Licenciatura en Ciencia de Datos

Algoritmos II

Repaso

¿Qué recuerdan del Paradigma Orientado a Objetos?

¿Qué son los objetos?

- Una **abstracción** de dato que captura: **Características + Comportamiento**
- Se **instancia** de una **Clase**
- Tienen un **estado** asociado
- Interactúan a través de **mensajes**
- Tienen una **representación interna** a través de los atributos de datos
- Tienen una **interfaz** para **interactuar** con otros objetos

Objetos en Python

¡¡¡TODO ES UN OBJETO EN PYTHON: clases, instancias de clases, funciones, módulos!!!

-Las **variables no** tienen asociado un **tipo de dato**, es un **lenguaje de tipado dinámico**

- **Consecuencia**: una **variable** es una **etiqueta** a una **referencia en memoria**. (dirección donde se aloja un objeto)

-Cuando se **genera una variable** a través de la **asignación =**, se **asocia** la **etiqueta** al **objeto**, y podemos accederlo mediante ella.

-Si la **variable** luego se **asigna a otro nuevo objeto** y **no** quedan **variables** que referencian al **objeto previo**, el **recolector de basura** se encargará de liberar de la memoria de ese objeto.

Identidad de Objetos

-Cada **objeto** tiene asociado un **identificador de identidad** que es **exclusivo** y lo distingue del resto de los objetos = **DIRECCIÓN DE MEMORIA**

-Se consulta con la operación **id()**

LA IDENTIDAD DE UN OBJETO NO PUEDE CAMBIAR DESPUÉS DE SER INSTANCIADO (una vez que se **crea un objeto en Python**, siempre tendrá la **misma dirección de memoria** mientras exista)

Ejemplo:

```
id(33)                          # 140721232205608
id('Esto es una cadena')        # 2365055278640
x = 5.5
id(x)                           # 2365048316752
```

Tipo de datos

-Se desprende de la **clase** desde la cual se instancia y **define** qué **operaciones** podemos realizar sobre este objeto

-Se puede consultar con la función **type().** El **valor devuelto** corresponde al **atributo especial __class__**

Ejemplo:

```
type('Esto es una cadena')   # <class 'str'>

type(33)                     # <class 'int'>
type([1,2])                  # <class 'list'>
type(len)                    # <class 'builtin_function_or_method'>
'Esto es una cadena'.__class__  # <class 'str'>
```

Clases

-Una clase es una “**plantilla”** que define los **atributos** (propiedades) y **métodos** (comportamientos) que un **objeto** puede tener.

-**TODO OBJETO ES UNA INSTANCIA DE UNA CLASE**

-Por ejemplo, una clase "Auto" puede tener atributos como "marca" y "modelo", y métodos como "arrancar" y "detener”

Sintaxis para definir una clase en Python:

```
class NombreClase:
    # Definición de atributos y métodos de la clase
```

Instanciación vs inicialización

INSTANCIACIÓN: proceso de **crear un objeto a partir de una clase**. Involucra dos **métodos especiales** en Python:

1. **Método `__new__()`**:
   - Responsable de la **CREACIÓN** de **una nueva instancia** de la clase.
   - Tarea principal: **reservar espacio en la memoria** para el nuevo objeto.
   - **Devuelve la instancia** del nuevo objeto.
2. **Método `__init__()`**:
   - Responsable de **INICIALIZAR** la **instancia** recién creada.
   - Se llama **después** de que la **instancia** se **creó**.
   - Se usa para **establecer el ESTADO INICIAL del objeto**, como asignar valores a sus propiedades.

¡¡¡AMBOS MÉTODOS CONFORMAN EL CONSTRUCTOR DE LA CLASE!!!

Cuando se **instancia** una clase, el **flujo** es:

- `__new__()` es llamado par**a crear y devolver una nueva instancia** del objeto.
- `__init__()` es llamado para **inicializar** esa instancia.

Ejemplo de definición de una clase con su constructor

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
```

- **__init__()** recibe como primer argumento la **instancia** (por **convención** se lo nombra siempre **self**), y luego los **argumentos** necesarios para **inicializar el objeto**.

-Los atributos self.nombre y self.edad son **propios de esta instancia**.

Ejemplo (cont)

-Para **instanciar** un objeto de la clase que definimos, llamamos al **nombre de la clase** seguido de **paréntesis** y los **argumentos definidos** en **__init__():**

```
juana = Persona("Juana", 23)
print(juana)                          # <__main__.Persona at 0x1d2bd5b1750>
Indica de qué clase se instanció (Persona) y su dirección de memoria
en valor hexadecimal.
print(hex(id(juana)))                 # 1d2bd5b1750
print(juana.nombre)                   # Juana
print(juana.edad)                     # 23
```

Miembros de instancia y miembros de clase

Los **miembros de una clase** pueden ser de **dos tipos**:

- miembros de **clase**
- miembros de **instancia**

Atributos

-**Atributos de clase**: son **compartidos por todas las instancias** de la clase.

- Se **definen fuera** de **cualquier método** de la clase
- Se **acceden** usando el **nombre de la clase.**
- Se pueden usar para **almacenar datos que son comunes a todas las instancias.**

-**Atributos de instancia**: son **específicos de cada objeto**

- Se **definen dentro** del método **__init__()** utilizando el parámetro **self**.
- **Cada instancia** de la clase tiene sus **propias copias** de los atributos de instancia.

Métodos de instancia

-Operan sobre **instancias** de la clase (**objetos**).

-**Acceden y modifican** los **atributos** del **objeto** específico.

-Se **definen dentro** de la **clase**

-Siempre **reciben al menos un parámetro, self,** que es una **referencia** a la **instancia actual** del objeto.

-**Uso**: Se utilizan para acceder y modificar los datos de la instancia.

Ejemplo método de instancia

```
class MiClase:
    def __init__(self, valor):
        self.valor = valor  # atributo de instancia

    def mostrar_valor(self):
        print(f'El valor es {self.valor}')

# Crear una instancia de la clase
obj = MiClase(10)

# Llamar a un método de instancia
obj.mostrar_valor()  # Salida: El valor es 10
```

Ejercicio

Implemente la clase Lamparita, que sirva para representar el estado de encendido de una lamparita (encendido o apagado). Defina, asimismo, dos métodos que permitan encender y apagar la luz de la lamparita y otro que indique en qué estado se encuentra. La lamparita inicialmente está apagada.

Ejercicio

Implemente una clase Monedero que permita gestionar la cantidad de dinero que una persona dispone en un momento dado. La clase deberá tener un constructor que permitirá crear un monedero con una cantidad de dinero inicial y deberá definir un método para meter dinero en el monedero, otro para sacarlo y finalmente, otro para consultar el disponible; solo podrá conocerse la cantidad de dinero del monedero a través de este último método. Por supuesto, no se podrá sacar más dinero del que haya en un momento dado en el monedero.

Métodos de clase

-**Operan** en la **clase** en lugar de en instancias de la clase.

-Pueden **acceder y modificar el estado de la clase** que **afecta a todas las instancias** de la clase.

-**Sólo** pueden **acceder a atributos de clase**, ya que el **primer parámetro** es la **clase**. Por convención se nombra **cls**

-Se definen utilizando el decorador **@classmethod** (lo vemos más adelante)

-**Uso**: **trabajar con datos comunes a todas las instancias de la clase**

Ejemplo (cont)

```
class Persona:
    contador_personas = 0

    def __init__(self, nombre, apellido):
        self.nombre = nombre
        self.apellido = apellido
        Persona.contador_personas += 1

    @classmethod
    def personas_creadas(cls):
        return cls.contador_personas

juana = Persona("Juana", "Lopez")
Persona.personas_creadas()          # 1
juana.personas_creadas()            # 1
```

Nota: una **instancia** puede acceder a **miembros de instancia** y **miembros de clase**

Métodos estáticos

-**No operan** ni sobre una **instancia** de la clase **ni** sobre la **clase** misma.

-**No reciben self** ni **cls** como primer parámetro.

-Son **funciones** que están asociadas con la clase para **propósitos organizacionales**.

-Se definen utilizando el decorador **@staticmethod.**

Ejemplo:

```
class MiClase:
    @staticmethod
    def metodo_estatico(x, y):
        return x + y
# Llamar a un método estático (notar que no estamos instanciando)
resultado = MiClase.metodo_estatico(5, 7)
print(resultado)  # Salida: 12
```

Métodos estáticos (cont)

-Al igual que con los métodos de clase, **no es necesario instanciar un objeto** de para su **invocarlos**.

-Son **buenos candidatos** para modelar **comportamiento de ayuda (helpers)**.

Ejercicio

Una cadena de restaurantes llamada "Delicioso Sabor" desea implementar un sistema de gestión de pedidos automatizado para mejorar la eficiencia en el manejo de sus ventas y optimizar el control de su inventario.

Requerimientos del Sistema:

1. **Clase Producto:**
   - Cada producto en el menú del restaurante debe ser representado por la clase `Producto`.
   - Los productos deben tener un nombre, precio unitario y cantidad inicial en stock.
   - Se debe poder actualizar la cantidad en stock de cada producto conforme se realicen pedidos.
2. **Clase Pedido:**
   - La clase `Pedido` debe registrar los detalles de cada pedido realizado por los clientes.
   - Cada pedido debe contener un número único de identificación, una lista de productos solicitados y su estado actual.
   - Se debe calcular el costo total del pedido, aplicando un descuento global del 10% por defecto.
   - Se debe poder actualizar el estado de los pedidos a medida que progresan en su preparación y entrega.

Ejercicio

La cadena de restaurantes "Delicias del Mar" desea mejorar su sistema de gestión de sus sucursales i

Requerimientos Funcionales:

1. **Clase Restaurante:**
    - Definir una clase llamada `Restaurante` que contenga como atributos: nombre del restaurante, ciudad donde se encuentra y número de empleados.
    - Implementar un método`obtener_numero_sucursales()` que retorne el número total de sucursales de la cadena.
    - Implementar un método `calcular_costo_operativo(empleado_promedio)` que calcule el costo mensual de operación de una sucursal. Considerar un salario promedio mensual por empleado de $2000.

Crear instancias de la clase `Restaurante` para representar diferentes sucursales con sus respectivos nombres, ciudades y número de empleados.

Usar el método `obtener_numero_sucursales()` para obtener y mostrar el número total de sucursales de la cadena.

Usar el método `calcular_costo_operativo()` para calcular y mostrar el costo mensual de operación de cada sucursal creada.

Helpers o funciones auxiliares

-Son **funciones o métodos** diseñados para realizar **tareas comunes, repetitivas o de soporte dentro del código.**

-Estas funciones son generalmente **pequeñas, específicas y encapsulan** una lógica particular que **puede ser reutilizada en diferentes partes del programa.**

-Permiten **actualizar y mantener el código** (Si se necesita **cambiar un comportamiento**, solo es necesario **modificar el helper**)

Ejemplo helper

```
class Persona:
    def __init__(self, nombre, apellido):
        self.nombre = nombre
        self.apellido = apellido

    def obtener_nombre_completo(self):
        return self._formatear_nombre_completo()

    def _formatear_nombre_completo(self):
        return f'{self.nombre} {self.apellido}'

# Uso de la clase y sus métodos
persona = Persona('Juan', 'Pérez')
print(persona.obtener_nombre_completo())
 # Salida: Juan Pérez
```

Método obtener_nombre_completo: Método que devuelve el nombre completo de la persona.

Método _formatear_nombre_completo: Método helper que se encarga de formatear y combinar el nombre y el apellido en una sola cadena.

Ejemplo de Helper como Función Interna

```python
class Persona:
    def __init__(self, nombre, apellido):
        self.nombre = nombre
        self.apellido = apellido

    def obtener_nombre_completo(self):
        # Función interna que actúa como helper
        def formatear_nombre_completo():
            return f'{self.nombre} {self.apellido}'

        return formatear_nombre_completo()

# Uso de la clase y sus métodos
persona = Persona('Juan', 'Pérez')
print(persona.obtener_nombre_completo())
# Salida: Juan Pérez
```

-Método obtener_nombre_completo: Método público que devuelve el nombre completo de la persona. Dentro de este método, se define una **función interna formatear_nombre_completo**: tiene **acceso a los atributos** self.nombre y self.apellido.

Este enfoque es **útil** cuando el **helper** es **simple** y su **uso** está **limitado a un solo método**, ayudando a **mantener el contexto y la lógica** dentro del mismo **alcance**

Métodos especiales (dunder methods o métodos mágicos)

-Son invocados **implícitamente** para ejecutar cierta **operación** sobre un tipo de dato.

-**Nombre**: comienzan y terminan con **doble guión bajo __**

-Se puede **definir un comportamiento diferente** para nuestras clases respecto a los operadores del lenguaje (**sobrecarga**)

Algunos métodos especiales (cont)

1) **__init__(self, ...)**
2) **__str__(self)**: Devuelve una **representación en cadena** "informal" o **"legible por humanos"** del objeto.

   Uso Principal: Es llamado por las funciones **str() y print()**.

   Implementación Típica: devolver una **cadena de texto** que **describa el objeto de manera clara y amigable**.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __str__(self):
        return f'Nombre: {self.nombre}, Edad: {self.edad}'

persona = Persona('Juan', 30)
print(str(persona))  # Salida: Nombre: Juan, Edad: 30
print(persona)       # Salida: Nombre: Juan, Edad: 30
```

Métodos especiales (cont)

3) __repr__(self): Devuelve una **representación en cadena** "oficial" o **"de desarrollador"** del objeto.

Objetivo: ser **preciso y sin ambigüedades,** proporcionando **suficiente información** para que un **desarrollador** pueda **recrear el objeto.**

Uso Principal: Es llamado por la función **repr()**, y cuando se evalúa el objeto en un **shell de Python**

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __repr__(self):
        return f'Persona(nombre={self.nombre!r}, edad={self.edad!r})'

persona = Persona('Juan', 30)
print(repr(persona))  # Salida: Persona(nombre='Juan', edad=30)
```

Métodos especiales (cont)

A veces se pueden usar ambos combinados:

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __str__(self):
        return f'Nombre: {self.nombre}, Edad: {self.edad}'

    def __repr__(self):
        return f'Persona(nombre={self.nombre!r}, edad={self.edad!r})'

persona = Persona('Juan', 30)
print(str(persona))    # Salida: Nombre: Juan, Edad: 30
print(repr(persona))   # Salida: Persona(nombre='Juan', edad=30)
print(persona)         # Salida: Nombre: Juan, Edad: 30
```

Métodos especiales (cont)

4)__len__(self): Devuelve el **número de elementos** de un objeto. Se suele implementar para **colecciones.** Se invoca cuando se solicita la longitud con el comando **len()**.

```
class MiLista:
    def __init__(self, elementos):
        self.elementos = elementos

    def __len__(self):
        return len(self.elementos)

mi_lista = MiLista([1, 2, 3])
print(len(mi_lista))  # Salida: 3
```

Métodos especiales (cont)

5) __getitem__(self, key): Permite el acceso a elementos mediante el **operador []**.

```
class MiLista:
    def __init__(self, elementos):
        self.elementos = elementos

    def __getitem__(self, index):
        return self.elementos[index]

mi_lista = MiLista([1, 2, 3])
print(mi_lista[1])  # Salida: 2
```

Métodos especiales (cont)

6) __setitem__(self, key, value): Permite la **asignación de valores** mediante el **operador []**.

7) __eq__(self, otro): Define la **comparación de igualdad entre objetos**. Es invocado cuando se utiliza el **operador ==**.

```
juana = Persona("juana", 23)
juana2 = Persona("juana", 23)
juana == juana2      # False
```

Métodos especiales (cont)

Implementamos **nuestro propio comparador de igualdad** sobreescribiendo __eq__():

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def __eq__(self, otro):
        return isinstance(otro, Persona) and self.nombre == otro.nombre and self.edad
== otro.edad

juana = Persona("juana", 23)
juana2 = Persona("juana", 23)

juana == juana2     # True
```

Métodos especiales

8) __hash__: Retorna un número entero que **identifica al objeto.** Si **dos objetos** son considerados **iguales** según el método __eq__, también deben tener el **mismo valor hash.** La documentación oficial recomienda **devolver el valor hash de la tupla** con los **atributos** utilizados en el **__eq__()**.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad
     def __eq__(self, otra):
        if isinstance(otra, Persona):
            return self.nombre == otra.nombre and self.edad == otra.edad
        return False
    def __hash__(self):
        return hash((self.nombre, self.edad))
```

Métodos especiales

```
# Creación de instancias de Persona
persona1 = Persona('Juan', 30)
persona2 = Persona('Juan', 30)
persona3 = Persona('Ana', 25)
# Verificación de valores hash
print(hash(persona1))  # Salida: un entero, por ejemplo 876456
print(hash(persona2))  # Salida: el mismo entero que persona1,
porque son iguales
print(hash(persona3))  # Salida: un entero diferente
```

Métodos especiales (cont)

9) __iter__(self) y __next__(self): Permiten que un objeto sea **iterable.**

```
class Contador:
    def __init__(self, limite):
        self.limite = limite
        self.actual = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.actual < self.limite:
            self.actual += 1
            return self.actual
        else:
            raise StopIteration

contador = Contador(5)
for num in contador:
    print(num)  # Salida: 1 2 3 4 5
```

Métodos especiales

10) __call__: permite que las **instancias** de una clase sean **llamadas** como si fueran **funciones.**

```
class Saludo:
    def __init__(self, mensaje):
        self.mensaje = mensaje
     def __call__(self, nombre):
        return f'{self.mensaje}, {nombre}!'
# Crear una instancia de la clase Saludo
saludo = Saludo('Hola')
# Llamar a la instancia como si fuera una función
print(saludo('Juan'))  # Salida: Hola, Juan!
print(saludo('Ana'))   # Salida: Hola, Ana!
```

Al llamar a **saludo('Juan')**, Python invoca el **método __call__** de la **instancia saludo**, pasando '**Juan**' **como argumento** al método.
La salida es 'Hola, Juan!'

¡¡¡Ver documentación para muchos más métodos especiales!!!

Ejercicio

Implemente la clase Punto (pares de coordenadas de tipo float x, y). Defina constructores y métodos para asignar valores a las coordenadas de los puntos, retornar el valor de cada coordenada, y sumar dos puntos, retornando su resultado. Definir un método booleano de igualdad entre dos puntos.

Ejercicio

Implemente la clase Fecha, que permita representar una terna de día, mes y año, todos de tipo entero. Programar un método que determine si una fecha es mayor a otra. Programar la sobrecarga del método __str__ y __gt__(operador mayor).

Accesibilidad a miembros de clase

¡¡¡En Python NO HAY UN MECANISMO que permita MODIFICAR LA VISIBILIDAD de los elementos de una clase!!!

-Se distingue entre miembros públicos y no públicos sólo mediante **UNA CONVENCIÓN DE NOMBRE**

-Un miembro de una clase con un nombre que **comienza con _,** se asume es no público. Python **no restringe el acceso desde afuera**, es una **señal** a quien consume la clase que no debe accederlo directamente.

-También puede nombrarse comenzando con **dos guiones bajos**. Python **cambia su nombre internamente** para **incluir el nombre de la clase**.

Ejemplo: si la clase **MiClase** tiene un atributo **__privado**, Python internamente lo renombrará a **_MiClase__privado**. → **CUIDADO CON ESTO**

Ejemplo

```
class Ejemplo1:
    def __init__(self, valor):
        self._protegido = valor

    def mostrar(self):
        return f'Valor protegido: {self._protegido}'

objeto1 = Ejemplo1(10)
print(objeto1.mostrar())  # Salida: Valor protegido: 10

# Acceder directamente al atributo "protegido"
print(objeto1._protegido)  # Salida: 10 NO TIRA ERROR!
```

Ejemplo 2

```
class Ejemplo2:
    def __init__(self, valor):
        self.__privado = valor
    def mostrar(self):
        return f'Valor privado: {self.__privado}'
objeto2 = Ejemplo2(20)
print(objeto2.mostrar())  # Salida: Valor privado: 20
# Intentar acceder directamente al atributo privado (esto fallará)
try:
    print(objeto2.__privado)
except AttributeError as e:
    print(e)  # Salida: 'Ejemplo2' object has no attribute '__privado'
# Acceder al atributo privado utilizando name mangling
print(objeto2._Ejemplo2__privado)  # Salida: 20, FUNCIONA!!!!
```

Atributos -> Propiedades

-El **encapsulamiento** en POO es un principio que permite **proteger** los atributos y métodos de una clase para evitar su acceso y **modificación directa desde fuera**

- En **Python**, una forma respecto del **acceso y la modificación** de los **atributos** de una clase es mediante el uso de **propiedades:** permiten **definir** métodos **"getter" y "setter"**.

Ejemplo

```
class Persona:
    def __init__(self, nombre):
        self._nombre = nombre  # Atributo "privado"
    @property
    def nombre(self):
        """Getter para el atributo nombre"""
        return self._nombre
    @nombre.setter
    def nombre(self, valor):
        """Setter para el atributo nombre"""
            self._nombre = valor
 # Crear una instancia de la clase Persona
persona = Persona('Juan')
#Acceder a los atributos a través de getters
print(persona.nombre)  # Salida: Juan
# Modificar los atributos a través de setters
persona.nombre = 'Ana'
print(persona.nombre)  # Salida: Ana
```

Herencia

-Permite la creación de **clases nuevas** basadas en clases **existentes**

-Se representa como una **jerarquía de abstracciones**

-Ej: clase para representar una **Cabaña** y otra clase para **Departamento**. definir una clase **Vivienda** que modele una **abstracción mayor.**Tenemos una relación de herencia entre **Vivienda-Cabaña** y **Vivienda-Departamento**.

-**Tipos**

- **Simple:** una clase sólo puede heredar de **una única superclase**
- **Múltiple**

Herencia en Python

-**Object** es la **CLASE BASE más general** de la cual **todas las demás clases derivan**.

-Define **varios MÉTODOS COMUNES** que están **disponibles para todas las instancias de las clases**. Algunos **son**:

1) __init__
2) __str__
3) __repr__
4) __eq__
5) __hash__

Herencia

-Sintaxis en Python:

```
class Persona:
    pass
class Estudiante(Persona):
    pass
juana = Estudiante()
isinstance(juana, Estudiante)    # True
isinstance(juana, Persona)       # True
isinstance(juana, object)        # True
```

-**Persona** es una **superclase** que hereda de **object**, mientras que **Estudiante** es una **subclase** también de **Persona** y **object**

Herencia (cont)

¡¡¡EN PYTHON HAY HERENCIA MÚLTIPLE!!!

```
class UserCampus(Estudiante, Docente):
    pass
```

Ejemplo de sintaxis de herencia simple:

```
class Persona:
    def __init__(self, nombre, apellido):
        self.nombre = nombre
        self.apellido = apellido

 class Estudiante(Persona):
    def __init__(self, nombre, apellido, matricula):
        super().__init__(nombre, apellido)  # Invoca inicializador de Persona
        self.matricula = matricula
juana = Estudiante("Juana", "Lopez", 1234)
```

Sobreescritura

-**Una SUBCLASE REDEFINE un método de su superclase** para **cambiar** su **comportamiento**.

```
class Animal:
    def hacer_sonido(self):
        print("Algún sonido genérico")

class Perro(Animal):
    def hacer_sonido(self):
        print("Guau guau")

perro = Perro()
perro.hacer_sonido()  # Salida: Guau guau
```

Polimorfismo

-**Capacidad** de **OBJETOS DE DIFERENTES CLASES** de responder al **MISMO MENSAJE DIFERENTE**.

- **Diferentes clases** comparten el **mismo nombre de método** pero lo **implementen** de **manera específica**+

- Es un **componente** esencial de la **herencia** que permite escribir código más **flexible** y **reutilizable**.

-Se apoya en el concepto de **sobreescritura**.

```
class Gato:
    def hacer_sonido(self):
        print("Miau")
class Perro:
    def hacer_sonido(self):
        print("Guau guau")
def hacer_sonido_animal(animal):  #NOTAR QUE LA FUNCIÓN ESTÁ FUERA de las clases
    animal.hacer_sonido()
gato = Gato()
perro = Perro()
hacer_sonido_animal(gato)  # Salida: Miau
hacer_sonido_animal(perro)  # Salida: Guau guau
```

Ejercicio

a) Crear una clase *Vehiculo* con los siguientes atributos y métodos:

- Atributos:
  - marca (String)
  - modelo (String)
  - precioBase (double).
- Métodos:
  - Un constructor que acepte la marca, modelo y precio base del vehículo.
  - Un método *calcularCostoAlquiler(int dias)* que calcule el costo de alquiler del vehículo durante el número de días especificado. El costo se calcula como precioBase * dias.

b) Crear dos subclases *Auto* y *Moto*, que hereden de la clase *Vehiculo*. Las subclases deben incluir un constructor que llame al constructor de la superclase y también deben sobrescribir el método *calcularCostoAlquiler(int dias)* de la siguiente manera:

- Para Auto, el costo de alquiler se calcula incrementando un 20% el costo común.
- Para Moto, el costo de alquiler se calcula con un descuento del 15% respecto al vehículo.

Ejercicio

Una empresa ferroviaria administra viajes en tren entre dos estaciones terminales de su red.
Un viaje tiene asociado un trayecto (desde una estación terminal de origen a una de destino, con una distancia determinada y una cantidad de estaciones), una cierta cantidad de vagones y una capacidad máxima de pasajeros.
También posee qué tipo de viaje corresponde en relación a sus características técnicas, si es un viaje con tecnología diesel, si es eléctrico o si es de alta velocidad (esto es independiente del trayecto recorrido).

- Viaje diesel: El tiempo de demora promedio -en minutos- es la distancia en kilómetros multiplicada por la cantidad de estaciones dividido 2 sumada a la cantidad de estaciones y de pasajeros dividido 10.
- Viaje eléctrico: El tiempo de demora promedio -en minutos- es la distancia en kilómetros multiplicada por la cantidad de estaciones dividido 2.
- Viaje de alta velocidad: El tiempo de demora promedio -en minutos-es la distancia en kilómetros dividido 10.

Definir dentro de la clase *Viaje* el método *tiempoDeDemora*, que retorne la cantidad de minutos que tarda en efectuar su recorrido con las siguientes variantes:
a) Especializando la clase *Viaje* en función del tipo de viaje.
b) Sin especializar la clase *Viaje*, relacionándola con la clase *TipoDeViaje*, que está especializada por cada tipo de viaje.

Ejercicio

Una editorial de libros y discos desea crear fichas que almacenen el título y el precio de cada publicación. Definir la correspondiente clase Publicacion que implemente los datos anteriores. Derive dos clases, una llamada Libro, que contenga para cada libro el número de páginas, año de publicación y precio, y la clase Disco, con la duración en minutos y precio. Programar una aplicación que pruebe las clases.

Method Resolution Order (MRO)

```
class A:
    def metodo1(self):
        return 'Metodo1 de A'
class B(A):
    def metodo1(self):
        return 'Metodo1 de B'

objeto_b = B()
objeto_b.metodo1()# Metodo1 de B
```

-El método de clase **mro()** nos ofrece información del orden:

```
A.mro()      # [__main__.A, object]
B.mro()      # [__main__.B, __main__.A,
object]
```

Clases abstractas

-Debemos heredar de la clase **abc.ABC**

- En Python **NO** tenemos un **MECANISMO** para **EVITAR INSTANCIARLAS PERO SÍ FORZAR** debemos **AGREGANDO al menos un MÉTODO ABSTRACTO** con el decorador **@abstractmethod**

```
from abc import ABC, abstractmethod
class Vehiculo(ABC):
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
    @abstractmethod
    def mostrar_info(self):
        raise NotImplementedError
vehiculo = Vehiculo("Toyota", "Corolla")     # TypeError: Can't instantiate abstract
class Vehiculo with abstract method mostrar_info
```

Abstracción

-Concepto:Capacidad de **definir clases que contienen métodos sin implementación completa**, **permitiendo** que las **subclases** las implementen

-Ventajas:

1)Facilita la creación de **interfaces comunes**.

2) **Oculta** detalles de **implementación**.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        print("Guau guau")

perro = Perro()
perro.hacer_sonido()  # Salida: Guau guau
```

Ejercicio

Definir la clase Automovil, que puede subclasificarse en AutoMediano o Camion. Los autos medianos son capaces de estar habilitados luego de la adquisición de un permiso en una fecha dada. Los camiones también podrán estar habilitados luego de la adquisición de un permiso, pero éste sólo podrá expedirse con la debida autorización previa de la concesionaria donde fue adquirido. Las concesionarias de camiones verifican ciertas características del camión para poder registrar al mismo. Este dato también es registrado dentro de la misma concesionaria.

Ejercicio

Definir la clase ExpresionAritmetica, que permita representar constantes numéricas enteras, operaciones binarias de suma y producto, y operaciones unarias de negación aritmética, incrementar y decrementar. Toda expresión deberá poder evaluar el resultado de la expresión, retornando el valor entero resultante. Definir tantas subclases como posibilidades existan de armar expresiones aritméticas.

Funciones internas

-Son **funciones definidas dentro** de **otras funciones**

-Pueden acceder a las **variables locales** y los **parámetros** de la **función externa**.

-Permiten **modularizar + encapsular**

```
def funcion_externa():
    def funcion_interna():
        return "Esta es una funcion interna."

    return funcion_interna()

print(funcion_externa())  # "Esta es una funcion interna."
print(funcion_interna())  # NameError: name 'funcion_interna' is not defined
```

Clausura

-Permite a una función **"recordar"** el **ámbito** en el que fue definida, manteniendo el **acceso** a las **variables locales de la función externa**

-Un caso común donde se utiliza es en la **currificación** (paradigma funcional)

```
def crear_clausura(x):

    def clausura(y):
        # La función interna puede acceder a 'x', que es una variable de la función externa.
        return x + y  # Retorna la suma de 'x' y 'y'.
    # Retornamos la función interna 'clausura'.
    # 'clausura' "recuerda" el valor de 'x' incluso después de que 'crear_clausura' haya terminado.
    return clausura

# Crear una instancia de la clausura.
# Llamamos a 'crear_clausura' con el valor 5, lo que devuelve la función 'clausura' con 'x' fijado en 5.
sumar_cinco = crear_clausura(5)
# Usar la clausura.
# Ahora, 'sumar_cinco' es una función que espera un argumento 'y' y sumará ese 'y' con el 'x' que recuerda
print(sumar_cinco(10))  # Salida: 15. Esto calcula 5 + 10 y devuelve 15.
print(sumar_cinco(20))  # Salida: 25. Esto calcula 5 + 20 y devuelve 25.
```

Decorador

-**Función** que **recibe otra función** como argumento y **devuelve** una **nueva función** con un **comportamiento modificado** o **extendido**. **Se aplican** a las funciones con el **símbolo @**

```
def mi_decorador(func):

    # Definimos una función interna 'nueva_funcion' que envolverá a la función original 'func'
    def nueva_funcion(*args, **kwargs):
        #Podemos agregar código que queremos que se ejecute antes de llamar a 'func'
        print("Antes de ejecutar la función")
       # Llamamos a la función original 'func' con los argumentos que recibió
        resultado = func(*args, **kwargs)
        #Podemos agregar código que queremos que se ejecute después de llamar a 'func'
        print("Después de ejecutar la función")
        # Devolvemos el resultado de la función original 'func'
        return resultado
    # Retornamos la función interna 'nueva_funcion'
    return nueva_funcion
# Aplicamos el decorador
@mi_decorador #mismo nombre que el decorador
def saludar(nombre):
    print(f"Hola, {nombre}")
# Llamamos a la función decorada 'saludar'
saludar("Juan") #Salida: Antes de ejecutar la función Hola, Juan Después de ejecutar la función
```

Main en Python

-¡¡¡En Python, no existe una función main estricta como en Java!!!

-Se sigue una **CONVENCIÓN** similar para organizar el código

La función **main()** se define para encapsular la lógica del programa, pero **PODRÍA ADOPTAR CUALQUIER OTRO NOMBRE**

-if __name__ == "__main__": permite distinguir entre el código que debe ejecutarse **cuando el script se ejecuta directamente** y cuando el **código no debe ejecutarse** cuando **se importa como un módulo**.

Ejemplos

```
def saludar(nombre):
    return f"Hola, {nombre}!"

def main():
    nombre = input("Ingresa tu nombre: ")
    saludo = saludar(nombre)
    print(saludo)

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

```
def saludar(nombre):
    return f"Hola, {nombre}!"

def hola():
    nombre = input("Ingresa tu
nombre: ")
    saludo = saludar(nombre)
    print(saludo)

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

Ejercicio

Implemente la clase Hora que contenga miembros datos separados para almacenar horas, minutos y segundos. Un constructor inicializará estos datos en 0 y otro a valores dados. Una función miembro deberá visualizar la hora en formato hh:mm:ss. Otra función miembro sumará dos objetos de tipo hora, retornando la hora resultante. Realizar otra versión de la suma que asigne el resultado de la suma en el primer objeto hora.

b) Programar un procedimiento main(), que cree dos horas inicializadas y uno que no lo esté. Se deberán sumar los dos objetos inicializados, dejando el resultado en el objeto no inicializado. Por último, se pide visualizar el valor resultante.