



UGR Universidad
del Gran Rosario

Trabajo práctico 1

Programación 2

Principios del Paradigma Orientado a Objetos

Cuatrimestre 1 - Año 2025

Alumno:

Brian Martin Ortiz

Objetivo:

Aplicar los principios fundamentales de la Programación Orientada a Objetos (POO) en Python, incluyendo **polimorfismo, abstracción, herencia y encapsulamiento**. Los estudiantes desarrollarán y manipularán colecciones de objetos, realizarán búsquedas y filtrados en estas colecciones, y resolverán problemas prácticos mediante la implementación de clases y métodos.

Problemática

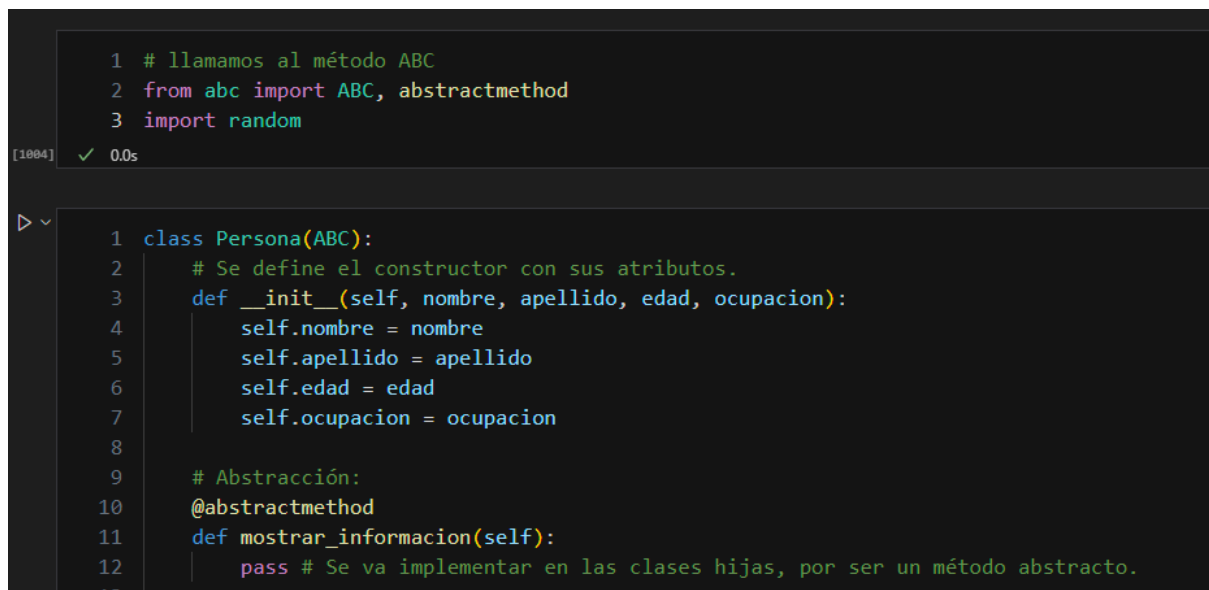
Se te ha contratado para desarrollar un sistema de gestión de personas para una institución educativa. El sistema debe permitir manejar la información de estudiantes, docentes y personal administrativo, y proporcionar herramientas para realizar diversas consultas y manipulaciones sobre los datos de estas personas.

Introducción y Preparación de Datos

1. Carga de Datos:

Comenzamos importando las librerías necesarias: **ABC** del módulo de **abc** y **random** (que luego nos servirá para hacer las IDs aleatorias)

Procedemos con la creación de la clase **Persona**, que será una clase **abstracta**, donde se definirán atributos como nombre, apellido, edad y ocupación.



```
1 # llamamos al método ABC
2 from abc import ABC, abstractmethod
3 import random

[1004] ✓ 0.0s

1 class Persona(ABC):
2     # Se define el constructor con sus atributos.
3     def __init__(self, nombre, apellido, edad, ocupacion):
4         self.nombre = nombre
5         self.apellido = apellido
6         self.edad = edad
7         self.ocupacion = ocupacion
8
9     # Abstracción:
10    @abstractmethod
11    def mostrar_informacion(self):
12        pass # Se va implementar en las clases hijas, por ser un método abstracto.
```

Imagen 1 - Creación de la clase Persona

Usaremos el decorador **@abstractmethod** para poder crear un método de clase Persona llamado **'mostrar_información'**. Este se implementará luego dentro de las clases hijas que vayamos a crear.

A continuación creamos varias **subclases** que hereden de **Persona**, las cuales van a ser **Estudiante, Docente y Administrativo**. Donde se incluye en cada una de ellas 1 atributo adicional específico.

Como se observa en la imagen 2, se utilizan los atributos de la clase padre mediante el uso de **super()**, y se añade el atributo **matricula**, el cual se va a generar automáticamente al momento de crear una instancia.

```
1 class Estudiante(Persona): # Hereda de la clase Persona (Padre)
2     def __init__(self, nombre, apellido, edad, ocupacion):
3         super().__init__(nombre, apellido, edad, ocupacion)
4         self.matricula = self.generar_id() # atributo adicional de la clase.
5
6     # Se implementa el 'mostrar_informacion' de la clase padre.
7     def mostrar_informacion(self):
8         return f"Mi nombre es {self.nombre} {self.apellido}, tengo {self.edad} años, soy {self.ocupacion}, Matricula: {self.matricula}"
9
```

Imagen 2 - Creación de la subclase Estudiante

Tanto estudiante como las siguientes subclases tendrán implementado, como dijimos anteriormente, el método “mostrar_información”. Donde se formatea una cadena para cada clase diferente.

En la clase Docente, se incluye como nuevo atributo la **asignatura**.

```
1 class Docente(Persona): # Hereda de la clase Persona (Padre)
2     def __init__(self, nombre, apellido, edad, ocupacion, asignatura):
3         super().__init__(nombre, apellido, edad, ocupacion)
4         self.asignatura = asignatura # atributo adicional de la clase.
5
6     # Se implementa el 'mostrar_informacion' de la clase padre.
7     def mostrar_informacion(self):
8         return f"Mi nombre es {self.nombre} {self.apellido}, tengo {self.edad} años, soy {self.ocupacion} de {self.asignatura}"
9
```

Imagen 3 - Creación de la subclase Docente

Por último, en la clase **Administrador** se incluye el atributo **legajo**, que también se generará automáticamente, y un atributo **salario**, el cual será de tipo **Privado**.

```
1 class Administrativo(Persona): # Hereda de la clase Persona (Padre)
2     def __init__(self, nombre, apellido, edad, ocupacion, salario):
3         super().__init__(nombre, apellido, edad, ocupacion)
4         self.legajo = self.generar_id() # atributo adicional de la clase.
5         self.__salario = salario # atributo PRIVADO de la clase.
6
7     # Se implementa el 'mostrar_informacion' de la clase padre.
8     def mostrar_informacion(self):
9         return f"Mi nombre es {self.nombre} {self.apellido}, tengo {self.edad} años, soy {self.ocupacion}, Legajo: {self.legajo}"
10
```

Imagen 4 - Creación de la subclase Administrativo

Lo siguiente que haremos será implementar el **encapsulamiento** utilizando los decoradores **@property** y **@setter** dentro de la clase Administrativo. Específicamente para el método 'Salario'.

```
# Encapsulamiento:
@property # Funciona como getter del método salario.
def salario(self):
    return f"Salario: {self.__salario}"

@salario.setter # Setter del método salario.
def salario(self, salario):
    if (isinstance(salario, float) or isinstance(salario, int)) and salario > 0: # validación del método salario.
        self.__salario = salario
    else:
        raise ValueError("Salario debe ser un valor numérico y positivo") # Mensaje de error personalizado.
```

Imagen 5 - Implementando Método @property y @Setter

Esto me permitirá acceder al valor del atributo privado o modificarlo como si fuera público. Luego, realizamos una validación que arroje un mensaje de error si no se cumple con la condición dada. (valor numérico y positivo)

Creamos algunas instancias de las diferentes subclases:

```
1 # Se crea una instancia de Estudiante.
estudiante1 = Estudiante("Juan", "Perez", 34, "estudiante")
Success
009] ✓ 0.0s
```

Imagen 6 - Instancia de Estudiante

```
1 # Se crea una instancia de Administrativo.
2 admin1 = Administrativo("Pablo", "Gomez", 27, "administrativo", 3000)
11] ✓ 0.0s
```

Imagen 7 - Instancia de Administrativo

```
> 1 # Se crea una instancia de Docente.
2 docente1 = Docente("Fernando", "Suarez", 45, "Docente", "Programación 2")
1016] ✓ 0.0s
```

Imagen 8 - Instancia de Docente

Se genera una lista de personas compuesta por objetos de diferentes subclases. Luego, mediante un ciclo **for**, se recorre esta lista para mostrar la información específica de cada objeto. Esto demuestra el uso del principio de **polimorfismo**, ya que cada subclase implementa su propia versión del método.

```
1 # Creo una colección de personas en una lista.
2 listPersonas = [admin1, estudiante1, docente1]

[1018] ✓ 0.0s

1 # Recorro con un for toda la lista (polimorfismo)
2 for per in listPersonas:
3     print(per.mostrar_informacion()) # se muestra la información específica de cada clase.

[1019] ✓ 0.0s

... Mi nombre es Pablo Gomez, tengo 27 años, soy administrativo, Legajo: GomLoAdm542
    Mi nombre es Juan Perez, tengo 34 años, soy estudiante, Matricula: PerAnEst868
    Mi nombre es Fernando Suarez, tengo 45 años, soy Docente de Programación 2
```

Imagen 9 - Aplicando polimorfismo

Análisis de la Colección de Datos

2. Filtrado y Búsqueda:

En esta ocasión, implementaremos la función **'filtrar_mayores_a'**, la cual devuelve una lista de personas cuya edad sea mayor al valor recibido como parámetro.

```
1 # función para filtrar por edad.
2 def filtrar_mayores_a(personas, edad):
3     '''Devuelve una lista de personas cuya edad es mayor al valor dado'''
4
5     if not isinstance(edad, int): # si no es de tipo entero...
6         print("Error: La edad debe ser un número entero.")
7         return
8
9
10    for per in personas:
11        if per.edad > edad:
12            print(per.mostrar_informacion())

[1020] ✓ 0.0s

1 filtrar_mayores_a(listPersonas, 20)

[1021] ✓ 0.0s

... Mi nombre es Pablo Gomez, tengo 27 años, soy administrativo, Legajo: GomLoAdm542
    Mi nombre es Juan Perez, tengo 34 años, soy estudiante, Matricula: PerAnEst868
    Mi nombre es Fernando Suarez, tengo 45 años, soy Docente de Programación 2
```

Imagen 10 - Filtrado por personas mayores de una cierta edad

Tal como se muestra en la *imagen 10*, se implementa una validación para manejar el error cuando el usuario introduce un valor no entero.

En la imagen 11 y 12 se implementan las funciones **filtrar_x_ocupación** y **filtrar_x_apellido**, que devuelven listas de personas según el valor ingresado. Ambas validan que el parámetro sea una cadena de texto y no distinguen entre mayúsculas y minúsculas al comparar.

```

1 # función para filtrar por ocupación.
2 def filtrar_x_ocupacion(personas, ocupacion):
3     encontrado = 0
4
5     if not isinstance(ocupacion, str): # si no es de tipo string...
6         print("Error: La ocupación debe ser un texto.(string)")
7         return
8
9     for per in personas:
10        if per.ocupacion.lower() == ocupacion.lower():
11            print(per.mostrar_informacion())
12            encontrado += 1
13
14    if encontrado == 0:
15        print("No se encontraron personas con esa ocupación.")
16
1022] ✓ 0.0s

1 filtrar_x_ocupacion(listPersonas, "docente")
1023] ✓ 0.0s

... Mi nombre es Fernando Suarez, tengo 45 años, soy Docente de Programación 2

```

Imagen 11 - Filtrado por personas con cierta ocupación

```

1 # función para filtrar por apellido.
2 def filtrar_x_apellido(personas, apellido):
3     encontrado = 0
4
5     if not isinstance(apellido, str): # Si no es de tipo string...
6         print("Error: El apellido debe ser un texto.(string)")
7         return
8
9     for per in personas:
10        if per.apellido.lower() == apellido.lower():
11            print(per.mostrar_informacion())
12            encontrado += 1
13
14    if encontrado == 0:
15        print("No existe ese apellido en la lista.")
16
1024] ✓ 0.0s

1 filtrar_x_apellido(listPersonas, "perez")
1025] ✓ 0.0s

... Mi nombre es Juan Perez, tengo 34 años, soy estudiante, Matricula: PerAnEst868

```

Imagen 12 - Filtrado por personas con un apellido específico

Manipulación y Operaciones Avanzadas

3. Operaciones Avanzadas:

Creamos un método **‘actualizar_información’** que permite modificar de forma selectiva los atributos de una instancia de la clase Persona. También validamos todo tipo de ingresos por parte del usuario.

```
# Se genera polimorfismo de 'actualizar_informacion'.
def actualizar_informacion(self, nombre=None, edad=None, apellido=None, ocupacion=None):
    if nombre is not None:
        if isinstance(nombre, str) and nombre.strip(): # validando nombre
            self.nombre = nombre
        else:
            print("Error: El nombre debe ser una cadena no vacía.")
    if apellido is not None:
        if isinstance(apellido, str) and apellido.strip(): # validando apellido
            self.apellido = apellido
        else:
            print("Error: El apellido debe ser una cadena no vacía.")
    if edad is not None:
        if isinstance(edad, int) and edad > 0: # validando edad
            self.edad = edad
        else:
            print("Error: La edad debe ser un número entero positivo.")
    if ocupacion is not None:
        if isinstance(ocupacion, str) and ocupacion.strip(): # validando ocupacion
            self.ocupacion = ocupacion
        else:
            print("Error: La ocupacion debe ser una cadena no vacía.")
```

Imagen 13 - método actualizar información (clase Persona)

Mediante el método **generar_id** (imagen 14), aplicamos polimorfismo entre Estudiante y Administrativo, permitiendo que la matrícula o legajo se actualicen automáticamente al modificar datos personales.

```
# Se genera polimorfismo entre subclases (Estudiante, Administrativo).
def generar_id(self):
    return f"{self.apellido[:3]}{self.nombre[-2:].capitalize()}{self.ocupacion[:3].capitalize()}{random.randint(100,999)}"

# Se genera polimorfismo de 'actualizar_informacion'.
def actualizar_informacion(self, nombre=None, edad=None, apellido=None, ocupacion=None):
    super().actualizar_informacion(nombre, edad, apellido, ocupacion)

    # Generar nuevo ID si hago un cambio en los datos.
    if nombre or apellido or ocupacion:
        self.matricula = self.generar_id()
```

Imagen 14 - método generar_id (clase Estudiante)

```

# Se genera polimorfismo entre subclases (Estudiante, Administrativo).
def generar_id(self):
    return f"{self.apellido[:3]}{self.nombre[-2:].capitalize()}{self.ocupacion[:3].capitalize()}{random.randint(100,999)}"

# Se genera polimorfismo de 'actualizar_informacion'.
def actualizar_informacion(self, nombre=None, edad=None, apellido=None, ocupacion=None, salario=None):
    super().actualizar_informacion(nombre, edad, apellido, ocupacion)

    # Generar nuevo ID si hago un cambio en los datos.
    if nombre or apellido or ocupacion:
        self.legajo = self.generar_id()

    if salario is not None:
        self.salario = salario # Pasa primero por el setter para validar, manteniendo el atributo en privado.

```

Imagen 15 - método generar_id (clase Administrativo)

Se prestó especial importancia que el valor del salario se mantenga en privado por más que este se haya actualizado. Se probó que al cambiar dichos datos, este primero pasa por la validación del setter para luego almacenarse dentro del atributo privado salario.

Usamos el método "actualizar_información" de Estudiante y generamos un ID para la matrícula.

```

1 estudiante1.mostrar_informacion()
[1026] ✓ 0.0s
... 'Mi nombre es Juan Perez, tengo 34 años, soy estudiante, Matricula: PerAnEst868'

1 estudiante1.actualizar_informacion(nombre="Franco", apellido=None, edad= 44, ocupacion= "albañil")
[1027] ✓ 0.0s

1 estudiante1.mostrar_informacion()
[1028] ✓ 0.0s
... 'Mi nombre es Franco Perez, tengo 44 años, soy albañil, Matricula: PerCoAlb208'

```

Imagen 14 - Uso del método actualizar información

En la *imagen 14* se actualizan el nombre, la edad y la ocupación de la instancia 'estudiante1', manteniendo el apellido original. Como resultado, se genera correctamente una nueva matrícula, cambiando de 'Per**AnEst868**' a 'Per**CoAlb208**'. Esto confirma que el identificador se actualiza automáticamente en función de los nuevos datos ingresados, tal como fue programado en el método correspondiente.

```

# Métodos de clase:
@classmethod
def mostrar_orden(cls, lista, atributo='edad', descendente=False):
    filtrados = []

    for per in lista:
        if isinstance(per, cls): # esta esa persona en la clase..?
            filtrados.append(per)

    try:
        ordenados = sorted(filtrados, key=lambda p: getattr(p, atributo), reverse=descendente) # ordenalo según el atributo recibido
    except:
        print(f"Error: No todos los objetos tiene el atributo '{atributo}'")
        return

    for persona in ordenados:
        print(persona)

```

Imagen 15 - método mostrar orden

En el método mostrado en la imagen 15, se decidió utilizar el decorador `@classmethod`, lo que permitió simplificar partes del código que se repetían. Gracias a esto, la misma lógica de ordenar puede aplicarse de manera reutilizable en distintas subclases de la clase `Persona`.

```

1 Persona.mostrar_orden(listPersonas, "edad")
✓ 0.0s
...
Estudiante: Carlos Ganzo (23)
Administrativo: Pablo Gomez (27)
Administrativo: Tomas Lopez (27)
Estudiante: Pablo Giménez (29)
Docente: Carlos Suarez (33)
Estudiante: Franco Perez (44)
Docente: Federico Pineda (44)

1 Docente.mostrar_orden(listPersonas, "edad")
✓ 0.0s
...
Docente: Carlos Suarez (33)
Docente: Federico Pineda (44)

1 Persona.mostrar_orden(listPersonas, "nombre")
✓ 0.0s
...
Docente: Carlos Suarez (33)
Estudiante: Carlos Ganzo (23)
Docente: Federico Pineda (44)
Estudiante: Franco Perez (44)
Administrativo: Pablo Gomez (27)
Estudiante: Pablo Giménez (29)
Administrativo: Tomas Lopez (27)

```

Imagen 16 - Uso del método mostrar orden

Al utilizar el método mostrado en la *imagen 16*, tanto la clase `Persona` como la subclase `Docente` logran un ordenamiento correcto de los objetos. Esta funcionalidad también se puede aplicar a otras subclases como `Estudiante` o `Administrador`, simplemente indicando como parámetro el criterio por el cual se desea ordenar. En este caso, se utilizaron los criterios "edad" y "nombre". Además, es posible incluir un tercer parámetro para especificar si el orden debe ser ascendente o descendente.

Cabe mencionar que se tuvo que formatear la salida de estas listas con ayuda de los métodos especiales `'__str__'` y `'__repr__'`, los cuales permiten representar los objetos de forma legible al imprimirlos.

```

# Formatear salida
def __str__(self):
    return f"{self.__class__.__name__}: {self.nombre} {self.apellido} ({self.edad})"

# Imprimir lista:
def __repr__(self):
    return self.__str__()

```

Imagen 17 - Uso de métodos especiales.