# Inheritance, Overriding and Polymorphism

# GRA-4152 OOP with Python

## Animal hierarchy

I have provided comments and docstrings to each class and method in the code. In the last programming exercise, I provided documentation like the book does it. However, I have come to believe that this is not the optimal way to write Python documentation. I would instead want to follow the official Python style guide, which is what I have done this time.

I included a simple describe() method for the classes and added two instance variables. The method provides a good example of inheritance.

For number two, here is an explanation of how I have used the three concepts in my answer:

### Inheritance

I've employed the concept of inheritance to create a class hierarchy where Cat and Dog are derived from the Animal class, and BigDog further inherits from the Dog class. By inheriting from Animal, these subclasses gain access to its methods, which means they automatically include the describe() method. In this case, I have made the instance variables name and age private, meaning the subclasses cannot directly access these variables from the Animal class. Furthermore, BigDog is a subclass of Dog, which means it inherits properties and behaviours from both Animal and Dog, an example of a multi-level inheritance structure.

### Overriding

I've overridden the greets() method in each subclass of Animal. Although Animal provides a template for the greets() method (it is an abstract method in Animal where I have chosen to raise a NotImplementedError. An alternative would be to use the @abstractmethod decorator from the abc module), I've replaced this with specific implementations in each subclass to reflect the unique way each animal greets: Cat outputs "meow", Dog outputs "woof", and BigDog outputs both "woof" and "woooof". I interpreted the diagram for BigDog such that the greets() method for BigDog first calls the greets() method of its superclass, i.e., Dog, and then extends the method by printing "woooof". Additionally, I've overridden the describe() method in each subclass to extend the description of the animal, which shows that I can not only replace but also extend the functionality of methods in the superclass.

### Polymorphism

I've demonstrated polymorphism through the use of the greets() and describe() methods. These methods behave differently depending on the type of object that invokes them, even though the method call looks the same. For example, when I call greets() on an instance of Cat, it results in "meow", while the same method call on an instance of Dog results in "woof". This shows that the same interface (greets() method) has multiple (poly) forms (morphs) of implementation depending on the object. This allows for flexibility in the code; I can add new animal types without changing the way existing classes or the code that uses them works.

### Simple test program

At the end of the animal\_hierarchy.py file I wrote a simple test program that tests the methods and demonstrates the three concepts above:

A screen shot of a computer program

Description automatically generated

A screenshot of a computer screen

Description automatically generated

## Programming Exercise: Business P10.24

I decided to add a little bit to the exercise by adding a timestamp of when the appointment was created. I do this with the in-built *datetime* module. This gets saved together with the appointment details when the user wants to save the appointments to a file. This way, the Appointment class can have a load() method, which P10.24 asks for. Since all the other appointment details are needed to create an instance of any of the appointment objects, a load method is not needed. In my case, this method just overrides the timestamp of the object to what is stored in the file.

The test file essentially does what the exercises in the book ask for; the user can specify what actions he/she wants to take. The options include adding appointments (to a list), checking which appointments (if any) happen on a specified date, saving the added appointments to a file of choosing and loading appointments from a file of choosing. I also added the option to print all added/loaded appointments. When adding a OneTime appointment, the code checks that the date is a valid date, and for a Monthly appointment that the day is between 1 and 31. When loading appointments from a file, the appointments get parsed line by line. On each line, I check for the occurrence of “One”, “Daily”, or “Monthly” to determine which type it is. The descriptions of the appointments are encased in single quotation marks, so I find the index of these to extract the descriptions.

The exercise steps:

1. Docstrings are provided for all classes and their methods.
2. To be able to use a full inheritance approach and still save different details depending on the type of appointment, I needed to include two instance variables in (almost) all the classes. I called them “\_type” and “\_strspecified\_time”. This way, when the save() method is called on, e.g., a OneTime object, the method itself is inherited from the Appointment class but, since the instance variables are redeclared in the OneTime’s constructor, “self.\_type” and “self.\_strspecified\_time” refers to OneTime’s instance variables. A Daily appointment object does not need to be saved with a specific time (as the appointment happens every day) so “\_strspecified\_time” does not need to be redeclared in its constructor (defaults to None from the Appointment’s constructor).
3. I decided to create the occursOn() an abstract method in the Appointment class, forcing its subclasses to define it to be able to use it. It is a very simple method; hence, it does not need to be further developed by the subclasses by first calling the superclass method and then doing something different. For OneTime appointments, it just checks whether the provided year, month and day provided match the object’s. For Daily appointments, the method returns True no matter what, as the appointment happens every day. For Monthly appointments, the method just checks if the day matches.
4. The \_\_repr\_\_ method is defined in each of the three subclasses, overriding the in-build method. I decided to return the text representation of the month. This means, e.g., print(one\_time\_appointment) could return something like: “One time appointment 9 November 2023 ‘Finish the second programming exercise’”. For Monthly I decided to check whether the given day is before or after the day saved in the object’s timestamp. If it is before, the monthly appointment will start in the month after as that day in the current month has passed.
5. The first time around, I created a save method in each of the three subclasses. The method would just call the superclass save method with the parameters that I later turned into instance variables. Since the save method is declared in every subclass, I realised that this technically would become a slight polymorphic approach. Generally, polymorphism refers to the ability of different objects to respond to the same method call in different ways. A clearer polymorphic example, in this case, would be to declare the save methods fully in each of the three subclasses such that the method would behave differently depending on which type of appointment objects the method is called on. In my code now, I declare instance variables that make the “output” of the save method differ, but the method itself is fully inherited from the Appointment class.