

ADVANCED COURSE IN PROGRAMMING, AUTUMN 2025, ONLINE EXAM 3

⌚ STARTED: 12:57 PM

ENDS: 4:57 PM

3:52

Exercise 1

Complete this in exercise template `exercise1.py`

Copy and paste the following class to the exercise template:

```
from math import sqrt

class Point:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y

    def distance_from_origo(self):
        return sqrt(self.x ** 2 + self.y ** 2)

    def __str__(self):
        return f"Point(x={self.x}, y={self.y})"
```

Write a class named `ComparablePoint` which inherits the class `Point`

The derived class must have following new traits:

Implementation for the operator `+`: the points are added together by summing both the x and y coordinates. The sum of the operation should be a `ComparablePoint` object.

Implementation for the operator `==`: two points are equal, if the x- and y-coordinate values of both points are exactly the same.

Implementation for the operator `>`: point a is greater than point b, if the distance of a from the origo is greater than the distance of b from the origo.

Exercise 2

Complete this in exercise template `exercise2.py`

In this exercise, you need to create a class `Car` which can be used to model the costs associated with driving a car.

The `Car` class should have the following properties:

- The constructor takes three values: `brand` (string), `purchase_year` (integer), and `purchase_price` (integer).
- The method `set_year` takes an integer value as a parameter, allowing you to set a new year.
- The `Car`-class should contain a feature that links different `Car` objects based on their year. When new `Car` objects are created with a purchase year later than the current year of all cars, the year is updated in every `car` to match the new year. Similarly, when the `set_year`

method is used for one `Car` object to set a new year, the new year must be greater than the previous year, and it affects all `Car` objects. **Do not use a global variable.**

- The method `drive` takes two values: `distance_driven` (integer) and `cost_per_kilometer` (float). When this method is called, the program keeps track of the total kilometers driven by the car and the total costs.
- The method `add_expense` allows you to increase the car's expenses. The method takes one parameter, which is an integer.
- The method `distance_driven_by_car` does not take any value when called. This method returns the total distance traveled so far as an integer.
- The method `current_value` does not take any values when called. It calculates the current value of the car, which decreases by 15% each year. The `current_value` method returns an integer.
- The method `cost_per_kilometer` does not take any values when called. It combines all the costs incurred by the vehicle and divides them by the kilometers driven. Costs consist of costs incurred from driving, expenses added with the `add_expense` method, and the depreciation of the car's value. Use the `current_value` method to calculate the depreciation. The method's return value is an unrounded float.
- The class should also have a `__str__` method. The return value of this method is in the format: `<brand>: purchase year <purchase_year>, value <current value>`.

Example use of the class:

```
toyota = Car("Toyota", 2020, 10000)
print(toyota)

toyota.drive(100, 0.10)
print(f"Distance driven with Toyota: {toyota.distance_driven_by_car()}")
toyota.set_year(2021)
print(f"Value of Toyota in 2021: {toyota.current_value()}")
print(toyota)
print(f"Cost per kilometer for Toyota in 2021: {toyota.cost_per_kilometer()}")
toyota.set_year(2022)
print(f"Value of Toyota in 2022: {toyota.current_value()}")
print(f"Cost per kilometer for Toyota in 2022: {toyota.cost_per_kilometer()}")

bmw = Car("BMW", 2023, 20000)
print(f"Value of Toyota after purchasing BMW: {toyota.current_value()}")
bmw.drive(200, 0.12)
bmw.drive(300, 0.13)
print(f"Distance driven with BMW: {bmw.distance_driven_by_car()}")
print(f"Cost per kilometer for BMW in 2023: {bmw.cost_per_kilometer()}")
bmw.add_expense(1000)
print(f"Cost per kilometer for BMW after a 1000 euro service: {bmw.cost_per_kilometer()}")
```

The program outputs:

```
Toyota: purchase year 2020, value 10000
Distance driven with Toyota: 100
Value of Toyota in 2021: 8500
Toyota: purchase year 2020, value 8500
Cost per kilometer for Toyota in 2021: 15.1
Value of Toyota in 2022: 7224
Value of Toyota after purchasing BMW: 6141
Distance driven (BMW): 500
Cost per kilometer for BMW in 2023: 0.126
Cost per Kilometer for BMW after a 1000 euro service: 2.126
```

Exercise 3

Complete this in exercise template `exercise3.py`

The task is to program a simple dice game.

Write a class called `Dice` with the following properties:

- A hidden or private attribute for the number of sides for the dice. If no number of sides is provided when creating an instance of `Dice`, it should default to a 6-sided dice.
- A method called `roll_dice`, which takes a parameter indicating how many times the die should be rolled. The method should return a list containing the results of the dice rolls.
- Method `__str__`, which returns information about the number of sides of the dice, as shown in the example below.
- Possible other methods or attributes are irrelevant to the user. If you create any, make them hidden or private.

Example of using the `Dice` class:

```
six_sided_dice = Dice()
eight_sided_dice = Dice(8)

print(six_sided_dice)
print(eight_sided_dice)

dice_roll = six_sided_dice.roll_dice(5)
print(dice_roll)

second_dice_roll = eight_sided_dice.roll_dice(2)
print(second_dice_roll)
```

Example output:

```
6-sided dice
8-sided dice
[1, 4, 5, 2, 4]
[8, 1]
```

Additionally, create a class called `DiceGame` with following properties:

- `DiceGame` takes a `Dice` object as an argument, which is used in the game. Changing dice in the middle of a game would be cheating, so make it private.
- `DiceGame` is played using 5 identical dice.
- A method `roll_once`, which prints the values of the dice in ascending order, as shown in the example below. If all the dice have the same value, print "Yatzy!" instead of the normal output.
- A method `roll_five_of_a_kind`, which prints the number of times the dice were rolled to get the same value for all dice, as shown in the example below. In the method, there is no need to consider the case where you get five of a kind on the first roll.
- Method `__str__` returns the rules of the game, as shown in the example below.
- Possible other methods or attributes are irrelevant to the user. If you create any, make them hidden or private.

Example of using `DiceGame` class:

```
six_sided_dice = Dice()
game = DiceGame(six_sided_dice)

print(game)
```

```
game.roll_once()  
game.roll_once()  
game.roll_once()  
game.roll_once()  
  
game.roll_five_of_a_kind()  
  
difficult_game = DiceGame(Dice(10))  
difficult_game.roll_five_of_a_kind()  
  
easy_game = DiceGame(Dice(1))  
easy_game.roll_once()  
easy_game.roll_once()  
easy_game.roll_once()  
easy_game.roll_once()
```

Example output:

```
The goal of the game is to roll the dice and get 5 of the same number. Using 6-sided dice.  
Rolled 5 dice and got 2, 2, 3, 4, 6.  
Rolled 5 dice and got 1, 1, 2, 5, 6.  
Yatzy!  
Rolled 5 dice and got 1, 2, 4, 6, 6.  
It took 886 rolls to get five of a kind.  
It took 4490 rolls to get five of a kind.  
Yatzy!  
Yatzy!  
Yatzy!  
Yatzy!
```

[END EXAM](#)

About

The University of Helsinki MOOC Center makes high-quality online education possible by developing and researching educational software and online learning materials. Teachers both within and without the University of Helsinki rely on our tools to create impactful teaching materials. Our popular Massive Open Online Courses (MOOCs) have been available through MOOC.fi since 2012.

This website is powered by an open source software developed by the University of Helsinki MOOC Center. Star the project on GitHub: [Project Github](#).

[Privacy](#)

[Accessibility statement](#)