Covers Lectures 6 & 7

Object Oriented Programming, Exercise 5.

Topics: Class attributes, class hierarchies, access modifiers (private and public), steps toward larger applications.

use Git. Make a Git commit at least after every coding task. Have each coding task in a separate file.

If you are unsure what to do try first with pseudocode, you get points also if you have written the program with pseudocode!

Code in Python3 and follow the style guide.

Deadline: 13 March 2025

1. Postcodes

The program contains the class definition City, which is a model of a single city.

Please add a class variable named postcodes which refers to a dictionary. The keys of the dictionary are the names of cities, and the values attached are the postcodes for those cities. Both are strings.

The dictionary should contain at least the following postcodes:

- Helsinki 00100
- Turku 20100
- Tampere 33100
- Salo 24100
- Oulu 90100
- Jyväskylä 40100

You do not need to implement any other functionality

```
class City:
    def __init__(self, nimi: str, population: int)
        self.__nimi = nimi
        self.__population = population
```

2. List helper

Please create a class named `ListHelper` which contains the following two class methods.

- `greatest_frequency(my_list: list)` returns the most common item on the list
- `doubles(my_list: list)` returns the number of unique items which appear at least twice on the list

It should be possible to use these methods without creating an instance of the class. An example of how the methods could be used:

```
numbers = [1, 1, 2, 1, 3, 3, 4, 5, 5, 5, 6, 5, 5, 5]
print(ListHelper.greatest_frequency(numbers))
print(ListHelper.doubles(numbers))
```

The program should be printing something like this:

5

3

3. Item, Suitcase and Cargo hold

In this series of exercises, you will create the classes Item, Suitcase and Cargo Hold, which will let you further practice working on objects which contain references to other objects.

Part 1: Item

Please create a class named `Item` which is used to create items of different kinds. Each item has a name and a weight (in grams).

You can use the following code to test your class:

```
book = Item("ABC Book", 200)
phone = Item("Nokia 3210", 100)

print("Name of the book:", book.name())
print("Weight of the book:", book.weight())

print("Book:", book)
print("Phone:", phone)
```

The program should print out this:

Name of the book: ABC Book

Weight of the book: 200

Book: ABC Book (200 g)

Phone: Nokia 3210 (100 g)

An `Item` should provide the methods `weight` and `name`, which return the values stored in those attributes.

The `name` and `weight` should be encapsulated within the class. The following code should not work:

```
book = Item("ABC Book", 200)
book.weight = 100
```

Part 2: Suitcase

Please write a class named `Suitcase`. You should be able to pack items into a suitcase. A suitcase also has a maximum combined weight for the items stored within.

Your class should contain the following members:

- a constructor which takes the maximum weight as an argument
- a method named `add_item` which adds the item given as an argument to the suitcase. The method has no return value.
- a `__str__` method which returns a string in the format "x items (y kg)"

The class should make sure that the combined weight of the items stored within any `Suitcase` does not exceed the maximum weight set for that instance. If the maximum weight would be exceeded when the `add_item` method is called, the new item should not be added to the suitcase.

Your class should work as follows:

```
book = Item("ABC Book", 200)
phone = Item("Nokia 3210", 100)
brick = Item("Brick", 400)

suitcase = Suitcase(500)
print(suitcase)

suitcase.add_item(book)
print(suitcase)

suitcase.add_item(phone)
print(suitcase)

suitcase.add_item(brick)
print(suitcase)
```

The program should have the following output:

0 items (0 g)

1 items (200 g)

2 items (300 g)

2 items (300 g)

Part 3: Mind your language:

The notification "1 items" is not very grammatical. Instead, it should say "1 item". Please make the required changes to your __str__ method.

The previous example should now print out (note the output when there is only one item):

0 items (0 g)

1 item (200 g)

2 items (300 g)

2 items (300 g)

Part 4: All the items:

Please add the following methods to your `Suitcase` class definition:

- `print_items` prints out all the items stored in the suitcase
- `weight` returns an integer number representing the combined weight of all the items stored in the suitcase

Your class should now work with the following program:

```
book = Item("ABC Book", 200)
```

```
phone = Item("Nokia 3210", 100)
brick = Item("Brick", 400)

suitcase = Suitcase(1000)
suitcase.add_item(book)
suitcase.add_item(phone)
suitcase.add_item(brick)

print("The suitcase contains the following items:")
suitcase.print_items()
combined_weight = suitcase.weight()
print(f"Combined weight: {combined_weight} g")
```

The suitcase contains the following items:

ABC Book (200 g)
Nokia 3210 (100 g)
Brick (400 g)
Combined weight: 700 g

If you have implemented your `Suitcase` class with more than two instance variables, please make the required changes so that each instance has only two data attributes: the maximum weight and a list of items within.

Part 5: The heaviest item

Please add a new method to your `Suitcase` class: `heaviest_item` should return the `Item` which is the heaviest. If there are two or more items with the same, heaviest weight, the method may return any one of these. The method should return a reference to an object. If the suitcase is empty, the method should return None.

Your class should now work with the following program:

```
book = Item("ABC Book", 200)
phone = Item("Nokia 3210", 100)
brick = Item("Brick", 400)

suitcase = Suitcase(1000)
suitcase.add_item(book)
suitcase.add_item(phone)
suitcase.add_item(brick)

heaviest = suitcase.heaviest_item()
print(f"The heaviest item: {heaviest}")
```

Executing the above program should print out this:
The heaviest item: Brick (400 g)

Part 6: Cargo hold

Note that cargo hold is handling the weight in kilos!

Please write a class named `CargoHold` with the following methods:

a constructor which takes the maximum weight as an argument

a method named `add_suitcase` which adds the suitcase given as an argument to the cargo hold

a `__str__` method which returns a string in the format "x suitcases, space for y kg"

The class should make sure that the combined weight of the items stored within any `CargoHold` does not exceed the maximum weight set for that instance. If the maximum weight would be exceeded when the `add_suitcase` method is called, the new suitcase should not be added to the cargo hold.

Your class should now work with the following program:

```
cargo_hold = CargoHold(100)
print(cargo_hold)

book = Item("ABC Book", 200)
phone = Item("Nokia 3210", 100)
brick = Item("Brick", 400)

adas_suitcase = Suitcase(1000)
adas_suitcase.add_item(book)
adas_suitcase.add_item(phone)

peters_suitcase = Suitcase(1000)
peters_suitcase.add_item(brick)

cargo_hold.add_suitcase(adas_suitcase)
print(cargo_hold)

cargo_hold.add_suitcase(peters_suitcase)
print(cargo_hold)
```

The output should look something like:
0 suitcases, space for 100 kg
1 suitcase, space for 99,7 kg
2 suitcases, space for 99,3 kg

Part 7: The contents of the cargo hold
Please add a method named print_items to your CargoHold class. It should print out all the items in all the suitcases within the cargo hold.

Your class should now work with the following program:

```
book = Item("ABC Book", 200)
phone = Item("Nokia 3210", 100)
brick = Item("Brick", 400)

adas_suitcase = Suitcase(1000)
adas_suitcase.add_item(book)
adas_suitcase.add_item(phone)

peters_suitcase = Suitcase(1000)
peters_suitcase.add_item(brick)
```

```
cargo_hold = CargoHold(100)
cargo_hold.add_suitcase(adas_suitcase)
cargo_hold.add_suitcase(peters_suitcase)

print("The suitcases in the cargo hold contain the following items:")
cargo_hold.print_items()
```

Executing the above program should print out this:
The suitcases in the cargo hold contain the following items:
ABC Book (200 g)
Nokia 3210 (100 g)
Brick (400 g)

4. Laptop computer

The exercise template contains a class definition for a Computer, which has the attributes model and speed.

Please define a class named LaptopComputer which inherits the class Computer. The constructor of the new class should take a third argument: weight, of type integer.

Please also include a __str__ method in your class definition. See the example below for the expected format of the string representation printed out.

```
laptop = LaptopComputer("NoteBook Pro15", 1500, 2)
print(laptop)
```

Output should be something like:

NoteBook Pro15, 1500 MHz, 2 kg

5. Game museum

The exercise template (available in Itslearning) contains class definitions for a `ComputerGame` and a `GameWarehouse`. A `GameWarehouse` object is used to store `ComputerGame` objects.

Please familiarize yourself with these classes. Then define a new class named `GameMuseum` which inherits the `GameWarehouse` class.

The `GameMuseum` class should override the `list_games()` method, so that it returns a list of only those games which were made before the year 1990.

The new class should also have a constructor which calls the constructor from the parent class `GameWarehouse`. The constructor takes no arguments.

You may use the following code to test your implementation:

```
museum = GameMuseum()
museum.add_game(ComputerGame("Pacman", "Namco", 1980))
museum.add_game(ComputerGame("GTA 2", "Rockstar", 1999))
museum.add_game(ComputerGame("Bubble Bobble", "Taito", 1986))
```

```
for game in museum.list_games():
    print(game.name)
```

The program should print out the following:

```
Pacman
Bubble Bobble
```

6. Word game

The exercise template contains the class definition for a `WordGame`. It provides some basic functionality or playing different word-based games:

```
import random

class WordGame():
    def __init__(self, rounds: int):
        self.wins1 = 0
        self.wins2 = 0
        self.rounds = rounds

    def round_winner(self, player1_word: str, player2_word: str):
        # determine a random winner
        return random.randint(1, 2)

    def play(self):
        print("Word game:")
        for i in range(1, self.rounds+1):
            print(f"round {i}")
            answer1 = input("player1: ")
            answer2 = input("player2: ")

            if self.round_winner(answer1, answer2) == 1:
                self.wins1 += 1
                print("player 1 won")
            elif self.round_winner(answer1, answer2) == 2:
                self.wins2 += 1
                print("player 2 won")
            else:
                pass # it's a tie

        print("game over, wins:")
        print(f"player 1: {self.wins1}")
        print(f"player 2: {self.wins2}")
```

The game is played as follows:

```
p = WordGame(3)
p.play()
```

```
Word game:
round 1
```

```
player1: longword
player2: ??
player 2 won
round 2
player1: i'm the best
player2: wut?
player 1 won
round 3
player1: who's gonna win
player2: me
player 1 won
game over, wins:
player 1: 2
player 2: 1
```

In this "basic" version of the game the winner is determined randomly. The input from the players has no effect on the result.

Part 1: Longest word wins

Longest word wins

Please define a class named LongestWord. It is a version of the game where whoever types in the longest word on each round wins.

The new version of the game is implemented by inheriting the WordGame class. The round_winner method should also be suitably overridden. The outline of the new class is as follows:

```
class LongestWord(WordGame):
    def __init__(self, rounds: int):
        super().__init__(rounds)

    def round_winner(self, player1_word: str, player2_word: str):
        # your code for determining the winner goes here
```

Now the games should work in the following way:

```
Word game:
round 1
player1: short
player2: longword
player 2 won
round 2
player1: word
player2: wut?
round 3
player1: i'm the best
player2: no, me
player 1 won
game over, wins:
player 1: 1
player 2: 1
```

Part 2: Most vowels wins

Please define another WordGame class named MostVowels. In this version of the game whoever has squeezed more vowels into their word wins the round.

Part 3: Rock, Paper, Scissors, Lizard, Spock

Similar to the traditional rock, paper, scissors game but two more variables:

- Scissors cuts Paper
- Paper covers Rock
- Rock crushes Lizard
- Lizard poisons Spock
- Spock smashes Scissors
- Scissors decapitates Lizard
- Lizard eats Paper
- Paper disproves Spock
- Spock vaporizes Rock
- (and as it always has) Rock crushes Scissors

If the input from either player is invalid, they lose the round. If both players type in something else than rock, paper, scissors, lizard or spock the result is a tie.

An example of how the game is played:

```
Word Game:
Round 1
Player 1: paper
Player 2: lizard
Player 2 wins
Round 2
Player 1: rock
Player 2: paper
Player 2 wins
Round 3
Player 1: spock
Player 2: scissors
Player 1 wins
Game over, wins:
Player 1: 1
Player 2: 2
```

7. UML Class Diagrams create base class and its sub-classes.

1. **Read through the Geeks for Geeks pages about UML class diagrams**:

   https://www.geeksforgeeks.org/unified-modeling-language-uml-class-diagrams/

   - Make sure you understand the basics of UML class diagrams, including classes, attributes, methods, inheritance, and relationships.

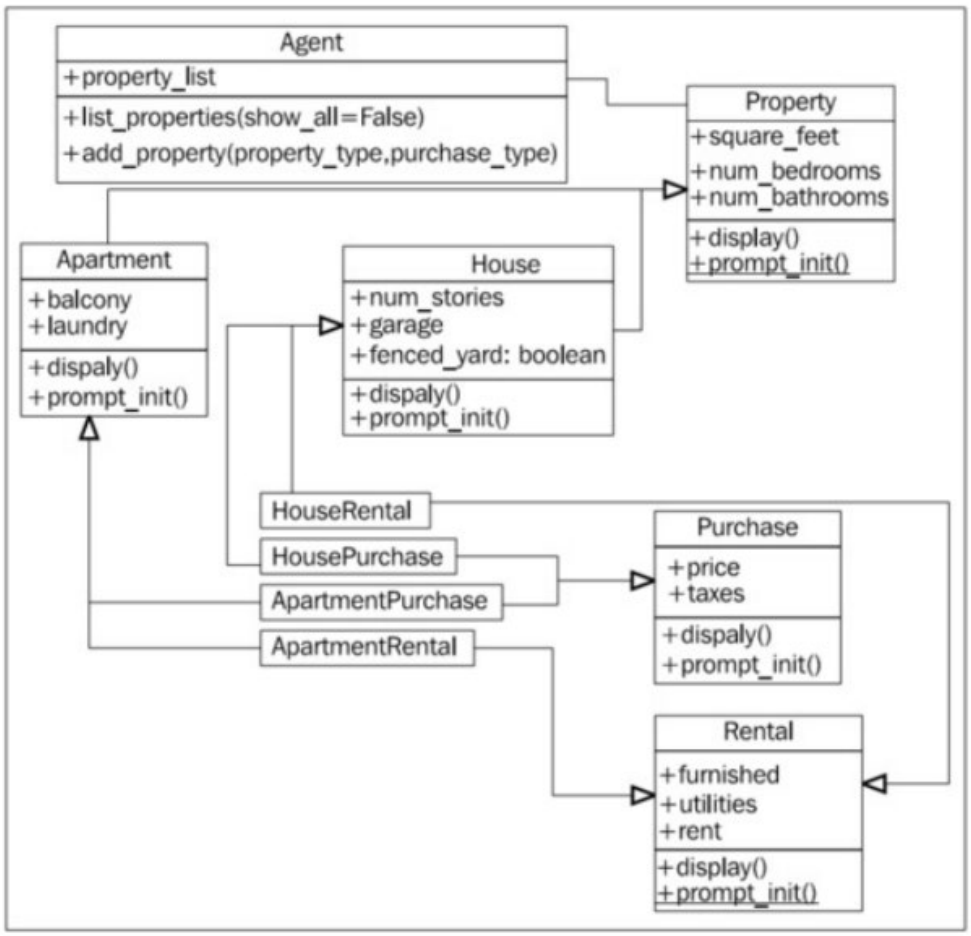
2. **Look around you and describe items in an inheritance hierarchy**:

   - Identify objects around you and think about how they can be categorized. Humans are really good at dividing the world into taxonomies, so this should not be too difficult.

3. **Model these objects as you would use them in a computer application**:
    - Determine the properties and methods that these objects would have. What properties and methods would they share? Think here our Person – Student – Teacher example that we went through during the lecture.
4. **Identify polymorphically overridden methods**:
    - Consider which methods would need to be overridden in subclasses. Here the Book Container example that we wen through during the class might be helpful.
5. **Identify completely different properties**:
    - Determine properties that are unique to each subclass.
6. **Use UML class diagram to visually represent the structure and relationships**:
    - Create a UML class diagram to show the inheritance hierarchy and relationships between the classes. You can use online tools like Lucidchart, draw.io, or even pen and paper.

Note you do not have to code anything, but if you do, write something similar with the audio_file.py that we went through during the lecture.

8. From class diagram to program (This exercise is two points, parts 1-3 one point and part 4 one point):



Part 1:

Write the class Property. the class has two methods, display prints out the information of the property and prompt_init method here is a class method which asks the user to enter the information of the property.

Part 2:

The Apartment class extends Property. The display and constructor methods call their respective parent class methods using super().

Again prompt_init is a class method, it uses the base class's prompt init to ask the basic information of the property and then adds a few questions related to Apartment class data attributes. Display method, similarly uses the base class and prints out also the information which is related to apartments

Part 3:

Continue in similar manner with House class

Part 4:

create also Purchase class and Rental class, which at the moment do not have to inherit any other classes.

NOTE it might be a good idea to use class attributes to handle the correct answers for the yes/no questions.

You do not need to handle any other errors (just make sure that the user answers either yes or no, if something else, the program should continue asking).

Here is the code for the different classes in the main function and below each part you should see how the console looks like after you ran each of the tests:

```
if __name__ == "__main__":
    property_details = Property.prompt_init()
    print(property_details)
    property = Property(**property_details)
    property.display()
```

```
Enter the square meter: 150
Enter number of bedrooms: 5
Enter number of baths: 3
{'square_meter': 150.0, 'beds': 5, 'baths': 3}
PROPERTY DETAILS
================
square meters: 150.00
bedrooms: 5
bathrooms: 3
```

```
    apartment_details = Apartment.prompt_init()
    print(apartment_details)
    apartment = Apartment(**apartment_details)
    apartment.display()
```

```
Enter the square meter: 85
Enter number of bedrooms: 2
Enter number of baths: 1
Does the apartment building have laundry?(yes/no) no
Does the property have balcony? (yes/no) yes
{'square_meter': 85.0, 'beds': 2, 'baths': 1, 'laundry': 'no', 'balcony': 'yes'}
PROPERTY DETAILS
================
square meters: 85.00
bedrooms: 2
bathrooms: 1
APARTMENT DETAILS
Laundry:  no
Balcony:  yes
```

```
    house_details = House.prompt_init()
    print(house_details)
    house = House(**house_details)
    house.display()
```

```
Enter the square meter: 250
Enter number of bedrooms: 4
Enter number of baths: 3
Is the yeard fenced? (yes/no) yes
Is there a garage? (attached/detached/none) attached
How many stories the house has? 2
{'square_meter': 250.0, 'beds': 4, 'baths': 3, 'fenced': 'yes', 'garage': 'attached', 'num_stories': '2'}
PROPERTY DETAILS
================
square meters: 250.00
bedrooms: 4
bathrooms: 3
HOUSE DETAILS
# of stories: 2
garage: attached
fenced yard: yes
```

```
    purchase_details = Purchase.prompt_init()
    print(purchase_details)
    purchase = Purchase(**purchase_details)
    purchase.display()
```

```
What is the selling price? 100000
What are the estimated taxes? 7500
{'price': 100000.0, 'taxes': 7500.0}
PURCHASE DETAILS
Selling price: 100000.00
Estimated taxes: 7500.00
```

```
    rental_details = Rental.prompt_init()
    print(rental_details)
    rental = Rental(**rental_details)
    rental.display()
```

```
What is the monthly rent? 850
What are the estimated utilities? 100
Is the property furnished? (yes/no)no
{'rent': 850.0, 'utilities': 100.0, 'furnished': 'no'}
RENTAL DETAILS
Rent: 850.00
Estimated utilities: 100.00
Furnished: no
```

9. Design a Library Catalog System

Step 1: Identify Library Items

Think about all the different types of items that library users can borrow from the library. Consider a variety of media and formats. List these items as part of your design process.

Step 2: Design the Class Diagram

1. **Base Class**: Create a base class that represents a general library item.
    - **Attributes**: Identify common attributes that all library items share (e.g., title, author, publication year, item ID).
    - **Methods**: Define methods that are common to all library items (e.g., displaying item information, borrowing an item, returning an item).
2. **Sub-Classes**: Create sub-classes for each specific type of library item you identified.
    - **Additional Attributes**: Identify attributes specific to each type of item.
    - **Methods**: Define any additional methods specific to each type of item.

Step 3: Define Basic Functions

Think about the basic functions needed for the library catalog to work properly:

- **Adding Items**: How will new items be added to the catalog?
- **Removing Items**: How will items be removed from the catalog?
- **Searching for Items**: How will users search for items in the catalog?
- **Borrowing Items**: How will users borrow items from the catalog?
- **Returning Items**: How will users return borrowed items to the catalog?

Step 4: Draw a Sequence Diagram

Create a sequence diagram to illustrate interactions between the user and the catalog system. Consider at least a few interactions, such as:

1. **Borrowing an Item**:
    - User searches for an item.
    - System displays search results.
    - User selects an item to borrow.
    - System checks item availability.
    - System updates item status to borrowed.
2. **Returning an Item**:
    - User returns an item.
    - System updates item status to available.

Resources

For more information on sequence diagrams, refer to this guide: https://www.geeksforgeeks.org/unified-modeling-language-uml-sequence-diagrams/