Recibe una cálida:

¡Bienvenida!

Te estábamos esperando 😁

alkemy

Clase grabada

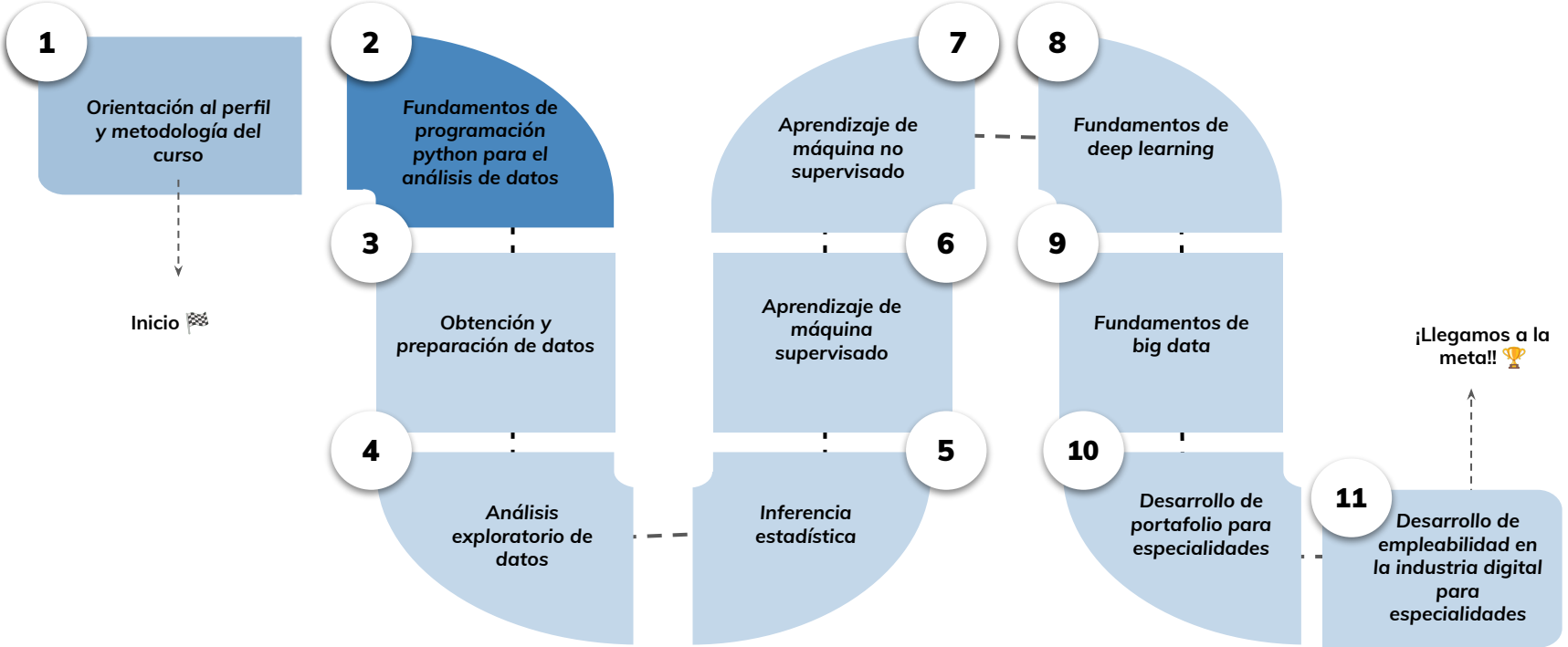
Programación orientada a objetos en python- Parte 2

Aprendizaje Esperado 7: Codificar una rutina utilizando clases provistas para la resolución de un problema simple de acuerdo al paradigma de orientación a objetos en el entorno python.

alkemy

Hoja de ruta

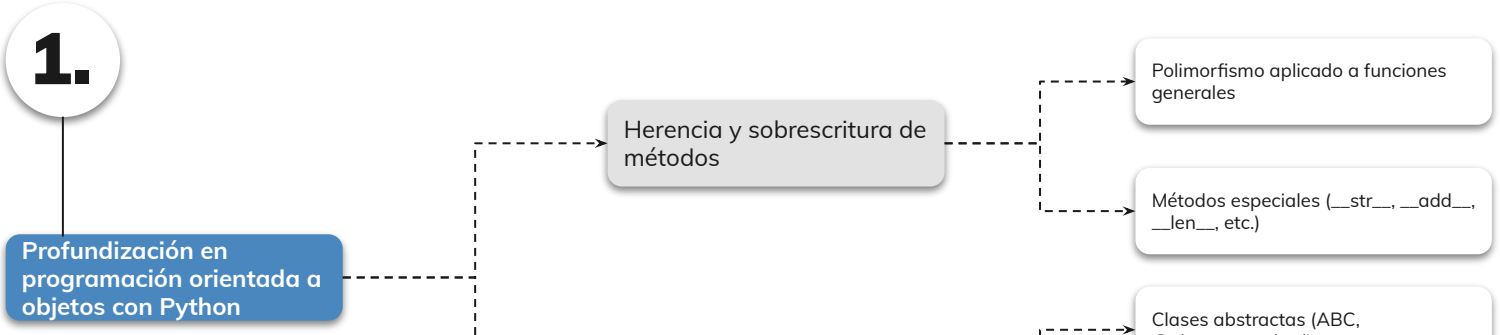
¿Cuáles skills conforman el programa? **Fundamentos de Ciencia de Datos**



alkemy

Learning Path

¿Cuáles temas trabajaremos hoy?

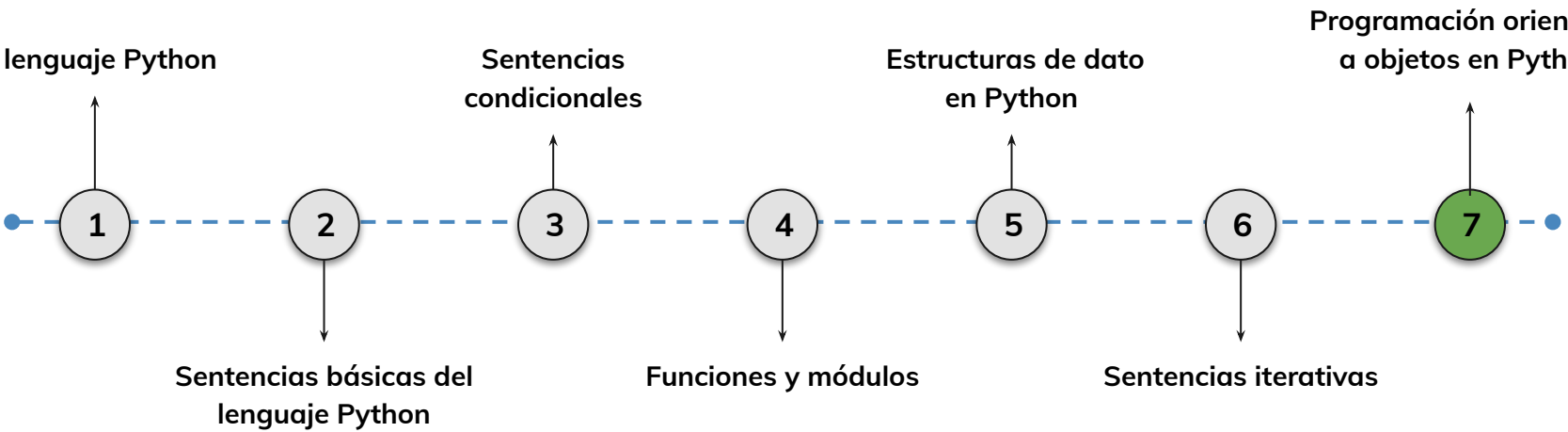


Conoceremos herramientas avanzadas del paradigma orientado a objetos en Python. Aprenderemos a aplicar herencia, polimorfismo, encapsulamiento, métodos especiales y clases abstractas, para estructurar mejor el código y reutilizar lógica en diferentes contextos.

alkemy

Roadmap de lecciones

*¿Cuáles **lecciones** estaremos estudiando en este módulo?*



alkemy

Objetivos de aprendizaje

¿Qué aprenderás?

- Aplicar herencia para extender funcionalidades de una clase base
- Implementar polimorfismo a través de métodos redefinidos
- Usar métodos especiales para personalizar el comportamiento de objetos
- Encapsular atributos y usar @property para controlar su acceso
- Comprender el uso de clases abstractas como contratos de diseño
- Codificar rutinas simples utilizando clases bien estructuradas

Repaso clase anterior

¿Quedó alguna duda?

En la clase anterior trabajamos 📚:

- → *Aprendimos los conceptos base de OOP: clase, objeto, atributo y método*
- → *Creamos nuestras propias clases con __init__() y métodos personalizados*
- → *Instanciamos objetos y manipulamos su estado con métodos públicos*
- → *Aplicamos POO para representar estudiantes o empleados de forma clara*

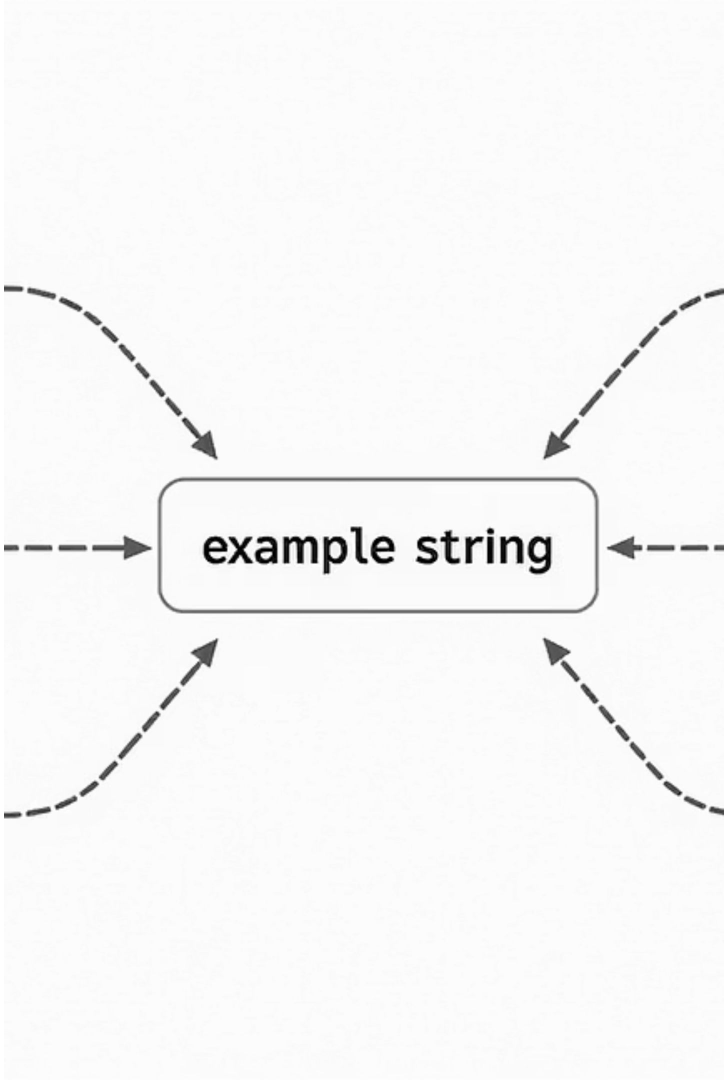
Programación orientada a objetos en python

alkemy

El Objeto String y sus Métodos Principales

En Python, las cadenas de texto o **strings** son objetos de la clase str, que ofrece una gran variedad de métodos para manipular y analizar texto. Los strings son inmutables, lo que significa que una vez creados, no se pueden modificar.

Sin embargo, sus métodos permiten realizar operaciones que devuelven nuevas cadenas basadas en la original, como cambiar de mayúsculas a minúsculas o buscar patrones específicos.



Métodos comunes de strings

upper()

Convierte todos los caracteres a mayúsculas.

lower()

Convierte todos los caracteres a minúsculas.

strip()

Elimina espacios en blanco al inicio y al final de la cadena.

replace()

Reemplaza una subcadena por otra.

split()

Divide una cadena en una lista de subcadenas según un separador.

Ejemplo de métodos de string

Cada uno de estos métodos devuelve una nueva cadena, modificar el string original. Los strings en Python tambié permiten el uso de operadores, como + para concatenar cadenas y * para repetirlas.

```
nombre = "Alice"
edad = 30
print(f"{nombre} tiene {edad} años")  # Output: "Alice tiene 30 años"
```

Interpolación de strings

Además, la interpolación de strings permite insertar valores en una cadena utilizando f-strings:

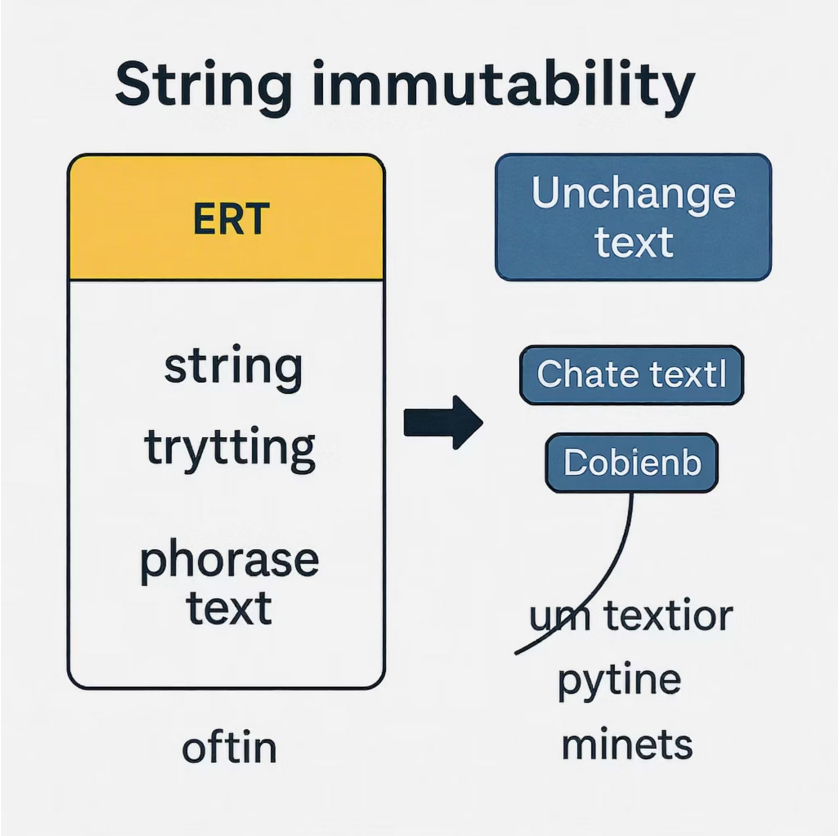
```
# Definir variables
nombre = "Ana"
edad = 25

# Usar una f-string para construir un mensaje
mensaje = f"Hola, me llamo {nombre} y tengo {edad} años."

# Imprimir el mensaje
print(mensaje)  # Imprime: Hola, me llamo Ana y tengo 25 años.
```

Esta característica hace que la creación de cadenas con valores dinámicos sea más legible y eficiente.

Inmutabilidad de strings

Es importante recordar que los strings en Python son inmutables. Esto significa que no se pueden modificar después de ser creados. Cualquier operación que parezca modificar un string en realidad está creando uno nuevo.

```
s = "hello"s = s.upper()  # Crea un nuevo stringprint(s)  #
Imprime: HELLO
```



alkemy

Métodos de búsqueda en strings

find() y index()

Buscan una subcadena dentro de otra y devuelven su posición. La diferencia es que index() lanza una excepción si no encuentra la subcadena, mientras que find() devuelve -1.

startswith() y endswith()

Verifican si una cadena comienza o termina con una subcadena específica, devolviendo True o False.

count()

Cuenta cuántas veces aparece una subcadena dentro de otra.

Métodos de validación de strings

isalpha()

Verifica si todos los caracteres de la cadena son letras.

isdigit()

Verifica si todos los caracteres de la cadena son dígitos.

isalnum()

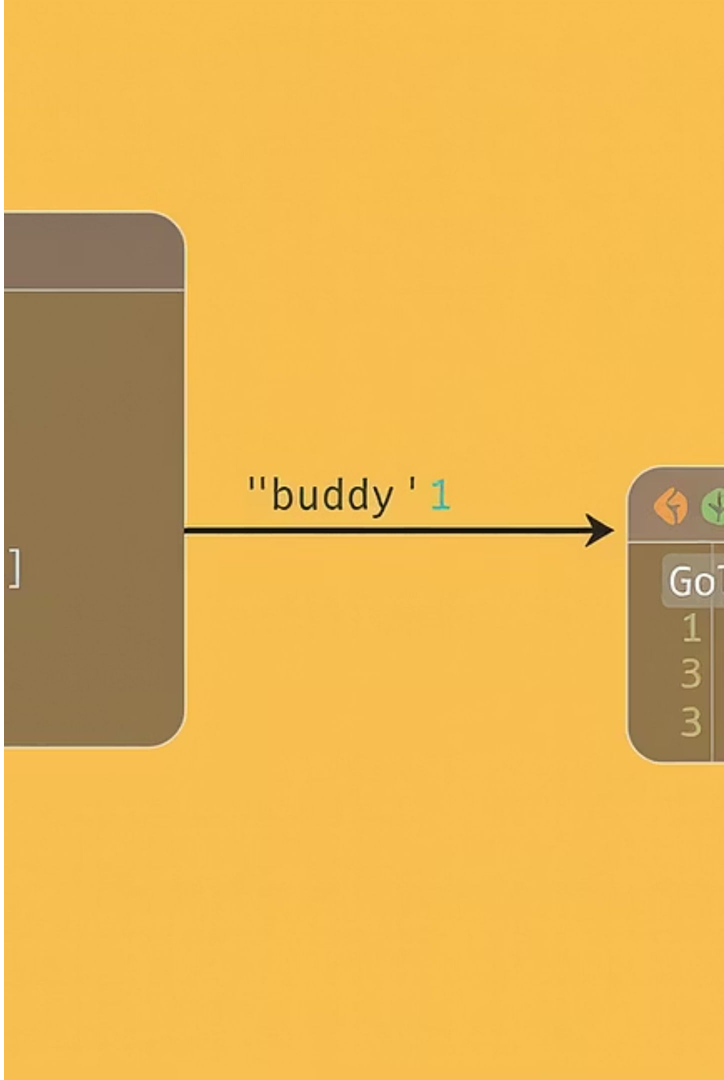
Verifica si todos los caracteres de la cadena son alfanuméricos (letras o dígitos).

islower() y isupper()

Verifican si todos los caracteres de la cadena están en minúsculas o mayúsculas, respectivamente.

Resumen de OOP en Python

La programación orientada a objetos en Python permite organizar el código en torno a clases y objetos, promoviendo una estructura modular y reutilizable. Los conceptos de clases, objetos, atributos y métodos son fundamentales para comprender y aplicar el paradigma de la OOP, y brindan las herramientas necesarias para desarrollar aplicaciones complejas de manera estructurada y eficiente.



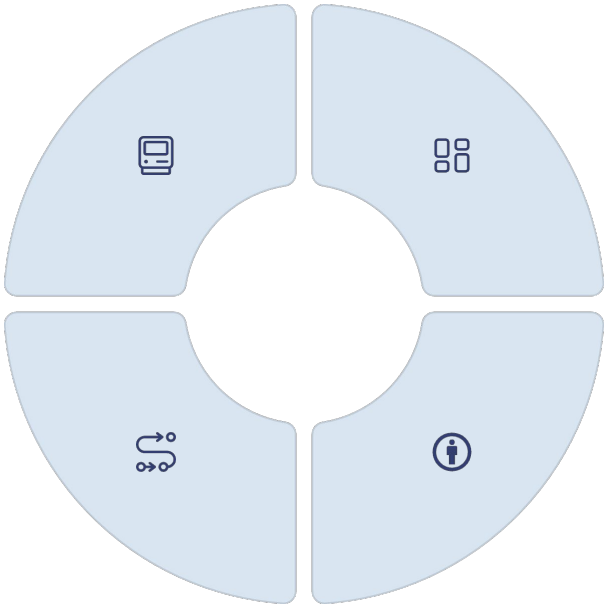
Conceptos clave de OOP en Python

Clases

Plantillas que definen propiedades y comportamientos

Métodos

Funciones que definen el comportamiento



Objetos

Instancias concretas de clases

Atributos

Propiedades que definen el estado

Principios fundamentales de OOP

Encapsulación

Ocultar detalles internos y proteger datos

Abstracción

Simplificar la complejidad mostrando solo lo esencial

Herencia

Derivar características de clases existentes

Polimorfismo

Responder de manera diferente a la misma interfaz

Beneficios de la OOP en Python

Modularidad

Divide el código en unidades independientes y reutilizables.

Mantenibilidad

Facilita la actualización y corrección de errores en el código.

Escalabilidad

Permite que el código crezca de manera organizada y estructurada.

Reutilización

Promueve la reutilización de código a través de la herencia y composición.

```
class Empleado:
    # Atributo de clase
    empresa = "TechCorp"

    # Constructor
    def __init__(self, nombre, salario):
        # Atributos de instancia
        self.nombre = nombre
        self._salario = salario  # Atributo privado

    # Método público
    def mostrar_info(self):
        return f"Empleado: {self.nombre}, Empresa:
{self.empresa}"

    # Método privado
    def _calcular_bono(self):
        return self._salario * 0.1

    # Método público que usa el método privado
    def obtener_bono(self):
        return f"Bono anual: {self._calcular_bono()}"
```

Ejemplo completo de clase en Python

alkemy

```
# Crear instancias de la clase Empleado
emp1 = Empleado("Ana López", 50000)
emp2 = Empleado("Carlos Ruiz", 60000)

# Acceder a atributos y métodos
print(emp1.nombre)              # Ana López
print(emp1.empresa)             # TechCorp
print(emp1.mostrar_info())      # Empleado: Ana López,
Empresa: TechCorp
print(emp1.obtener_bono())      # Bono anual: 5000.0

# Modificar el atributo de clase
Empleado.empresa = "NewTech"

# Ver cómo afecta a todas las instancias
print(emp1.empresa)             # NewTech
print(emp2.empresa)             # NewTech
```

Uso de la clase Empleado

```
# Definir una subclase llamada Gerente que hereda de
Empleado
class Gerente(Empleado):
    def __init__(self, nombre, salario, departamento):
        # Llamar al constructor de la clase padre
        super().__init__(nombre, salario)
        self.departamento = departamento

    # Sobrescribir un método de la clase padre
    def mostrar_info(self):
        return f"Gerente: {self.nombre}, Dept:
{self.departamento}"

    # Método específico de esta clase
    def asignar_tareas(self):
        return f"Asignando tareas en {self.departamento}"
# Crear instancia de la clase hija
gerente = Gerente("Laura Martínez", 80000, "Desarrollo")

# Usar métodos sobrescritos y heredados
print(gerente.mostrar_info())     # Gerente: Laura
Martínez, Dept: Desarrollo
print(gerente.obtener_bono())     # Bono anual: 8000.0
print(gerente.asignar_tareas())   # Asignando tareas en
Desarrollo
```

Herencia en acción

Polimorfismo en acción

```
# Clase base
class Animal:
    def sonido(self):
        pass  # Método vacío (se espera que sea
sobrescrito)

# Subclase Perro
class Perro(Animal):
    def sonido(self):
        return "¡Guau!"

# Subclase Gato
class Gato(Animal):
    def sonido(self):
        return "¡Miau!"
```

```
# Función que acepta cualquier objeto que
implemente el método sonido

def hacer_sonido(animal):

print(animal.sonido())

# Crear instancias de las subclases

perro = Perro()

gato = Gato()

# Llamar a la función con diferentes objetos

hacer_sonido(perro) # ¡Guau!

hacer_sonido(gato) # ¡Miau!
```

Métodos especiales en Python

- **__init__(self, ...)**

  Constructor que inicializa un nuevo objeto.

- **__str__(self)**

  Define la representación en string del objeto cuando se usa str() o print().

- **__repr__(self)**

  Define la representación oficial del objeto, útil para debugging.

- **__len__(self)**

  Define el comportamiento cuando se usa len() en el objeto.

- **__add__(self, other)**

  Define el comportamiento del operador + entre objetos.

Ejemplo de métodos especiales

```
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Método que define la representación en texto
    def __str__(self):
        return f"({self.x}, {self.y})"

    # Método que define la suma con otro objeto Punto
    def __add__(self, other):
        return Punto(self.x + other.x, self.y + other.y)

    # Método que define la longitud (módulo del vector)
    def __len__(self):
        return int((self.x**2 + self.y**2)**0.5)
```

```
# Crear dos objetos Punto

p1 = Punto(3, 4)

p2 = Punto(2, 3)

# Pruebas

print(p1) # (3, 4)

print(p1 + p2) # (5, 7)

print(len(p1)) # 5
```

Propiedades en Python

Las propiedades permiten definir métodos especiales que se comportan como atributos, proporcionando control sobre el acceso a los atributos de un objeto.

```
class Persona:
    def __init__(self, nombre):
        self._nombre = nombre  # Atributo privado

    # Getter: permite acceder al atributo como si fuera público
    @property
    def nombre(self):
        return self._nombre

    # Setter: define cómo modificar el atributo, con validación
    @nombre.setter
    def nombre(self, valor):
        if not isinstance(valor, str):
            raise TypeError("El nombre debe ser una cadena")
        self._nombre = valor
```

```
p = Persona("Juan")
print(p.nombre)       # Juan

p.nombre = "Ana"      # Usa el setter
print(p.nombre)       # Ana

p.nombre = 123        # Lanzaría TypeError
```

Clases abstractas en Python

Las clases abstractas definen interfaces que las clases hijas deben implementar. Se utilizan para establecer un contrato que las subclases deben cumplir.

```
from abc import ABC, abstractmethod
# Clase abstracta
class FormaGeometrica(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimetro(self):
        pass

# Subclase que implementa todos los métodos
abstractos
class Rectangulo(FormaGeometrica):
    def __init__(self, ancho, alto):
        self.ancho = ancho
        self.alto = alto
```

```
def area(self):

return self.ancho * self.alto

def perimetro(self):

return 2 * (self.ancho + self.alto)

# No se puede instanciar una clase abstracta directamente

# forma = FormaGeometrica() # ❌ Esto lanzaría un error

# Crear una instancia válida de una subclase concreta

rect = Rectangulo(5, 3)

print(rect.area()) # 15

print(rect.perimetro()) # 16
```

Composición vs. Herencia

Herencia

Establece una relación "es un" entre clases. Por ejemplo, un Gerente es un Empleado.

```
class Empleado:    # ...class Gerente(Empleado):
# ...
```

Composición

Establece una relación "tiene un" entre clases. Por ejemplo, un Coche tiene un Motor.

```
class Motor: # ...class Coche: def __init__(self):
self.motor = Motor() # ...
```

Buenas prácticas en OOP con Python

Nombres descriptivos

Usar nombres claros y descriptivos para clases, métodos y atributos.

Responsabilidad única

Cada clase debe tener una única responsabilidad y razón para cambiar.

Encapsulación adecuada

Usar convenciones de privacidad (_atributo) para proteger el estado interno.

Herencia con cuidado

Preferir composición sobre herencia cuando sea posible.

Documentación

Documentar clases y métodos con docstrings para facilitar su uso.

Ejemplo de clase bien documentada

```
class CuentaBancaria:
    def __init__(self, titular, saldo=0):
        self.titular = titular
        self.saldo = saldo

    def depositar(self, monto):
        if monto > 0:
            self.saldo += monto

    def retirar(self, monto):
        if monto <= self.saldo:
            self.saldo -= monto

    def consultar_saldo(self):
        return self.saldo
```

Implementación de atributos protegidos y validación en métodos

```
class CuentaBancaria:
    def __init__(self, titular, saldo_inicial=0):
        """
        Inicializa una nueva cuenta bancaria.

        Args:
            titular (str): Nombre del titular de la cuenta.
            saldo_inicial (float, optional): Saldo inicial. Por
defecto 0.
        """
        self.titular = titular
        self._saldo = saldo_inicial
        self._numero = self._generar_numero()
```

```
    def depositar(self, cantidad):
        """
        Añade dinero a la cuenta.

        Args:
            cantidad (float): Cantidad a depositar.

        Returns:
            float: Nuevo saldo de la cuenta.

        Raises:
            ValueError: Si la cantidad es negativa.
        """
        if cantidad < 0:
            raise ValueError("No se puede depositar una cantidad
negativa")
        self._saldo += cantidad
        return self._saldo
```

alkemy

< >

Conclusión

Este manual ha presentado los conceptos clave de la OOP en Python, explorando la abstracción, el ocultamiento, la creación de clases y objetos, así como el uso de constructores y métodos. Al aplicar estos principios, los desarrolladores pueden crear aplicaciones más organizadas, extensibles y fáciles de mantener, aprovechando la capacidad de Python para implementar el paradigma de la OOP de forma accesible y potente.

La programación orientada a objetos no solo mejora la estructura y claridad del código, sino que también fomenta la reutilización y la colaboración en proyectos de software. Al seguir aprendiendo y practicando estos conceptos, los desarrolladores podrán construir sistemas robustos y modulares que faciliten la resolución de problemas complejos.

Live Coding

¿En qué consistirá la Demo?

Vamos a modelar un sistema de empleados con jerarquías. Crearemos clases con herencia, métodos sobrescritos y aplicaremos encapsulamiento, propiedades y polimorfismo.

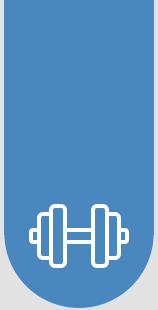
1. *Crear clase base Empleado con atributos y método mostrar_info()*
2. *Crear clase hija Gerente que hereda de Empleado y sobrescribe métodos*
3. *Implementar método obtener_bono() y control de acceso con _atributo*
4. *Añadir @property y @setter para controlar el atributo nombre*
5. *Aplicar __str__() para impresión amigable*
6. *Crear clase Desarrollador con método exclusivo*
7. *Usar una función mostrar_datos(objeto) para demostrar polimorfismo*
8. *Mostrar cómo crear una clase abstracta con métodos obligatorios*

Tiempo: 25 Minutos

Momento:

Time-out!

⏰5 -10 min.



Ejercicio N° 1

Sistema de empleados con herencia y polimorfismo

Sistema de empleados con herencia y polimorfismo

Contexto: 🙌

Una empresa necesita gestionar distintos tipos de empleados, como administrativos, gerentes y técnicos. Desea implementar un sistema que permita registrar a cada uno con sus datos básicos, calcular sus bonos, y mostrar la información de forma personalizada según el tipo de empleado.

Consigna: ✍️

Codificar una rutina utilizando clases provistas para la resolución de un problema simple de acuerdo al paradigma de orientación a objetos en el entorno Python.

Tiempo 🕛**:** *35 Minutos*

Sistema de empleados con herencia y polimorfismo

Paso a paso: ⚙️

1. Crear una clase base Empleado con:

- nombre (str), salario (float)
- Método mostrar_info() y obtener_bono() (10% del salario)

2. Crear clases hijas:

- Gerente: sobrescribe obtener_bono() con 20%
- Técnico: agrega atributo especialidad y redefine mostrar_info()

3. Aplicar polimorfismo con una función ver_info(empleado) que reciba cualquier objeto y muestre su información.
4. Agregar a la clase base una propiedad nombre con getter y setter que valide que sea string.
5. Instanciar objetos de cada clase, almacenarlos en una lista y recorrerla mostrando los datos.

¿Alguna consulta?

alkemy

Resumen

¿Qué logramos en esta clase?

- ✓ ***Aplicamos herencia para reutilizar código entre clases relacionadas***
- ✓ ***Usamos polimorfismo para ejecutar métodos personalizados desde una misma interfaz***
- ✓ ***Implementamos métodos especiales para personalizar objetos***
- ✓ ***Encapsulamos atributos y controlamos su acceso con @property***
- ✓ ***Creamos clases abstractas para definir estructuras obligatorias***
- ✓ ***Diseñamos clases modulares, reutilizables y mantenibles***

¡Ponte a prueba!

Momento de ejercitación

Te invitamos a aprovechar esta última sección del espacio sincrónico para realizar de manera individual las ***actividades disponibles en la plataforma.*** *Estas propuestas son claves para afianzar lo trabajado y* ***forman parte obligatoria del recorrido de aprendizaje.***

💪 Análisis de caso ——— 💪 Selección Múltiple

💪 Comprensión lectora

Si al resolverlas surge alguna duda, compartela o tráela al próximo encuentro sincrónico.

alkemy

¡Muchas gracias!

alkemy