

Object Oriented Programming Language

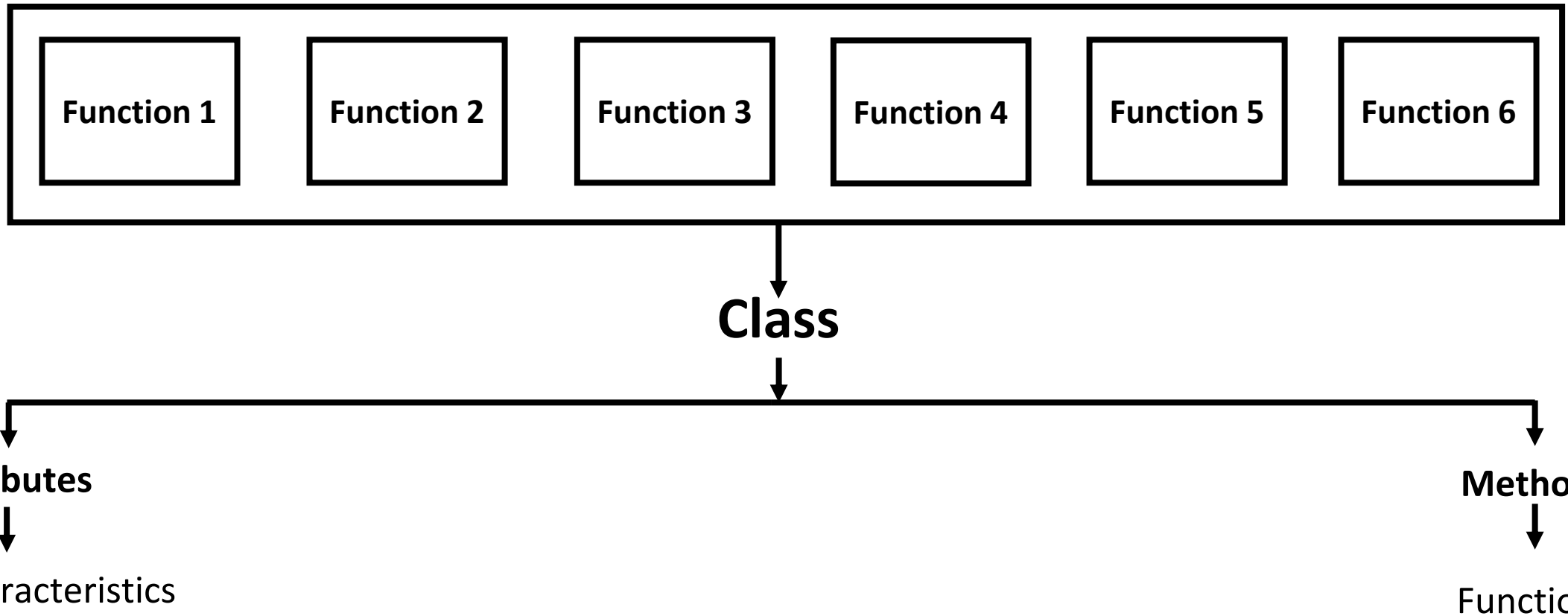
Agenda:

| | |
|----------|-----------------------------|
| 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)

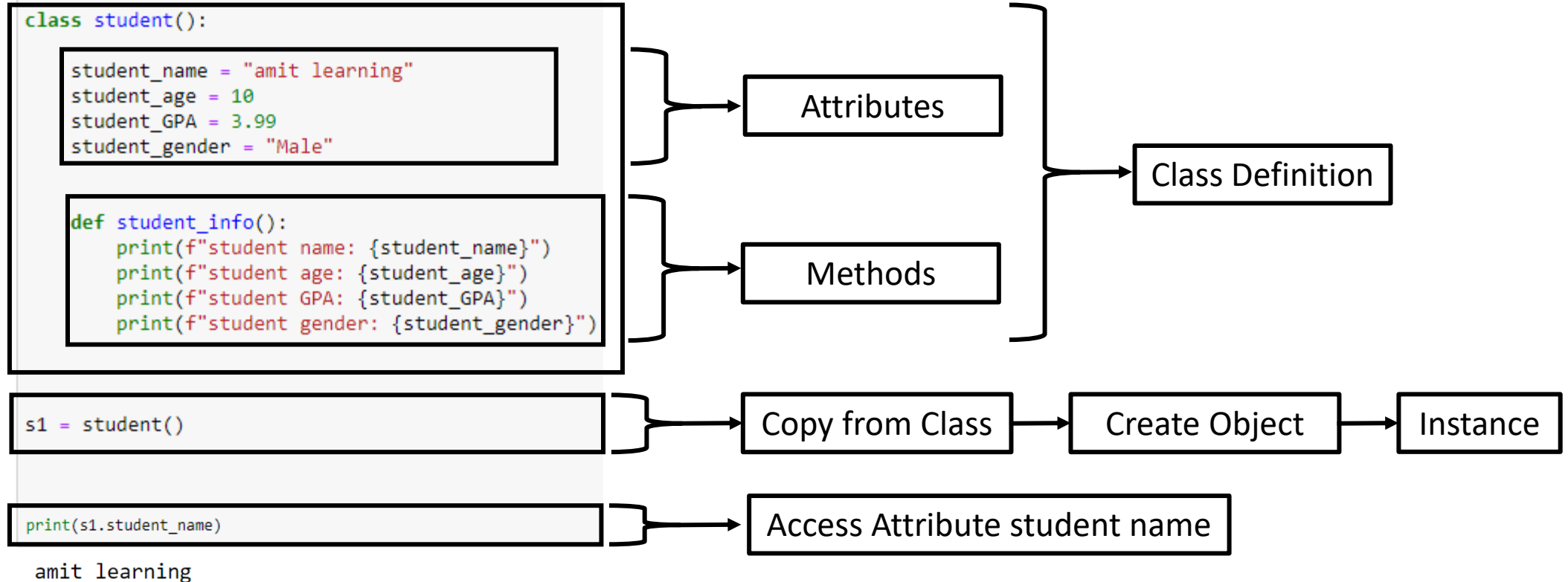
Object-Oriented Programming (OOP) is a programming paradigm that structures code using objects, which bundle data and methods.



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

```
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

```
s1 = student()
```

Instance

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

Edit Attribute

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

Print Attribute After Editing

Sameh Albadry

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.

```
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()

```
# 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?

```
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

```
s1 = student()
```

Object From Class

```
s1.student_info()
```

Implement method inside class

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

Traceback (most recent call last)

TypeError

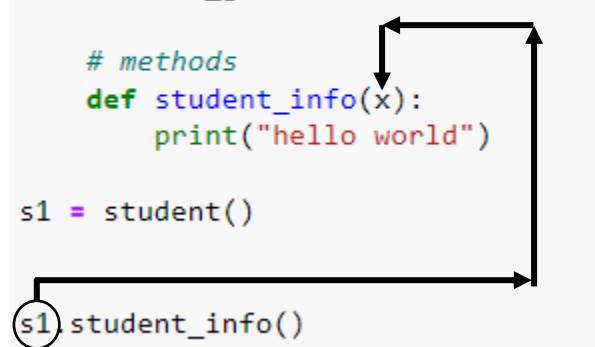
```
TypeError: student.student_info() takes 0 positional arguments but 1 was given
```

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

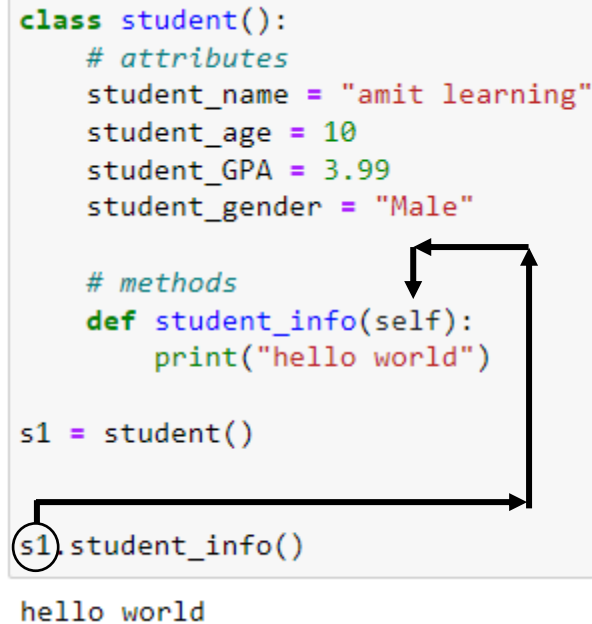
```
class student():  
    # attributes  
    student_name = "amit learning"  
    student_age = 10  
    student_GPA = 3.99  
    student_gender = "Male"  
  
    # methods  
    def student_info(x):  
        print("hello world")  
  
s1 = student()  
  
s1.student_info()  
  
hello world
```



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.

Object Oriented Programming Language

| Constructor | Abstraction | Encapsulation |
|--|---|---|
| <p>A constructor in Python is a special method named <code>__init__</code> 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.</p> | <p>A destructor in Python, denoted by the <code>__del__</code> 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.</p> | <p>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.</p> |
| Polymorphism | Inheritance | Multiple Inheritance |
| <p>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.</p> | <p>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 <code>super()</code> for parent class method calls. Enhances code organization and modeling relationships.</p> | <p>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.</p> |

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
```


3. Abstraction

Abstraction

Abstraction in python is defined as a process of handling complexity by hiding unnecessary information from the user.

```
from abc import ABC , abstractmethod

class Animal(ABC):
    @abstractmethod
    def move(self):
        pass

class Bird(Animal):
    def move(self):
        print("move from bird")

class Cat(Animal):
    def move(self):
        print("move from Cat")

a = Bird()

a.move()

move from bird
```

Abstraction

Abstraction in python is defined as a process of handling complexity by hiding unnecessary information from the user.

```
from abc import ABC , abstractmethod

class Animal(ABC):
    @abstractmethod
    def move(self):
        pass

class Bird(Animal):
    def move(self):
        print("move from bird")

class Cat(Animal):
    def move(self):
        print("move from Cat")

a = Animal()

a.move()
```

```
-----
TypeError                                 Traceback (most recent call last)
Cell In[21], line 16
     13     def move(self):
     14         print("move from Cat")
--> 16 a = Animal()
     18 a.move()
```

TypeError: Can't instantiate abstract class Animal with abstract method move

```
from abc import ABC , abstractmethod

class Animal(ABC):
    @abstractmethod
    def move(self):
        pass

class Bird(Animal):
    pass

class Cat(Animal):
    def move(self):
        print("move from Cat")

a = Bird()

a.move()
```

```
-----
TypeError                                 Traceback (most recent call last)
Cell In[22], line 15
     12     def move(self):
     13         print("move from Cat")
--> 15 a = Bird()
     17 a.move()
```

TypeError: Can't instantiate abstract class Bird with abstract method move

4. Encapsulation

Encapsulation

Encapsulation is the practice of hiding the internal data of a class and exposing only what is necessary for the outside world to interact with the class.

Private Attribute

Private variables and methods are those that can only be accessed within the class.

Public Attribute

Public variables and methods are those that can be accessed by any part of the program.

Private Attribute

Private variables and methods are those that can only be accessed within the class.

```
class Car:
    # private Attribute
    __engine_capacity = "2000cc"

    # private method
    def __start_engine(self):
        print("Engine started!")

c1 = Car()

print(c1.__engine_capacity)

c1.__start_engine()
```

```
-----
AttributeError                                Traceback (most recent call last)
Cell In[3], line 11
      7         print("Engine started!")
      9 c1 = Car()
----> 11 print(c1.__engine_capacity)
      14 c1.__start_engine()

AttributeError: 'Car' object has no attribute '__engine_capacity'
```

Public Attribute

Public variables and methods are those that can be accessed by any part of the program.

```
class Rectangle:
    length = 5
    width = 10

    def area(self):
        return self.length * self.width

r = Rectangle()

print(r.length)
print(r.width)
print(r.area())
```

```
5
10
50
```

5. Polymorphism

Polymorphism

Polymorphism is a concept in object-oriented programming that allows objects of different types to be treated as objects of a common type.

```
class Shape:
    def calculate_area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * (self.radius)**2

class square(Shape):
    def __init__(self, side):
        self.side = side

    def calculate_area(self):
        return self.side ** 2

x = Circle(5)
y = square(4)

print("Area: ", x.calculate_area())
print("Area: ", y.calculate_area())
```

Area: 78.5

Area: 16

6. Inheritance

Inheritance

Inheritance is a fundamental concept in object-oriented programming (OOP) where a new class (subclass/derived class) can inherit attributes and behaviors from an existing class (superclass/base class).

```
class Calculator1:
    def summation(self,n1,n2):
        return n1+n2

    def subtraction(self,n1,n2):
        return n1-n2

    def multiplication(self,n1,n2):
        return n1*n2

    def division(self,n1,n2):
        return n1/n2

class Calculator2(Calculator1):

    def power(self,n1,n2):
        return n1**n2

c1 = Calculator2()
print(c1.summation(2,5))
```

7. Multiple Inheritance

Multiple Inheritance

Multiple Inheritance is a concept in object-oriented programming where a class can inherit attributes and behaviors from more than one parent class.

```
class A():
    def do_this(self):
        print("I am in A")

class B(A):
    pass

class C():
    def do_this(self):
        print("I am in C")

class D(B,C):
    pass

x = D()
x.do_this()
```

I am in A

```
class A():
    def do_this(self):
        print("I am in A")

class B(A):
    pass

class C():
    def do_this(self):
        print("I am in C")

class D(C,B):
    pass

x = D()
x.do_this()
```

I am in C

Thank You