1. Introducción.

En esta unidad se estudiarán aspectos avanzados relacionados con la programación orientada a objetos. Veremos términos como delegación, agregación, herencia, polimorfismo, clases abstractas, interfaces. Para proceder con esta unidad es imprescindible haber comprendido los conceptos básicos de la POO vistos en la unidad anterior.

2. Delegación.

Cuando una clase contiene entre sus atributos una o más instancias de otras clases, existe lo que se conoce como delegación. Las instancias de las clases contenidas se crean dentro de la clase contenedora. Si el objeto de la clase contenedora desaparece, lo harán también los objetos contenidos. Existe una relación de dependencia entre la clase contenida y la contenedora.

Estudiemos un ejemplo de delegación con las clases ***Vehículo y Motor***. Una instancia de la clase ***Vehículo*** contendrá un objeto ***Motor*** entre sus atributos. Observa cómo este objeto ***Motor*** se crea dentro del objeto ***Vehículo*** .

La primera imagen contiene el diseño de las clases, la segunda un ejemplo de uso de las mismas.

```
1   # Clase contenida
2   class Motor:
3       def __init__(self, cilindros, tipo):
4           self.cilindros = cilindros
5           self.tipo = tipo
6
7       def __str__(self):
8           return f"Motor de {self.cilindros} cilindros {self.tipo}"
9
10      def __eq__(self, otro):
11          # Compara si dos motores son iguales
12          return self.cilindros == otro.cilindros and self.tipo == otro.tipo
13
14  # Clase contenedora
15  class Vehiculo:
16      def __init__(self, marca, modelo, cilindros, tipo):
17          self.marca = marca
18          self.modelo = modelo
19          self.motor = Motor(cilindros, tipo)  # delegación
20
21      def __str__(self):
22          return f"{self.marca} {self.modelo} con motor {self.motor}"
23
24      def compara_motor(self, otro):
25          return self.motor == otro.motor
```

- Línea 10. Comparación de los motores de dos vehículos.
- Línea 19. Construcción del objeto contenido.
- Línea 24. Comparar el motor de dos vehículos.

```python
from clases import Vehiculo, Motor

fiesta = Vehiculo("Ford", "Fiesta", 4, "gasolina")
rio = Vehiculo("Kia", "Rio", 4, "gasolina")

print(fiesta)
print(rio)

# comparar motores de rio y fiesta
print(fiesta.compara_motor(rio))  # True
```

3. Agregación.

La agregación coincide con la delegación salvo en el hecho de que los objetos contenedor y contenido tienen vidas independientes. Si se destruye el objeto contenedor, los objetos contenidos permanecen.

Estudiaremos las clases ***Campeonato y Equipo*** como ejemplo de agregación, de tal manera que un campeonato está constituido por varios equipos.

Estudia las siguientes imágenes.

```python
# clase contenida
class Equipo:
    def __init__(self, nombre, ciudad):
        self.nombre = nombre
        self.ciudad = ciudad

    def __str__(self):
        return f"{self.nombre} de {self.ciudad}"

    def __eq__(self, other):
        # dos equipos son iguales si tienen el mismo nombre y ciudad
        return self.nombre == other.nombre and self.ciudad == other.ciudad


# clase contenedora
class Campeonato:
    def __init__(self, nombre, anio, descripcion):
        self.nombre = nombre
        self.anio = anio
        self.descripcion = descripcion
        self.equipos = []

    def __str__(self):
        return f"{self.nombre} de {self.ciudad} - {self.equipo}"

    def agregar_equipo(self, equipo):
        # comprobar si el equipo ya está en la lista
        # funciona al estar definido el método __eq__ en la clase Equipo
        if equipo in self.equipos:
            return False
        self.equipos.append(equipo)
        return True
```

```python
from clases import *

liga = Campeonato('Liga', 2021, 'Campeonato de fútbol')

madrid = Equipo('Real Madrid', 'Madrid')
betis = Equipo('Betis', 'Sevilla')

liga.agregar_equipo(madrid)
liga.agregar_equipo(betis)
liga.agregar_equipo(Equipo('Athletic', 'Bilbao'))
liga.agregar_equipo(Equipo('Athletic', 'Bilbao'))

for equipo in liga.equipos:
    print(equipo)
```

- Las instancias ***madrid y betis*** se crean fuera del objeto ***liga*** . Si se elimina el objeto ***liga*** , los otros dos permanecen.
- Al añadir el equipo Athletic no obtenemos referencia alguna.

4. Herencia.

La herencia es uno de los pilares fundamentales de la POO. Mediante la herencia se definen clases basadas en otras ya existentes. Se le llama **superclase** a la clase principal o clase de la que se hereda, y **subclase** a la clase que hereda de la principal.

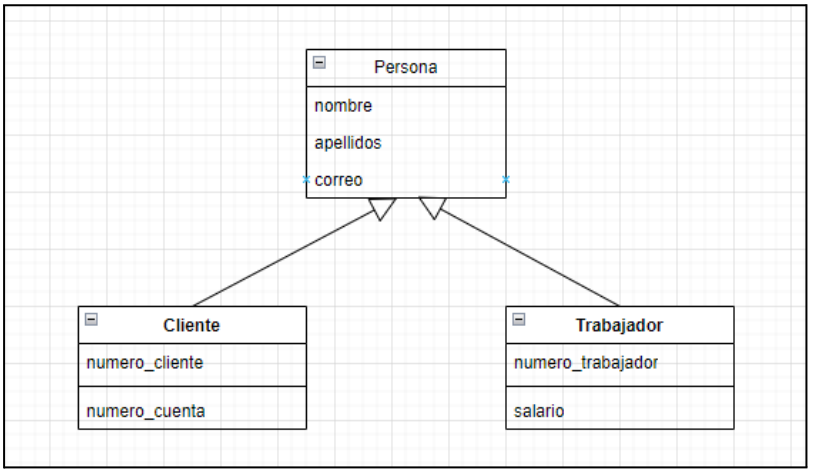
La herencia permite la reutilización del código . Las líneas de código escritas en la superclase pueden reutilizarse en las subclases. Las subclases pueden incluir nuevos métodos y además pueden **sobrescribir** los heredados.

A la superclase se le suele llamar también clase base o principal, mientras que a la subclase se la conoce también como clase derivada o clase hija.

Las superclases suelen estar formadas por atributos genéricos y las subclases heredan estos atributos genéricos y además definen otros específicos.

Estudiemos estos conceptos con un ejemplo. *Debemos desarrollar una aplicación Python que trabaje con personas. Estas personas pueden ser clientes o trabajadores. Existen atributos comunes que comparten tanto clientes como trabajadores, pero otros son exclusivos de cada tipo de persona. De todas las personas se guarda nombre, apellidos y correo electrónico. De los clientes se guarda, además el número de cliente y el número de cuenta. De los trabajadores se necesita el número de trabajador y el salario.*

Dadas estas características podemos pensar en crear una clase ***Persona*** con los atributos comunes a todas las personas, y crear también las clases **Cliente y Trabajador** que heredan de ***Persona*** y que además tienen atributos específicos. Gráficamente se vería como indica la imagen.



El código Python asociado sería este:

```python
# definición de las clases
# clase base
class Persona:
    def __init__(self, nombre, apellidos, correo):
        self.nombre = nombre
        self.apellidos = apellidos
        self.correo = correo

    def __str__(self):
        return f"{self.nombre} {self.apellidos} - {self.correo}"

# clase derivada
class Cliente(Persona):
    def __init__(self, nombre, apellidos, correo, num_cliente, numero_cuenta):
        super().__init__(nombre, apellidos, correo)
        self.num_cliente = num_cliente
        self.numero_cuenta = numero_cuenta

    def __str__(self):
        return super().__str__() + f" - {self.num_cliente} - {self.numero_cuenta}"


# clase derivada
class Trabajador(Persona):
    def __init__(self, nombre, apellidos, correo, num_trabajador, salario):
        super().__init__(nombre, apellidos, correo)
        self.num_trabajador = num_trabajador
        self.salario = salario

    def __str__(self):
        return super().__str__() + f" - {self.num_trabajador} - {self.salario}"
```

- La superclase ***Persona*** contiene constructor y el método __str__().
- ***Las clases Cliente y Trabajador heredan de Persona.*** Esto se indica haciendo referencia a la clase Persona entre paréntesis, líneas 13 y 24.
- En el constructor de ***Cliente*** se pasan como argumentos los atributos comunes y los específicos (número de cliente y número de cuenta). Para inicializar los atributos comunes en esta clase derivada se llama al constructor de la superclase utilizando la forma **super().__init__(nombre, apellidos, correo)** .
- En el constructor de la clase ***Trabajador*** se realiza la misma operación.
- El método __str__() se ha redefinido en las subclases, llamando al método heredado mediante super.__str__() y añadiendo los atributos específicos. Líneas 20 y 31.

En la imagen siguiente se observa como se instancian objetos ***Cliente y Trabajador***.

```python
from clases import Cliente, Persona, Trabajador

cli1 = Cliente("Juan", "García", "juan@gmail.com", 10, "ES123456789")
tra1 = Trabajador("Ana", "López", "ana@gmail.com", 100, 1500)
tra2 = Trabajador("Pedro", "Martínez", "pedro@gmail.com", 200, 2000)
```

Python permite:

- Herencia simple. Una clase herede de otra.
- Herencia jerárquica. Se da cuando de una superclase heredan varias subclases.
- Herencia múltiple. Consiste en que una subclase hereda de varias superclases.
- Herencia multinivel. En este caso una subclase hereda de una superclase que a su vez es subclase de otra.

5. Funciones type(), issubclass() e isinstance().

Estas funciones nos ayudarán en el trabajo con clases y herencia.

- **type(object).** Retorna la clase asociada al objeto pasado como argumento.
- **isinstance(object, class).** Retorna True si el objeto pasado como primer argumento es una instancia de la clase pasada como segundo.
- **issubclass(class1, class2).** Retorna True si class1 es una subclase de class2.
- **Mediante type(objeto).__name__** se obtiene el nombre de la clase a la que pertenece un objeto.

Continuando con el ejemplo anterior (Persona, Cliente, Trabajador), estudiemos el siguiente código.

```
1   from clases import *
2
3   # lista de objetos Persona
4   personas = [
5       Cliente('Juan', 'Perez', 'juan@gmail.com', 1, '1234567890'),
6       Trabajador('Maria', 'Lopez', 'maria@gamil.com', 1, 1000),
7       Trabajador('Pedro', 'Gomez', 'pedro@ggg.com', 2, 2000),
8       Cliente('Ana', 'Martinez', 'ana@hotdog.es', 2, '0987654321'),
9       Cliente('Luis', 'Garcia', 'lliss@nomore.com' , 3, '1234567890')
10  ]
11
12  for persona in personas:
13      if isinstance(persona, Cliente):
14          print(f'{persona.nombre} {persona.apellidos} - {persona.correo}, es un cliente')
```

- Línea 4 y siguientes. Creación de una lista que almacena clientes y trabajadores.
- Línea 10. Bucle for que recorre la lista personas y por cada elemento de la misma se pregunta si se trata de un objeto ***Cliente*** mediante **isinstance(p, Cliente) .** En caso afirmativo, se imprime su nombre. Es decir, haciendo uso de la función isinstance() podemos seleccionar o filtrar los objetos por tipo o clase.

Autoevaluación-1.

1. ¿Qué habría sucedido si cambiamos el bucle for por el siguiente?

```
for p in personas:
  if isinstance(p, Persona):
    print(f'{p.nombre} {p.apellidos}')
```

6. Polimorfismo.

Polimorfismo significa varias formas . Pero ¿cómo se aplica este concepto a la programación orientada a objetos?. Pues bien, utilizando el polimorfismo podemos invocar al mismo método con objetos de distintas clases y obtener resultados distintos, según sea el tipo o la clase del objeto utilizado. Es decir, tenemos en varias clases, un método común con distintas implementaciones, y el intérprete de Python llamará al método correcto según sea la clase del objeto que realiza la llamada.

Siguiendo con el ejemplo anterior basado en las clases ***Persona, Cliente y Trabajador***, estudiaremos el código siguiente.

```
1   from clases import *
2
3   # lista de objetos Persona
4   personas = [Cliente("Juan", "García", "juangarcia@gmail.com", 1, "ES1234567890"),
5               Cliente("Ana", "López", "analopez@ganga.es", 2, "ES0987654321"),
6               Trabajador("Pedro", "Martínez", "pedromm@gmail.com", 1, 1500),
7               Trabajador("María", "Gómez", "mariagomez@hotmail.com", 2, 2000),
8               Trabajador("Luis", "González", "luisgonz@gonzi.net", 3, 1800)]
9
10  for p in personas:
11      print(p)
```

La salida sería la siguiente:

```
Juan García - juangarcia@gmail.com - 1 - ES1234567890
Ana López - analopez@ganga.es - 2 - ES0987654321
Pedro Martínez - pedromm@gmail.com - 1 - 1500
María Gómez - mariagomez@hotmail.com - 2 - 2000
Luis González - luisgonz@gonzi.net - 3 - 1800
```

- Al imprimir el objeto se está llamando al método __str__(), pero en la lista existen tanto objetos ***Cliente como Trabajador***, ¿a qué método __str__() se está llamando? Gracias al polimorfismo, se llamará al método correcto en función del objeto que realice la llamada.

Hemos explicado el polimorfismo basándonos en un sistema de clases donde se utiliza herencia, pero Python va incluso más allá. En el ejemplo que sigue, partimos de dos clases (Triángulo y Círculo) que comparten el método área , que como es natural tiene una implementación distinta según se trate de un triángulo o de un círculo. Al ser Python un lenguaje que utiliza el tipado dinámico podemos crear una función, *calcular_area(figura)*, que reciba una instancia de ***Triángulo o de Círculo*** y que llame al método ***área*** correspondiente.

```python
class Triangulo:
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    def area(self):
        return self.base * self.altura / 2


class Circulo:
    def __init__(self, radio):
        self.radio = radio

    def area(self):
        return 3.14159 * self.radio ** 2
```

```python
from figuras import Triangulo, Circulo

def calcular_area(figura):
    return figura.area()


circulo = Circulo(3)
triangulo = Triangulo(4, 3)

print(f'Área de un circulo de radio 3: {calcular_area(circulo)}')
print(f'Área de un triángulo de base 4 y altura 3: {calcular_area(triangulo)}')
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  COMMENTS

TERMINAL

PS C:\Users\jara2\Documents\python_apps\Curso2425\POO\Unidad_01> & C:/Users/jara2/Ap
sers/jara2/Documents/python_apps/Curso2425/POO/Unidad_01/tests/test_03_polimorfismo/
Área de un circulo de radio 3: 28.27431
Área de un triángulo de base 4 y altura 3: 6.0
```

7. Clases base abstractas.

Imaginemos el siguiente contexto: *Debemos crear una aplicación que trabaje con empleados, y guarde nombre y salario de cada uno de ellos. Estos empleados pueden ser de dos tipos: directores y vendedores. De los directores se almacena, además un bono y de los vendedores una comisión.* ***Es necesario mencionar que en esta aplicación no se crearán objetos de tipo empleado, sólo se crearán directores y vendedores.***

En estas situaciones en las que no se instancian objetos de la clase base y esta solo sirve de plantilla para las subclases que heredan de ella, es donde se puede recurrir a la creación de clases abstractas.

Una clase base abstracta es aquella que no puede ser instanciada, y además obliga a que las subclases que heredan de ella, implementen los métodos que en la clase base no tienen implementación. A estos métodos se les llama métodos abstractos.

En Python, para que una clase sea considerada abstracta debe tener al menos un método abstracto. En otro caso, podrá ser instanciada.

Para utilizar clases abstractas en Python se debe importar la clase ***ABC*** y el decorador ***@abstractmethod*** del módulo abc tal y como se muestra en la imagen de la derecha.

La clase base debe heredar de ***ABC*** para ser considerada abstracta y tener al menos un método abstracto.

Los métodos abstractos deben llevar el decorador ***@abstracmethod***. Observa como ***get_info()*** es un método sin implementación.

```python
# definición de clases
from abc import ABC, abstractmethod


# clase abstracta
class Empleado(ABC):
    def __init__(self, nombre, salario):
        self.nombre = nombre
        self.salario = salario

    # método abstracto
    @abstractmethod
    def get_info(self):
        pass
```

Por otro lado, las subclases quedarían como se observa en la imagen de la derecha. Ambas implementan el método ***get_info()*** que fue declarado abstracto en la superclase. **Una clase que hereda de otra que es abstracta, tiene la obligación de implementar los métodos abstractos heredados.**

```python
class Director(Empleado):
    def __init__(self, nombre, salario, bono):
        super().__init__(nombre, salario)
        self.bono = bono

    def get_info(self):
        return (
            f"Nombre: {self.nombre}, Salario: {self.salario},"
            f" Comisión: {self.comision}"
        )


class Vendedor(Empleado):
    def __init__(self, nombre, salario, comision):
        super().__init__(nombre, salario)
        self.comision = comision

    def get_info(self):
        return (
            f"Nombre: {self.nombre}, Salario: {self.salario},"
            f" Comisión: {self.comision}"
        )

```

Las diferentes implementaciones de un método abstracto pueden añadir nuevos parámetros a la definición original.

8. Interfaces.

Para entender el concepto de interfaz estudiaremos el siguiente ejemplo: *Un grupo de desarrolladores debe diseñar una aplicación que trabaje con varios tipos de archivos para leer y escribir datos en los mismos. Estos archivos podrán ser de texto, csv y pdf, aunque no se descarta que en el futuro se extienda la funcionalidad de la misma a otros tipos de archivos.*

Para dar solución al problema podemos pensar en diseñar una clase que tenga métodos para leer y escribir en archivos de texto, otra para leer y escribir en archivos csv, y otra para documentos pdf. La idea es que todas estas clases incorporen los mismos métodos y éstos tengan una implementación basada en el tipo de archivos que tratan. El hecho de que estas y las futuras clases que deban diseñarse se deban ajustar a un patrón, lo solucionamos con la incorporación de interfaces a la aplicación. Si cada desarrollador del grupo diseña una de estas clases, mediante la interfaz nos aseguramos de que los métodos de estas clases sean los mismos aunque difieran en su implementación.

Una interfaz sirve de plantilla para definir clases. Los métodos definidos en una interfaz son abstractos y deben ser implementados en las clases que se ajustan a esta interfaz. Las interfaces son útiles para garantizar que ciertas funcionalidades se implementan en diferentes clases sin importar la jerarquía.

La diferencia entre una interfaz y una clase abstracta es que la primera solo incluye métodos abstractos, mientras que la clase abstracta define métodos abstractos y/o concretos.

A continuación estudiaremos dos formas de trabajar con interfaces.

8.1. Interfaces informales

Una interfaz informal es una clase simple donde no se implementan los métodos definidos. Aunque es viable, **no obliga a implementar los métodos definidos en la interfaz.**

En el ejemplo que sigue se crea una interfaz informal, GestorArchivosInterface , donde se definen dos métodos sin código para leer y escribir en archivos. Las clases ArchivosTexto y ArchivosCSV implementan esta interfaz para leer/escribir archivos de texto y csv respectivamente.

Piensa que el ejemplo está pensado para estudiar las interfaces, no para tratar cómo se lee y escribe en archivos. No te centres en la implementación de los métodos *leer_archivo* y *escribir_archivo* .

```python
import csv

# Creación de la interfaz informal
class GestorArchivosInterface:
    def leer_archivo(self, archivo):
        pass

    def escribir_archivo(self, archivo, contenido):
        pass
```

```python
# Implementación de la interfaz informal
class ArchivosTexto(GestorArchivosInterface):
    def leer_archivo(self, archivo):
        with open(archivo, "r") as f:
            return f.read()

    def escribir_archivo(self, archivo, contenido):
        with open(archivo, "w") as f:
            f.write(contenido)


# Implementación de la interfaz informal
class ArchivosCSV(GestorArchivosInterface):
    def leer_archivo(self, archivo):
        with open(archivo, newline="", encoding="utf-8") as f:
            return list(csv.reader(f))

    def escribir_archivo(self, archivo, contenido):
        with open(archivo, "w", encoding='utf-8') as f:
            for sublista in contenido:
                f.write(",".join(map(str, sublista)))
                f.write("\n")
```

```python
import os
from clases import ArchivosCSV, ArchivosTexto

# Creación de instancias
a_texto = ArchivosTexto()
a_csv = ArchivosCSV()

# Obtener la ruta del archivo actual
ruta = os.path.dirname(os.path.abspath(__file__))

# Leer el archivo texto.txt
contenido_txt = a_texto.leer_archivo(f"{ruta}\\texto.txt")
print("Contenido del archivo de texto", contenido_txt, sep="\n")

# Leer el archivo equipos.csv
c_csv = a_csv.leer_archivo(f"{ruta}\\equipos.csv")
print("Contenido del archivo csv", c_csv, sep="\n")

# Escribir en los archivos texto_nuevo.txt y equipos_nuevo.csv
a_texto.escribir_archivo(f"{ruta}\\texto_nuevo.txt", "Hola mundo")
a_csv.escribir_archivo(f"{ruta}\\equipos_nuevo.csv", [["Cáceres", 12], ["Madrid", 10], ["Sevill
```

8.2. Interfaces formales.

Las interfaces formales en Python están basadas en la utilización de clases base abstractas. La idea es definir una interfaz o clase base abstracta (hereda de ABC) que incluya un conjunto de métodos abstractos y **las subclases que hereden de ella deben implementar obligatoriamente estos métodos.**

Utilizando esta forma de crear interfaces si que se obliga a que las clases que implementen la interfaz asignen código a esos métodos abstractos heredados.

Al estudiar líneas de código Python donde se definan interfaces, ya sean formales o informales, estas no se distinguen de las clases propiamente dichas, por lo que es aconsejable nombrarlas de forma distinta. **Podemos, por ejemplo, incluir la palabra Interface en el nombre, o incluir la letra `I` como primera letra del mismo.**

El ejemplo que se muestra a continuación es el mismo que el del apartado anterior, la gestión de archivos de texto y csv. Observa cómo la interfaz hereda de ***ABC*** y sus métodos son abstractos.

```python
import csv
from abc import ABC, abstractmethod

# Creación de la interfaz informal
class GestorArchivosInterface(ABC):

    @abstractmethod
    def leer_archivo(self, archivo):
        pass

    @abstractmethod
    def escribir_archivo(self, archivo, contenido):
        pass

```

```python
# Implementación de la interfaz formal
class ArchivosTexto(GestorArchivosInterface):
    def leer_archivo(self, archivo):
        with open(archivo, "r") as f:
            return f.read()

    def escribir_archivo(self, archivo, contenido):
        with open(archivo, "w") as f:
            f.write(contenido)


# Implementación de la interfaz formal
class ArchivosCSV(GestorArchivosInterface):
    def leer_archivo(self, archivo):
        with open(archivo, newline="", encoding="utf-8") as f:
            return list(csv.reader(f))

    def escribir_archivo(self, archivo, contenido):
        with open(archivo, "w", encoding='utf-8') as f:
            for sublista in contenido:
                f.write(",".join(map(str, sublista)))
                f.write("\n")
```

Autoevaluación-2.

1. Diseñar la clase Persona con nombre, apellidos y correo. Diseñar la interfaz ICrud con los métodos create, read, update, delete y mostrar.
   Diseñar la clase GestionPersona que implemente la interfaz ICrud. Esta clase dispone de una lista donde guardar personas. Asigna código a los métodos para:
   - Create. Crear una persona nueva.
   - Read. Buscar una persona por su correo.
   - Delete. Eliminar una persona de la que se conoce el correo.
   - Update. Actualiza el nombre de persona de la que se conoce correo.
   - Show. Muestra el contenido de la lista.

9. Sobrecarga de operadores y de funciones incorporadas.

Mediante la sobrecarga de operadores Python permite utilizar operadores nativos del lenguaje para llevar a cabo operaciones y comparaciones entre objetos definidos por el usuario. **La sobrecarga de operadores se lleva a cabo añadiendo determinados métodos a las clases sobre las que vamos a trabajar.**

Estudiemos el siguiente ejemplo: *Debemos diseñar una clase Trayecto para trabajar con distancias. De cada trayecto se guardará origen, destino y distancia entre ellos. La clase permitirá sumar y restar trayectos entre sí, utilizando los operadores aritméticos + y - respectivamente. Para ello se deben programar los métodos __add__() y __sub__() de la clase Trayecto.*

Estudia la clase ***Trayecto*** que se observa en la figura.

```
1   class Trayecto:
2       def __init__(self, origen, destino, distancia):
3           self.origen = origen
4           self.destino = destino
5           self.distancia = distancia
6
7       def __str__(self):
8           return f'{self.origen} - {self.destino} ({self.distancia} km)'
9
10      def __add__(self, otro):
11          return self.distancia + otro.distancia
12
13      def __sub__(self, otro):
14          return self.distancia - otro.distancia
```

- Línea 10. Sobrecarga del operador "+" mediante el método dunder __add__() . Se indica que la suma de dos trayectos es la suma de sus distancias.
- Línea 13. Sobrecarga del operador "-" mediante el método __sub__(). La diferencia entre dos trayectos es la resta de sus distancias.

En la imagen siguiente se muestran ejemplos de llamadas a estos métodos.

```
1   from clases import *
2
3   t1 = Trayecto('Madrid', 'Barcelona', 621)
4   t2 = Trayecto('Badajoz', 'Madrid', 400)
5
6   print(t1 + t2) # sumar trayectos
7   print(t1 - t2) # restar trayectos

ROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   COMMEN
TERMINAL
PS C:\Users\jara2\Documents\python_apps\Curso24
apps/Curso2425/POO/Scripts/python.exe c:/Users/
09_sobrecarga_operadores/pruebas.py
1021
221
```

Autoevaluación 3.

1. Modifica la clase Trayecto de forma que permita comparar dos trayectos entre sí y determinar si uno es mayor que otro utilizando el operador > .

Estas operaciones también es posible realizarlas con funciones incorporadas (built-in functions). En el ejemplo siguiente se hace uso de la función ***len()*** para determinar el número de proyectos en los que trabaja un empleado. Esto se realiza sobrecargando el método ***__len__()*** de la clase ***Empleado***. Por otro lado, se sobrecarga el método ***__bool__()*** que retorna True si el salario del empleado es mayor que cero. Recuerda que la función incorporada ***bool()*** retorna el valor booleano de un objeto.

```
4   class Empleado:
5       def __init__(self, nombre, salario):
6           self.nombre = nombre
7           self.salario = salario
8           self.proyectos = []
9
10      def __len__(self):
11          return len(self.proyectos)
12
13      def __bool__(self):
14          return self.salario > 0
15
16
17  emp1 = Empleado("Juan", 1000)
18  emp1.proyectos = ["Desarrollo web Aula", "App móvil apuestas", "React 2"]
19
20  print(len(emp1))  # len retorna el número de proyectos del empleado
21  print(bool(emp1))  # bool retorna True si el salario es mayor que 0
```

10. Enlaces de interés.
    - https://www.youtube.com/watch?v=bZXXZKBeCmQ&list=PLM-p96nOrGcaB_xm0Urir1yiQX4-rlXU3&index=24
    - https://ellibrodepython.com/abstract-base-class
    - https://realpython.com/python-interface/
    - https://pypi.org/project/python-interface/
    - https://realpython.com/operator-function-overloading/
    -