail to be included in the __repr__
        # string.  The method gets the common base string from the parent
        # class and then adds the additional detail before the closing
        # bracket.

        super_class_str = super().__repr__()

        # Transform None and 0 to "--" for optional fields.
        # Since "if x" will not be true if x is None or x is 0, we
        # can set the field to "--" then test on x and adjust.

        distance_token = "--"
        if self.distance:
            distance_token = self.distance

        max_speed_token = "--"
        if self.maximum_speed:
            max_speed_token = self.maximum_speed

        avg_speed_token = "--"
        if self.average_speed:
            avg_speed_token = self.average_speed

        power_token = "--"
        if self.normalized_power:
            power_token = self.normalized_power

        f_1 = f"Distance (miles): {distance_token} "
        f_2 = f"Max Speed (mph): {max_speed_token} "
        f_3 = f"Avg Speed (mph): {avg_speed_token} "
        f_4 = f"Normalized Power (watts): {power_token}"

        addendum_str = " " + f_1 + f_2 + f_3 + f_4

        return super_class_str[:-1] + addendum_str + super_class_str[-1:]


class Run(Activity):
    """py_athletics Run Activity subclass.  A Run object may include all
    Activity attributes as well as distance, type, maximum_speed and
    average_speed attributes.  Run type is a string and can be one of
    'track', 'road' or 'treadmill'.  The maximum_speed and average_speed
    attributes are expressed in minutes per mile.  Run type is accepted but
    not yet used.
    """

    run_type_set = {"track", "road", "treadmill"}

    def __init__(
```

```
    self, start: datetime.datetime, duration: datetime.timedelta, **kwargs
):
    """Create a Run Activity.  This method requires start and duration
    arguments and will accept keyword arguments for all other Activity and
    Run attributes.
    """

    # Remove distance, type, maximum_speed and average_speed
    # from the argument dictionary and pass the remainder
    # to Activity for handling.

    distance = kwargs.pop("distance", None)
    type = kwargs.pop("type", None)
    maximum_speed = kwargs.pop("maximum_speed", None)
    average_speed = kwargs.pop("average_speed", None)

    super().__init__(start, duration, **kwargs)

    if distance and not isinstance(distance, Decimal):
        raise TypeError("distance must be a Decimal")
    if distance and distance <= 0:
        raise ValueError("distance must be positive")

    if type and not isinstance(type, str):
        raise TypeError("type must be a string")
    if type and type not in Run.run_type_set:
        raise ValueError("invalid run type")

    if maximum_speed and not isinstance(maximum_speed, datetime.time):
        raise TypeError("maximum speed must be a time object")

    if average_speed and not isinstance(average_speed, datetime.time):
        raise TypeError("average speed must be a time object")

    self.distance = distance
    self.type = type
    self.maximum_speed = maximum_speed
    self.average_speed = average_speed

def __repr__(self):

    # The Run class adds distance, maximum_speed, and average_speed
    # to the detail to be included in the __repr__string.  The method
    # gets the common base string from the parent class and then adds
    # the additional detail before the closing bracket.

    super_class_str = super().__repr__()

    # Transform None and 0 to "--" for optional fields.
    # Since "if x" will not be true if x is None or x is 0, we
    # can set the field to "--" then test on x and adjust.
    # Speed measures are in time, representing minutes per mile.

    distance_token = "--"
    if self.distance:
        distance_token = self.distance

    max_speed_token = "--"
    if self.maximum_speed:
        max_speed_token = self.maximum_speed.strftime("%M:%S")

    avg_speed_token = "--"
    if self.average_speed:
        avg_speed_token = self.average_speed.strftime("%M:%S")

    f_1 = f"Distance (miles): {distance_token} "
```

```
        f_2 = f"Max Speed (minutes/mile): {max_speed_token} "
        f_3 = f"Avg Speed (minutes/mile): {avg_speed_token}"

        addendum_str = " " + f_1 + f_2 + f_3

        return super_class_str[:-1] + addendum_str + super_class_str[-1:]


class Tennis(Activity):
    """py_athletics Tennis Activity subclass.  A Tennis object may include all
    Activity attributes as well as type and partner attributes. The type
    attribute is a string and can be one of "ball_machine", "cardio", "drill",
    "hitting_session", "lesson", or "match".  The partner attribute is a
    string.  The type and partner attributes are accepted but not yet used.
    """

    tennis_type_set = {
        "ball_machine",
        "cardio",
        "drill",
        "hitting_session",
        "lesson",
        "match",
    }

    def __init__(
        self, start: datetime.datetime, duration: datetime.timedelta, **kwargs
    ):
        """Create a Tennis Activity.  This method requires start and duration
        arguments and will accept keyword arguments for all other Activity and
        Tennis attributes."""

        # Remove type and partner from the argument dictionary
        # and pass the remainder to Activity for handling.

        partner = kwargs.pop("partner", None)
        type = kwargs.pop("type", None)

        super().__init__(start, duration, **kwargs)

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

        if type and not isinstance(type, str):
            raise TypeError("type must be a string")
        if type and type not in Tennis.tennis_type_set:
            raise ValueError("invalid tennis type")

        self.partner = partner
        self.type = type


class Walk(Activity):
    """py_athletics Walk Activity subclass.  A Walk object may include all
    Activity attributes as well as distance, type, maximum_speed and
    average_speed attributes.  Walk type is a string and can be one of
    'track', 'road' or 'treadmill'.  The maximum_speed and average_speed
    attributes are expressed in minutes per mile.  The type attribute is
    accepted but not yet used.
    """

    walk_type_set = {"track", "road", "treadmill"}

    def __init__(
        self, start: datetime.datetime, duration: datetime.timedelta, **kwargs
    ):
```

```
        """Create a Walk Activity.  This method requires start and duration
        arguments and will accept keyword arguments for all other Activity and
        Walk attributes.
        """

        # Copy distance, type, maximum_speed and average_speed
        # from the argument dictionary and call Activity for handling.
        # Process these arguments after the Activity initialization is
        # complete.

        distance = kwargs.pop("distance", None)
        type = kwargs.pop("type", None)
        maximum_speed = kwargs.pop("maximum_speed", None)
        average_speed = kwargs.pop("average_speed", None)

        super().__init__(start, duration, **kwargs)

        if distance and not isinstance(distance, Decimal):
            raise TypeError("distance must be a Decimal")
        if distance and distance <= 0:
            raise ValueError("distance must be positive")

        if type and not isinstance(type, str):
            raise TypeError("type must be a string")
        if type and type not in Walk.walk_type_set:
            raise ValueError("invalid walk type")

        if maximum_speed and not isinstance(maximum_speed, datetime.time):
            raise TypeError("maximum speed must be time object")

        if average_speed and not isinstance(average_speed, datetime.time):
            raise TypeError("average  speed must be time object")

        self.distance = distance
        self.type = type
        self.maximum_speed = maximum_speed
        self.average_speed = average_speed

    def __repr__(self):

        # The Walk class adds distance, maximum_speed, and average_speed
        # to the detail to be included in the __repr__string.  The method
        # gets the common base string from the parent class and then adds
        # the additional detail before the closing bracket.

        super_class_str = super().__repr__()

        # Transform None and 0 to "--" for optional fields.
        # Since "if x" will not be true if x is None or x is 0, we
        # can set the field to "--" then test on x and adjust.
        # Speed measures are in time, representing minutes per to cover
        # a mile

        distance_token = "--"
        if self.distance:
            distance_token = self.distance

        max_speed_token = "--"
        if self.maximum_speed:
            max_speed_token = self.maximum_speed.strftime("%M:%S")

        avg_speed_token = "--"
        if self.average_speed:
            avg_speed_token = self.average_speed.strftime("%M:%S")

        f_1 = f"Distance (miles): {distance_token} "
```

```
        f_2 = f"Max Speed (minutes/mile): {max_speed_token} "
        f_3 = f"Avg Speed (minutes/mile): {avg_speed_token}"

        addendum_str = " " + f_1 + f_2 + f_3

        return super_class_str[:-1] + addendum_str + super_class_str[-1:]


class Workout(Activity):
    """py_Athletics Workout Activity subclass.  A Workout object may
    include all Activity attributes as well as type and trainer attributes.
    The type attribute is a string and can be either "personal_training" or
    "solo".  The trainer attribute is a string.  The type and trainer
    attributes are accepted but not yet used.
    """

    workout_type_set = {"personal_training", "solo"}

    def __init__(
        self, start: datetime.datetime, duration: datetime.timedelta, **kwargs
    ):
        """Create a Workout Activity.  This method requires start and duration
        arguments and will accept keyword arguments for all other Activity and
        Workout attributes."""

        # Remove type and trainer from the argument dictionary and pass the
        # remainder to Activity for handling.

        trainer = kwargs.pop("trainer", None)
        type = kwargs.pop("type", None)

        super().__init__(start, duration, **kwargs)

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

        if type and not isinstance(type, str):
            raise TypeError("type must be a string")
        if type and type not in Workout.workout_type_set:
            raise ValueError("invalid workout type")

        self.trainer = trainer
        self.type = type
```