

CLASSES AND OBJECTS. CONSTRUCTOR. CREATING OBJECT
ATTRIBUTES**Onarqulov Maqsadjon Karimberdiyevich**Associate Professor, Department of Applied
Mathematics and Informatics, Fergana State University

E-mail: maxmaqsad@gmail.com

Odina Isroiljon qizi Ismoiljonova3rd-year student of the Applied Mathematics program,
Group 22-08, Fergana State University
E-mail: ismoiljonovaodina88@gmail.com

Annotation: This article explores the core concepts of object-oriented programming — including classes, objects, constructors, and the process of creating object attributes. It emphasizes the practical importance of the object-oriented approach in enabling proper software structuring and code reusability. Each concept is explained with clear examples in Python, making the material accessible for beginner programmers.

Keywords: Class, object, constructor, attribute, Python, object-oriented programming, __init__ method, methods, encapsulation

Аннотация: В данной статье рассматриваются основные понятия объектно-ориентированного программирования — классы, объекты, конструкторы и процесс создания атрибутов объектов. Подчёркивается практическая значимость объектно-ориентированного подхода для правильного проектирования программной структуры и повторного использования кода. Все понятия

сопровожаются понятными примерами на языке Python, что делает материал доступным для начинающих программистов.

Ключевые слова: Класс, объект, конструктор, атрибут, Python, объектно-ориентированное программирование, метод `__init__`, методы, инкапсуляция

Annotatsiya: Ushbu maqolada obyektga yo'naltirilgan dasturlashning asosiy tushunchalari — klasslar, obyektlar, konstruktorlar va obyekt atributlarini hosil qilish jarayonlari yoritiladi. Dasturchilar uchun dasturiy strukturalarni to'g'ri loyihalash va kodni qayta ishlatishga imkon beruvchi obyektga yo'naltirilgan yondashuvning amaliy ahamiyati tushuntiriladi. Har bir tushuncha Python dasturlash tilida aniq misollar bilan izohlangan bo'lib, yangi o'rganuvchilar uchun tushunarli tarzda bayon etilgan.

Kalit so'zlar: Klass, obyekt, konstruktor, atribut, Python, obyektga yo'naltirilgan dasturlash, `__init__` metodi, metodlar, encapsulation (inkapsulyatsiya)

Introduction

In modern software engineering, object-oriented programming (OOP) plays a central role in solving complex computational problems by modeling real-world systems through modular and reusable components. The fundamental constructs of OOP — such as classes, objects, constructors, and attributes — allow developers to encapsulate data and behavior into organized structures. This article aims to provide a comprehensive overview of these concepts using examples in the Python programming language, making it accessible for both beginners and those seeking to deepen their understanding of OOP.

What is a Class?

In object-oriented programming (OOP), a class is an abstract, user-defined data type that serves as a template or blueprint for creating individual objects. It

encapsulates data for the object and defines behaviors that the objects created from the class can exhibit. A class typically combines two main components:

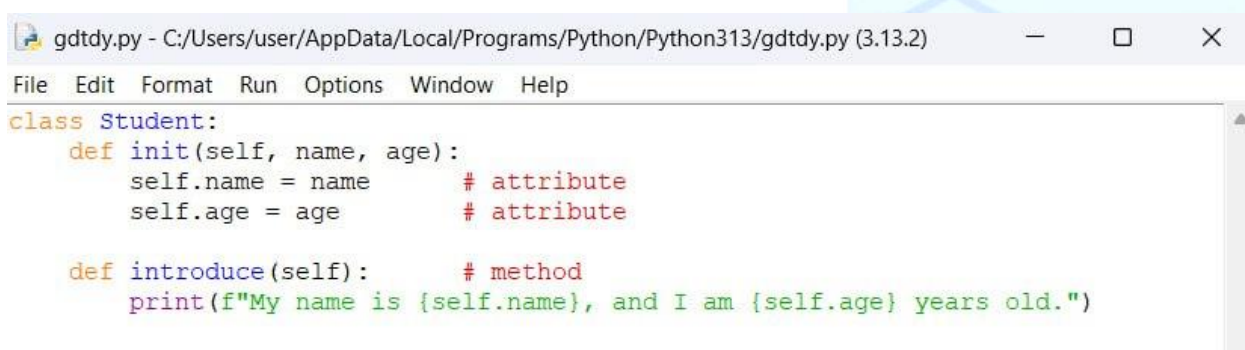
Attributes (also known as fields or properties): variables that hold the state or characteristics of the object.

Methods: functions that define the behaviors or operations that can be performed on the object.

Thus, a class represents both structure (what data is stored) and functionality (what operations can be done on that data) in a unified framework.

Structure of a Class (in Python syntax)

A simple class in Python is declared using the class keyword:



```
gtdty.py - C:/Users/user/AppData/Local/Programs/Python/Python313/gtdty.py (3.13.2)
File Edit Format Run Options Window Help
class Student:
    def init(self, name, age):
        self.name = name      # attribute
        self.age = age        # attribute

    def introduce(self):      # method
        print(f"My name is {self.name}, and I am {self.age} years old.")
```

Here:

Student is the name of the class.

__init__() is the constructor, a special method that is called when a new object is instantiated.

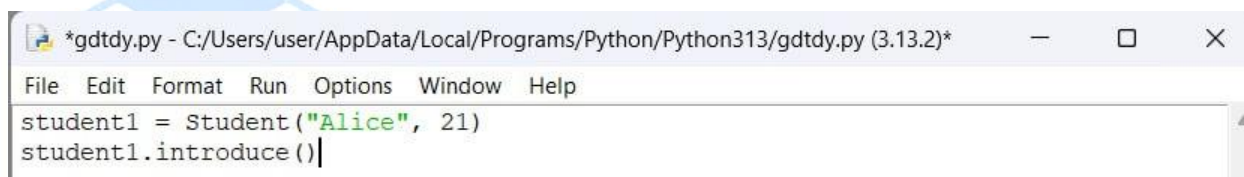
self.name and self.age are attributes.

introduce() is a method.

This class defines the structure and behavior of a “Student” object.

Instantiation of Objects

A class does not hold data itself — instead, it defines how data should be organized. An object (or instance) is a concrete occurrence of a class with actual data assigned to its attributes.



```
*gtdtdy.py - C:/Users/user/AppData/Local/Programs/Python/Python313/gtdtdy.py (3.13.2)*
File Edit Format Run Options Window Help
student1 = Student("Alice", 21)
student1.introduce()
```

Each time a class is instantiated, a new object is created in memory with its own copy of the attributes defined in the class.

Scientific Benefits of Using Classes

Abstraction: Hides internal implementation and shows only relevant details.

Encapsulation: Bundles data and related methods into a single unit, restricting direct access to some components.

Inheritance: Classes can be extended into subclasses, promoting code reuse.

Polymorphism: Classes can define methods that behave differently depending on the object's type or context.

These principles collectively enable systematic, modular, and scalable software development — a cornerstone of modern engineering practices in computing.

What is an Object?

In object-oriented programming (OOP), an object is a concrete instance of a class — a self-contained unit that bundles both data (attributes or fields) and behavior (methods or functions). While a class defines the structure and capabilities, the object is the actual entity that occupies memory and can perform actions.

Formally, an object is characterized by:

A unique identity

A defined state (via attribute values)

A defined behavior (via callable methods)

The object serves as a model of a real-world entity, encapsulating both static properties and dynamic operations relevant to that entity.

Structure of an Object

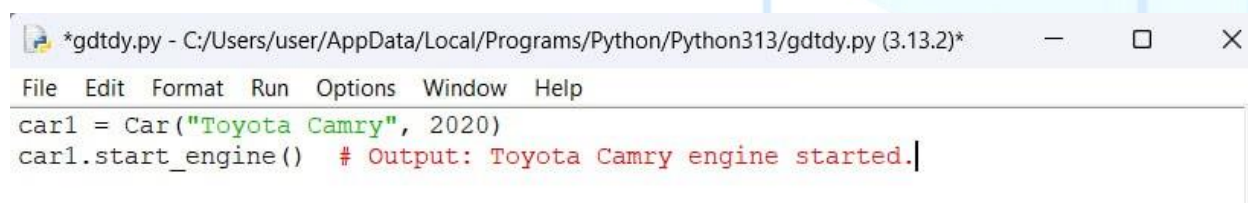
In Python and many other OOP languages, objects are created (or instantiated) from classes. Consider the following class:



```
*gtdty.py - C:/Users/user/AppData/Local/Programs/Python/Python313/gtdty.py (3.13.2)*
File Edit Format Run Options Window Help
class Car:
    def init(self, model, year):
        self.model = model
        self.year = year

    def start_engine(self):
        print(f"{self.model} engine started.")
```

Creating an object:



```
*gtdty.py - C:/Users/user/AppData/Local/Programs/Python/Python313/gtdty.py (3.13.2)*
File Edit Format Run Options Window Help
car1 = Car("Toyota Camry", 2020)
car1.start_engine() # Output: Toyota Camry engine started.
```

Here:

car1 is an object of class Car.

It has its own model and year values (state).

It can invoke the method start_engine() (behavior).

Each object has a separate memory address and can function independently of other objects of the same class.

Scientific Characteristics of Objects

1. Identity

Every object has a unique identity during its lifetime. This identity distinguishes it from all other objects.

2. State

The state is defined by the values stored in the object's attributes. The state can change during the object's lifetime via methods.

3. Behavior

The behavior of an object is defined by the methods associated with its class. Methods operate on the object's state and can interact with other objects.

4. The Constructor Method (`__init__`)

In Python and many object-oriented programming (OOP) languages, a constructor is a special method used to initialize newly created objects. In Python, this method is called `__init__`. It is invoked automatically when a class is instantiated, and it sets up the object's initial state by assigning values to its attributes.

Theoretical Foundations

From a theoretical standpoint, the constructor method supports the principle of encapsulation and plays a vital role in ensuring object integrity and state consistency upon creation. It is essential for:

Initialization of instance-specific data.

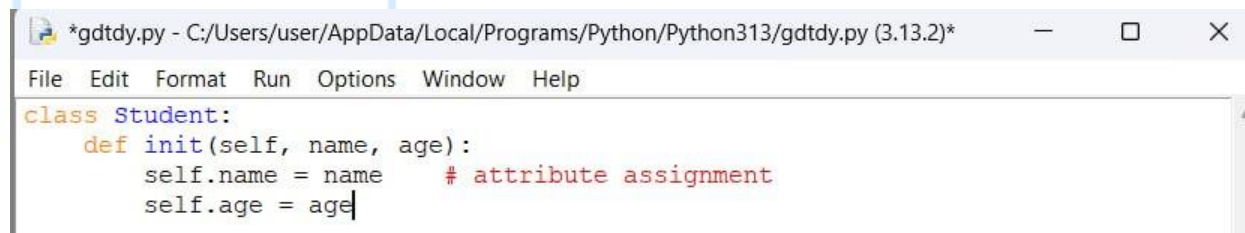
Establishing invariants, i.e., conditions that must hold true throughout the object's lifetime.

Providing overloaded behavior, allowing parameterized construction of diverse object configurations.

In formal terms, the constructor can be seen as part of the object creation protocol, separating memory allocation from initialization logic.

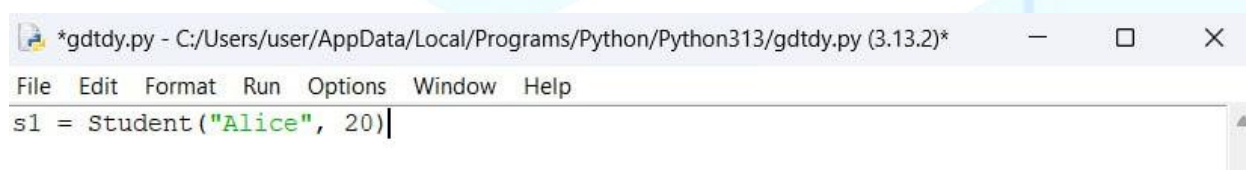
Syntax and Structure (Python)

The constructor in Python has the following structure:



```
*gtdtdy.py - C:/Users/user/AppData/Local/Programs/Python/Python313/gtdtdy.py (3.13.2)*
File Edit Format Run Options Window Help
class Student:
    def __init__(self, name, age):
        self.name = name    # attribute assignment
        self.age = age
```

Object instantiation and initialization:



```
*gtdtdy.py - C:/Users/user/AppData/Local/Programs/Python/Python313/gtdtdy.py (3.13.2)*
File Edit Format Run Options Window Help
s1 = Student("Alice", 20)
```

Here:

The call to `Student("Alice", 20)` creates an object.

The `__init__` method is automatically invoked with `self`, `"Alice"`, and `20` as arguments.

The instance variables `self.name` and `self.age` are initialized.

Functionality of `__init__`

The `__init__` method serves as:

A parameterized constructor, accepting arguments during object creation.

A mechanism for assigning default values or conducting pre-processing before an object is used.

A gateway for dependency injection, allowing external resources or configurations to be passed into the object.

Example with default values:

```
*gtdty.py - C:/Users/user/AppData/Local/Programs/Python/Python313/gtdty.py (3.13.2)*
File Edit Format Run Options Window Help
class Circle:
    def init(self, radius=1.0):
        self.radius = radius
```

Scientific and Practical Significance

1. State Integrity

The constructor guarantees that all attributes of an object are properly initialized before the object is accessed, ensuring consistency in object behavior.

2. Encapsulation and Modularity

By controlling how an object is initialized, the constructor enforces encapsulation, and allows abstraction by hiding internal initialization complexity.

3. Overload Flexibility

Although Python does not natively support constructor overloading, default parameters and conditional logic within `__init__` can simulate multiple constructor behaviors, allowing for flexible instantiation patterns.

4. Integration with Class Design

A well-designed constructor complements the Single Responsibility Principle from software engineering: it should only initialize the object, avoiding complex business logic.

5. Object Attributes

Object attributes are the data members or properties that belong to an object and represent its state or characteristics. Attributes store information that is specific to each individual instance of a class and help define the unique identity and current condition of that object.

In OOP, attributes are typically represented as variables bound to the object. They distinguish one object from another, even if the objects are instances of the same class.

Theoretical Background

Attributes are fundamental to the principle of encapsulation in OOP, as they:

Hold the object's internal state.

Are usually accessed and modified through methods (getters and setters) to maintain control over how data changes.

Support data hiding by making attributes private or protected, preventing unauthorized external access.

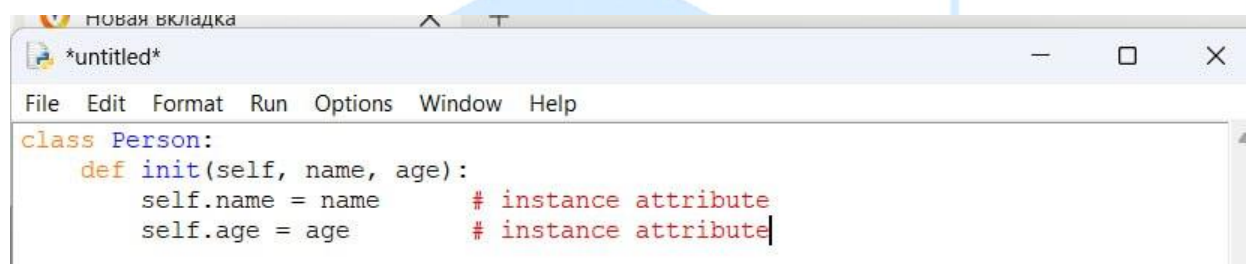
From an abstract viewpoint, attributes define the state space in which an object operates, and the changes in attribute values reflect the dynamics of the object's behavior.

Types of Attributes

InstanceAttributes

These are attributes that belong to a specific object instance. Each object has its own copy, and changes to one object's attributes do not affect others.

Example in Python:



```
class Person:
    def init(self, name, age):
        self.name = name      # instance attribute
        self.age = age        # instance attribute
```

ClassAttributes

These attributes are shared among all instances of a class. They represent properties

common to all objects.

```
*untitled*
File Edit Format Run Options Window Help
class Person:
    species = "Homo sapiens" # class attribute
```

PrivateAttributes

By convention, attributes prefixed with underscores (_ or __) are treated as private or protected, restricting direct external access

```
*untitled*
File Edit Format Run Options Window Help
class Person:
    def init(self, name):
        self.__name = name # private attribute
```

Attribute Access and Modification

Attributes are generally accessed and modified through methods (also called accessors and mutators):

Getters retrieve attribute values.

Setters update attribute values, often with validation logic.

Example:

```
*untitled*
File Edit Format Run Options Window Help
class Person:
    def init(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

    def set_name(self, new_name):
        if isinstance(new_name, str):
            self.__name = new_name
```

Scientific Importance of Attributes

Encapsulation and Abstraction: Attributes encapsulate data, enabling abstraction by exposing only necessary information through controlled interfaces.

State Management: Attributes reflect the dynamic state of an object, crucial for behavior and decision-making processes.

Data Integrity: Proper management (e.g., through setters) ensures attribute values remain valid and consistent.

6. Encapsulation (inkapsulyatsiya)

Encapsulation is one of the fundamental principles of object-oriented programming (OOP). It refers to the process of bundling data (attributes) and methods (functions) that operate on that data into a single unit, called an object. Encapsulation also restricts direct access to some of an object's components, which is a means of data hiding and helps protect the integrity of the object's internal state.

Theoretical Background

Encapsulation provides a mechanism for data hiding and controlled access to the internal state of an object. It ensures that:

The internal representation (state) of an object is shielded from direct modification by external code.

Access to the object's data is only possible through well-defined interfaces (methods).

This reduces complexity, increases security, and preserves the consistency of the object's state by preventing unauthorized or unintended interference.

Mechanisms of Encapsulation

Programming languages implement encapsulation through:

Access Modifiers:

public — accessible from anywhere.

private — accessible only within the class itself.

protected — accessible within the class and its subclasses.

Getter and Setter Methods

Special methods that control access to private attributes, enabling validation, logging, or other processing during data retrieval or modification.

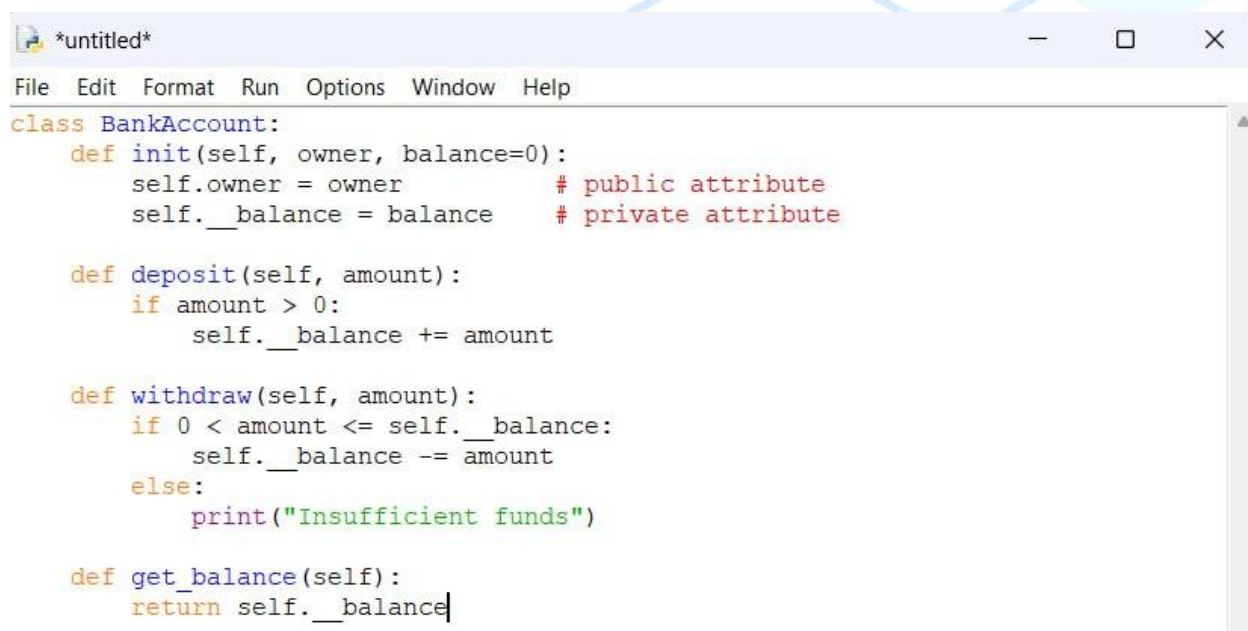
Scientific and Practical Importance

Data Integrity and Security: Encapsulation safeguards an object's data from unauthorized or incorrect modification.

Abstraction: It hides complex internal details, exposing only what is necessary for interaction.

Maintainability and Scalability: Encapsulation promotes modularity, making code easier to maintain and extend without unintended side effects.

Example in Python



```
class BankAccount:
    def init(self, owner, balance=0):
        self.owner = owner          # public attribute
        self.__balance = balance    # private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount

    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds")

    def get_balance(self):
        return self.__balance
```


Bu misolda __balance atributi private hisoblanadi va to'g'ridan-to'g'ri tashqi koddan o'zgartirilishi mumkin emas. Faqat maxsus metodlar orqali unga kirish va boshqarish mumkin.

Conclusion

The fundamental concepts of object-oriented programming—classes, objects, constructors, object attributes, and encapsulation—play a crucial role in organizing efficient and maintainable software systems. Classes serve as blueprints for creating objects, which are individual instances possessing unique states. Constructors ensure that objects are initialized properly and consistently, thereby maintaining data integrity within the program. Object attributes define the specific characteristics of each object, enabling the representation of diverse real-world entities. Encapsulation protects data by bundling attributes and methods together and restricting direct external access, which enhances security, modularity, and maintainability.

Together, these principles facilitate the development of software that is reusable, scalable, and adaptable to complex problem domains. Mastering the concepts of classes and objects, along with their supporting mechanisms, is essential for modern programmers aiming to build robust, flexible, and reliable applications.

References

- Booch, Grady. Object-Oriented Analysis and Design with Applications. 3rd Edition. Addison-Wesley, 2007.
— Comprehensive coverage of OOP concepts, including classes, objects, and encapsulation.
- Stroustrup, Bjarne. The C++ Programming Language. 4th Edition. Addison-Wesley, 2013.
— Detailed explanation of OOP principles and constructors in C++.

Lippman, Stanley B., Lajoie, Josée, Moo, Barbara E. C++ Primer. 5th Edition. Addison-Wesley, 2012.

— Fundamentals of classes and attributes in object-oriented programming.

Schildt, Herbert. Java: The Complete Reference. 11th Edition. McGraw-Hill, 2018.

— Practical guide on encapsulation and constructors in Java.

Gamma, Erich, Helm, Richard, Johnson, Ralph, Vlissides, John. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

— Classic resource for design patterns and advanced OOP concepts.

Sebesta, Robert W. Concepts of Programming Languages. 11th Edition. Pearson, 2015.

— Theoretical background on programming languages, including OOP and encapsulation.

Python Software Foundation. Python Language Reference, version 3.9. Available at:

<https://docs.python.org/3/reference/>

— Official documentation on constructors and attributes in Python.