```python
"""
Encapsulation in Python
Kapsülleme, soyutlama, kalıtım ve polimorfizm dahil nesne yönelimli
programlamadaki (OOP) temel kavramlardan biridir .
Python'da kapsülleme , verileri ve yöntemleri tek bir birim içinde
paketleme kavramını tanımlar . Örneğin, bir sınıf oluşturduğunuzda , bu,
kapsülleme uyguladığınız anlamına gelir. Bir sınıf, tüm veri üyelerini
( örnek değişkenleri ) ve yöntemleri tek bir birime bağladığı için bir
kapsülleme örneğidir .Bu örnekte, ad ve maaş gibi çalışan niteliklerini bir
örnek değişken olarak tanımlayarak ve work()ve show()örnek yöntemlerini
kullanarak davranış uygulayarak bir Çalışan sınıfı oluşturuyoruz."""
class Employee:
    # constructor
    def __init__(self, name, salary, project):
        # data members
        self.name = name
        self.salary = salary
        self.project = project

    # method
    # to display employee's details
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)

    # method
    def work(self):
        print(self.name, 'is working on', self.project)

# creating object of a class
emp = Employee('Jessa', 8000, 'NLP')
# calling public method of the class
emp.show()
#Name:  Jessa Salary: 8000
emp.work()
"""
#   Jessa is working on NLP
# ================================================================================
# Kapsüllemeyi kullanarak, bir nesnenin iç temsilini dışarıdan gizleyebiliriz.
# Buna bilgi gizleme denir.Ayrıca kapsülleme, değişkenlere ve yöntemlere doğrudan
# erişimi kısıtlamamıza ve bir sınıf içinde özel veri üyeleri ve yöntemler
# oluşturarak yanlışlıkla veri değişikliğini önlememize olanak tanır.
# Kapsülleme, sınıf dışından yöntemlere ve değişkenlere erişimi kısıtlamanın bir
# yoludur. Sınıfla çalışırken ve hassas verilerle uğraşırken, sınıf içinde
# kullanılan tüm değişkenlere erişim sağlamak iyi bir seçim değildir.
# Örneğin, bir nesnenin dışından görünmeyen bir özniteliğiniz olduğunu ve bunu
# okuma veya yazma erişimi sağlayan yöntemlerle bir araya getirdiğinizi
# varsayalım. Bu durumda, belirli bilgileri gizleyebilir ve nesnenin dahili
# durumuna erişimi kontrol edebilirsiniz. Kapsülleme, programa bir sınıfın tüm
# değişkenlerine tam kapsamlı erişim sağlamadan gerekli değişkene erişmemiz için
# bir yol sunar. Bu mekanizma, bir nesnenin verilerini diğer nesnelerden korumak
# için kullanılır.
# ================================================================================
# ================================================================================
# Access Modifiers in Python
# Kapsülleme, bir sınıfın veri üyelerini ve yöntemlerini özel veya korumalı
# olarak bildirerek gerçekleştirilebilir.But In Python, we don’t have direct
```

```
# access modifiers like public, private, and protected. We can achieve this by
# using single underscore and double underscores.
# ============================================================================
Access modifiers limit access to the variables and methods of a class.
Python provides three types of access modifiers private, public, and protected.
Public Member: Accessible anywhere from otside oclass.
Private Member: Accessible within the class
Protected Member: Accessible within the class and its sub-classes
Public Member
Public data members are accessible within and outside of a class. All member
variables of the class are by default public."""
class Employee:
    def __init__(self, name, salary):
        # public data members
        self.name = name
        self.salary = salary
    # public instance methods
    def show(self):
        # accessing public data member
        print("Name: ", self.name, 'Salary:', self.salary)
# creating object of a class
emp = Employee('Jessa', 10000)
# accessing public data members
print("Name: ", emp.name, 'Salary:', emp.salary)
#Name:  Jessa Salary: 10000
# calling public method of the class
emp.show()
#Name:  Jessa Salary: 10000

"""Private Member
We can protect variables in the class by marking them private. To define a
private variable add two underscores as a prefix at the start of a variable name.
Private members are accessible only within the class, and we can't access them
directly from the class objects."""
class Employee:
    # constructor
    def __init__(self, name, salary):
        # public data member
        self.name = name
        # private member
        self.__salary = salary

# creating object of a class
emp = Employee('Jessa', 10000)

# accessing private data members
print('Salary:', emp.__salary)
#'Employee' object has no attribute '__salary'
"""In the above example, the salary is a private variable. As you know, we can't
access the private variable from the outside of that class. We can access
private members from outside of a class using the following two approaches

Create public method to access private members
Use name mangling
Let's see each one by one"""
Public method to access private members
Example: Access Private member outside of a class using an instance method
```

```
115
116 class Employee:
117     # constructor
118     def __init__(self, name, salary):
119         # public data member
120         self.name = name
121         # private member
122         self.__salary = salary
123
124     # public instance methods
125     def show(self):
126         # private members are accessible from a class
127         print("Name: ", self.name, 'Salary:', self.__salary)
128
129 # creating object of a class
130 emp = Employee('Jessa', 10000)
131
132 # calling public method of the class
133 emp.show()
134 #Name:  Jessa Salary: 10000
135 """Name Mangling to access private member
136 We can directly access private and protected variables from outside of a
137 class through name mangling. The name mangling is created on an identifier by
138 adding two leading underscores and one trailing underscore, like this
139 _classname__dataMember, where classname is the current class, and data member
140 is the private variable name."""
141 Example: Access private member
142
143 class Employee:
144     # constructor
145     def __init__(self, name, salary):
146         # public data member
147         self.name = name
148         # private member
149         self.__salary = salary
150
151 # creating object of a class
152 emp = Employee('Jessa', 10000)
153
154 print('Name:', emp.name)
155 #Name: Jessa
156 # direct access to private member using name mangling
157 print('Salary:', emp._Employee__salary)
158 #Salary: 10000
159 Protected Member
160 Protected members are accessible within the class and also available to its
161 sub-classes. To define a protected member, prefix the member name with a
162 single underscore _. Protected data members are used when you implement
163 inheritance and want to allow data members access to only child classes.
164
165 Example: Proctecd member in inheritance.
```

```
Example: Proctecd member in inheritance.

# base class
class Company:
    def __init__(self):
        # Protected member
        self._project = "NLP"

# child class
class Employee(Company):
    def __init__(self, name):
        self.name = name
        Company.__init__(self)

    def show(self):
        print("Employee name :", self.name)
        # Accessing protected member in child class
        print("Working on project :", self._project)

c = Employee("Jessa")
c.show()
#Employee name : Jessa
#Working on project : NLP

# Direct access protected data member
print('Project:', c._project)
#Project: NLP

Getters and Setters in Python
To implement proper encapsulation in Python, we need to use setters and getters.
 The primary purpose of using getters and setters in object-oriented programs
 is to ensure data encapsulation. Use the getter method to access data members
 and the setter methods to modify the data members.

In Python, private variables are not hidden fields like in other programming
languages. The getters and setters methods are often used when:

When we want to avoid direct access to private variables
To add validation logic for setting a value

class Student:
    def __init__(self, name, age):
        # private member
        self.name = name
        self.__age = age

    # getter method
    def get_age(self):
        return self.__age

    # setter method
    def set_age(self, age):
        self.__age = age

stud = Student('Jessa', 14)
```

```
stud = Student('Jessa', 14)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())
#Name: Jessa 14
# changing age using setter
stud.set_age(16)

# retrieving age using getter
print('Name:', stud.name, stud.get_age())
#Name: Jessa 16
Nesne niteliklerinizin (veri üyesi) değerlerini değiştirmeden önce bilgi
gizleme ve ek doğrulama uygulamak için kapsüllemenin nasıl kullanılacağını
gösteren başka bir örnek alalım.
Example: Information Hiding and conditional logic for setting an object attribute
class Student:
    def __init__(self, name, roll_no, age):
        # private member
        self.name = name
        # private members to restrict access
        # avoid direct data modification
        self.__roll_no = roll_no
        self.__age = age

    def show(self):
        print('Student Details:', self.name, self.__roll_no)

    # getter methods
    def get_roll_no(self):
        return self.__roll_no

    # setter method to modify data member
    # condition to allow data modification with rules
    def set_roll_no(self, number):
        if number > 50:
            print('Invalid roll no. Please set correct roll number')
        else:
            self.__roll_no = number

jessa = Student('Jessa', 10, 15)

# before Modify
jessa.show()
#Student Details: Jessa 10
# changing roll number using setter
jessa.set_roll_no(120)
#Invalid roll no. Please set correct roll number

jessa.set_roll_no(25)
jessa.show()
#Student Details: Jessa 25

Advantages of Encapsulation
Security: The main advantage of using encapsulation is the security of the data.
Encapsulation protects an object from unauthorized access. It allows private
and protected access levels to prevent accidental data modification.
```

Advantages of Encapsulation

Security: The main advantage of using encapsulation is the security of the data.

Encapsulation protects an object from unauthorized access. It allows private

and protected access levels to prevent accidental data modification.

Data Hiding: The user would not be knowing what is going on behind the scene.

They would only be knowing that to modify a data member, call the setter method.

To read a data member, call the getter method. What these setter and getter methods are doing is hidden from them.

Simplicity: It simplifies the maintenance of the application by keeping classes separated and preventing them from tightly coupling with each other.

Aesthetics: Bundling data and methods within a class makes code more readable and maintainable