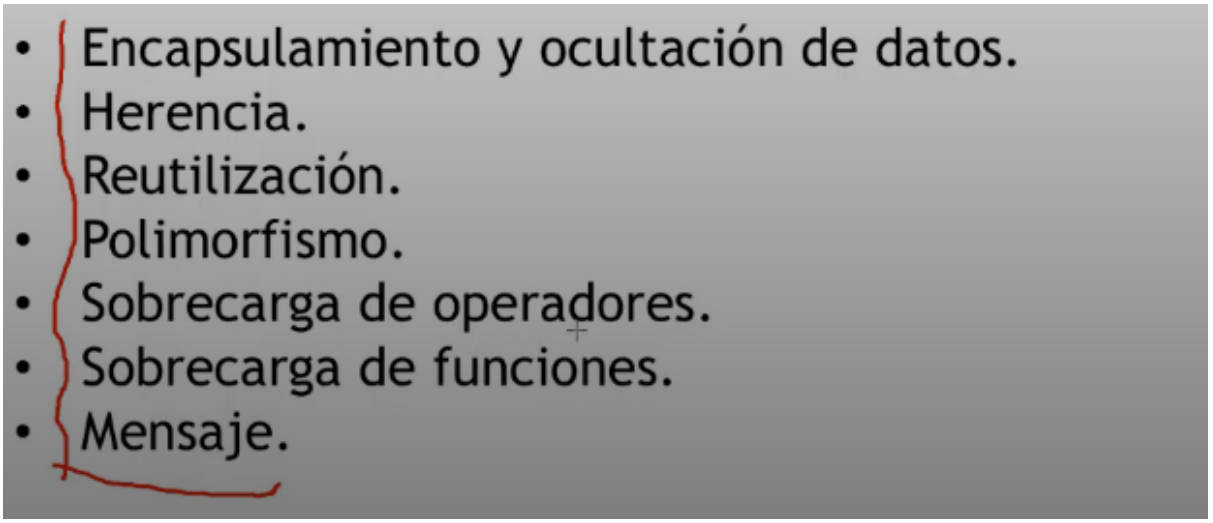

Encapsulamiento y ocultación de datos

Permite proteger los datos internos del objeto.

El encapsulamiento se logra agrupando datos (atributos) y métodos (funciones) que operan sobre esos datos en una sola unidad (clase). La idea es que los datos de un objeto deben ser accedidos y modificados principalmente a través de sus métodos, y no directamente.

Ocultación de Datos (Data Hiding) en Python

Python logra la "ocultación de datos" por convención y mediante el uso de "name mangling" para atributos que se consideran privados.

python

```
class Persona:
    def __init__(self, nombre):
        self.__nombre = nombre  # atributo privado

    def get_nombre(self):
        return self.__nombre

    def set_nombre(self, nuevo_nombre):
        self.__nombre = nuevo_nombre

p = Persona("Ana")
print(p.get_nombre())  # Ana
```

---

2. Herencia

Herencia

Por supuesto, una característica del lenguaje no sería digna del nombre «clase» si no soportara herencia. La sintaxis para una definición de clase derivada se ve así:

python
CopiarEditar

```
class Animal:
    def hablar(self):
        print("Hace un sonido")

class Perro(Animal):
    def hablar(self):
        print("Ladra")

mi_perro = Perro()
mi_perro.hablar()  # Ladra
```

---

3. Reutilización

Cuando escribes mucho código Python en un proyecto, o en varios proyectos, a menudo querrás crear partes de código reutilizables.

DSS proporciona varios mecanismos para reutilizar el código Python:

Empaquetar su código como funciones o módulos y ponerlos a disposición en un proyecto específico

Importar código que se ha puesto a disposición de un proyecto a otro

Empaquetar su código como funciones o módulos y hacerlos disponibles en todos los proyectos

Empaquetar su código como un complemento reutilizable y ponerlo a disposición tanto de usuarios programadores como de usuarios no programadores.

python

```
class Vehiculo:
    def __init__(self, marca):
        self.marca = marca
```

```
    def mostrar(self):
        print(f"Vehículo marca {self.marca}")

class Auto(Vehiculo):
    pass

a = Auto("Toyota")
a.mostrar()  # Vehículo marca Toyota
```

---

4. Polimorfismo

El polimorfismo es uno de los pilares básicos en la programación orientada a objetos, por lo que para entenderlo es importante tener las bases de la POO y la herencia bien asentadas.

El término polimorfismo tiene origen en las palabras poly (muchos) y morfo (formas), y aplicado a la programación hace referencia a que los objetos pueden tomar diferentes formas. ¿Pero qué significa esto?

Pues bien, significa que objetos de diferentes clases pueden ser accedidos utilizando el mismo interfaz, mostrando un comportamiento distinto (tomando diferentes formas) según cómo sean accedidos.

python

```
class Gato:
    def hablar(self):
        print("Miau")

class Perro:
    def hablar(self):
        print("Guau")

def hacer_hablar(animal):
    animal.hablar()

hacer_hablar(Gato())  # Miau
hacer_hablar(Perro())  # Guau
```

---

5. Sobrecarga de operadores

En Python, puedes definir cómo se comportan los operadores con tus clases.

python
CopiarEditar

```
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, otro):
        return Punto(self.x + otro.x, self.y + otro.y)

p1 = Punto(1, 2)
p2 = Punto(3, 4)
resultado = p1 + p2
print(resultado.x, resultado.y)  # 4 6
```

---

6. Sobrecarga de funciones

Python no permite sobrecarga tradicional, pero se puede simular:

python
CopiarEditar

```
def saludar(nombre=None):
    if nombre:
        print(f"Hola, {nombre}")
    else:
        print("Hola")

saludar()           # Hola
saludar("Pedro")    # Hola, Pedro
```

---

7. Mensaje

Es la comunicación entre objetos (llamar métodos).

python
CopiarEditar

```
class Emisor:
    def enviar_mensaje(self, receptor):
        receptor.recibir_mensaje()
```

```
class Receptor:
    def recibir_mensaje(self):
        print("Mensaje recibido")

e = Emisor()
r = Receptor()
e.enviar_mensaje(r)  # Mensaje recibido
```