Unit 4
Encapsulation and polymorphism

Encapsulation is the process of bundling data and methods within one unit. A class may be an example of data encapsulation, when creating a class, instance variables and class specific methods.

Encapsulation in code could look like this:

```
class Person:
    # constructor
    def __init__(self, first_name, last_name, profession):
        # data members
        self.first_name = first_name
        self.last_name = last_name
        self.profession = profession

    # method
    # to display the person's name
    def show(self):
        # accessing public data member
        print("Name: ", self.first_name, " " + self.last_name)

    # method
    def work(self):
        print(self.first_name, 'is a', self.profession +'.')


# creating object of a class
emp = Person('Maja', 'Tagt', 'student')

# calling public method of the class
emp.show()
emp.work()
```

By encapsulating data, access can be restricted to methods and variables and gives the creator of the system greater control of an objects internal state. Controlling the visibility of data is achieved in Python by underscores; referred to as access modifiers.

```
class Dog:
    def __init__(self, name, breed, cost):
        self.name = name
        self._breed = breed
        self.__cost = cost
```

the name of the dog is accessible both outside and inside the class, (public data member), its breed is only available within the class and potential sub-classes. The cost of the dog is only accessible within the class (private member).

Getters and setters are used in OOB to ensure proper data encapsulation. The getter method is used to access data members, whilst setter method is used to modify data members. Primary purpose of getters and setters are to avoid direct access to private data members, and to add validation logic for setting a value.

A coding example could look like this:

```
class Phone:
    def __init__(self, model, storage, megapixels):
        self._model = model
        self._storage = storage
        self._megapixels = megapixels

    def get_model(self):
        return self._model

    def get_storage(self):
        return self._storage

    def get_megapixels(self):
        return self._megapixels

    def set_model(self, new_model):
        self._model = new_model

    def set_storage(self, new_storage):
        self._storage = new_storage

    def set_megapixels(self, new_megapixels):
        self._megapixels = new_megapixels


my_phone = Phone("iPhone", 256, 12)
print(my_phone.get_model())
my_phone.set_model("Galaxy S20")
print(my_phone.get_model())
```

Polymorphism permits an object to exist in many forms.

Method overriding
For example, polymorphism permits a method with the same name to exist in a parent class and child class to perform the same task, however slightly different. In code, it could look like this:

```
class Alpha:
    def show(self):
        print("I am from class Alpha")


class Bravo(Alpha):
    def show(self):
        print("I am from class Bravo")


test_object = Alpha()
test_object.show()
```

By changing the test_object to be Bravo, the output would be

```
I am from class Bravo
```

Method Overloading
Is the same method that can take different parameters. Python however does not support it and will raise a TypeError if a method is overloaded. When two (or more) methods have the same name, Python will only recognise the last method. Instead, a method with optional parameters can be created.

Operator overloading
Entails changing the default behaviour of an operator, depending on the values. One operator can be used for multiple purposes.

The + operator will perform an arithmetic addition operator when used with numbers and perform concatenation when used with strings. In code it could look like this:

```
# add 2 numbers
print(15 + 15)

# concatenate two strings
print('Maja' + 'Tagt')

# merge two lists
print([5, 10, 15] + ['Maja', 'Karro', 'Tagt'])
```

Outputs:

```
30
MajaTagt
[5, 10, 15, 'Maja', 'Karro', 'Tagt']
```

Operator + for customer objects

To add two objects with a binary + operator, is not directly possible in Python – it will raise an error. To circumvent it, an __add__() method can be created. Example in code would look like this:

```
class Folder:
    def __init__(self, pages):
        self.pages = pages

    # Overloading + operator with magic method
    def __add__(self, other):
        return self.pages + other.pages

b1 = Folder(400)
b2 = Folder(300)
print("Total number of pages: ", b1 + b2)
```

Overloading the * operator

The * operator is implemented by the __mul__() method. In code, it could look like this:

```
class Employee:
    def __init__(self, name, salary):
        self.name = name
        self.salary = salary

    def __mul__(self, timesheet):
        print('Worked for', timesheet.days, 'days')
        # calculate salary
        return self.salary * timesheet.days


class TimeSheet:
    def __init__(self, name, days):
        self.name = name
        self.days = days


emp = Employee("Maja", 1000)
timesheet = TimeSheet("Maja", 55)
print("salary is: ", emp * timesheet)
```

Duck typing

Duck typing is used to determine the suitability of an object based on what it does and is related to dynamically typed programming languages. It comes from the saying *"If it walks like a duck and talks like a duck, then it must be a duck."* If it acts like a duck, it is a duck. Duck typing is an example of polymorphism because it only matters if it quacks, not if it actually is a duck.

Magic methods in Python to overload methods:

Operator Name	Symbol	Magic method
Addition	+	__add__(self, other)
Subtraction	-	__sub__(self, other)
Multiplication	*	__mul__(self, other)
Division	/	__div__(self, other)
Floor Division	//	__floordiv__(self,other)
Modulus	%	__mod__(self, other)
Power	**	__pow__(self, other)
Increment	+=	__iadd__(self, other)
Decrement	-=	__isub__(self, other)
Product	*=	__imul__(self, other)
Division	/+	__idiv__(self, other)
Modulus	%=	__imod__(self, other)
Power	**=	__ipow__(self, other)
Less than	<	__lt__(self, other)
Greater than	>	__gt__(self, other)
Less than or equal to	<=	__le__(self, other)
Greater than or equal to	>=	__ge__(self, other)
Equal to	==	__eq__(self, other)
Not equal	!=	__ne__(self, other)