ÍNDICE

# 1. Introducción

En el mundo del desarrollo de proyectos en Python, es fundamental mantener un código organizado y estructurado. En este curso, nos enfocaremos en dos enfoques clave para lograr esto: el uso de módulos y paquetes, así como la programación orientada a objetos.

Los módulos y paquetes nos permiten dividir nuestro código en diferentes scripts, lo que facilita la gestión y reutilización de funciones y variables en distintas partes de nuestro proyecto. Además, aprenderemos cómo organizar nuestros scripts en carpetas para una mejor estructura y mantenimiento.

Por otro lado, la programación orientada a objetos (POO) es un paradigma esencial en Python. Con POO, encapsulamos datos y comportamientos en objetos, lo que nos brinda una forma más intuitiva y eficiente de trabajar con nuestra información. A través de clases y objetos, podemos crear estructuras complejas y aplicar herencia, lo que nos permite aprovechar al máximo la reutilización del código.

## 1.1. Objetivos

- Aprenderemos a crear y utilizar módulos y paquetes para dividir y organizar nuestro código de manera eficiente.
- Dominaremos las técnicas para importar módulos y paquetes de forma adecuada en nuestros proyectos.
- Conoceremos cómo documentar nuestros módulos y paquetes de manera profesional para una colaboración efectiva con otros desarrolladores.
- Aplicaremos los conceptos fundamentales de la programación orientada a objetos en Python para crear código más modular, mantenible y escalable.
- Exploraremos cómo utilizar la herencia para aprovechar la reutilización de código y construir jerarquías de clases sólidas.

## 2. Módulos y Paquetes

En esta sección, exploraremos a fondo el uso de módulos y paquetes en Python, dos herramientas esenciales para organizar y estructurar nuestro código en proyectos de cualquier escala. Aprenderemos cómo dividir nuestro código en fragmentos lógicos y funcionales, lo que nos permitirá mejorar la legibilidad, reutilización y mantenimiento de nuestras aplicaciones.

### 2.1. Módulos

Los módulos en Python son archivos que contienen funciones, variables y clases que pueden ser reutilizados en diferentes partes de nuestro programa. Permiten dividir el código en fragmentos lógicos y funcionales, lo que mejora la legibilidad y facilita el mantenimiento del código.

- **Creación de módulos:** Crear módulos personalizados en Python nos permite organizar nuestras funciones y clases relacionadas en archivos separados, lo que facilita la reutilización del código y mejora la mantenibilidad de nuestros proyectos. Aquí te guiaré a través de los pasos para crear un módulo personalizado:
    - En primer lugar, crea un archivo Python con las funciones y clases que deseas incluir en el módulo. Asegúrate de que el archivo tenga la extensión ".py". Por ejemplo, crearemos un módulo llamado "modulo_aritmetica.py" que contendrá algunas funciones matemáticas:

```python
# modulo_aritmetica.py

def suma(a, b):
    return a + b

def resta(a, b):
    return a - b

def multiplicacion(a, b):
    return a * b
```

    - Guarda el archivo en el mismo directorio que tu programa principal o en una ubicación donde puedas acceder a él.

- **Importación de módulos en nuestro código:** Una vez que hemos creado nuestros módulos personalizados, necesitamos importarlos en otros scripts para utilizar sus funciones y variables. Existen varias formas de importar módulos en Python, como "import", "from ... import" e "import ... as". Exploraremos cuándo y cómo utilizar cada método de importación.

```
# script principal: main.py
import modulo_aritmetica

resultado = modulo_aritmetica.suma(5, 3)
print("Suma:", resultado)

resultado = modulo_aritmetica.resta(10, 4)
print("Resta:", resultado)
```

En ocasiones, los nombres de los módulos pueden ser largos o poco intuitivos. Para mejorar la legibilidad del código, podemos utilizar alias o nombres cortos para referenciar a los módulos importados.

```
# script principal: main.py
import modulo_aritmetica as ma

resultado = ma.suma(5, 3)
print("Suma:", resultado)

resultado = ma.resta(10, 4)
print("Resta:", resultado)
```

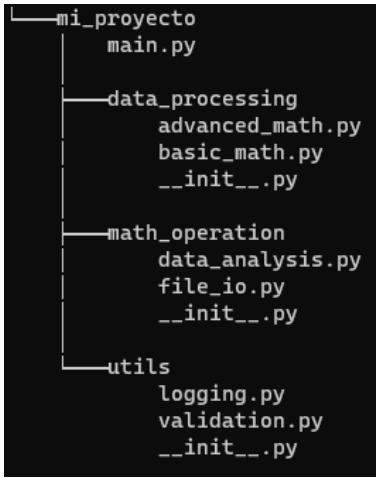
- **Exploración de los módulos de la biblioteca estándar de Python:** La biblioteca estándar de Python es una colección de módulos que vienen incluidos con la instalación básica del lenguaje. Estos módulos proporcionan una amplia gama de funcionalidades que cubren tareas comunes, desde operaciones del sistema hasta manipulación de datos, acceso a redes y mucho más.

```
import os
directorio_actual = os.getcwd()
print("Directorio actual:", directorio_actual)
```

- **Mejores prácticas para estructurar y diseñar módulos en Python:** Una estructura de módulos bien organizada y diseñada es esencial para proyectos grandes y complejos, ya que facilita la legibilidad, el mantenimiento y la reutilización del código. Aquí tienes algunas mejores prácticas para diseñar y organizar módulos de manera efectiva:
    - **Nombres descriptivos:** Nombra los módulos de manera coherente y descriptiva para que sea fácil entender su propósito. Utiliza nombres en minúsculas y, si es necesario, separa palabras con guiones bajos (snake_case). Evita nombres demasiado genéricos o ambiguos.
    - **División de funciones relacionadas:** Agrupa funciones y clases relacionadas en módulos separados. Cada módulo debería tener un propósito claro y centrarse en una tarea o funcionalidad específica. Esto promueve la cohesión y facilita la navegación dentro del proyecto.
    - **Módulos pequeños y cohesivos:** Es preferible tener varios módulos pequeños y cohesivos en lugar de un solo módulo grande y monolítico. Esto mejora la mantenibilidad y permite una reutilización más eficiente del código.
    - **Evitar módulos con demasiadas dependencias:** Intenta evitar que un módulo tenga demasiadas dependencias externas, ya que esto puede complicar su uso y dificultar la comprensión del código.
    - **Evitar dependencias circulares:** Asegúrate de que no haya dependencias circulares entre módulos, ya que esto puede causar problemas en la ejecución y es una mala práctica de diseño.
    - **Separación de preocupaciones:** Diseña módulos que sigan el principio de "separación de preocupaciones". Cada módulo debe centrarse en resolver una tarea específica sin mezclar lógica de diferentes áreas.
    - **Organización de directorios:** Utiliza una estructura de directorios lógica para organizar tus módulos. Agrupa módulos relacionados en directorios temáticos o funcionales.

- **Evitar importaciones excesivas:** Evita importar módulos o funciones innecesarias en tu código. Importa solo lo que necesitas para reducir la complejidad y mejorar el rendimiento.
- **Documentación:** Proporciona documentación adecuada para tus módulos y funciones. Utiliza docstrings para describir el propósito, la entrada y la salida de las funciones, y añade comentarios explicativos si es necesario.
- **Mantenimiento regular:** Mantén tus módulos actualizados y realiza mantenimiento regularmente. Elimina funciones o clases obsoletas y refactoriza el código según sea necesario para mantenerlo limpio y eficiente.

**Ejemplo de estructura de directorios para un proyecto:**

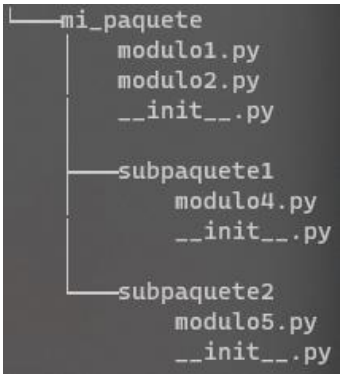



## 2.2. Paquetes

Los paquetes en Python son directorios que contienen módulos relacionados, permitiendo una organización más avanzada del código. Son una extensión natural de los módulos y nos ayudan a estructurar proyectos más grandes y complejos.

- Un paquete es simplemente un directorio que contiene un archivo especial llamado __init__.py. Este archivo es obligatorio y se utiliza para convertir el directorio en un paquete válido de Python. Puedes dejarlo en blanco, pero también puede contener código de inicialización o importaciones que se ejecutarán cuando se importe el paquete.

- Los paquetes pueden contener otros paquetes (subpaquetes) y/o módulos. Esto permite una organización jerárquica y modular de tu código.
- Los paquetes nos permiten evitar conflictos de nombres y mantener la cohesión temática entre módulos y subpaquetes. Por ejemplo, si tienes varios módulos con funciones llamadas "util", puedes agruparlos en un paquete llamado "utilidades" para evitar confusiones.
- La estructura de directorios de los paquetes sigue la estructura de los nombres de los paquetes. Por ejemplo, si tienes un paquete llamado "mi_paquete" que contiene un módulo llamado "mi_modulo", la estructura de directorios sería: mi_paquete/ mi_modulo.py.
- Para importar módulos desde un paquete, se utiliza la sintaxis de puntos. Por ejemplo, si tienes un paquete llamado "mi_paquete" que contiene un módulo llamado "mi_modulo", la importación se vería así: from mi_paquete import mi_modulo.




**Ejemplo de Importación de módulos**

```python
# script principal: main.py
from mi_paquete import modulo1, modulo2
from mi_paquete.subpaquete1 import modulo4

resultado = modulo1.suma(5, 3)
print("Suma:", resultado)

resultado = modulo2.resta(10, 4)
print("Resta:", resultado)

resultado = modulo4.multiplicacion(6, 2)
print("Resta:", resultado)
```

**Paquetes como contenedores de módulos relacionados:**

- **Mejora la cohesión:** Al agrupar módulos que comparten una temática o funcionalidad común en un paquete, aumentamos la cohesión del código. Esto significa que los módulos dentro del paquete están relacionados lógicamente y tienen una responsabilidad similar, lo que facilita el mantenimiento y la comprensión del proyecto.
- **Organización lógica:** Los paquetes permiten una organización más lógica de los módulos, lo que simplifica la navegación y búsqueda de funcionalidades específicas en un proyecto más grande. En lugar de tener todos los módulos en un solo directorio, los paquetes nos permiten dividir el código en unidades más manejables.
- **Evitar conflictos de nombres:** Al agrupar módulos relacionados en un paquete, se evitan conflictos de nombres entre funciones y clases que pueden tener el mismo nombre, pero con propósitos diferentes. Cada paquete proporciona un espacio de nombres independiente.
- **Reutilización y mantenibilidad:** Los paquetes facilitan la reutilización del código en diferentes proyectos, ya que puedes importar y utilizar todo el paquete o módulos individuales donde sea necesario. Esto también mejora la mantenibilidad, ya que los cambios en un módulo relacionado conllevan una menor probabilidad de afectar otros módulos no relacionados.
- **Facilita la colaboración en equipo:** Al utilizar paquetes, la colaboración en equipo se vuelve más sencilla, ya que los desarrolladores pueden trabajar en módulos y paquetes específicos sin afectar otras partes del proyecto. Además, los paquetes proporcionan una estructura clara y definida que ayuda a los miembros del equipo a comprender la arquitectura general del proyecto.

## 3. Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) es un paradigma de programación que se centra en organizar y estructurar el código en torno a objetos. Un objeto es una representación concreta de una entidad del mundo real que posee características y comportamientos. La POO busca modelar el mundo real en el código, lo que facilita la creación de programas más mantenibles, escalables y reutilizables.

### 3.1. Clase y Objeto:

Una **clase** en la POO es una plantilla o molde que define la estructura y el comportamiento de los objetos que se van a crear. La estructura de una clase se compone de atributos y métodos. Los atributos son variables que representan las características del objeto, mientras que los métodos son funciones que definen el comportamiento del objeto. Una clase puede tener uno o varios constructores, que son métodos especiales utilizados para inicializar los atributos del objeto cuando se crea una nueva instancia de la clase.

Un **objeto** es una instancia concreta de una clase. Representa un elemento específico del mundo real y tiene sus propios valores para los atributos definidos en la clase. Los objetos se crean a partir de una clase usando la sintaxis de llamada a la clase con paréntesis. Cada objeto creado a partir de la misma clase tendrá sus propios valores para los atributos.

```
class Persona:
    def saludar(self, nombre, edad):
        print(f"Hola, mi nombre es {nombre} y tengo {edad} años.")
```

**Nota:

En Python, self es una convención utilizada para referirse al propio objeto dentro de una clase. Cuando defines métodos en una clase, el primer parámetro de esos métodos se llama self. Este parámetro hace referencia a la instancia del objeto en sí mismo, lo que permite acceder a sus atributos y métodos.

Cuando llamas a un método en una instancia de clase, Python automáticamente pasa la referencia a la instancia como el primer argumento (self) al método. Esto te permite manipular y acceder a los atributos específicos de esa instancia. La palabra self no es una palabra clave en sí misma, pero es una convención ampliamente aceptada para este propósito.

### 3.2. Encapsulación:

La encapsulación es un concepto clave en la POO que permite ocultar los detalles internos de un objeto y exponer solo las funcionalidades necesarias para interactuar con él. En Python, la encapsulación se logra mediante el uso de atributos y métodos con modificadores de acceso como public, private o protected (aunque en Python no existen modificadores de acceso estrictos).

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

    def saludar(self):
        print(f"Hola, mi nombre es {self._nombre} y tengo {self._edad}
años.")
```

Se utiliza una convención para indicar que un atributo o método es privado o interno, mediante el uso de un guión bajo al inicio del nombre, como _atributo o _metodo. __init__ es un método especial en Python que se llama automáticamente cuando se crea una nueva instancia (objeto) de una clase. Es conocido como el "constructor" de la clase. El nombre __init__ es un nombre reservado en Python para este método.

****Nota**

__init__ es un método especial en Python que se llama automáticamente cuando se crea una nueva instancia de una clase. Es el constructor de la clase y se utiliza para inicializar los atributos y realizar cualquier configuración necesaria en el momento de la creación del objeto.

El nombre __init__ es reservado y no debes cambiarlo. Si defines este método en una clase, Python lo llamará automáticamente cuando crees un objeto de esa clase utilizando la sintaxis de instanciación, como objeto = Clase().

El método __init__ puede aceptar cualquier número de argumentos, pero el primer argumento siempre debe ser self, que hace referencia a la instancia del objeto que se está creando. A través de self, puedes acceder y configurar los atributos de la instancia.

## 3.3. Abstracción

La abstracción es un principio que permite simplificar y representar objetos complejos del mundo real mediante clases con atributos y métodos. Al utilizar la abstracción, podemos enfocarnos solo en las características y comportamientos relevantes de un objeto y omitir los detalles innecesarios.

Por ejemplo, si tenemos una clase Coche, podemos definir atributos como modelo, color, velocidad, etc., y métodos como acelerar(), frenar(), etc., sin preocuparnos por cómo se implementan internamente esas funcionalidades.

```
# Definición de la clase Coche con abstracción

class Coche:
    def __init__(self, modelo, color):
        self.modelo = modelo
        self.color = color
        self.velocidad = 0

    def acelerar(self, incremento):
        self.velocidad += incremento

    def frenar(self, decremento):
        if self.velocidad >= decremento:
            self.velocidad -= decremento
        else:
            self.velocidad = 0

# Uso de la clase Coche

# Creamos una instancia del coche
mi_coche = Coche("Sedan", "Rojo")

# Aceleramos el coche
```

```
mi_coche.acelerar(30)
print("Velocidad actual:", mi_coche.velocidad)

# Frenamos el coche
mi_coche.frenar(10)
print("Velocidad actual después de frenar:", mi_coche.velocidad)
```

### 3.4. Herencia

La herencia es un mecanismo mediante el cual una clase (denominada "clase hija" o "subclase") puede heredar atributos y métodos de otra clase (denominada "clase padre" o "superclase"). Esto permite reutilizar código y crear jerarquías de clases.

En Python, la herencia se especifica entre paréntesis al definir una clase. La clase hija tendrá acceso a los atributos y métodos públicos y protegidos de la clase padre. Esto facilita la creación de clases especializadas que comparten características con la clase padre pero también pueden tener sus propias funcionalidades adicionales.

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

    def emitir_sonido(self):
        pass  # Los animales generalmente no hacen sonidos específicos,
esta es una clase abstracta

class Perro(Animal):
    def __init__(self, nombre, raza):
        super().__init__(nombre)
        self.raza = raza

    def emitir_sonido(self):
        return "Guau, guau!"
```

En este ejemplo, la clase Perro hereda de la clase Animal. La clase Perro tiene su propio atributo raza y un método emitir_sonido() que es específico para

los perros, pero también hereda el atributo nombre y el método emitir_sonido() de la clase Animal.

Animal: Esta es una clase base que representa a un animal genérico. Tiene un constructor __init__ que acepta un parámetro nombre y lo asigna al atributo self.nombre. También tiene un método llamado emitir_sonido, que se define como pass. Esto indica que esta clase es abstracta y se espera que las clases derivadas proporcionen una implementación para este método.

Perro: Esta clase hereda de la clase Animal. Tiene un constructor __init__ que acepta dos parámetros, nombre y raza, y llama al constructor de la clase base usando super().__init__(nombre) para inicializar el atributo nombre. Además, se inicializa el atributo raza con el valor proporcionado. La clase Perro también tiene un método emitir_sonido, que devuelve el string "Guau, guau!".

****nota:**

super() es una función incorporada en Python que se utiliza para acceder y llamar a los métodos de una clase padre (superclase) en una clase hija (subclase). Permite una forma más flexible y dinámica de llamar a los métodos de la superclase en lugar de referenciarlos directamente por su nombre.

### 3.5. Polimorfismo

El polimorfismo es la capacidad de diferentes clases de responder a una misma función o método de manera diferente. Esto permite utilizar un mismo método con diferentes objetos y obtener comportamientos distintos según el tipo de objeto.

El polimorfismo se logra a través del uso de la herencia y la sobrescritura de métodos. Cuando una clase hija redefine un método de la clase padre, se dice que está sobrescribiendo el método y el comportamiento se adapta al de la clase hija.

```
class Figura:
    def calcular_area(self):
        pass

class Circulo(Figura):
    def __init__(self, radio):
```

```
        self.radio = radio

    def calcular_area(self):
        return 3.1416 * self.radio**2

class Cuadrado(Figura):
    def __init__(self, lado):
        self.lado = lado

    def calcular_area(self):
        return self.lado**2
```

En este ejemplo, las clases Circulo y Cuadrado sobrescriben el método calcular_area() de la clase Figura para calcular el área específica de cada figura.

Figura: Esta es la clase base que define un método abstracto calcular_area(). Al ser abstracto, no proporciona una implementación concreta para el cálculo del área. Esto significa que todas las subclases deben implementar su propio método calcular_area().

Circulo: Esta clase hereda de Figura. Tiene un constructor que acepta un parámetro radio y lo asigna al atributo self.radio. Además, proporciona su propia implementación del método calcular_area(), que utiliza la fórmula del área del círculo para realizar el cálculo.

Cuadrado: Al igual que Circulo, esta clase también hereda de Figura. Tiene un constructor que acepta un parámetro lado y lo asigna al atributo self.lado. La clase Cuadrado también proporciona su propia implementación del método calcular_area(), utilizando la fórmula del área del cuadrado.

## 4. Encapsulación y Abstracción

Encapsulación y Abstracción son dos conceptos fundamentales en la Programación Orientada a Objetos (POO). A continuación, profundizaremos en cada uno de ellos y también hablaremos sobre la creación de interfaces y clases abstractas, así como la herencia y la creación de jerarquías de clases

La encapsulación es un mecanismo que permite ocultar los detalles internos de un objeto y proteger sus atributos y métodos para que solo puedan ser accedidos o modificados a través de métodos específicos. Es un principio clave para mantener la integridad de los datos y para evitar modificaciones no autorizadas o accidentales desde el exterior de la clase.

En Python, a diferencia de otros lenguajes de programación que tienen modificadores de acceso explícitos (como public, private y protected), el encapsulamiento se logra mediante convenciones y el uso de guiones bajos en el nombre de los atributos y métodos.

- **Atributos privados:** Se definen con un guion bajo al inicio del nombre del atributo, como _nombre_atributo. Esto indica que el atributo debería ser tratado como privado, y no se debería acceder o modificar directamente desde fuera de la clase.
- **Métodos privados:** Se definen de manera similar con un guion bajo al inicio del nombre del método, como _nombre_metodo(). Los métodos privados son utilizados generalmente para realizar operaciones internas dentro de la clase.

### 4.1. Métodos getter y setter:

Los métodos getter y setter son métodos públicos que se utilizan para acceder y modificar los atributos privados de una clase de manera controlada. Los getter obtienen el valor de un atributo y los setter permiten cambiar el valor de un atributo, asegurando que se realicen validaciones u operaciones adicionales si es necesario.

En Python, los getter y setter se definen utilizando el decorador @property para el getter y @nombre_atributo.setter para el setter.

**Ejemplo de uso de getter y setter:**

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

    @property
    def nombre(self):
        return self._nombre

    @nombre.setter
    def nombre(self, nuevo_nombre):
        self._nombre = nuevo_nombre

persona = Persona("Juan", 30)
print(persona.nombre)  # Salida: "Juan"
persona.nombre = "Pedro"
print(persona.nombre)  # Salida: "Pedro"
```

Se define la clase Persona con un constructor __init__ que recibe dos parámetros nombre y edad. El constructor inicializa los atributos privados _nombre y _edad con los valores proporcionados en los parámetros.

Se utiliza el decorador @property para crear un método nombre() que se comporta como un getter para el atributo _nombre. Esto significa que podemos acceder al valor de _nombre desde fuera de la clase a través de este método como si fuera un atributo normal, sin necesidad de usar paréntesis. A continuación, se utiliza el decorador @nombre.setter para crear un método nombre() que actúa como un setter para el atributo _nombre. Esto permite cambiar el valor del atributo _nombre desde fuera de la clase utilizando la sintaxis de asignación.

### 4.2. Abstracción:

La abstracción es el proceso de identificar las características y comportamientos esenciales de un objeto del mundo real y representarlos en forma de clases y métodos. Permite enfocarse en los aspectos relevantes del objeto y ocultar los detalles innecesarios.

En POO, la abstracción se logra mediante el uso de interfaces y clases abstractas:

- **Interfaces:** Una interfaz es una especificación de métodos que deben ser implementados por las clases que la heredan. En Python, las interfaces se definen utilizando clases abstractas con métodos abstractos. Una clase abstracta es una clase que no puede ser instanciada directamente y debe ser subclaseada para ser utilizada.

```
from abc import ABC, abstractmethod

class Figura(ABC):
    @abstractmethod
    def calcular_area(self):
        pass
```

- **Clases abstractas:** Una clase abstracta es una clase que contiene uno o más métodos abstractos. Estas clases se utilizan para proporcionar una estructura común para las clases que la heredan, pero no se pueden instanciar directamente.

```
class Figura(ABC):
    @abstractmethod
    def calcular_area(self):
        pass

class Cuadrado(Figura):
    def __init__(self, lado):
        self.lado = lado

    def calcular_area(self):
        return self.lado**2

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

    def calcular_area(self):
        return 3.1416 * self.radio ** 2
# No se puede instanciar directamente una clase abstracta
# figura = Figura()  # Esto generaría un TypeError

# Creamos objetos de las clases concretas
cuadrado = Cuadrado(5)
circulo = Circulo(3)

# Llamamos al método calcular_area de cada objeto
```

```
print("Área del cuadrado:", cuadrado.calcular_area())  # Salida:
"Área del cuadrado: 25"
print("Área del círculo:", circulo.calcular_area())    # Salida:
"Área del círculo: 28.2744"
```

En este ejemplo, hemos definido una clase abstracta Figura que contiene un método abstracto calcular_area(). Esta clase no puede ser instanciada directamente, sino que se utiliza como base para otras clases concretas que heredan de ella.

Luego, hemos definido dos clases concretas: Cuadrado y Circulo, que heredan de la clase Figura. Cada una de estas clases implementa su versión del método calcular_area(), que calcula el área del cuadrado y el círculo respectivamente.

Al final, hemos creado objetos de las clases Cuadrado y Circulo, y llamamos al método calcular_area() en cada objeto para obtener el área correspondiente.

El uso de abstracción nos permite definir una interfaz común para todas las formas geométricas (en este caso, el método calcular_area()), mientras que cada clase concreta proporciona su propia implementación específica del cálculo del área. Esto facilita la extensibilidad y la claridad del código en nuestro programa.

## 5. Polimorfismo y Composición

El polimorfismo es un concepto fundamental en la Programación Orientada a Objetos (POO) que se refiere a la capacidad de diferentes clases de responder a una misma función o método de manera diferente. En otras palabras, diferentes objetos pueden interpretar un mismo mensaje o invocar un mismo método, pero actuarán de forma particular según su tipo o clase.

El polimorfismo permite tratar objetos de distintas clases de manera uniforme, lo que facilita la reutilización de código y mejora la flexibilidad del diseño. Gracias al polimorfismo, podemos crear código más genérico y fácil de mantener, ya que podemos utilizar un mismo método para diferentes objetos sin preocuparnos por su tipo específico.

### A. Aplicaciones del polimorfismo:

- **Sobreescritura de métodos:** En la herencia, una subclase puede redefinir (sobrescribir) un método heredado de su clase padre para ajustarlo a su comportamiento específico. Esto permite que un método con el mismo nombre en diferentes clases tenga un comportamiento diferente.
- **Interfaces:** Al definir una interfaz, especificamos qué métodos deben implementar las clases que la heredan. Cada clase puede proporcionar su propia implementación de los métodos, permitiendo que diferentes clases interactúen de manera polimórfica a través de la interfaz.
- **Polimorfismo paramétrico:** Al utilizar genéricos o plantillas (dependiendo del lenguaje de programación), podemos crear estructuras de datos o clases que trabajen con diferentes tipos de datos. Esto permite reutilizar el mismo código para diferentes tipos de objetos.

### B. Sobrecarga de métodos y operadores:

La sobrecarga de métodos y operadores es la capacidad de definir múltiples métodos con el mismo nombre o operador, pero con diferentes parámetros o comportamientos. Esto se denomina polimorfismo ad-hoc, ya que el método o el operador actuará de manera diferente según los parámetros con los que se llame.

En Python, no se permite la verdadera sobrecarga de métodos en el sentido de otros lenguajes como Java o C++, donde se pueden definir múltiples métodos con el mismo nombre y diferentes listas de argumentos. Sin embargo, Python sí permite el uso de valores predeterminados en los argumentos, lo que da lugar a un comportamiento similar a la sobrecarga.

```
class Calculadora:
    def sumar(self, a, b):
        return a + b

    def sumar(self, a, b, c):
        return a + b + c

# Creamos un objeto de la clase Calculadora
calc = Calculadora()

# Llamamos a los métodos sumar con diferente cantidad de argumentos
print(calc.sumar(2, 3))      # Salida: Error, ya que falta un argumento
para el segundo método sumar
print(calc.sumar(2, 3, 5))   # Salida: 10, se llama al segundo método
sumar
```

En este ejemplo, el segundo método sumar sobrescribe al primero debido a que ambos tienen el mismo nombre. Si intentamos llamar al primer método con dos argumentos, Python lanzará un error.

## C. Composición: Relación entre objetos

La composición es una relación entre objetos en la que un objeto contiene o está compuesto por otros objetos. Es una forma de asociación entre clases, donde una clase es "compuesta" por instancias de otras clases. La composición permite crear objetos complejos mediante la combinación de objetos más simples y reutilizar código de manera efectiva.

La relación entre los objetos es generalmente una relación "tiene un" o "está compuesto por". Esto significa que un objeto puede tener otros objetos como atributos, y estos objetos están completamente contenidos dentro del objeto principal.

```
class Motor:
    def encender(self):
        print("Motor encendido")

    def apagar(self):
        print("Motor apagado")

class Auto:
    def __init__(self):
        self.motor = Motor()

    def encender_auto(self):
        self.motor.encender()

    def apagar_auto(self):
        self.motor.apagar()

auto = Auto()
auto.encender_auto()  # Salida: "Motor encendido"
auto.apagar_auto()    # Salida: "Motor apagado"
```

En este ejemplo, la clase Auto tiene un atributo motor, que es una instancia de la clase Motor. La clase Auto está compuesta por un motor, y los métodos encender_auto() y apagar_auto() utilizan el motor para encender y apagar el auto respectivamente.

**Ejemplo de Uso de clases y objetos en un escenario práctico:**

Imaginemos un escenario práctico en el que estamos desarrollando un sistema para una biblioteca. Podríamos modelar este sistema utilizando clases y objetos de la siguiente manera:

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

class Biblioteca:
    def __init__(self):
        self.libros = []

    def agregar_libro(self, libro):
        self.libros.append(libro)
```

```
    def mostrar_libros(self):
        for libro in self.libros:
            print(f"{libro.titulo} - {libro.autor}")

# Crear algunos libros
libro1 = Libro("El principito", "Antoine de Saint-Exupéry")
libro2 = Libro("Don Quijote de la Mancha", "Miguel de Cervantes")

# Crear una biblioteca
biblioteca = Biblioteca()

# Agregar libros a la biblioteca
biblioteca.agregar_libro(libro1)
biblioteca.agregar_libro(libro2)

# Mostrar los libros en la biblioteca
biblioteca.mostrar_libros()
```

En este escenario, hemos creado dos clases: Libro y Biblioteca. La clase Libro representa un libro con atributos titulo y autor, mientras que la clase Biblioteca representa una colección de libros con un atributo libros que es una lista de objetos Libro. La clase Biblioteca tiene métodos para agregar libros a la biblioteca y mostrar la lista de libros que contiene.

## 6. Herencia y Polimorfismo

### A. Herencia múltiple y resolución de conflictos

En Python, es posible realizar herencia múltiple mediante la declaración de una clase que herede de dos o más clases superiores. Sin embargo, cuando una clase hereda de varias clases superiores que tienen métodos o atributos con el mismo nombre, puede surgir un conflicto en cuanto a cuál método o atributo debe utilizar la clase hija.

Veamos un ejemplo para entender cómo se resuelve el conflicto de herencia múltiple en Python:

```
class A:
    def metodo(self):
        print("Método de clase A")

class B:
    def metodo(self):
        print("Método de clase B")

class C(A, B):
    pass

objeto_c = C()
objeto_c.metodo()
```

En este ejemplo, tenemos tres clases: A, B y C. Tanto A como B tienen un método llamado metodo(). La clase C hereda de ambas clases, A y B. Cuando llamamos al método metodo() en un objeto de la clase C, Python utiliza la implementación del método de la clase más a la izquierda en la lista de clases superiores que se está heredando. En este caso, la clase C hereda de A primero y luego de B, por lo que la implementación de metodo() de la clase A tiene prioridad y se utiliza.

### B. Clases abstractas y herencia múltiple

En Python, podemos implementar clases abstractas utilizando el módulo abc y la clase ABC como base. Un método abstracto es un método que se

declara en la clase abstracta pero no se proporciona una implementación. Las clases hijas deben implementar los métodos abstractos de la clase abstracta.

Veamos un ejemplo de cómo se utilizan las clases abstractas con herencia múltiple:

```
from abc import ABC, abstractmethod

class Figura(ABC):
    @abstractmethod
    def calcular_area(self):
        pass

class Cuadrado(Figura):
    def __init__(self, lado):
        self.lado = lado

    def calcular_area(self):
        return self.lado ** 2

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

    def calcular_area(self):
        return 3.1416 * self.radio ** 2

class Triangulo(Figura):
    def __init__(self, base, altura):
        self.base = base
        self.altura = altura

    def calcular_area(self):
        return (self.base * self.altura) / 2

# Crear objetos de las clases que implementan la clase abstracta Figura
cuadrado = Cuadrado(5)
circulo = Circulo(3)
triangulo = Triangulo(4, 6)

# Llamamos al método calcular_area con diferentes objetos
print("Área del cuadrado:", cuadrado.calcular_area())  # Salida: "Área
del cuadrado: 25"
print("Área del círculo:", circulo.calcular_area())    # Salida: "Área
del círculo: 28.2744"
print("Área del triángulo:", triangulo.calcular_area())  # Salida: "Área
del triángulo: 12.0"
```

En este ejemplo, hemos definido una clase abstracta Figura con un método abstracto calcular_area(). Luego, hemos creado tres clases, Cuadrado, Circulo y Triangulo, que heredan de la clase abstracta Figura e implementan su propio método calcular_area(). Estas clases proporcionan diferentes implementaciones para calcular el área de un cuadrado, círculo y triángulo.

### C. Polimorfismo con interfaces:

El polimorfismo con interfaces se logra mediante la implementación de interfaces por parte de diferentes clases. Una interfaz define una lista de métodos que deben ser implementados por las clases que la heredan, pero no proporciona una implementación concreta de esos métodos.

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def hacer_sonido(self):
        pass

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

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

def hacer_ruido(animal):
    print(animal.hacer_sonido())

# Crear objetos de las clases que implementan la interfaz Animal
perro = Perro()
gato = Gato()

# Llamamos a la función hacer_ruido con diferentes objetos
hacer_ruido(perro)  # Salida: "Guau, guau!"
hacer_ruido(gato)   # Salida: "Miau, miau!"
```

En este ejemplo, hemos definido una interfaz Animal con un método abstracto hacer_sonido(). Luego, hemos creado dos clases, Perro y Gato, que

heredan de la interfaz Animal e implementan su propio método hacer_sonido(). La función hacer_ruido utiliza polimorfismo con la interfaz Animal, lo que le permite imprimir el sonido de diferentes objetos de clases que implementan dicha interfaz.

El polimorfismo con interfaces nos permite tratar objetos de diferentes clases de manera uniforme cuando implementan una interfaz común, lo que facilita la creación de código reutilizable y fácil de mantener. En este ejemplo, hemos podido llamar al método hacer_sonido() de diferentes clases Perro y Gato a través de la misma función hacer_ruido() sin importar sus detalles de implementación.

# ANEXO

# Python: Métodos especiales de clases

Está página este pensada para ser leida como una continuación al tutorial de Python de Guido Van Rossum, que puede obtenerse en http://www.python.org. y en esta misma web en la sección de El Bazar. Trata algunos conceptos algo más avanzados (aunque no por ello muy difíciles de entender) que nos permitirán aprovechar las características más avanzadas del lenguaje.

**Nota:** A lo largo de esta guía se utilizará el nombre "objeto" para referinos a una instancia de una clase que contiene el método que estemos tratando.

Las clases tienen una serie de métodos con un significado especial. Por ejemplo, ya conocemos el significado del método __init__() que se utiliza como constructor de clase, pero hay otros también muy interesantes. Vamos a verlos:

## Personalización Básica

**__init__(self, args...)**

Constructor del objeto. Si es una clase derivada, es importante llamar al método __init__ de la clase base dentro del __init__ de la clase derivada, para asegurar una inicialización adecuada.

**__del__(self)**

Es el destructor del objeto. Se le llama cuando la instancia va a ser borrada. De nuevo, si la clase base tiene un método __del__, el de la clase derivada debe llamarlo para asegurar una limpieza adecuada. No se garantiza que los métodos __del__() de todas las instancias de clase existentes puedan ser llamados en el momento en el que el interprete acabe.

del x no tiene porqué llamar automáticamente al destructor del objeto x. Lo que hace es decrementar la cuenta de referencias a x, y si esa cuenta llega a cero entonces si que se llama a __del__.

Dadas las circunstancias bajo las que se llama a __del__, dentro de este las excepciones son ignoradas (se imprime una advertencia en sys.stderr). Por ello __del__ debería realizar lo mínimo imprescindible en cuanto a la limpieza del objeto.

**__repr__(self)**

Método para convertir el objeto a una cadena, de modo que podamos llamar a la función interena repr() (comillas invertidas) sobre el objeto. Normalmente esta cadena tiene el aspecto de una expresión Python que podría utilizarse para recrear otro objeto con el mismo valor, o una cadena descriptiva en el caso de objetos complejos que podrían tener una representación muy complicada.

**__str__(self)**

Este método también debería devolver una representación en forma de cadena del objeto. Se le llama cuando imprimimos una instancia del objeto con print o llamamos a la función interna str() sobre el mismo. A diferencia de __repr()__ la cadena resultante no necesita ser una expresión Python válida.

Ejemplo:

```
class prueba:

   def __init__(self):
                self.a = 0
```

```
    def __inc__(self, cuanto = 1):
        self.a = self.a + cuanto
    def __str__(self):
        return str(a)
p = prueba()
p.inc()
print p.a
[devuelve 1]
print p
[devuelve 1]
```

**__cmp__(self,otro) (¡actualizar!)**

Este método es llamado por operaciones de comparación. Debe devolver un entero negativo si self < otro, cero si self == otro y un entero positivo si self > otro. Si no se ha definido __cmp__ los objetos son comparados por su identidad (su dirección de memoria).

**__hash__(self)**

Se utiliza para obtener la clave del objeto si es insertado en un diccionario, o se llama a la función interna hash() sobre él. Debe devolver un entero de 32 bits. Es importante que se cumpla que todos los objetos que se __comparan__ con igualdad devuelvan el mismo valor de __hash__. Si un objeto no define __cmp__ tampoco debería definir __hash__. Si define __cmp__ pero no __hash__, sus instancias no podrán utilizarse como claves en diccionarios. Si una clase define objetos mutables e implementa un método __cmp__ no debe implementar un __hash__ dado que las claves de los diccionarios deben de ser objetos inmutables.

**__nonzero__(self)**

Se utiliza para implementar la comprobación de certeza. Debe devolver 0 o 1 (falso o verdadero). Cuando este método no está definido, se utiliza __len__() si está definido. Si en una clase no están definidos __nonzero__ ni __len__(), todas sus instancias se tomarán como ciertas (valor 1).

## Emulando objetos invocables

**__call__(self,[args])**

Se utiliza para hacer que el objeto pueda ser llamado (como una función), de modo que si tenemos una instancia x que define __call__(self, valor) podemos hacer x(valor), lo que en realidad es un atajo a x.__call__(valor).

## Emulando tipos secuenciales y mapeables [Revisar]

**__len__(self)**

Utilizada para implementar la función interna len(). Debe devolver un entero >=0 que indique la longitud del objeto. Los métodos que no definen un método __nonzero__() y cuyo método __len__ devuelve 0 se consideran falsos en un contexto booleano (todos los demas se considerarán ciertos).

**__getitem__(self, clave)**

Se utiliza para implementar la evaluación de objeto[clave]. Para tipos secuenciales, las claves aceptadas deberían ser enteros. Nótese que la interpretación especial de índices negativos, depende del la interpretación que __getitem__ haga de los mismos.

**__setitem__(self, clave, valor)**

Se utiliza para implementar la asignación a objeto[clave]. Sólo debería utilizarse con mapeados si el objeto soporta cambios en los **valores** de las claves (no en la misma clave) o si pueden añadirse nuevas claves, y sólo

con secuencias si los elementos de estas pueden reemplazarse.

**__delitem__(self, clave)**

Se utiliza para implementar el borrado de objeto[clave]. Sólo para mapeados en los que el objeto permite borrado de claves, o para secuencias que permitan borrados de elementos.

## Métodos especiales para la emulación de tipos secuenciales

Los tipos secuenciales inmutables sólo deberían definir __getslice__. Los tipos secuenciales mutables deben definir los tres siguientes:

**__getslice__(self, i, j)**

Se utiliza para implementar la evaluación de objeto[i:j]. El objeto devuelto debe ser del mismo tipo que "objeto". Nótese que en caso de faltar alguno de los valores se sustituirá i por 0 o j por sys.maxint. La interpretación de índices negativos o índices mayores que la longitud de la secuencia es responsabilidad del método.

**__setslice__(self, i, j, secuencia)**

Se utiliza para implementar la asignación a objeto[i:j]. Se aplican las mismas notas a i y j que a __getslice__.

**__delslice__(self, i, j)**

Utilizado para implementar el borrado de objeto[i:j]. Se aplican las mismas notas a i y j que a __getslice__.

## Emulando tipos numéricos

**__add__(self, otro)**

**__sub__(self, otro)**

**__mul__(self, otro)**

**__div__(self, otro)**

**__mod__(self, otro)**

**__divmod__(self, otro)**

**__pow__(self, otro, [modulo])**

**__lshift__(self, otro)**

**__rshift__(self, otro)**

**__and__(self, otro)**

**__xor__(self, otro)**

**__or__(self, otro)**

Las funciones anteriores implementan los peradores aritméticos binarios +, -, *, /, % ,divmod , pow(), **, <<, >>, &, [comilla], |. Por ejemplo x+y llamaría al método x.__add__(y)

**__radd__(self, otro)**

**__rsub__(self, otro)**

**__rmul__(self, otro)**

**__rdiv__(self, otro)**

**__rmod__(self, otro)**

**__rdivmod__(self, otro)**

**__rpow__(self, otro)**

**__rlshift__(self, otro)**

**__rrshift__(self, otro)**

**__rand__(self, otro)**

**__rxor__(self, otro)**

**__ror__(self, otro)**

Implementan las operaciones anteriores, pero sólo si el operando de la izquierda no implementa las operaciones binarias aritméticas. Por ejemplo, si en x+y x no implementa __add__ se llamará y.__radd__(x).

**__neg__(self)**

**__pos__(self)**

**__abs__(self)**

**__invert__(self)**

Se utiliza para implementar las operaciones unarias -,+, abs(), y inversión.

**__complex__(self)**

**__int__(self)**

**__long__(self)**

**__float__(self)**

Para implementar las funciones de conversión internas complex(), int(), long() y float(). Deben devolver un valor del tipo adecuado.

**__oct__(self)**

**__hex__(self)**

Para implementar las funciones internas oct() y hex(). Deben devolver una cadena.

[Falta __coerce__]

## Personalizando los accesos a atributos (¡incluir propiedades!)

Los métodos que se detallan a continuación se utilizan para definir la forma en la que se accede a un atributo (bien por uso, asignación o borrado) x.nombre para las intancias de la clase. Por motivos de rendimiento, estos métodos se cachean en el objeto de la clase cuando esta se define y por lo tanto no pueden cambiarse una vez que la definición de la clase se ejecuta.

**__getattr__(self, nombre)**

Se le llama cuando no se encuentra un atributo por medio de los mecanismos normales de búsqueda (atributos de una instancia o en el arbol de la clase). 'nombre' es el nombre del atributo. Este método debería devolver el valor calculado del atributo o lanzar una excepción de tipo AtributeError.

Nótese que si el atributo se encuentra por medio de los mecanismos normales no se llama a __getattr__().

Por ejemplo: class prueba: def __init__(self): self.existente = 1 def __getattr__(self, nombre): print 'Creando atributo', nombre, 'y devolviendo el valor 3' self.nombre = 3 return self.nombre p = prueba() print .p.existente [Escribe "1"] p.otro_existente = 2 print p.otro_existente [Escribe "2"] print p.dinamico [Escribe "Creando atributo dinamico y devolviendo el valor 3" y despues escribe "3"] print p.dinamico [Escribe "3"]
**__setattr__(self, nombre, valor)__**

Se le llama cuando se intenta una asignación a un atributo, en lugar de proceder al mecanismo normal (almacenar el valor en el diccionario de atributos de la instancia). 'nombre' es el nombre del atributo, y no debería hacer "self.nombre = valor" porque esto ocasionaría una llamada recursiva a si mismo, sino insertar el valor en el diccionario de los atributos de las instancia (self.__dict__[nombre] = valor).

**__delattr__(self, nombre)**

Igual que __setattr__ pero para el borrado de atributos en lugar de para su asignación. Este método sólo debería implementarse si la expresión "del objeto.nombre" tiene algún significado para el objeto.