ISPC
INSTITUTO SUPERIOR
POLITÉCNICO CÓRDOBA

TECNICATURA SUPERIOR EN

Desarrollo Web y Aplicaciones Digitales

Programación II

Módulo

Dirección General de EDUCACIÓN TÉCNICA Y FORMACIÓN PROFESIONAL
Ministerio de EDUCACIÓN
GOBIERNO DE LA PROVINCIA DE CÓRDOBA
entre todos
Hacemos

ÍNDICE

Programación Orientada a Objetos

La programación orientada a objetos (abreviada de ahora en más como POO), es un conjunto de reglas y principios de programación (o sea, un paradigma de programación) que busca representar las entidades u objetos del dominio (o enunciado) del problema dentro de un programa, de la forma más natural posible.

En el paradigma de programación tradicional o estructurado, que es el que hemos usado hasta aquí, el programador busca identificar los procesos (en forma de subproblemas o módulos) a fin de obtener los resultados deseados. Y esta forma de proceder no es en absoluto incorrecta: la estrategia de descomponer un problema en subproblemas es una técnica elemental de resolución de problemas que los programadores orientados a objetos siguen usando dentro de este nuevo paradigma. Entonces, ¿a qué viene el paradigma de la POO?

La programación estructurada se basa en descomponer procesos en subprocesos y programando cada uno como rutina, función, o procedimiento (todas formas de referirse al mismo concepto) Sin embargo, esta forma de trabajar resulta difícil al momento de desarrollar sistemas realmente grandes o de mucha complejidad. Es decir que, no alcanza dividir el problema a resolver en subproblemas cuando el sistema es muy complejo dado que, se torna difícil de visibilizar y hasta realizar las más pequeñas modificaciones del código fuente lo que hace, casi imposible replantear el sistema para agregar nuevas funcionalidades o resolver fallos.

La POO significa una nueva visión en la forma de programar, buscando aportar claridad y naturalidad en la manera en que se plantea un problema. Ahora, el objetivo no es identificar los procesos sino que además, se trata de identificar actores: las entidades u objetos que aparecen en el escenario o el dominio del problema. En este punto es importante agregar que, esos objetos no sólo tienen datos asociados sino que también, presentan comportamientos capaces de ejecutar.

Un buen ejemplo son los videojuegos. Por ejemplo, el juego Mario Bros, tiene varios personajes, Mario Bros. y Luigi.

Figura 1: Luigi y Mario Bros

Estos personajes son objetos virtuales y tienen sus propias características y comportamientos ya que, representan objetos del mundo real. Es decir que, estos personajes son capaces de saltar y caminar e interactuar con otros objetos tales como: el escenario y las monedas a medida que avanza la partida.

Ventajas de la POO

- Es la forma natural de entender la realidad y por ende, más fácil de representar/modelar.
- Permite comprender el código fuente y resolver más fácilmente problemas de gran complejidad.
- Facilita el mantenimiento y escalabilidad de las aplicaciones.
- Es más adecuado para la construcción de entornos GUI.
- Fomenta la reusabilidad y provee un gran impacto sobre la productividad y confiabilidad.

Objetos

Para comprenderlo veamos la siguiente imagen y luego intentemos responder ¿Cuáles son los objetos que se pueden abstraer para ver televisión? ¿Cómo podrías describirlos?

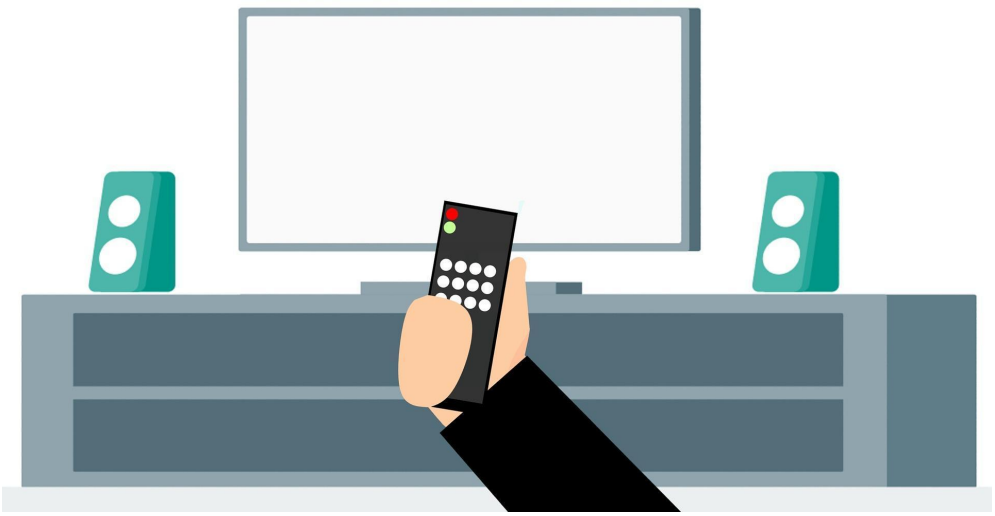

Figura 2: Ver la televisión.

En el problema planteado se pueden identificar tres elementos: la persona, el televisor y el control remoto. Cada elemento posee sus propias características y comportamientos. En la POO a estos elementos se los conoce bajo el nombre de OBJETOS, a las características o estados que identifican a cada objeto ATRIBUTOS y, a los comportamientos o acciones, MÉTODOS.

Entonces podemos resumir:

ELEMENTO	DESCRIPCIÓN
Persona	Tiene sus propios atributos: Apellido, Nombre, Altura, género, Color ojos, Cabello, etc. Y tiene un comportamiento: Ver , escuchar, hablar, correr, saltar, etc.
Control Remoto	Tiene sus propios atributos: Tamaño, color, tipo, batería, etc. Y tiene un comportamiento: Enviar señal, codificar señal, cambiar canal, aumentar volumen, ingresar a menú, prender TV, etc.
Televisor	Tiene sus propios atributos: pulgadas, tipo, número parlantes, marca , etc. Y tiene un comportamiento: decodificar señal, prender, apagar, emitir señal, emitir audio, etc.

Del ejemplo anterior podemos inferir el concepto de objeto:

Un objeto es un elemento que contiene un estado (atributos) y un comportamiento (métodos) que realiza por sí solo o bien, interactuando con otros objetos.

Sin embargo, además de las características (atributos) y acciones (métodos)… ¿Qué otras características podrías mencionar? Observa la siguiente imagen para analizar..

Figura 4: Televisor

De acuerdo a la imagen anterior podemos decir que un objeto además, se identifica por un nombre o identificador único que lo diferencia de los demás (en este caso puede ser el nro de serie), un estado (encendido, apagado), un tipo (televisor), nos abstrae dejándonos acceder sólo a las funciones encendido, apagado, cambiar canal, etc. y, por supuesto tiene un tiempo de vida.

En base a lo expuesto anteriormente podemos expresar las características generales de los objetos en POO:

- Se identifican por un nombre o identificador único que los diferencia de los demás.
- Poseen estados.
- Poseen un conjunto de métodos.
- Poseen un conjunto de atributos.
- Soportan el encapsulamiento (nos deja ver sólo lo necesario).

- Tienen un tiempo de vida.
- Son instancias de una clase (es de un tipo).

Para hacer eso, los lenguajes de programación orientados a objetos (como TypeScripts) usan descriptores (plantillas) de entidades conocidas como clases.

Clases

Una clase es la descripción de una entidad u objeto de forma tal que pueda usarse como plantilla para crear muchos objetos que respondan a dicha descripción. Para establecer analogías, se puede pensar que una clase se corresponde con el concepto de tipo de dato de la programación estructurada tradicional, y los objetos creados a partir de la clase (llamados instancias en el mundo de la POO) se corresponden con el concepto de variable de la programación tradicional. Así como el tipo es uno solo y describe la forma que tienen todas las muchas variables de ese tipo, la clase es única y describe la forma y el comportamiento de los muchos objetos de esa clase.

Para describir objetos que responden a las mismas características de forma y comportamiento, se definen las clases.

Veamos el siguiente ejemplo:

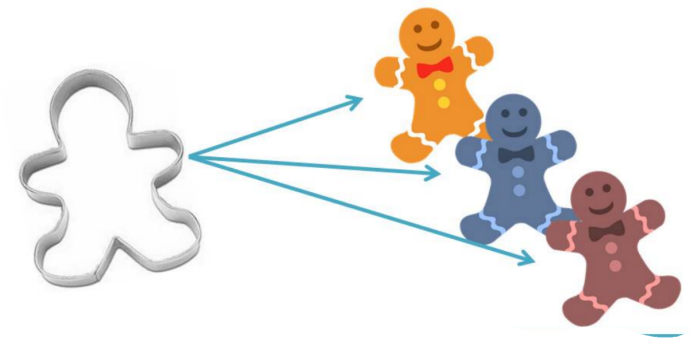

Figura 5: Molde de galletitas

Fuente:https://platzi.com/clases/1629-java-oop/21578-abstraccion-que-es-una-clase/

En el mismo podemos observar un molde para hacer galletitas (la clase) y el resultado que consiste en una o más galletas (objeto o instancia de clase).

Características generales de las clases en POO.

- Poseen un alto nivel de abstracción.
- Se relacionan entre sí mediante jerarquías.
- Los nombres de las clases deben estar en singular.

Representación gráfica de clases

La representación gráfica de una o varias clases se realiza mediante los denominados Diagramas de Clase. Para ello, se utiliza la notación que provee el Lenguaje de Modelación Unificado (UML, ver www.omg.org), a saber:

- Las clases se denotan como rectángulos divididos en tres partes. La primera contiene el nombre de la clase, la segunda contiene los atributos y la tercera los métodos.

- Los modificadores de acceso a datos y operaciones, a saber: público, protegido y privado; se representan con los símbolos +, # y – respectivamente, al lado derecho del atributo. (+ público, # protegido, - privado).

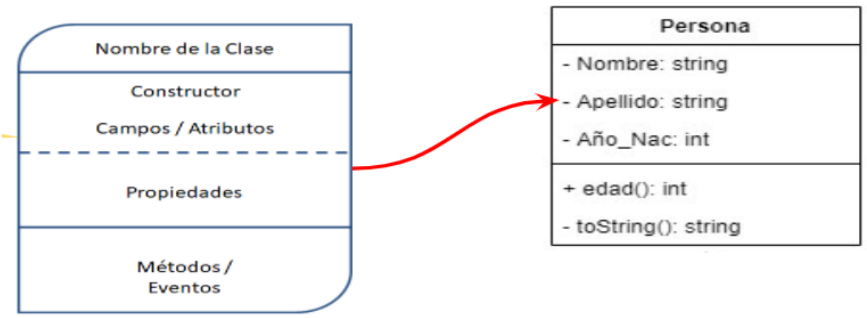

Figura 6: Estructura de una clase (Diagrama de clases)

Relación entre clases y objetos

Algorítmicamente, podemos entender a las clases como PLANTILLAS que describen objetos. Su objeto es definir nuevos tipos conformados por atributos y operaciones. Es decir que, las clases son una especie de molde de fábrica, en base al cual son construidos los objetos.

Por su parte, los objetos son instancias individuales de una clase. Durante la ejecución de un programa sólo existen los objetos, no clases.

A continuación enumeramos algunos puntos a saber:

- La declaración de una variable de una clase NO crea el objeto.
- La creación de un objeto, debe ser indicada explícitamente por el programador (instanciación). Es decir, así como inicializamos las variables

con un valor, deberemos iniciar los objetos sólo que, para los objetos lo realizaremos a través de un método especial. El CONSTRUCTOR.

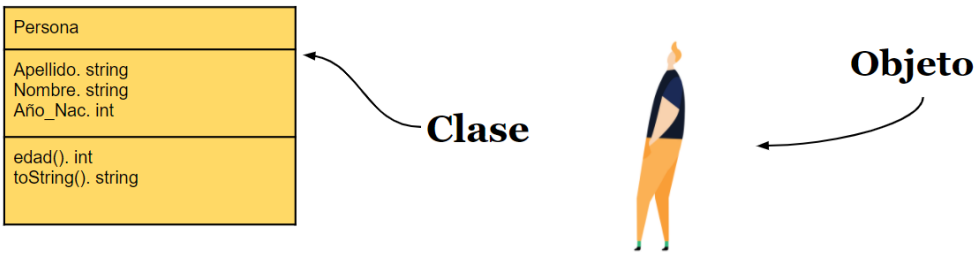

Figura 7: Relación entre clases y objetos.

Clases e instancias en Python

Clases

Una clase en Python no es más que una secuencia de símbolos (o caracteres). Esta secuencia de símbolos forma lo que se denomina el código fuente de la clase. Hay dos aspectos que determinan si una secuencia de símbolos es correcta o no: la sintaxis y la semántica.

Por un lado, la sintaxis de Python nos permite determinar de qué manera los símbolos del vocabulario pueden combinarse para escribir código fuente correcto mientras que la semántica, guarda una estrecha relación con lo que es código hace permitiendo así determinar el significado de la secuencia de símbolos para que se lleve a cabo la acción por la computadora.

Así, por ejemplo, en el lenguaje natural son las reglas de la sintaxis las que nos permiten determinar que la frase “programa hombre el autómata” no es correcta y, las reglas semánticas nos posibilitan detectar que la siguiente frase “el

autómata programa al hombre” nos es correcta en términos de semántica (aunque en términos de sintaxis si lo es)

Sintaxis para la definición de clases en Python:

```
class <NombreDeLaClase>:
    <nombreDeAtributoDeClase> = <valor>
    def __init__(self,<parametro1>, <parametro2>, ...):
        self.<atributo1> = <parametro1>
        self.<atributo2> = <parametro2>
        .
        .
        .
        # Tantos atributos como se necesite.

    def <nombre_de_metodo>(self):
            # Código del método
            # Tantos métodos como se necesite
```

Como podemos observar, dentro de las clases podemos encontrar:

- Atributos: Son variables que se declaran dentro de la clase, y sirven para indicar la forma o características de cada objeto representado por esa

clase. Los atributos, de alguna manera, muestran lo que cada objeto es, o también, lo que cada objeto tiene.

En el ejemplo demostrado arriba, < nombreDeAtributoDeClase > es el nombre del atributo de la clase, y <valor> es el valor inicial del atributo de la clase. Los atributos de clase se definen fuera de cualquier método de la clase, y se pueden acceder desde cualquier instancia de la clase utilizando la sintaxis objeto.nombre_de_atributo_de_la_clase

Por otro lado, self.<atributo> es la sintaxis que se utiliza para definir un atributo de instancia en Python. Los atributos de instancia se definen dentro del método __init__ de la clase y se inicializan con los valores de los parámetros que se pasan al crear una instancia de la clase. Los atributos de instancia se pueden acceder desde cualquier método de la instancia utilizando la sintaxis self.atributo (https://www.freecodecamp.org/).

- Métodos: Son funciones, procedimientos o rutinas declaradas dentro de la clase, usados para describir el comportamiento o las acciones de los objetos descritos por esa clase. Los métodos, de alguna manera, muestran lo que cada objeto hace.

  La sintaxis es:

  ```
  class <NombreDeLaClase>:
      def <nombreDelMétodo>(self, <parametro1>, <parametro2> ...):
  ```

```
        # Código del método
```

En este ejemplo, <nombreDelMétodo> es el nombre del método, y <parametro1>, <parametro2>, etc. son los argumentos que recibe el método. El primer argumento de un método siempre es self, que se refiere como mencionamos previamente a la instancia de la clase que está siendo manipulada.

Es común (pero no obligatorio) que los atributos de la clase se declaren antes que los métodos. El conjunto de atributos y métodos de una clase se conoce como el conjunto de miembros de la clase.

Finalmente es importante mencionar que las clases no se construyen para que trabajen de manera aislada, la idea es que ellas se puedan relacionar entre sí, de manera que puedan compartir atributos y métodos sin necesidad de volver a escribirlos.

- Constructores: es un método especial que permite instanciar un objeto. Su nombre está definido por la palabra __init__. Puede recibir 1 a n parámetros.

  Sintaxis:

```
class <NombreDeLaClase>:
    def __init__(self,<parametro1>, <parametro2>, ...):
        # Código del constructor
```

El constructor suele usarse para la inicialización de los atributos del objeto a crear, sobre todo cuando el valor de éstos no es constante o incluye acciones más allá de una asignación de valor.

- Propiedades (getters y los setters). Las propiedades getter y setter en Python son métodos especiales que se utilizan para acceder y modificar los atributos privados de una clase (convencionalmente se utiliza el _ antes del nombre del atributo para indicar que es privado).

  Ejemplo:

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

def get_nombre(self):
    return self.__nombre

def set_nombre(self, nombre):
    self.__nombre = nombre

def get_edad(self):
    return self.__edad

def set_edad(self, edad):
    self.__edad = edad
```

```
    def imprimir_datos(self):
        print(f"Nombre: {self.get_nombre()}")
        print(f"Edad: {self.get_edad()}")

p = Persona("Juan", 30)
p.imprimir_datos() # Imprime "Nombre: Juan" y "Edad: 30"
p.set_nombre("Pedro")
p.set_edad(35)
p.imprimir_datos() # Imprime "Nombre: Pedro" y "Edad: 35"
```

Como puedes observar en este ejemplo, la clase Persona tiene dos atributos privados, __nombre y __edad. Para acceder y modificar estos atributos, se definen los métodos get_nombre, set_nombre, get_edad y set_edad. Los métodos get_nombre y get_edad son los getters, que se utilizan para acceder a los atributos privados, mientras que los métodos set_nombre y set_edad son los setters, que se utilizan para modificarlos.

Por otro lado, es posible definir las propiedades getter y setter de una forma más sencilla utilizando los decoradores @property y @x.setter como se muestra en el ejemplo:

```
class MyClass:
    def __init__(self):
        self._x = None
    @property
```

```
    def x(self):
        return self._x
    @x.setter
    def x(self, value):
        self._x = value
```

En este ejemplo, x es una propiedad que encapsula el atributo _x (privado por convención). La propiedad x se define utilizando el decorador @property, que indica que el método x es un getter. El setter de la propiedad se define utilizando el decorador @x.setter, que indica que el método x es un setter.

- Modificadores de acceso: En Python no existen modificadores de acceso como en otros lenguajes de programación orientados a objetos. Todos los miembros de una clase en Python son públicos por defecto. Sin embargo, se utiliza una convención de nomenclatura para indicar que un atributo o método de una clase no debería ser accedido desde fuera de la clase (ej. __atributoPrivado).

Instancias

En Python, una instancia de clase es un objeto que se crea a partir de una clase. Para crear el objeto, utilizamos la sintaxis **nombre_de_clase().**

Sintaxis para instanciar objetos:

```
<nombre_objeto>= <Nombre_de_Clase>(<parámetros>)
```

Ejemplo:

```
objeto = MiClase(1, 2)
```

Sintaxis para inicializar un objeto:

Hay 3 maneras de inicializar un objeto. Es decir, proporcionar datos a un objeto.

1. Por referencia a variables

Ejemplo:

```
persona = new Persona();
persona.apellido="Rosas";
persona.nombre ="Maria";
```

2. Por medio del constructor de la clase:

Ejemplo:

```
persona= Persona("Maria","Rosas");
```

3. Por medio de la propiedad setter:

Ejemplo:

```
persona= Persona();
persona.Apellido="Rosas";
persona.Nombre ="Maria";
```

Recomendaciones

Aunque cada programador puede definir su propio estilo de programación, una buena práctica es seguir el estilo utilizado por los diseñadores del lenguaje pues, de seguir esta práctica será mucho más fácil analizar el fuente de terceros y, a su vez, que otros programadores analicen y comprendan nuestro fuente.

- Evitar en lo posible líneas de longitud superior a 80 caracteres.
- Identar los bloques de código.
- Utilizar identificadores nemotécnicos, es decir, utilizar nombres simbólicos adecuados para los identificadores lo suficientemente autoexplicativos por

sí mismos para dar una orientación de su uso o funcionalidad de manera tal que podamos hacer más claros y legibles nuestros códigos.

- Los identificadores de clases, módulos, interfaces y enumeraciones deberán usar PascalCase.
- Los identificadores de objetos, métodos, instancias, constantes, propiedades y métodos de los objetos deberán usar camelCase.
- Utilizar comentarios con moderación (recuerda que si sientes la necesidad de comentar el código, posiblemente tu código no sea limpio), éstos deberán seguirán un formato general de fácil portabilidad y que no incluya líneas completas de caracteres repetidos. Los que se coloquen dentro de bloques de código deben aparecer en una línea independiente identada de igual forma que el bloque de código que describen.

Fundamentos del enfoque orientado a objetos (EOO)

El Enfoque Orientado a Objeto se basa en cuatro principios que constituyen la base de todo desarrollo orientado a objetos. Estos principios son:

- Jerarquías (herencia, agregación y composición)
- Abstracción
- Encapsulamiento
- Modularidad

Otros elementos a destacar (aunque no fundamentales) son:

- Polimorfismo
- Tipificación
- Concurrencia
- Persistencia.

Los cuales se describirán a continuación:

Jerarquías

Las clases no se diseñan para que trabajen de manera aislada, el objeto es que se puedan relacionar entre sí de manera que puedan compartir atributos y métodos y así resolver un problema.

La capacidad de establecer jerarquías entre las clases es una característica que hace que la programación orientada a objetos sea diferente a la programación tradicional (estructurada). Esto se debe principalmente a que el código existente se puede escalar y reutilizar sin tener que volver a escribirlo cada vez.

Herencia

En Programación Orientada a Objetos la herencia es un proceso mediante el cual se puede crear una clase hija que hereda de una clase padre, compartiendo sus métodos y atributos. Además de ello, una clase hija puede sobreescribir los métodos o atributos, o incluso definir unos nuevos. Para crear una clase hija se debe pasar como parámetro la clase de la que se quiere heredar.

Por ejemplo en python, se puede definir una clase padre llamada Auto y crear una clase hija llamada Convertible que hereda de Auto de la siguiente manera:

```python
# Definimos una clase padre
class Auto:
    pass
# Creamos una clase hija que hereda de la padre
class Convertible(Auto):
    pass
```

En resumen, la herencia es un mecanismo por el cual podemos definir una nueva clase B en términos de otra clase A ya definida, pero de forma que la clase B obtiene todos los miembros definidos en la clase A sin necesidad de hacer una redeclaración explícita. El sólo hecho de indicar que la clase B hereda (o deriva) desde la clase A, hace que la clase B incluya todos los miembros de A como propios (a los cuales podrá acceder en mayor o menor medida de acuerdo al modificador de acceso [public, private] que esos miembros tengan en A).

Es decir que la herencia permite la definición de un nuevo objeto a partir de otros, agregando las diferencias entre ellos (Programación Diferencial) y evitando así la repetición de código y permitiendo la reusabilidad.

Cuando la clase B hereda de la clase A, se dice que hay una relación de herencia entre ellas, y se modela en UML con una flecha continua terminada en punta cerrada. La flecha parte de la nueva clase (o clase derivada) que sería B en nuestro ejemplo, y termina en la clase desde la cual se hereda (que es A en nuestro caso):

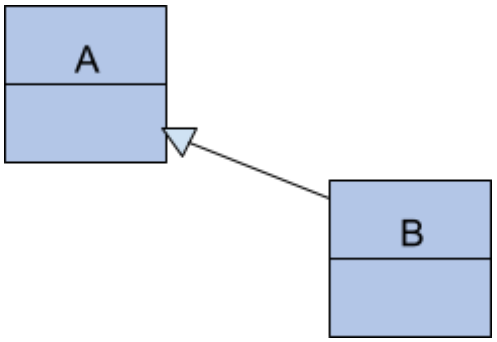

Figura 8: (UML) Diagrama de clases: Representación de la Herencia

La clase desde la cual se hereda, se llama super clase, y las clases que heredan desde otra se llaman subclases o clases derivadas: de hecho, la herencia también se conoce como derivación de clases.

Una jerarquía de clases es un conjunto de clases relacionadas por herencia. La clase en la cual nace la jerarquía que se está analizando se designa en general como clase base de la jerarquía. La idea es que la clase base reúne en ella características que son comunes a todas las clases de la jerarquía, y que por lo tanto todas ellas deberían compartir sin necesidad de hacer redeclaraciones de esas características. El siguiente gráfico muestra una jerarquía de clases:

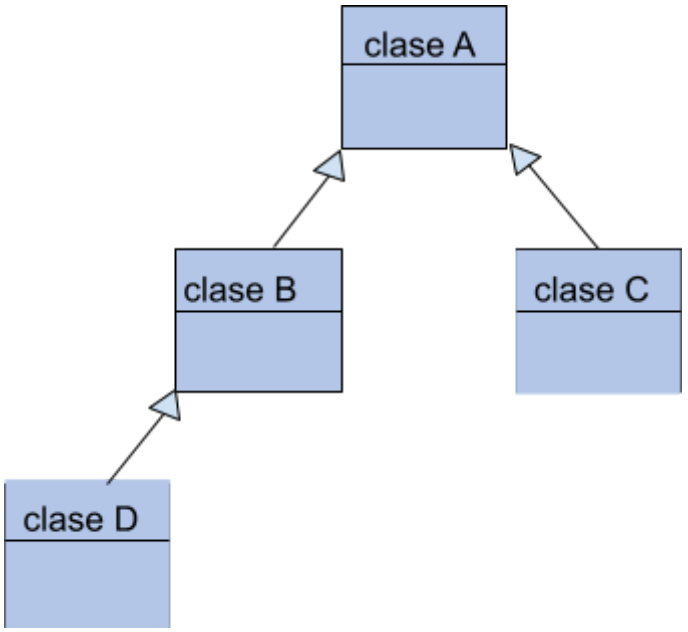

Figura 9: (UML) Diagrama de clases: Jerarquía de Clases

En esta jerarquía, la clase base es “Clase A”. Las clases “Clase B” y “Clase C” son derivadas directas de “Clase A”. Note que “Clase D” deriva en forma directa desde “Clase B”, pero en forma indirecta también deriva desde “Clase A”, por lo tanto todos los elementos definidos en “Clase A” también estarán contenidos en “Clase D”. Siguiendo con el ejemplo, “Clase B” es super clase de “Clase D”, y “Clase A” es super clase de “Clase B” y “Clase C”.

En general, se podrían definir esquemas de jerarquías de clases en base a dos categorías o formas de herencia:

Herencia simple: Si se siguen reglas de herencia simple, entonces una clase puede tener una y sólo una superclase directa. El gráfico anterior es un ejemplo de una jerarquía de clases que siguen herencia simple. La Clase D tiene una sola superclase directa que es Clase B. No hay problema en que a su vez

esta última derive a su vez desde otra clase, como Clase A en este caso. El hecho es que en herencia simple, a nivel de gráfico UML, sólo puede existir una flecha que parta desde la clase derivada hacia alguna superclase.

Herencia múltiple: Si se siguen reglas de herencia múltiple, entonces una clase puede tener tantas superclases directas como se desee. En la gráfica UML, puede haber varias flechas partiendo desde la clase derivada hacia sus superclases. El siguiente esquema muestra una jerarquía en la que hay herencia múltiple: note que Clase D deriva en forma directa desde las clases Clase B y Clase C, y esa situación es un caso de herencia múltiple. Sin embargo, note también que la relación que existe entre las Clase B y Clase C contra Clase A es de herencia simple; tanto Clase B como Clase C tienen una y sólo una superclase directa: Clase A.

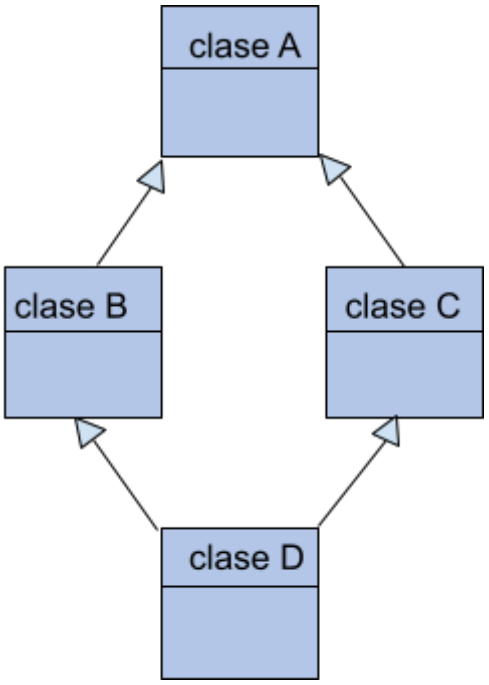

Figura 10: (UML) Diagrama de Clases: Herencia Múltiple

No todos los lenguajes orientados a objetos soportan (o permiten) la herencia múltiple: este mecanismo es difícil de controlar a nivel de lenguaje y en general se acepta que la herencia múltiple lleva a diseños más complejos que los que se podrían obtener usando sólo herencia simple y algunos recursos adicionales como la implementación de interfaces.

Veamos un ejemplo en Python:

Necesitamos crear dos clases que llamaremos Suma y Resta que derivan de una superclase llamada Operación como sigue:

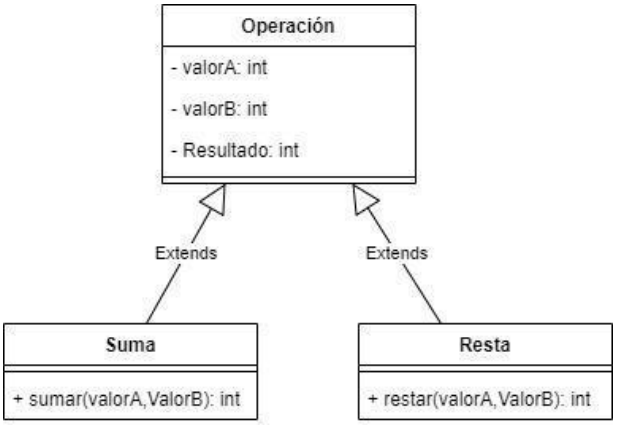

Figura 11: (UML) Diagrama de Clases de las operaciones operaciones Suma y Resta.

En python, seguir los siguientes pasos:

1- Definir la superclase operación como sigue:

```
class Operacion:

    def __init__(self, valorA, valorB):
        self.__valorA = valorA
```

```
        self.__valorB = valorB

    @property
    def valorA(self):
        return self.__valorA

    @valorA.setter
    def valorA(self, valor):
        self.__valorA = valor

    @property
    def valorB(self):
        return self.__valorB

    @valorB.setter
    def valorB(self, valor):
        self.__valorB = valor
```

2- Luego, extender las subclases suma y resta como sigue:

```
class Suma(Operacion):

    def __init__(self, valorA, valorB):
        super().__init__(valorA, valorB)

    def sumar(self):
        return self.valorA + self.valorB
```

```
class Resta(Operacion):

    def __init__(self, valorA, valorB):
        super().__init__(valorA, valorB)

    def restar(self):
        return self.valorA - self.valorB
```

3- Crear instancias de la clase suma y resta:

```
# Crear una instancia de la clase Suma
mi_suma = Suma(10, 5)

# Llamar al método sumar
resultado = mi_suma.sumar()

# Imprimir el resultado en consola
print("El resultado es:", resultado)
```

```
# Crear una instancia de la clase Resta
mi_resta = Resta(10, 5)

# Llamar al método restar
resultado = mi_resta.restar()
# Imprimir el resultado en consola
print("El resultado es:", resultado)
```

Agregación y Composición

Las jerarquías de agregación y composición son asociaciones entre clases del tipo “es parte de”.

Para comprenderlo mejor, pensemos en un auto y sus partes. Aquellas partes del auto que son elementales para su existencia y funcionamiento, como por ej. el motor, corresponden a las jerarquías de composición mientras que, aquellas partes que no lo son, ej. radio, corresponden a la jerarquía de agregación.

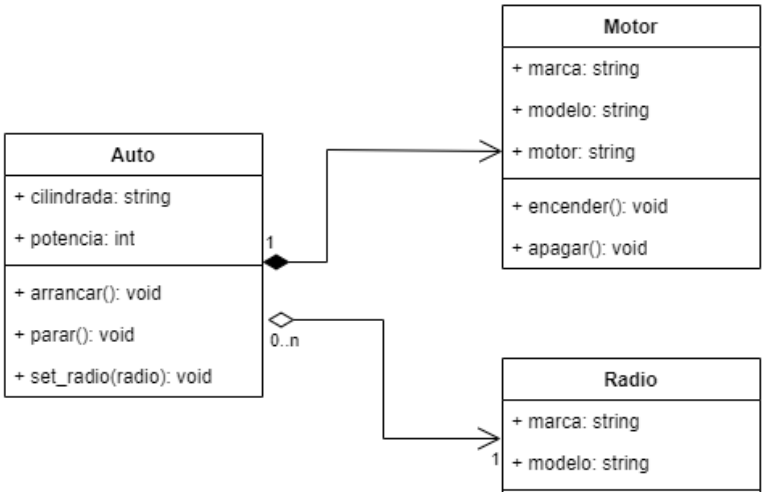

Figura 12: (UML) Diagrama de clases: Agregación y Composición

Ejemplo en Python:

```python
class Motor:
    def __init__(self, cilindrada, potencia):
        self.cilindrada = cilindrada
        self.potencia = potencia

    def encender(self):
        print("El motor se ha encendido.")

    def apagar(self):
```

```
        print("El motor se ha apagado.")

class Auto:
    def __init__(self, marca, modelo, motor):
        self.marca = marca
        self.modelo = modelo
        self.motor = motor

    def set_radio(self, radio):
        self.radio = radio

    def arrancar(self):
        print("El auto ha arrancado.")
        self.motor.encender()

    def parar(self):
        print("El auto se ha detenido.")
        self.motor.apagar()

class Radio:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
```

Como puedes observar en el ejemplo, la clase Auto está compuesta por un motor y puede o no tener una radio (observa que no se requiere la radio en el constructor) por ende, el motor es necesario para apagar y encender el vehículo siendo una parte elemental del auto (composición) pero no siendo así, con la radio (agregación). Es decir que, podríamos crear una instancia de Auto sin necesidad de agregar la radio. El mismo, sería capaz de seguir funcionando.

Abstracción

Una abstracción denota las características esenciales de un objeto (datos y operaciones), que lo distingue de otras clases de objetos. Decidir el conjunto correcto de abstracciones de un determinado dominio, es el problema central del diseño orientado a objetos.

"Una abstracción se centra en la visión externa de un objeto por lo tanto sirve para separar el comportamiento esencial de un objeto de su implementación. La decisión sobre el conjunto adecuado de abstracciones para determinado dominio es el problema central del diseño orientado a objetos. Se puede caracterizar el comportamiento de un objeto de acuerdo a los servicios que presta a otros objetos, así como las operaciones que puede realizar sobre otros objetos"

(http://ceaer.edu.ar/wp-content/uploads/2018/04/Apunte-Teorico-de-Programacion-OO.pdf)

Los mecanismos de abstracción usados en el EOO para extraer y definir las abstracciones son:

"1- La GENERALIZACIÓN. Mecanismo de abstracción mediante el cual un conjunto de clases de objetos son agrupados en una clase de nivel superior (Superclase), donde las semejanzas de las clases constituyentes (Subclases) son enfatizadas, y las diferencias entre ellas son ignoradas.

En consecuencia, a través de la generalización:

- La superclase almacena datos generales de las subclases
- Las subclases almacenan sólo datos particulares.

2- La ESPECIALIZACIÓN es lo contrario de la generalización. La clase Médico es una especialización de la clase Persona, y a su vez, la clase Pediatra es una especialización de la superclase Médico.

3- La AGREGACIÓN. Mecanismo de abstracción por el cual una clase de objeto es definida a partir de sus partes (otras clases de objetos). Mediante agregación se puede definir por ejemplo un computador, por descomponerse en: la CPU, la ULA, la memoria y los dispositivos periféricos. El contrario de agregación es la descomposición.

4- La CLASIFICACIÓN. Consiste en la definición de una clase a partir de un conjunto de objetos que tienen un comportamiento similar. La ejemplificación es lo contrario a la clasificación, y corresponde a la instanciación de una clase, usando el ejemplo de un objeto en particular”

(http://www.udla.edu.co/documentos/docs/Programas%20Academicos/Tecnologia%20en%20Informatica%20y%20sistemas/Compilados/Compilado%20Programacion%20II.pdf)

La clasificación es el medio por el que ordenamos, el conocimiento ubicado en las abstracciones.

En resumen, las clases y objetos deberían estar al nivel de abstracción adecuado. Ni demasiado alto ni demasiado bajo.

Encapsulamiento

El encapsulamiento se refiere a la técnica de denegar el acceso directo a los atributos y métodos internos de una clase desde el exterior. A diferencia de otros lenguajes de programación orientados a objetos, Python no tiene una sintaxis específica para definir miembros privados. Sin embargo, se puede simular el encapsulamiento precediendo los atributos y métodos con dos barras bajas

como indicando que son "especiales". Esto hace que los miembros sean más difíciles de acceder desde el exterior, pero aún pueden ser accedidos si se sabe cómo hacerlo.

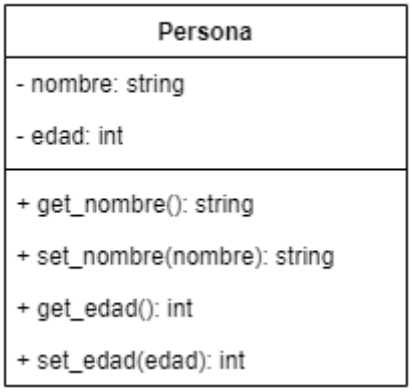

Figura 13: (UML) Diagrama de clases: Encapsulamiento.

A continuación, podemos ver el siguiente ejemplo:

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

    def get_nombre(self):
        return self.__nombre

    def set_nombre(self, nombre):
        self.__nombre = nombre

    def get_edad(self):
        return self.__edad

    def set_edad(self, edad):
        self.__edad = edad
```

En este ejemplo, la clase Persona tiene dos atributos privados, __nombre y __edad, que solo pueden ser accedidos desde dentro de la clase. También tiene cuatro métodos, dos para obtener (get) y dos para establecer (set) el valor de cada uno de los atributos. Los métodos get y set son necesarios porque los

atributos son privados y no pueden ser accedidos directamente desde fuera de la clase. De esta manera, el encapsulamiento protege los atributos de una clase de ser modificados incorrectamente desde el exterior.

Modularidad

La modularidad se refiere a la técnica de dividir un programa en módulos o componentes más pequeños y manejables, que pueden ser reutilizados en diferentes partes del programa o en diferentes programas. Cada módulo contiene una parte del programa completo y se puede importar a otro módulo para su uso. La modularidad en Python ayuda a mantener el código organizado, facilita la reutilización de código y hace que el proceso de desarrollo sea más eficiente.

A continuación, podemos ver el siguiente ejemplo:

Supongamos que tenemos un programa que necesita realizar algunas operaciones matemáticas como la suma, la resta, la multiplicación y la división. En lugar de escribir todas las funciones en un solo archivo, podemos dividir el programa en módulos separados para cada operación matemática. Podemos crear un archivo suma.py que contenga una función sumar que suma dos números, un archivo resta.py que contenga una función restar que resta dos números, un archivo multiplicacion.py que contenga una función multiplicar que multiplica dos números, y un archivo division.py que contenga una función dividir que divide dos números. Luego, podemos importar cada módulo en nuestro programa principal y usar las funciones según sea necesario.

```python
# suma.py
def sumar(a, b):
    return a + b

# resta.py
def restar(a, b):
    return a - b

# multiplicacion.py
def multiplicar(a, b):
    return a * b

# division.py
def dividir(a, b):
    if b == 0:
        return "No se puede dividir por cero"
    else:
        return a / b

# programa principal
import suma
import resta
import multiplicacion
import division

a = 10
b = 5

print("La suma de", a, "y", b, "es", suma.sumar(a, b))
print("La resta de", a, "y", b, "es", resta.restar(a, b))
print("La multiplicación de", a, "y", b, "es", multiplicacion.multiplicar(a, b))
print("La división de", a, "y", b, "es", division.dividir(a, b))
```

En este ejemplo, se han creado cuatro módulos separados, cada uno con una función que realiza una operación matemática diferente. Luego, se han importado los módulos en el programa principal y se han utilizado las funciones según sea necesario. Este enfoque de modularidad hace que el código sea más fácil de mantener, ya que cada módulo se puede modificar o actualizar por separado sin afectar al resto del programa. Además, si necesitamos realizar las mismas operaciones matemáticas en otro programa, podemos simplemente importar los módulos existentes en lugar de tener que volver a escribir todo el código.

Polimorfismo

El polimorfismo se refiere a la capacidad de un objeto de tomar diferentes formas o comportarse de diferentes maneras en función del contexto en el que se utiliza. En otras palabras, el polimorfismo permite que un objeto de una clase se comporte como si fuera de otra clase. Esto se puede lograr (entre otras formas) mediante el uso de herencia y sobrescritura de métodos..

Para comprenderlo mejor, muchas veces tenemos una subclase que hereda de una clase base por lo que obtiene todos los métodos, campos, propiedades y eventos de la clase base pero, en ocasiones vamos a necesitar un comportamiento diferente en las clases derivadas (o subclase).

Ejemplo:

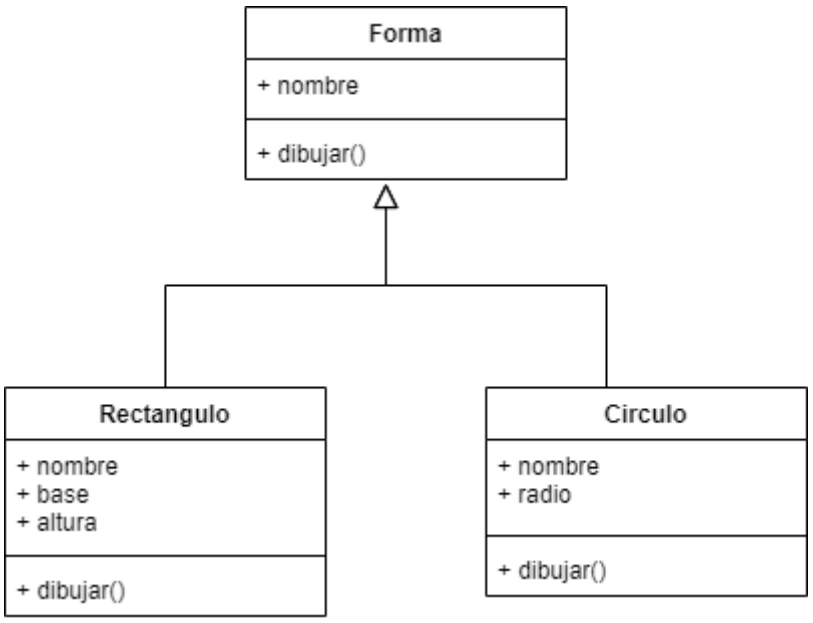

Figura 13. (UML) Diagrama de Clases: Polimorfismo.

Del diagrama de clases anterior podemos deducir que no será lo mismo dibujar un rectángulo que un círculo por lo que el comportamiento deberá ser distinto (polimorfismo).

Ejemplo de polimorfismo en base a herencia en python

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

    def dibujar(self):
        pass

class Rectangulo(Forma):
    def dibujar(self):
        return "dibuja un rectángulo!"

class Circulo(Form):
    def dibujar(self):
        return "dibuja un círculo!"



forma1 = Rectangulo("R1")
forma2 = Circulo("C1")

dibujar(forma1) # Imprime "dibuja un rectángulo!"
dibujar(forma2) # Imprime "dibuja un círculo!"
```

Tipificación

“Los tipos son la puesta en vigor de la clase de los objetos, de modo que los objetos de tipos distintos no pueden intercambiarse o, como mucho, pueden intercambiarse sólo de formas muy restringidas.” (http://ceaer.edu.ar/wp-content/uploads/2018/04/Apunte-Teorico-de-Programacion-OO.pdf)

Concurrencia

La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo.

La concurrencia permite a dos objetos actuar al mismo tiempo.

Persistencia

La persistencia es la propiedad de un objeto por la que su existencia trasciende el tiempo (el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (la posición del objeto varía con respecto al espacio de direcciones en el que fue creado).

Conserva el estado de un objeto en el tiempo y en el espacio.

Clases estáticas

En Python, una clase estática es una clase que no se puede heredar y no se puede instanciar directamente. Se puede implementar una clase estática haciendo que sus métodos y variables sean estáticos.

Para definir una clase estática debemos utilizar el decorador @classmethod como sigue:

```
class StaticClass:
    @classmethod
    def static_method(cls):
        print("Este es un método estático de la clase.")

StaticClass.static_method() # Imprime "Este es un método estático de la clase."
```

Como podemos observar arriba, se define una clase llamada StaticClass con un método llamado static_method, que se define como un método estático utilizando el decorador @classmethod. El método estático se puede llamar sin instanciar la clase como puedes observar en la última línea del ejemplo.

Interfaces

Una interface es un contrato que toda clase que la implemente debe cumplir. En Python, aunque no existe una keyword interface como en otros lenguajes de programación, es posible definir interfaces de dos formas: informales y formales.

Las interfaces formales se pueden definir en Python utilizando el módulo por defecto llamado ABC (Abstract Base Classes). Los ABC fueron añadidos a Python en la PEP3119 y permiten definir una forma de crear interfaces a través de metaclases. En estas interfaces se definen métodos, pero no se implementan, y se fuerza a las clases que usan ese interfaz a implementar los métodos.

A continuación un ejemplo de cómo definir una clase abstracta utilizando el módulo ABC:

```python
from abc import ABC, abstractmethod


class Mando(ABC):
    @abstractmethod
    def encender(self):
        pass

    @abstractmethod
    def apagar(self):
        pass

class Television(Mando):
    def encender(self):
        print("La televisión está encendida.")

    def apagar(self):
        print("La televisión está apagada.")

tv = Television()
tv.encender() # Imprime "La televisión está encendida."
tv.apagar() # Imprime "La televisión está apagada."
```

En este ejemplo, se define una clase abstracta Mando que hereda de ABC y define dos métodos abstractos: encender y apagar. Luego, se define una

subclase Televisión que hereda de Mando y sobrescribe los métodos abstractos para implementar el comportamiento específico de la televisión. Se puede instanciar la subclase Televisión y llamar a los métodos encender y apagar en la instancia, lo que hace que la televisión se comporte de manera específica en función de su relación con la clase abstracta Mando. Este es un ejemplo de interfaz formal en Python utilizando ABC.

Referencias

Apuntes Programa Clip - Felipe Steffolani

https://www.freecodecamp.org/

Laura Álvarez, Helmer Avendaño, Yeison García, Sebastián Morales,

Edwin Bohórquez, Santiago Hernandez, Sebastián Moreno, Cristian Orjuela. Programación Orientada a Objetos.

https://ferestrepoca.github.io/paradigmas-de-programacion/poo/poo_teoria/concepts.html

https://www.typescriptlang.org/

https://barcelonageeks.com/composicion-de-funciones-en-python/

111Mil. Módulo Programación Orientada a Objetos.

https://github.com/111milprogramadores/apuntes/blob/master/Programacion%20Orientada%20a%20Objetos/Apunte%20Teorico%20de%20Programacion%20OO.pdf