COMP11124 Object Oriented Programming

Inheritance and Polymorphism

Learning Outcomes

In this lab, you will learn about inheritance and polymorphism in Python. You will learn how to create subclasses and how to override methods. We will also look at the **`super()`** function and how to use it to call the parent class methods. After you have completed this lab, you should be able to:

- critically evaluate the use of inheritance and polymorphism
- demonstrate the use of inheritance and polymorphism in Python
- understand how we use inheritance and polymorphism in OOP

Topics Covered

Python: inheritance, polymorphism**, `super()`** function

Getting Started Task: You should have completed the ToDoApp from last week. If you have not, please complete it now. You will need it for this week's lab. Copy and paste it (so that you have some indication of the progress you have made) and then rename the folder to ToDoAppWeek5.
There are some exercises that you should use a new **`lab_week5.py`** file for. In others, you will use the ToDoAppWeek5 folder. The exercises will tell you where to use which file/folder.

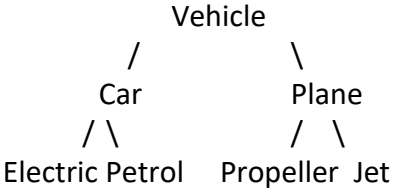
IMPORTANT NOTE: Last week we worked on the ToDo app and have included type hints and docstrings. The first part of this omits this in its examples for brevity. However, this does not mean that you should just forget about type hints and docstrings. Try to change any code that you add to any of the files to include type hints and docstrings. This will help you to get into the habit of adding them to your code.

1. Inheritance

Let us quickly recap the lecture content. Inheritance is a way to create new classes from existing classes; We are trying to model an "is-a" relationship. For example, a Plane is a Vehicle. We can therefore create a Plane class that inherits from the Vehicle class.

In Python, every class technically inherits from the object class (although we do not need to specify this). There are several reasons why we would want to use inheritance.

1) The major reason is that it allows us to reuse code. For each child class that we create, we do not need to redefine the methods and attributes that are already defined in the parent class. This saves you from possibly rewriting the same code multiple times.
2) It allows us to create a hierarchy of classes. This makes it easier to understand the code and allows us to create more complex programs. In the previous example we could have a hierarchy like this:



All vehicles have certain attributes and methods that are shared, for example, colour, weight, max speed, etc. However, each child class may have additional attributes and methods that are specific to that child class. For example, a plane has a wingspan, but a car does not.

3) It enables a concept called overriding. This means that we can change the functionality of a method in a child class. For example, we may have a method called move in the Vehicle class. This method may simply print "The vehicle is moving". However, we want to be more specific for each child class. For example, for a car we may want to print "The car is driving" and for a plane, we may want to print "The plane is flying". In overriding we simply recreate the method in the child-class and change the functionality.

Ok, enough theory, let us get started with some exercises. We will start with a simple example and then move on to the ToDoApp.

Exercise 1: Simple Inheritance

For this part, use the **`lab_week5.py`** file.

Let us assume the example from above. We are going to create this hierarchy.
First, we need to create our base **`Vehicle`** class. This class will contain all the attributes and methods that are shared by all vehicles.

```
class Vehicle:
    def __init__(self, colour, weight, max_speed):
        self.colour = colour
        self.weight = weight
        self.max_speed = max_speed

    def move(self, speed):
        print(f"The vehicle is moving at {speed} km/h")
```

We can see that each vehicle has a **`colour`**, **`weight`** and **`max_speed`**. This is quite simplistic, and we could add more attributes and methods. However, for the sake of this lab, we will stick with these three attributes and one method.

Now let us create our cars. We need to create a class called **`Car`** that inherits from the **`Vehicle`** class. The syntax to inherit from a class is as follows:

`class NewClassName(ParentClassName)`:

So, for the **`Car`**, we can do this as follows:

```
class Car(Vehicle):

    def move(self, speed):
        print(f"The car is driving at {speed} km/h")
```

Here, we have specified that the **`Car`** class inherits from the **`Vehicle`** class, which means all attributes will also be available in the **`Car`** class. We have also overridden the **`move`** method. This means that the **`move`** method in the **`Car`** class will be used instead of the **`move`** method in the **`Vehicle`** class.

Let us test this by creating two separate objects, one of the **Vehicle** class and one of the **Car** class.

```
generic_vehicle = Vehicle("red", 1000, 200)
generic_vehicle.move(100)

car = Car("blue", 1500, 250)
car.move(150)
```

As you will see when you run the code above, the **move** methods both print different messages.

Ok, so what if we wanted to add a new attribute to the **Car** class? For example, we may want to add the form factor, such as Hatchback or SUV. We can do this by adding the **__init__** method to the **Car** class and adding the **form_factor** attribute (you may need to replace the code above with the code below):

```
class Car(Vehicle):
    def __init__(self, colour, weight, max_speed, form_factor):
        self.colour = colour
        self.weight = weight
        self.max_speed = max_speed
        self.form_factor = form_factor

    def move(self, speed):
        print(f"The car is driving at {speed} km/h")
```

If you now change the code that creates the car object to include the **form_factor** parameter:

```
car = Car("blue", 1500, 250, "SUV")
car.move(150)
```

everything should work as expected. However, there is one caveat. We have now duplicated the code that is in the **__init__** method of the **Vehicle** class. (**self.colour = colour, self.weight = weight**, etc…)

Exercise 2: Super() function

Whilst the previous bit of code changes only a few lines of code, this can become a problem if you have a lot of attributes in the **Vehicle** class.
We can avoid this by using the **super()** function. This function allows us to call the **__init__** method of the parent class, which essentially initialises the attributes of the parent class from within the child class (replace the **__init__** code again):

```
class Car(Vehicle):
    def __init__(self, colour, weight, max_speed, form_factor):
        super().__init__(colour, weight, max_speed)
        self.form_factor = form_factor
```

See? We have now reduced the amount of code we need to write in the subclass by quite a bit.
Look at this link until Example 2 to see some other examples of how to use the **super()** function: https://www.programiz.com/python-programming/methods/built-in/super

In its core essence, **`super()`** is similar to calling the parent class directly. For example, we could have also written the code above as follows (do not copy)

```
class Car(Vehicle):
    def __init__(self, colour, weight, max_speed, form_factor):
        super().__init__(colour, weight, max_speed)
        self.form_factor = form_factor
```

but the **`super()`** function is more flexible.

After we have initialized our parent class, we have the code that is more specific to the child class. In this case, we have added the **`form_factor`** attribute.

Let's take this example a few steps further: by adding the **`ElectricCar`** class and the **`PetrolCar`** class. Both classes inherit from the **`Car`** class.

```
class Electric(Car):
    def __init__(self, colour, weight, max_speed, form_factor, battery_capacity):
        super().__init__(colour, weight, max_speed, form_factor)
        self.battery_capacity = battery_capacity

    def move(self, speed):
        print(f"The electric car is driving at {speed} km/h")

class Petrol(Car):
    def __init__(self, colour, weight, max_speed, form_factor, fuel_capacity):
        super().__init__(colour, weight, max_speed, form_factor)
        self.fuel_capacity = fuel_capacity

    def move(self, speed):
        print(f"The petrol car is driving at {speed} km/h")
```

Similarly to the **`Car`** class, we have added the **`__init__`** method to both the **`Electric`** and **`Petrol`** classes. We have also added the **`move`** method to both classes. As the hierarchy gets deeper, we are becoming more and more specific.

Let us test this by creating some objects:

```
electric_car = Electric("green", 1200, 200, "Hatchback", 100)
electric_car.move(100)

petrol_car = Petrol("red", 1500, 250, "SUV", 50)
petrol_car.move(150)

generic_vehicle = Vehicle("red", 1000, 200)
generic_vehicle.move(100)
```

You will see, as before, that the **`move`** methods print different messages.

To demonstrate the benefit of inheritance, let us assume that we want to add a new attribute to the **`Electric`** and **`Petrol`** classes: the maximum range. We can do this by simply adding the attribute to the **`__init__`** method of both classes but this would just duplicate code.

Instead, let us think about the hierarchy in general: do only cars have a maximum range? No, all vehicles have a maximum range. Therefore, we should add the maximum range attribute to the **`Vehicle`** class. This way, all child classes will also have access to this attribute.

Additionally, to allow for backwards compatibility, we want to make this attribute optional. Whether one supplies a maximum range or not should be up to the user of the program.
So, modify the **`Vehicle`** class as follows:

```
class Vehicle:
    def __init__(self, colour, weight, max_speed, max_range=None):
        self.colour = colour
        self.weight = weight
        self.max_speed = max_speed
        self.max_range = max_range
```

To now allow access to this max range attribute, we need to modify the **`__init__`** methods of all child classes to include the **`max_range`** parameter.

```
class Car(Vehicle):
    def __init__(self, colour, weight, max_speed, form_factor, max_range=None):
        super().__init__(colour, weight, max_speed, max_range)
        self.form_factor = form_factor
```

...

```
class Electric(Car):
    def __init__(self, colour, weight, max_speed, form_factor, battery_capacity, max_range=None):
        super().__init__(colour, weight, max_speed, form_factor, max_range)
        self.battery_capacity = battery_capacity
```

...

```
class Petrol(Car):
    def __init__(self, colour, weight, max_speed, form_factor, fuel_capacity, max_range=None):
        super().__init__(colour, weight, max_speed, form_factor, max_range)
        self.fuel_capacity = fuel_capacity
```

As you can see, this incurs quite a lot of code changes. Before we explore a better way to do this, let us test whether everything still works as expected.

Task: Change the **`move`** method of the electric and petrol cars to include the maximum range of the vehicle. For example, the **`move`** method of the electric car could print "The electric car is driving at 100 km/h and has a maximum range of 100 km".
Then modify the code that creates the electric and petrol car objects to include the maximum range parameter. Run the code and see if your changes work as expected.

Exercise 3: **kwargs

Luckily, there is a better way to add the **max_range** and any further attributes to the **Vehicle** class. We can use the ****kwargs** parameter. This allows us to pass any number of keyword arguments to the **__init__** method. These keyword arguments are then stored in a dictionary data structure (which we will explore later in this module).
What we need to do to enable this functionality is to add ****kwargs** as a parameter to any child class that is derived from the **Vehicle** class. We can then pass the keyword arguments to the **super()** function.

Let us modify the **Car** class as follows:

```
class Car(Vehicle):
    def __init__(self, colour, weight, max_speed, form_factor, **kwargs):
        super().__init__(colour, weight, max_speed, **kwargs)
        self.form_factor = form_factor
```

In very simple terms, what this now does is take any additional keyword arguments that are passed to the **Car** class and pass them to the **Vehicle** class via the **super()** function.

Task: Modify the **Electric** and **Petrol** classes to include ****kwargs** in the **__init__** method and subsequently pass the keyword arguments to the **Vehicle** class via the **super()** function. Then run the code with the objects that you have created previously and see whether everything still works as expected.

To show the benefits of this approach, let us add another attribute to the vehicle class: the number of seats. We can do this by simply adding the **seat** parameter to the **__init__** method of the **Vehicle** class. But now, we do not need to change any of the child classes. We can simply create a new object and pass the **seat** parameter to the **Car**, **Electric** or **Petrol** classes.

```
class Vehicle:
    def __init__(self, colour, weight, max_speed, max_range=None, seats=None):
        self.colour = colour
        self.weight = weight
        self.max_speed = max_speed
        self.max_range = max_range
        self.seats = seats
```

Then, we can test whether this works as expected by creating an object:

```
generic_electric_car = Electric("red", 1000, 200, "SUV", 100, max_range=500, seats=5)
generic_electric_car.move(100)
print(generic_electric_car.seats)
```

and et voila, we have added and tested a new attribute to the Vehicle class without having to change any of the child classes.

Note: We can not only use keyword arguments (e.g. argument=some value when creating an object), but also positional arguments (the ones without the = sign such as colour, weight ...). It works the same way

but would add too much content to this lab. If you are interested, you can read more about it here: https://www.digitalocean.com/community/tutorials/how-to-use-args-and-kwargs-in-python-3

Task: Change the code so that it now also includes the child class of **`Vehicle`**: **`Plane`** with its additional child classes **`Propeller`** and **`Jet`**. Ensure that the **`move`** method of each class derived from **`Plane`** says "Fly" instead of "Drive". Additionally, add a **`wingspan`** attribute to the **`Plane`** class, a **`propeller_diameter`** attribute to the **`Propeller`** class and an **`engine_thrust`** attribute to the Jet class. Ensure that you can create objects of each class and that the attributes are set correctly.

3. Multiple Inheritance

So far, we have built some simple hierarchies. However, in the real world, hierarchies are often more complex. Imagine the following example:

Here we have an instance of something that does not fully match our car or our plane hierarchy. It is a car, but it can also fly. So how do we model this? This is where multiple inheritance comes in. Multiple inheritance allows us to inherit from multiple classes, essentially combining the functionality.

To demonstrate this, let us create a **`FlyingCar`** class that inherits from both the **`Car`** and the **`Plane`** class.

```
class FlyingCar(Car, Plane):
    def __init__(self, colour, weight, max_speed, form_factor, wingspan, **kwargs):
        # we need to add the wingspan to the keyword arguments so that following the MRO, the Plane class gets all
the keyword arguments it needs
        super().__init__(colour, weight, max_speed, form_factor=form_factor, wingspan=wingspan, **kwargs)

    def move(self, speed):
        print(f"The flying car is driving or flying at {speed} km/h")
```

You will see that this time, we have followed the class name with two parent classes, separated by a comma. This enables multiple inheritance. Next to that, we have also added the arguments that are required by both the **Car** and the **Plane** class (**form_factor** for the **Car** and **wingspan** for the **Plane**). We have also added the ****kwargs** parameter to allow for any additional keyword arguments to be passed to the **__init__** method (such as **seats** or **max_range**).

We have also overridden the move method to print a different message.

Let's test this by creating an object:

```
generic_flying_car = FlyingCar("red", 1000, 200, "SUV", 30, seats=5)
generic_flying_car.move(100)
print(generic_flying_car.seats, generic_flying_car.wingspan,
generic_flying_car.form_factor)
```

You will see that the move method prints a different message and that we can access the attributes of both the **Car** and the **Plane** class. Although, the creation of an object can get quite convoluted.
Python is quite good in that it allows us to pass keyword arguments for any arguments required by the parent classes.

To make this a bit clearer, let us create a generic flying car (that is the same but with some clearer code):

```
generic_flying_car_2 = FlyingCar(colour="red", weight=1000, max_speed=200,
form_factor="SUV", wingspan=30, seats=5)
generic_flying_car_2.move(100)
```

You see when one has a lot of arguments, it may be worth just turning them all into keyword arguments to make your code more readable.

In the lecture, we also talked about **mixins**. They are a way to add functionality to a class without having to inherit from it. Whilst it is not likely that you will use mixins in your code, it is important to understand the concept. So, whilst we are at it, here is a link to Stackoverflow that explains mixins quite well: https://stackoverflow.com/questions/533631/what-is-a-mixin-and-why-are-they-useful

You may ask, why are we linking to StackOverflow rather than having the explanation here? Well, across your programming journey, you will most likely encounter a lot of problems that you can't wrap your head around. StackOverflow is a great resource to find answers to your questions (and many professional developers use it). However, it is important to understand that you should not just copy and paste code from StackOverflow. Rather, you should try to understand the code and then adapt it to your own needs!

4. Polymorphism

Polymorphism, as covered in the lecture, allows us to use the same method name in different classes. In more OOP specific terms, we call the method that all the various classes have in common an interface.

Revisiting our example from above, we can see that all vehicles have a move method. However, the **move** method is implemented differently in each class. For example, a car drives, a plane flies and a flying car drives and flies. The important term to remember in this regard is called method overriding: we override the method in the parent class with a method in the child class.

Note: Other languages implement a concept called method overloading. This means that you can have multiple methods with the same name in the same class but with different parameters. An example of "pseudo-code" would be:

```
function add_numbers(number1, number2):
    return number1 + number2

function add_numbers(number1, number2, number3):
    return number1 + number2 + number3
```

This means it would not matter whether you pass two or three numbers to the **add_numbers** function. However, Python does not support method overloading. If you try to do this, you will get an error message.

Polymorphism does not only extend to classes that have a common parent class. For example, we could also have a class **Animal** that has a move method.

```python
class Animal:
    # we omit the __init__ method for brevity

    def move(self, speed):
        print(f"The animal is moving at a speed of {speed}")

generic_animal = Animal()
generic_animal.move(20)
```

Let's pretend we have some kind of functionality in our program that "moves" things.
It would not matter if we had any of our cars/planes or animals. We can call the same interface (**move**) and the program will do the right thing. This is the power of polymorphism.

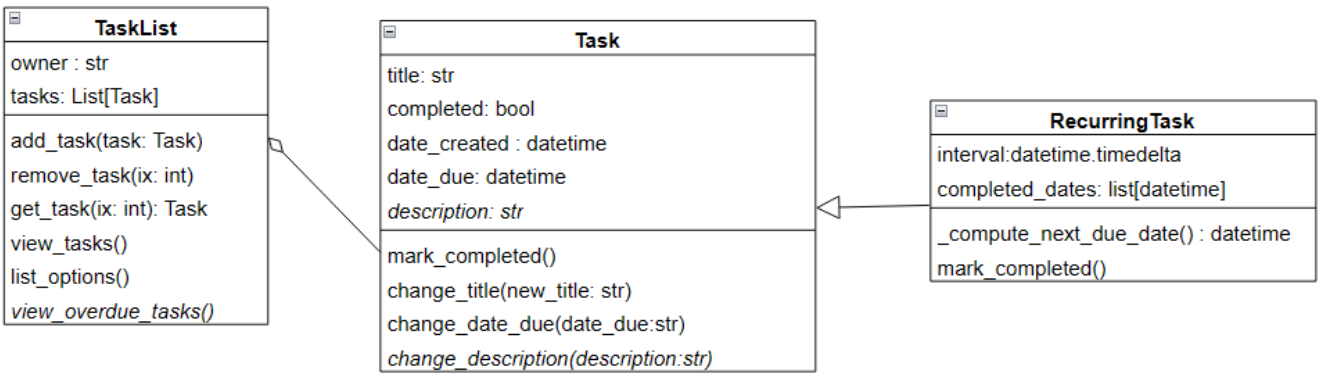
```python
for movable_object in [generic_vehicle, generic_electric_car, generic_flying_car,
generic_animal]:
    movable_object.move(20)
```

6. ToDO:

From here on, we are finished working in the **lab_week5.py** file. Save this file (but keep it handy to refer to). As mentioned at the start of this lab, you should have added type hints and docstrings to all your code. The examples from here will now include this again.

Let us apply what we have learned to our ToDoApp. Tasks are often not once-and-done. They are often recurring. For example, you may want to do your laundry every week or you may want to clean your room every two weeks. Whilst we could manually add these tasks to our task list, it would be much easier if we could just add a task once and then have it automatically added to our task list every week or every two weeks.

For this, we want to create a new type of task called **`RecurringTask`**. This task should have all the attributes and methods of the **`Task`** class but should also have an additional attribute called **`interval`**. This attribute should be a string and should indicate how often the task should be repeated.



Add this to the tasks module.

```python
class RecurringTask(Task):
    """Represents a recurring task in a to-do list.

    Args:
        Task (Task): The task to be repeated.
    """

    def __init__(self, title: str, date_due: datetime.datetime, interval: datetime.timedelta):
        """Creates a new recurring task.

        Args:
            title (str): Title of the task.
            date_due (datetime.datetime): Due date of the task.
            interval (datetime.timedelta): Interval between each repetition.
        """
        super().__init__(title, date_due)
        self.interval = interval
        self.completed_dates : list[datetime.datetime] = [] # list of datetime.datetime objects

    def _compute_next_due_date(self) -> datetime.datetime:
        """Computes the next due date of the task.

        Returns:
            datetime.datetime: The next due date of the task.
        """
        return self.date_due + self.interval

    def __str__(self) -> str:
```

```
        return f"{self.title} - Recurring (created: {self.date_created}, due: {self.date_due},
completed: {self.completed_dates}, interval: {self.interval})"
```

As you can see, we have added the **interval** attribute to the **__init__** method. This is a **timedelta** object that indicates how often the task should be repeated. We have also added a new attribute called **completed_dates**. This is a list that will store all the dates on which the task was completed (so that we have some form of history). We have also added a new method called **_compute_next_due_date**. This method computes the next due date of the task based on the interval. Additionally, (as some form of polymorphism) we have overridden the **__str__** method to include the interval and the completed dates. This is because otherwise, when listing the tasks, we would not know which tasks are recurring and which are not.

We can now modify the main module to allow the user to add recurring tasks.
Start by importing the **RecurringTask** class at the top of the file that contains the **main()** method.

```
from tasks import Task, RecurringTask
```

Task:
Modify choice "1" in the **main** function to allow the user to add a recurring task. For this, you should:
- ask the user whether they want to add a recurring task or a normal task
- if they want to add a recurring task, ask them to input a **timedelta** object in days. You can do this by asking them to input a number and use this: **interval = datetime.timedelta(days=int(interval))** to convert the number to a **timedelta** object.
- then create a **RecurringTask** object and add it to the task list
- change the conditional that if it is simply a normal task, create a **Task** object and add it to the task list

Then, to test whether everything works as expected, we are going to modify the "test" code that we added to the main module last week in the "**propagate_task_list**" function.

Add the following before the return statement of this function:

```
    # sample recurring task
    r_task = RecurringTask("Go to the gym", datetime.datetime.now(),
datetime.timedelta(days=7))
    # propagate the recurring task with some completed dates
    r_task.completed_dates.append(datetime.datetime.now() - datetime.timedelta(days=7))
    r_task.completed_dates.append(datetime.datetime.now() - datetime.timedelta(days=14))
    r_task.completed_dates.append(datetime.datetime.now() - datetime.timedelta(days=22))
    r_task.date_created = datetime.datetime.now() - datetime.timedelta(days=28)

    task_list.add_task(r_task)
```

You will see, this simply creates a recurring task and adds some completed dates to it. Then, it adds the recurring task to the task list.

Now, simply run the **main.py** file and see whether everything works as expected. You should be able to add a recurring task and see it in the list of tasks. One thing that may not work fully as expected is to change the due date of a recurring task. And the new functionality that we have added does not "recur" the task. This is something that we will implement next.

In our **`choice == 5`** we mark a task as completed. however, a recurring task will only be marked as completed and not get a new due date. Using polymorphism, we can override the **`mark_completed`** method in the **`RecurringTask`** class to also update the due date.

Task: Override the **`mark_completed`** method in the **`RecurringTask`** class to also update the due date. For this, you should:
- add the current date to the **`completed_dates`** list (that is created when one calls the **`mark_completed`** method of this recurring task)
- update the **`date_due`** attribute of the recurring task by calling **`the _compute_next_due_date`** method

Then, test whether this works as expected by running the **`main.py`** file and marking a recurring task as completed. You should see that the due date of the recurring task is updated.

If everything works as expected, you have now successfully implemented inheritance and polymorphism in your ToDoApp. Congratulations!

We are now going to do some cleanup and refactoring before we move on to the portfolio task.

Exercise 4 - Encapsulation

In OOP we want to ensure that we use interfaces to perform actions. If you look at choice 4/5 in the **`main`** function, you can see what we access properties of the **`TaskList`** using **`task_list.tasks[ix].`** This is not ideal. We are essentially bypassing the **`TaskList`** class and accessing the tasks directly. It would be much better if our **`TaskList`** (that should handle all the tasks) had a method that allows us to access the tasks in a more controlled way.

Task: Add a method to the **`TaskList`** class that allows us to access the tasks in a more controlled way. For this, you should:
- Modify the **`TaskList`** class to include a method called **`get_task`** that takes an index as a parameter and returns the task at that index.
- Modify the **`main`** function to use this method instead of accessing the tasks directly.

Then, test whether everything still works as expected by running the **`main.py`** file. You should see that everything still works as before.

Whilst this is not directly related to polymorphism or inheritance it covers an important concept in OOP: ***encapsulation***. Encapsulation means that we want to hide the inner workings of our classes from the user of our classes. We want to ensure that the user of our classes can only access the functionality that we want them to access. In this case, we want to ensure that the user of the **`TaskList`** class can only access the tasks via the **`get_tasks`** method. Other parts in our app where we alread had used encapsulation are the **`add_task`** and **`remove_task`** methods. We could have simply allowed the user of the **`TaskList`** class to access the tasks directly and add/remove tasks from the list.
You may ask, why is this important in our ToDo case? Think of it this, way: if we want to change the way we store the tasks (for example, we may want to store them in a database rather than a list), we only need to change the **`get_tasks`** method. We do not need to change the **`main`** function. This is because

the main function does not know how the tasks are stored, it only knows that it can access them via the **`get_tasks`** method.

7. Portfolio Exercises

In this portfolio task, we want to expand the part of the ToDoApp that handles owners.
We want to add functionality that not only stores a string of the owner's name but also has user and owner classes.

Portfolio Exercise 3

Modify the UML diagram to include the new functionality. You can use draw.io to do this (or any other tool such as StarUML). You can find the draw.io UML diagram starting point on Aula (**`ToDo_UML.drawio`**). You can import this into the web app at https://app.diagrams.net/ and modify it. You can then export it as a PNG file and add it to your portfolio. (Use this guide as guidance on how to modify class diagrams: https://www.drawio.com/blog/uml-class-diagrams)

At the minimum, you should add the following:
- A new class called **`User`**. This class should have the following attributes: **`name`**, **`email`**.
- A new class called **`Owner`**. This class should inherit from the **`User`** class.
- Modifications to the **`TaskList`** class to include an owner attribute. This attribute should be of type **`Owner`** and you should select an appropriate UML relationship between the **`TaskList`** and **`Owner`** classes.

Portfolio Exercise 4

Add the functionality to the Python code. For this, you should:
- Create a new module called **`users`**. This module should contain the **`User`** and **`Owner`** classes both with an **`__str__`** method that represents their attributes and whether they are **`Owner`** or **`User`**.
- Modify the **`TaskList`** class to include an **`owner`** attribute. This attribute should be of type **`Owner`** and should be set when constructing a new **`TaskList`** object within the **`__init__`** method.
- modify the **`main`** function to store and create the default task list with an **`owner`** instance.