Implementing Methods in a Class

In this lesson, you will get to know about the role of methods in classes and what method overloading is.

WE'LL COVER THE FOLLOWING

Now that we have learned about adding properties to a class, we will move towards the interaction between these properties and other objects. This is where methods come into play. There are three types of methods in Python:

1. **instance methods**
2. **class methods**
3. **static methods**

We will be discussing the *instance methods* in this lesson since they are used the most in Python OOP.

> **Note:** We will be using the term **methods** for **instance methods** in our course since they are most commonly used. **Class methods** and **static methods** will be named explicitly as they are.

The Purpose of Methods

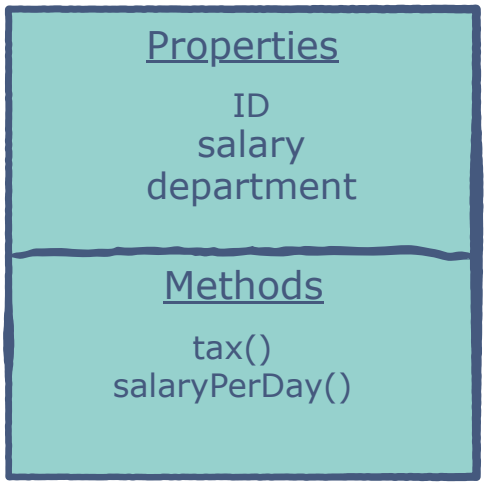
Methods act as an interface between a program and the properties of a class in the program.

These methods can either alter the content of the properties or use their values to perform a particular computation.

Definition and Declaration

> A **method** is a group of statements that performs some operations and may or may not return a result.

We will extend the `Employee` class example that we studied in the previous lesson by adding methods to it.



Employee class

Method Parameters

Method **parameters** make it possible to pass values to the method. In Python, the first parameter of the method should ALWAYS be `self` (will be discussed below) and this is followed by the remaining parameters.

Return Statement

The **return** statement makes it possible to get the value from the method.

> There is no need to specify the return type since data types are not

specified in Python.

The return statement must be immediately followed by the return value.

The `self` Argument

One of the major differences between functions and methods in Python is the first argument in the method definition. Conventionally, this is named `self`. The user can use different names as well, but `self` is used by almost all the developers working in Python. We will also be using this convention for ease of understanding.

This pseudo-variable provides a reference to the calling object, that is the object to which the method or property belongs to. If the user does not mention the `self` as the first argument, the first parameter will be treated for reference to the object.

> **Note:** The `self` argument only needs to be passed in the method definition and not when the method is called.

Below is an example of implementing methods in a class:

```
class Employee:
    # defining the initializer
    def __init__(self, ID=None, salary=None, department=None):
        self.ID = ID
        self.salary = salary
        self.department = department

    def tax(self):
        return (self.salary * 0.2)

    def salaryPerDay(self):
        return (self.salary / 30)


# initializing an object of the Employee class
Steve = Employee(3789, 2500, "Human Resources")

# Printing properties of Steve
print("ID =", Steve.ID)
print("Salary", Steve.salary)
print("Department:", Steve.department)
print("Tax paid by Steve:", Steve.tax())
print("Salary per day of Steve", Steve.salaryPerDay())
```

Method Overloading

> Overloading refers to making a method perform different operations based on the nature of its arguments.

Unlike in other programming languages, methods **cannot** be explicitly overloaded in Python but can be implicitly overloaded.

In order to include optional arguments, we assign default values to those arguments rather than creating a duplicate method with the same name. If the user chooses not to assign a value to the optional parameter, a default value will automatically be assigned to the variable.

```
class Employee:
    # defining the properties and assigning them None to the
    def __init__(self, ID=None, salary=None, department=None):
        self.ID = ID
        self.salary = salary
        self.department = department

    # method overloading
    def demo(self, a, b, c, d=5, e=None):
        print("a =", a)
        print("b =", b)
        print("c =", c)
        print("d =", d)
        print("e =", e)

    def tax(self, title=None):
        return (self.salary * 0.2)

    def salaryPerDay(self):
        return (self.salary / 30)


# cerating an object of the Employee class
Steve = Employee()

# Printing properties of Steve
print("Demo 1")
Steve.demo(1, 2, 3)
print("\n")

print("Demo 2")
Steve.demo(1, 2, 3, 4)
print("\n")

print("Demo 3")
Steve.demo(1, 2, 3, 4, 5)
```

In the code above, we see the same method behaving differently when encountering different types of inputs.

If we redefine a method several times and give it different arguments, Python uses the latest method definition for its implementation.

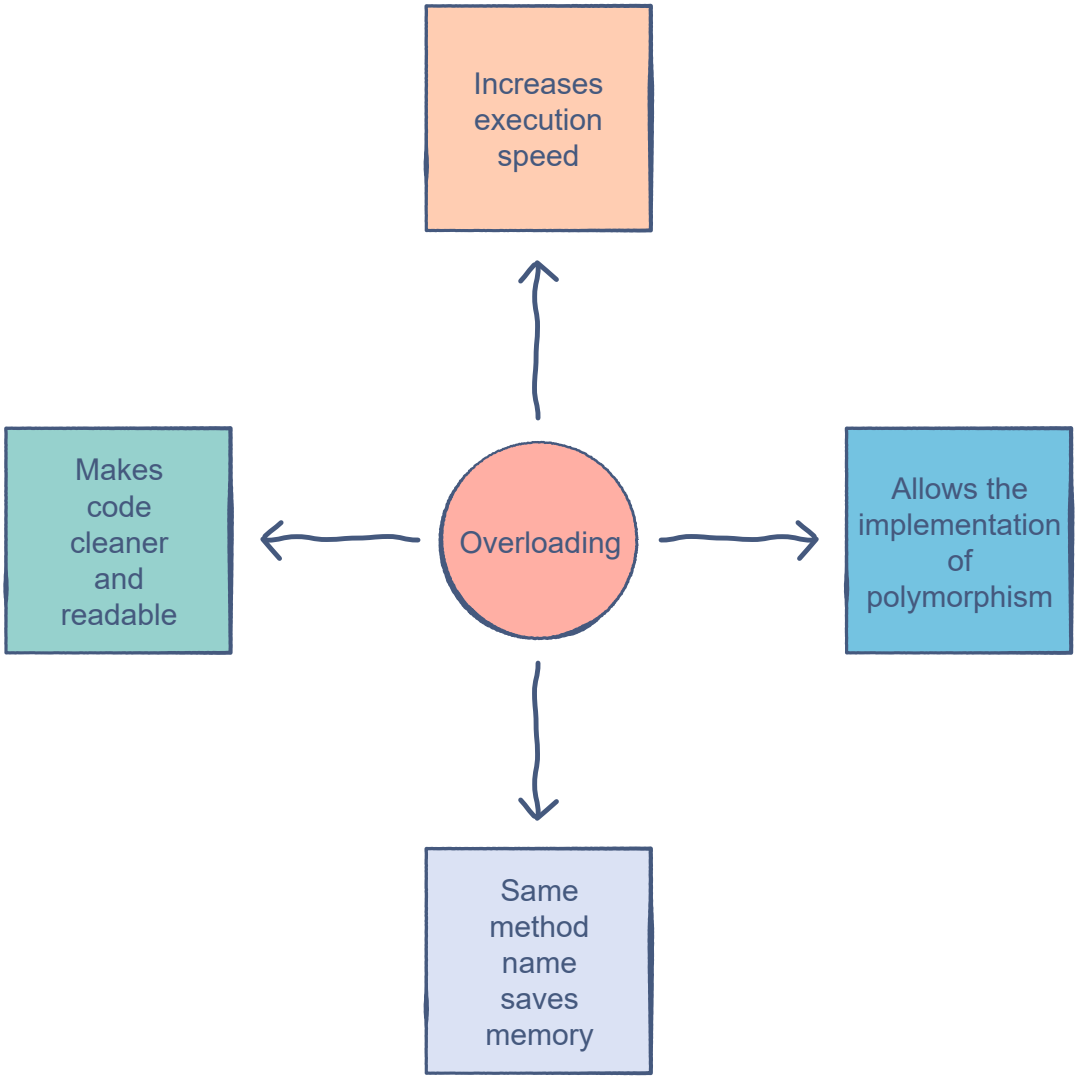
Advantages of Method Overloading

One might wonder that we could simply create new methods to perform different jobs rather than overloading the same method. However, under the hood, overloading saves us memory in the system. Creating new methods is costlier compared to overloading a single one.

Since they are memory-efficient, overloaded methods are compiled faster compared to different methods, especially if the list of methods is long.

An obvious benefit is that the code becomes simple and clean. We don’t have to keep track of different methods.

Polymorphism is a very important concept in object-oriented programming. It will come up later on in the course, but method overloading plays a vital role in its implementation.



In the next lesson, we will discuss the utility of class methods and static methods.