PRO - UT2-A2. Encapsulamiento

Encapsulamiento

El **encapsulamiento** es la propiedad que permite asegurar que la **información y los métodos de un objeto** están **ocultos** del mundo exterior.

Cuando mediante el proceso de abstracción diseñamos un clase puede darse el caso de que incluya atributos o métodos que por seguridad o para evitar inconsistencias solo queremos que sean accedidos internamente en el objeto, pero no de forma externa.

Tipos de acceso

En la POO se distinguen tres niveles de acceso:

- **Público**
  - Los datos y los métodos de una clase son accesibles desde cualquier parte del programa.
  - Es el nivel más bajo de protección.
  - Asignaré esta protección a la parte de mis objetos que puede ser accedida de forma externa.
- **Protegido**
  - Los datos y métodos con este nivel de protección no son accesibles externamente.
  - Solo pueden ser accedidos desde la propia clase o desde subclases (más adelante veremos las subclases)
- **Privado**
  - Es el nivel más alto de protección
  - Los datos y métodos con este nivel de protección solo son accesibles desde la propia clase.

En muchos lenguajes de programación se usan modificadores (palabras reservadas) que se ponen al principio de la definición del atributo o método para asignarle un nivel de protección. Por ejemplo, en Java se usan los modificadores `public`, `protected` y `private`

```
class UniversityStudent {
    private int id;
    public string name;
    ...
}
```

En Python se siguen otros principios para establecer el nivel de protección de los miembros de una clase

> Cuando hablamos de **miembros de una clase** estamos hablando de los atributos y métodos que incluye

Miembros públicos en Python

En Python los atributos y métodos son públicos por defecto. Cualquier miembro puede ser accedido desde el exterior con la notación punteada como hemos visto hasta ahora.

```
class Student:
    school_name = 'XYZ School' # class attribute

    def __init__(self, name, age):
        self.name=name # instance attribute
        self.age=age # instance attribute
    def show_name(self):
        return self.name
```

Podemos acceder a los atributos y métodos de la clase e incluso modificar su valor:

```
>>> sebastian = Student("Sebastián", 15)
>>> print(sebastian.age)
15
>>> sebastian.name = "Antonio"
>>> print(sebastian.show_name())
'Antonio'
```

Miembros privados

Como hemos visto es el nivel más alto de protección y los métodos y atributos así definidos solo pueden ser accedidos internamente.

En Python no existe un modificador que nos permita definir con dicha protección un miembro, para conseguirlo se sigue el convenido de que el nombre del mismo debe tener como prefijo un doble guión bajo (`__`)

```
class Student:
    __school_name = 'XYZ School' # private class attribute

    def __init__(self, name, age):
        self.__name=name  # private instance attribute
        self.__age=age # private instance attribute
    def __get_age(self):  # private method
        return self.__age
```

Para comprobarlo:

```
>>> std = Student("Bill", 25)
>>> std.__school_name
AttributeError: 'Student' object has no attribute '__school_name'
>>> std.__name
AttributeError: 'Student' object has no attribute '__name'
>>> std.__get_age()
AttributeError: 'Student' object has no attribute '__get_age'
```

Vemos que obtenemos error al intentar acceder a dichos atributos y métodos desde fuera del objeto.

Cuando Python ve que un miembro empieza con `__` lo que hace es enmascararlos, renombrándolos internamente añadiendo al nombre el prefijo `_NombreClase`.

Para la clase anterior:

```
>>> std = Student("Bill", 25)
>>> std.__name
AttributeError: 'Student' object has no attribute '__name'

# Con el prefijo _Student si podemos acceder a los miembros

>>> std._Student__name
'Bill'
>>> std._Student__name = 'Steve'
>>> std._Student__name
'Steve'
>>> std._Student__get_age()
25
```

> Fíjate que el atributo `__name` y el método `__get_age()` se les ha añadido el prefijo `_Student` y son accesibles externamente mediante `_Student__name` y `_Student__get_age()`. El objetivo no es impedir el acceso, sino que el programador conozca el comportamiento y no intente acceder a miembros si su definición especifica que no se debería acceder.

Acceso externo a miembros protegidos

Cuando diseñamos nuestros objetos definimos como privados los miembros que no queremos que sean accedidos directamente desde fuera del objeto.

Si queremos que externamente se pueda acceder a determinada información lo que hacemos es crear metodos públicos que accedan a los miembros privados de la clase.

Si por ejemplo queremos crear un método que permita externamente mostrar la información de un estudiante:

```
class Student:
    __school_name = 'XYZ School' # private class attribute

    def __init__(self, name, age):
        self.__name=name  # private instance attribute
        self.__age=age # private instance attribute
    def __get_age(self):  # private method
        return self.__age

    def show_student(self):  # public method
        age_info = self.__get_age() # can access private method
        return f"{self.__name} is {age_info} years old and member of
{self.__school_name}"  # can access private attributes
```

Para comprobarlo:

```
>>> std = Student("Bill", 25)
Bill is 25 years old and member of XYZ School
```

> Fíjate que para que un método pueda acceder a los atributos y métodos de la clase debe utilizar la notación **punteada** poniendo en primer lugar `self` para indicar que los atributos y métodos a los que queremos acceder son del propio objeto

Lo que conseguimos dando protección a ciertos miembros de una clase es **encapsular** su información y comportamiento y restringir el acceso externo a determinados miembros de la clase que acceden de forma controlada.

Miembros protegidos

Cómo vimos, son los atributos que queremos que sean accedidos solo internamente o desde las clases heredadas (subclases)

> En breve veremos la herencia en POO

Para definir un miembro como protegido seguimos el convenio de poner en su nombre como prefijo el carácter subrayado (`_`)

```
class Student:
    _school_name = 'XYZ School' # protected class attribute

    def __init__(self, name, age):
        self._name=name  # protected instance attribute
        self._age=age    # protected instance attribute
    def _get_age(self):  # protected method
        return self._age
```

En este caso no se realiza ninguna traducción interna por parte de Python, por tanto aunque formalmente el miembro es protegido, realmente si es accesible externamente.

```
>>> std = Student("Swati")
>>> std._name
'Swati'
>>> std._name = 'Dipa'
>>> std._name
'Dipa'
```

Los creadores de Python siguieron este esquema por conveniencia y siguieron la regla de que debe ser el programador el que siga dicho convenio y no intente acceder desde donde no debe a los miembros así definidos si quiere que si aplicación sea coherente en su comportamiento.

> Extraido de comentario en stackoverflow.com: "Python does not support access protection as C++/Java/C# does. **Everything is public**. The motto is, "We're all adults here." Document your classes, and insist that your collaborators read and follow the documentation.
> The culture in Python is that names starting with underscores mean, "don't use these unless you really know you should." You might choose to begin your "protected" methods with underscores. But keep in mind, this is just a convention, it doesn't change how the method can be accessed.
> Names beginning with double underscores (__name) are mangled, so that inheritance hierarchies can be built without fear of name collisions. Some people use these for "private" methods, but again, it doesn't change how the method can be accessed."

Resumiendo

Si queremos que un miembro de una clase sea privado, protegido o público para conseguirlo simplemente debemos seguir estos convenios:

- Miembro **privado**: su nombre tiene como prefijo doble subrayado (`__`)
- Miembro **protegido**: su nombre tiene como prefijo un subrayado (`_`)

- Miembro **público**: su nombre no tiene ningún caracter subrayado al principio.

Recursos

- Python - Public, Protected, Private Members - TutorialsTeacher.com
- Access Modifiers in Python : Public, Private and Protected - geeksforgeeks.org
- Python Access Modifiers(Public, Protected, Private - tutlane.com)