TECNICATURA SUPERIOR EN ANÁLISIS DE SISTEMAS (Paradigmas de Programación)

Parcial

2. Programación orientada a objetos

2.1. ¿Cuáles son los pilares? ¿Explicar cada uno de ellos?

2.2. ¿Cuál es la diferencia entre una clase y un objeto en POO?

2.3. ¿Qué son los métodos de acceso y los métodos de modificación?

2.4. ¿Cuál es el propósito de los métodos de modificación?

2.5. Explica la diferencia entre una variable de clase y una variable de instancia.

2.5. ¿Qué es el polimorfismo y cómo se implementa?

2.6. ¿Qué es la encapsulación y cómo se implementa? ¿Por qué es importante?

2.6. ¿Cuáles son los tipos de visibilidades? ¿Cómo se implementan en Python?

3. Diseño de clases

3.1. ¿Qué es la relación “Es un”? Explicar y dar ejemplo en código

3.2. ¿Qué es la relación “Tiene un”? Explicar y dar ejemplos en código

4. Buenas prácticas

4.1. ¿Qué son las buenas prácticas de programación? ¿Por qué es importante seguir un estándar?

4.2. ¿Por qué es fundamental evitar la duplicación de código? ¿Cómo logramos esto?

4.2. ¿Por qué es esencial gestionar adecuadamente las importaciones en un proyecto?

5. Algoritmos

5.1. ¿Qué es la recursividad? Explicar y dar ejemplos de uso

5.2. ¿Qué son los administradores de contexto? ¿Cuáles son sus principales usos?

5.3. ¿Qué son los decoradores? ¿Podemos crear nuestros propios decoradores?

5.4. ¿Cuándo utilizarías administrador de contexto en lugar de un decorador?

5.5. ¿Qué tipos de imports existen?

Diseñar clases utilizando Programación Orientada a Objetos y Composición

2. Programación orientada a objetos

2.1. ¿Cuáles son los pilares? Explicar cada uno de ellos?

Los pilares de la Programación Orientada a Objetos (POO) son (HEPA):

- Abstracción: Es el proceso de simplificar los sistemas complejos, ignorando los detalles menos relevantes. En POO, se utilizan clases para representar conceptos del mundo real de manera simple y clara.
- Encapsulamiento: Se refiere a la ocultación de los detalles internos o mecanismos de un objeto y la exposición solo de las partes necesarias. Esto asegura que los objetos no puedan cambiar el estado interno de otros objetos de manera inesperada.
- Herencia: Permite crear nuevas clases a partir de clases existentes, heredando sus atributos y métodos. Esto facilita la reutilización de código y la representación de relaciones de tipo “es un” entre objetos.
- Polimorfismo: Permite que un objeto pueda tomar diferentes formas. En POO, se refiere a la capacidad de una clase para tener diferentes implementaciones de un mismo método.

2.2. ¿Cuál es la diferencia entre una clase y un objeto en POO?

En POO, una clase es como un plano o plantilla que define las propiedades y comportamientos (métodos) que un objeto de esa clase tendrá. Un objeto es una instancia de una clase, es decir, es un elemento creado a partir de la plantilla que la clase proporciona.

2.3. ¿Qué son los métodos de acceso y los métodos de modificación? ¿Cómo se usan en Python?

Los métodos de acceso (o getters) y los métodos de modificación (o setters) se utilizan en POO para proporcionar acceso a variables privadas de un objeto.

Los métodos de acceso se utilizan para obtener el valor de una variable privada, mientras que los métodos de modificación se utilizan para establecer o modificar el valor de una variable privada. En Python, aunque no hay una distinción formal entre métodos públicos y privados, se utiliza un guion bajo (_) antes del nombre del método o atributo para indicar que es privado.

2.4. ¿Cúal es la diferencia entre variable de clase y de instancia?

Una variable de clase es una variable que es compartida por todas las instancias de una clase. Es decir, si cambia el valor de una variable de clase, ese cambio se reflejará en todas las instancias de la clase.

Por otro lado, una variable de instancia es una variable que es única para cada instancia de una clase. Cada objeto tiene su propia copia de la variable de instancia, y los cambios en una no afectan a las demás.

2.5. ¿Qué es el polimorfismo y cómo se implementa?

El polimorfismo es la capacidad de un objeto de tomar diferentes formas y comportarse de diferentes maneras según el contexto en el que se encuentre. En POO, el polimorfismo se refiere a la capacidad de una clase para tener diferentes implementaciones de un mismo método. En Python, el polimorfismo se implementa permitiendo que diferentes clases tengan métodos con el mismo nombre. Cuando se llama a un método, Python determina automáticamente qué implementación de método usar basándose en el objeto que está invocando el método.

2.5. Explica la diferencia entre una variable de clase y una variable de instancia.

Las variables de clase y las variables de instancia son dos tipos de variables que se utilizan en la programación orientada a objetos, y cada una tiene un propósito y comportamiento específicos:

Variable de Clase: Una variable de clase, también conocida como variable miembro de dato estático, es una variable que es propia de la clase que la contiene y no de instancias de la misma. Esto significa que todos los objetos que se crean de esta clase comparten su valor. En otras palabras, una variable de clase es una variable que se declara dentro de una clase pero fuera de cualquier método, y se inicializa una vez y existe hasta que finaliza el programa.
En Java, por ejemplo, se declara con la palabra clave static1.

Variable de Instancia: Una variable de instancia, también conocida como variable miembro de dato, es una variable que se relaciona con una única instancia de una clase. Cada vez que se crea un objeto, el sistema crea una copia de todas las variables de instancia que están vinculadas con dicha clase, haciéndolas propias de esa instancia (objeto). En otras palabras, una variable de instancia es una variable que se declara dentro de una clase pero fuera de cualquier método, y se crea cada vez que se crea un objeto de la clase

2.6. ¿Cuáles son los tipos de visibilidades? ¿Cómo se implementan en Python?

En Python, todos los atributos y métodos de una clase son públicos por defecto. Sin embargo, se puede indicar que un atributo o método es privado precediéndolo con un guion bajo (_). Esto no impide técnicamente el acceso al atributo o método, pero es una convención que indica que no debería ser accedido directamente. Python no tiene modificadores de acceso como private o protected como en otros lenguajes de programación orientados a objetos como JAVA y C#.

2.6. ¿Cuáles son los tipos de visibilidades? ¿Cómo se implementan en Python?

En programación, la visibilidad se refiere a la capacidad de un objeto para tener una referencia a otro. Existen cuatro formas básicas de visibilidad:

1. Por atributo: El objeto B es un pseudoatributo de A. Es una visibilidad permanente ya que existe mientras A y B existan. Es el tipo de visibilidad más común en sistemas orientados a objetos.
2. Por parámetro: El objeto B fue recibido como parámetro en un método de A. Es una visibilidad temporal ya que existe solamente en el alcance del método.
3. Local: El objeto B es declarado localmente en un método de A. También es una visibilidad temporal ya que existe en el alcance del método.
4. Global: El objeto B es visible en forma global. Es una visibilidad relativamente permanente ya que existe mientras A y B existan.

En el caso de Python, no existen las mismas reglas de visibilidad que en otros lenguajes de programación como Java o C++. Sin embargo, Python tiene su propia convención para indicar la visibilidad de los atributos y métodos en una clase:

1. Público: Todos los miembros de una clase son públicos por defecto en Python. Cualquier objeto puede acceder a un miembro público.
2. Privado: Si el nombre de un miembro comienza con dos guiones bajos __, se vuelve privado. Solo puede ser accedido dentro de su clase.
3. Protegido: Los miembros de una clase son considerados protegidos cuando su nombre comienza con un guión bajo _. Pueden ser accedidos desde la clase y sus subclases, pero es una convención, no una regla estricta.

3. Diseño de clases

3.1. ¿Qué es la relación “Es un”? Explicar y dar ejemplo en código

La relación “Es un” en la Programación Orientada a Objetos (POO) se refiere a la herencia, que es un mecanismo que permite a un objeto heredar propiedades y comportamientos de otro objeto.

En términos simples, la herencia es una relación entre dos objetos en la que uno de los objetos es una versión especializada del otro objeto.

```
class Vehiculo:
  def __init__(self, color, ruedas):
    self.color = color
    self.ruedas = ruedas

class Moto(Vehiculo):
  def __init__(self, color, ruedas, velocidad, cilindrada):
    super().__init__(color, ruedas)
```

```
        self.velocidad = velocidad
        self.cilindrada = cilindrada

mi_moto = Moto("rojo", 2, 200, 1200)
```

En este ejemplo, Moto es un Vehiculo, ya que hereda las propiedades y comportamientos de la clase Vehiculo.

3.2. ¿Qué es la relación “Tiene un”? Explicar y dar ejemplos en código

La relación “Tiene un” en POO se refiere a la composición, que es una técnica de programación que se utiliza para implementar objetos que contengan variables de instancia que hagan referencia a otros objetos. Se implementa una relación de un objeto con otras clases

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

class Coche:
    def __init__(self, color, ruedas, motor):
        self.color = color
        self.ruedas = ruedas
        self.motor = motor

mi_motor = Motor(1200)
mi_coche = Coche("rojo", 4, mi_motor)
```

4. Buenas prácticas

4.1. ¿Qué son las buenas prácticas de programación? ¿Por qué son importantes?

Las buenas prácticas de programación se refieren a un conjunto de técnicas, principios y metodologías que los programadores deben implementar en su software para que sea rápido, fácil y seguro de desarrollar y desplegar.

Estas prácticas son fundamentales para el desempeño de todo aquel que busque especializarse en el desarrollo o simplemente busque avanzar mejorando su metodología de programación. Con la implementación de buenas prácticas, se puede obtener un código limpio, reutilizable, escalable y con mayor cambiabilidad, lo cual ayuda a disminuir el tiempo de dedicación en las tareas, prevenir y sortear errores comunes durante las diferentes etapas de realización que puede tener un proyecto.

Principio de Responsabilidad Única (SRP - Single Responsibility Principle):

Cada clase debe tener una única razón para cambiar. Esto significa que una clase debe tener una única responsabilidad o tarea. Si una clase realiza múltiples tareas o tiene múltiples motivos para cambiar, es más difícil de mantener y entender. Diseña clases que sean cohesivas y centradas en una tarea específica.

Simplificación de conceptos, resumiendo las ideas

Uno de los principios clave en el diseño de clases es la abstracción. La abstracción implica la identificación y representación de los aspectos más importantes y relevantes de un objeto del mundo real en una clase. En lugar de modelar cada detalle minuciosamente, se capturan solo los atributos y comportamientos esenciales. Este proceso de abstracción simplifica conceptos complejos, lo que es crucial para la comprensión y gestión de datos y procesos en un sistema de software. Al centrarse en lo esencial, el código se vuelve más claro y manejable.

En pocas palabras, abstrae los conceptos relevantes de la vida real en tus clases. Identifica las propiedades y comportamientos esenciales que deben ser representados y encapsulados en la clase. Esto te ayudará a crear modelos más claros y precisos.

Control de acceso a los datos, protege los datos

Utiliza la encapsulación para ocultar los detalles internos de una clase y proporciona una interfaz clara para interactuar con ella. Esto se logra utilizando atributos privados y métodos públicos o propiedades para acceder y modificar esos atributos. En Python, puedes usar guiones bajos (_atributo) para indicar que un atributo es "privado", aunque aún puede ser accedido desde fuera de la clase.

La encapsulación es esencial para proteger los datos y garantizar la integridad del sistema. Al restringir el acceso directo a los atributos internos, se evita la modificación no autorizada de los datos, lo que puede causar comportamientos inesperados. Extiende tus clases para compartir comportamientos Utiliza la herencia cuando tengas una relación "es un" entre clases y la composición cuando tengas una relación "tiene un". La herencia se utiliza para crear

nuevas clases basadas en clases existentes, mientras que la composición implica que una clase contiene una instancia de otra. La composición generalmente es preferible a la herencia, ya que es más flexible y evita problemas potenciales de herencia múltiple.

Es importante tener cuidado al usar la herencia. Demasiada herencia puede llevara una jerarquía de clases compleja y difícil de mantener. En lugar de heredar por capricho, se debe usar cuando existe una relación "es un" lógica y coherente entre las clases.

4.2. ¿Por qué es esencial gestionar adecuadamente las importaciones en un proyecto?

La gestión adecuada de las importaciones en un proyecto de programación es esencial por varias razones.

Primero, ayuda a mantener el código organizado y fácil de leer, lo que facilita su mantenimiento y depuración.

Segundo, permite reutilizar código de manera eficiente, evitando la duplicación de código.

Tercero, al importar solo las funciones o clases necesarias, se puede mejorar la eficiencia del código y reducir su tiempo de ejecución.

Finalmente, una gestión adecuada de las importaciones puede ayudar a evitar conflictos de nombres y problemas de dependencias

5. Algoritmos

5.1. ¿Qué es la recursividad? Explicar y dar ejemplos de uso

La recursividad es una técnica de programación en la que una función se llama a sí misma para resolver un problema.

Es una técnica muy poderosa y flexible que se utiliza en muchas áreas de la informática, como la inteligencia artificial, la criptografía, la toma de decisiones

y la resolución de problemas complejos.

Un ejemplo clásico de recursividad es el cálculo del factorial de un número.

```
def factorial(n):
  if n == 1:
    return 1
  else:
    return n * factorial(n - 1)
```

5.2. ¿Qué son los administradores de contexto? ¿Cuáles son sus principales usos?

Un administrador de contexto en Python es una construcción que permite configurar algún recurso, por ejemplo, abrir una conexión, y automáticamente maneja la limpieza cuando terminamos con ella.

El caso de uso más común es abrir un archivo

```
with open('/path/to/file.txt', 'r') as f:
  for line in f:
    print(line)
```

5.3. ¿Qué son los decoradores?

Un decorador en Python es una función que toma una función como argumento y devuelve una versión modificada de la función que se pasó. Los decoradores permiten modificar el comportamiento de una función envolviéndola dentro de otra función.

5.4. ¿Cuándo utilizarías un decorador en vez de administrador de contexto?

Tanto los decoradores como los administradores de contexto se utilizan para modificar el comportamiento de una función o un bloque de código. Sin embargo, se usa un decorador cuando quiero modificar el comportamiento de una función de manera que se aplique a todas las invocaciones de esa función. Por otro lado, se usa un administrador de contexto cuando quiero asegurarme de que ciertas operaciones se realicen antes y después de un bloque de código específico

5.5. ¿Qué tipos de imports existen?

En Python, hay varios tipos de importaciones:

1 Importación absoluta: Importa un módulo completo.

```
import math
```

2 Importación relativa: Importa una parte específica de un módulo.

```
from math import sqrt
```

3 Alias de importación: Importa un módulo o una parte de un módulo y le asigna un alias.

```
import math as m
```

```
from math import sqrt as square_root
```