Una Ambiciosa Introducción a Python
Parte II

Una empresa necesita desarrollar una aplicación que permita gestionar la liquidación de sueldos de sus empleados. La empresa tiene diferentes tipos de empleados, cada uno con características y reglas de liquidación particulares. La aplicación debe cumplir con los siguientes requisitos:

Reglas Generales:

- Todos los empleados tienen un **nombre** y un **salario bruto** que define el valor total antes de cualquier deducción.
- Todos los empleados deben poder liquidar su salario, lo que implica calcular el salario neto después de las deducciones correspondientes.

Tipos de Empleados:

1. **Empleado General**:
   - Además de su salario bruto, un empleado general puede tener **horas extra**.
   - Cada hora extra se paga a un valor fijo.
   - A estos empleados se les deduce el 15% del salario bruto (más las horas extra) en concepto de aportes.
2. **Gerente**:
   - Los gerentes, además de su salario bruto, pueden recibir un **bono**.
   - A los gerentes se les deduce el 30% de su salario bruto (más el bono) en concepto de ganancias, y otro 15% en concepto de aportes.
3. **Directivo**:
   - Los directivos pueden tener un número de **acciones** de la empresa, cuyo valor por acción es variable.
   - A los directivos se les deduce el 40% del total de su salario bruto (más el valor de las acciones) en concepto de ganancias, y otro 20% en concepto de aportes.

Funcionalidades de la Aplicación:

1. La aplicación debe permitir agregar empleados de diferentes tipos (Empleado General, Gerente, Directivo) a un sistema de nóminas.
2. Cada empleado debe poder **liquidar su sueldo** de acuerdo a las reglas mencionadas.
3. La aplicación debe permitir **modificar** los datos de los empleados, como salario, horas extra, bono o acciones, dependiendo del tipo de empleado.
4. El sistema debe poder realizar la **liquidación de sueldos** de todos los empleados de manera general, y también permitir la liquidación del sueldo de un empleado específico.

Implementación:

- Crea una clase abstracta Empleado que defina los atributos y el comportamiento general de los empleados.
- Define las subclases EmpleadoGeneral, Gerente y Directivo que implementen los detalles específicos de cada tipo de empleado.
- Utiliza polimorfismo para que cada clase implemente el método liquidar() de manera diferente según las reglas de deducción.
- Crea una clase gestora llamada AdministradorNominas que permita agregar, modificar y liquidar los sueldos de los

empleados.

Desafío:

- Debes asegurarte de que los atributos de los empleados sean privados y gestionados mediante **propiedades** utilizando los decoradores @property y @setter.
- Utiliza la clase abstracta Empleado para garantizar que el comportamiento común sea definido y heredado, pero que las subclases puedan modificar el comportamiento según las reglas particulares.

Ejemplo:

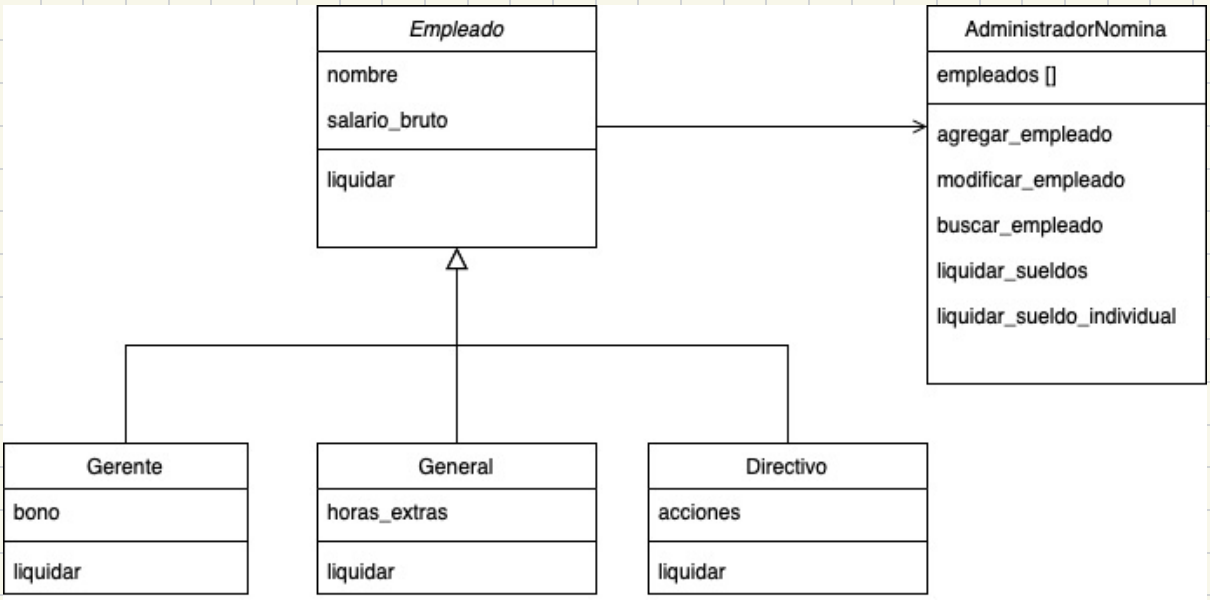
Un **Empleado General** con un salario bruto de $100,000 y 10 horas extra, con cada hora extra valorada en $1,000, debe tener una deducción del 15% sobre el total de su salario (salario bruto + horas extra).

Un **Gerente** con un salario bruto de $200,000 y un bono de $50,000, debe tener una deducción del 30% en concepto de ganancias y del 15% en concepto de aportes, ambos aplicados sobre el total del salario (salario bruto + bono).

Un **Directivo** con un salario bruto de $300,000, 100 acciones y un valor de acción de $200, debe tener una deducción del 40% en concepto de ganancias y del 20% en concepto de aportes sobre el total de su salario (salario bruto + valor de las acciones).

Tu tarea será modelar esta situación utilizando **clases y objetos**, implementando las reglas descritas y el sistema de nóminas para la empresa.

Una Ambiciosa Introducción a Python
Parte II



Lic. Franco Herrera

Una Ambiciosa Introducción a Python
Parte II

El polimorfismo es otro de los conceptos fundamentales de la Programación Orientada a Objetos (POO), que permite que diferentes objetos respondan de manera distinta a un mismo mensaje o método. En otras palabras, el polimorfismo permite que una misma operación o método pueda comportarse de manera diferente dependiendo del tipo de objeto con el que se esté trabajando.

Este método pertenece a una clase gestora/manager que administra los diferentes empleados de una empresa.

```
def liquidar_sueldos(self):
        for empleado in self.__empleados:
            print(empleado.liquidar())
```

El método liquidar que se ejecutara dependerá de "duck typing", no se verificara los tipos de los objetos en tiempo de compilación como en otros lenguajes, Python verificara si un objeto tiene los métodos o atributos necesarios en tiempo de ejecución. Si es así, el objeto puede ser utilizado como si fuera de ese tipo.
Justamente a eso hace referencia "duck typing" que nos dice "si camina como un pato y ganza como un pato, entonces es un pato"

Ventajas del polimorfismo en Python:

- Flexibilidad: Permite escribir código más genérico y reutilizable.
- Facilidad de uso: El duck typing hace que el código sea más conciso y legible.
- Extensibilidad: Es fácil agregar nuevas clases y métodos sin modificar el código existente.

Lic. Franco Herrera

Una Ambiciosa Introducción a Python
Parte II

Estos son algunos de los métodos y funciones más importantes relacionados con la herencia en Python

```
def modificar_empleado(self, nombre, nuevo_salario=None,**kwargs):
        empleado = self.buscar_empleado(nombre)
        if empleado:
            if nuevo_salario is not None:
                empleado.salario_bruto = nuevo_salario
                print(f"Salario de {empleado.nombre} actualizado a {nuevo_salario}.")
            if isinstance(empleado, General) and "horas_extra" in kwargs:
                empleado.horas_extra = kwargs['horas_extra']
                print(f"Horas extra de {empleado.nombre} actualizadas a {kwargs['horas_extra']}.")
            elif isinstance(empleado, Gerente) and "bono" in kwargs:
                empleado.bono = kwargs['bono']
                print(f"Bono de {empleado.nombre} actualizado a {kwargs['bono']}.")
            elif isinstance(empleado, Directivo):
                if "acciones" in kwargs:
                    empleado.acciones = kwargs['acciones']
                    print(f"Acciones de {empleado.nombre} actualizadas a {kwargs['acciones']}.")
                if "valor_accion" in kwargs:
                    empleado.valor_accion = kwargs['valor_accion']
                    print(f"Valor de acción de {empleado.nombre} actualizado a
{kwargs['valor_accion']}.")
        else:
            print(f"Empleado con nombre {nombre} no encontrado.")
```

isinstance(): Verifica si un objeto es instancia de una clase o subclase.

```
>>> from gerente import Gerente
>>> issubclass(Gerente, Empleado)
True
```

issubclass(): Verifica si una clase es subclase de otra.

```
>>> Empleado.__bases__
(<class 'abc.ABC'>,)
>>> dir(Gerente)
['__abstractmethods__', '__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getstate__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__slots__', '__str__', '__subclasshook__', '__weakref__', '_abc_impl', 'bono', 'l
iquidar', 'nombre', 'salario_bruto']
```

__bases__: Muestra las clases base de una clase.

dir(): Lista los atributos y métodos disponibles en un objeto.

```
>>> type(franco)
<class 'gerente.Gerente'>
>>> franco.__class__
<class 'gerente.Gerente'>
>>> getattr(franco,'bono')
230
>>> setattr(franco,'bono',235)
>>> hasattr(franco,'nombre')
True
```

type() y __class__: Muestran la clase a la que pertenece un objeto.

getattr(), setattr(), hasattr(): Permiten trabajar con los atributos de un objeto de manera dinámica.

Lic. Franco Herrera