IIT Madras
BSc Degree

# Object Oriented Programming

## Classes and Objects

Let us continue with the `Student` class. For now, don't bother too much about the keyword `self`. We will get to that soon.

```python
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks

    def update_marks(self, marks):
        self.marks = marks

    def print_details(self):
        print(f'{self.name}:{self.marks}')
```

As we saw at the end of the previous lesson, an object of the class `Student` can be created like this:

```python
anish = Student('Anish', 80)
```

Notice that we have used the name of the class in the RHS of the assignment statement. This invokes what is called the **constructor** — `__init__` method — of the class. Since the constructor has two parameters (ignore `self` for now) `name` and `marks`, we have to pass them as arguments while creating the object.

The two arguments are then assigned to `self.name` and `self.marks` respectively. These two variables are called the **attributes** of the object. Attributes can be accessed using the `.` operator:

```python
print(anish.name)
print(anish.marks)
```

`__init__`, `update_marks` and `print_details` are called **methods**. A method is effectively just another function. Methods can be accessed using an object. If we wish to update Anish's marks to $95$, then we invoke the method using the object `anish`:

```
anish.update_marks(95)
```

When `anish.update_marks(95)` is called, the attribute `marks` that is tied to the object `anish` is updated to $95$.

To summarize, `anish` is an object of type `Student` having two attributes — `name` and `marks` — that can be accessed using the `.` operator. This object is also equipped with two methods (ignoring the constructor), one to update the marks and the other to print the details of the object. Attributes define the state of an object. Different objects of the same class could have different attributes. Naively speaking, methods help to update the values of the attributes. Therefore, the methods capture the behaviour of the object.

## `self`

Some of you might be wondering about the variable `self` that crops in so many places in the definition of the class. The variable `self` is used to point to the current object. To get a better understanding, let us create two different students (objects):

```
anish = Student('Anish', 90)
lakshmi = Student('Lakshmi', 95)
```

How do we print the details of the student Lakshmi?

```
lakshmi.print_details()
```

When this method is called, Python actually ends up invoking the following function:

```
Student.print_details(lakshmi)
```

That is, it passes the current object as an argument. So, the variable `self` points to the current object. Another example:

```
anish.update_marks(95)
```

This is equivalent to the function call:

```
Student.update_marks(anish, 95)
```

This is a mechanism that Python uses to know the object that it is dealing with. And for this reason, the first parameter in every method defined in a class will be `self`, and it will point to the current object.

This should also clear up any confusion that lines 3 and 4 could have caused:

```python
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks
```

`self.name = name` is the following operation: assign the value of the argument `name` to the current object's attribute `self.name`. A similar operation follows for `self.marks`.

## Class Attributes vs Object Attributes

So far all attributes that we have seen are object attributes. Given an attribute, say `name` or `marks`, it is different for different objects. The `name` attribute of `anish` is different from the corresponding attribute fo the object `lakshmi`. Now, we shall see another kind of attribute.

Let us say that we wish to keep track of the number students in our program. That is, when a new student joins our program, we need to update a counter. How do we do that? We need an attribute that is common to all objects and is not tied to any individual object. At the same time, we should be able to update this attribute whenever a new object is created. This is where the concept of class attributes comes in:

```python
class Student:
    counter = 0
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks
        Student.counter += 1

    def update_marks(self, marks):
        self.marks = marks

    def print_details(self):
        print(f'{self.name}:{self.marks}')
```

Now, let us say that three students join the program:

```python
madhavan = Student('Madhavan', 90)
print('Number of students in the program =', Student.counter)
andrew = Student('Andrew', 85)
print('Number of students in the program =', Student.counter)
usha = Student('Usha', 95)
print('Number of students in the program =', Student.counter)
```

This gives the following output:

```
Number of students in the program = 1
Number of students in the program = 2
Number of students in the program = 3
```

Notice that we have used `Student.counter` to access the attribute `counter`. Such attributes are called "class attributes". All objects of the class share this attribute. At this stage, we can try the following exercise:

```python
print(madhavan.counter)
```

A class attribute can be accessed by any of the objects. But, now, try to run this code:

```python
1  madhavan.counter = -1
2  print(Student.counter)
3  print(madhavan.counter)
```

This seems confusing! But a moment's thought will convince you that it is not so hard. In line-1, we are creating an object attribute with the same name as the class attribute! If the same attribute name occurs in both an object and a class, then Python prioritizes the object attribute. This demonstrates an important fact: class attributes cannot be updated by an object! At best, they can be referenced or accessed using an object.

This also introduces another important point: object attributes can be created dynamically during runtime. So far, we have seen object attributes being created within the constructor. This is not binding. For example, consider the following snippet:

```python
class Student:
    def __init__(self, name):
        self.name = name

anish = Student('Anish')
anish.maths = 100
anish.physics = 90
anish.chem = 70
```

We have created three more object attributes on the fly. It is interesting to note the subtle difference between the attribute `name` and the three new attributes `maths`, `physics` and `chem`. Any object of `Student` will have the attribute `name` when it is initially created, of course with a different value for `name` depending on the object. But the attributes `maths`, `physics` and `chem` are unique to the object `anish`.