CONQUER BLOCKS

PYTHON

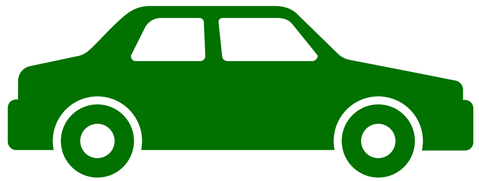
PROGRAMACION ORIENTEADA A OBJETOS (POO)

OBJETOS Y CLASES

CLASES → **Representaciones del mundo real**

Ejemplo:

COCHE

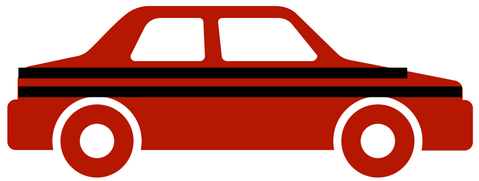


OBJETOS Y CLASES

OBJETOS → **Una instancia de esa clase**

Ejemplo:

MI COCHE



FERRARI

OBJETOS Y CLASES

OBJETO

Atributos

Datos

- Motor
- Puertas

Métodos

Funcionalidades

- Ajustar la velocidad
- Girar
- Tocar el claxon

PROGRAMACION ORIENTADA A OBJETOS (POO)

```
class Fruta:
    def __init__(self, nombre, color):
        self.nombre = nombre
        self.color = color
        self.caducidad = "01.03.2023"
    def details(self):
        print("mi " + self.nombre +
              " es " + self.color )
        print("caduca el ", self.caducidad)

manzana = Fruta("manzana", "roja")
manzana.details()
```

✓ 0.0s

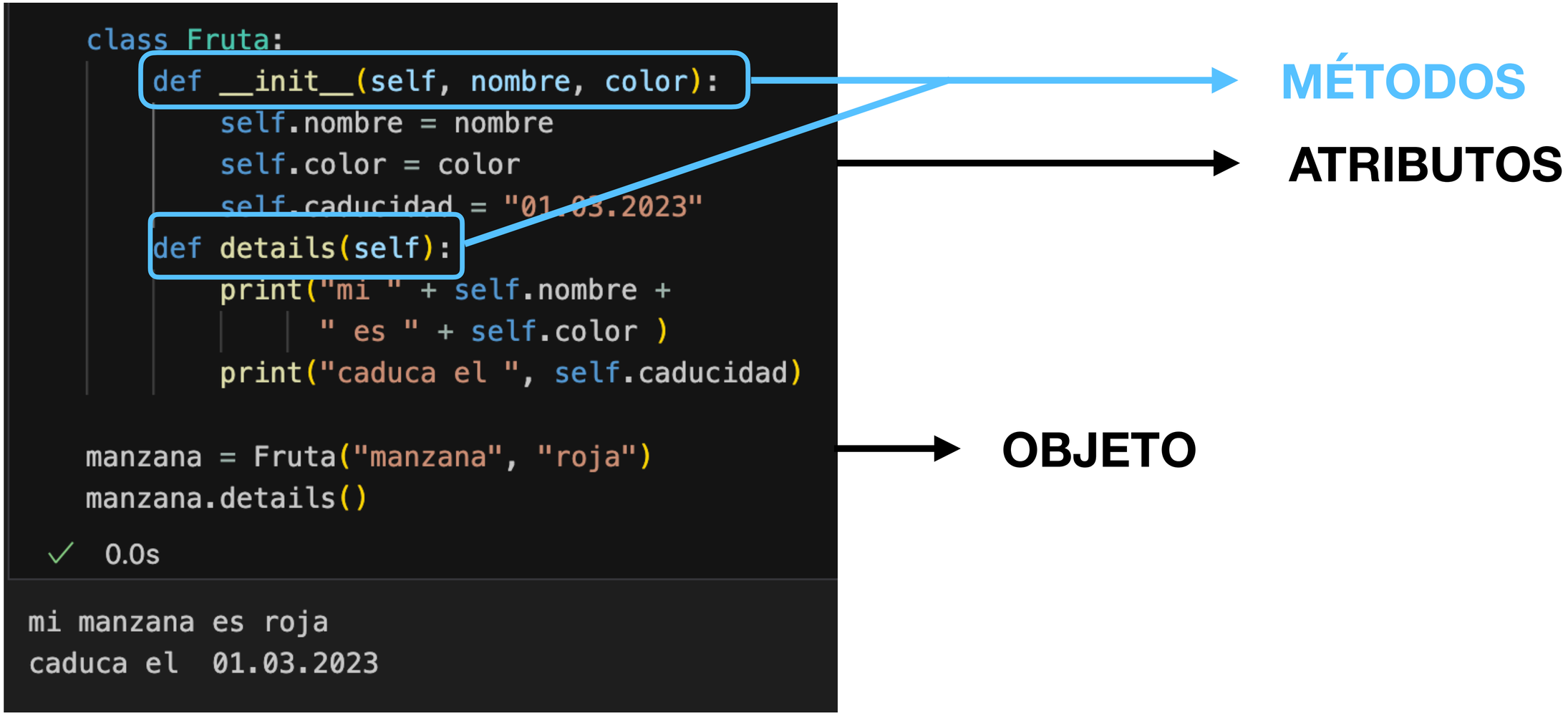
```
mi manzana es roja
caduca el  01.03.2023
```

MÉTODOS

ATRIBUTOS

OBJETO

PROGRAMACION ORIENTADA A OBJETOS (POO)



PROGRAMACION ORIENTADA A OBJETOS (POO)

```
class Fruta:
    def __init__(self, nombre, color):
        self.nombre = nombre
        self.color = color
        self.caducidad = "01.03.2023"
    def details(self):
        print("mi " + self.nombre +
              " es " + self.color )
        print("caduca el ", self.caducidad)

manzana = Fruta("manzana", "roja")
manzana.details()
```

✓ 0.0s

```
mi manzana es roja
caduca el  01.03.2023
```

MÉTODOS

ATRIBUTOS

OBJETO

PROGRAMACION ORIENTADA A OBJETOS (POO)

```
class Fruta:
    def __init__(self, nombre, color):
        self.nombre = nombre
        self.color = color
        self.caducidad = "01.03.2023"
    def details(self):
        print("mi " + self.nombre +
              " es " + self.color )
        print("caduca el ", self.caducidad)

manzana = Fruta("manzana", "roja")
manzana.details()
```

✓ 0.0s

```
mi manzana es roja
caduca el  01.03.2023
```

MÉTODOS

ATRIBUTOS

OBJETO

CUATRO PILARES DEL POO

1. ABSTRACCIÓN

Simplificacion y representación de entidades del mundo real en forma de objetos en el código. Estos encapsulan datos y comportamientos, permitiendo al programador enfocarse en lo que el objeto hace sin preocuparse del cómo.

2. ENCAPSULAMIENTO:

Agrupamiento de datos (atributos) y métodos (funciones) relacionados en un mismo objeto. La idea es que el objeto actúe como una cápsula que oculta los detalles internos y revela solo una interfaz para interactuar con él. De esta manera, se mejora la seguridad y se evita que el código externo modifique directamente el estado interno del objeto.

3. HERENCIA:

La herencia permite que una clase (subclase) herede los atributos y métodos de otra clase (superclase). La subclase puede extender o modificar la funcionalidad de la superclase y agregar nuevos atributos o métodos. Esto facilita la reutilizaron de código y la jerarquización.

4. POLIMORFISMO:

El polimorfismo es la capacidad de diferentes objetos de una jerarquía de clases para responder a una misma llamada de método de manera diferente. El polimorfismo permite que objetos de diferentes clases puedan ser tratados de manera uniforme, pero actúen de forma diferente según su propia implementación. Esto aumenta la flexibilidad del código y facilita la creación de interfaces genéricas que trabajen con múltiples clases.

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio

    def crear_descripcion(self):
        """Devolver una descripcion bien formateada"""
        nombre_extenso = str(self.anio) + " " + self.marca + " " + self.modelo
        return nombre_extenso

mi_coche = Coche("audi", "a4", "2016")
print(mi_coche.crear_descripcion())
```

✓ 0.0s

```
2016 audi a4
```

Añadimos un atributo con un valor por defecto:

```
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        """Devolver una descripcion bien formateada"""
        nombre_extenso = str(self.anio) + " " + self.marca + " " + self.modelo
        return nombre_extenso

mi_coche = Coche("audi", "a4", "2016")
print(mi_coche.crear_descripcion())
print(mi_coche.cuenta_kilometros)
```

✓ 0.0s

```
2016 audi a4
0
```

Añadimos un método para leer el cuentakilómetros

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        pass
        #-- snip --

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilmetros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")

mi_coche = Coche("audi", "a4", "2016")
mi_coche.leer_cuentakilometros()
```

✓ 0.0s Pyth

```
Este coche ha recorrido un total de 0 km
```

Podemos modificar el atributo de kilometraje directamente…

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        pass
        #-- snip --

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilmetros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")

mi_coche = Coche("audi", "a4", "2016")
mi_coche.cuenta_kilometros = 100
mi_coche.leer_cuentakilometros()
```

✓ 0.0s Pytho

```
Este coche ha recorrido un total de 100 km
```

Podemos modificar el atributo de kilometraje mediante un método...

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        pass
        #-- snip --

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilmetros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")

    def actualiza_cuentakilometros(self, kilometros):
        """Actualiza el valor del cuentakilometros al valor dado"""
        self.cuenta_kilometros = kilometros

mi_coche = Coche("audi", "a4", "2016")
mi_coche.actualiza_cuentakilometros(100)
mi_coche.leer_cuentakilometros()
```

✓ 0.0s Pytho

```
Este coche ha recorrido un total de 100 km
```

Esto es mucho más conveniente ya que así podemos asegurarnos de que no se cambie el kilometraje a uno menor del actual

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        pass
        #-- snip --

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilmetros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")

    def actualiza_cuentakilometros(self, kilometros):
        """Actualiza el valor del cuentakilometros al valor dado"""
        if kilometros >= self.cuenta_kilometros:
            self.cuenta_kilometros = kilometros
        else:
            print("No puedes disminuir el kilometraje")


mi_coche = Coche("audi", "a4", "2016")
mi_coche.actualiza_cuentakilometros(100)
mi_coche.leer_cuentakilometros()
mi_coche.actualiza_cuentakilometros(50)
mi_coche.leer_cuentakilometros()
```

✓ 0.0s

```
Este coche ha recorrido un total de 100 km
No puedes disminuir el kilometraje
Este coche ha recorrido un total de 100 km
```

Incrementar el valor de un atributo mediante un método...

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        pass
        #-- snip --

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilmetros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")

    def actualiza_cuentakilometros(self, kilometros):
        """Actualiza el valor del cuentakilometros al valor dado"""
        if kilometros >= self.cuenta_kilometros:
            self.cuenta_kilometros = kilometros
        else:
            print("No puedes disminuir el kilometraje")

    def aumentar_cuentakilometros(self, kilometros):
        """Suma el valor dado al cuentakilometros """
        self.cuenta_kilometros += kilometros

mi_coche_usado = Coche("audi", "a4", "2016")
mi_coche_usado.actualiza_cuentakilometros(23500)
mi_coche_usado.leer_cuentakilometros()
mi_coche_usado.aumentar_cuentakilometros(100)
mi_coche_usado.leer_cuentakilometros()
```

✓ 0.0s

```
Este coche ha recorrido un total de 23500 km
Este coche ha recorrido un total de 23600 km
```

Y si ahora tengo un coche eléctrico?

Puedo crear otra clase CocheElectrico y añadirle los mismo atributos que a la clase Coche

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        pass
        #-- snip --

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilmetros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")

    def actualiza_cuentakilometros(self, kilometros):
        """Actualiza el valor del cuentakilometros al valor dado"""
        if kilometros >= self.cuenta_kilometros:
            self.cuenta_kilometros = kilometros
        else:
            print("No puedes disminuir el kilometraje")
```

Y si ahora tengo un coche eléctrico?

Puedo crear otra clase CocheElectrico y añadirle los mismo atributos que a la clase Coche

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        pass
        #-- snip --

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilmetros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_k

    def actualiza_cuentakilometros(self, kilometros):
        """Actualiza el valor del cuentakilometros al valor dado"""
        if kilometros >= self.cuenta_kilometros:
            self.cuenta_kilometros = kilometros
        else:
            print("No puedes disminuir el kilometraje")
```

```python
class CocheElectrico:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    ...
```

Y si ahora tengo un coche eléctrico?

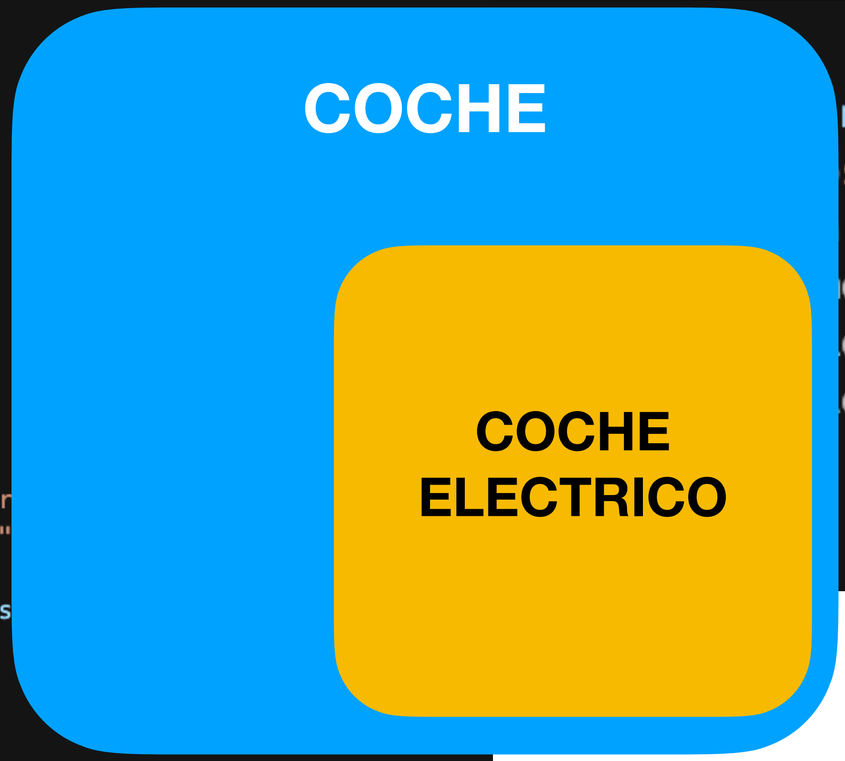
Puedo crear otra clase CocheElectrico y añadirle los mismo atributos que a la clase Coche ———>>> esto es muy INEFICIENTE

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        pass
        #-- snip --

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilmetros recorr
        print("Este coche ha recorrido un total de"

    def actualiza_cuentakilometros(self, kilometros
        """Actualiza el valor del cuentakilometros
        if kilometros >= self.cuenta_kilometros:
            self.cuenta_kilometros = kilometros
        else:
            print("No puedes disminuir el kilometraje")
```




```
marca, modelo, anio):
s atributos del coche"""
rca
odelo
o
ometros = 0
```

Y si ahora tengo un coche eléctrico? Usaremos la *herencia* para crear una clase *hija*

PARENT CLASS (SUPERCLASS) / CLASE PADRE

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.cuenta_kilometros = 0

    def crear_descripcion(self):
        """Devolver una descripcion bien formateada"""
        nombre_extenso = str(self.anio) + " " + self.marca + " " + self.modelo
        return nombre_extenso

    def leer_cuentakilometros(self):
        """Imprime el valor de los kilmetros recorridos"""
        print("Este coche ha recorrido un total de", self.cuenta_kilometros, "km")

    def actualiza_cuentakilometros(self, kilometros):
        """Actualiza el valor del cuentakilometros al valor dado"""
        if kilometros >= self.cuenta_kilometros:
            self.cuenta_kilometros = kilometros
        else:
            print("No puedes disminuir el kilometraje")

    def aumentar_cuentakilometros(self, kilometros):
        """Suma el valor dado al cuentakilometros """
        self.cuenta_kilometros += kilometros
```

CHILD CLASS (SUBCLASS) / CLASE HIJO

```python
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        super().__init__(marca, modelo, anio)

mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
print(mi_tesla.crear_descripcion())
```

✓ 0.0s

```
2016 tesla modelo s
```

Y si ahora tengo un coche eléctrico? Usaremos la *herencia* para crear una clase *hija*

CHILD CLASS (SUBCLASS) / CLASE HIJO

```
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        super().__init__(marca, modelo, anio)

mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
print(mi_tesla.crear_descripcion())
```

✓ 0.0s

```
2016 tesla modelo s
```

Y si ahora tengo un coche eléctrico? Usaremos la *herencia* para crear una clase *hija*

CHILD CLASS (SUBCLASS) / CLASE HIJO

```
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        super().__init__(marca, modelo, anio)
mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
print(mi_tesla.crear_descripcion())
```

Nombre de la superclase

Función que indica a python que conecte con la superclase

✓ 0.0s

```
2016 tesla modelo s
```

Y si ahora tengo un coche eléctrico? Usaremos la *herencia* para crear una clase *hija*

CHILD CLASS (SUBCLASS) / CLASE HIJO

```python
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        super().__init__(marca, modelo, anio)
```

Indica a python que llame a la función __init__ de la superclase.
Esto concede a la subclase todos los atributos de la superclase

HERENCIA

Y si ahora tengo un coche eléctrico? Usaremos la *herencia* para crear una clase *hija*

CHILD CLASS (SUBCLASS) / CLASE HIJO

```
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        super().__init__(marca, modelo, anio)

mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
print(mi_tesla.crear_descripcion())
```

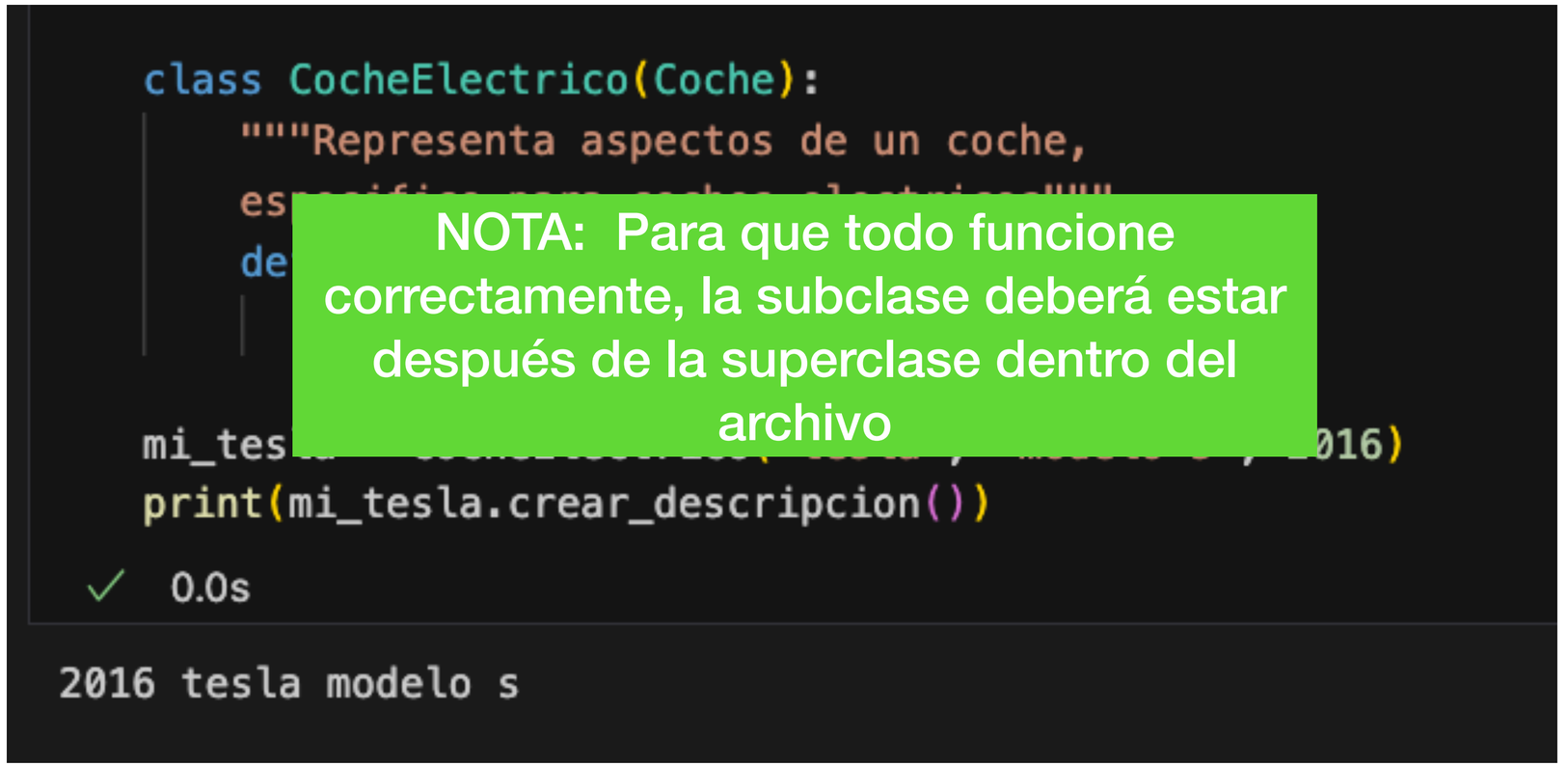
✓ 0.0s

También ha heredado los métodos de la superclase

```
2016 tesla modelo s
```

Y si ahora tengo un coche eléctrico? **Usaremos la *herencia* para crear una clase *hija***

CHILD CLASS (SUBCLASS) / CLASE HIJO



NOTA: Para que todo funcione correctamente, la subclase deberá estar después de la superclase dentro del archivo

Una vez nuestro coche eléctrico ha heredado los atributos y métodos de la superclase coche, podemos añadir métodos que solo apliquen a los coches eléctricos

```
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        """ Inicializar atributos de superclase.
        Despues inicializar atributos especificos
        del coche electrico """
        super().__init__(marca, modelo, anio)
        self.tamanio_bateria = 70
```

```
    def descripcion_bateria(self):
        """Imprime una descripcion de la bateria"""
        print("Este coche tiene una bateria de tamaño",
                self.tamanio_bateria, "-kWh")

mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
print(mi_tesla.crear_descripcion())
mi_tesla.descripcion_bateria()
```

✓ 0.0s

```
2016 tesla modelo s
Este coche tiene una bateria de tamaño 70 -kWh
```

También podemos sobreescribir métodos de la superclase

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.deposito = 0

    def llenar_deposito(self):
        """Simula el llenado del deposito del coche"""
        self.deposito = 100
        print("El deposito esta a", self.deposito)
```

```python
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        """ Inicializar atributos de superclase.
        Despues inicializar atributos especificos
        del coche electrico """
        super().__init__(marca, modelo, anio)
        self.tamanio_bateria = 70

    def llenar_deposito(self):
        """Los coches electricos no tienes deposito"""
        print("Este coche no tiene un deposito")
```

```python
mi_audi = Coche("audi", "a4", 2015)
mi_audi.llenar_deposito()
mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
mi_tesla.llenar_deposito()
```

✓ 0.0s

```
El deposito esta a 100
Este coche no tiene un deposito
```

También podemos sobreescribir métodos de la superclase

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.deposito = 0

    def llenar_deposito(self):
        """Simula el llenado del deposit
        self.deposito = 100
        print("El deposito esta a", self.deposito)
```

POLIMORFISMO

```python
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        """ Inicializar atributos de superclase.
        Despues inicializar atributos especificos
        del coche electrico """
        super().__init__(marca, modelo, anio)
        self.tamanio_bateria = 70

    def llenar_deposito(self):
        """Los coches electricos no tienes deposito"""
        print("Este coche no tiene un deposito")
```

```python
mi_audi = Coche("audi", "a4", 2015)
mi_audi.llenar_deposito()
mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
mi_tesla.llenar_deposito()
```

✓ 0.0s

```
El deposito esta a 100
Este coche no tiene un deposito
```

Cuando estemos modelizando un elemento del mundo real puede que nos encontremos añadiendo cada vez más detalles dentro de una clase. Quizás nos encontremos con una lista creciente de atributos y métodos que hacen que nuestros archivos sean muy extensos.

Ejemplo: Si continuamos añadiendo detalles a la clase CocheElectrico puede que notemos que estamos añadiendo muchos atributos y métodos relacionados con la bateria del coche.

En estos momentos toca parar y mover estos atributos y métodos a una clase nueva llamada batería…

CONQUER BLOCKS

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.deposito = 0

    def llenar_deposito(self):
        """Simula el llenado del deposito del coche"""
        self.deposito = 100
        print("El deposito esta a", self.deposito)

class Bateria:
    """Un intento simple de modelizar
    una bateria"""

    def __init__(self, tamanio_bateria = 70):
        """Inicializamos los atributos de la bateria"""
        self.tamanio_bateria = tamanio_bateria

    def describir_bateria(self):
        """Imprimir el tamaño de la bateria"""
        print("Este coche tiene una bateria de tamaño",
        self.tamanio_bateria,"-kWh.")
```

```python
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        """ Inicializar atributos de superclase.
        Despues inicializar atributos especificos
        del coche electrico """
        super().__init__(marca, modelo, anio)
        self.bateria = Bateria()

mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
mi_tesla.llenar_deposito()
mi_tesla.bateria.describir_bateria()
```

Instancia como atributo

✓ 0.0s

```
El deposito esta a 100
Este coche tiene una bateria de tamaño 70 -kWh.
```

CONQUER BLOCKS

```python
class Coche:
    def __init__(self, marca, modelo, anio):
        """Inicializamos atributos del coche"""
        self.marca = marca
        self.modelo = modelo
        self.anio = anio
        self.deposito = 0

    def llenar_deposito(self):
        """Simul
        self.dep
        print("E

class Bateria:
    """Un intento simple de modelizar
    una bateria"""

    def __init__(self, tamanio_bateria = 70):
        """Inicializamos los atributos de la bateria"""
        self.tamanio_bateria = tamanio_bateria

    def describir_bateria(self):
        """Imprimir el tamaño de la bateria"""
        print("Este coche tiene una bateria de tamaño",
        self.tamanio_bateria,"-kWh.")
```

```python
class CocheElectrico(Coche):
    """Representa aspectos de un coche,
    especifico para coches electricos"""
    def __init__(self, marca, modelo, anio):
        """ Inicializar atributos de superclase.
        Despues inicializar atributos especificos
        del coche electrico """
        super().__init__(marca, modelo, anio)
        self.bateria = Bateria()

mi_tesla = CocheElectrico("tesla", "modelo s", 2016)
```

```
El deposito esta a 100
Este coche tiene una bateria de tamaño 70 -kWh.
```

TAMBIEN PODRIAMOS AÑADIR UN METODO QUE CALCULE EL RANGO DE RECORRIDO QUE LE QUEDA AL COCHE DAD LA CARGA DE LA BATERIA...

TAMBIEN PODRIAMOS AÑADIR UN METODO QUE CALCULE EL RANGO DE RECORRIDO QUE LE QUEDA AL COCHE DADA LA CARGA DE LA BATERIA…

¿Ese método correspondería a la clase Batería o al CocheEléctrico?

Si tenemos solo un coche quizás podríamos poner el método en la clase Bateria.

Pero si tenemos una linea de coches diferentes, cada uno puede consumir la batería a distinto ritmo. Podría ser conveniente que el método lea el estado de la batería y calcule el rango dado el modelo —> debería estar en la clase CocheElectrico

TAMBIEN PODRIAMOS AÑADIR UN METODO QUE CALCULE EL RANGO DE RECORRIDO QUE LE QUEDA AL COCHE DADA LA CARGA DE LA BATERIA…

Estas cuestiones nos llevan a un punto interesante en el crecimiento como programadores —>>> Estamos en un nivel de lógica superior.

No estamos pensando a un nivel sintáctico. No pensamos en Python si no en como modelizar el mundo a nuestro alrededor con código.

ASÍ QUE...

Algunos enfoques son más eficientes que otros, pero se necesita práctica para encontrar las representaciones más eficientes. Si tu código está funcionando como deseas, ¡lo estás haciendo bien!

No te desanimes si descubres que estás desglosando tus clases y reescribiéndolas varias veces utilizando enfoques diferentes. En la búsqueda de escribir un código preciso y eficiente, todos pasan por este proceso.

CONQUER BLOCKS