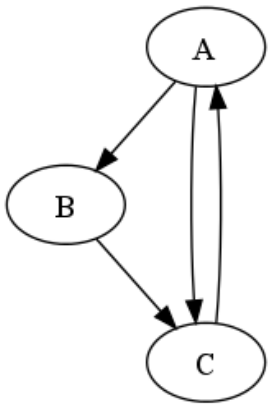
OOP and Design patterns Exercises

1. (20 min) The `graph` module (provided on the Wiki page) 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 public properties; try to understand what all of them do.

   (b) Figure out how different classes are depending on each other.

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



   (d) What are the weaknesses of this design? The code contains comments that raise some questions about the design, think about these.

   (e) How would you improve this design? How might an alternative design look like? We will come back to this later, so don't spend too much time on it now.

2. (40 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.

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

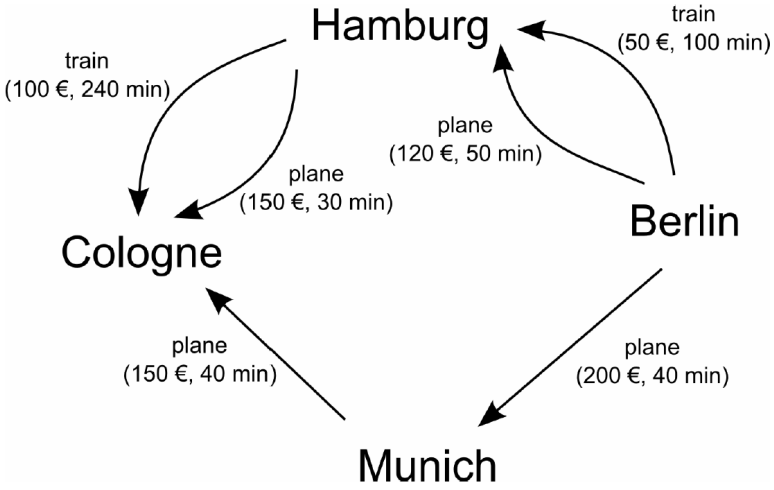
   (c) (*Optional*) Write unittests for your code and add propper docstrings.

3. (60 min) In this exercise, we want to solve a travel planning problem 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. Cities must have a name assigned to them. Transportation edges have three properties (in addition to connecting two cities): the travel time, the cost and the kind of transportation (e.g., a string "train").

   (a) Think about how you can extend the original design to accomodate the new requirements. Can you come up with a better design? For example, the popular NetworkX graph library for Python uses quite a different design approach (you can look at its online documentation for inspiration).

   (b) Change the design as you like (refactor) and implement the new requirements. If you prefer to stick with the old design you are free to do that as well (e.g., deriving a `CityNode` and `TransporationEdge` class). Optionally update the unittests as well (unittests will be discussed later in this school, so you can skip this step for now).

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



   (d) Now find the quickest path from Berlin to Cologne, using the algorithm that is already provided in the `graph` module.

   (e) Provide the quickest path for alternative weight functions (hint: maybe get rid of inheritance being used to define the weight function). Use this to find the the cheapest and fastest paths between Berlin and Cologne.

   (f) Turn your graph into an iterable (providing an `__iter__` method).