Una Ambiciosa Introducción a Python
Parte II

```
class Profesor:
    def __init__(self, nombre, materia):
        self.nombre = nombre
        self.__materia = materia  # atributo privado

    def get_materia(self):
        return self.__materia

    def set_materia(self, nueva_materia):
        self.__materia = nueva_materia


    def __str__(self):
        return f"Profesor: {self.nombre}, Materia: {self.__materia}"
```

Class nos permite definir una clase, luego de esta palabra reservada viene el nombre de la clase escrita en CamelCase seguido de los **:** que definen el cuerpo de la clase

Este método,dunder, __init__ . Nos permite inicializar los atributos de los objetos, asignarles valores iniciales. Solo podemos definir un inicializador por clase.

Importante
. Es ejecutado de manera automática.
. No devuelve ningún valor.
. Siempre toma al menos un valor self.
. Puede tener otros parámetros.

. Garantiza un esta inicial consistente.
. Facilita la creación de objetos.
. Separa la creación del objetos de su posterior uso.

self: Este parámetro es una convención que utilizan los métodos de una clase para hacer referencia al objeto actual.
Permite a Python hacer que los métodos y atributos dentro de las clases sean dinámicos y estén ligados a las instancias individuales

```
class Persona:
    def __init__(self, nombre, edad, apellido):
        self.nombre = nombre        #Atributo público
        self._edad = edad           #Atributo protegido
        self.__apellido=apellido    #Atributo privado
    def saludar(self):
        return f"Hola, mi nombre es {self.nombre} y tengo {self._edad} años."

    def _actualizar_edad(self, nueva_edad):
        self._edad = nueva_edad

    def __str__(self):
        return f"Persona: {self.nombre}, Edad: {self._edad}"

    def __repr__(self):
        return f"Persona(nombre={self.nombre!r}, edad={self._edad!r})"
```

Los atributos y métodos pueden tener diferentes niveles de acceso.
En python es solo una convención.

Métodos “dunder”

__str__: nos da una representación amigable de un objeto al imprimirlo.

__repr__: Esta representación mas detallada, utilizada principalmente
para depuración

Lic. Franco Herrera

Una Ambiciosa Introducción a Python
Parte II

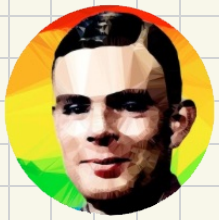
```
class Cd:
    def __init__(self, author,title,year,price,offer='no'):
        self.__author = author
        self.__title = title
        self.__year = year
        self.__price = price
        self.__offer = offer

    def __str__(self):
        return f"""
    - Autor: {self.__author}
    - Título: {self.__title}
    - Año de publicación: {self.__year}
    - Precio: {self.__price}
    - En oferta: {"si" if self.__offer else "no"}
    """

    def get_title(self):
        return self.__title
```

Creación de instancias, objetos, de la clase Cd

```
cd1 = CD("Bad Bunny", "Un Verano Sin Ti", 2022, "Reggaeton", 19.99, 'si')

cd2 = CD("Taylor Swift", "Midnights", 2022, "Pop", 15.99)

cd3 = CD("Kendrick Lamar", "Mr. Morale & the Big Steppers", 2022, "Hip Hop", 24.99, 'no')

#invocamos al método especially __str__

print(cd2)

#invocamos a un método

print(cd3.get_title())

#accedemos a un atributo que por convención establecimos que es privado.

cd1._Cd__title. #no recomendado ya que definimos que es privado
```

Lic. Franco Herrera

Una Ambiciosa Introducción a Python
Parte II



Atributos de instancias: son variables que pertenecen a un objeto particular. se definen en __init__

```python
class Cd:

    def __init__(self, author,title,year,price,offer):
        self.__author = author
        self.__title = title
        self.__year = year
        self.__price = price
        self.__offer = offer


    def __str__(self):
        return f"""
    - Autor: {self.__author}
    - Título: {self.__title}
    - Año de publicación: {self.__year}
    - Precio: {self.__price}
    - En oferta: {"si" if self.__offer else "no"}
    """

    @property
    def title(self):
        return self.__title

    @title.setter
    def title(self, title):
        if isinstance(title,str) and title:
            self.__title=title
        else:
            print("El titulo debe ser una cadena no vacia")
```

El decorador @property convierte un método en un atributo de solo lectura, es decir, permite acceder a un atributo como si fuera una variable, pero detrás de escena se ejecuta un método. Este método puede realizar operaciones adicionales, como validaciones o cálculos.

El decorador @nombre.setter (donde nombre es el nombre del método original) permite definir un método que se ejecuta cuando se intenta modificar el atributo. Este método se encarga de validar o modificar el valor antes de asignarlo.

```
>>> print(cd)

    - Autor: Franco
    - Título: mi album
    - Año de publicación: 2023
    - Precio: 12.45
    - En oferta: si

>>> cd.title=''
El titulo debe ser una cadena no vacia
>>> print(cd.title)
mi album
```

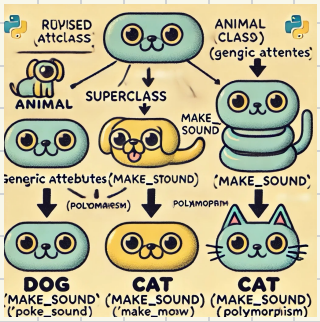
Los decoradores @property y @setter son una característica en Python que permiten definir métodos que se comportan como si fueran atributos, facilitando el acceso y modificación de los atributos de una clase de manera controlada. Esto es útil para encapsular atributos privados y aplicar validaciones o realizar acciones adicionales cuando se acceden o modifican.

```python
class BusinessType(Enum):
    GASTRONOMICO = 1
    MINORISTA = 2
    SERVICIO = 3
    ENTRETENIMIENTO = 4
    SALUD = 5
    TECNOLOGICO = 6
    ESPECIALIZADO = 7
    EDUCATIVO = 8
```

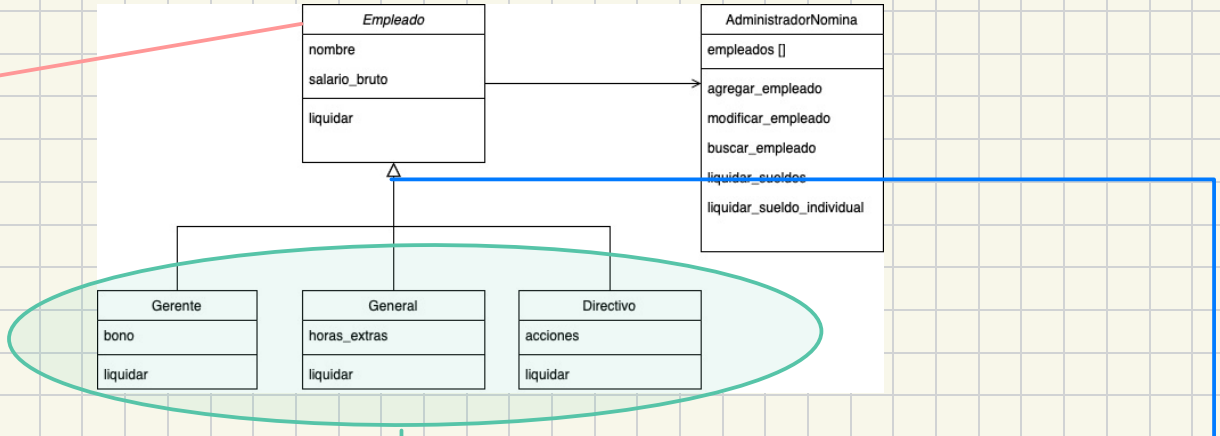
Atributos de clase: Son compartidos por todas las instancias de una clase. Se definen fuera de cualquier método. No es necesario tener una instancia para acceder a ellos.

Lic. Franco Herrera

Una Ambiciosa Introducción a Python
Parte II



La herencia es uno de los pilares fundamentales de la Programación Orientada a Objetos. Nos permite crear nuevas clases (llamadas clase hijas o subclases) a partir de clases existentes (super clases o clases padres) heredando atributos y métodos. Esto facilita la reutilización de código y permite crear relaciones jerárquicas entre clases.

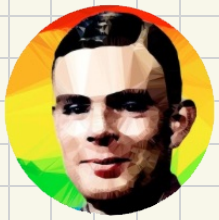


Conceptos claves:

1. Clase Padre (Superclase): Es la clase de la cual otra clase hereda. Proporciona atributos y comportamientos comunes que las subclases pueden utilizar.
2. Clase Hija (Subclase): Es la clase que hereda de una clase padre. Además de heredar los atributos y métodos de la clase padre, puede definir sus propios atributos y métodos, o sobrescribir (modificar) los métodos heredados.
3. Herencia Simple: Una clase hija hereda de una única clase padre.
4. Herencia Múltiple: Una clase hija puede heredar de más de una clase padre en Python. Esto se logra listando varias clases padres en la definición de la subclase.

Lic. Franco Herrera

Una Ambiciosa Introducción a Python
Parte II



Una clase abstracta es una plantilla para crear clases, pero a diferencia de las clases regulares, no puede ser instanciada directamente. Sirve como una especie de contrato, definiendo los métodos que deben ser implementados por las clases que heredan de ella. Esto asegura una coherencia y un diseño común en una jerarquía de clases.

Características claves de las clases abstractas:

- Métodos abstractos: Son métodos declarados sin implementación. Su propósito es obligar a las subclases a proporcionar su propia implementación.
- No se pueden instanciar: No puedes crear objetos directamente a partir de una clase abstracta.
- Fomenta la polimorfismo: Al definir una interfaz común, las clases abstractas permiten que objetos de diferentes clases sean tratados de manera uniforme.

Para crear clases abstractas en Python, se utiliza el módulo abc (Abstract Base Classes).
Este módulo proporciona la clase base ABC y el decorador abstractmethod

```python
from abc import ABC, abstractmethod

class Empleado(ABC):

  def __init__(self, nombre, salario_bruto):
    self.__nombre=nombre
    self.__salario_bruto=salario_bruto


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

  @nombre.setter
  def nombre(self, nombre):
    self.__nombre = nombre


  @property
  def salario_bruto(self):
    return self.__salario_bruto


  @salario_bruto.setter
  def salario_bruto(self, salario_bruto):
    self.__salario_bruto = salario_bruto


  @abstractmethod
  def liquidar(self):
    pass


  def __str__(self):
    return f"{self.__nombre} posee un salario
bruto de: {self.__salario_bruto} "
```

En la definición de la clase entre paréntesis especificamos de clase/s hereda.

El método liquidar se declara como abstracto utilizando el decorador @abstractmethod
Las clases que hereden de Empleado deberán todos los métodos abstractos declarados en la clase base.

Lic. Franco Herrera

Una Ambiciosa Introducción a Python
Parte II

Heredamos, extendemos, de la clase Empleado

```python
from empleado import Empleado
class Gerente(Empleado):

  def __init__(self, nombre, salario_bruto, bono):
    super().__init__(nombre, salario_bruto)
    self.__bono=bono


  @property
  def bono(self):
    return self.__bono


  @bono.setter
  def bono(self, bono):
    self.__bono = bono


  def liquidar(self):
      # El gerente paga 30% de ganancias más 15% de aportes
      salario_bruto_con_bono = self.salario_bruto + self.bono
      descuento_ganancias = salario_bruto_con_bono * 0.30
      descuento_aportes = salario_bruto_con_bono * 0.15
      salario_neto = salario_bruto_con_bono - descuento_ganancias - descuento_aportes
      return f"Gerente: {self.nombre}, Salario neto: {salario_neto:.2f}, Bono: {self.bono}"


  def __str__(self):
     return f"Gerente: {super().__str__()} "
```

invocamos al inicializador de la superclase pasando sus atributos usando la función super() que ademas nos permitirá invocar otros métodos de la clase padre.

'Definimos propiedades, atributos, particulares de la clase.

Damos implementación al método abstracto declarado en la clase padre

Beneficios de la herencia:

Reutilización de código: No es necesario repetir atributos o métodos comunes en varias clases.

Extensibilidad: Se pueden crear clases más específicas que amplíen el comportamiento de clases más generales.

Organización: Facilita estructurar el código de manera jerárquica y lógica.

Lic. Franco Herrera