

COMP11124 Object Oriented Programming

Debugging, Properties and Persistence

Learning Outcomes

In this lab, you will learn about how to debug your Python code, find out the usage of Python properties and a way to implement persistence: topics are important for your programming. After you have completed this lab, you should be able to:

- Understand and apply debugging techniques to your code.
- Design and implement persistence, following OOP principles.
- Demonstrate the use of the Python *property* to handle managed attributes.

Topics Covered

Python: Debugging, *property* in Python and persistence using the DAO pattern with CSV and Pickle

Getting Started Task: Download the file called `lab_week_6_debugging.py` from Aula and open it in VSCode.

1. Debugging

The *debugging* process is an important part of programming. It allows you to find and fix errors in your code. A debugger is a tool that allows you to step through your code and see what is happening at each step. This saves you from writing lots and lots of print statements to see what is happening in your code.

Read the first part of this link to understand some more of the key terminology:
<https://aws.amazon.com/what-is/debugging/>

In this exercise, we are going to use the debugger in VSCode to debug a simple Python program. This **lab_week_6_debugging.py** file contains a simple Python program that has a bug (error) in it. This error is not a syntax error, but a logical error. The program runs, but it does not produce the correct output. Those errors are the most difficult to find and fix and a debugger can help you with this.¹

¹The example has been adapted from <https://www.jetbrains.com/help/pycharm/creating-and-running-your-first-python-project.html> and <https://www.jetbrains.com/help/pycharm/debugging-your-first-python-application.html> to fit the context of this lab (e.g. using VSCode instead of PyCharm).

Exercise 1: Finding the Problem

First, read the code within the file and try to understand it. Then, run the script, accelerate the car once, and then brake it twice by typing the corresponding console window.

```
ct Oriented Programming/Practicals/lab_week_6_debugging.py"
I'm a car!
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?A
Accelerating...
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?B
Braking...
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?B
Braking...
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?
```

Now press **o** followed by **Enter** to show the car's odometer:

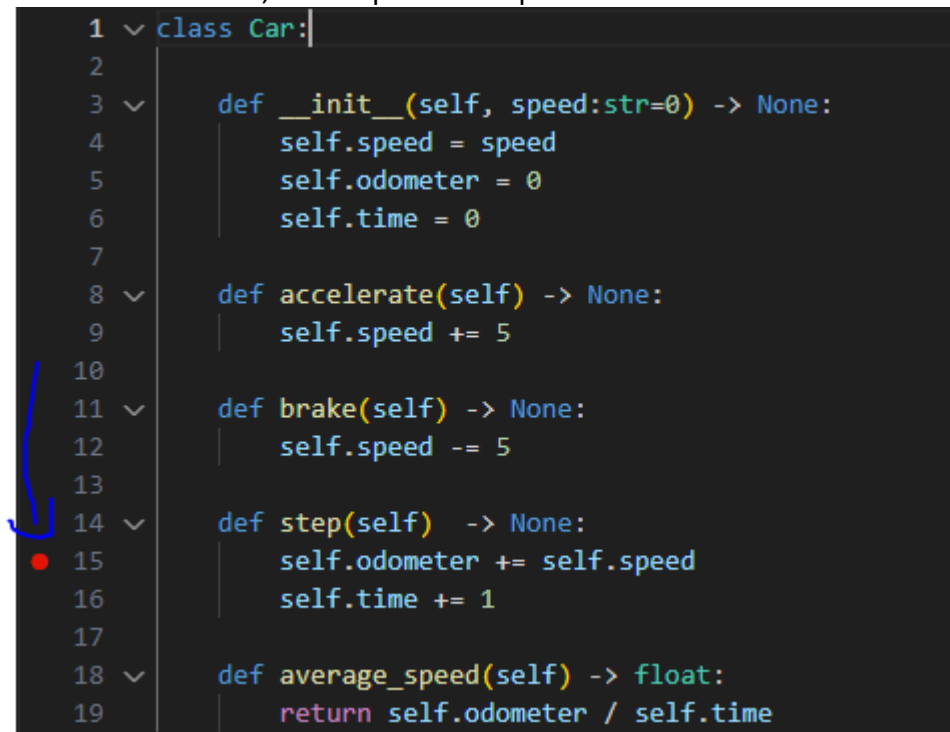
```
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?O
The car has driven 0 kilometers
```

The script tells us that the car has travelled 0 kilometres! It's an unexpected result because we've pushed the accelerator once, so the car should have covered some distance. Let's debug the code to find out the reason for that.

To start debugging, you have to set a breakpoint first. The debugger will stop just before executing the line with the breakpoint, and you will be able to examine the current state of the program. The state will include the values of all variables, the call stack and more.

To set a breakpoint, click on the part to the left of the line number where you want to set the breakpoint. A red dot will appear before the line number, indicating that a breakpoint is set.

The car's odometer is set on line 15, so let's put a breakpoint there. Click the area before the line number:

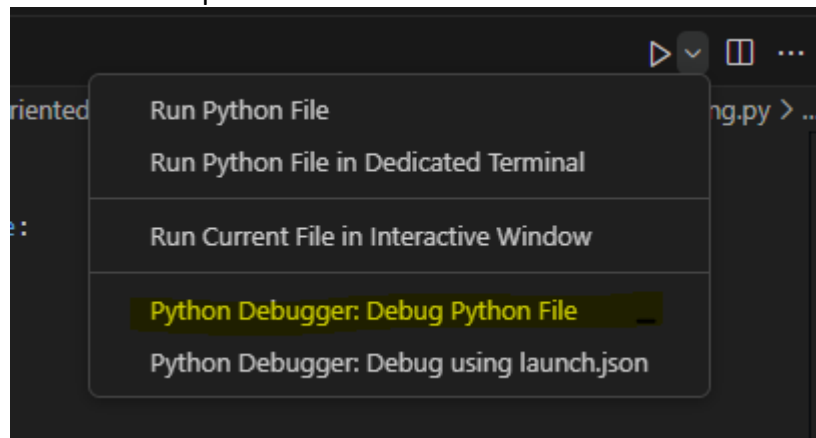


```

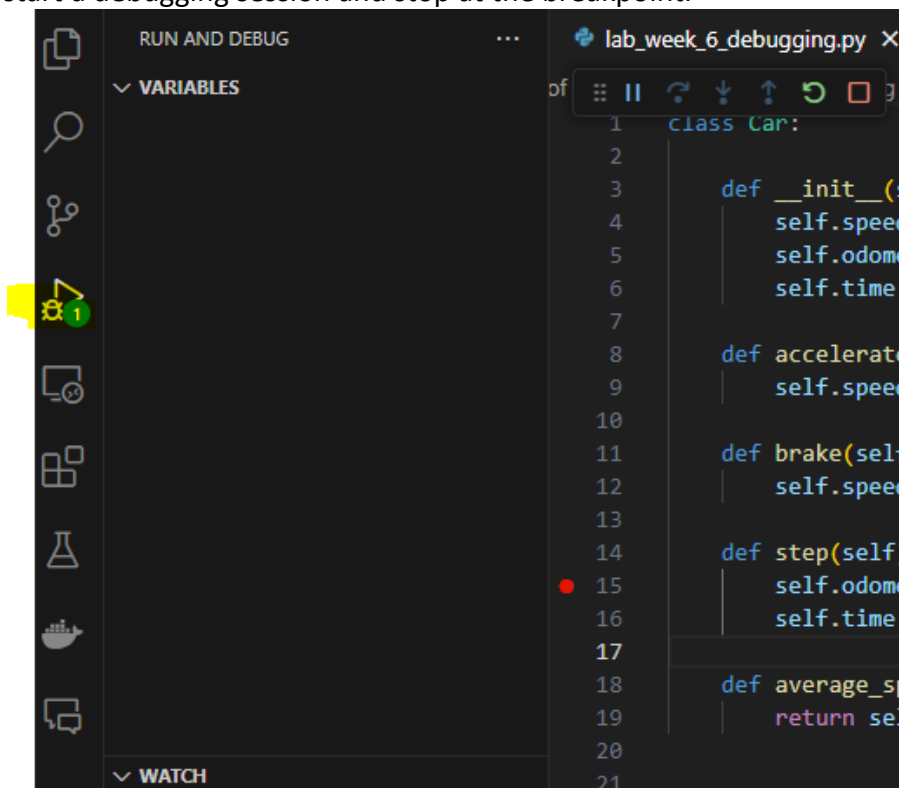
1  class Car:
2
3  def __init__(self, speed:str=0) -> None:
4      self.speed = speed
5      self.odometer = 0
6      self.time = 0
7
8  def accelerate(self) -> None:
9      self.speed += 5
10
11 def brake(self) -> None:
12     self.speed -= 5
13
14 def step(self) -> None:
15     self.odometer += self.speed
16     self.time += 1
17
18 def average_speed(self) -> float:
19     return self.odometer / self.time

```

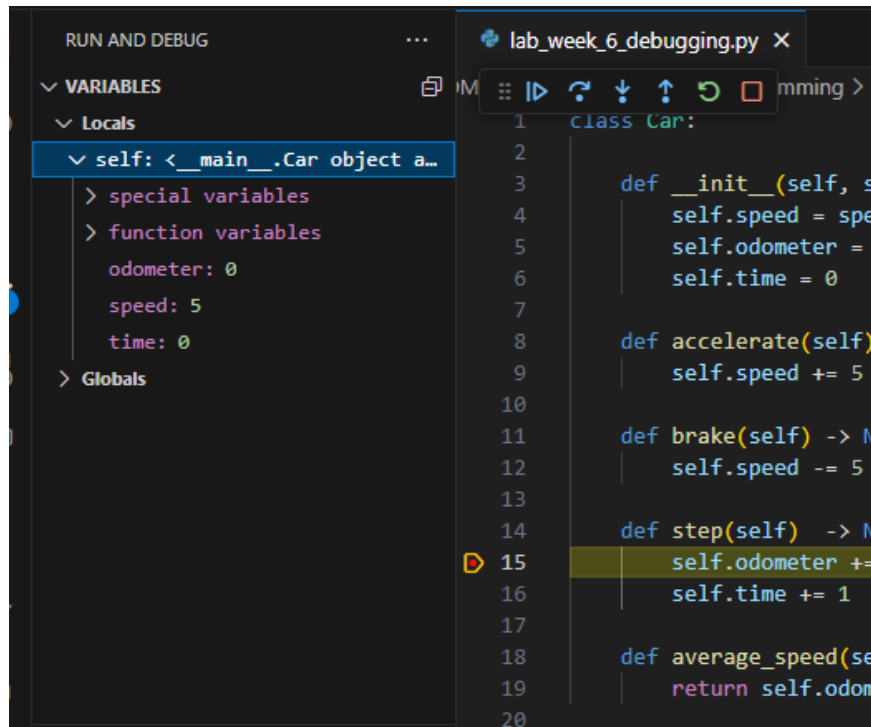
Next, click the small down arrow next to the green triangle in the top right corner of the window and select the "Python: Debug File" option. This will start the debugger and run the script in debug mode. The debugger will stop at the first breakpoint it encounters.



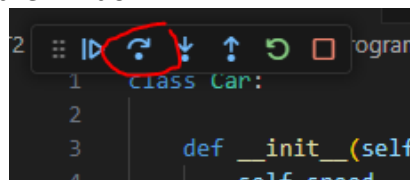
VSCode will then start a debugging session and stop at the breakpoint.



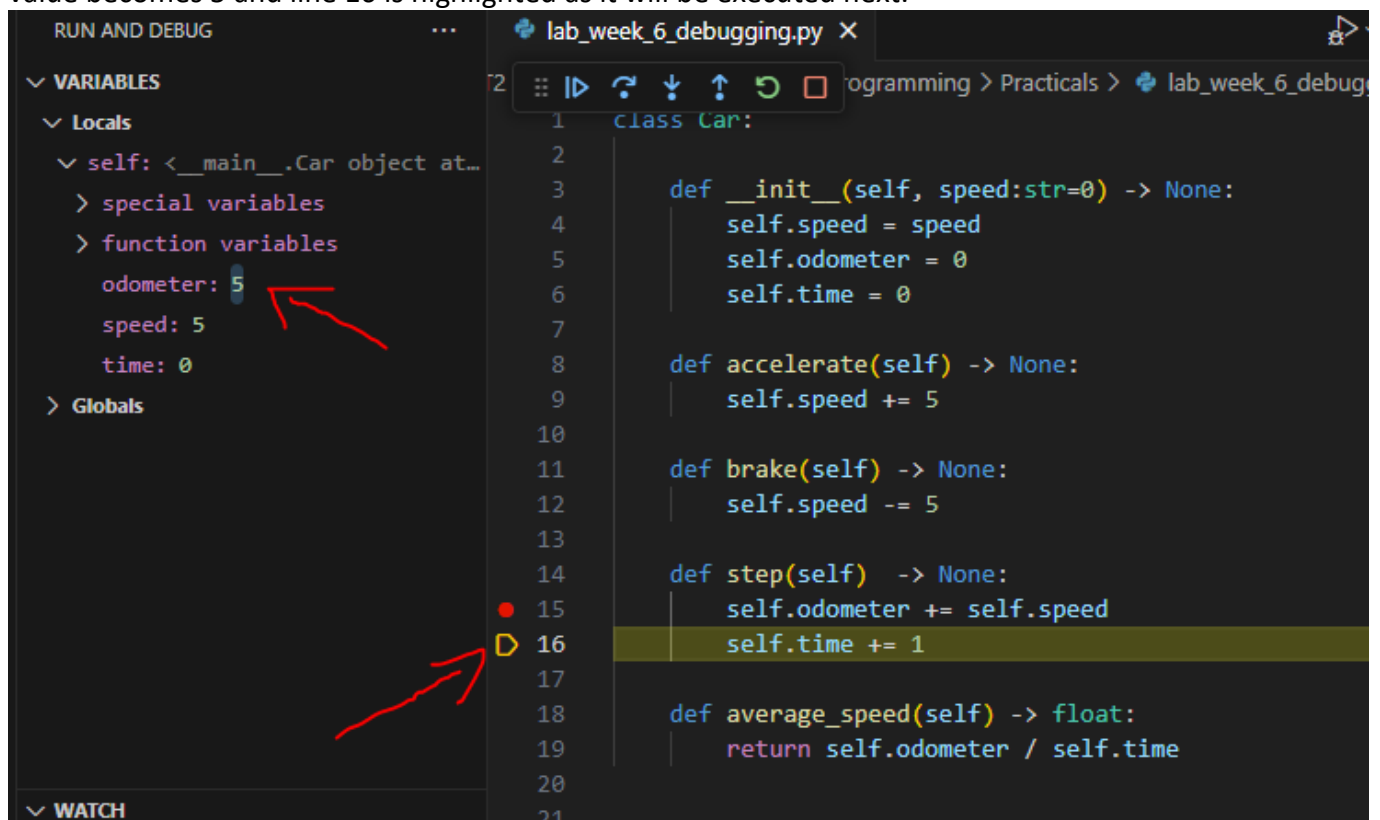
Press **a** followed by **Enter** to accelerate the car. This will execute the script and stop at the line you have set the breakpoint. On the left-hand side (unless you have moved it) you will see the *Variables* that are currently in scope. You will see local and global ones. If you expand the **self** variable, you will see the attributes of the car object. You will see that the **odometer** attribute is set to 0, the speed to 5 and the time to 0.



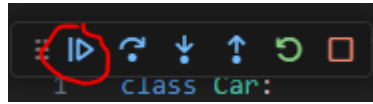
Now, press the *Step Over* button in the window:



This will execute the line with the breakpoint and stop at the next line. You will see that the odometer value becomes 5 and line 16 is highlighted as it will be executed next.

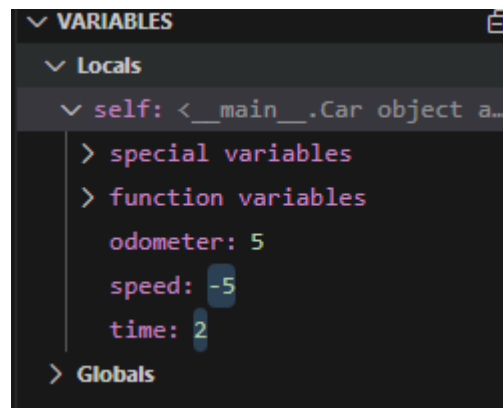


Now click Resume Program (blue play button) to continue the execution of the program.



Instruct the car to brake by pressing **b** followed by **Enter**. Look back at the *Variables* section and see how the speed attribute changes. (The speed will be 0)

Click *resume program* and brake again (**b + enter**). Now let us look at the variables and analyse what will happen next.



You can see, that the value of **odometer** is 5, and the value of **speed** is -5. That's why when we resume the execution, **odometer** will become 0. You can click the *Step Over* button to skip the next line and see that this happens.

```
What should I do? [A]ccelerate, [B]rake, show [0]dometer, or show average [5]peed?0
The car has driven 0 kilometers
```

So, the reason for an unexpected result is that the car's speed becomes negative when we brake. This is not a bug in the code, but a logical error.

Exercise 2: Fixing the Problem

We need to add a check to the brake method to ensure that the speed does not become negative.

Select the statement **speed -= 5** and modify it so that it looks like this:

```
def brake(self):
    if self.speed >= 5:
        self.speed -= 5
    else:
        self.speed = 0
```

Let's run the script again and see if the problem is fixed. Accelerate the car once, and then brake it twice. Press **o** followed by **Enter** to show the car's odometer. You will see that the car has travelled 5 kilometers, which is the expected result.

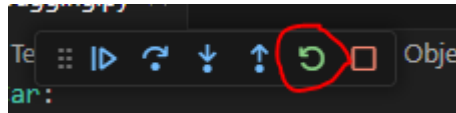
```

ing/Practicals/lab_week_6_debugging.py"
I'm a car!
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?A
Accelerating...
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?B
Braking...
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?B
Braking...
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?O
The car has driven 5 kilometers
What should I do? [A]ccelerate, [B]rake, show [O]dometer, or show average [S]peed?

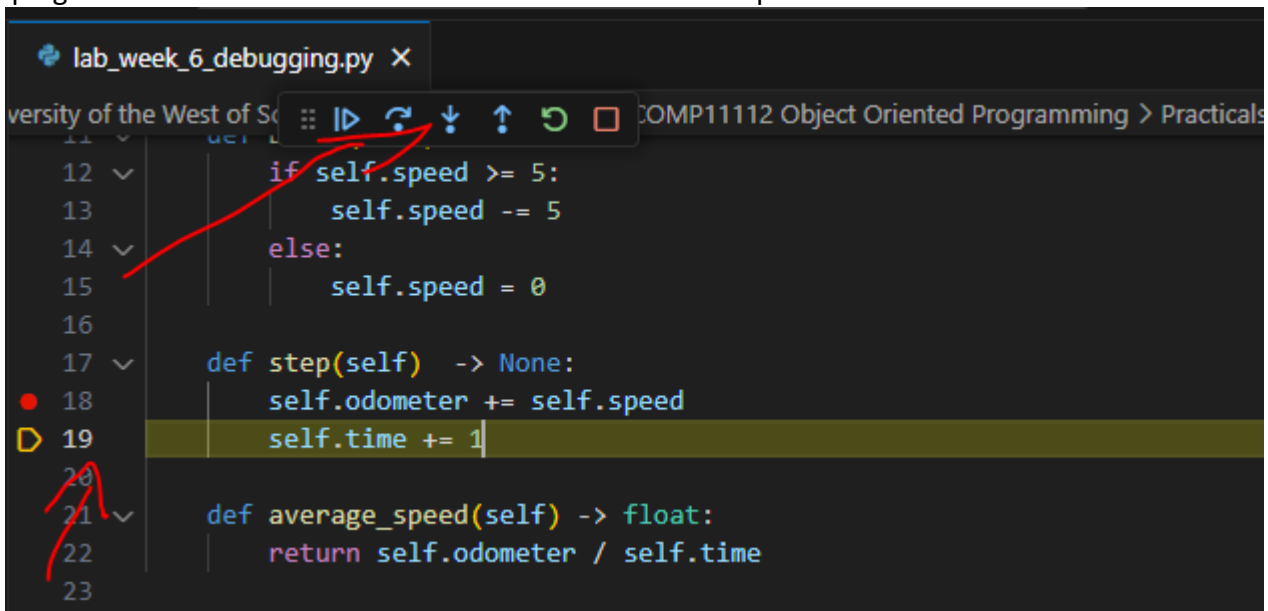
```

Exercise 3: Stepping through the Code

If you want to see what your code does line by line, there's no need to put a breakpoint on every line, you can step through your code. Let's see what it looks like to step through our example program. Start or restart the debugger by using the *restart* button:



In the console, press **a** to accelerate the car. The debugger will stop at the breakpoint. We can use the stepping toolbar buttons to choose which line we'd like to stop on next.



For example, click the *Step Over* button to see the marker moving to the next line of code.

Enter the action **a** and then keep clicking the *Step Over* button until you are on the

```
my_car.accelerate()
```

line. Now, if you click the *Step Into* button, you will see that the debugger goes into the method that accelerates the car.

However, if you continue using *Step Over*, you'll see that your application just continues to the next loop

```

14         continue
15     if action == 'A':
16         my_car.accelerate()
17         print("Accelerating...")
18     elif action == 'B':

```

```

8      def accelerate(self) -> None:
9          self.speed += 5

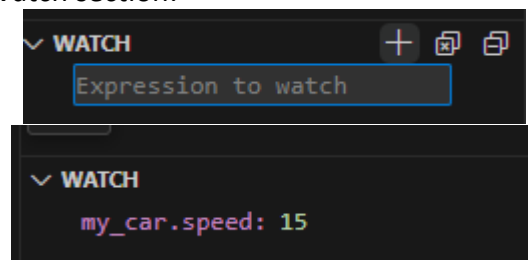
```

The *Step Into* and *Step Out* functionality is useful when you want to see what a function does. If you simply skip through the code line by line, the debugger will not jump into the function automatically. You will need to do that manually.

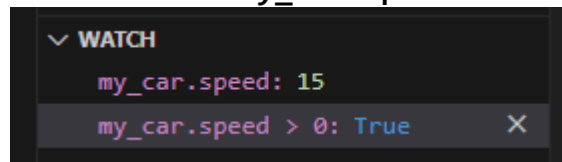
Exercise 4: Watching Variables or Expressions

VSCode allows you to watch any variable. Just type the name of the variable you want to watch in the "Watch" view.

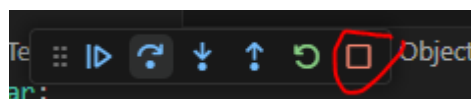
Add `my_car.speed` into the Watch section:



This allows you to see values of specific variables without needing to expand the section with "Variables". It also allows you to *watch* expressions such as `"my_car.speed > 0"`.



To finish the debugging session, click the red square in the top right corner of the window. This will stop the debugger and return you to the normal editor view.



Hopefully, from now on, you will not need to be using lots of print statements to debug your code. You can simply use the debugger to step through your code and see what is happening at each step. This will save you a lot of time and effort in the long run.

Although this was only a quick guide, it is recommended that you also have a look at: <https://code.visualstudio.com/docs/editor/debugging> as this contains much more detail.

2. Properties using the @property decorator

For this exercise, we will use your code for the `ToDoApp`. If you have not completed the `ToDoApp`, you can use the code available on Aula as a starting point (but it is highly recommended that you complete the `ToDoApp` first as this feeds directly into the portfolio tasks you are completing in the course of this module). As for each lab, it would be best to create a new copy of your code and work on that.

Note: The code shown in this lab is mainly geared towards the *basic* ToDo app, without any portfolio tasks. If you have completed them and created attributes such as a “task description” additional changes may be needed.

Recall from the lecture that properties in Python allow you to define a method that can be accessed like an attribute. This is very useful when you have the data available but need to perform some kind of computation before returning the data. Rather than using a method to do this, you can use a property.

In our ToDo example, when we start using it over time, we will probably amass quite a lot of tasks that we have completed. However, no matter whether completed or not, we store them in the **tasks** attribute of the **TaskList** class.

If you would want to filter the tasks that are completed and only shows ones that are yet to be done, you would need to create a method to do that.

However, it would be much easier to have a property so that we can just access **task_list.uncompleted_tasks** and get a list of all the tasks that we have yet to do.

To enable that, let us modify the **TaskList** class. First, we start by creating a new method:

```
class TaskList:
    # ....
    def uncompleted_tasks(self) -> list[Task]:
        return [task for task in self.tasks if not task.completed]
```

This method returns a list of all the tasks that are not completed using list comprehension.

List comprehension is a simple way of looping through a list and performing some kind of computation on each element.

Although we could have also used code like this (do not copy):

```
def completed_tasks(self) -> list[Task]:
    completed_tasks = []
    for task in self.tasks:
        if not task.completed:
            completed_tasks.append(task)
    return completed_tasks
```

list comprehension is a more elegant way of doing it. (Read more on this here: https://www.w3schools.com/python/python_lists_comprehension.asp)

Now, every time we want to get the tasks we have still to do, we need to call the method. However, we can use the **@property** decorator to make this method accessible like an attribute.

This decorator goes into the line above the method definition like this:

```
class TaskList:
    # ....
    @property
    def uncompleted_tasks(self):
```



```
return [task for task in self.tasks if not task.completed]
```

Task: Modify the **view_tasks** method of the **TaskList** class to only show the uncompleted tasks using the new property. Change the text that is printed to something along the lines of "The following tasks are still to be done:" so that users know that they are looking at the uncompleted tasks. Ensure that you show the right indices by removing the **"enumerate"** statement and using the **index()** method to get the index for printing out the task info in the format: **print(f"{ix}: {task}")**. This is important as you are now using the **uncompleted_tasks** property and are *hiding* the completed tasks.

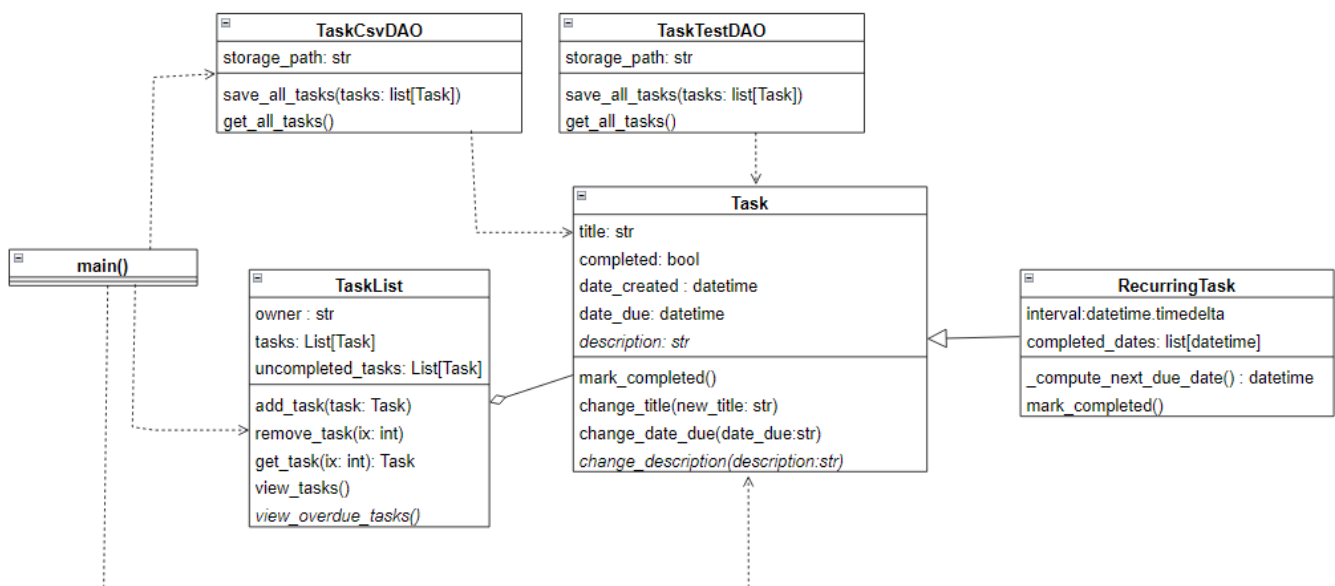
Then, test whether everything still works as expected by running the main.py file. You should see that only the uncompleted tasks are shown when you view the tasks.

3. Implementing Persistence

One important feature that our ToDo app is still missing is the ability to save the state of the program. Whether this is to a file or a database, we want to be able to save the tasks that we have created (and completed) so that we can access them later. This is called *persistence* and using OOP principles, we can implement this in a few steps.

First, we need to decide on a format in which we want to save the tasks. For this exercise, we are going to use a CSV file. CSV stands for Comma Separated Values. It is a simple file format that is used to store tabular data, such as a spreadsheet: each line in a CSV file represents a row of data and each value in a line is separated by a comma.

After completing this part, the UML class diagram representation of your ToDoApp (barring the *user* portfolio task) should look similar to this:



Note that rather than having the **uncompleted_tasks** method within the method section of **TaskList** we are treating this as an attribute (due to being a Python property).

You may also notice that we have added the **main()** method that orchestrates handling the Task/TaskLists.

Exercise 1: DAO

We can use a design pattern called Data Access Object (DAO) to implement the persistence. Design patterns are a way of solving common problems in software development by using a standardised approach. Many books have been written about design patterns and they are an important part of software engineering: Often you will find that a problem you are trying to solve has already been solved by someone else and you can use their solution (albeit with some modifications).

The DAO pattern allows us to separate the data access logic from the business logic. In our case, we can create a **TaskDAO** class that will handle the reading and writing of tasks to a CSV file.

You can find some more information on the DAO pattern here <https://www.digitalocean.com/community/tutorials/dao-design-pattern> . The code example is in Java, but the concept is the same (and it is good to get exposure to other programming languages).

In your **main** module, you will find the code that we use to propagate the task list. We have used this code to test whether the task list works as expected and to add some sample tasks to the task list. However, one thing that we can do to move this logic away from the main module, is to create a DAO that handles the creation of this task list (pretending that some persistence has been implemented in a database or CSV file).

Let us create a new module called **dao**. Within this module, we create a **TaskTestDAO** class.

```
class TaskTestDAO:
    def __init__(self, storage_path: str) -> None:
        self.storage_path = storage_path

    def get_all_tasks(self) -> list[Task]:
        task_list = [
            Task("Buy groceries", datetime.datetime.now() - datetime.timedelta(days=4)),
            Task("Do laundry", datetime.datetime.now() - datetime.timedelta(days=-2)),
            Task("Clean room", datetime.datetime.now() + datetime.timedelta(days=-1)),
            Task("Do homework", datetime.datetime.now() + datetime.timedelta(days=3)),
            Task("Walk dog", datetime.datetime.now() + datetime.timedelta(days=5)),
            Task("Do dishes", datetime.datetime.now() + datetime.timedelta(days=6))
        ]

        # sample recurring task
        r_task = RecurringTask("Go to the gym", datetime.datetime.now(),
datetime.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.append(r_task)

        return task_list
```

```
def save_all_tasks(self, tasks: list[Task]) -> None:
    pass
```

Note that this is a **test** DAO. We are not actually going to save the tasks to a file. For now, we are simply pretending that we are loading and saving the tasks to a file and moving the logic that creates our sample tasks to this class. The `__init__` method takes a **storage_path** parameter. This could be a file path, but you could also rewrite this class to use a database. Another thing that we do not do within this DAO is to handle the **date_created** for each task, but we will look at this later during this lab.

Task: Modify the **ToDo** app to use the **TaskTestDAO** class to create the task list rather than doing it within the **propagate_task_list** method of the main module.

To achieve this:

- remove the **propagate_task_list** function from the main module
 - change the **main** function to remove the call to the **propagate_task_list** function
 - add two new choices to the main function: one to load the tasks from the DAO and one to save the tasks to the DAO.
 - when the user uses one of these choices, ask them for a file path and then create an instance of the **TaskTestDAO** class using this file path. Then, use the appropriate method to load/save the tasks. (Note: when loading the tasks, you may need to use a loop to add the tasks to the task list one by one.)
- Then, run the app and view all tasks. If you have done it correctly, you should see (before loading the tasks) that the task list is empty. Then, when you load the tasks, you should see the sample tasks that we have created in the **TaskTestDAO** class.

Note: It is important to understand that whilst we have only implemented methods that save/get all tasks, often in such DAO you also have methods that read/update/delete single records. This is necessary because if you have a lot of data and multiple people using the same **ToDo** lists concurrently, you do not want to read and write the whole file every time you want to change something. However, for this exercise, we are keeping it simple and simply using the DAO to serve as an export/import mechanism for the task list.

Exercise 2: CSV Persistence

Now that we have implemented the DAO, we can implement the persistence using a CSV file. You will find a file called **tasks.csv** on Aula which contains the same tasks that we have hard-coded into the **TaskTestDAO** class.

Open the file (in Excel or even in VSCode) and see how the data is stored. You will see that each line represents a task and that the values are separated by a comma.

We are now going to add functionality that allows us to read the tasks from this file, create the respective **Task** objects and add them to the **TaskList**. Additionally, we are going to add functionality that allows us to save the tasks to this file as well.

This aligns with the concept of *Serialization*. Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. The main purpose of serializing an object is to be able to recreate it when needed. In our case, we are going to serialize the tasks to a CSV file. There may be better formats out there (see the extra, optional task at the end of this lab) but the CSV format is simple and easy to understand and serves well for the purpose of this exercise.

Copy/paste the following code into your DAO module:

```

import csv
class TaskCsvDAO:
    def __init__(self, storage_path: str) -> None:
        self.storage_path = storage_path

        self.fieldnames = ["title", "type", "date_due", "completed", "interval", "completed_dates", "date_created"]

    def get_all_tasks(self) -> list[Task]:
        task_list = []
        with open(self.storage_path, "r") as file:
            reader = csv.DictReader(file)

            for row in reader:
                task_type = row["type"]
                task_title = row["title"]
                task_date_due = row["date_due"]
                task_completed = row["completed"]
                task_interval = row["interval"]
                task_date_created = row["date_created"]
                task_completed_dates = row["completed_dates"]

                # YOUR CODE TASK A
                task_list.append(task)
        return task_list

    def save_all_tasks(self, tasks: list[Task]) -> None:

        with open(self.storage_path, "w", newline='') as file:
            writer = csv.DictWriter(file, fieldnames=self.fieldnames)
            writer.writeheader()
            for task in tasks:
                row = {}

                # YOUR CODE FOR TASK B BELOW

                row["title"] = None
                row["type"] = None
                row["date_due"] = None
                row["completed"] = None
                row["interval"] = None
                row["completed_dates"] = None
                row["date_created"] = None
                writer.writerow(row)

```

Then, move the **import csv** statement to the top of the file (since imports should always be located there).

The code will not run and show some squiggly lines, which we will fix in a moment. There are several new additions to this code. First, a new attribute called **fieldnames**. This reflects the list of column names that we have used in the CSV file (and overlaps with all the attributes of the **Task** class and its **RecurringTask** child class).

We then have the same two methods that we have used in the **TaskTestDAO** class. The **get_all_tasks** method reads the tasks from the CSV file and creates the respective task objects. The **save_all_tasks** method writes the tasks to the CSV file.

Both of those methods use the **open()** function to open the file. The **open()** function takes two parameters: the file path and the mode. The mode "r" is used to open the file for reading and "w" is used to open the file for writing. The **newline** parameter is used to ensure that the file is written in a way that operating systems can understand. Have a look at this quick tutorial to understand more about the **open()** function: https://www.w3schools.com/python/python_file_open.asp

We then call the imported CSV module with the **csv.DictReader** and **csv.DictWriter** classes. These classes allow us to read and write CSV files using dictionaries; A data structure which we will explore in more detail in one of the next lectures. For now, use the scaffolding code provided to complete the two methods. The **get_all_tasks** method simply loops through the file line-by-line and stores the relevant data from the row dictionary in the variables.

Task A: Complete the **get_all_tasks** method to read the tasks from the CSV file and create the respective task objects. For this, you should:

- Change the main module to use the **TaskCsvDAO** class (rather than the **TaskTestDAO** class) to create the task list when importing the tasks
- Use the debugger to step through the code and look at the variable values and how the data is read from the file
- Create a new **Task** or **RecurringTask** object for each row in the file and add it to the **task_list**. Here you should build a conditional that checks the type and acts accordingly.
- Ensure you handle the variable types correctly. You can parse the dates using the **datetime.datetime.strptime(DATE_VAR, "%Y-%m-%d")** method, which returns a **datetime** object (similar to what we have done before). Do not forget that the **completed_dates** attribute is a list of dates, which should be parsed as well (Hint: you can use the **str.split** method to split a string into a list of strings and then use a loop to parse each string into a **datetime** object). Additionally, ensure that when parsing the **interval** you only select the first number in the stored string (Hint: use the **.split()** method).

If you have completed this task correctly, you should see that the tasks are loaded from the CSV file and added to the task list when you run the main.py file.

Task B: Complete the **save_all_tasks** method to write the tasks to the CSV file. For this, you should:

- Change the main module to use the **TaskCsvDAO** class to save the tasks to the file when the user selects the save option
- Replace the **None** values in this method with the correct values from the task object. You should use the **datetime.datetime.strftime(DATE_VAR, "%Y-%m-%d")** method to convert the **datetime** objects to strings and also use the **str.join** method to join the list of completed dates into a single string that is separated by a comma.
- Store the type of the task as a string in the CSV file by using a conditional with the **isinstance** method. If the task is an instance of the **RecurringTask** class, store "RecurringTask" in the type column. If it is an instance of the **Task** class, store "Task" in the type column.

(https://www.w3schools.com/python/ref_func_isinstance.asp) Note that you can add lines of code as needed.

After you have completed this task, run the **main.py** file and save the tasks to the CSV file. If you think it works, add a new task, and save the tasks again. Then open the CSV file and see whether the new task has been added to the file. If it has, you have successfully implemented the persistence using a CSV file. If you encounter any problems, use the debugger to step through the code and see what is happening at each step. Most of the time there will be a simple mistake such as incorrect variable types which you can easily fix. Otherwise, please speak to your lab tutor.

Optional Exercise 3: Serialization using Pickle

If you have time, you can also implement the serialization of the tasks using the **pickle** module. This is a module that allows you to serialize and deserialize Python objects (without needing to handle the attributes of the object individually). As you may see, this is much easier to use than the csv module, especially when you have complex objects and start changing the structure of the object as your codebase grows.

Have a look at how to use **pickle** here: <https://wiki.python.org/moin/UsingPickle>

Then, implement a new DAO called **TaskPickleDAO** that uses the pickle module to save and load the tasks to/from a file. Use the same method names but change the implementation to use the pickle module to save the full list of task objects to a file and load them from a file.