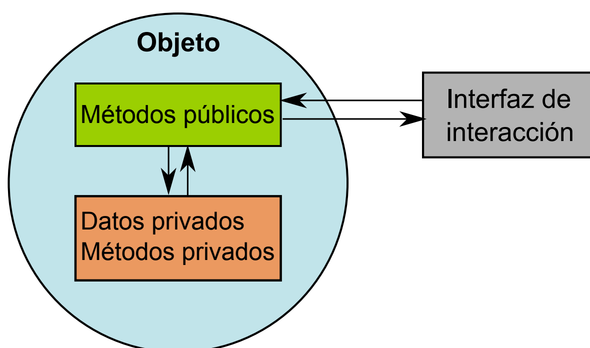
1. Concepto general

El término *encapsulamiento* suele utilizarse para referirse a cosas diferentes:

1. la agrupación de estado y comportamiento en una unidad conceptual (la clase);
2. la aplicación de mecanismos de restricción de acceso a los atributos y métodos;
3. el ocultamiento de información como principio de diseño.

De acuerdo al primer significado, el **encapsulamiento** refleja el hecho de que los objetos son entidades que agrupan los atributos y los métodos que operan sobre ellos. Literalmente, un objeto es una cápsula cuyo interior (su implementación) sólo se debería poder utilizar a través de puntos de contacto con el exterior (su interfaz) bien definidos.



El segundo significado se refiere al control de la **accesibilidad** o visibilidad de los métodos y atributos desde el exterior, en muchos lenguajes esto se logra mediante palabras claves (frecuentemente denominadas *especificadores de acceso*), que definen qué miembros forman parte de la interfaz y cuáles son parte de la implementación.

Finalmente, el **ocultamiento de información** es un principio que sugiere, como condición necesaria para obtener un buen diseño, que todos los detalles de la implementación deben ser invisibles desde el exterior.

Cuando se menciona el término "encapsulamiento" sin más, generalmente se está haciendo referencia a las tres definiciones como un todo.

Matías S. Ávalos© CC BY-NC-SA 4.0

2. Implementación en Python

En **Python**, la declaración de una clase de objetos garantiza la primera acepción de encapsulamiento. Sin embargo, esto no garantiza de ningún modo el ocultamiento de información. El ocultamiento de información se consigue haciendo que los atributos sean privados y que el acceso a los mismos se lleve a cabo mediante métodos públicos.

A diferencia de otros lenguajes en **Python** no existen palabras clave para dar accesos restringidos, como podría ser `public`, `protected` o `private` en **Java** y/o **C++**. De hecho, nada impide el acceso a los miembros de cualquier objeto, es decir, no existe *per se* un método de ocultar información.

Programa 1: Atributos públicos

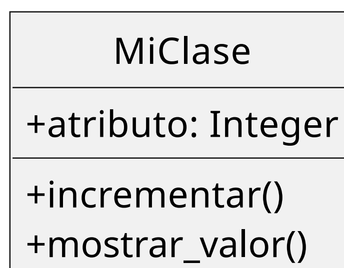
```python
class MiClase:
  def __init__(self, valor : int) -> None:
    self.atributo = valor

  def incrementar(self) -> None:
    self.atributo += 1

  def mostrar_valor(self) -> None:
    print(f"El atributo vale: {self.atributo}")

if __name__ == "__main__":
  obj = MiClase(10)
  obj.atributo += 1 # Acceso al atributo "público"
  obj.incrementar()
  obj.mostrar_valor()
```

Esto mostrará por pantalla el atributo con valor 12, consideraremos que al no poner ningún guion será considerado **publico**, es decir, parte de la interfaz accesible del objeto. En los diagramas de UML esto se señaliza utilizando el signo más '+' delante del atributo o método en cuestión.



Matías S. Ávalos© 4.0

Como se mencionó, no hay ocultamiento real en **Python**. Sin embargo, existe una convención que indica que cualquier elemento que empiece con un guion bajo debe considerarse un miembro interno que no debe utilizar en su interfaz pública:

Programa 2: Atributos protegidos

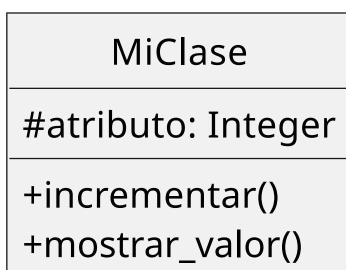
```python
class MiClase:
  def __init__(self, valor : int) -> None:
    self._atributo = valor

  def incrementar(self) -> None:
    self._atributo += 1

  def mostrar_valor(self) -> None:
    print(f"El atributo vale: {self._atributo}")

if __name__ == "__main__":
  obj = MiClase(10)
  obj._atributo += 1 # Acceso al atributo "protegido"
  obj.incrementar()
  obj.mostrar_valor()
```

Aquí, nuevamente, es posible acceder al miembro violando la convención. Estos miembros están pensados para ser accedidos por clases derivadas únicamente. Pero siempre es posible accederlos desde cualquier otro módulo o clase. A estos atributos se los considerara **protegidos**, y serán accedidos únicamente a través de clases derivadas. En los diagramas UML se los indica con el símbolo numeral '`#`'.



En versiones más actuales de **Python** se incluye una metodología llamada **Manipulación de nombres** (*Name Mangling*). Estos son los atributos que empiezan con doble guion bajo. A los fines prácticos estos atributos no son fácilmente accesibles, ya que **Python**, por fuera de la clase donde fueron declarados, aplica un cambio de nombre al intentar accederlos. La manera de poder hacerlo es `_<NombreClase>__<atributo>`.

Matías S. Ávalos© CC BY-NC-SA 4.0

Programa 3: Atributos privados

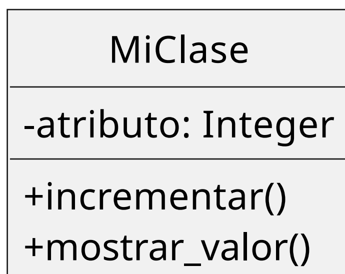
```python
class MiClase:
  def __init__(self, valor : int) -> None:
    self.__atributo = valor

  def incrementar(self) -> None:
    self.__atributo += 1

  def mostrar_valor(self) -> None:
    print(f"El atributo vale: {self.__atributo}")

if __name__ == "__main__":
  obj = MiClase(10)
  # obj.__atributo += 1 # Error: atributo "privado"
  obj._MiClase__atributo += 1 # Acceso forzado con name mangling
  obj.incrementar()
  obj.mostrar_valor()
```

Si no se utiliza el *Name Mangling* para forzar el acceso al atributo, podemos decir que este cumple con el concepto de atributo **privado**, que se suele utilizar en la programación orientada a objetos más clásica. Incluso, no puede accederse, ni siquiera, desde las clases derivadas, que es el comportamiento esperado. En los diagramas UML esto se simboliza con el menos ‘-’.



3. Referencias

Para más información puede consultar los siguientes enlaces:

- Documentación oficial: https://docs.python.org/3/tutorial/classes.html
- Uso del guion bajo en Python: https://realpython.com/python-double-underscore/
- Interprete Visual de Python: http://www.pythontutor.com/visualize.html

Matías S. Ávalos© CC BY-NC-SA 4.0