PROGRAMACIÓN III

Clase 6

Clase 5

Clases y objetos

- **Paradigmas de programación.**
- **Programación estructurada vs POO.**
- **Clases, objetos y atributos.**
- **Métodos de clase y métodos especiales: init, del y str.**

Clase 6

Colaboración entre clases y Encapsulamiento

- Mensajes y Métodos.
- Colaboración entre clases.
- Atributos de clase.
- Composición
- Encapsular atributos y métodos.
- Decorators.

Clase 7

Herencia y Polimorfismo

- Herencia.
- Polimorfismo.
- Herencia simple y múltiple.
- Clases Abstractas.
- Diagrama de Clases.

Colaboración entre clases y encapsulamiento

Colaboración de clases

Normalmente en un problema resuelto con la metodología de programación orientada a objetos no interviene una sola clase, sino que hay muchas clases que interactúan y se comunican.
Esta colaboración entre clases se manifiesta como la posibilidad de utilizar en una clase objetos que son instancias de una clase diferente.
Este mecanismo potencia el alcance que tiene la Programación Orientada a Objetos, facilitando aún más el desarrollo de soluciones a problemas complejos sin necesidad de escribir código extenso o difícil de leer.

Colaboración de clases

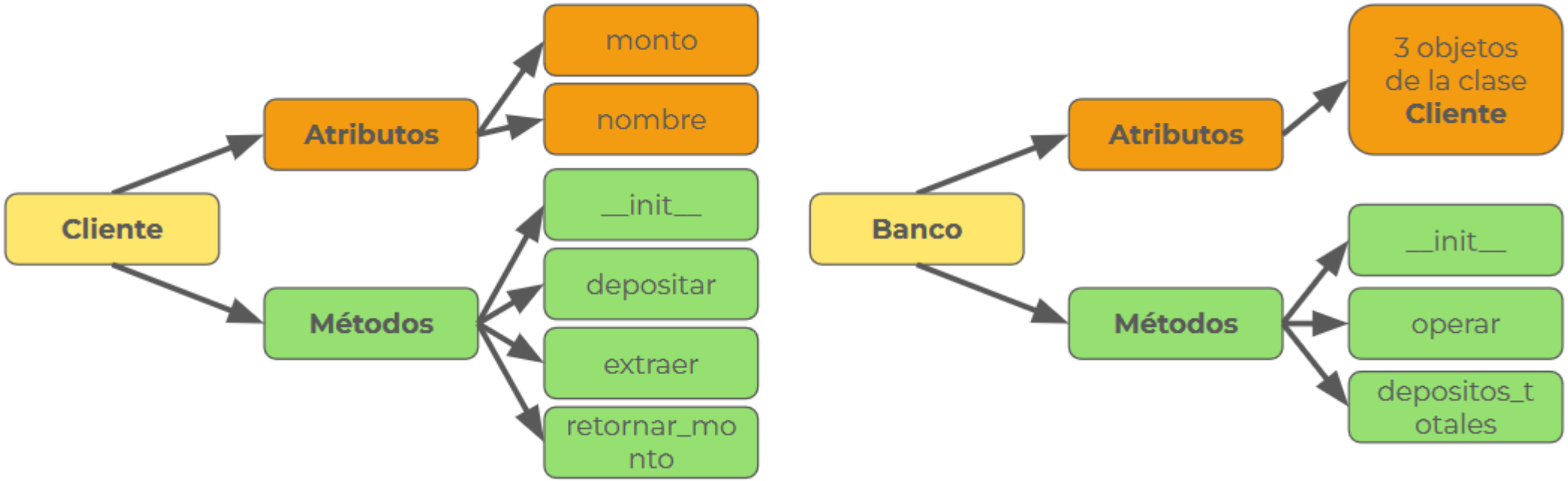
Para entender cómo se produce la colaboración entre clases vamos a desarrollar algunos ejemplos, explicando paso a paso que estamos haciendo.
Enunciado 1:

- ✓ Un *banco* tiene 3 *clientes* que pueden hacer *depósitos* y *extracciones*.
- ✓ El banco necesita obtener, al final del día, un reporte de la cantidad de dinero que sus clientes han depositado.

Del enunciado se deduce que necesitamos objetos de dos clases: *clientes* y *bancos*. Para cada una de estas entidades necesitamos crear una clase, y luego instanciar la cantidad necesaria de cada objeto.

Colaboración de clases

Identificadas las clases, que llamaremos Cliente y Banco respectivamente, definimos qué atributos y métodos necesitamos implementar en cada una.



Colaboración de clases | Clase Cliente

En el método __**init**__ se inicializan los atributos nombre con el valor del parámetro, y monto con el valor cero.
Los métodos depositar, extraer, retornar_monto e imprimir se definen como se ve a la derecha.
No se instancian objetos de esta clase en el cuerpo del programa, ya que serán atributos de la clase Banco.

Clase Cliente:

```
class Cliente:
    def __init__(self,nombre):
        self.nombre=nombre
        self.monto=0

    def depositar(self,monto):
        self.monto=self.monto+monto
    def extraer(self,monto):
        self.monto=self.monto-monto
    def retornar_monto(self):
        return self.monto
    def imprimir(self):
        print("{} tiene depositada la suma de {}"
        .format(self.nombre,self.monto))
```

Colaboración de clases | Banco

Al definir la **clase Banco** se incluye en su constructor la instancia de tres objetos de la **clase Cliente**. Estos objetos serán utilizados por la clase Banco para que su instancia (sólo se instancia un objeto de la clase Banco) sea capaz de operar con los clientes.

En el método operar se invocan algunos métodos de la clase Cliente, para generar las transacciones (depósitos y extracciones).

Con el método depositos_totales se calcula la suma de los montos depositados por los objetos de la clase Cliente, y se guardan en la variable total, que luego se muestra en la terminal mediante print()

Colaboración de clases | Banco

Clase Banco:

```
class Banco:
    def __init__(self):
        self.cliente1=Cliente("Juan")
        self.cliente2=Cliente("Ana")
        self.cliente3=Cliente("Diego")
    def operar(self):
        self.cliente1.depositar(100)
        self.cliente2.depositar(150)
        self.cliente3.depositar(200)
        self.cliente3.extraer(150)
    def depositos_totales(self):
        total=self.cliente1.retornar_monto()+self.cliente2.retornar_monto()+self.cliente3.retornar_monto(
)
        print("El total de dinero en el banco es: {}".format(total))
        self.cliente1.imprimir()
        self.cliente2.imprimir()
        self.cliente3.imprimir()
```

Colaboración de clases | Programa principal

En el programa principal se instancia un objeto de la clase Banco, y se invocan sus *métodos operar() y depositos_totales():*

Programa principal

```
# Programa principal:
banco1 = Banco()
banco1.operar()
banco1.depositos_totales()
```

Terminal

```
El total de dinero en el banco es: 300
Juan tiene depositada la suma de 100
Ana tiene depositada la suma de 150
Diego tiene depositada la suma de 50
```

Al invocar el método *operar()* de la clase Banco se llama a los métodos *depositar()* y *extraer()* de la clase Cliente. Al llamar al método *depositos_totales()* de la clase Banco se llama al método *retornar_monto()* e *imprimir()* de la clase Cliente. Ambas clases colaboran para resolver la situación planteada en el enunciado.

Atributos de clase

Los atributos de clase, a diferencia de los atributos de instancia, son compartidos por todos los objetos de dicha clase. Los atributos de clase se definen dentro de la clase pero fuera de sus métodos:

Programa principal

```
class Persona:
    variable=20 # Atributo de clase

    def __init__(self, nombre):
        self.nombre=nombre

p1 = Persona("Aldo")
p2 = Persona("Berta")
print("Valor en p1:",p1.variable)
print("Valor en p2:",p2.variable)
p1.variable = 30
print("Valor en p1:",p1.variable)
print("Valor en p2:",p2.variable)
```

Terminal

```
Valor en p1: 20
Valor en p2: 20
Valor en p1: 30
Valor en p2: 20
```

Importante: `p1.variable = 30` *no* modifica el valor del *atributo* de clase del mismo nombre, sino que crea *un atributo de instancia* que existe solo para ese objeto.
Por lo tanto, cuando volvemos a consultar su valor, vemos “30” que es el valor de su atributo de instancia.
Si existe un atributo de clase y uno de instancia con el mismo nombre, tiene prioridad el de la instancia. El cambio de valor no afecta a p2, que sigue mostrando el valor del atributo de clase.

Atributos de clase

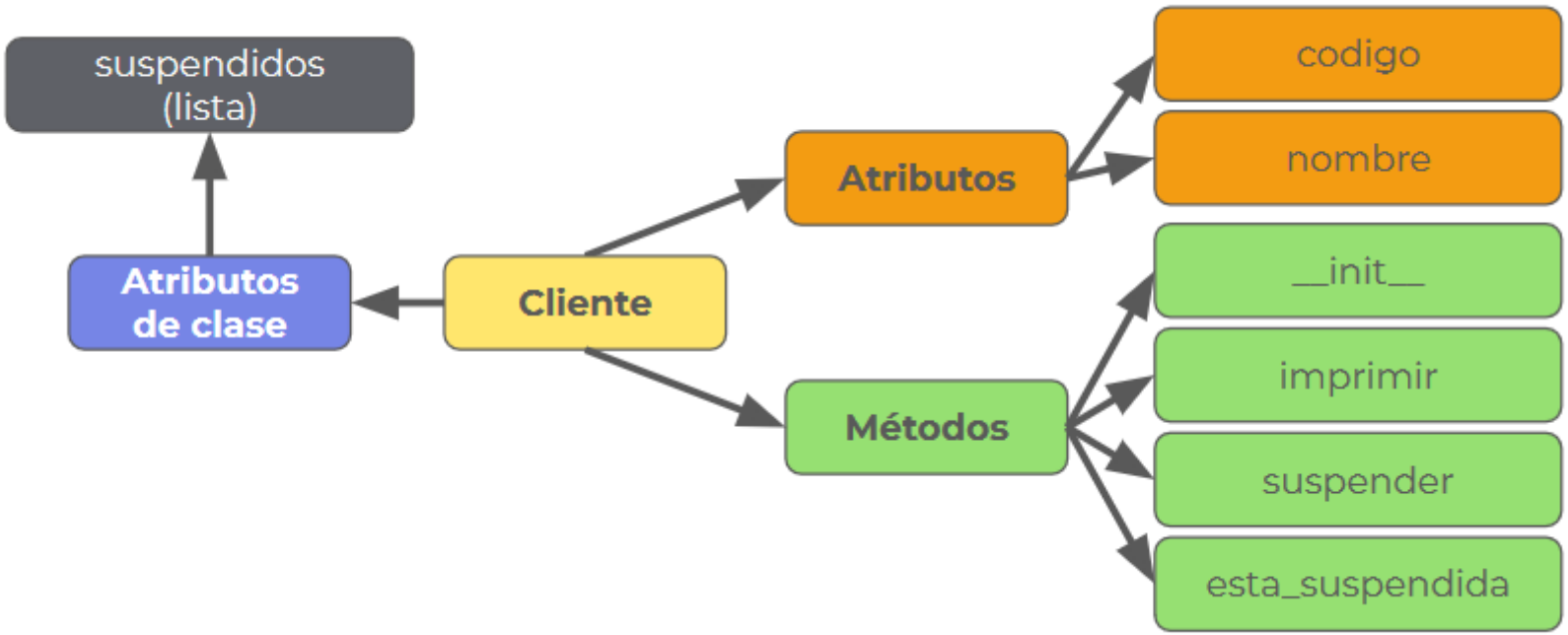
Apliquemos el concepto de atributo de clase en un caso concreto, que implementa una lista compartida entre los objetos de esa clase:

Enunciado 2:

1. Define una clase Cliente que almacena un código de cliente y un nombre.
2. En la clase Cliente define un atributo de clase de tipo lista que almacene todos los clientes con sus cuentas corrientes suspendidas.
3. Imprimir por pantalla todos los datos de clientes y el estado en que se encuentra su cuenta corriente.

Atributos de clase | Clase Cliente

La clase Cliente posee varios atributos de instancia, métodos y un atributo de clase, que es del tipo lista:



Atributos de clase | Clase Cliente

La lista *suspendidos* es compartida por todos los objetos de la clase.

El método *imprimir* muestra los datos y estado de la cuenta de cada instancia de la clase Cliente.

El método *suspender* agrega el código de un cliente a la lista del cliente suspendido.

Clase Cliente (Parte I):

```
class Cliente:
    suspendidos=[] #Atributo de Clase
    def __init__(self,codigo,nombre):
        self.codigo=codigo #Atributo de instancia
        self.nombre=nombre #Atributo de instancia

    def imprimir(self):
        print(f"Codigo: {self.codigo}")
        print(f"Nombre: {self.nombre}")
        self.esta_suspendido()

    def suspender(self):
        Cliente.suspendidos.append(self.codigo)
```

Atributos de clase | Clase Cliente y programa ppal.

El método **esta_suspendido** verifica si el código del cliente se encuentra en la lista suspendidos, que es un **atributo de clase**. Dentro del cuerpo principal del programa se instancian cuatro clientes, y se suspenden dos de ellos mediante el método correspondiente:

Clase Cliente (Parte I) y programa ppal:

```
def esta_suspendido(self):
    if self.codigo in Cliente.suspendidos:
        print("Esta suspendido")
    else:
        print("No esta suspendido")
        print("_"*20)

# Programa principal:
cliente1 = Cliente(1,"Juan")
cliente2 = Cliente(2,"Ana")
cliente3 = Cliente(3,"Diego")
cliente4 = Cliente(4,"Pedro")
cliente3.suspender()
cliente4.suspender()
)
```

Atributos de clase | Ejecución

Si en este punto invocamos el método imprimir de cada instancia, vemos:

Programa principal

```
# Programa principal:
cliente1 = Cliente(1,"Juan")
cliente2 = Cliente(2,"Ana")
cliente3 = Cliente(3,"Diego")
cliente4 = Cliente(4,"Pedro")
cliente3.suspender()
cliente4.suspender()
cliente1.imprimir()
cliente2.imprimir()
cliente3.imprimir()
cliente4.imprimir()
```

Terminal

```
Codigo: 1
Nombre: Juan
No esta suspendido
_____________________
Codigo: 2
Nombre: Ana
No esta suspendido
_____________________
Codigo: 3
Nombre: Diego
Esta suspendido
_____________________
Codigo: 4
Nombre: Pedro
Esta suspendido
_____________________
```

Es importante recordar que todos los objetos acceden a una única lista llamada suspendidos gracias a que se definió como un atributo de clase.

Podemos ver el contenido de la lista con:

Programa principal

```
print(Cliente.suspendidos)
```

Terminal

```
[3, 4]
```

Composición

Cada clase que definimos es un nuevo tipo de datos, con sus atributos y métodos. Y al igual que los demás tipos de datos, los objetos instanciados a partir de ellas se pueden utilizar como parte de colecciones o incluso ser parte de otras clases.

La **composición** es uno de los conceptos fundamentales de la POO. Tiene lugar cuando una clase (clase “que tiene”) hace referencia a uno o más objetos de otras clases (clases “que son parte de”), como una variable de instancia.
+info

Al usar el nombre de la clase o al crear el objeto, se accede a los miembros de una clase dentro de otra clase. La composición permite crear tipos complejos mediante la combinación de objetos de diferentes clases.

Composición

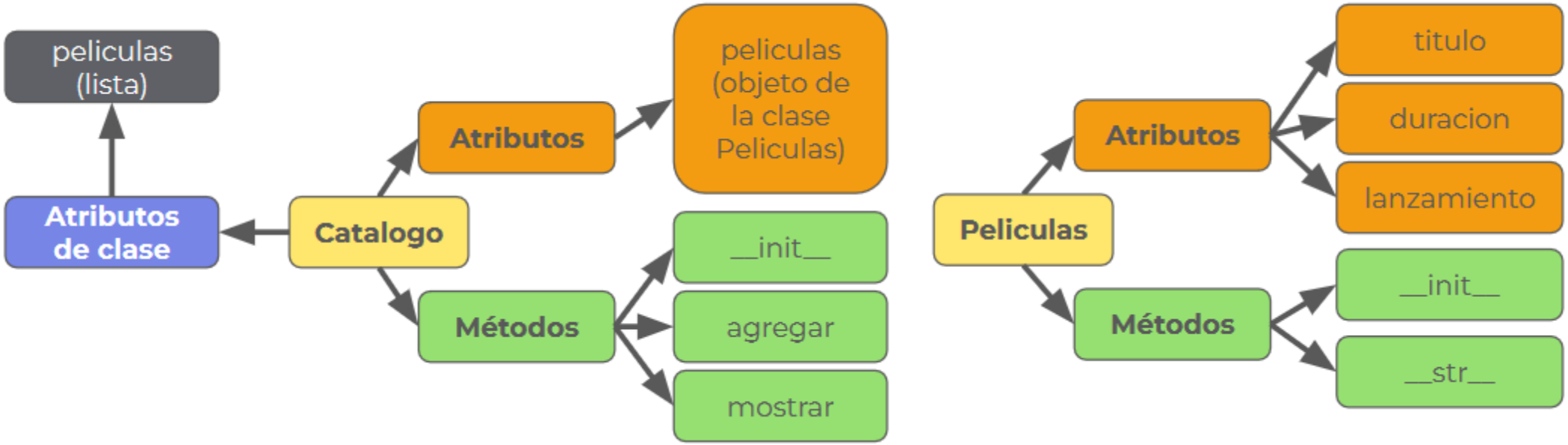
Mediante la composición solo se puede utilizar la otra clase (la “que es parte de”), no se puede modificar ni extender la funcionalidad de la misma. No proporciona funciones adicionales. Así, cuando se necesita usar otra clase sin ninguna modificación, se recomienda la composición

Enunciado 3:

- ✓ Define una clase *Película* que gestione datos de películas
- ✓ La clase *Película* tiene los atributos *título*, *duración* y *lanzamiento*, e implementa __*str*__ para mostrar sus datos.
- ✓ Define la clase *Catalogo*, que administra una lista de *películas*. Estas películas son objetos de la clase Película. Debe implementar los métodos *mostrar* y *agregar películas*.

Composición

Se implementan dos clases, que llamaremos **Película** (“es parte de”) y **Catalogo** (clase “que tiene”) respectivamente



Composición | Clase Pelicula

Definición de la clase **Película**. Esta es la clase que, en la composición, se comporta como **"clase que es parte de",** ya que sus instancias serán parte de la clase **Catalogo**.

Clase Película:

```
class Pelicula:
# Constructor de clase
    def __init__(self, titulo, duración, lanzamiento):
        self.titulo = titulo
        self.duracion = duración
        self.lanzamiento = lanzamiento
        print(f'Se ha creado la película: {self.titulo}')
    def __str__(self):
        return f'{self.titulo} ({self.lanzamiento})
```

Composición | Clase Catalogo

Definición de la clase **Catalogo**. En la composición se comporta como **“clase que tiene”**, ya uno de sus atributos de instancia es un objeto de la clase **Película**.

Clase Catálogo:

```
class Catalogo:
    peliculas = [] # Lista de objetos de la clase Película
    # Constructor de clase
    def __init__(self, películas=[]):
        Catalogo.películas = películas

    def agregar(self, p): # p es un objeto Pelicula
        Catalogo.peliculas.append(p)

    def mostrar(self):
        for p in Catalogo. Películas:
        print(p) # Print toma por defecto str(p)
```

Composición | Programa principal

Definición de la clase **Catalogo**. En la composición se comporta como "**clase que tiene**", ya uno de sus atributos de instancia es un objeto de la clase **Pelicula**.

Programa principal

```
# Instanciamos una película
peli1 = Película("El Padrino", 175, 1972)
# Instanciamos un catálogo que contiene una película
c = Catalogo([peli1])
c.mostrar()
# Añadimos una nueva película al catálogo:
c.agregar(Película("El Padrino: Parte 2", 202, 1974))
c.mostrar()
```

Terminal

```
Se ha creado la película: El Padrino
El Padrino (1972)
Se ha creado la película: El Padrino: Parte 2
El Padrino (1972)
El Padrino: Parte 2 (1974)
```

Encapsulación

Encapsulación se refiere al ocultamiento de los atributos o métodos de una clase al exterior, para que no se puedan acceder ni modificar desde fuera. Por defecto Python no oculta los atributos y métodos de una clase al exterior:

Programa principal

```
class Clase:
    atributo_clase = "Hola"

    def __init__(self, atributo_instancia):
        self.atributo_instancia = atributo_instancia

mi_clase = Clase("Que tal")
print(mi_clase.atributo_clase)
print(mi_clase.atributo_instancia)
```

Terminal

```
Hola
Que tal
```

Para ocultar métodos y atributos del exterior se usa doble guion bajo __ en el comienzo de su nombre. Esto hará que Python los interprete como “privados”.

Encapsulación | Atributos protegidos

Si el nombre de un atributo está precedido por un guión bajo, significa que es un **atributo protegido**. Solo puede ser accedido por esa clase y es una buena práctica para los atributos (o métodos) de uso interno, sobre todo en programas que recurren a muchas clases. De lo contrario, resulta, en realidad, innecesario.

Programa principal

```
class Vehiculo:
    def __init__(self, marca, color, placa):
        self._marca = marca
        self._color = color
        self._placa = placa

coche = Vehiculo("Ford", "Rojo", "AB123CD")
print(coche.marca)
```

Terminal

```
Traceback (most recent call last):
File "prueba.py", line 13, in <module>
print(coche.marca)
AttributeError: 'Vehiculo' object has no
attribute
'marca'. Did you mean: '_marca'?
```

Encapsulación

Programa principal

```
class Clase:
    atributo_clase = "Hola" # Accesible desde el exterior
    __atributo_clase = "Hola" # No accesible

    def __mi_metodo(self): # No accesible desde el exterior
        print("Que tal")
        self.__variable = 0

    def metodo_normal(self): # Accesible desde el exterior
        self.__mi_metodo()

mi_clase = Clase()
#mi_clase.__atributo_clase #AttributeError (atributo no accesible)
#mi_clase.__mi_metodo() #AttributeError (método no accesible)
print(mi_clase.atributo_clase)
mi_clase.metodo_normal()
```

Terminal

```
Hola
Que tal
```

Encapsulación

En el ejemplo se ve como los atributos y métodos de la clase **Clase** cuyos nombres comienzan con __ no son “visibles” desde el exterior de la clase.
En efecto, __atributo_clase y __mi_metodo() están “**encapsulados**” en la Clase.
Se puede acceder y modificar desde su interior, pero cuando intentamos hacerlo desde el exterior Python arroja un error explicando que no es posible.
Por ejemplo, en el código anterior al intentar usar __atributo_clase:

Terminal

```
Traceback (most recent call last):
File "prueba.py", line 13, in <module>
mi_clase.__atributo_clase
AttributeError: 'Clase' object has no attribute '__atributo_clase' Did you mean:
'atributo_clase'?
```

Encapsulación

Entonces, los atributos de una clase pueden presentar tres niveles de privacidad:

- ✓ **Públicos**: Creando un objeto de dicha clase y usando la sintaxis del punto, podemos acceder y modificar desde otra clase cualquier atributo.
- ✓ **Protegidos**: Un atributo solo puede ser accedido y modificado por la clase en sí misma (y sus clases hijas, como veremos). Para definir un atributo como protegido, se declara con un “guión bajo”.
- ✓ **Privados**: Pueden ser accedidos únicamente desde la clase donde fueron definidos. Su nombre empieza con “dobles guiones bajo”.

Los últimos necesitan implementar *getters* y *setters* para implementar el acceso a ellos.

Encapsulación | Decoradores - setters

Un **setter** permite crear métodos que permiten modificar el valor de un atributo privado.
En Python se declaran escribiendo antes del método un **decorador** con su nombre seguido de **.setter**.
Junto con el decorador @property permiten leer y escribir los atributos privados de manera segura.

Getters y setters:

```
class Bebidas:

    def __init__(self):
        self.__bebida = 'Naranja'

    @property
    def favorita(self):
        return f"La bebida preferida es: {self.__bebida}"

    @favorita.setter
    def favorita(self, bebida):
        self.__bebida = bebida

# Programa principal
obj1 = Bebidas()
obj1.favorita = "Pomelo"
print(obj1.favorita) # Pomelo
```

Encapsulación | Decoradores - getters

Un getter es un mecanismo que permite acceder a un método o atributo privado. En Python se declaran creando un método con el decorador **@property**:

Ejemplo del decorador @property

```
class Bebidas:
    def __init__(self):
        self.__bebida = 'Naranja'

    @property
    def favorita(self):
        return f"La bebida preferida es: {self.__bebida}"

obj1 = Bebidas()
print(obj1.favorita)
```

Terminal

```
La bebida
preferida es:
Naranja
```

Colaboración y encapsulación

Para finalizar, vamos a analizar un nuevo ejemplo:

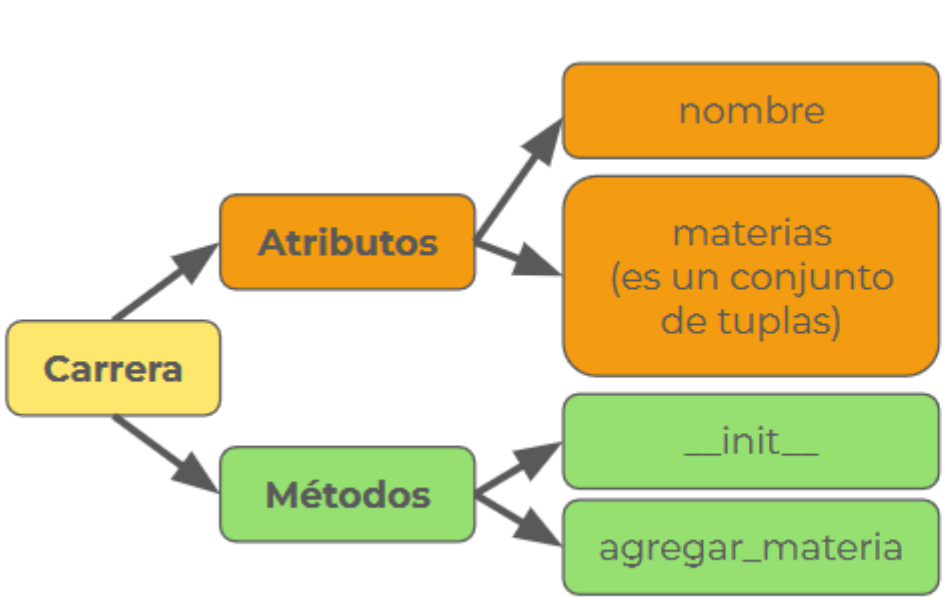
Enunciado 4:

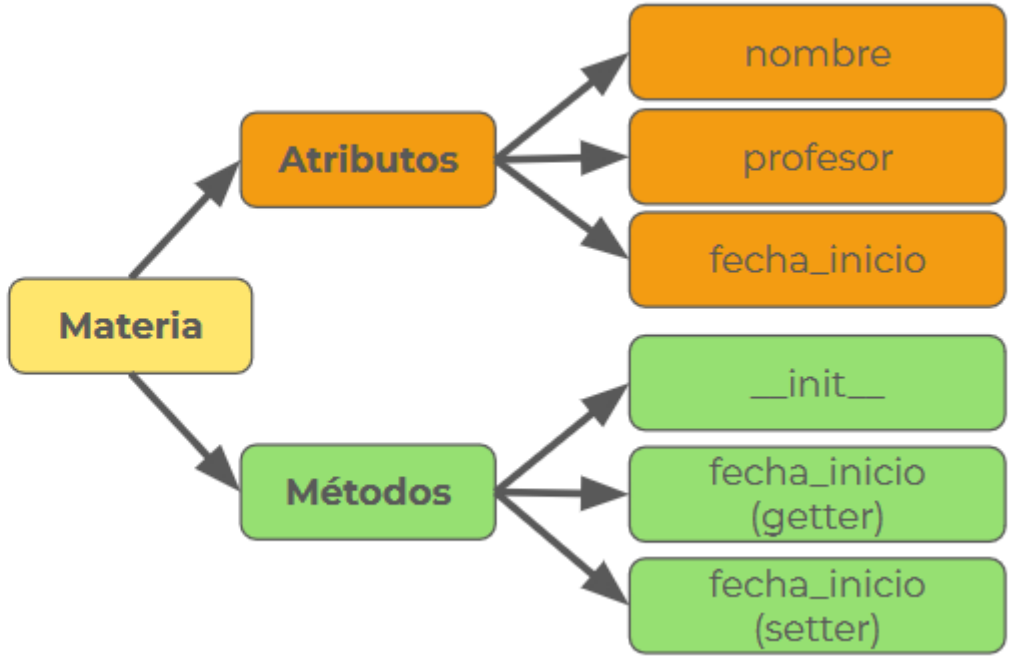
- ✓ *Crear una clase Carrera, que tenga un método constructor y otro que permita agregar materias a la carrera.*
- ✓ *Crear otra clase, Materia, que tenga los atributos privados nombre, profesor y fecha_inicio. Proporcionar un método para acceder a la fecha_inicio.*

Del enunciado se deduce que necesitamos objetos de dos clases: **Carrera** y **Materia**. Para cada una de estas entidades necesitamos crear una clase, con sus atributos y métodos.

Colaboración y encapsulación

Identificadas las clases, que llamaremos Carrera y Materia respectivamente, definimos qué atributos y métodos necesitamos implementar en cada una.





Colaboración y encapsulación

Clase Materia

```
class Materia:
    def __init__(self, nombre, profesor, fecha):
        self.nombre = nombre
        self.profesor = profesor
        #Fecha no puede ser anterior a 2006
        self.fecha_inicio = fecha

    @property
    def fecha_inicio(self):
        #print("Estoy obteniendo la fecha de
inicio")
        return self._fecha_inicio

    @fecha_inicio.setter
    def fecha_inicio(self, fecha):
        # print("Estoy seteando la fecha de inicio")
        if fecha < 2006:
            self._fecha_inicio = 2006
        else:
            self._fecha_inicio = fecha
```

Clase Carrera

```
class Carrera:

    def __init__(self, nombre):
        self.nombre = nombre
        #Contendrá tuplas (código, materia)
        self.materias = {}

# Este método agrega materias a la
# colección de materias
    def agregar_materia(self, materia, codigo):
        self.materias[codigo] = materia
```

Colaboración y encapsulación

Clase Materia

```
#Creamos una carrera y tres materias
carrera1 = Carrera("Ingeniería en Sistemas")
algebra = Materia("Algebra", "Juan Quinteros", 2004)
fisica = Materia("Física", "Pedro Perez", 2012)
programacion = Materia("Programación", "Lorena Ríos",2022)
#Agregamos las materias a la carrera:
carrera1.agregar_materia(201,algebra)
carrera1.agregar_materia(202,fisica)
carrera1.agregar_materia(203,programacion)
# Veamos la fecha de inicio de dictado de algunas materias:
print(f"La fecha de inicio de {algebra.nombre} es
{algebra.fecha_inicio}")
print(f"La fecha de inicio de {fisica.nombre} es
{fisica.fecha_inicio}")
print(f"La fecha de inicio de {programacion.nombre} es
{programacion.fecha_inicio}")
```

Terminal

```
La fecha de inicio de Álgebra es 2006

La fecha de inicio de Física es 2012

La fecha de inicio de Programación es
2022
```

Ejercicios

1. Desarrollar un programa que cargue los datos de un triángulo. Implementar una clase con los métodos para inicializar los atributos, imprimir el valor del lado con un tamaño mayor y el tipo de triángulo que es (equilátero, isósceles o escaleno).
2. Realizar un programa en el cual se declaren dos valores enteros por teclado utilizando el método __init__. Calcular después la suma, resta, multiplicación y división. Utilizar un método para cada una e imprimir los resultados obtenidos. Llamar a la clase Calculadora.
3. Realizar una clase que administre una agenda. Se debe almacenar para cada contacto el nombre, el teléfono y el email. Además, deberá mostrar un menú con las siguientes opciones
    - Añadir contacto
    - Lista de contactos
    - Buscar contacto
    - Editar contacto
    - Cerrar agenda
4. Realizar un programa que modele una cuenta corriente. Cada cliente se identifica con un nro. de cuenta, CBU, nombre y monto total. Las operaciones permitidas para la cuenta son: ver saldo, ver datos de la cuenta, realizar extracciones y realizar depósitos
5. Modelar un sanatorio donde los pacientes tienen los siguientes atributos: dni, obra social, nombre, apellido, número habitación, estado (habitación común, terapia intermedia, terapia intensiva). Implementar métodos que permitan ver los datos de los pacientes, ver su estado y cambiar de estado

MATERIAL EXTRA

Artículos de interés

Material extra:

- ✓ Guía de la Programación Orientada a Objetos (POO), en Kinsta
- ✓ ¿Qué es el encapsulamiento en Python?, en pythones.net
- ✓ Programación Orientada a Objetos, en carmoreno.com

Videos:

- ✓ Objetos parte III, parte IV y parte V en Píldoras informáticas
- ✓ Clases y Objetos en Python, por yacklyon