Info1 Midterm Prep

Carol V. Alexandru-Funakoshi

October 31, 2023

1 About the Midterm

You should all have received an Email with the subject *Information on the Examination ``Informatics I'' (Midterm on Monday, November 6).* Please study the attached PDFs carefully! Here's a reminder for the most important things -- but don't rely only on what is said here:

1.0.1 When, where and how?

- **Monday**, 6th of November, 2023
- You must come to the room assigned to you (see AINF02_Raumzuteilung.pdf). Don't show up in the wrong room!
- You must be there by 18:00
- The exam starts at 18:30 and ends at 19:30. You cannot leave early.
- You must bring your own laptop to work on the exam.

1.0.2 What is allowed?

- The exam is basically ``open book''. You can search the web, look up things in the lecture scripts, read documentation, etc.
- You can re-use code you find online (although it's unlikely this will help you much).
- You can (and probably should) use an IDE to work on the tasks.

1.0.3 What is not allowed?

- **Any sort of communication with others is strictly forbidden.** You're not allowed to ask for help on StackOverflow, Discord or any other communication media. You're not allowed to chat with your colleagues or anyone else.
- **The use of Generative AI is strictly forbidden.** No ChatGPT, GitHub Copilot, or any other LLM-based code assistants are permitted. Basic code completion

We will be monitoring your screens from the rear. Be aware that the UZH takes transgressions against these rules very seriously.

1.0.4 What's in the exam?

- Basically everything discussed in the first 6 weeks

1.0.5 What will the exam look like?

- roughly ``1 point per minute''

- 3 Kprim questions (2 points each)
- 6 programming tasks (6, 6, 7, 8, 12 and 15 points)
- You can navigate the entire exam from the beginning and you can jump back and forth between tasks, as long as you don't end the exam.

Mange your time!

1.0.6 How to work on programming tasks

1. Read the task description carefully, including the template, example usage and expected output. **All parts may be necessary to fully specify the requirements!**
2. Copy the code template to your programming environment (e.g., IDE) and implement a solution.
3. Make sure to test your solution thoroughly using the provided examples and additional calls. Even if your code produces the same output as the examples, that doesn't necessarily mean it's completely correct.
4. Copy your solution to the exam website. Include everything necessary to execute your solution. For example, if you need to import something from `math` or `random`, **make absolutely sure to include those imports!** To reduce the risk of fundamental errors when attempting to run your solution, do not include any code that is not necessary (such as example calls or other irrelevant code).
5. Go to the next question.

2 Classes and inheritance revisited

Last week you saw how inheritance can be used to build hierarchies of concepts. Super-classes define common structure, attribute or behavior, while sub-classes implement more specific features. Here's how this manifested for last week's example about Shapes:

The superclass `Shape`… * defines a constructor and attributes for x/y coordinates and color * defines a method `calculate_area` without any implementation

The subclasses `Circle` and `Rectangle`… * define additional constructor parameters and attributes for shape-specific things (i.e., radius and width/length) * provide the concrete implementations for `calculate_area`

In the `Circle` and `Rectangle` classes, the constructor (`__init__`) must call the super-constructor (`super().__init__`) such that attributes defined there are set properly.

2.0.1 Abstract classes and methods

On ACCESS, you also learned about ``abstract'' classes and methods.

- An *abstract method* is simply a method without an implementation, which sub-classes are supposed to implement. Such a method has no body (just `pass`) and is annotated with `@abstractmethod`
- An *abstract class* is a class with at least one abstract method. Such a class should inherit from `ABC`
- Abstract classes cannot be instantiated. That means, you cannot create objects of that class. You can only define subclasses and if those provide implementations for all abstract methods, then those can be instantiated.

Let's update last weeks example with *abstract*:

```
[1]: from math import pi

     class Shape():
         def __init__(self, cord_x, cord_y, color):
             self.cord_x = cord_x
             self.cord_y = cord_y
             self.color = color

         def calculate_area(self):
             pass

     class Circle(Shape):
         def __init__(self, cord_x, cord_y, color, radius):
             super().__init__(cord_x, cord_y, color)
             self.radius = radius

         def calculate_area(self):
             return pi * self.radius**2

     class Rectangle(Shape):
         def __init__(self, cord_x, cord_y, color, width, length):
             super().__init__(cord_x, cord_y, color)
             self.width = width
             self.length=length

         def calculate_area(self):
             return self.width * self.length
```

```
[2]: s = Shape(0, 0, "red")                  # impossible to create an instance of␣
      ↪Shape, because it has an abstract method
     c = Circle(10, 10, "red", 10)
     r = Rectangle(5, 5, "black", 500, 200)
     print(c.calculate_area())
     print(r.calculate_area())
```

```
314.1592653589793
100000
```

3 How to prepare for the exam?

Just one word: **practice**

It doesn't matter how you do it, but you will need to invest **time**. It's the only way to become a proficient programmer. You could...

- Solve old exam tasks (see the OLAT materials folder)

- Modify and extend tasks from ACCESS by yourself. For example, you could invert the requirements for many existing tasks.
- Just come up with arbitrary challenges for yourself. It's probabl the best way to practice.

3.0.1 Coming up with arbitrary tasks to practice on

- Pick any area that you're familiar with, for example a hobby (let's say: sports)
- Consider what data exists in this area (teams, team members, points, matches, referees, rules…)
- Consider how to represent such data (dictionaries, lists, sets)
- Consider what functionlity exists (add a point, add a member, create a match between two teams, declare a winner, …)
- Implement functions and/or classes to represent these concepts and behavior to write actual programs that model them

```
[ ]:
```