# Módulo 4 - Segunda parte - Teoría

**Profesor Adjunto:** Mag. Bioing. Baldezzari Lucas

V2022

---

## Abstracción

¿Cuántos atributos y métodos debe poseer un objeto? ¿Y cuanto de esto debo mostrar al usuario?

La cantidad de atributos y métodos dependerá de **qué tan complejo queramos que sea nuestro objeto**.

La abstracción trata del **nivel de detalle** –atributos y métodos- que queremos modelar del mundo real. Trata de resaltar lo importante.

Debemos concentrarnos en ¿Qué hace? nuestra clase y no en ¿cómo lo hace?.

**...Menos es más...**

## Encapsulamiento

El encapsulamiento nos permite prescindir de informar de atributos, métodos complejos y la implementación de un objeto **para resaltar lo necesario**.

> *Encapsular sirve para ocultar detalles de implementación*.

Los atributos -sobre todo los privados- deberían ser modificados (idealmente) a través de métodos de clase.

Por ejemplo, ¿qué es lo importante de un *Respirador()*? ¿Qué el usuario conozca el funcionamiento del algoritmo que controla las modalidades del flujo entregado al paciente o solo que pueda configurar diferentes modalidades al paciente?

Para mencionar otro ejemplo de encapsulamiento, pensemos en un televisor. En general una persona tiene acceso a ciertas funcionalidades del televisor, como encender/apagar, cambiar de canal, cambiar la entrada (cable, HDMI, VGA, etc), sin embargo la mayoría de las personas no sabe cómo el televisor implementa estas funcionalidades. Los detalles de cambiar de canal se hacen "internamente" y no son visibles por el usuario.

### Motivación para usar encapsulamiento

El encapsulamiento también es una forma de proteger los datos de muestra clase utilizando *atributos y métodos privados*.

En muchos casos podríamos tener métodos privados que sirvan para chequear o monitorear que los atributos de mi objeto están dentro de rangos especícios o bien que sean estos métodos los únicos que puedan modificar ciertos atributos.

Veremos estas aplicaciones en las próximas secciones.

## Atributos y Métodos privados

Algunos legunajes de programación soportan el concepto de *atributo privado* y de *método privado*, los cuales solamente pueden ser accesibles por el objeto, es decir, no podríamos acceder a estos atributos y métodos

desde \"*afuera*" mediante el operador . (punto).

Python no soporta esto directamente, sin embargo, existe una convención entre los desarrolladores y las desarrolladoras de Python de que los atributos y métodos que se declaren como privados no deberían ser modificados ni accesibles por el usuario final.

En la literatura se declara algo como,

> \"*Somos adultos*\"

Imaginemos que cuando creamos nuestra clase en Python la misma tiene un *contrato* que especifica que lo que es declarado como privado, no debe ser accedido ni modificado por el usuario o cliente final y será responsabilidad absoluta del cliente el manejo adecuado de los atributos y métodos.

Son embargo, veremos qué herramientas y metodoloígas utilizar para intentar minimizar al máximo posibles fallas en nuestro programa debido al mal uso de estos atributos y métodos.

Definimos lo siguiente:

**Atributo privado**

> Un atributo que no es modificable por el código del cliente.

**Método privado**

> Un método que no es accesible por el código del cliente.

*¿Cuando debemos declarar un atributo y/o un método como privado?*

Esto dependerá de nuestra aplicación y va muy de la mano con el concepto de encapsulamiento. Siguiendo con el ejemplo del televisor. Podríamos tener un método que cambie de canal haciendo,

```
televisor.cambiarDeCanal(canal = 4)
```

El método *cambiarDeCanal()* intermanente podría acceder a atributos y métodos *privados* para llevar a cabo el cambio de canal y será él quien tenga la potestad de hacerlo. Sería una mala práctica (y algo no recomendable) que el usuario final pueda cambiar el canal directamente haciendo algo como,

```
televisor._canalActual = 4
```

Haciendo esto último estaríamos violando el concepto de encapsulamiento y correríamos el riesgo de que nuestro programa crashee.

## Declarando atributos privados

La sintáxis básica para delcarar un atributo como privado es anteponer al nombre de la variable un *guión bajo*.

```
class MiClase():
    def __init__(self, param):
        self._atributoPrivado = param
```

Como hemos mencionado, los valores de los atributos privados deberían ser editados y/o cambiados **desde la propia clase y no desde afuera**. Esto quiere decir que el cliente o usuario final no tendría que poder (o no debería) acceder a ellos de manera directa.

### Python vs JAVA

A modo de ejemplo, comparemos cómo se declaran atributos privados en JAVA vs Python.

**JAVA**

```java
public class MiClaseJAVA
{
    //Los atributos se declaran fuera de los métodos
    private int CI; //será de sólo lectura
    private int numUCsAprobadas;
    public String name;

    //Constructor de la clase
    public MiClaseJAVA(String name, int CI, int numUCsAprobadas = 0)
        {
            this.CI = param;
            this.name = name;
        }

    //getter
    public String getCI() {
        return CI;
    }

    //getter
    public getUCsAprobadas()
    {
        return numUCsAprobadas;
    }

    //setter
    public setUCsAprobadas(int aprobadas)
    {
        numUCsAprobadas = aprobadas;
    }

    public addUCAprobada(numUCsAprobadas = 1)
    {
        numUCsAprobadas += numUCsAprobadas;
    }

}
```

Si quisiéramos acceder al atributo *CI* de algún objeto de la clase *MiClaseJAVA()* obtendríamos un error ya que JAVA no nos permitiría acceder al atributo desde afuera. Lo mismo aplicaría al atributo numUCsAprobadas.

Sin embargo, sería posible obtener el valor de CI través de lo que se conoce como los *getters*. En los casoso de *numUCsAprobadas* y *CI* podríamos ver sus valores con sus *getters*. Por otro lado, en el caso de *numUCsAprobadas* podríamos setear un número determinado de UCs aprobadas utilizando *setUCsAprobadas()*. Esto último se conoce como *setter*. Como sus nombres lo indican, los *getters* nos dan algo y los *setters* nos permiten setear algo.

En el ejemplo presentado estamos suponiendo que la cédula de identidad es de solo lectura y que nunca la podríamos modificar.

Si se presta atención a los métodos *getCI()*, *getUCsAprobadas()*, *setUCsAprobadas()* y *addUCAprobada()* veremos que el acceso a los atributos *CI* y *numUCsAprobadas* es de manera indirecta, no es el usuario quien accede a ella sino el método de la instancia.

**Python**

```
class MiClasePython():
    def __init__(self, name:str, CI:int, numUCsAprobadas:int):
        self.name = name
        self._CI = CI
        self._numUCsAprobadas
```

Notemos que los atributos *_CI* y *_numUCsAprobadas* empiezan con un guion bajo, lo cual indica que son privados.

A diferencia de JAVA, si quisieramos acceder al atributo _CI de algún objeto *MiClasePython* no tendríamos ningún error.

Veamos...

In [5]:
```
class MiClasePython():
    def __init__(self, name:str, CI:int, numUCsAprobadas:int):
        self.name = name
        self._CI = CI
        self._numUCsAprobadas = numUCsAprobadas

clase = MiClasePython(name = "Morfeo", CI = 555, numUCsAprobadas = 52)
clase._CI #CHAN!
```

Out[5]:
```
555
```

La pregunta que nos podríamos hacer es, ¿cómo solucionamos el problema de que acceder sin inconveniente a algo que se supone es privado?

Una posibilidad es implementar los *getters* y *setters* como vimos en el caso de JAVA, lo cual nos quedaría algo así.

In [11]:
```
class MiClasePython():
    def __init__(self, name:str, CI:int, numUCsAprobadas:int):
        self.name = name
        self._CI = CI
        self._numUCsAprobadas = numUCsAprobadas

    def getCI(self):
        return self._CI

    def setnumUCsAprobadas(self, numUCsAprobadas):
        self._numUCsAprobadas = numUCsAprobadas

    def getUCsAprobadas(self):
        return self._numUCsAprobadas

    def addUCAprobada(self, numUCsAprobadas = 1):
        self._numUCsAprobadas += numUCsAprobadas

clase = MiClasePython(name = "Morfeo", CI = 64445551, numUCsAprobadas = 40)

clase.getCI()
print(clase.getUCsAprobadas())
clase.addUCAprobada()
print(clase.getUCsAprobadas())
```

Out[11]:
```
40
41
2312312341224
```

## Utilizando @property

Si bien lo anterior es completamente válido, no es una practica habitual en Python el uso de getters y setters.

Python tiene una mejor forma de proveer acceso a los atributos que queremos trabajar como privados a través de lo que se conoce como *propiedad* o *property* en inglés.

### Definición de Property

> Un método para acceder o modificar el valor de un atributo de un objeto. Permite que un cliente pueda utilizar una sintáxis especial para que parezca que se esta accediendo al atributo de manera directa en vez de llamar al método en cuestión.

Podemos utilizar el decorador *@property* para hacer que un método de un objeto se comporte como una propiedad.

Veamos esto con un ejemplo.

### Getters

La sintáxis para definir un *getter* con el uso de decoradores es,

```
class MiClase():
        @property
        def nombrePropiedad(self): #este método se convierte en un getter
            ##hacer algo
            pass
```

### Setters

La sintáxis para definir un *setter* con el uso de decoradores es,

```
class ClassName():
    @nombrePropiedad.setter
    def nombrePropiedad(self, parameter): #se comportará como un setter
            ##hacer algo
            pass
```

Veamos...

In [28]:

```
class MiClasePython():
    def __init__(self, name:str, CI:int, numUCsAprobadas:int):
        """Constructor
        Params:
            - name (str): nombre
            - CI (int): Cédula identidad (sólo lectura)
            - numUCsAprobadas (int): Cantidad de UCs aprobadas.
        """
        self.name = name
        self._cedIdentForre = CI
        self._numUCsAprobadas = numUCsAprobadas

    @property
    def CI(self): #este método se convierte en un getter
        """Propiedad de CI"""
        return self._cedIdentForre

    @property
    def numUCsAprobadas(self): #este método se convierte en un getter
        return self._numUCsAprobadas

    @numUCsAprobadas.setter
```

```
    def numUCsAprobadas(self, newUCs): #este método se convierte en un setter
        self._numUCsAprobadas = newUCs

    def addUCAprobada(self, numUCsAprobadas = 1):
        self._numUCsAprobadas += numUCsAprobadas

clase = MiClasePython(name = "Morfeo", CI = 64445551, numUCsAprobadas = 0)
clase.CI

# si quisieramos dar un valor a CI obtendríamos un error.
# clase.CI = 45556663

# Esto no pasa con la variable numUCsAprobadas
# clase.numUCsAprobadas
# clase.numUCsAprobadas = 40
# clase.numUCsAprobadas
# help(clase)
```

Out[28]:
```
64445551
```

El decorador *@property* convierte los métodos *CI()* y *numUCsAprobadas()* en *getters*. Podemos ver también que haciendo *@numUCsAprobadas.setter* podemos lograr que *numUCsAprobadas()* también se comporte como un *setter*. El intérprete de python dinamicamente utilizará uno u otro método dependiendo del contexto. Por ejemplo, al hacer,

```
clase.numUCsAprobadas = 52
```

el intérprete sabe que debe llamar al método *numUCsAprobadas* para en su forma de *setter* para asignar un nuevo valor al atributo *_numUCsAprobadas* ya que la propiedad esta definida como sigue,

```
@numUCsAprobadas.setter
def numUCsAprobadas(self, newUCs): #este método se convierte en un setter
    self._numUCsAprobadas = newUCs
```

Si se presta atención a la implementación de la clase *MiClasePython()* podemos ver que no tenemos un setter para *_CI*, solamente tenemos un getter, es decir, estamos tratando al valor *_CI* como *solo lectura*. Si el usuario quisiera cambiar el valor del atributo haciendo *clase.CI = 64445550* obtendría un error.

Veamos.

In [4]:
```
clase.CI = 64445550
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_21180/3707945039.py in <module>
----> 1 clase.CI = 64445550

AttributeError: can't set attribute 'CI'
```

En los ejemplos anteriores podemos ver que en realidad no estamos accediendo directamente a los atributos, sino que son los métodos, comportándose como propiedades, los que acceden a ellos.

Esto nos permite aumentar el nivel de encapsulamiento.

Sin embargo, a pesasr del uso de *property* si quisiéramos podríamos acceder estos atributos con el operador punto.

Veamos...

In [ ]:
```
clase._CI
```

Pero como ya hemos dicho, la responsabilidad final es del cliente/usuario. Nosotros debemos hacer un buen uso del encapsulamiento y de las herramientas que nos ofrece Python, pero no más que eso.

## Utilizando la función built-in property()

Otra posibilidad al decorador *@property* es usar la función *property()* la cual tiene la siguiente sintáxis.

```
class property(fget=None, fset=None, fdel=None, doc=None)
```

In [ ]:

```
### Ejemplo de la documentación

class C:
    def __init__(self):
        self._x = None

    def getx(self):
        return self._x

    def setx(self, value):
        self._x = value

    def delx(self):
        del self._x

    x = property(getx, setx, delx, "I'm the 'x' property.")

c = C()

c.x = 1 ## acá llamamos al método set
print(c.x) ##print llama al método getx
# help(c) ##aca llamamos a la documentacións
```

In [ ]:

```
#Otro ejemplo
class Worker():
    def __init__(self, name:str, salario:float):
        "Constructor"
        self._name = name
        self._salario = salario

    def _getnombre(self):
        return self._name

    def _getsalario(self):
        return self._salario

    def _setnombre(self, _name):
        self._name = _name

    def _setsalario(self, salario):
        self._salario = salario

    nombre = property(fget = _getnombre,
                      fset = _setnombre,
                      doc = "Propiedad 'nombre'")
    salario = property(fget = _getsalario,
                       doc = "Propiedad 'salario'")

# worker1 = Worker("Carrie-Anne Moss", 58000)
# worker1.name = "Trinity"  # Llama al método "fset"
# print(worker1.name, worker1.salario)  # Llama a los métodos "fget" de name y salario
```

La única diferencia entre la función *property()* y el decorador *@property* es que la función recibe los métodos como parámetros, pero ambas logran que los métodos funcionen como propiedades.

## Bonus! Aumentando la privacidad con dos guines bajos

Si quisieramos aumentar más el nivel de privacidad de un atributo podríamos anteponer al nombre del mismo dos guines bajos.

```
class MiClase():
        def __init__(self):
            self.__atributoPrivado
```

Haciendo esto, el ínterprete de Python dinámicamente cambia el nombre de la variable de tal manera de intentar aumentar el nivel de encapsulamiento.

Veamos como se ve esto...

In [36]:
```
class MiClase():
    def __init__(self):
        self._atr1 = "Holis"
        self.__atributoPrivado = "Nooooooooo"


clase = MiClase()
# clase._atr1
# clase.__atributoPrivado
clase._MiClase__atributoPrivado
```

Out[36]:
```
'Nooooooooo'
```

Vemos que si intenetamos acceder a *__atributoPrivado* obtenemos un error, esto es debido a que la variable no se llama *__atributoPrivado* sino que se llama *_MiClase__atributoPrivado*.

Esto lo podemos constatar usando la función *dir()* que trae Python, la cual nos devuelve una lista de los atributos y métodos que trae la clase que creamos.

In [ ]:
```
# dir(clase)
```

Podemos ver que en la lista anterior figura el atributo *_MiClase__atributoPrivado*.

La pregunta es, ¿es nuestro atributo *_MiClase__atributoPrivado* realmente privado?. La respuesta es que no, ya que si quisieramos podríamos acceder a dicho atributo y modificarlo o ver su valor. Sin embargo, llegar a estos extremos es realmente una muy mala práctica de programación.

In [ ]:
```
## Esto no debe hacerse!!
clase._MiClase__atributoPrivado
```

In [ ]:
```
class Test():
    """Una clase para testeo"""
    def __init__(self):
        self.atributo1 = "APub1"
        self.atributo2 = "APub2"
        self.__atributoPrivado = "APriv"

    def __dir__(self):
        return ['atributo1', 'atributo2']
```

```
t = Test()
dir(t)
```

### Declarando métodos privados

Al igual que con los atributos, un método privado se declara anteponiendo un guion bajo antes del nombre del método. Podríamos también agregar dos guiones bajos, dificultando un poco más el acceso a dicho método.

Pero, en cualquier caso podremos acceder a los métodos.

Veamos.

In [ ]:
```
class Test2():
    """Una clase para testeo"""
    def __init__(self):
        self.atributo1 = "APub1"
        self.atributo2 = "APub2"
        self.__atributoPrivado = "APriv"

    def _privateMethod(self):
        print("Soy un método declarado con _")

    def __otherPrivateMethod(self):
        print("Soy un método declarado con __")

    def __dir__(self):
        return ['atributo1', 'atributo2']

# t2 = Test2()
# t2._privateMethod() #Mala idea acceder a un astributo privado
# t2._Test2__otherPrivateMethod() #Pésima idea de acceso!!
```

Los métodos privados, al igual que con los atributos, deben ser accedidos solamente por el objeto y no desde el exterior.

## Ejercicio

Escriba una clase -puede utilizar ejercicios que ya ha trabajado o crear una nueva- que trabaje con atributos privados y utilice el decorador @property para accceder y/o modificar atributos declarados como privados.

---

## Herencia

La herencia permite que una **clase herede o extienda sus atributos y métodos a partir de una clase base**. En ocasiones, no es necesario empezar desde cero la escritura de una clase. Si la clase que queremos escribir es una *especialización* de otra clase que tenemos escrita, tal vez se podría utilizar herencia. Cuando una clase *hereda* o *extiende* de otra clase, toma todos los atributos y métodos de la clase base. Se supone que una clase que deriva de otra tiene alguna relación con la clase base.

La clase original suele llamarse *base*, *padre*, *madre* o *super class* (de superior), mientras que la clase que hereda se llama *clase derivada*, *hija* (de *child* en inglés).

Si bien la clase derivada es la que hereda (extiende) todos los atributos y métodos de la clase base, también podría sobrescribir (*overrride*) o bien sobrecargar (*overload*) los atributos y métodos de esta última.

## Sintáxis de herencia

Para que una clase extienda o herede de una clase base basta con pasarle el nombre de la clase base entre paréntisis en la declaración de la clase hija.

```
class MiClase(ClaseBase):
        def __init__(self):
            pass
```

En el ejemplo anterior, *MiClase* extenderá los atributos y métodos de *ClaseBase*.

## Método *__init__()* de la clase hija

A menudo cuando escribimos una clase que se extiende de otra nos interesa utilizar el método *__init__()* de la clase base. De esta manera inicializaremos cualquier atributo que este definido en la super clase y estarán disponible para la clase hija.

La forma de llamar a un método de la super clase es mediante la función especial llamada *super()*.

```
class ClaseBase():
        def __init__(self, atributoBase):
            self.atributoBase = atributoBase

    class MiClase(ClaseBase):
        def __init__(self, atributo1, atributo2):
            super().__init__(atributo1)
            self.atributoHija = atributo2
```

Veamos esto con un ejemplo.

In [ ]:

```
"""
Original: "Python crash course - Hands on - Project-Bases Introduction to programming"
Editado por: Prof. adjunto Mag. Bioing. BALDEZZARI Lucas.
"""

class Car:
    """Clase para intentar representar un auto"""
    def __init__(self, fabricante:str, modelo:str, year:int):
        self.fabricante = fabricante
        self.modelo = modelo
        self.year = year
        self.odometer_reading = 0 #si es cero indica que el auto es 0km

    def getDescriptiveName(self):
        """Info del auto"""
        return f"{self.fabricante} - {self.modelo} - {self.year}".title()

    def readOdometer(self):
        """Kilometraje actual"""
        return self.odometer_reading

    def updateOdometer(self, kilometraje:int):
        """Se actualiza los kilómetros recorrido por el vehículo"""
        if kilometraje >= self.odometer_reading:
            self.odometer_reading = kilometraje
        else:
            print("No puede volver a atrás el odómetro, pillin!")

    def incrementOdometer(self, kms:int):
        """Aumento del kilometraje en una determinada cantidad de kms
        Params:
            - kms: Kilómetros recorridos
```

```
        """
        self.odometer_reading += kms

    def carAutonomy(self, consumo:int):
        """Cálculo de autonomía según el tipo de vehículo"""
        print("Aún no tengo definido qué tipo de vehículo soy")
        return None
```

In [ ]:
```
## Clase de vehículo eléctrico

class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""
    def __init__(self, fabricante, modelo, year):
        """Iniciamos atributos"""
        super().__init__(fabricante, modelo, year)
```

In [ ]:
```
## Algunas pruebas
myTesla = ElectricCar('Tesla', 'modelo s', 2021)
#Accediento a los atributos de la clase padre
# print(myTesla.year, myTesla.modelo)

# print(myTesla.getDescriptiveName())
# print(f"El auto tiene {myTesla.readOdometer()}km")
```

En la línea 31 se llama al constructor de la clase base a través de la función especial *__super__()* para inicializar los atributos de la clase base.

En la línea 33 creamos un objeto *ElectricCar* y le pasamos la misma información que le pasaríamos a un objeto del tipo *Car*. De esta manera vemos que la herencia funciona correctamente. Al crear una instancia del objeto *ElectricCar* estamos llamando al método *__init__()* de la clase *ElectricCar* y luego este llamará al método *__init__()* de la clase base *Car*.

## Definiendo atributos y métodos en la clase derivada

Hasta ahora hemos visto cómo podemos inicializar los atributos de la clase base *Car* y hemos utilizado alguno de sus métodos.

Una vez heredados los atributos y métodos de la clase base podríamos querer agregar nuevos para así diferenciar la clase hija de su clase padre/madre.

Agreguemos algunos atributos y métodos propios de un auto eléctrico, como podría ser la capacidad de la batería expresada en $kilowatts \times h$ y un método para leer una descripción de la misma.

In [ ]:
```
class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""
    def __init__(self, fabricante, modelo, year):
        """Iniciamos atributos"""
        super().__init__(fabricante, modelo, year)
        self._battery = 75 #kWh
        self.batteryCharge = 1 # 1 = 100%

    def getBatteryInfo(self):
        """Info de la batería"""
        return f"El auto posee una batería de {self._battery}kWh"

    def updateBatteryCharge(self, charge:int):
        """Actualiza la carga de la batería"""
        self.batteryCharge = charge #%
```

Hemos agregado los atributos *_battery* y *batteryCharge* y el método *getBatteryInfo()* a nuestra clase *ElectricCar*. Es importante recordar que las variables *_battery* y *batteryCharge* son atributos de instancia que estarán presentes en cada instancia nueva de *ElectricCar* pero **no** serán parte de *Car*.

Podríamos agregar tantos atributos y métodos como quisiéramos. Sin embargo, recordemos el concepto de *abstracción*, el cual nos dice que debemos hacer nuestros objetos lo suficientemente abstractos para no complejizarlos demasiado, pero al mismo tiempo que nos permitan realizar las tareas que queremos realizar.

Creemos un nuevo vehículo eléctrico y verifiquemos lo que hemos implementado.

In [ ]:
```
otroTesla = ElectricCar("Tesla", "S", 2022)
print(otroTesla.getBatteryInfo())
```

## Sobreescribiendo y sobrecargando métodos en la clase derivada

Una de las ventajas de la herencia es que podemos sobreescribir o sobrecargar los métodos de la clase base.

Tomemos la clase *ElectricCar* y sobreescribamos el método *autonomy()* para que nos entregue un estimado de la autonomía del vehículo en base a un consumo.

Una opción podría ser,

```
##************************##
    ## Dentro de ElectricCar ##
    ##************************##
    def getAutonomy(self, consumo:int) -> float:
        """Cálculo de autonomía según el tipo de vehículo.
        Retorna la estimación de cuantos km puede hacer el vehículo
        Params:
            - consumo: consumo actual en kWh/100km
        Return:
            - autonomía
        """
        return (self._battery/consumo*self.batteryCharge)*100#km
```

## Objetos como atributos

En algunas ocasiones cuando modelamos algo del mundo real en código uno podría agregar más y más líneas de código a nuestra clase a medida que agregamos detalles.

Bajo estas circuinstancias uno podría darse cuanta que parte del código que se esta implementando podría ser otra clase. Entonces podríamos tomar una clase y partirla en clases mas pequeñas.

En el ejemplo de la clase *ElectricCar* podríamos pensar que el banco de baterías podría ser tomado como un objeto distinto y que formará parte del vehículo eléctrico como un atributo.

Definamos entonces una clase *Battery* con algunos atributos y métodos.

In [ ]:
```
class Battery:
    """Clase para simular un banco de baterías"""
    def __init__(self, batterySize:int = 75):
        """Iniciamos atributos del banco de baterías"""
        self._batterySize = batterySize
        self._batteryCharge = 1 # 1 = 100%

    def getBatteryInfo(self):
```

```
        """Info de la batería"""
        return f"El banco de batería es de {self._batterySize}kWh"

    def getBatteryCharge(self):
        """Retorna carga de la batería"""
        return self._batteryCharge

    def updateBatteryCharge(self, charge:int):
        """Actualiza la carga de la batería"""
        if charge <0 or charge >1:
            print("La carga de la batería debe ser un valor entre 0 y 1")
            return None #se podría retornar un error
        else:
            self._batteryCharge = charge #%

    def batteryAutonomy(self, consumo:int) -> float:
        """Cálculo de autonomía según el tipo de vehículo.
        Retorna la estimación de cuantos km puede hacer el vehículo.
        Params:
            - consumo: consumo actual en kWh/100km
        Return:
            - autonomía
        """
        return (self._batterySize/consumo*self._batteryCharge)*100#km
```

Observemos que la clase *Battery* posee todos los métodos que teníamos en *ElectricCar*. Desde un punto de vista lógico, esto tiene sentido ya que el banco de batería si bien forma parte del vehículo eléctrico es una de las tantas piezas que lo conforman (como las ruedas, luces, etc).

Imaginemos por un momento que quisieramos agregar dentro de la clase *ElectricCar* atributos y métodos para simular ruedas, luces, etc. Esto podría ser realmente contraproducente. En estos casos es mejor tomar las ruedas y las luces como objetos separados, cada uno con sus atributos y métodos y luego usarlos como parte del *ElectricCar*.

Haciendo esto estaremos cumpliendo con los conceptos de abstracción y encapsulamiento, además de que haremos nuestro código mucho más legible y depurable en un futuro.

In [ ]:
```
## Testeando baterías
battery = Battery(75)
print(battery.getBatteryInfo())
battery.updateBatteryCharge(0.8)
print(f"La carga de la batería es {battery.getBatteryCharge()}")
print(f"La aunomoía es de {battery.batteryAutonomy(20)} kms")
```

Una vez creada la clase *Battery* nuestra clase *ElectricCar* nos podría quedar de la siguiente manera.

In [ ]:
```
class ElectricCar(Car):
    """Represent aspects of a car, specific to electric vehicles."""

    def __init__(self, fabricante, modelo, year):
        """Iniciamos atributos"""
        super().__init__(fabricante, modelo, year)
        self.battery = Battery(batterySize = 75)

    def carAutonomy(self, consumo:int) -> float:
        """Cálculo de autonomía según el tipo de vehículo"""
        return self.battery.batteryAutonomy(consumo)
```

Podemos ver que ahora la clase *ElectricCar* posee un atributo *Battery*. Esto puede ser interpretado como una

relación de **composición** ya que un auto eléctrico no podría funcionar sin un banco de baterías.

## Importando módulos

Cada script que implementamos podría ser considerado como un *módulo* o *librerías* en Python y por lo tanto podríamos importarlo utilizando la sentencia *import*. Información oficial de Python en Módulos.

Veamos algunos ejemplos de cómo utilizar esto con paquetes propios de Python, de terceros y con *car.py*.

### Importando librearías completas

El primer ejemplo sirve para mostrar cómo podemos importar una librería completa.

```
import library
```

En este caso estamos importando la librería *library*. Podríamos acceder a cualquiera de las funciones y/o clases dentro de dicha librería mediante el operador *punto*.

### Importando algunas funciones

Podríamos también importar aquellas funciones y/o clases de la librería que usaremos. Esto muchas veces es útil para evitar cargar en el *scope* o *ambiente* de trabajo con funciones y clases que no usaremos. En estos casos la sintáxis es la siguiente.

```
from library import func1, func2
```

En el ejemplo anterior hemos importado *func1* y *func2* desde la librería *library*.

Veamos algunos ejemplos sencillos.

In [ ]:
```
import os #libreria propia de python
from math import cos, sin, pi, radians
from math import pi

carpetaActual = os.getcwd()
print(carpetaActual[-24:])

n = 180
print(f"Coseno de n es {cos(radians(n)):0.2f}")
print(f"Seno de n es {sin(radians(n)):0.2f}")
```

En el ejemplo anterior hemos importado las funciones *cos*, *sin* y *radinas* de la librería math propia de Python. También hemos importado el número *pi* desde math, el cual es un valor constante que representa el número Pi.

Sin embargo, de la librería *os* hemos importado todas sus funciones, por tal razón podemos accedor a la función *getcwd()* mediante el operador punto.

### Importando módulos de terceras partes

Importar un módulos creados por otras personas es exactamente igual a lo que hemos visto. La diferencia es que debemos tener instalada dicha librería antes de poder importarla o importar algo de ella.

Veamos esto con un ejemplo.

In [ ]:
```
import numpy#la librería numpy ya esta instala

matrix = numpy.zeros((2,3))
print(matrix)
```

Podríamos importar una librería y asignarle un nombre diferente, esto lo hacemos de la siguiente manera.

In [ ]:
```
import numpy as np #ahora el ínterprete de Python tomará a numpy como np

otraMatrix = np.zeros((5,5))
print(otraMatrix)
```

### Importando módulos propios

Podríamos también importar nuestros propios módulos. Las librerías de python están en la carpeta */pythonx.x/scripts/*. No obstante, la forma más sencilla de utilizar un módulo propios es colocando el o los archivos dentro del mismo directorio donde tenemos.

Vamos a utilizar el módulo *car.py*.

In [ ]:
```
import car
from car import Battery
from car import ElectricCar
from car import ElectricCar as autitoABata

# unTesla = car.ElectricCar("Tesla", "M", 2019)
# unTesla.updateOdometer(32500)
# print(f"kilometraje actual del {unTesla.getDescriptiveName()} es {unTesla.readOdometer(

# bmw = ElectricCar("BMW", "i4", 2022)
# print(f"kilometraje actual del {bmw.getDescriptiveName()} es {bmw.readOdometer()} kms"+"

# otroBMW = autitoABata("BMW", "iX", 2021)
# otroBMW.updateOdometer(12500)
# print(f"kilometraje actual del {otroBMW.getDescriptiveName()} es {otroBMW.readOdometer(

# bata = Battery(90)
# print(bata.getBatteryInfo())
# bata.updateBatteryCharge(0.7)
# print(f"La carga de la batería es {bata.getBatteryCharge()*100}%")
# print(f"La aunomoía es de {bata.batteryAutonomy(20)} kms")
```

## Utilizando nuestros módulos como scripts

En algunas ocasiones quisieramos que nuestros módulos funcionen como script, es decir, cada vez que ejecutemos nuestro archivo *car.py* este haga algo.

Esto lo podemos hacer creando un método llamado *main()*.

Supongamos que estemos dentro del archivo *car.py*, entonces podríamos hacer.

```
##############################
## Dentro del archivo car.py ##
##############################
def main():
    ## Testeando baterías
    battery = Battery(75)
    print(battery.getBatteryInfo())
    battery.updateBatteryCharge(0.8)
    print(battery.getBatteryCharge())
    print(f"La aunomoía es de {battery.batteryAutonomy(20)}kms")

    mitesla = ElectricCar('Tesla', 'modelo s', 2021)
```

```
    print(f"La carga de la batería es {battery.getBatteryCharge()}")
    print(f"Autnomía de {mitesla.carAutonomy(29):0.2f}kms")

if __name__ == "__main__":
    main()
```

Si ejecutamos el archivo *car.py* veríamos lo que ejecutamos dentro de *main()*.

In [ ]:
```
# !python car.py
```

¿Qué pasaría si no implementamos la función *main()*?

En ese caso, cada vez que importemos algún módulo ya sea completamente o alguna de sus funciones, estaríamos ejecutandolo como script.

Veamos...

In [ ]:
```
## TODO
import testModule
```

In [ ]:
```
# !python testModule.py
```

### Ejercicio

Se pide,

- Implemente una clase llamada *HybridCar* que extienda de la clase *Car*.
- Utilice la clase *Battery* para dotar al vehículo híbrido de una batería.
- Implemente una clase *gasTank* la cual servirá para dotar al vehículo de un tanque de nafta. Defina métodos y atributos que crea conveniente.

Escriba la clase dentro del archivo *car.py* y en una nueva celda importe *HybridCar* y realice algunas pruebas de funcionamiento de su implementación.

## Bonus! Clase suprema *object*

Técnicmaente, cada clase que creamos utiliza herencia. Todas las clases de python son en realidad subclases de una clase llamada **object**.

Esta clase provee una cierta cantidad de datos y atributos y algunas funcionalidades implícitas de tal forma de que todos los objetos sean tratados consistentemente por el intérprete de Python.

Veamos un ejemplo.

```
class testing(object):
    pass
```

La clase *testing* extiende de *object*. Podriamos obtener el mismo resultado si hubieramos hecho,

```
class testing:
    pass
```

Es decir que si no especificámos nada a la hora de declarar la clase, el intérprete de Python entenderá que nuestra clase extiende de *object*.

Es importante mencionar que a través de herencia podemos generar **jerarquías** de clases.

La herencia nos da reusabilidad y legibilidad en nuestro código.

Python acepta herencia múltiple.

In [ ]:
```
class testing1(object):
    pass

t = testing1()
print(dir(t))
```

In [ ]:
```
class testing2:
    pass

c = testing2()
print(dir(c))
```

Podemos ver que las clases *testing1* y *testing2* extienden de *object* independientemente de si al declararlas le pasamos *object* o no.

**Ejemplo completo**

Un ejemplo completo de herencia y también de composición puede verse en Inheritance and Composition: A Python OOP Guide

## Polimorfismo

El polimorfismo es la capacidad de que un método o un objeto se comporten de manera diferente según el contexto.

En otras palabras, el polimorfismo es la habilidad (en programación) de utilizar la misma interfaz de un método u objeto en diferentes tipos de escenarios.

Podemos implementar polimorfismo sobrescribiendo un método o sobrecargando un método.

Veamos un ejemplo.

In [ ]:
```
class Shape():
    """Clase abstracta"""
    def __init__(self, name):
        self.name = name

    def area(self):
        return None

from math import pi

class Circle(Shape):
    """Clase círculo"""
    def __init__(self, name, radio):
        super().__init__(name)
        self.radio = radio

    def area(self):
        return pi*self.radio**2

class Square(Shape):
    """Clase cuadrado"""
    def __init__(self,name, lado):
        super().__init__(name)
```

```
        self.lado = lado

    def area(self):
        return self.lado*self.lado

shape1 = Circle("Soy un circulo", 2)
print(shape1.name)
print(f"{shape1.area():.02f} cm cuadrados", "\n")

shape2 = Square("Soy un cuadrado", 5)
print(shape2.name)
print(f"{shape2.area()} cm cuadrados")
```

# Fin

Aquí finaliza la parte 2 de la teoría del Módulo 4.