University of Zurich[UZH]
Department of Physics

Scientific Programming with Python

OOP in Python Exercises

June 25, 2018

Exercise 0a: Ducks (20 min)

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

- normal duck
- redheaded duck
- black duck
- rubber duck
- decoy duck

Once you defined your classes:

(a) Create a group of 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) Let one of the normal ducks break its wings, make sure it will not be able to fly.

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

(d) 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.

Exercise 0b: Vectors (30 min)

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

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

(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 `__pow__` (the operator `**`) as cross-product[2]. Important: The implementation should NOT lead to any change in the parent class.

Exercise 0c: Scatter plot (15 min)

Take a 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 1: Understanding OOP (20 min)

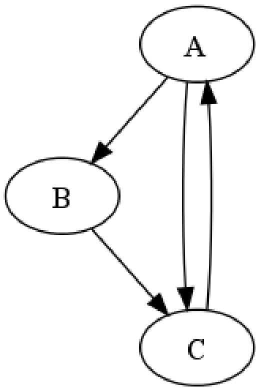
The `graph` module (provided in the archive) contains a set of classes for representing graphs. On a piece of paper reverse engineer its design:

(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 2: Decorator Pattern (30 min)

Modify the code in `starbuzz.py` to use the Decorator Pattern.

(a) Define a class `BeverageDecorator` which is instantiated with a beverage object and contains two methods: `get_cost` which adds the cost of the decorator to the total drink cost and `get_description` which updates the description of the drink. (You should be able to simplify the existing classes alot, when doing this.)

(b) Subclassing `BeverageDecorator` define new ingredients: Milk and Cream. Use the ingredients to produce new drinks combinations.

Exercise 3: Iterator Pattern (30 min)

The iterator pattern allows to build classes in Python that have instances that you can loop over *i.e.* with `for o in obj`. Implement such a Python iterator which iterates over string characters (ASCII only) returning their ASCII code (obtained by `ord` function):

(a) Define a new iterator class which contains two methods:

- `__init__` – a constructor taking the ASCII string as a argument
- `__next__` – returns the ASCII code of the next character or raises a `StopIteration` exception if the string end was encountered.

(b) Define a new iterable class which wraps around a string and contains `__iter__` method which returns the iterator instance.

(c) Test your code using a for loop.

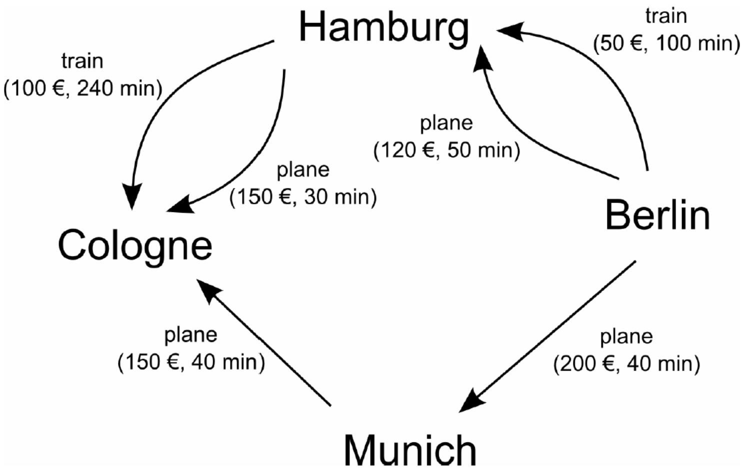
Exercise 4: Extending Classes (55 min)

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 `Node` class by a new property `name` which is defined on class instantiation.

(b) Define a class `TransporationEdge` extending `Edge` class. The edges should be directed and have two kinds of weights: travel `time` and `cost` and 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 from Berlin to Cologne. Open `shortest_path.py` file. It contains `SearchAlgorithm` class, which implements Dijkastra algorithm for finding the shortest path in a graph.

(e) Define a new class `SearchGraph` extending Graph class with methods for searching for the shortest path. Which design pattern 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*