# COMP11124 Object Oriented Programming

## **Classes and Objects**

### Learning Outcomes

In this lab, you will advance your understanding of Python and will learn how to create Python classes and how to use them in your code. You will also learn how to add (and use) methods and attributes to your classes as well as be able to discuss how and why one should modularize and document their code.

### Topics Covered

Python: Classes, Methods, Objects, Modularizing code, documentation using docstrings and comments.

**Getting Started Task:** To retain all the code that you write and be able to work through the exercises in this lab, ***create a new Python file called:*** lab\_week\_4.py like how we did it last week. You can include the code of this week here and should continuously execute it to see the output. If you want to see the value of a variable, please *print* it using the print() function, as I have omitted that in the sample code.

# Python Classes

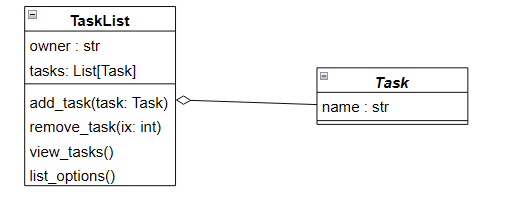
## Exercise 1: Creating Classes and Initializing Objects.

The last task of last week’s class was to create a To Do List program using functions.

This poses ideal to be turned into an OOP program.

Remember, we had a "ToDo list manager" loop that checked for our input, and we could add, remove, and list tasks.

Look at the following UML diagram to see an example of how this functionality could be implemented in an OOP program.



First, we have a class called TaskList. The top part shows us the Names of the classes, followed by their attributes. In this case, we have a Python list called tasks that will store all our tasks. We also have a string called owner, which will store the name of the owner of the task list.

This class handles the task-management functionality. It has four methods: add\_task, remove\_task, view\_tasks and list\_options:

- The add\_task method takes a parameter called task which should be a Task object. It then adds the task to the tasks list.

- The remove\_task method takes a parameter ix which is the index of the task that should be removed from the tasks list.

- The view\_tasks method prints all tasks in the tasks list.

- The list\_options method prints the options that the user must interact with the task list and handles the user input.

We also have a class called Task. To keep it simple, we do not have added methods (yet). We simply have included a task title as an attribute.

If you want to define a class in Python, you use the class keyword followed by the name of the class. In our case, this is as follows:

class TaskList:

We can now add the attributes to the class. To do this, we need to define them in the \_\_init\_\_ method. This method is always called when we create an object of the class. We can define the attributes in the \_\_init\_\_ method by using the following code:

class TaskList:

    def \_\_init\_\_(self, owner):

        self.owner = owner

        self.tasks = []

This then allows us to use the class and create new objects.

my\_task\_list = TaskList("John")

print(my\_task\_list.owner)

Note that we have passed the owner parameter when creating the object. This is because we have defined the owner attribute in the \_\_init\_\_ method and therefore need to pass a value for it when creating it. If you leave out the owner parameter, you will get an error.

Within the print statement, we have used the dot notation to access the owner attribute of the my\_task\_list object. This is because the owner attribute is an attribute of the my\_task\_list object.

So, if you create a new object of the TaskList class, you can access the owner attribute of that object by using the dot notation.

someone\_else\_task\_list = TaskList("Jane")

print(someone\_else\_task\_list.owner)

Within the \_\_init\_\_ method we can not only define attributes but also handle other logic such as calling methods. If your task list hypothetically wants to store the owner's name in uppercase, you could do this as follows:

class TaskList:

    def \_\_init\_\_(self, owner):

        self.owner = owner.upper()

        self.tasks = []

## Exercise 2: Adding Methods

Now that we have defined the attributes of the TaskList class, we can add the methods. Let us start with the add\_task method. As specified, the method takes a parameter called task which should be a Task object. It then adds the task to the tasks list.

Although we have not yet defined the Task class, we can still build the method. As lists allow us to store any type of data, we can simply assume that the task parameter will be a string (for now).

class TaskList:

    def \_\_init\_\_(self, owner):

        self.owner = owner.upper()

        self.tasks = []

    def add\_task(self, task):

        self.tasks.append(task)

If you want to define a method within a class, you will need to follow the same syntax as when defining a function. The only difference is that you need to add the self parameter as the first parameter. This is because the method is part of the class and therefore needs to have access to the attributes of the class.

The new method now allows us to add tasks to the tasks list. We access the object's tasks list by using the dot notation and use the self parameter to access the current object's tasks list. If you recall, the append method adds an item to the end of a list.

It is very important that you understand the self parameter. It is always a reference to the current instance of the class and is used to access variables that belong to the class.

We can now add the remove\_task method. This method takes a parameter ix which is the index of the task that should be removed from the tasks list. We can do this as follows:

class TaskList:

    def \_\_init\_\_(self, owner):

        self.owner = owner.upper()

        self.tasks = []

    def add\_task(self, task):

        self.tasks.append(task)

    def remove\_task(self, ix):

        # Your code here

**Task**: Add the code to the remove\_task method. Hint: You can use the del keyword to remove an item from a list. For example, if you wanted to remove the item at index 0 from a list called my\_list, you could do this as follows: del my\_list[0].

It is now up to you to add the view\_tasks method. This method should print all tasks in the tasks list.

**Task:** Add the code to the view\_tasks method. Rather than printing the tasks list directly, you should iterate over the tasks list and print each task individually and also include its index. *Hint*: You can use a for loop to iterate over the tasks list. (*Hint* 2: If you want to be very efficient, read up on the enumerate function in Python https://sparkbyexamples.com/python/for-loop-enumerate-in-python/ )

At last, we can add the list\_options method. This method prints the options that the user has to interact with the task list and handles the user input. For this, you can modify your solutions from last week or use the following code as a starting point:

    def list\_options(self):

        while True:

            print("To-Do List Manager")

            print("1. Add a task")

            print("2. View tasks")

            print("3. Remove a task")

            print("4. Quit")

            choice = input("Enter your choice: ")

            if choice == "1":

                task = input("Enter a task: ")

                self.add\_task(task)

            elif choice == "2":

                self.view\_tasks()

            elif choice == "3":

                ix = int(input("Enter the index of the task to remove: "))

                self.remove\_task(ix)

            elif choice == "4":

                break

## Exercise 3: Testing the Functionality

Although we do not have fully followed the UML diagram, we have now implemented the functionality of the TaskList class (albeit we still use a string for the task rather than a Task object).

To check whether you have coded the methods correctly, you can use the following code to create a new TaskList object and test the functionality in your script:

my\_task\_list = TaskList("YOUR NAME")

# This part is just to test the functionality by adding some tasks to the list

my\_task\_list.tasks = ["Do Homework", "Do Laundry", "Go Shopping"]

my\_task\_list.list\_options()

Try using the add, view and remove functionality and see whether it works as expected. If you have any problems, you can always look back at the previous examples or ask your lab tutor for help.

## Exercise 4: Composition

Composition is a way to combine objects or data types into more complex ones. Here, we are using this to integrate the Task class into the TaskList class (as a task is part of a task list).

For now, we have stored the tasks in the tasks list as strings. However, we now want to store them as Task objects. To be able to create Task objects, we need to define the Task class.

**Task:** Define the Task class. The class should have one attribute called title. The \_\_init\_\_ method should take one parameter called title which should be assigned to the title attribute of that object.

Now that we have defined the class, we can use it in the TaskLis. To do this, we need to change the choice where we add a task to the tasks list. Instead of adding the task as a string, we need to create a Task object and add it to the tasks list. We can do this as follows in the list\_options method:

if choice == "1":

    title = input("Enter a task: ")

    task = Task(title)

    self.add\_task(task)

Try out the functionality and see whether it works as expected.

You will most likely get an output similar to this:

0: <**\_\_main\_\_**.Task object at 0x000001D8F63B8410>

when you try to view the tasks. This is because we have not yet defined a method that returns a string representation of the Task object. We can do this by adding the following method to the Task class:

def \_\_str\_\_(self):

    return f"Task: {self.title}"

The \_\_str\_\_ method is a special method that returns a string representation of the object. This is useful when we want to print the object or convert it to a string. In the next lecture, we will look at the \_\_repr\_\_ method which is similar to the \_\_str\_\_ method but returns a string representation of the object that is more useful for debugging (but that is a topic for another time).

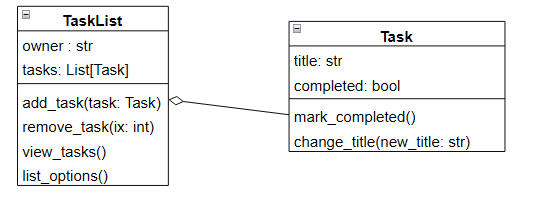
To test whether everything still works, replace the line of code with my\_task\_list.tasks with:

my\_task\_list.tasks = [Task("Do Homework"), Task("Do Laundry"), Task("Go Shopping")]

and run it.

Now it is time to add some more functionality.

Look at the following UML diagram to see what functionality we want to add to the Task class:



You will see that we have expanded the Task class to include an attribute called completed. This attribute should be a boolean and indicates whether the task has been completed or not. We have also added a method called mark\_completed which should change the completed attribute to True.

Additionally, we have also added a method called change\_title which should take a parameter called new\_title and change the title attribute to the new title.

**Task**: Change the code in the Task class accordingly and test the functionality.

You will need to add the completed attribute to the \_\_init\_\_ method and set it to False. You will also need to add the mark\_completed method and the change\_title method to the Task class, with functionality as described above (and do not forget to add the completed attribute to the task’s \_\_str\_\_ method).

**Task:** Now that we have added the functionality to the Task class, we can provide options for the user to utilize this functionality. For this, you will need to modify the list\_options method in the TaskList class.

Add the following options to the list\_options method:

- Mark a task as completed

- Change the title of a task

Note: You will need to add additional if statements to the list\_options method to handle the new options.

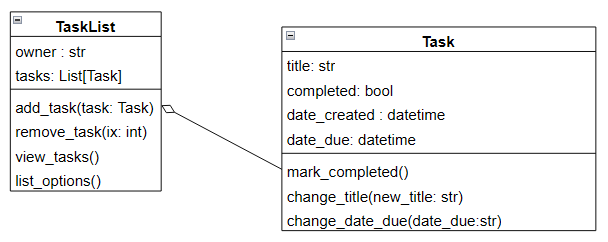
# Python Libraries

Python libraries are collections of functions and methods that allow you to perform actions, without having written the code yourself. Several are included in the standard Python installation, such as the math or datetime libraries.

## Exercise 1: Adding Dates

What would a to-do list be without dates? Let's add due dates and created dates to our tasks.

To do this, we change the Task class as in the following class diagram:



You will see that we have added two attributes to the Task class: created and due. The created attribute should store the date and time when the task was created. The due attribute should store the date and time when the task is due. Both attributes should be datetime objects. Additionally, we also add a change\_date\_due method which should take a parameter called date\_due and change the due attribute to the new date due.

One option would be to store any date as a string. However, this would make it difficult to perform any calculations with the dates. For example, if we wanted to calculate how many days are left until the due date, we would need to convert the string to a datetime object first. Luckily, Python has a library called datetime that allows us to work with dates and times.

To use the datetime library, we need to import it first. Now, it may be tempting to put the following code statement only where we need it, but it is good practice to put all import statements at the top of your script. This makes it easier to see which libraries you are using in your script.

Therefore, add the following import statement to the top of your script:

import datetime

Now that we have imported the datetime library, we can use it to create datetime objects. We can do this by using the datetime class from the datetime library. We can then use the datetime class to create a datetime object. For example, if we wanted to create a datetime object for the 1st of January 2021 at 12:00, we could do this as follows (this is an example, but does NOT need to be added to your script):

date = datetime.datetime(2021, 1, 1, 12, 0)

date = datetime.datetime.strptime("2021-01-01 12:00", "%Y-%m-%d %H:%M")

The latter example uses the strptime method from the datetime library. This method takes two parameters: the date as a string and the format of the date. You can find examples of this format here: <https://www.w3schools.com/python/python_datetime.asp>

Change the code in the Task class accordingly and test the functionality.

First, change the \_\_init\_\_ method to include the date\_due:

def \_\_init\_\_(self, title, date\_due):

and then add the attributes to the class:

self.date\_created = datetime.datetime.now()

self.date\_due = date\_due

Now, you should also change the TaskList class to use the functionality.

First, change the option where we add a task to the tasks list. Instead of adding the task as a string, we need to create a Task object with title and date\_due attributes and add it to the tasks list. We can do this as follows in the list\_options method (note, replace the code that you have added previously):

            if choice == "1":

                title = input("Enter a task: ")

                input\_date = input("Enter a due date (YYYY-MM-DD): ")

                date\_object = datetime.datetime.strptime(input\_date, "%Y-%m-%d")

                task = Task(title, date\_object)

                self.add\_task(task)

**Task**: In the same format as above, add the option to change the due date of a task. It is most likely that in your user options, you will need to add an additional option for editing a task, where you can group the option to change the title and the due date.

**Task:** Once you have added the functionality to change the due date of a task, please test the functionality. You want to ensure that your code works as expected. For example, if you change the due date of a task, you want to ensure that the due date is actually changed and that you can add or edit the tasks.

# Modularizing your Code

## Exercise 1: Restructuring

As discussed during the lecture, modularizing your code is an important aspect of programming. It allows you to break down your code into smaller, more manageable chunks. This makes it easier to read and understand your code and also makes it easier to extend your code in the future.

Modularizing our code allows us to have a single script that contains the main functionality of our program. This is called the main script. This is our entry point to the program and is the script that we run to start the program.

We can then move the Task and TaskList classes into separate scripts (e.g. files), called modules. These modules can then be imported into the main script, tying the functionality together.

Whilst just now, it may seem like a lot of work to modularize your code, it will pay off in the long run.

**Task:** Create a new folder called ToDoApp wherever you want to store your code. In this folder, create a new file called main.py. This will be our main script. Then create two new files called tasks.py and task\_list.py. These will be our modules.

Fill the tasks.py file with the Task class and the task\_list.py file with the TaskList class code

Do not forget to import the datetime library in both modules. When you fill the TaskList you may also see a squiggly line under the Task class. This is because the Task class is not yet defined in the TaskList module. We will fix this by importing the Task class into the TaskList module.

Add the following import statement to the top of the task\_list.py file:

from tasks import Task

At the end, your file structure should look like this:

*ToDoApp/*

*main.py*

*tasks.py*

*task\_list.py*

## Exercise 2: Main()

Now let us populate the main entry point of our program, the main.py file.

First, we need to import the TaskList class from the modules. (Note: we do not need to import the Task class as it is not directly used here) Then we can build a main() function that contains the main functionality of our program.

This is followed by only executing the main() function if the script is run directly. This is important because if we import the main.py file into another script, we do not want the main() function to be executed. Therefore, we place the call to this function into a conditional:

from tasklist import TaskList

def main():

    task\_list = TaskList("YOUR NAME")

    task\_list.list\_options()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Task:** Test whether your code works as expected. You should be able to run the main.py file and see the options to interact with the task list.

In OOP we want to ensure that our classes are as independent as possible. This means that we want to avoid having to change multiple classes if we change something in one class. A lot of the ToDo logic is however still within the TaskList class within the list\_options method.

Let us take those instructions out of the TaskList class and move them into the main.py file. The rationale for this is that whilst the TaskList class is responsible for managing the tasks, it is not responsible for the user interaction. This could be the responsibility of another class (maybe one that builds a GUI), but for now, we will simply move it into the main.py file.

To do this, copy all the code from the list\_options method in the TaskList class into the main() function in the main.py file. Then, remove the list\_options method from the TaskList class. Ensure that in the main() function you have fixed the indentation of the code that you have copied over.

Then remove the line:

    task\_list.list\_options()

as we do not need to call this method anymore (since we moved the logic).

If you are using VS Code you may see a few yellow squiggly lines under the code that you have copied over. This is because the code is now outside of the TaskList class and therefore does not have access to the self parameter.

For this, we can employ a very easy fix. Simply replace all instances of self. with task\_list. (or whatever you have called your TaskList object). This will ensure that the code still works as expected.

(Note: If you are using VS Code you can simply double click on the self keyword and then press CTRL + F2 to select all instances of the self keyword. You can then simply type task\_list. and it will replace all instances of self with task\_list.)

Now you will need to import the required datetime and Task classes:

from tasks import Task

import datetime

To allow us to test the program without always manually adding tasks, we can add some code to the main() function that adds some tasks to the task list.

Copy the following code into the main.py file:

def propagate\_task\_list(task\_list: TaskList) -> TaskList:

    """Propagates a task list with some sample tasks.

    Args:

        task\_list (TaskList): Task list to propagate.

    Returns:

        TaskList: The propagated task list.

    """

    task\_list.add\_task(Task("Buy groceries", datetime.datetime.now() - datetime.timedelta(days=4)))

    task\_list.add\_task(Task("Do laundry", datetime.datetime.now() - datetime.timedelta(days=-2)))

    task\_list.add\_task(Task("Clean room", datetime.datetime.now() + datetime.timedelta(days=-1)))

    task\_list.add\_task(Task("Do homework", datetime.datetime.now() + datetime.timedelta(days=3)))

    task\_list.add\_task(Task("Walk dog", datetime.datetime.now() + datetime.timedelta(days=5)))

    task\_list.add\_task(Task("Do dishes", datetime.datetime.now() + datetime.timedelta(days=6)))

    return task\_list

Additionally, add the following code to the main() function in the line below the while True statement:

    # propagate the task list with some sample tasks

    task\_list = propagate\_task\_list(task\_list)

This allows us to test the program without having to manually add tasks to the task list.

**Task:** Whilst the actual functionality of the program has not changed, we have now modularized our code further. To ensure that everything still works as expected, test your code again. You should be able to run the main.py file and see the options to interact with the task list (but this time the logic mainly happens in the main.py file rather than in the TaskList class).

# Type Checking and Documenting your Code.

## Exercise 1: Type Checking

Type checking is an important aspect of programming. It allows you to ensure that the data you are using is of the correct type. For example, if you want to add two numbers together, you need to ensure that both numbers are of the type int or float. If you try to add a string to an int, you will get an error. Other programming languages such as Java or C++ require you to specify the type of a variable when you define it. This allows us to catch errors early on. Python on the other hand is a dynamically typed language. Dynamically typed languages do not require you to specify the type of a variable when you define it. This allows you to be more flexible but also means that you need to be more careful when using variables.

However, it is very good practice to add type hints to your code. Type hints allow you to specify the type of a variable or function parameter. This makes it easier to use an IDE such as VS Code to check whether you are using the correct type (since it will show you an error if you are not). It also makes it easier for other people to understand your code.

You will have seen examples of type checking in the lecture already. Here is a very good small cheat sheet that you can refer to if you are unsure how to add type hints to your code: <https://mypy.readthedocs.io/en/stable/cheat_sheet_py3.html>

Generally, the syntax is as follows:

variable\_name: type

This can be added to the variable definition or any function/method parameter. For example, if we wanted to add type hints to the Task class, we could do this as follows:

class Task:

    def \_\_init\_\_(self, title: str, date\_due: datetime.datetime):

    # remaining code

What we specify here is that the title parameter should be a string and the date\_due parameter should be a datetime.datetime object.

We can also specify custom types. For example, in our tasklist in the add\_task method, we want to ensure that the task parameter is a Task object. We can do this as follows:

    def add\_task(self, task: Task) -> None:

    # more code.

You may also see the arrow syntax -> . This is used to specify the return type of a function or method. In this case, we specify that the add\_task method does not return anything (None).

In some cases, we have a method that will return something, such as the \_\_str\_\_ method in the Task class. We can specify this as follows:

def \_\_str\_\_(self) -> str:

    # more code.

**Task:** Ensure that all your classes and functions have type hints. If you are unsure about the type of a variable, you can always use the type() function to check the type of a variable.

Therefore, add type hints for the return types of all methods and functions and for all parameters and variables where possible:

- all methods in the Task class

- all methods in the TaskList classs

- the main function in the main.py file

**Please Note:** Depending on your computer setup, the follow part may not work smoothly. As type-checking before running code is optional (but recommended) you can skip this if you experience an error.

To allow us to check whether we have added type hints correctly, we can use a tool called mypy. This is a static type checker for Python. Follow the guide here: <https://mypy.readthedocs.io/en/stable/getting_started.html> to install mypy on your computer. If you are using your own PC, you only need to do this once. If you are using the UWS computers, you will need to do this every time you work on a new computer/class.

To use mypy, you simply need to run the following command in the terminal (at the bottom of VS Code):

mypy --strict main.py

This will check whether you have added type hints correctly. If you have not, you will see error messages telling you where you have made a mistake. You can then go back to your code and fix the errors.

If everything is correct, you will see something akin to this:

A computer screen with green text

Description automatically generated

## Exercise 2: Docstrings and Comments

Another important aspect of programming is to document your code. This makes it easier for other people to understand your code and makes it easier for you to understand your code if you have not looked at it for a while.

There are two main ways of documenting your code: docstrings and comments.

For docstrings, it may be worth reading the first part of this guide: <https://peps.python.org/pep-0257/> to understand what they are and how they are used.

Docstrings are used to document classes, functions, and methods. They are placed right after the definition of the class, function or method and are enclosed in triple quotes. For example, if we wanted to add a docstring to the Task class, we could do this as follows:

class Task:

    """Represents a task in a to-do list. <-- this is a class docstring.

    """

    def \_\_init\_\_(self, title: str, date\_due: datetime.datetime):

        """Creates a new task. <-- this is a method docstring.

        Args:

            title (str): Title of the task.

            date\_due (datetime.datetime): Due date of the task.

        """

    # some other code

    def mark\_completed(self) -> None:

        """Marks the task as completed."""

        self.completed = True

Generally, the information that you should include in a docstring is:

- a description of what the class, function or method does

- a description of the parameters and their types

- a description of the return type (if applicable)

There is no set format for docstrings, but it is important that you are consistent and that you include all the information that is required to understand the class, function or method. If in doubt, use the example above as a starting point.

**Task:** Add docstrings to all classes, functions and methods in your code. Ensure that you have included all the information that is required to understand the class, function or method. You do not need to add docstrings to the main.py file or the \_\_init\_\_ and \_\_str\_\_ methods of the Task and TaskList classes.

Another form of documentation are comments. Comments are used to explain parts of your code. They are generally used to explain why you have done something in a certain way. For example, if you have used a certain algorithm, you may want to explain why you have used this algorithm.

Comments are created by using the # symbol. Anything after the # symbol is ignored by Python. For example, if we wanted to add a comment to the main function in the main.py file, we could do this as follows:

        if choice == "1":

            title = input("Enter a task: ")

            input\_date = input("Enter a due date (YYYY-MM-DD): ")

            date\_object = datetime.datetime.strptime(input\_date, "%Y-%m-%d")

            # create a new task object based on the title entered and the date entered

            task = Task(title, date\_object)

            task\_list.add\_task(task)

You should use comments sparingly and only use them to explain why you have done something in a certain way or it is not obvious what a certain part of your code does.

# Portfolio Exercises

The following portfolio exercises will form part of the coursework portfolio you will need to submit. There will be no solutions provided to those exercises after the class, but you should complete them to the best of your ability.

## Portfolio Exercise 1

Add a description attribute to the Task class. This should be a string that describes the task but is entirely optional for the user of your program to provide. For this, you should:

- add the description attribute to the Task class and allow for it to be passed as a parameter to the \_\_init\_\_ method.

- add a method called change\_description that allows the user to change the description of a task

- change the \_\_str\_\_ method to include the description of the task

- change the main() function to allow the user to change the description of a task in choice 4 where the user can also change the title and due date of a task

## Portfolio Exercise 2

Add a method to the TaskList class that allows the user to view all overdue tasks. For this, you should:

- add a method called view\_overdue\_tasks that prints all tasks that are overdue based on the current date.

- change the main() function to allow the user to view all overdue tasks in an additional choice

A visualisation of this is included in the following UML diagram, with Portfolio Exercise 1’s additions being underlinedand Portfolio Exercise 2's additions being *italic.*

