Plan de Formación Profesional y Continua

curso gratuito

[Análisis de datos]

Programación Orientada a Objetos en Python

alkemy

fomentar empleo

Ministerio de Trabajo, Empleo y Seguridad Social Argentina

Introducción

El paradigma de objetos es una forma de organizar y estructurar el código en programación, que se basa en la conceptualización de objetos como entidades que encapsulan datos y comportamiento. Python, como lenguaje de programación orientado a objetos, ofrece un conjunto de características y conceptos que permiten aplicar este paradigma de manera eficiente y elegante.

En este documento podrás acceder a lectura complementaria te será de utilidad para profundizar los temas vistos en clase y en los demás recursos asincrónicos. Aquí exploraremos algunos aspectos fundamentales del paradigma de objetos en Python, centrándonos en temas que complementan los conceptos básicos ya conocidos, como polimorfismo, herencia, abstracción y encapsulamiento. Nuestro objetivo es brindar una visión más completa y profunda de cómo utilizar el paradigma de objetos en Python para desarrollar aplicaciones robustas y flexibles.

En las secciones siguientes, exploraremos temas como decoradores, métodos especiales, composición, métodos estáticos y descriptores. Estos conceptos ampliarán tu comprensión de cómo diseñar y estructurar tus clases de manera eficiente, aprovechando al máximo las características y capacidades que Python ofrece.

Programación Orientada a Objetos

Características de la POO en Python 📖

La Programación Orientada a Objetos (POO) es un paradigma de programación que utiliza objetos para organizar el código. Python es un lenguaje de programación que soporta la POO de manera completa.

Las características de la Programación Orientada a Objetos (POO) en Python son:

Clases y Objetos:

La POO organiza el código en unidades llamadas clases, que actúan como plantillas para crear objetos. Los objetos son instancias específicas de una clase y encapsulan datos y comportamientos relacionados:

- **Clase:** Define un modelo o plantilla para crear objetos. Puedes pensar en una clase como un plano para construir objetos.
- **Objeto:** Es una instancia concreta de una clase. Los objetos tienen atributos y métodos basados en la definición de la clase.

Encapsulación:

La encapsulación se refiere a la capacidad de una clase para ocultar sus detalles internos y exponer solo lo necesario. En Python, la encapsulación se logra mediante la convención de nombres (atributos y métodos privados se marcan con un guion bajo doble).

Herencia:

La herencia permite que una clase adquiera atributos y métodos de otra clase, facilitando la reutilización del código. Se establece una relación "es un/a" entre la clase base (superclase) y la clase derivada (subclase).

Polimorfismo:

El polimorfismo permite que objetos de diferentes clases se utilicen de manera uniforme si cumplen con una interfaz común. Esto facilita la escritura de código más genérico y flexible, ya que objetos de clases diferentes pueden responder a los mismos métodos.

Abstracción:

La abstracción implica la simplificación de la representación de un objeto. Las clases abstractas e interfaces permiten definir una estructura común sin preocuparse por los detalles de implementación.

Clases y objetos 📖

La POO organiza el código en unidades llamadas clases, que actúan como plantillas para crear objetos. Los objetos son instancias específicas de una clase y encapsulan datos y comportamientos relacionados.

La POO organiza el código mediante el uso de clases y objetos. Una clase es una plantilla para crear objetos, y un objeto es una instancia específica de una clase. Las clases definen atributos (variables) y métodos (funciones) que operan en esos atributos.

Clases:

Una clase es un modelo o plano para crear objetos. Define un conjunto de atributos y métodos que tendrán los objetos creados a partir de ella.

Sintaxis en Python:

```
class MiClase:
    # Atributos y métodos de la clase
```

Ejemplo Técnico:

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

    def saludar(self):
  return  f"Hola,  soy  {self.nombre}  y  tengo  {self.edad}
años."
```

Objetos:

Un objeto es una instancia concreta de una clase. Se crea a partir de una clase y tiene atributos y métodos específicos.

Sintaxis en Python (Creación de un Objeto):

```
# Creación de un objeto a partir de la clase
mi_persona = Persona("Juan", 30)
```

Uso y Acceso a Atributos/Métodos:

```
# Acceso a los atributos y métodos del objeto
print(mi_persona.nombre)     # Salida: Juan
print(mi_persona.edad)       # Salida: 30
print(mi_persona.saludar())  # Salida: Hola, soy Juan y
tengo 30 años.
```

Uso Práctico:

```
# Crear instancias de la clase Persona
persona1 = Persona("Ana", 25)
persona2 = Persona("Carlos", 35)

# Acceder a los atributos y métodos de los objetos
print(persona1.nombre)     # Salida: Ana
print(persona2.saludar())  # Salida: Hola, soy Carlos y
tengo 35 años.
```

En este ejemplo, Persona es la clase, y persona1 y persona2 son dos objetos instanciados a partir de esa clase. Cada objeto tiene su propio conjunto de atributos y métodos, pero comparten la misma estructura definida por la clase. La POO proporciona un marco poderoso para organizar y estructurar el código.

Clases: Proporcionan un modelo para crear objetos. Definen atributos y métodos comunes a todos los objetos de esa clase.

Objetos: Son instancias concretas de una clase. Tienen atributos y métodos específicos según la definición de la clase.

Vamos a explorar otro ejemplo práctico utilizando clases y objetos en Python. Consideremos la representación de un Banco con clientes y cuentas bancarias.

Ejemplo Práctico: Banco, Cliente y Cuenta Bancaria

```
class Cliente:
    def __init__(self, nombre, edad, direccion):
        self.nombre = nombre
        self.edad = edad
        self.direccion = direccion

class CuentaBancaria:
    def __init__(self, cliente, saldo_inicial=0):
        self.cliente = cliente
        self.saldo = saldo_inicial

    def depositar(self, cantidad):
        self.saldo += cantidad
            return f"Depósito exitoso. Nuevo saldo:
{self.saldo}"

    def retirar(self, cantidad):
        if cantidad <= self.saldo:
            self.saldo -= cantidad
                return f"Retiro exitoso. Nuevo saldo:
{self.saldo}"
        else:
            return "Fondos insuficientes."

    def obtener_saldo(self):
        return f"Saldo actual: {self.saldo}"

# Crear clientes
cliente1 = Cliente("Juan", 30, "Calle A")
cliente2 = Cliente("Ana", 25, "Calle B")

# Crear cuentas bancarias
cuenta1 = CuentaBancaria(cliente1, 1000)
cuenta2 = CuentaBancaria(cliente2, 500)

# Realizar operaciones bancarias
print(cuenta1.depositar(500))  # Saldo: 1500
print(cuenta2.retirar(200))    # Saldo: 300
print(cuenta1.obtener_saldo()) # Saldo: 1500
```

En este ejemplo, hemos definido dos clases: Cliente y CuentaBancaria. Cada cliente tiene un nombre, edad y dirección. Cada cuenta bancaria está asociada a un cliente y tiene un saldo inicial. La clase CuentaBancaria también tiene métodos para depositar, retirar y obtener el saldo.

Luego, creamos dos clientes (cliente1 y cliente2) y dos cuentas bancarias asociadas a esos clientes (cuenta1 y cuenta2). Realizamos algunas operaciones bancarias como depósitos y retiros, y consultamos los saldos de las cuentas.

Este ejemplo ilustra cómo las clases y objetos pueden utilizarse para modelar entidades del mundo real y facilitar la organización y operación de un programa. La POO proporciona una forma efectiva de abstraer y estructurar la lógica de un sistema.

Constructores 🛠️

Un constructor es un método especial en una clase que se llama automáticamente cuando se crea un objeto. En Python, el constructor se llama __init__ y se utiliza para inicializar los atributos del objeto.

Ejemplo Técnico:

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

# Crear un objeto de la clase Persona
persona1 = Persona("Juan", 25)
```

En el ejemplo anterior, __init__ es el constructor de la clase Persona que inicializa los atributos nombre y edad cuando se crea un nuevo objeto. La palabra clave self se refiere al propio objeto y se utiliza para acceder a sus atributos.

En Python, un constructor es un método especial que se llama automáticamente cuando se crea una instancia de una clase. El constructor se utiliza para realizar cualquier inicialización necesaria para la instancia. En Python, el constructor se llama __init__.

Constructor __init__:

El método __init__ es un constructor en Python y se llama automáticamente cuando se crea un objeto de la clase.

Es utilizado para realizar la inicialización de los atributos de la instancia.

```
class MiClase:
    def __init__(self, parametro1, parametro2, ...):
        # Inicialización de atributos
        self.atributo1 = parametro1
        self.atributo2 = parametro2
        # Otro código de inicialización si es necesario
```

Ejemplo Técnico:

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

Uso del Constructor:

```
# Creación de una instancia de la clase Persona
juan = Persona("Juan", 30)

# Acceso a los atributos inicializados en el constructor
print(juan.nombre)  # Salida: Juan
print(juan.edad)    # Salida: 30
```

En este ejemplo, el constructor __init__ se utiliza para inicializar los atributos nombre y edad cuando se crea una instancia de la clase Persona. La instancia juan tiene los valores proporcionados en la creación.

Constructor sin Parámetros:

```
class Animal:
    def __init__(self):
        self.tipo = "Desconocido"

# Creación de una instancia de la clase Animal sin proporcionar parámetros
animal_generico = Animal()
```

```
# Acceso al atributo inicializado en el constructor
print(animal_generico.tipo)  # Salida: Desconocido
```

Si el constructor no toma parámetros, aún debe tener el parámetro self. En este caso, el constructor de la clase Animal inicializa el atributo tipo con un valor predeterminado.

El constructor es una parte fundamental de la Programación Orientada a Objetos en Python, y su uso es común para garantizar que las instancias de una clase se inicialicen correctamente.

La función isinstance() ⚒️

La función isinstance() en Python se utiliza para verificar si un objeto es una instancia de una clase dada o de una tupla de clases. Su sintaxis básica es la siguiente:

```
persona1 = Persona("Juan", 25)
```

- objeto: El objeto cuya instancia se desea verificar.
- clase_o_tupla_de_clases: Una clase o una tupla de clases que se utilizará para la comparación.

La función devuelve True si el objeto es una instancia de la clase especificada o de alguna de las clases en la tupla, y False en caso contrario.

Ejemplos:

1. Verificar si un objeto es una instancia de una clase específica:

```
class Persona:
    pass

juan = Persona()
resultado = isinstance(juan, Persona)
print(resultado)  # Salida: True
```

2. Verificar si un objeto es una instancia de alguna clase en una tupla:

```
class Animal:
```

```
    pass

class Perro(Animal):
    pass

class Gato(Animal):
    pass

mi_perro = Perro()
resultado = isinstance(mi_perro, (Perro, Gato))
print(resultado)  # Salida: True
```

La función isinstance() es útil en situaciones donde necesitas verificar el tipo de un objeto dinámicamente, especialmente en contextos de herencia y polimorfismo. Puede ayudarte a tomar decisiones basadas en el tipo del objeto en tiempo de ejecución.

3. Uso con Herencia:

```
class Figura:
    pass

class Circulo(Figura):
    pass

class Rectangulo(Figura):
    pass

figura = Circulo()
resultado = isinstance(figura, Figura)
print(resultado)  # Salida: True
```

En este ejemplo, figura es una instancia de la clase Circulo, que hereda de la clase Figura. isinstance() aún devuelve True cuando se verifica contra la clase base.

La función isinstance() es útil en situaciones donde necesitas verificar el tipo de un objeto dinámicamente, especialmente en contextos de herencia y polimorfismo. Puede ayudarte a tomar decisiones basadas en el tipo del objeto en tiempo de ejecución.

Estados y atributos 📋

En Python, el "estado" de un objeto se refiere a los valores almacenados en sus atributos en un momento dado. Los "atributos" son variables asociadas a una instancia de una clase y definen las características del objeto.

```
class Persona:

    def __init__(self, nombre, edad):

        self.nombre = nombre  # Atributo: nombre

        self.edad = edad      # Atributo: edad
```

En este caso, nombre y edad son atributos de la clase Persona. El "estado" de una instancia particular de Persona estaría dado por los valores específicos de esos atributos.

Atributos y métodos 📝

Atributos:

Los atributos son variables que contienen información sobre una instancia de una clase. Pueden ser accesibles y modificables desde fuera de la clase, o pueden ser privados y tener acceso limitado.

```
class Coche:
    def __init__(self, marca, modelo):
        self.marca = marca    # Atributo público
        self.__modelo = modelo  # Atributo privado
```

Métodos:

Los métodos son funciones asociadas a una clase y se utilizan para realizar operaciones en los atributos de la clase. Pueden ser métodos de instancia, métodos de clase o métodos estáticos.

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

```
    def obtener_informacion(self):
        return f"{self.marca} {self.modelo}"

    @staticmethod
    def fabricar_coche():
        return Coche("Desconocida", "Desconocido")
```

En este ejemplo, obtener_informacion es un método de instancia que devuelve información sobre el coche, y fabricar_coche es un método estático que crea y devuelve una instancia de la clase Coche con valores predeterminados.

Uso en la Práctica:

```
mi_coche = Coche("Toyota", "Camry")
print(mi_coche.obtener_informacion())
  # Salida: Toyota Camry

coche_predeterminado = Coche.fabricar_coche()
print(coche_predeterminado.obtener_informacion())
  # Salida: Desconocida Desconocido
```

En este ejemplo, mi_coche es una instancia de la clase Coche con atributos específicos. Se accede a los atributos y se llama a un método para obtener información. Además, se utiliza el método estático fabricar_coche para crear un coche con valores predeterminados.

Atributos de Clase:

Los atributos de clase son compartidos por todas las instancias de una clase. Pueden ser accedidos usando la sintaxis Clase.atributo_clase.

```
class Estudiante:
    total_estudiantes = 0  # Atributo de clase

    def __init__(self, nombre):
        self.nombre = nombre
        Estudiante.total_estudiantes += 1
```

Métodos de Acceso (Getter y Setter):

Los métodos de acceso, como getters y setters, permiten controlar el acceso y la modificación de los atributos. Se utilizan para garantizar la encapsulación.

```python
class Empleado:
    def __init__(self, nombre, salario):
        self.__nombre = nombre

    def obtener_nombre(self):
        return self.__nombre

    def establecer_nombre(self, nuevo_nombre):
        if len(nuevo_nombre) > 0:
            self.__nombre = nuevo_nombre
```

Atributos de Solo Lectura:

Puedes crear atributos que solo se puedan leer, pero no modificar desde fuera de la clase, utilizando propiedades.

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

    @property
    def radio(self):
        return self.__radio
```

Atributos Dinámicos:

Python permite agregar nuevos atributos a una instancia de clase dinámicamente.

```python
class Animal:
    pass

perro = Animal()
perro.nombre = "Fido"  # Atributo añadido dinámicamente
```

Atributos con Valor Predeterminado:

Puedes asignar valores predeterminados a los atributos al definir la clase.python

```python
class Libro:
    def __init__(self, titulo, autor="Desconocido"):
        self.titulo = titulo
```

```
        self.autor = autor
```

Atributos Privados:

Aunque Python no tiene atributos verdaderamente privados, la convención es utilizar un guión bajo doble (__) como prefijo para indicar que un atributo debe ser tratado como privado.

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

Atributos de Solo Lectura (Solo Getter):

Si solo deseas permitir la lectura de un atributo, podes proporcionar un getter sin un setter correspondiente.

```
class SensorTemperatura:
    def __init__(self, temperatura):
        self.__temperatura = temperatura

    @property
    def temperatura(self):
        return self.__temperatura
```

Estas prácticas adicionales en el manejo de atributos en Python te permitirán adaptar la estructura de tus clases de acuerdo a las necesidades específicas de tu programa.

La función dir() 🗒️

La función dir() en Python se utiliza para obtener una lista de nombres definidos en un espacio de nombres específico, como el espacio de nombres de un módulo, una clase o incluso el espacio de nombres global. Esta función puede proporcionar una visión general de los atributos y métodos disponibles para un objeto.

Sintaxis:

```
dir([objeto])
```

objeto (opcional): El objeto del cual se desea obtener la lista de nombres. Si no se proporciona, dir() devuelve los nombres en el espacio de nombres actual.

Ejemplos de Uso:

1. En el Espacio de Nombres Global:

```
variables_globales = dir()
print(variables_globales)
```

Este código imprimirá una lista de los nombres en el espacio de nombres global, que incluirá variables, funciones y otros elementos definidos en el script.

2. En un Módulo o Clase:

```
import math

nombres_modulo_math = dir(math)
print(nombres_modulo_math)
```

En este caso, dir() se utiliza para obtener la lista de nombres disponibles en el módulo math. Podes hacer lo mismo con instancias de clases para ver los atributos y métodos asociados a esa instancia.

3. Para Explorar un Objeto:

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

juan = Persona("Juan", 30)
atributos_juan = dir(juan)
print(atributos_juan)
```

Acá, **dir()** se usa para explorar los atributos y métodos disponibles en la instancia juan de la clase Persona.

Salida Típica:

La salida de dir() es una lista de cadenas que representan los nombres en el espacio de nombres especificado. La lista puede incluir tanto nombres de variables como de métodos.

Es importante tener en cuenta que dir() no proporciona detalles sobre la funcionalidad de cada nombre; simplemente lista los nombres disponibles.

Para obtener información detallada sobre un objeto, podes utilizar help() junto con dir().

```
help(math.sqrt)
# Proporciona información detallada sobre la función sqrt del módulo math
```

La función dir() es una herramienta útil para la exploración y el descubrimiento en Python, especialmente cuando trabajas con módulos, clases y espacios de nombres.

Métodos especiales ⚒️

Métodos Especiales (Métodos Mágicos) en Python:

En Python, los "métodos especiales" o "métodos mágicos" son métodos con nombres especiales rodeados por doble guion bajo (__). Estos métodos permiten que las clases implementen comportamientos específicos y se utilizan en situaciones particulares.

Métodos especiales comunes:

__init__:

Método de inicialización que se llama cuando se crea una nueva instancia de la clase.

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

__str__:

Método llamado por la función str() y por la representación de cadenas (como print). Debe devolver una representación de cadena legible de la instancia.

```
class Persona:
    def __str__(self):
        return f"{self.nombre}, {self.edad} años"
```

__len__:

Método llamado por la función len(). Debe devolver la longitud de la instancia.

```
class ListaPersonalizada:
    def __init__(self, elementos):
        self.elementos = elementos

    def __len__(self):
        return len(self.elementos)
```

__getitem__ y __setitem__:

Métodos llamados para acceder o asignar valores utilizando la notación de corchetes ([]).

```
class ListaPersonalizada:
    def __getitem__(self, indice):
        return self.elementos[indice]

    def __setitem__(self, indice, valor):
        self.elementos[indice] = valor
```

La función help() 📋

La función help() se utiliza para obtener información sobre un objeto en Python, como una clase, módulo, función, o incluso un método. Puedes utilizarla en la consola interactiva de Python o en tu código.

```
help(list)
```

Este comando mostrará información detallada sobre la clase list y sus métodos, incluyendo la documentación de cada método.

Para obtener información específica sobre un objeto de instancia, podes usar help() con ese objeto:

```
mi_persona = Persona("Juan", 25)
help(mi_persona)
```

Esto proporcionará información sobre la clase Persona, incluyendo los métodos especiales y otros métodos definidos en la clase.

La función help() es una herramienta valiosa para explorar y comprender las capacidades de los objetos y clases en Python, y es especialmente útil cuando trabajas con bibliotecas o módulos externos.

Métodos de Clase y Métodos Estáticos 🛠️

En Python, los métodos de clase y los métodos estáticos son dos tipos de métodos que pueden ser definidos dentro de una clase. Aunque ambos tipos de métodos están asociados a la clase en lugar de a una instancia específica, tienen propósitos y comportamientos ligeramente diferentes.

Métodos de Clase:

Un método de clase está asociado con la clase y no con instancias específicas. Se utiliza para realizar operaciones que se aplican a la clase en su conjunto, en lugar de a instancias individuales.

Declaración:

Se declara con el decorador @classmethod y toma como primer parámetro la propia clase (cls por convención).

Uso Típico:

Los métodos de clase se utilizan a menudo para realizar operaciones que involucran a toda la clase, como la creación de instancias alternativas o el acceso a atributos de clase.

Ejemplo Técnico:

```
class Punto:
    total_puntos = 0

    def __init__(self, x, y):
        self.x = x
        self.y = y
        Punto.total_puntos += 1

    @classmethod
    def obtener_total_puntos(cls):
       return cls.total_puntos
```

Métodos Estáticos:

Un método estático es un método asociado con la clase y no con instancias específicas ni con la clase en sí. No tiene acceso a los atributos de la instancia ni de la clase.

Declaración:

Se declara con el decorador @staticmethod.

Uso Típico:

Los métodos estáticos se utilizan para operaciones que no dependen de instancias específicas ni de la clase, sino que son relevantes a nivel de la clase.

Ejemplo Técnico:

```
class Calculadora:
    @staticmethod
    def sumar(x, y):
        return x + y
```

Uso en la Práctica:

Métodos de Clase:

```
punto1 = Punto(1, 2)
punto2 = Punto(3, 4)

print(Punto.obtener_total_puntos())  # Salida: 2
```

Métodos Estáticos:

```
resultado = Calculadora.sumar(5, 3)
print(resultado)  # Salida: 8
```

Ambos tipos de métodos, de clase y estáticos, son herramientas útiles en Python para organizar y estructurar el código en clases, brindando flexibilidad y modularidad.

Abstracción 🛠️

La abstracción es un concepto fundamental en la Programación Orientada a Objetos (POO) que implica la creación de modelos simplificados y representaciones de la realidad para facilitar la comprensión y el manejo de sistemas complejos. En Python, la abstracción se logra mediante el uso de clases y objetos.

La abstracción consiste en identificar las características esenciales de un objeto y ocultar los detalles innecesarios. Proporciona una vista de alto nivel y permite al programador centrarse en la funcionalidad principal, sin preocuparse por la complejidad interna.

Implementación en Python:

```python
from abc import ABC, abstractmethod

# Clase abstracta (interfaz)
class Forma(ABC):
    @abstractmethod
    def area(self):
        pass

# Clases concretas que implementan la interfaz
class Cuadrado(Forma):
    def __init__(self, lado):
        self.lado = lado

    def area(self):
        return self.lado * self.lado

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

    def area(self):
        return 3.14 * self.radio * self.radio
```

En este ejemplo, Forma es una clase abstracta que define un método abstracto area(). Las clases concretas Cuadrado y Circulo implementan esta interfaz proporcionando su propia implementación del método area(). Al hacerlo, se

están abstrayendo las características esenciales de una forma (en este caso, el cálculo del área) y proporcionando una interfaz común.

👍**Ventajas de la Abstracción en Python:**

- Simplificación del Diseño: Permite enfocarse en la funcionalidad esencial sin preocuparse por detalles internos.
- Reutilización del Código: Facilita la creación de clases y objetos reutilizables al definir interfaces claras.
- Modularidad: Facilita la división del código en componentes independientes y bien definidos.

📌**Consejos para la Abstracción en Python:**

- Identificar Funcionalidades Clave: Determine las funciones clave que deben estar disponibles en una interfaz.
- Ocultar Detalles Internos: Oculte detalles de implementación innecesarios para simplificar el uso de la clase.
- Usar Clases Abstractas e Interfaces: Utilice clases abstractas e interfaces para definir una estructura común.

La abstracción en Python permite construir sistemas complejos de manera más clara y comprensible, facilitando la creación de software robusto y mantenible.

Encapsulación y ocultamiento 🔏

La encapsulación y el ocultamiento son conceptos relacionados en la Programación Orientada a Objetos que se utilizan para controlar el acceso a los atributos y métodos de una clase. En Python, la encapsulación y el ocultamiento no son implementados de manera estricta como en algunos otros lenguajes, pero se pueden lograr mediante convenciones y técnicas específicas.

Encapsulación:

La encapsulación es el proceso de limitar el acceso a los detalles internos de un objeto y ocultar la implementación de manera que solo ciertos métodos puedan interactuar con él.

Implementación en Python: Se logra prefijando el nombre del atributo con un guion bajo (_) para indicar que el atributo es de uso interno.

Ejemplo Técnico:

```
class Coche:
    def __init__(self, modelo):
            self._modelo = modelo    # Convención de
encapsulación

    def obtener_modelo(self):
        return self._modelo
```

Ocultamiento:

El ocultamiento va más allá de la encapsulación y implica esconder los detalles de implementación, incluso de clases derivadas. En Python, se utiliza un doble guion bajo (__) para indicar que un atributo es privado y para realizar un "name mangling".

Implementación en Python:

```
class CuentaBancaria:
    def __init__(self, saldo):
            self.__saldo = saldo    # Convención de
ocultamiento

    def obtener_saldo(self):
        return self.__saldo
```

Ambos conceptos, encapsulación y ocultamiento, ayudan a mejorar la seguridad y la integridad de la clase al restringir el acceso a ciertos atributos y métodos. Sin embargo, es importante destacar que en Python, estas son convenciones y no restricciones impuestas por el lenguaje. Un desarrollador puede acceder a atributos "privados" si es necesario, aunque esto no es una práctica recomendada a menos que sea absolutamente necesario. La filosofía de Python es "somos todos adultos aquí", confiando en que los desarrolladores actuarán de manera responsable.

Composición 🛠️

En Python, la composición es un concepto de programación orientada a objetos que permite crear objetos más grandes y complejos combinando varios objetos más pequeños. Esto se logra al incluir instancias de una clase dentro de otra clase como atributos. La clase que contiene las instancias de otras clases se conoce como la clase compuesta.

Para entender mejor la composición en Python, veamos un ejemplo. Supongamos que estamos modelando una tienda en línea y tenemos dos clases: `Producto` y `CarritoDeCompras`. La clase `Producto` representa un producto que se puede comprar y tiene atributos como `nombre`, `precio` y `cantidad en stock`. La clase `CarritoDeCompras` representa el carrito de compras de un cliente y contiene una lista de productos que el cliente ha seleccionado para comprar.

```
class Producto:
                def    __init__(self,    nombre,    precio,
cantidad_en_stock):
        self.nombre = nombre
        self.precio = precio
        self.cantidad_en_stock = cantidad_en_stock

class CarritoDeCompras:
    def __init__(self):
        self.productos = []

    def agregar_producto(self, producto):
        self.productos.append(producto)

    def calcular_total(self):
        total = 0
        for producto in self.productos:
            total += producto.precio
        return total
```

En este ejemplo, la clase `CarritoDeCompras` contiene una lista de productos (`self.productos`) como uno de sus atributos. Esto es un ejemplo de composición, ya que la clase `CarritoDeCompras` está compuesta por instancias de la clase `Producto`.

Podemos crear objetos de estas clases y utilizar la composición para agregar productos al carrito de compras y calcular el total de la compra:

```
# Creamos algunos productos
producto1 = Producto("Camiseta", 20, 5)
producto2 = Producto("Pantalón", 30, 3)
producto3 = Producto("Zapatos", 50, 2)

# Creamos un carrito de compras y agregamos productos
carrito = CarritoDeCompras()
carrito.agregar_producto(producto1)
carrito.agregar_producto(producto2)
carrito.agregar_producto(producto3)

# Calculamos el total de la compra
total_compra = carrito.calcular_total()
print("Total de la compra:", total_compra)
```

👉🏽En este ejemplo, hemos utilizado la composición para combinar varios objetos `Producto` y crear un objeto más grande y complejo `CarritoDeCompras`. La composición nos permite crear estructuras más complejas y reutilizables en nuestros programas, lo que facilita la organización y el mantenimiento del código.

Herencia y Polimorfismo en Python 🖥️

En Python, la herencia es un mecanismo mediante el cual una clase puede heredar atributos y métodos de otra clase, llamada clase base o superclase. La clase que hereda se conoce como clase derivada o subclase. Este concepto es fundamental en la programación orientada a objetos y permite crear jerarquías de clases, donde las clases derivadas pueden extender y especializar la funcionalidad de la clase base.

La sintaxis para definir una clase con herencia en Python es la siguiente:

```
class ClaseBase:
  # Definición de atributos y métodos de la clase base

class ClaseDerivada(ClaseBase):
 # Definición de atributos y métodos propios de la clase derivada
```

Al heredar de la clase base, la clase derivada adquiere todos los atributos y métodos de la clase base. Esto significa que la clase derivada puede utilizar y extender la funcionalidad definida en la clase base. Además, la clase derivada puede agregar nuevos atributos y métodos propios, lo que le permite especializarse y adaptarse a las necesidades específicas.

Un ejemplo sencillo de herencia en Python sería el siguiente:

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

    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        return "Guau!"

class Gato(Animal):
    def hacer_sonido(self):
        return "Miau!"
```

En este ejemplo, tenemos una clase base llamada Animal que tiene un atributo nombre y un método hacer_sonido(). Luego, creamos dos clases derivadas, Perro y Gato, que heredan de Animal. Cada clase derivada implementa su propio método hacer_sonido() para representar el sonido que hace cada animal.

La herencia es una herramienta poderosa en Python y nos permite crear jerarquías de clases que facilitan la reutilización de código y la organización de la funcionalidad en nuestros programas. Sin embargo, es importante usarla de manera adecuada y evitar jerarquías excesivamente complejas para mantener el código limpio y fácil de mantener.

Herencia simple 🖨️

La herencia simple es un concepto clave en la Programación Orientada a Objetos que permite que una clase, llamada subclase o clase derivada, herede los atributos y métodos de otra clase, llamada superclase o clase base.

```
class Animal:
```

```
    def __init__(self, nombre):
        self.nombre = nombre

    def hacer_sonido(self):
        pass  # Método a ser implementado por las subclases
```

En este ejemplo, Animal es la clase base que tiene un atributo nombre y un método hacer_sonido, que será implementado por las subclases.

Creación de una Subclase:

```
class Perro(Animal):
    def hacer_sonido(self):
        return "Guau"
```

La clase Perro es una subclase de Animal. Al heredar de Animal, la subclase Perro obtiene automáticamente el atributo nombre y el método hacer_sonido.

Creación de **Objetos y Uso** de la Herencia:

```
mi_perro = Perro("Buddy")
print(mi_perro.nombre)
# Acceder al atributo heredado de la superclase
print(mi_perro.hacer_sonido())
# Llamar al método implementado en la subclase
```

En este caso, mi_perro es un objeto de la subclase Perro, que hereda el atributo nombre y proporciona su propia implementación del método hacer_sonido.

👍**Ventajas de la Herencia Simple:**

- Reutilización de Código: Permite reutilizar el código de la clase base en la subclase.
- Extensibilidad: Permite agregar nuevos atributos y métodos a la subclase sin modificar la clase base.

👎**Desventajas de la Herencia Simple:**

- Jerarquías Profundas: Pueden volverse difíciles de gestionar si la jerarquía de herencia se vuelve demasiado profunda.

- Acoplamiento: Las subclases están fuertemente acopladas a la implementación de la superclase.

Herencia múltiple 💼

La herencia múltiple es un concepto en la Programación Orientada a Objetos que permite que una clase herede atributos y métodos de más de una clase base. En Python, esta funcionalidad está soportada. A continuación, se presenta un resumen de cómo se implementa la herencia múltiple:

Definición de Clases Base:

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

    def amamantar(self):
        pass

class Volador:
    def volar(self):
        pass
```

En este ejemplo, tenemos dos clases base: Mamifero y Volador. Cada una tiene sus propios métodos y atributos.

Creación de una **Subclase** con Herencia Múltiple:

```
class Murcielago(Mamifero, Volador):
    def amamantar(self):
        return "Amamanta a sus crías con leche."

    def volar(self):
        return "Vuela usando sus alas membranosas."
```

La clase Murciélago hereda de ambas clases base, Mamífero y Volador. Esto significa que la subclase Murciélago tiene tanto el método amamantar como el método volar.

Creación de **Objetos y Uso** de la Herencia Múltiple:

```
mi_murcielago = Murcielago("Battie")
```

```
print(mi_murcielago.amamantar())   # Llamada al método de la
clase base Mamifero
print(mi_murcielago.volar())       # Llamada al método de la
clase base Volador
```

En este caso, mi_murcielago es un objeto de la subclase Murcielago que hereda tanto de Mamifero como de Volador. Puede acceder a los métodos de ambas clases base.

👍**Ventajas de la Herencia Múltiple:**

- Reutilización de Código: Permite la reutilización de código desde múltiples fuentes.
- Diseño Modular: Facilita la creación de clases modulares que pueden ser combinadas de manera flexible.

👎 **Desventajas de la Herencia Múltiple:**

- Complejidad: Puede hacer que el código sea más complejo y difícil de entender.
- Problemas de Diamante: Pueden surgir conflictos si dos clases base comparten un método con el mismo nombre, creando lo que se conoce como el "problema del diamante".

Herencia múltiple

Python es un lenguaje de programación ampliamente utilizado en el ámbito del análisis de datos debido a su simplicidad y potencia. Para trabajar de manera eficiente en el análisis de datos, es crucial entender cómo importar módulos y cómo documentar las librerías que se utilizan.

Importación de Módulos en Python:

La importación de módulos en Python es esencial para acceder a funcionalidades adicionales. Existen varias formas de importar módulos:

Uso de la Declaración import:

```python
import modulo
```

Este método importa todo el módulo y requiere que utilice el nombre del módulo para acceder a sus funciones y variables.

Uso de la Declaración from...import:

```python
from modulo import funcion
```

Esta forma permite importar específicamente solo las funciones o variables que necesita del módulo, lo que puede hacer su código más legible y eficiente.

Documentación de Librerías en Python para Análisis de Datos

Documentar las librerías utilizadas es fundamental para comprender su funcionamiento y maximizar su uso.

Documentación en Python

Python cuenta con un sistema de documentación integrado llamado "docstrings". Estos son cadenas de texto ubicadas al comienzo de módulos, funciones, clases o métodos que explican su funcionamiento.

```python
def funcion():
    """Esta función realiza algo."""
    pass
```

Documentación de Librerías Específicas

Muchas librerías populares de Python tienen una documentación exhaustiva en línea que incluye tutoriales, guías de usuario y referencias de API. Por ejemplo, Pandas y NumPy tienen documentación completa en sus sitios web oficiales.

Ejemplos Prácticos:

```python
# Ejemplo de importación de módulos y documentación

import pandas as pd

# Utilizando la documentación de Pandas
# Consultamos la documentación de la función read_csv
help(pd.read_csv)
```

Documentación librerías:

```python
def my_function(param1, param2):
    """
    Esta función realiza una operación específica.

    Parameters:
    param1 (int): Descripción del primer parámetro.
    param2 (str): Descripción del segundo parámetro.

    Returns:
    bool: True si la operación fue exitosa, False de lo contrario.
    """
    # Cuerpo de la función
    return True
```

La primera línea después de la declaración de la función es una cadena de documentación entre comillas triples ("""). Esto es un docstring en Python.

La sección Parameters describe los parámetros que toma la función, seguidos de su tipo (entre paréntesis) y una descripción.

La sección Returns describe lo que devuelve la función, seguido de su tipo y una descripción.

Recuerda que la documentación adecuada ayuda a otros desarrolladores (y a ti mismo en el futuro) a entender cómo usar la función sin necesidad de leer el código fuente.

Además, muchas librerías de Python también proporcionan documentación en línea detallada, incluyendo ejemplos de uso, guías de referencia y tutoriales. Por ejemplo, NumPy y Pandas tienen documentación exhaustiva en sus sitios web oficiales:

Siempre es una muy buena práctica y a la documentación oficial:

Documentación de NumPy

Documentación de Pandas

Algunas otras librerías populares para análisis de datos que también tienen una buena documentación incluyen Matplotlib, Seaborn, Scikit-learn, TensorFlow, entre otras. Puedes encontrar más información sobre estas librerías y sus respectivas documentaciones en sus sitios web oficiales o en los repositorios de documentación de Python.

Cierre

En conclusión, tanto la herencia como la composición son conceptos fundamentales en la programación orientada a objetos en Python. Ambos mecanismos permiten crear jerarquías de clases y reutilizar código de manera eficiente, pero tienen enfoques diferentes para lograrlo.

Es importante tener en cuenta que tanto la herencia como la composición son herramientas poderosas, pero deben utilizarse de manera adecuada y consciente. En general, se prefiere la composición sobre la herencia cuando sea posible, ya que la composición promueve una mayor flexibilidad y evita problemas de dependencias complejas.

En resumen, la elección entre herencia y composición dependerá de la estructura y las necesidades específicas de cada proyecto. Al entender estas dos técnicas, podemos diseñar aplicaciones más robustas, flexibles y fáciles de mantener en Python.

Referencias

- https://barcelonageeks.com/herencia-y-composicion-en-python/
- Documentación de Pandas: https://pandas.pydata.org/docs/
- Documentación de NumPy: https://numpy.org/doc/

¡Muchas gracias!

Nos vemos en la próxima lección 👋