HINTS

DIAGRAMA DE CLASES

Es un tipo de diagrama estático que describe la estructura de un sistema ilustrando sus clases, orientados a objetos.

DUCK TYPING EN PYTHON

El *duck typing*, o tipado de pato, es un concepto relacionado con la programación que aplica a ciertos lenguajes orientados a objetos, y que tiene origen en la siguiente frase:

"If it walks like a duck and it quacks like a duck, then it must be a duck"

Lo que se podría traducir al español como "Si camina como un pato y habla como un pato, entonces tiene que ser un pato".

¿Y qué relación tienen los patos con la programación? Pues bien, se trata de un símil en el que los patos son objetos y hablar/andar métodos. Es decir, que si un determinado objeto tiene los métodos que nos interesan, nos basta, siendo su tipo irrelevante.

Dicho de otra manera, no mires si es un pato. Fíjate si habla como un pato, camina como un pato, entre otros. Si cumple con todas estas características, ¿no podríamos acaso decir que se trata de un pato?

El concepto de *duck typing* se fundamenta en el razonamiento inductivo, donde una serie de premisas apoyan la conclusión, pero no la garantizan. Si vemos a un animal que parece un pato, habla como tal y anda como tal, sería razonable pensar que se trata de un pato, pero sin un test de ADN nunca estaríamos al cien por ciento seguros.

Una vez entendido el origen del concepto, veamos lo que realmente significa esto en Python. En pocas palabras, a Python le dan igual los tipos de objetos, lo único que le importan son los métodos.

Definamos una clase Pato con un método `hablar()`:

```
class Pato:
    def hablar(self):
        print("¡Cua!, Cua!")
```

Y llamamos al método de la siguiente forma:

```
p = Pato()
p.hablar()
# ¡Cua!, Cua!
```

Hasta aquí nada nuevo, pero vamos a definir una función `llama_hablar()`, que llama al método `hablar()` del objeto que se le pase.

```
def llama_hablar(x):
    x.hablar()
```

Como se puede observar, en Python no es necesario especificar los tipos, simplemente indicamos que el parámetro de entrada tiene el nombre x, pero no especificamos su tipo.

Cuando Python entra en la función y evalúa `x.hablar()`, le da igual el tipo al que pertenezca x, siempre y cuando tenga el método `hablar()`. Esto es el *duck typing* en todo su esplendor.

```
p = Pato()
llama_hablar(p)
# ¡Cua!, Cua!
```

¿Y qué pasa si usamos otros objetos que no son de la clase Pato? Pues bien, como hemos dicho, a la función `llama_hablar()` le da igual el tipo. Lo único que le importa es que el objeto tenga el método `hablar()`.

Definamos tres clases de animales distintos que implementan el método `hablar()`. Nótese que no existe herencia entre ellas, son clases totalmente independientes. De haberla estaríamos hablando de polimorfismo.

```
class Perro:
    def hablar(self):
        print("¡Guau, Guau!")

class Gato:
    def hablar(self):
        print("¡Miau, Miau!")

class Vaca:
    def hablar(self):
        print("¡Muuu, Muuu!")
```

Y como es de esperar, la función `llama_hablar()` funciona correctamente con todos los objetos.

```
llama_hablar(Perro())
llama_hablar(Gato())
llama_hablar(Vaca())
```

Salida:

```
# ¡Guau, Guau!
# ¡Miau, Miau!
# ¡Muuu, Muuu!
```

Otra forma de verlo es iterando una lista con diferentes animales, donde animal va tomando los valores de cada objeto animal. Todo un ejemplo del *duck typing* y del tipado dinámico en Python.

```
lista = [Perro(), Gato(), Vaca()]
for animal in lista:
    animal.hablar()
```

Salida:

```
# ¡Guau, Guau!
# ¡Miau, Miau!
# ¡Muuu, Muuu!
```

DUCK TYPING CON LEN()

Podemos ver el *duck typing* en todo su esplendor con la función `len()`. Esta lo único que realiza por debajo es llamar al método mágico `__len__()`. Definamos dos clases:

`Foo` implementa el método `__len__()`.

`Bar` no lo implementa.

```
class Foo():
    def __len__(self):
        return 99

class Bar():
    pass
```

Como ya hemos explicado, a la función `len()` no le importa el tipo del objeto que se le pase, siempre y cuando tenga el método `__len__()` implementado. Por ello, en el segundo caso falla.

```
print(len(Foo())) # 99
print(len(Bar())) # Error
```

DUCK TYPING CON MULTIPLICAR

Por otro lado, cuando hacemos una multiplicación utilizando el operador aritmético *****, el resultado depende de los tipos que estemos usando.

No es lo mismo multiplicar dos enteros que un entero y cadena.

```
print(3*3)    # 9
print(3*"3")  # 333
```

Una vez más, podemos ver el *duck typing* en Python. Simplemente se busca que los objetos a la izquierda y derecha del * tengan implementado el `__rmul__` o `__mul__`.

Finalmente, Python es un lenguaje que soporta el *duck typing*, lo que hace que el tipo de los objetos no sea tan relevante, siendo más importante lo que pueden hacer (sus métodos). Otros lenguajes como Java no soportan el *duck typing*, pero se puede conseguir un comportamiento similar cuando los objetos comparten un interfaz (si existe herencia entre ellos). Este concepto relacionado es el polimorfismo.