Actividad 1. Concepto de clase.

¿Qué es una clase?

Imagina que tienes un **molde para hacer un zorrito**. Con ese molde, puedes hacer muchos zorritos que tienen la misma forma, pero cada zorrito puede ser de un color o tamaño diferente. En Python, una **clase** es como ese molde: te permite crear muchas **cosas** (objetos) que son parecidas, pero cada una puede tener sus propias características.

Una **clase** es una plantilla o un plano para crear objetos. Es un concepto fundamental de la Programación Orientada a Objetos (POO) que permite organizar el código de manera modular y reutilizable. Una clase define un tipo de dato personalizado que agrupa **atributos** (datos) y **métodos** (funciones) que operan sobre esos datos.

Partes de una clase

1. **Atributos:**
   - Características de las cosas que creas.
   - Por ejemplo, si tienes un molde para hacer perros, los atributos podrían ser el nombre, el color y la edad del perro.
2. **Métodos:**
   - Acciones que pueden hacer las cosas que creas.
   - Por ejemplo, un perro puede ladrar o correr.
3. **Constructor:**
   - Es como la receta que usas para hacer las cosas.
   - Te dice cómo crear un objeto y qué características debe tener.

Ejemplo: Clase "Perro"

Vamos a crear una clase llamada Perro. Con esta clase, podremos crear muchos perritos diferentes, cada uno con su propio nombre, color y edad.

```python
# Definimos la clase Perro
class Perro:
    # Constructor (receta para crear un perro)
    def __init__(self, nombre, color, edad):
        self.nombre = nombre  # Atributo: nombre del perro
        self.color = color    # Atributo: color del perro
        self.edad = edad      # Atributo: edad del perro

    # Método: acción de ladrar
    def ladrar(self):
        return "¡Guau guau!"

    # Método: acción de correr
    def correr(self):
        return f"{self.nombre} está corriendo."
# Creamos un perro llamado "Roco"
roco = Perro("Roco", "Marrón", 3)

# Creamos otro perro llamado "Luna"
luna = Perro("Luna", "Blanco", 2)

# Usamos los métodos y atributos de los perros
print(roco.nombre)  # Salida: Roco
print(roco.color)   # Salida: Marrón
print(roco.ladrar())  # Salida: ¡Guau guau!
print(luna.correr())  # Salida: Luna está corriendo.
```

En resumen, ¿qué es una clase?

Actividad 2. Objeto

Un **objeto** es como una **cosa** que creas usando un **molde** (la clase). En nuestro ejemplo, la clase Perro es el molde, y los objetos son los **perritos** que creas con ese molde. Cada perrito es un objeto único, con sus propias características y acciones.

Ejemplo: Objetos de la clase Perro
En el ejemplo anterior, creamos dos objetos de la clase Perro: roco y luna. Vamos a verlos en detalle.

```
# Creamos un perro llamado "Roco"
roco = Perro("Roco", "Marrón", 3)

# Creamos otro perro llamado "Luna"
luna = Perro("Luna", "Blanco", 2)
```

¿Qué son roco y luna?

- roco:
  - Es un objeto de la clase Perro.
  - Tiene sus propios atributos:
    - Nombre: "Roco"
    - Color: "Marrón"
    - Edad: 3
  - Puede hacer las acciones (métodos) definidas en la clase, como ladrar() y correr().
- luna:
  - Es otro objeto de la clase Perro.
  - Tiene sus propios atributos:
    - Nombre: "Luna"
    - Color: "Blanco"
    - Edad: 2
  - También puede hacer las acciones (métodos) definidas en la clase, como ladrar() y correr().

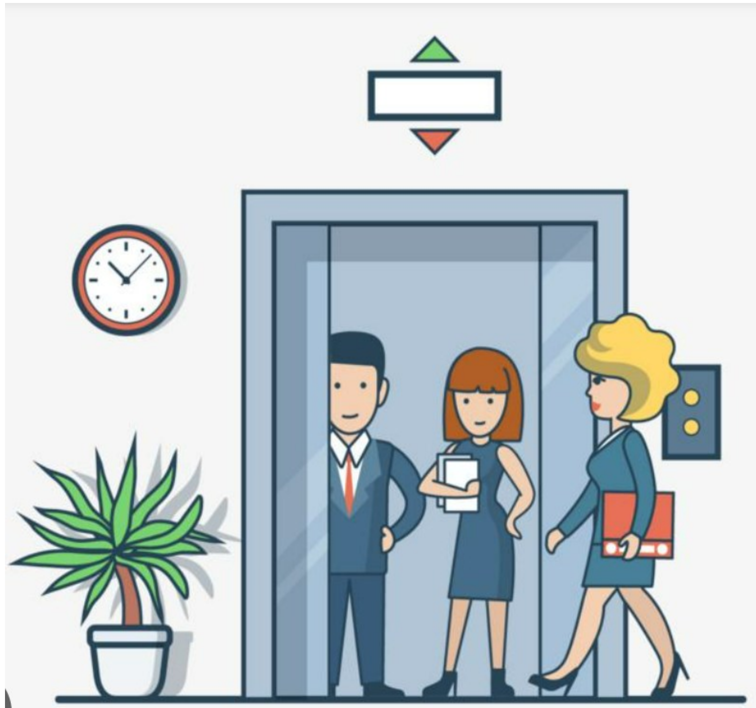
Acciones de los objetos
Cada objeto puede hacer las acciones definidas en la clase. Por ejemplo:

```
# Roco ladra
print(roco.ladrar())  # Salida: ¡Guau guau!

# Luna corre
print(luna.correr())  # Salida: Luna está corriendo.
```

Crea un perrito llamado Max, de color negro y con 5 años y haz que ladre.

Actividad 3. Elevador código y representación.



Actividad 4. ¿Qué es __init__?

El método __init__ es como la **receta** o las **instrucciones** que se usan para crear un objeto. El método __init__ en Python es un **método especial** (también conocido como **constructor**) que se utiliza para inicializar un objeto cuando se crea una instancia de una clase. Es uno de los métodos más importantes en la Programación Orientada a Objetos (POO) en Python, ya que permite definir el estado inicial de un objeto.

En nuestro ejemplo de la clase Perro, el método __init__ es el que nos dice cómo debe ser un perrito cuando lo creamos.

```
class Perro:
    # Método __init__ (receta para crear un perro)
    def __init__(self, nombre, color, edad):
        self.nombre = nombre  # Atributo: nombre del perro
        self.color = color    # Atributo: color del perro
        self.edad = edad      # Atributo: edad del perro

    # Método: acción de ladrar
    def ladrar(self):
        return "¡Guau guau!"

    # Método: acción de correr
    def correr(self):
        return f"{self.nombre} está corriendo."
```

¿Para qué sirve __init__?

1. **Inicializa los atributos**:
    - Le da valores iniciales a las **características** del objeto.
    - Por ejemplo, cuando creas un perrito, el método __init__ se asegura de que el perrito tenga un **nombre**, un **color** y una **edad**.
2. **Se ejecuta automáticamente**:
    - Cuando creas un objeto (por ejemplo, roco = Perro("Roco", "Marrón", 3)), el método __init__ se llama automáticamente.
    - No tienes que llamarlo tú directamente.
3. **Recibe parámetros**:
    - Puedes pasarle información cuando creas el objeto.
    - Por ejemplo, cuando creas un perrito, le pasas su nombre, color y edad.
4. **No retorna ningún valor**:
    - El método __init__ no debe retornar ningún valor explícitamente (ni siquiera None). Su propósito es inicializar el objeto, no devolver algo.
5. **Nombre reservado**:
    - El nombre __init__ es un nombre reservado en Python. No puedes usar otro nombre para el constructor.

¿Por qué es importante __init__?

- Sin __init__, no podríamos darle valores iniciales a los atributos del objeto.
- Es como si tuvieras un molde para hacer galletas, pero no tuvieras la receta para preparar la masa. ¡No podrías hacer galletas!

```
perro_misterioso = Perro()
```

¿Qué pasaría si creas un perrito sin pasarle los valores al método __init__?______________

1. Público (Public)

- Por defecto, todos los atributos y métodos en Python son públicos.
- Esto significa que pueden ser accedidos y modificados desde cualquier parte del código.
- **Convención**: No se usa ningún prefijo especial.

2. Protegido (Protected)

- Los atributos y métodos protegidos están destinados a ser accedidos solo dentro de la clase y sus subclases.
- **Convención**: Se usa un guion bajo _ al inicio del nombre para indicar que es protegido.
- **Nota**: Esto es solo una convención, no impide el acceso desde fuera de la clase.

3. Privado (Private)

- Los atributos y métodos privados están destinados a ser accedidos solo dentro de la clase que los define.
- **Convención**: Se usa un doble guion bajo __ al inicio del nombre para indicar que es privado.
- **Mecanismo**: Python realiza un proceso llamado **name mangling** (ofuscación de nombres) para evitar el acceso directo desde fuera de la clase. El nombre del atributo o método se modifica internamente para incluir el nombre de la clase.
- **Nota**: Aunque es más difícil acceder a estos elementos, no es imposible.

```python
class Ejemplo:
    def __init__(self):
        self.publico = "Soy público"  # Atributo público
        self._protegido = "Soy protegido"  # Atributo protegido
        self.__privado = "Soy privado"  # Atributo privado

    def metodo_publico(self):
        return "Método público"

    def _metodo_protegido(self):
        return "Método protegido"

    def __metodo_privado(self):
        return "Método privado"

# Uso de la clase
objeto = Ejemplo()

# Acceso a atributos públicos
print(objeto.publico)  # Correcto
print(objeto.metodo_publico())  # Correcto

# Acceso a atributos protegidos (no recomendado)
print(objeto._protegido)  # Funciona, pero no es recomendado
print(objeto._metodo_protegido())  # Funciona, pero no es recomendado

# Acceso a atributos privados (no permitido directamente)
# print(objeto.__privado)  # Error
# print(objeto.__metodo_privado())  # Error

# Acceso "forzado" a atributos privados (no recomendado)
print(objeto._Ejemplo__privado)  # Funciona, pero no es recomendado
print(objeto._Ejemplo__metodo_privado())  # Funciona, pero no es recomendado
```

Completa la siguiente tabla:

Tipo de acceso	Definición	Convención	Ejemplo	Acceso desde fuera de la clase
Público				
Protegido				
Privado				

¿Qué pasaría si intentas acceder directamente a un atributo privado?

```
print(perro.__nombre)
```

La **herencia** es como cuando un hijo hereda características de sus padres. En programación, una clase puede heredar atributos y métodos de otra clase. La clase que hereda se llama **subclase** (o clase hija), y la clase de la que se hereda se llama **superclase** (o clase padre).

Ejemplo: Herencia con la clase Perro

Imagina que queremos crear una clase para un tipo especial de perro, como un PerroGuardian. Este perro guardián es un perro normal, pero tiene una habilidad adicional: puede **vigilar**.

```
# Clase padre (superclase)
class Perro:
    def __init__(self, nombre, color, edad):
        self.nombre = nombre
        self.color = color
        self.edad = edad

    def ladrar(self):
        return "¡Guau guau!"

    def correr(self):
        return f"{self.nombre} está corriendo."

# Clase hija (subclase)
class PerroGuardian(Perro):  # Hereda de la clase Perro
    def vigilar(self):
        return f"{self.nombre} está vigilando la casa."
```

```
# Crear un objeto de la clase PerroGuardian
guardian = PerroGuardian("Rex", "Negro", 4)

# Usar métodos heredados de la clase Perro
print(guardian.ladrar())  # Salida: ¡Guau guau!
print(guardian.correr())  # Salida: Rex está corriendo.

# Usar el método propio de PerroGuardian
print(guardian.vigilar())  # Salida: Rex está vigilando la casa.
```

Crea una clase PerroPolicia que tenga un método olfatear_drogas()