PROGRAMACIÓN III

Clase 7

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.

Clase 8

Manejo de excepciones y Módulos y paquetes

- Manejo de excepciones.
- Errores vs. excepciones.
- Múltiples excepciones, invocación y creación de excepciones propias.
- Módulos y packages.
- Librerías.
- Collections, datetime, math y módulo random.

Herencia y polimorfismo

Herencia

La herencia es un mecanismo de la Programación Orientada a Objetos (POO) que permite crear clases nuevas a partir de clase preexistentes.
Usando este concepto, las clases nuevas pueden tomar (heredar) atributos y métodos de clases anteriores. Incluso pueden modificarlos para modelar una nueva situación.
La clase que aporta los métodos y atributos para heredar se la denomina clase base, superclase o clase padre, y a las que se construyen a partir de ella clases derivadas, subclases o clases hijas.

Herencia, superclases y subclases

Una **superclase** es una clase superior o clase base. Si habíamos considerado las clases como “plantillas” para construir objetos, siguiendo esa analogía las superclases serían “plantillas de plantillas”.
A partir de una superclase se pueden definir **subclases** clases (clases derivadas), cada una compartiendo atributos y métodos con la clase superclase, aunque también pueden sumar métodos y atributos propios.
Para utilizar el concepto de **herencia** es necesario identificar una clase base (la superclase) **que posea los atributos y métodos comunes** y luego crear las demás clases heredando de ella (las subclases), extendiendo los métodos y atributos que sean necesarios.

HERENCIA SIMPLE

Este es un ejemplo muy simple del concepto de herencia. El objeto **hijo1** ha heredado los métodos y atributos de la superclases **Padre**, y podemos utilizarlos en hijo1. Vemos cómo **llevar()** (de la superclase) funciona en hijo1 de la misma manera que si fuese un método de la subclase. Lo mismo ocurre con el atributo apellido: estamos viendo el que ha heredado de la clase Padre.

Herencia simple:

```
class Padre: # Superclase
    def __init__(self):
        self.apellido = "Volpin"

    def llevar(self):
        print("Papá me lleva al colegio.")

class Hijo(Padre): # Subclase
    def estudiar(self):
        print("Estoy en el colegio.")


hijo1 = Hijo() # Instanciamos hijo1
hijo1.llevar() # Papá me lleva al colegio.
hijo1.estudiar() # Estoy estudiando
print(hijo1.apellido) # Volpin (heredado)
```

Herencia, superclases y subclases

De una superclase se pueden construir muchas subclases derivadas, o clases que heredan de ellas. Por ejemplo, de la superclase Persona podríamos derivar Docente, Empleado, Cliente, Proveedor, o las que sean necesarias para la aplicación que estemos desarrollando.
En el diseño de jerarquías de herencia no siempre es fácil decidir cuándo una clase debe extender a otra. La regla práctica para decidir si una clase S puede ser definida como heredera de otra T es que debe cumplirse que "S es un T".
Por ejemplo, Perro es un Animal, pero Vehículo no es un Motor.
Recordemos: las subclases heredan de las superclases.

Herencia simple

Veamos un ejemplo más complejo. Supongamos que necesitamos una clase que sea capaz de instanciar objetos que representen a los alumnos de **Programación III**.
Dado que los alumnos comparten buena parte de sus atributos y métodos con otras personas (alumnos de otras instituciones, docentes, etcétera) sería útil contar con una **superclase** llamada **Persona** que contenga los atributos y métodos comunes a todas las personas, y que la **subclase AlumnosProgramacionIII** herede de ella esos elementos, y sume los que la superclase no tenga. Por ejemplo, podría tener un atributo que indique en qué curso se encuentra matriculado, o una lista con los trabajos prácticos que ha realizado.

Herencia simple | Superclase

La definición de la superclase **Persona** no posee ninguna característica particular. Se define como una clase más:

Superclase Persona

```
class Persona: # Clase que representa una persona.
    def __init__(self, identificacion, nombre, apellido, dni):
    #Constructor de Persona
        self.id = identificacion # Atributo de instancia
        self.nombre = nombre # Atributo de instancia
        self.apellido = apellido # Atributo de instancia
        self.dni = dni # Atributo de instancia

    # Método str:
    def __str__(self):
        return f"{self.id} - DNI - {self.dni} {self.apellido}, {self.nombre}"
```

Herencia simple | Superclase

Esta clase Persona, que será la superclase de nuestro ejemplo, posee varios atributos de instancia y un método. A pesar de que la utilizaremos como superclase, aún es posible instanciar objetos Persona:

Programa principal

```
# Programa principal:
p1 = Persona(3, "Carlos", "Kleiber", 32456812)
print(p1)
```

Terminal

```
3 - DNI - 32456812 Kleiber, Carlos
```

El **método __str__** de la **superclase Persona** muestra la cadena de texto que contiene una representación de los atributos del objeto p1. Este método también será heredado por la **subclase AlumnoProgramacionIII**.

Herencia simple | Subclase

La definición de la **subclase AlumnoProgramacionIII** incluye en su declaración el argumento “Persona”, que hace referencia a la superclase. Utiliza, en su constructor, el método constructor de la superclase. Vemos que agrega un nuevo atributo de clase, “curso”, que no está presente en la **superclase Persona**. La **subclase AlumnoProgramacionIII** hereda el método __str__ de la superclase.

Subclase AlumnoProgramacionIII

```
class AlumnoProgramacionIII(Persona): # Parámetro: superclase
# Clase que representa a un alumno de programacion III
    def __init__(self, identificacion, nombre, apellido, dni, curso):
        # Constructor de Alumno
        # Invocamos al constructor de la superclase:
        Persona.__init__(self, identificación, nombre, apellido, dni)
        # agregamos el atributo propio del alumno
        self. Curso = curso
```

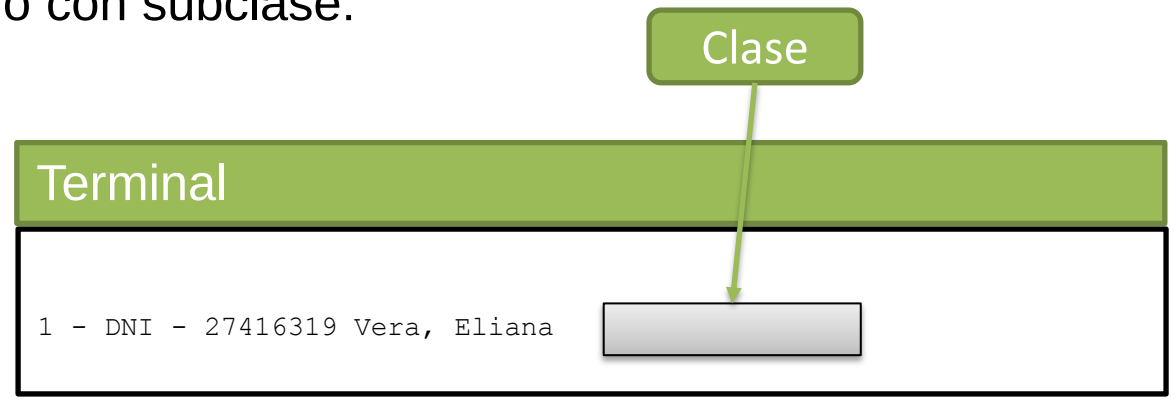
Herencia simple | Subclase

En este punto, tenemos la superclase y la subclase definidas. Vimos que podíamos instanciar objetos de la superclase. Y ahora también podemos hacerlo con subclase:

Programa principal

```
a1 = AlumnoProgramacionIII(1, "Eliana", "Vera", 27416319,
"Full Stack")
print(a1)
```

Terminal



Al imprimir el objeto **a1**, estamos usando el **método __str__** de la superclase **Persona**, que ha sido heredado por la subclase **AlumnoProgramacionIII**. Es por ello que el curso al que pertenece no se muestra. Pero es posible agregar en la subclase un nuevo método __str__ que solucione este problema.

Herencia simple | Subclase

Agregamos a la **subclase AlumnoProgramacionIII** su propio método __str__ , que reemplazará al heredado de la superclase:

Método __str__ de la Subclase AlumnoProgramacionIII

```
# Método str propio de la subclase AlumnoProgramacionIII:
def __str__(self):
    cadena = f"{self.id} - DNI - {self.dni} \n"
    cadena += f"{self.apellido},{self.nombre} \n"
    cadena += f"Carrera: {self.curso}"
    return cadena
```

Programa principal

```
p1 = Persona(3, "Carlos", "Kleiber", 32456812)
print(p1)
a1 = AlumnoProgramacionIII(1, "Eliana", "Vera", 27416319,
"Full Stack")
print(a1)
```

Terminal

```
3 - DNI - 32456812 Kleiber, Carlos
1 - DNI - 27416319
Vera,Eliana
Carrera: Full Stack
```

Herencia simple y herencia múltiple

Los casos que hemos analizado tienen lo que se conoce como **herencia simple**, que tiene lugar cuando una clase derivada (subclase) hereda atributos y métodos de una única clase base (superclase).
La **herencia múltiple** ocurre cuando una subclase deriva de dos o más clases base. Al escribir el código de la subclase, las superclases de las que heredará métodos y atributos se indican de la misma forma, separando cada una con una coma.
Por ejemplo, una subclase **Hijo** podría heredar de dos superclases: **Padre** y **Madre**.

HERENCIA MULTIPLE

Vemos como el objeto hijo1 ha heredado los métodos de las superclases Padre y Madre.
Podemos utilizar hijo1 los métodos propios o los de las superclases. En caso de que ambas superclases tengan un método con el mismo nombre, se hereda el que se escriba
primero en la declaración (Padre en el ejemplo):

Herencia múltiple:

```
class Padre: # Superclase 1
    def llevar(self):
        print("Papá me lleva al colegio.")

class Madre: # Superclase 2
    def programar(self):
        print("Mamá programa en Python.")

class Hijo(Padre, Madre): # Subclase
    def amar(self):

print("Quiero a mis padres")
hijo1 = Hijo() # Instanciamos hijo1
hijo1.llevar() # Papá me lleva al colegio.
hijo1.programar()# Mamá programa en Python.
hijo1.amar() # Quiero a mis padres
```

Clases abstractas

Un concepto importante en Programación Orientada a Objetos es el de las clases abstractas. Son clases en las que se pueden definir tanto métodos como propiedades, pero que no pueden ser instanciadas directamente. Solamente se pueden usar para construir subclases
(como si fueran moldes), permitiendo tener una única implementación de los métodos compartidos. Esto evita la duplicación de código.
Las clases abstractas definen una interfaz común para las subclases.
Proporcionan atributos y métodos comunes para todas las subclases evitando así la necesidad de duplicar código, imponiendo además los métodos que deben ser implementados para evitar inconsistencias entre las subclases.

CLASES ABSTRACTAS

Propiedades de las clases abstractas:

- ✓ No pueden ser instanciadas, simplemente proporcionan una interfaz para las subclases derivadas evitando así la duplicación de código.
- ✓ No es obligatorio que tengan una implementación de todos los métodos necesarios. Pudiendo ser estos abstractos. Los métodos abstractos son aquellos que solamente tienen una declaración, pero no una implementación detallada de las funcionalidades.
- ✓ Las clases derivadas de las clases abstractas deben implementar necesariamente todos los métodos abstractos para poder crear una clase que se ajuste a la interfaz definida. En el caso de que no se defina alguno de los métodos no se podrá crear la clase.

Clases abstractas | Creación

Para poder crear clases abstractas en Python es necesario importar la clase ABC y el decorador abstractmethod del módulo abc (Abstract Base Classes).

Clase abstracta

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod

    def mover(self):
        pass # permite “pasar” sin contenido

perro = Animal()
#TypeError: Can't instantiate abstract class
Animal with abstract method mover
```

Si se intenta crear una instancia de la clase Animal, Python no lo permite.

Si la clase no hereda de ABC o contiene por lo menos un método abstracto, Python permitirá crear instancias de la clase.

Clases abstractas

Las **subclases** tienen que implementar **todos** los métodos abstractos definidos en la clase abstracta, o Python no permitirá instanciar la clase hija.

Desde los métodos de las subclases se pueden utilizar las implementaciones de la clase abstracta con el comando **super()** seguido del nombre del método. Por ejemplo:

Clase abstracta

```
def emitir_sonido(self):
    super().emitir_sonido()
```

Clase abstracta

```
class Animal(ABC):
    @abstractmethod
    def mover(self):
        pass

    @abstractmethod
    def comer(self):
        print("El animal come")
```

CLASES ABSTRACTAS | EJEMPLO

Crearemos una clase abstracta Animal de la que heredarán métodos dos subclases (Gato y Perro).

- ✓ No podemos instanciar directamente objetos de la clase Animal, ya que se trata de una clase abstracta.
- ✓ La clase Animal debe heredar de ABC, e implementar solo métodos abstractos, para que funcione como clase abstracta.
- ✓ Gato y Perro completan los métodos abstractos de Animal, cada uno con la característica propia de los objetos de esas clases.

CLASES ABSTRACTAS | EJEMPLO

Clase abstracta Animal

```
from abc import ABC,
abstractmethod
class Animal(ABC):
    @abstractmethod
    def mover(self):
        pass
    @abstractmethod
    def emitir_sonido(self):
        print("Animal dice: ",
end="")
```

Subclase Gato

```
class Gato(Animal):
    def mover(self):
        print("El gato
se   mueve.")

    def emitir_sonido(self):
        super().emitir_sonido()
        print("Miau!")
```

Subclase Perro

```
class Perro(Animal):
    def mover(self):
        print("El perro se está
moviendo.")

    def emitir_sonido(self):
        super().emitir_sonido()
        print("Guau, Guau!")
```

Los métodos **mover()** y **emitir_sonido()** de las subclases **Gato** y **Perro** completan o complementan el código disponible en los métodos abstractos homónimos de la clase abstracta **Animal**. Solo nos resta instanciar objetos y ver cómo se comportan.

CLASES ABSTRACTAS | EJEMPLO

Programa principal

```
g1 = Gato()
g1.mover()
g1.emitir_sonido()

p1 = Perro()
p1.mover()
p1.emitir_sonido()
```

Consola

```
El gato se mueve.
Animal dice: Miau!
El perro se está moviendo.
Animal dice: Guau, Guau!
```

g1.mover() ejecuta el código correspondiente al método de la clase abstracta, y luego el propio de la subclase Gato. Lo mismo ocurre con **emitir_sonido().**
Cómo es lógico, la subclase **Perro** hace lo propio, y modifica ambos métodos abstractos de la clase abstracta Animal.

Polimorfismo

El polimorfismo es uno de los pilares básicos en la Programación Orientada a Objetos. El término polimorfismo tiene origen en las palabras poly (muchos) y morfo (formas), y aplicado a la programación hace referencia a que los objetos pueden tomar diferentes formas.
Esto significa que objetos de diferentes clases pueden ser accedidos utilizando la misma interfaz, mostrando un comportamiento distinto (tomando diferentes formas) según cómo sean accedidos.

Polimorfismo

En lenguajes de programación como Python, que tiene tipado dinámico, el polimorfismo va muy relacionado con el duck typing (dak tiping) o “tipado del pato”, que se resume en una frase: “Si camina como un pato y habla como un pato, entonces tiene que ser un pato”.
Al ser Python un lenguaje de tipado dinámico no es necesario que los objetos compartan una interfaz, simplemente basta con que todos tengan los métodos que se quieran utilizar.
Vamos, paso a paso, a crear un ejemplo.

Polimorfismo | Ejemplo

Definimos tres clases, Pato, Perro y Cerdo, todas con un método hablar().
Cada una de las clases implementa la versión que necesita del método, pero es importante ver que en todas tienen el mismo nombre.

Clases Pato, Perro y Cerdo

```
class Pato:
    def hablar(self):
        print("¡Cuac! "*3)

class Perro:
    def hablar(self):
        print("!Guau! "*3)

class Cerdo:
    def hablar(self):
    print("!Oink! "*3)
```

Polimorfismo | Ejemplo

Clase abstracta

```
def hacer_hablar(x):
    x.hablar()

#Creamos un pato y lo hacemos "hablar:"
mi_pato = Pato()
hacer_hablar(mi_pato)

#Creamos un perro y lo hacemos "hablar:"
mi_perro = Perro()
mi_perro.hablar()

#Creamos un cerdo y lo hacemos "hablar:"
mi_cerdo = Cerdo()
mi_cerdo.hablar()
```

El programa principal define una función que recibe un objeto, y utiliza (sin importar cual sea) su método hablar().
Esto es posible gracias al polimorfismo:
no es necesario escribir código para acceder a un mismo atributo o método de objetos de distinta clase, cuando estos tienen el mismo nombre.

Diagrama de clases

Un diagrama de clases en Lenguaje Unificado de Modelado (UML) es un tipo de diagrama de estructura estática que describe la estructura de un sistema mostrando las clases del sistema, sus atributos, operaciones (o métodos), y las relaciones entre los objetos.
UML proporciona mecanismos para representar los miembros de la clase, como atributos y métodos, así como información adicional sobre ellos.

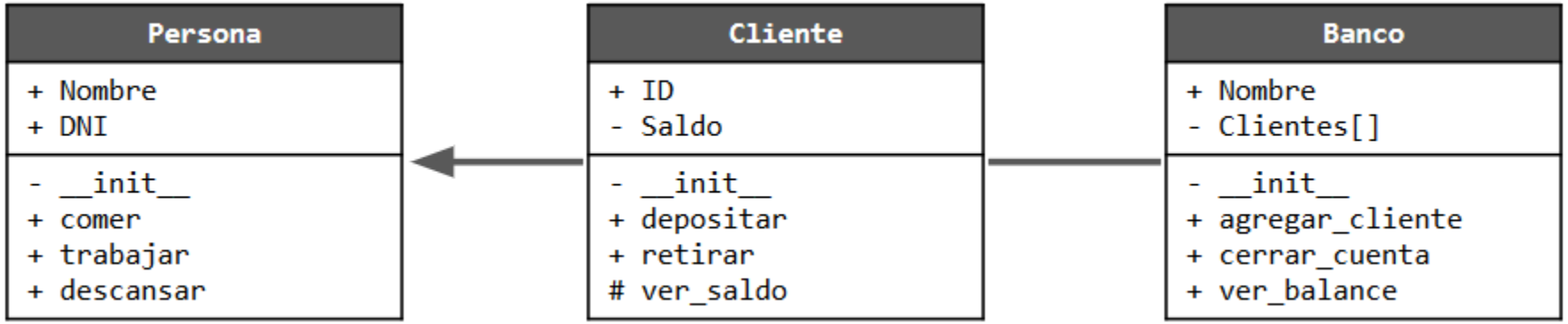
Diagramas de clases

En los lenguajes orientados a objetos, los algoritmos se expresan definiendo 'objetos' y haciendo que los objetos interactúen entre sí. Los lenguajes orientados a objetos dominan el mundo de la programación porque modelan los objetos del mundo real, y la relación entre sus clases, atributos y métodos puede modelarse mediante UML.

La figura de clase en sí misma consiste en un rectángulo de tres filas. La fila superior contiene el nombre de la clase, la fila del centro contiene los atributos de la clase y la última expresa los métodos o las operaciones que la clase puede utilizar.

Diagramas de clases

Cada atributo y método de una clase está ubicado en una línea separada:



La relación de herencia se simboliza mediante una línea de conexión recta con una punta de flecha cerrada que señala a la superclase. La relación predeterminada entre clases se representa mediante una línea recta.

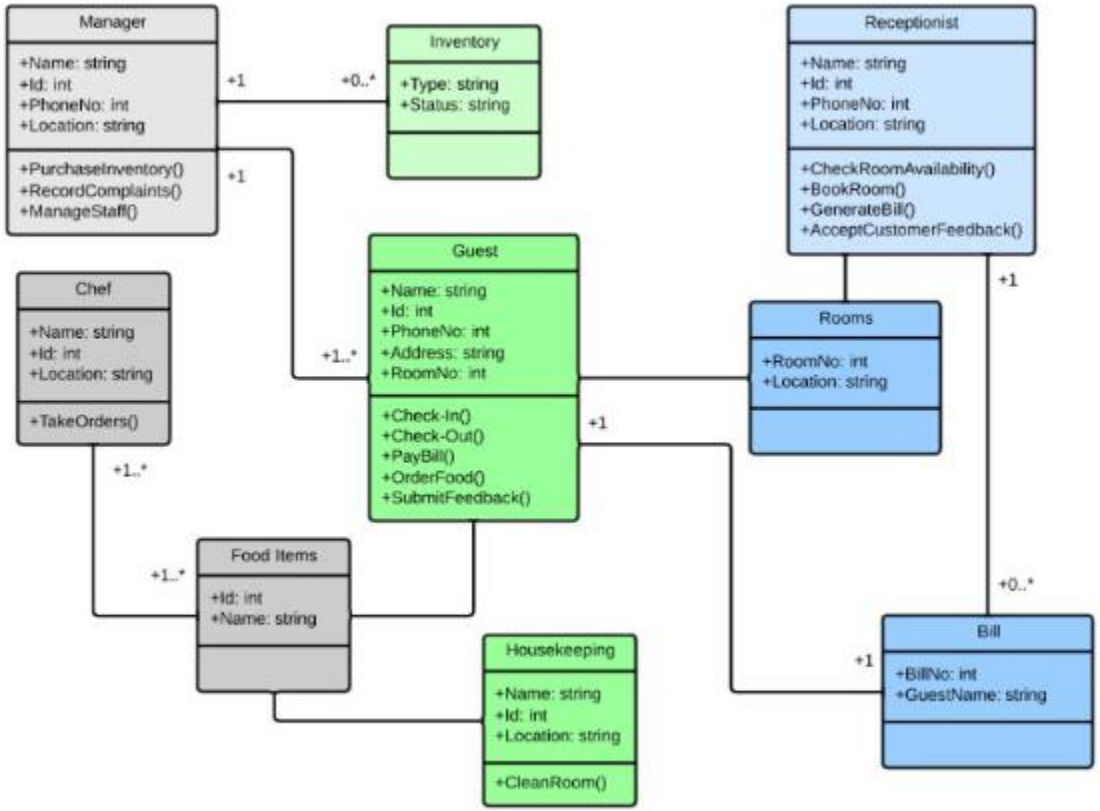
Diagramas de clases

Para especificar la visibilidad de un miembro de la clase (es decir, cualquier atributo o método), se coloca uno de los siguientes signos delante de ese miembro:

- ✓ Público (+)
- ✓ Privado (-)
- ✓ Protegido (#)

El siguiente ejemplo proporciona un diagrama de clases que representa gráficamente las relaciones entre las clases de un sistema administrativo hotelero.

Diagramas de clases



El diagrama de la izquierda, a pesar de su complejidad, nos proporciona una gran cantidad de información sobre las clases, sus métodos y atributos, y sus relaciones en un formato de fácil lectura. +info

MATERIAL EXTRA

Artículos de interés

Material extra:

- ✓ ¿Qué es el Duck Typing en Python?
- ✓ Herencia, clases abstractas y polimorfismo
- ✓ Diagramas de clase

Videos:

- ✓ Herencia y herencia múltiple en Python
- ✓ Polimorfismo en Python
- ✓ Clases abstractas