**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



**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:

**Graphical user interface, text

Description automatically generated**

**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) |  |
|  |  |  |  |