Machine Learning Diploma

Session 4 : Object Oriented Programming Language

AMIT

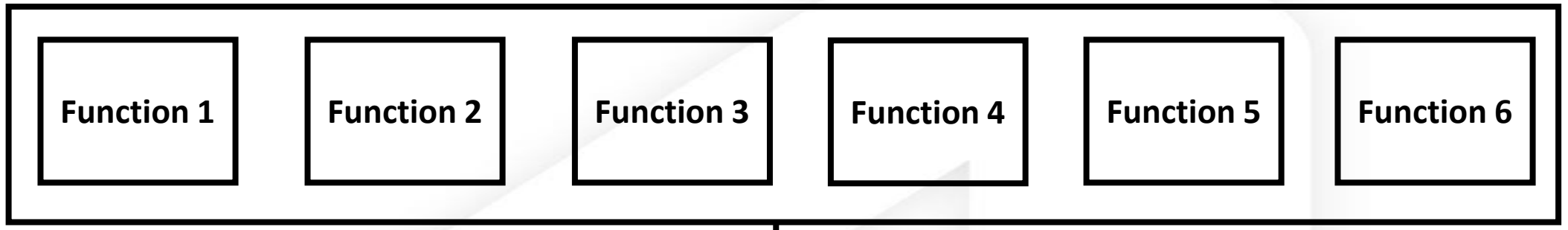
Agenda:

1	Introduction to OOP
2	Constructor
3	Abstraction
4	Encapsulation
5	Polymorphism
6	Inheritance
7	Multiple Inheritance

1. Introduction To Object Oriented Programming (OOP)

What is Object Oriented Programming (OOP)

Object-Oriented Programming (OOP) is a programming paradigm that structures code using objects, which bundle data and methods.



Class

Attributes

Data - characteristics

Methods

Functions

Why OOP?

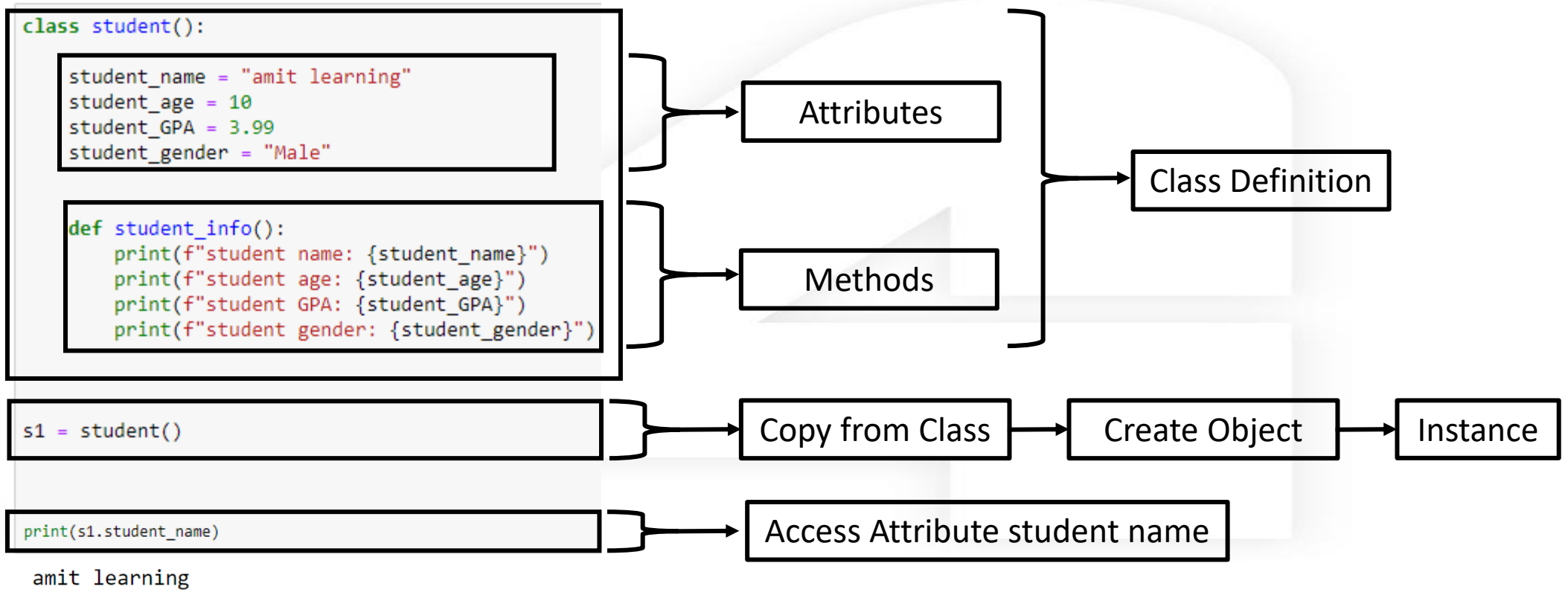
1. It simplifies complex systems

2. enhances maintainability

3. supports the creation of scalable and adaptable software.

4. OOP is commonly used for developing large-scale applications where structured design is essential.

How To Access Attribute



modify the value of an Attribute

```python
class student():
    # attributes
    student_name = "amit learning"
    student_age = 10
    student_GPA = 3.99
    student_gender = "Male"

    # methods
    def student_info():
        print(f"student name: {student_name}")
        print(f"student age: {student_age}")
        print(f"student GPA: {student_GPA}")
        print(f"student gender: {student_gender}")
```

Class Definition

```python
s1 = student()
```

Instance

```python
s1.student_name = "Sameh Albadry"
```

Edit Attribute

```python
print(s1.student_name)
```

Print Attribute After Editing

```
Sameh Albadry
```

AMIT

Why shouldn't we directly change attributes from the class?

Why not edit the class directly?

- Editing the class would affect all objects created from it.

- Objects allow us to work with different data and behaviors independently.

- It's like having multiple copies of a blueprint, each customized to represent something specific.

```python
class student():
    # attributes
    student_name = "amit learning"
    student_age = 10
    student_GPA = 3.99
    student_gender = "Male"

    # methods
    def student_info():
        print(f"student name: {student_name}")
        print(f"student age: {student_age}")
        print(f"student GPA: {student_GPA}")
        print(f"student gender: {student_gender}")


s1 = student()


print(s1.student_name)
```

```
amit learning
```

Class student()

```python
    # attributes
    student_name = "amit learning"
    student_age = 10
    student_GPA = 3.99
    student_gender = "Male"

    # methods
    def student_info():
        print(f"student name: {student_name}")
        print(f"student age: {student_age}")
        print(f"student GPA: {student_GPA}")
        print(f"student gender: {student_gender}")


print(s1.student_name)
```

```
amit learning
```

s1 (copy from class) (Instance)

How can a method be accessed?

```python
class student():
    # attributes
    student_name = "amit learning"
    student_age = 10
    student_GPA = 3.99
    student_gender = "Male"

    # methods
    def student_info():
        print("hello world")
```

Class Definition

```python
s1 = student()
```

Object From Class

```python
s1.student_info()
```

Implement method inside class

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[8], line 15
     10         print("hello world")
     12 s1 = student()
---> 15 s1.student_info()

TypeError: student.student_info() takes 0 positional arguments but 1 was given
```

TypeError

Why?

How To Access Method

In Python's object-oriented programming paradigm, instance methods within a class conventionally include a reference to the instance itself through the first parameter. This allows the method to access

```
class student():
    # attributes
    student_name = "amit learning"
    student_age = 10
    student_GPA = 3.99
    student_gender = "Male"

    # methods
    def student_info(x):
        print("hello world")

s1 = student()

s1.student_info()
```

```
hello world
```

How To Access Method

The use of self in Python's object-oriented programming serves as a convention to refer to the instance of the class within its methods. When you define a method inside a class and include **self** as its first parameter, you are essentially telling Python that this method is meant to operate on an instance of the class.

```
class student():
    # attributes
    student_name = "amit learning"
    student_age = 10
    student_GPA = 3.99
    student_gender = "Male"

    # methods
    def student_info(self):
        print("hello world")

s1 = student()


s1.student_info()
```

```
hello world
```

Prove that self variable is a reference to the instance of the class.

```
class test():
    def info(self):
        print(self)

x = test()
print(x)
x.info()
```

```
<__main__.test object at 0x0000027BA4B92790>
<__main__.test object at 0x0000027BA4B92790>
```

∵ Instance Address = Self Address

∴ self variable is a reference to the instance of the class.

AMIT

Object Oriented Programming Language

Constructor	Abstraction	Encapsulation
A constructor in Python is a special method named __init__ that automatically initializes the attributes of an object when it's created. It's used to set initial values and ensure the object starts with a predefined state upon instantiation.	A destructor in Python, denoted by the __del__ method, is automatically called before an object is destroyed. However, it's not recommended for critical cleanup due to its unpredictable timing. Alternative methods like context managers or explicit cleanup functions are preferred for reliable resource management.	Encapsulation in Python's OOP is about bundling data and methods within a class, hiding internal details, and controlling access. It promotes code organization, reduces interference, and ensures a modular and maintainable structure.
Polymorphism	**Inheritance**	**Multiple Inheritance**
Polymorphism in Python allows objects to have multiple forms. It involves method overloading (compile-time) using default arguments or variable-length lists and method overriding (run-time) for a common interface to represent various object types. Enhances code flexibility and reusability in OOP.	Inheritance in Python enables a class to inherit attributes and methods from another class, promoting code reuse and creating a class hierarchy. It supports method overriding and uses super() for parent class method calls. Enhances code organization and modeling relationships.	Multiple Inheritance in Python allows a class to inherit from more than one parent class. It introduces the "diamond problem" and uses the C3 linearization algorithm for Method Resolution Order (MRO). Careful design is crucial to avoid ambiguity, and mixins are often used for specific functionality in multiple inheritance scenarios.

2. Constructor

Constructor

In Python, a constructor is like a helper that sets up an object automatically when you create it. It uses a special method called __init__ to give the object its starting values and make sure it begins in a specific state.

```
class student():
    # method
    def __init__(self):
        print("hello world")

s1 = student()
```

```
hello world
```