

1. Introducción

Buscando dar un paso consecuente a las temáticas que cubrimos en la asignatura, personalmente estimo que hay más continuidad de Python como lenguaje multiparadigma, al seleccionar **'la programación orientada a objetos'**, porque de alguna manera, la programación funcional la alcanzamos a avanzar en sus fundamentos; las otras propuestas posibles para el paper, para mi proceso personal representaría buscar avanzar mucho, saltándome utilidades que no puedo dejar pasar. Antes que todo, la definición temporal de 'paradigma de programación' que apporto es la siguiente: Modelo, concepción o enfoque, que busca orientar el proceso de construcción de aplicaciones, al implementar a través del modo propio del funcionamiento de los ordenadores para recibir instrucciones, un estilo particular de comunicación con sus procesadores. Para lograr lo que queremos, debemos utilizar inherentes 'metodologías' (por decirlo así de alguna manera), según lo que estamos propuestos a obtener en un contexto particular.

La Programación Orientada a Objetos **POO** (o OOP en inglés), nos brinda la oportunidad de organizar programas de una manera que se asemejen notablemente a como pensamos en el mundo real, utilizando colectivos llamados clases, que pueden agrupar conjuntos de variables y funciones. Este paradigma no se dio desde un comienzo en la informática, por eso hay bastantes lenguajes de programación que no la toleran. Se dio naturalmente como una respuesta a que muchas aplicaciones realizaban tareas similares, al identificarse esos patrones, no habría el porqué de repetirlos, ahorrándonos código y logrando mas organización al reutilizar variables y funciones, que pueden ser requeridas con características particulares creando los objetos. La POO facilita mantener el código, porque cada objeto es responsable de su específico comportamiento, al lograr organizar el software en torno a ellos, combinando datos y relaciones, consiguiendo en ultimas que sea más fácil identificar y solucionar problemas sin afectar otras partes del programa

El surgimiento de este modelamiento con objetos, logra su importancia en la esencia misma de los lenguajes de programación que hacen de uso, una estrategia, para poder desarrollar un sinfín de aplicaciones, desde las mas triviales y rutinarias, hasta las mas elaboradas con fines corporativos, de entretenimiento, o de suplir servicios en varias áreas sociales del quehacer humano, o del intelecto en la ciencia, mejor dicho, en todo lugar que se requiera una solución u optimización mediante medios computacionales. Por poner uno que otro ejemplo, contribuye a la programación web, con arquitectura multicapa, presentación hipertextual, etiquetación HTML, JavaScript y CSS; aporta al acceso de bases de datos relacionales en varios niveles de su diseño, manejando sqlexceptions, creando tablas, recuperando información; otra funcionalidad que podemos citar es que por su modularidad favorece la creación de videojuegos, ludismo que se mantiene diverso e interesante por sus propuestas en continuo desarrollo, que logran atrapar cada vez más la imaginación del ser humano mediante sus sentidos; para el mejor desenvolvimiento de esta multimedia los

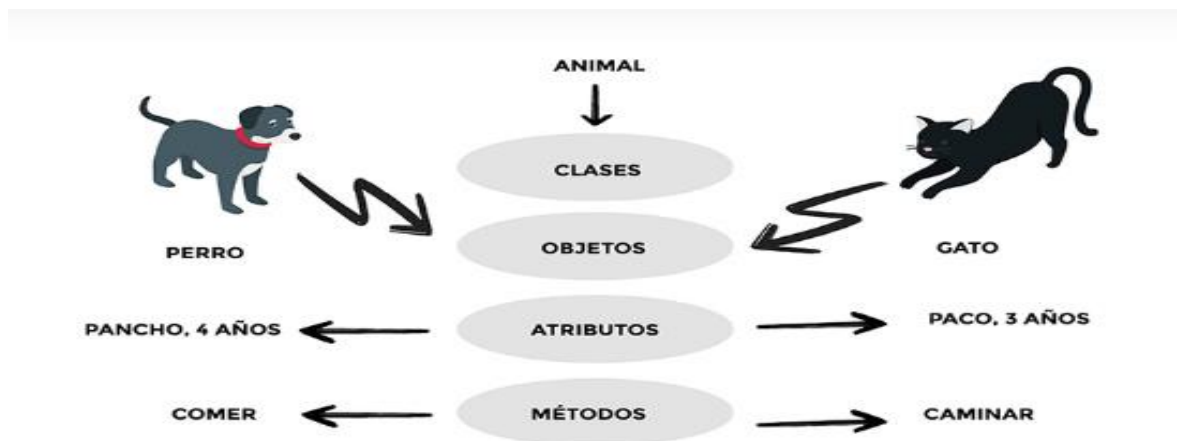
objetos pueden diseñarse de manera que sean independientes y puedan interactuar entre sí de manera controlada, contribuyendo todo lo anterior a la colaboración en equipos de desarrollo para crear juegos más escalables. Se vuelve muy complejo construir los juegos que tenemos hoy en día sin POO, ya que requieren la creación de personajes, entornos, interacciones y muchos elementos complicados, en donde puedan organizarse y representarse mejor. Podemos tener también aplicación del paradigma en desarrollo móvil, sistemas embebidos, inteligencia artificial, creación de simulaciones y modelos de sistemas múltiples. Brindas además garantías de seguridad con palabra clave, como lo son `private` o `protect`, que bien pueden ser utilizadas por entidades monetarias para transacciones financieras.

La relevancia entonces empieza a entreverse, por ejemplo, la capacidad de trabajar en áreas específicas de un proyecto sin afectar otras áreas, además reutilizando objetos en distintas partes del código, aumenta considerablemente la productividad, liberando un gran ahorro de tiempo. La aparición de la OOP marcó un cambio sustancial en como que abordaba el software, proporcionando una estructura organizativa que no solo mejora la legibilidad del código, sino también la eficiencia. Se puede obtener una organización más lógica del software, siendo mas comprensible, para permitir a los programadores crear sistemas de manera mas intuitiva. En el diseño web, los objetos representan componentes visuales y funcionales, su reutilización ayuda a simplificar el proceso de desarrollo, traducándose en un diseño mas coherente, con mas ahorro de tiempo para implementar nuevas funcionalidades; es por eso, que quienes laboran con web, pueden crear componentes reutilizables, como botones, formularios y menús, encapsulando su funcionalidad y mejorando su aspecto visual. Logra también integrarse a aplicaciones móviles, permitiendo solidez, con lenguajes como **Kotlin (para Android)** y **Swift(para iOS)**, facilitando la creación de componentes reutilizables y la implementación de patrones de diseño que aportan a mejorar la arquitectura de las aplicaciones. Por último, el poder que puede tener POO en las tecnologías de la información es fundamental, debido a su capacidad de ordenamiento del código, para facilitar la reutilización y el mantenimiento, promoviendo la seguridad e integridad del sistema; este modo de programación proporciona una mayor flexibilidad y extensibilidad en la evolución del software, porque objetos y clases pueden ser adaptados y ampliados para satisfacer nuevos requisitos y funcionalidades. Esto es especialmente importante en tecnologías de la información, donde los sistemas necesitan avanzar constantemente para seguir siendo competitivos. La capacidad de extender y adaptar el software existente de manera eficiente es un factor clave para el éxito en esta área. La encapsulación y la abstracción contribuyen a la seguridad y fiabilidad del software, al limitar el acceso a los datos privados y exponer solo una interfaz controlada para su manipulación, reduciendo la posibilidad de errores y vulnerabilidades. Podemos entonces generalizar que la POO es muy importante porque permite a los desarrolladores crear programas mas grandes, complejos y fáciles de mantener, y al poder utilizar reiteradas veces un código, se puede con base en ello construir nuevos programas.

Marco Teórico

Definición del paradigma de la Programación Orientada a Objetos:

Es un estilo generalizado de escribir programas computacionales, edificando su estructura en piezas simples, que pueden utilizar planos de códigos (clases) las veces que se quiera para crear instancias individuales de objetos. Dicho de otra manera, en lugar de ver un programa como una serie de instrucciones lineales, la POO permite agrupar datos(objetos) y las operaciones relacionadas con ellos en entidades(clases). Se valida entonces organizar códigos, de manera que se asemeje notoriamente a como pensamos en la realidad utilizando una clase particular, que está facultada para agrupar un conjunto de variables y funciones. Entidades de toda índole, como por decir un animal como un gato, puede ser representado así, de tal forma que las clases de gatos normalmente tienen diferentes características(atributos), como una edad determinada, un nombre, una raza; las clases tienen un conjunto de funcionalidades(métodos) o cosas que pueden hacer, para el ejemplo del gato puede ser como maullar o ronronear. Obviamente existen diferentes tipos de gato, para el ejemplo podemos tener alguno que se llame cristal o Fígaro, estas clases de gato son los objetos, es decir el concepto inteligible de gato es la clase, pero Figaro (o Garfield) o cualquier gato específico será el objeto. La pretensión de programar de esta manera, es dejar de centrarse en la lógica pura de los programas, para así empezar a pensar en objetos, porque en vez de razonar solo en funciones, pensamos en relaciones o interacciones de los diferentes componentes del sistema. Los objetos contienen información en forma de campos (referidos como atributos-propiedades) y código en forma de métodos, además de ser capaces de interactuar y modificar los valores que contienen sus campos o atributos(estado) a través de sus métodos(comportamiento), para de esta manera lograr el desenvolvimiento de las instrucciones. Centralmente un programador puede diseñar software organizando piezas de información y comportamientos relacionados en una plantilla llamada 'clase', luego, se crean los objetos individuales a partir de la plantilla, esperando que todo el software se ejecute(adapte) interrelacionándose.



Principios Fundamentales que lo caracterizan:

Los principios fundamentales de la Programación Orientada a Objetos (POO) son esenciales para entender cómo este paradigma facilita la creación de software estructurado. Estos principios no solo ayudan a organizar y diseñar el código, sino que también permiten a los desarrolladores manejar la complejidad de los sistemas de software a medida que estos crecen y evolucionan.

Abstracción: Este principio consiste en simplificar la complejidad del mundo real modelando solo los aspectos relevantes para un propósito particular, mientras se ocultan los detalles innecesarios. La abstracción permite a los desarrolladores concentrarse en lo que es importante para la aplicación, sin necesidad de entender cada detalle del comportamiento o estado interno de los objetos. Esto reduce la complejidad y mejora la comprensibilidad del código.

Encapsulamiento: El encapsulamiento es el concepto de ocultar los detalles internos del funcionamiento de un objeto, protegiendo su estado interno de cambios no autorizados y exponiendo solo una interfaz pública para la interacción. Esto se logra mediante el uso de métodos que actúan como los únicos puntos de acceso para los atributos del objeto. El encapsulamiento no solo protege la integridad de los datos, sino que también reduce las dependencias entre diferentes partes del código, facilitando su modificación y mantenimiento.

Herencia: La herencia permite que una clase derive o herede propiedades y comportamientos de otra clase, conocida como su clase padre. Esto facilita la reutilización del código y la extensión de las funcionalidades existentes sin tener que reescribir mucho código. Además, la herencia promueve la creación de una jerarquía de clases que puede reflejar relaciones naturales entre objetos más amplios y específicos, mejorando la organización del código.

Polimorfismo: El polimorfismo se refiere a la capacidad de un objeto para tomar varias formas y comportarse de manera diferente en diferentes contextos, usando la misma interfaz. Esto se logra mediante el uso de la herencia y la sobreescripción de métodos, permitiendo que un mismo método tenga diferentes implementaciones en distintas clases. El polimorfismo no solo hace el software más flexible y adaptable a nuevas situaciones, sino que también permite tratar objetos de diferentes clases de manera uniforme.

Juntos, estos principios forman la base sobre la cual se construyen aplicaciones robustas y escalables en la POO, permitiendo a los desarrolladores abordar problemas complejos de software de una manera más eficiente y efectiva.

Cohesión: hace referencia al grado de **relación entre los elementos de un módulo**. En el diseño de una función, es importante pensar bien la tarea que va a realizar, intentando que

sea única y bien definida. Cuantas más cosas diferentes haga una función sin relación entre sí, más complicado será el código de entender.

Acoplamiento: (denominado **coupling** en inglés) es un concepto que mide la dependencia entre dos módulos distintos de software, como pueden ser por ejemplo las clases. El acoplamiento puede ser de dos tipos:

- Acoplamiento **débil**, que indica que no existe dependencia de un módulo con otros. Esto debería ser la meta de nuestro software.
- Acoplamiento **fuerte**, que por lo contrario indica que un módulo tiene dependencias internas con otros.

El término acoplamiento está muy relacionado con la cohesión, ya que acoplamiento débil suele ir ligado a cohesión fuerte. En general lo que buscamos en nuestro código es que tenga acoplamiento débil y cohesión fuerte, es decir, que no tenga dependencias con otros módulos y que las tareas que realiza estén relacionadas entre sí. Un código así es fácil de leer, de reusar, mantener y tiene que ser nuestra meta. Nótese que se suele emplear alta y baja para designar fuerza y débil respectivamente.

Comparación con otros paradigmas—ventajas y desventajas

La POO como paradigma se diferencia de otros por su enfoque. La separación con el primer paradigma denominado estructural, es que en este último la importancia está en la información, en cambio en la POO, la importancia está en los objetos que manejan la información. En general la POO tiene un nivel de seguridad más alto, porque quien puede manipular la información es el mismo objeto. No es conveniente afirmar que el modelo de POO es más eficiente que otros paradigmas, pero es mucho más claro. El POO nos permite reducir la cantidad de código, además, la organización del código es más clara porque está encapsulada en el objeto. Al favorecer que el código sea más estructurado, encapsulado y extensible, puede añadir cierta complejidad y sobrecarga en términos de diseño y administración de memoria.

La programación funcional puede hacer que el código sea más elegante, modular y comprobable, pero también tiene una curva de aprendizaje empinada. Algunos teóricos la engloban como un subconjunto de la programación declarativa, que trata la computación como la evaluación de funciones matemáticas. En la programación funcional se elimina el uso del **efecto secundario** en las funciones así como en la programación estructurada se desaprueba o incluso elimina el uso de la sentencia 'goto'. La programación declarativa abstrae los detalles de cómo se ejecuta el programa y permite que la computadora descubra la mejor manera de lograr el resultado deseado. Sin embargo tiene algunas limitaciones en términos de rendimiento y depuración. La **programación funcional** cuyo lenguaje más expresivo y culmen sea seguramente el lenguaje **Haskell**, tiene su similar en la **programación lógica** donde el campeón es **Prolog** (ampliamente usado en ambientes académicos).

La programación declarativa es la que se **describe la lógica de computación** necesaria para resolver un problema **sin describir un flujo de control** de ningún tipo. Efectivamente, en la programación declarativa no es necesario definir **algoritmos** puesto que se detalla **la solución** del problema en lugar de **como llegar** a esa solución. En la programación declarativa, la solución es alcanzada a través de **mecanismos internos** de control pero no se especifica exactamente como llegar a ella. Las variables son utilizadas con **transparencia referencial**, es decir una expresión puede ser sustituida por el resultado de ser evaluada en el programa sin alterarlo **semánticamente**.

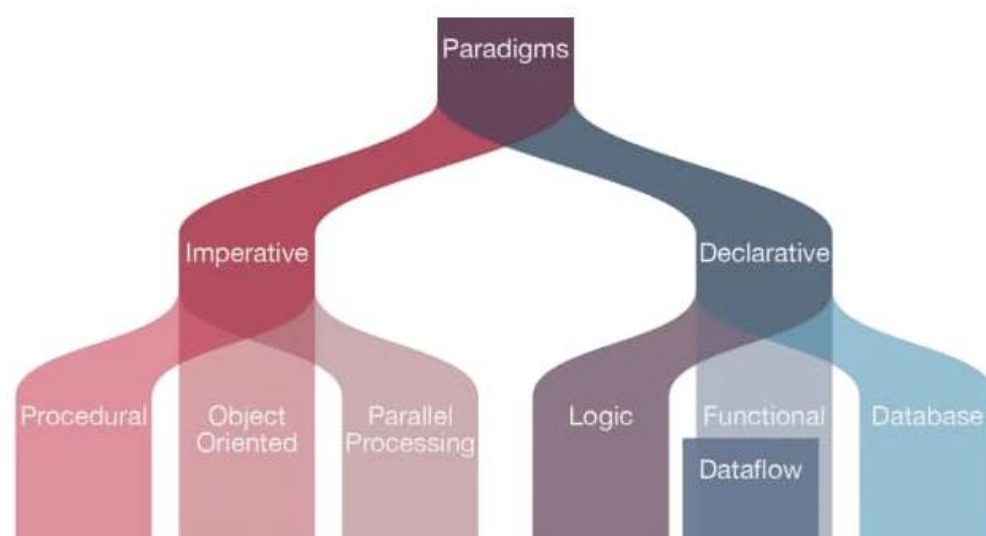
En la programación imperativa se **describen sentencias que modifican el estado de un programa**. En muchos sentidos la programación imperativa es la programación natural para las CPUs que se basan en ese paradigma al nivel **más básico**. En este paradigma se expresa **como debe solucionarse** un problema especificando una **secuencia de acciones** a realizar a través de uno o más procedimientos denominados subrutinas o funciones.

Para uno que otro autor, la programación orientada a objetos **puede concebirse como una forma de programación imperativa** puesto que al programar orientando a objetos se describe la secuencia que debe seguir el programa para resolver un problema dado. Dentro de esta categoría engloban la **programación estructurada**, la **programación modular** y la mencionada **POO**. Cada una de estas extensiones o evoluciones han permitido mejorar la **mantenibilidad** y la **calidad** de los programas imperativos.

La diferencia con otras formas de programación imperativas, como la programación estructurada es que en la orientación a objetos se hace uso de **estructuras de datos** llamadas objetos que **aglutinan propiedades y métodos** conjuntamente con sus interacciones. Aunque la diferencia entre la programación estructurada y la programación orientada a objetos **es grande**, realmente la mayor diferencia se encuentra en las ramas superiores. Las diferencias entre programación imperativa y programación declarativa son mucho **más severa y compleja** que las diferencias entre diferentes sub-paradigmas dentro de la programación imperativa. La principal diferencia entre ambos paradigmas, es que en la programación imperativa se describe paso a paso un conjunto de instrucciones que han de ejecutarse con la finalidad de variar el estado del programa y resolver un problema para hallar una solución. Es decir, se describe un algoritmo en el que se detallan los pasos secuenciales necesarios a seguir para la resolución de un problema; en la programación declarativa solo se describe el problema pero no los pasos necesarios para llegar a su solución, la cual es hallada mediante mecanismos internos de **inferencia de información** a partir de la descripción del problema en sí. La programación imperativa se basa en la máquina de Turing mientras que la programación declarativa se basa en el **cálculo lambda**.

Por último la programación concurrente es el paradigma que permite al programa ejecutar múltiples tareas o procesos al mismo tiempo. La programación simultánea puede mejorar

el rendimiento, la capacidad de respuesta y la escalabilidad del programa, especialmente para aplicaciones que tratan con grandes cantidades de datos, comunicación de red o interacción del usuario. La programación simultánea también puede plantear algunos desafíos en términos de sincronización, coordinación y comunicación entre las tareas o procesos concurrentes, así como posibles errores y errores debido a las condiciones de carrera, los bloqueos y los bloqueos en vivo.



Tipos de paradigmas de programación

Breve Historia y Evolución de la Programación Orientada a Objetos

Década de 1960 : Los primeros conceptos relacionados con la programación orientada a objetos comenzaron a emerger en la década de 1960 con la creación del lenguaje Simula.

Desarrollado por Ole-Johan Dahl y Kristen Nygaard en el Centro Noruego de Computación (Norsk Regnesentral), Simula fue originalmente creado para realizar simulaciones.

Simula 67 fue la versión de Simula que introdujo conceptos como clases, objetos y herencia. Aunque no era conocido por el nombre de “orientado a objetos” en ese momento, estos conceptos fundamentales sentaron las bases para futuros desarrollos.

Década de 1970 : En la década de 1970, Alan Kay, un investigador de Xerox PARC, acuñó el término “orientación a objetos” para describir un enfoque de programación basado en la simulación de sistemas biológicos. Kay desarrolló el lenguaje de programación Smalltalk, que se convirtió en una influencia importante en el desarrollo posterior de la programación orientada a objetos.

Smalltalk fue el primer lenguaje de programación en implementar completamente el paradigma orientado a objetos tal como lo conocemos hoy. Smalltalk introdujo una serie de

ideas revolucionarias, incluyendo el envío de mensajes entre objetos, la encapsulación y la herencia dinámica.

Evolución y Popularización

Década de 1980: La década de 1980 fue testigo del crecimiento y la popularización de la programación orientada a objetos. Bjarne Stroustrup en AT&T Bell Labs desarrolló C++, un lenguaje que evolucionó a partir del lenguaje C con la adición de características orientadas a objetos.

C++ incorporó clases y objetos, así como constructores, destructores y sobrecarga de operadores. Su compatibilidad con C y su capacidad para manejar tanto programación de bajo nivel como de alto nivel hicieron de C++ un lenguaje popular en la industria del software.

En la misma década, Brad Cox y Tom Love desarrollaron Objective-C, combinando las capacidades orientadas a objetos de Smalltalk con el lenguaje de programación C. Objective-C se convirtió en el lenguaje principal para el desarrollo de software en plataformas de Apple durante muchos años.

Muchos otros lenguajes pasaron a adoptar el paradigma de programación orientada a objetos. Por ejemplo, Ada, desarrollado inicialmente por el Departamento de Defensa de EE.UU., incorporó también características orientadas a objetos en su revisión de 1995 (Ada 95).

Década de 1990

En la década de 1990, Java se convirtió en uno de los lenguajes de programación más influyentes en el ámbito de la programación orientada a objetos. Desarrollado por Sun Microsystems (ahora parte de Oracle) y diseñado por James Gosling y su equipo, Java combinó la sintaxis de C++ con una arquitectura orientada a objetos.

De la misma época surge Python, un lenguaje que ha actualmente ha ganado gran popularidad, creado por Guido van Rossum. Aunque no fue originalmente diseñado como un lenguaje de programación orientado a objetos puro, sí disponía el concepto desde sus primeras versiones. Con el tiempo, ha evolucionado para mejorar el soporte a la POO, adaptado a sus propias necesidades.

Década 2000

C# desarrollado por Microsoft como parte de su plataforma .NET, se lanzó a principios de la década de 2000. Influenciado por C++ y, muy fuertemente por JavaScript, C# ha sido adoptado como el lenguaje principal para el desarrollo en la plataforma .NET, incluyendo aplicaciones de escritorio, web y móviles.

Década 2010 en adelante

JavaScript, inicialmente desarrollado a mediados de la década de 1990 por Brendan Eich en Netscape, ha evolucionado rápidamente para incorporar capacidades orientadas a objetos.

Con la introducción de ES6 (ECMAScript 2015), JavaScript se volvió mucho más orientado a objetos, con soporte para clases y herencia, aumentando su uso en el desarrollo web frontend y backend.

Casos de usos

Una de las principales aplicaciones de este paradigma son los proyectos grandes que involucran a varios programadores, ya que, gracias a su modularidad, el producto final puede ser ensamblado sin mayores problemas. Al estar orientada la programación a los objetos, podemos distribuir el trabajo de los programadores y equipos de desarrollo y entregar parte del software y volverlo a unir y tendría que funcionar de mejor manera. Por ejemplo, en el desarrollo de videojuegos, la programación orientada a objetos se convierte en una herramienta invaluable.

La POO permite a los desarrolladores identificar objetos y sus relaciones en el juego. Por ejemplo, un juego puede tener objetos como jugadores, enemigos, armas y obstáculos. Cada uno de estos objetos tiene propiedades y comportamientos únicos que se pueden modelar de manera eficiente utilizando la programación orientada a objetos. Asimismo, los videojuegos son proyectos complejos que pueden crecer rápidamente en tamaño y complejidad a medida que se desarrollan. La programación orientada a objetos facilita el mantenimiento del código, ya que cada objeto es responsable de su propio comportamiento. Esto hace que sea más sencillo identificar y solucionar problemas sin afectar a otras partes del juego. La modularidad es otra ventaja de la POO en la creación de videojuegos. Los objetos pueden diseñarse de manera que sean independientes y puedan interactuar entre sí de manera controlada. Esto facilita la colaboración en equipos de desarrollo y permite la creación de juegos más escalables. En el desarrollo por ejemplo de aplicaciones móviles, con lenguajes como **Kotlin (para Android)** y **Swift(para iOS)**, se facilitó la creación de componentes reutilizables y la implementación de patrones de diseño que aportan a mejorar la arquitectura de las aplicaciones.

Lenguajes de programación asociados:

Actualmente hay distintos lenguajes de programación orientada a objetos, como **C++, Objective C, Ruby, Visual Basic, Java, Visual C Sharp, Perl, TypeScript, Simula, Smalltalk, Python o PHP**. Java y C++ son los dos lenguajes de programación orientada a objetos más usados, asimismo, PHP, Python y Ruby son otros lenguajes de programación orientada a objetos muy populares, pero están **más enfocados en la programación, desarrollo web y de aplicaciones para móviles**.

Existen otros lenguajes de programación orientada a objetos de carácter más específico como es el caso de **ADA**, que se basa en un tipado muy fuerte y su enfoque es el de la

seguridad. Este lenguaje se suele utilizar para desarrollar aplicaciones de defensa y gestión de tráfico aéreo, además de la industria aeroespacial.

El Implementar la Programación Orientada a Objetos (POO) en diferentes lenguajes implica utilizar las características y sintaxis específicas de cada uno. A continuación, te indicamos una breve descripción de cómo realizarlo en algunos lenguajes populares y cómo aprovechar sus ventajas para construir aplicaciones sólidas y flexibles:

1.

Java:

- Define clases utilizando la palabra clave «class» y especifica atributos y métodos dentro de ellas.
- Utiliza la herencia con la palabra clave «extends» para crear subclases que hereden características de una clase base.
- Aprovecha las interfaces para definir contratos que las clases deben implementar.
- Aprovecha el polimorfismo para tratar objetos de diferentes clases de manera uniforme, utilizando métodos con la misma firma en diferentes clases.
- Utiliza la encapsulación para ocultar los detalles internos de los objetos y proporcionar una interfaz pública para interactuar con ellos.

2. **C++:**

- Define clases utilizando la palabra clave «class» y especifica atributos y métodos dentro de ellas.
- Utiliza la herencia con los operadores de acceso «public», «private» y «protected» para establecer la visibilidad de los miembros heredados.
- Aprovecha las funciones virtuales y la herencia múltiple para lograr polimorfismo.
- Utiliza la encapsulación con los modificadores de acceso «public», «private» y «protected» para controlar el acceso a los miembros de una clase.

3. **Python:**

- Define clases utilizando la palabra clave «class» y especifica atributos y métodos dentro de ellas.
- Aprovecha la herencia utilizando la clase base entre paréntesis al definir una clase.
- Utiliza la herencia múltiple y el polimorfismo de manera natural, ya que Python lo permite.
- Aprovecha las propiedades y los decoradores para implementar la encapsulación y controlar el acceso a los atributos de una clase.

4. **C#:**

- Define clases utilizando la palabra clave «class» y especifica atributos y métodos dentro de ellas.
- Utiliza la herencia con los operadores de acceso «public» y «protected» para establecer la visibilidad de los miembros heredados.
- Aprovecha las interfaces para definir contratos que las clases deben implementar.
- Utiliza el polimorfismo utilizando la palabra clave «virtual» para los métodos en la clase base y «override» en las subclases.
- Utiliza los modificadores de acceso «public», «private» y «protected» para lograr encapsulación y controlar el acceso a los miembros de una clase.

Ejemplo de código ilustrativo:

El código anterior actúa como un vendedor de cafetería. Te pedirá un presupuesto y luego te «venderá» el mejor café que seas capaz de comprar.

Intenta ejecutarlo en la [terminal](#). Se ejecutará paso a paso, dependiendo de tu entrada.

El código siguiente representa una **clase** llamada «Coffee». Tiene dos atributos – «Name» y «Price» – y ambos se utilizan en los métodos. El método principal es «Sell», que procesa toda la lógica necesaria para completar el proceso de venta.

```
small = Coffee('Small', 2)
regular = Coffee('Regular', 5)
big = Coffee('Big', 6)

try:
    user_budget = float(input('What is your budget? '))
except ValueError:
    exit('Please enter a number')

for coffee in [big, regular, small]:
    coffee.sell(user_budget)

class Coffee:
    # Constructor
    def __init__(self, name, price):
        self.name = name
        self.price = float(price)
    def check_budget(self, budget):
```

```

        # Check if the budget is valid
        if not isinstance(budget, (int, float)):
            print('Enter float or int')
            exit()
        if budget < 0:
            print('Sorry you don\'t have money')
            exit()
    def get_change(self, budget):
        return budget - self.price

    def sell(self, budget):
        self.check_budget(budget)
        if budget >= self.price:
            print(f'You can buy the {self.name} coffee')
            if budget == self.price:
                print('It\'s complete')
            else:
                print(f'Here is your change
{self.get_change(budget)}$')

        exit('Thanks for your transaction')

```

Parte Práctica con principios(pilares) en que se basa la POO

Veamos un ejemplo inicial de 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 sobrescribir los métodos o atributos, o incluso definir unos nuevos.

Se puede crear una clase hija con tan solo pasar como parámetro la clase de la que queremos heredar. En el siguiente ejemplo vemos como se puede usar la herencia en Python, con la clase `Perro` que hereda de `Animal`.

```

# Definimos una clase padre
class Animal:
    pass

# Creamos una clase hija que hereda de la padre
class Perro(Animal):
    pass

print(Perro.__bases__)
# (<class '__main__.Animal'>,)

```

```
print (Animal.__subclasses__())  
# [<class '__main__.Perro'>]
```

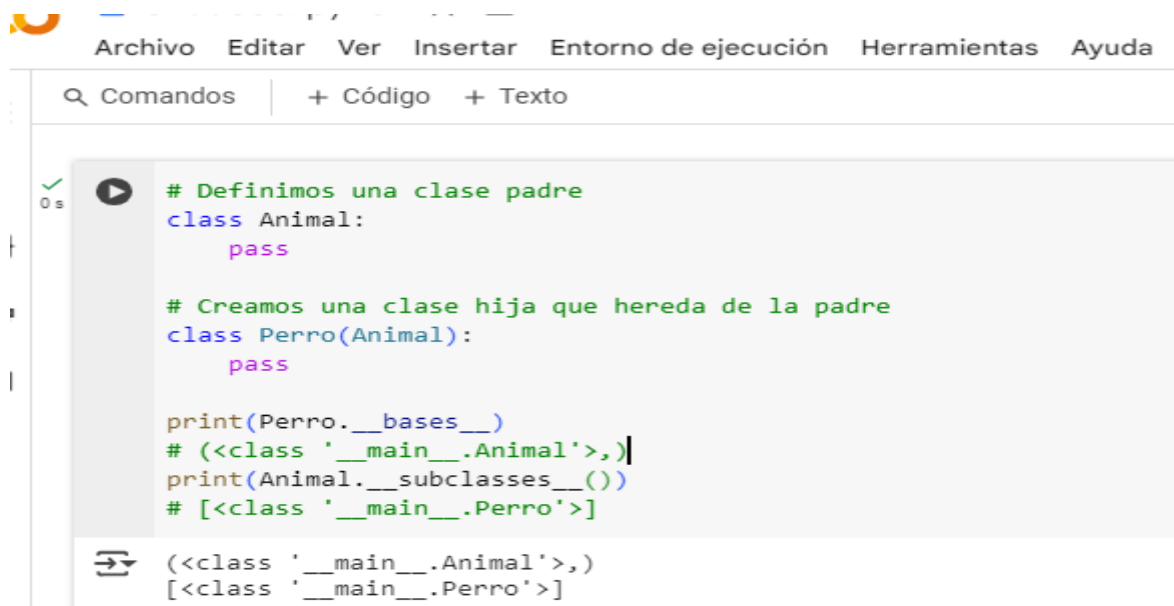
Definiendo métodos

En realidad cuando usamos `__init__` anteriormente ya estábamos definiendo un método, solo que uno especial. A continuación vamos a ver como definir métodos que le den alguna funcionalidad interesante a nuestra clase, siguiendo con el ejemplo de perro.

Vamos a codificar dos métodos, ladrar y caminar. El primero no recibirá ningún parámetro y el segundo recibirá el número de pasos que queremos andar. Como hemos indicado anteriormente `self` hace referencia a la instancia de la clase. Se puede definir un método con `def` y el nombre, y entre `()` los parámetros de entrada que recibe, donde siempre tendrá que estar `self` el primero.

```
class Perro:  
    # Atributo de clase  
    especie = 'mamífero'  
  
    # El método __init__ es llamado al crear el objeto  
    def __init__(self, nombre, raza):  
        print(f"Creando perro {nombre}, {raza}")  
  
        # Atributos de instancia  
        self.nombre = nombre  
        self.raza = raza  
  
    def ladra(self):  
        print("Guau")  
  
    def camina(self, pasos):  
        print(f"Caminando {pasos} pasos")  
  
mi_perro = Perro("Toby", "Bulldog")  
mi_perro.ladra()  
mi_perro.camina(10)  
  
# Creando perro Toby, Bulldog  
# Guau  
# Caminando 10 pasos
```

Pantallazo principio Herencia



```
Archivo  Editar  Ver  Insertar  Entorno de ejecución  Herramientas  Ayuda

Comandos  + Código  + Texto

# Definimos una clase padre
class Animal:
    pass

# Creamos una clase hija que hereda de la padre
class Perro(Animal):
    pass

print(Perro.__bases__)
# (<class '__main__.Animal'>,)
print(Animal.__subclasses__())
# [<class '__main__.Perro'>]
```

Output: (<class '__main__.Animal'>,)
[<class '__main__.Perro'>]

Creacion Método:



```
class Perro:
    # Atributo de clase
    especie = 'mamífero'

    # El método __init__ es llamado al crear el objeto
    def __init__(self, nombre, raza):
        print(f"Creando perro {nombre}, {raza}")

        # Atributos de instancia
        self.nombre = nombre
        self.raza = raza

    def ladra(self):
        print("Guau")

    def camina(self, pasos):
        print(f"Caminando {pasos} pasos")

mi_perro = Perro("Toby", "Bulldog")
mi_perro.ladra()
mi_perro.camina(10)

# Creando perro Toby, Bulldog
# Guau
# Caminando 10 pasos
```

Output: Creando perro Toby, Bulldog
Guau
Caminando 10 pasos

Dificultades al implementar este paradigma:

puede añadir cierta complejidad y sobrecarga en términos de diseño y administración de memoria.

Análisis de rendimiento

Acá la POO es muy eficiente por el tiempo que se gana en resumir y presentar, igual presenta muy buen nivel de escalabilidad y mantenibilidad al estar en subgrupos su código.

Comparación con otros paradigmas

Estos problemas particulares si se pueden resolver con otros enfoques, por ejemplo el funcional y estructural, con ciclos y bloques, el problema es lo extenso que puede ser, con mas facilidad al error en el indexar.

Cuando utilizar este enfoque

Definitivamente cuando podemos subagrupar, y vemos que podemos estar utilizando el mismo código en varias ocasiones, con diferentes resultados al evaluarlos.

Conclusiones

La POO como paradigma se diferencia de otros por su enfoque. La separación con el primer paradigma denominado estructural, es que en este ultimo la importancia está en la información, en cambio en la POO, la importancia está en los objetos que manejan la información. En general la POO tiene un nivel de seguridad más alto, porque quien puede manipular la información es el mismo objeto. No es conveniente afirmar que el modelo de POO es más eficiente que otros paradigmas, pero es mucho más claro. El POO nos permite reducir la cantidad de código, además, la organización del código es más clara porque está encapsulada en el objeto. Al favorecer que el código sea más estructurado, encapsulado y extensible, puede añadir cierta complejidad y sobrecarga en términos de diseño y administración de memoria.

Conceptos importantes y relacionados con la POO:

- [Herencia](#)
- [Cohesión](#)
- [Abstracción](#)
- [Polimorfismo](#)
- [Acoplamiento](#)
- [Encapsulamiento](#)

Aplicación al área de carrera

Total, son muchas las cosas que se derivan de tal metodología, en sí, la forma de pensar y agrupar es muy funcional sin ser del paradigma funcional. Solo un sencillo ejemplo, la programación de microcontroladores puede hacerse mediante Python y la OPP es una herramienta sin reemplazo para tantos detalles de variables.

Enlace al repositorio:

<https://github.com/hugoramuribe/Parcial-3-PC-18/upload/main>