University of Zurich[UZH]

Department of Physics **Scientific Programming with Python**

OOP in Python Exercises

June 28, 2021

Exercise 1: Ducks [basic]

Look again at the slides on the strategy pattern and use the code examples to define the following duck-classes[1]:

- NormalDuck (use as base class)
- RedheadedDuck
- BlackDuck
- RubberDuck
- DecoyDuck

Once you defined these classes:

(a) Write a script in which you create the following duck-instances: 3 normal, 3 redheaded, 1 black, 1 rubber and 1 decoy duck. Store all ducks in a list and then call `display` for each of the ducks.

(b) In your script, let one of the normal ducks ,,break its wings". (Replace the `flying_behavoir` of the unlucky duck-object.) Compare what happens when you call the flying-functions of the unlucky duck and one of the other normal ducks.

(c) Change your duck classes such that you can give your individual ducks a name. Use that name when displaying the duck.

(d) *[intermediate]* Change your duck classes such that they store their position (as a string). Let `fly_to` change the position and display the present position on take off and landing.

(e) Add more functionality, be creative.

[1] You are of course free to use real duck breeds (see `https://en.wikipedia.org/wiki/List_of_duck_breeds`) if you prefer.

Exercise 2: Vectors [basic]

The file `vector.py` contains an implementation of an n-dimensional vector. Some of the functions are not yet complete:

(a) Implement the addition of two vectors via the magic function `__add__`. Make sure that the dimensions of the two vectors are aligned.

(b) Do the same for the scalar product with the function `__mul__`.

(c) Implement `__str__` which is the magic function to represent the vector as string (*i.e.* `str(v)`). Define a reasonable string representation.

(d) Create the property `length` that

- returns the Euclidiean length of the vector
- allows to scale the vector to a new length by `v.length = <new_length>`
- sets the vector to zero via `del v.length`.

(e) Create a subclass for three-dimensional vectors `Vector3D` with a suitable constructor and implement the magic function `__matmul__` (the operator `@`) as cross-product[2]. Important: The implementation should NOT lead to any change in the parent class.

Exercise 3: Scatter plot [basic]

Take pen and paper and design a class representing scatter plots that can be drawn.

- What variables does a scatter plot have?
- What methods does it have?
- How do the signatures of these methods look like?

Exercise 4: Understanding OOP [basic]

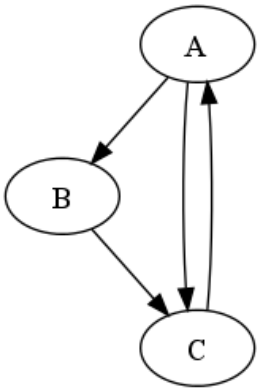
The `graph` module (provided in the archive) contains a set of classes for representing graphs (i.e. nodes and edges connecting them). On a piece of paper reverse engineer its design:

(a) Write down all class names, their methods and data attributes; try to understand what all of them do (read the documentation!).

(b) Figure out how different classes are related. Where is inheritance used, where composition? Draw a simple diagram.

[2]The cross-product is defined as $v = v_1 \times v_2 = \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \times \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} = \begin{pmatrix} y_1 z_2 - y_2 z_1 \\ z_1 x_2 - z_2 x_1 \\ x_1 y_2 - x_2 y_1 \end{pmatrix}$

(c) Use the classes to construct the following graph:



Exercise 5: Decorator Pattern [intermediate]

Look at the code in `starbuzz.py` and estimate the changes needed to add two more ingredients Cream and Sprinkles. Then adapt the code to use the Decorator Pattern.

(a) Remove all the ingredients-code from `Beverage`.

(b) Use the `Ingredient` class given in the slides to define Milk and Sugar ingredients.

(c) Improve and simplify your code further. For Example by moving as much as possible of the functionality from the subclasses into `Ingredient` (instead of repeating it).

(d) Define two more ingredients Cream and Sprinkles.

(e) Use the ingredients to produce new drinks combinations.

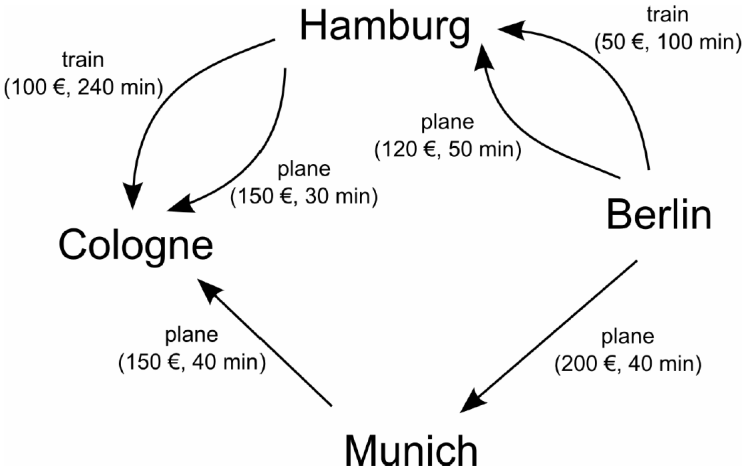
Exercise 6: Extending Classes [advanced]

Extend the `graph` library to solve a search problem. In this exercise, your goal is to write a travel planning application based on the `graph` module. We want to represent a set of cities as nodes in a graph, with edges between nodes representing different kinds of transportation.

(a) Define a class `CityNode` which extends the `Node` class by a new property `name` which is defined on class instantiation.

(b) Define a class `TransporationEdge` extending the `Edge` class. The edges should be directed and have two kinds of weights: travel `time` and `cost`. Furthermore, they should have a short `description` defining the means of transportation.

(c) Implement the following city graph as an example:



(d) Now we want to find the quickest path from Berlin to Cologne. Open the `shortest_path.py` file. It contains a `SearchAlgorithm` class, which implements the Dijkastra algorithm for finding the shortest path in a graph.

(e) Define a new class `SearchGraph` extending the Graph class with methods for searching the shortest path. Which design pattern(s) can you use in the example?

(f) Define new search algorithms to find the cheapest and fastest paths.

(g) Find the cheapest and fastest paths between Berlin and Cologne.

This exercise sheet is based on the exercises written by Bartosz Telenczuk, Niko Wilbert for the *Advanced Scientific Programming in Python School 2011*