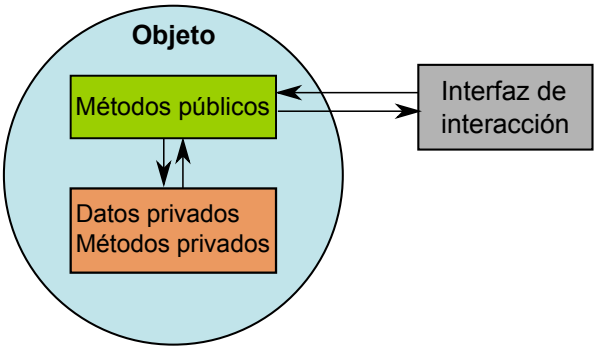
1. Concepto general

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 restricción de acceso a los atributos y métodos;
3. el ocultamiento de información como principio de diseño.

De acuerdo con el primer significado, el **encapsulamiento** refleja el hecho de que los objetos son entidades que agrupan los atributos y métodos que operan sobre ellos. Literalmente, un objeto es una cápsula cuyo interior (su implementación) debería emplearse únicamente 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 clave (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 lograr un buen diseño, que todos los detalles de implementación sean invisibles desde el exterior.

Cuando se menciona el término "encapsulamiento" sin más, generalmente se se hace 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: no existe, *per se*, un modo de ocultar información..

Programa 1: Atributos públicos

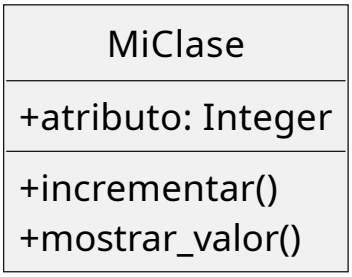
```python
class MiClase:
  def __init__(self, valor: int) -> None:
    self.atributo = valor

  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. . Al no poner ningún guion bajo, se considera **público**, es decir, parte de la interfaz accesible del objeto. En los diagramas UML se indica con el signo más + delante del atributo o método.



Como se mencionó, no hay ocultamiento real Python. Sin embargo, existe una conven-

Matías S. Ávalos© CC BY NC SA 4.0

ción: cualquier elemento que empiece con un guion bajo (_) debe considerarse un miembro interno que no debe usarse en la interfaz pública:

Programa 2: Atributos protegidos

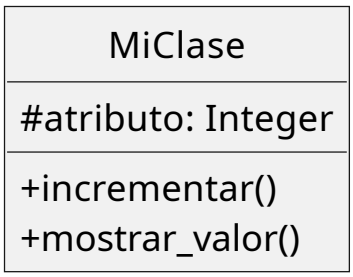
```python
class MiClase:
  def __init__(self, valor: int) -> None:
    self._atributo = valor

  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 el uso en subclases; sin embargo, siempre es posible accederlos desde cualquier otro módulo o clase. A estos atributos se los considerará **protegidos**, y deberían accederse únicamente desde subclases. En los diagramas UML, se indican con el símbolo numeral '#'.



A partir de versiones recientes de `Python` introduce el mecanismo de **Manipulación de nombres** (*Name Mangling*). Consiste en atributos que comienzan con doble guion bajo (`__atributo`), los cuales `Python` renombra internamente. 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. Para accederlos, debe usarse la forma `_<NombreClase>__<atributo>`.

Matías S. Ávalos© CC BY NC SA 4.0

Programa 3: Atributos privados

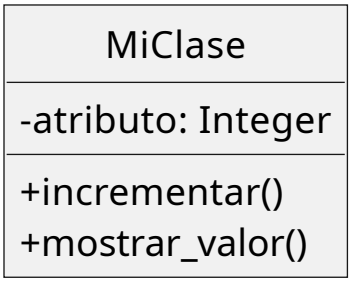
```python
class MiClase:
  def __init__(self, valor: int) -> None:
    self.__atributo = valor

  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, podemos decir que este cumple con el concepto de atributo **privado**, que se suele utilizar en la programación OO más clásica. Incluso, no puede accederse, ni siquiera, desde las subclases, 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/`.
- Intérprete Visual de Python: `http://www.pythontutor.com/visualize.html`.

Matías S. Ávalos© CC BY NC SA 4.0