Programación 1

Tecnicatura Universitaria en Inteligencia Artificial

2022

Apunte sobre Programación Orientada a Objetos

1. Paradigmas de programación

Los paradigmas de programación son modelos para el desarrollo estructurado de algoritmos. Distintos lenguajes de programación adhieren a uno u otro modelo.

1.1. Paradigma imperativo

Un lenguaje imperativo permite escribir programas como un conjunto de instrucciones que se ejecutan una por una, de principio a fin, de modo secuencial excepto cuando intervienen instrucciones de salto de secuencia o control. En los lenguajes imperativos, el concepto central para la ejecución de un programa es el concepto de *estado*, que va modificándose a medida que asignamos y cambiamos valores a *variables*.

Algunos ejemplos de lenguajes que adhieren a este modelo son:

- El tradicional lenguaje **C**, diseñado por Kernighan y Ritchie en el año 1969 y usado ampliamente en la implementación de sistemas operativos.
- **FORTRAN** un lenguaje de programación de propósito general especialmente adaptado al cálculo numérico y a la computación científica.

Este es el paradigma en el que venimos trabajando hasta ahora.

1.2. Paradigma Orientado a Objetos

Este paradigma evolucionó de los lenguajes imperativos. El enfoque orientado a objetos guarda analogía con la vida real. El desarrollo de software Orientado a Objetos se basa en el diseño y construcción de objetos que contienen tanto información especifica del objeto como procedimientos para trabajar con ellos, modularizando el desarrollo de programas. La idea central de este paradigma fue primero distinguir en el paradigma imperativo dos tipos de instrucciones muy distintas, por un lado la asignación de valores a variables y por otro lado la definición de funciones. Luego, se trató de forzar un diseño de programas donde la asignación de variables y la definición de funciones relacionadas semánticamente con estas variables estuvieran también relacionadas sintácticamente. Es decir, se intenta lograr que las funciones que están fuertemente relacionadas con un conjunto de variables aparezcan "cerca" de estas en el código. El concepto central sigue siendo el de *estado*, pero la diferencia es que ahora, además del estado global del programa, cada objeto tiene un estado propio.

2. Python como lenguaje multiparadigma

El lenguaje de programación Python es ampliamente usado y su popularidad solo va en ascenso, desplazando incluso a lenguajes tradicionales de algunas áreas de dominio especifico.

- En el área de cálculo numérico y computación científica, ha comenzado a desplazar al tradicional FORTRAN.
- En el área de desarrollo de aplicaciones para la web, comienza a ser cada vez más utilizado, junto con lenguajes como Java, Node o PHP.
- Para aplicaciones de estudio de grandes conjuntos de datos y estadísticas, se ha cementado como uno de los lenguajes principales, junto al lenguaje **R**.

Entre las razones de que se pueda usar el mismo lenguajes para desarrollar aplicaciones tan disimiles se encuentran la versatilidad de Python como lenguaje de programación y su relativa simplicidad. En este sentido, Python fue diseñado para soportar múltiples paradigmas de programación. En mayor o menor medida, Python soporta los paradigmas imperativo, funcional, orientado a objetos y procedural.

Problema: Dada una lista de números, calcular la suma del doble de los cuadrados de los números en la lista. [1]
Solución: Suponiendo que la lista es A y tiene n elementos, la solución es calcular la siguiente sumatoria:

$$sum = \sum_{i=0}^{n} 2 * (A[i]^2)$$

2.1. Como lenguaje imperativo

En una solución en estilo imperativo, el foco estará en *como* obtenemos el resultado. El programa tendrá un estado que irá cambiando a medida que el programa progrese, con el fin de encontrar una solución.

```
my_list = [1, 2, 3, 4, 5]
sum = 0

for x in my_list:
    square = x * x
    doble = 2 * square
    sum += doble

print(sum)
```

En este ejemplo, el estado del programa está íntimamente ligado a las variables `sum` que lleva la suma parcial de los elementos vistos hasta ahora, y las variables `x`, `square` y `doble` que computan los resultados parciales para cada uno de los elementos vistos.

2.2. Como lenguaje orientado a objetos

En una solución orientada a objetos, el foco estará en conseguir que el programa sea lo más reutilizable posible. Usaremos un objeto propio para definir la funcionalidad que buscamos.

```
class MyList():
    def __init__(self, any_list):
        self.any_list = any_list
```

[1] Contenido adaptado del blog https://newrelic.com/blog/nerd-life/python-programming-styles

```
    def square(self):
        return [ x * x for x in self.any_list ]

    def doble(self):
        return [ 2 * x for x in self.square() ]

    def sum(self):
        return sum(self.doble())

my_list = MyList([1, 2, 3, 4, 5])

print(my_list.sum())
```

Veremos la sintaxis de esta solución en detalle en secciones futuras, pero por ahora podemos adelantar que diremos que el *objeto* `my_list` es una *instancia* de la clase `MyList`, y que el *método* `sum` realiza los cálculos, sin necesidad de conocer desde fuera de la clase los métodos `square` y `doble`.

3. Programación orientada a objetos

3.1. Historia

El lenguaje de programación **Simula** sentó en la década del '60 las bases conceptuales de clase y objeto. Estos conceptos se trasladaron a los lenguajes **Smalltalk** y **Lisp** en la década de los '70. Fue en los años '80 que el interés por estos conceptos se extendieron en el público general de programadores, y se comenzó a trabajar en extensiones del lenguaje para C para soportar clases.
Estos esfuerzos dieron nacimiento a los lenguajes **Objective-C** y **C++**, el último de los cuáles sigue siendo popular hoy en día. También nace en esta época el lenguaje **Eiffel**, el primero explícitamente diseñado para ser orientado a objetos. Durante la década del '90 la programación orientada a objetos se convirtió en el paradigma dominante para el desarrollo de software industrial y surgieron decenas de lenguajes.
La popularidad se incrementó ha medida que también lo hacía la necesidad de programar interfaces gráficas. Al mismo tiempo, investigadores académicos buscaban formalizar la teoría de la programación orientada a objetos mediante la investigación de tópicos como la abstracción de datos y la modularización de programas. El científico de la computación Alan Kay fue pionero en esta investigación y fue el quien acuño los términos *objeto* y *clase*. Alan Kay recibió el premio Turing, honor equivalente a un premio Nobel en el campo de la computación, "por ser pionero en muchas de las ideas en la raíz de de los lenguajes orientados a objetos contemporáneos [...] y por sus contribuciones fundamentales a la computadora personal".
En 1996 surge un desarrollo llamado **JAVA**, en principio pensado como una extensión de C++, se transformó en un lenguaje independiente. Es considerado por muchos como el lenguaje orientado a objetos definitivo, debido a su explícita cohesión con este paradigma.
Actualmente, el paradigma de programación orientada a objetos sigue siendo el paradigma principal utilizado en la industria del desarrollo de software.
Más recientemente comenzaron a popularizarse lenguajes que soportan un enfoque no purista o mixto de la orientación a objetos, entre ellos **Python**, **Ruby** y **Javascript**.

3.2. Introducción

La programación orientada a objetos es un paradigma de programación que nos permite organizar el código de manera que se asemeje a como pensamos en la vida real. Así, pensamos en los problemas como un conjunto de *objetos* que se relacionan entre sí. Estos *objetos* nos permiten agrupar un conjunto de variables y funciones relacionadas en un mismo espacio de nombres, facilitando la abstracción de pensamiento y la modularización del programa.
Cosas de lo más cotidianas pueden pensarse como un objeto, desde un gato o un auto. Cada uno de estos objetos tiene ciertas características. Por ejemplo, para el caso de un gato podemos decir cual es su tamaño, su color de pelo o su mestizaje. A estas características las llamamos *atributos*.

Por otro lado, cada objeto tiene una serie de comportamientos que lo distingue, por ejemplo en el caso de perro podrían ser caminar o maullar. A estos comportamientos o funcionalidad propia de cada objeto los llamaremos *métodos*.

Definición Programación Orientada a Objetos La programación orientada a objetos es un paradigma de programación basado en la creación de objetos que pueden contener datos y funciones.

Definición Objeto Un objeto es un ente que consta de identidad, estado y de un comportamiento.

3.3. Clases como tipos de datos

Una *clase* es una plantilla para la creación de nuevos objetos.
Las clases nos permiten modularizar los datos y la funcionalidad asociada a un tipo de objeto. Una forma sencilla de pensar a las clases es pensarlas como un nuevo tipo de dato. Así, mientras representaremos a distintos gatos con distintos objetos, representaremos la idea abstracta de un gato mediante una clase Gato. Al definir una clase debemos definir que atributos y que métodos contendrán los objetos pertenecientes a esta clase. Cuando un objeto pertenezca a una clase, diremos que ese objeto es una *instancia* de la clase. Cada objeto individual tiene sus propios atributos y métodos, la clase solo nos provee una vista unificada de lo que tienen en común todos los objetos de este tipo.

3.4. Composición de Clases

Como una clase nos define un nuevo tipo de dato, se puede utilizar este tipo de dato en la definición de nuevas clases. Por ejemplo, suponiendo que tenemos que conservar más información sobre el dueño del gato, podríamos definir una clase `PetOwner` con la información especifica del dueño y luego utilizar el atributo `owner` para guardar un objeto de tipo `PetOwner`. En el diagrama de clase esto lo representamos uniendo ambas clases con una flecha con un rombo blanco en el extremo de la clase.

4. POO en Python

En Python todo con lo que hayamos trabajo es un objeto (aunque no lo supiéramos!). Esto quiere decir que, en Python, el número `5`, el string `"foo"` y la lista `[1,2,3,4,5]` son objetos.

Python provee algunas clases ya definidas para nosotros, como por ejemplo `int`, `str` y `list`.

4.1. Definición de clases propias

Python también nos ofrece sintaxis para declarar y utilizar nuestras propias clases, de manera muy similar a funciones. Para definir nuevas clases, se hace uso de la palabra reservada `class`. Por convención, nombraremos en python a las clases en CamelCase (primera letra de cada palabra en mayúscula). Esto nos ayudará a distinguir a golpe de vista cuando hablamos de un clase y cuando hablamos de un objeto.

La clase más sencilla que podemos definir esta vacía:

```
class Vacia:
    """esta clase no hace nada!"""
    pass
```

Nota: En esta definición utilizamos además la palabra reservada `pass` [2]. Esta instrucción no hace nada y puede utilizarse siempre que necesitemos que haya una instrucción pero no necesitamos que pase nada.

Esta es una clase sin atributos y sin métodos, que puede parecer poco útil, pero demuestra lo esencial de la definición de clases: la palabra reservada seguida del nombre de la clase, y con dos puntos abrimos un nuevo nivel de identación. Al igual que con las funciones, llamaremos a esta primer linea el *encabezado* de la clase,

[2] https://docs.python.org/3/tutorial/controlflow.html#pass-statements

y seguiremos inmediatamente con un comentario a modo de documentación de la clase. Esto nos permite tener esa documentación a mano utilizando la función `help`:

```
>>> help(Vacia):
esta clase no hace nada!
```

¿Como creamos nuevos objetos que pertenezcan a la clase Vacia? Fácil, simplemente invocamos a Vacia como si fuera una función

```
>>> x = Vacia()
>>> print(x)
<__main__.Vacia object at 0x7fbfab6b6d90>
>>> y = Vacia()
>>> print(y)
<__main__.Vacia object at 0x7f035c710370>
>>> z = x
>>> print(z)
<__main__.Vacia object at 0x7fbfab6b6d90>
>>> x is y
False
>>> x is z
True
```

¿Que resulta de imprimir un objeto de tipo Vacia? Cuando Python no sabe como mostrar por pantalla un objeto predeterminado, simplemente escribe en la pantalla que tipo de objeto es y cuál es su ubicación en la memoria física. Más adelante veremos como mejorar esto con información mas interesante, pero por ahora nos concentraremos en la *identidad* de los objetos. En Python los objetos se guardan en una posición de memoria especifica y esa posición de memoria es utilizada como identidad del objeto. Las variables no son mas que *referencias* al objeto subyacente, es por eso que, en el ejemplo, `x` e `y` son objetos distintos, pero `x` y `z` son el mismo (mas precisamente, son el objeto `0x7fbfab6b6d90`).

Nota: Podemos usar el operador `is` para ver si dos variables son realmente el mismo objeto o no.

Si pensamos en las clases como nuevos tipos de datos, podemos, por ejemplo, crear un nuevo tipo de datos que represente un punto en el espacio bidimensional. En notación matemática, los puntos son a menudo escritos en paréntesis con una coma que separa las coordenadas. Por ejemplo, (0, 0) representa el origen, y (x, y) representa el punto que está x unidades hacia la derecha e y unidades hacia arriba, a partir del origen. Hay muchas maneras en las cuales podríamos representar puntos en Python:

- Podríamos almacenar las coordenadas de manera separada en dos variables, x e y.
- Podríamos almacenar las coordenadas como elementos en una lista o tupla.
- Podríamos crear un tipo nuevo para representar puntos como objetos.

```
class Point:
      """Representa un punto en un espacio 2-D."""
      def __init__(self, x, y):
          self.x = x
          self.y = y
```

Si queremos crear objetos de tipo `Point`, invocamos a `Point` como si fuera una función.

```
v = Point(1, 2)
```

El valor de retorno de la llamada `Point(1, 2)` es un nuevo objeto de tipo `Point`, que asignamos a la variable `v`.

Definición El proceso de crear un objeto nuevo se llama **instanciación**. Decimos que el objeto es una `instancia` de la clase.

Los atributos son **observables**, podemos obtener su valor utilizando la notación de punto:

```
print(v.x) # Imprime 1
print(v.y) # Imprime 2
```

Los atributos son **asignables**, podemos re-asignarles valores utilizando la notación de punto:

```
v.x = 3
v.y = 4
```

A nivel general, la expresión `v.x` significa "Ve al objeto `v` y obtén el valor de su atributo `x`"

Observación Cada objeto tiene su propio *scope* de variables. Por ejemplo, en el siguiente fragmento:

```
x = Point(1.3, -1)
y = Point(1, 13)
print(x.x) # Imprime 1.3
print(y.x) # Imprime 1
```

No hay ningún conflicto respecto al nombre de la variable `x`, puesto que se encuentra definida globalmente y no dentro de un objeto. A su vez, no hay ningún conflicto entre los atributos `x.x` e `y.x` pues se refieren a objetos distintos.

Intentemos ahora definir alguna clase más divertida

```
class Cat:
    def __init__(
        self, name, weight,
        owner, sex, hair_color,
    ):
        self.name = name
        self.weight = weight
        self.owner = owner
        self.sex = sex
        self.hair_color = hair_color
```

Los atributos `name`, `weight`, `owner`, `sex` y `hair_color` son atributos de instancia, varían de gato a gato.

En esta clase también vemos nuestra primera definición de un método, el método `__init__`. Este método es muy especial, nos dice como armar un nuevo objeto de tipo `Cat` a partir de las características que lo describen. A este tipo de métodos los llamamos *constructores*. El constructor `__init__` es al que Python llamará cuando creemos una nueva instancia de la clase

```
melon = Cat(
    "Melon", 6, "Sofi", "Macho",
    ["blanco", "naranja"],
)

amelia = Cat(
    "Amelia", 4, "Pauli", "Hembra",
```

```
    ["negro", "marron", "blanco"],
)

bobo = Cat("Bobo", 5, "Mela", "Macho", ["Gris"])
```

Con cada llamada a la clase `Cat`, Python se esta encargando de llamar al método constructor `__init__`.

Por convención, el primer argumento de cada método lo llamaremos `self`. Este argumento se refiere a la instancia actual del objeto con el que estamos trabajo.

La clase `Cat` podría tener más métodos

```
class Cat:
    def __init__(
        self, name, weight,
        owner, sex, hair_color,
    ):
        self.name = name
        self.weight = weight
        self.owner = owner
        self.sex = sex
        self.hair_color = hair_color

    def meow(self):
        print(f"{self.name} says 'meow'")

    def play(self, toy):
        print(f"{self.name} is playing with {toy}!")
```

Cada instancia puede ahora hacer uso de esos métodos. Notar que el argumento `self` solo tiene sentido al momento de definir el método en la clase, ya que cuando una vez que instanciamos, el objeto ya tiene identidad propia.

```
melon = Cat(
    "Melon", 6, "Sofi", "Macho",
    ["blanco", "naranja"],
)

melon.meow() # No necesita argumento, imprime
# Melon says 'meow'

melon.play('ball') # Solo lleva un argumento, imprime
# Melon is playing with ball
```

4.2. Composición: Ejemplos

A veces es obvio cuáles deberían ser los atributos de un objeto, pero otras veces hay que tomar decisiones. Por ejemplo, supongamos que estamos diseñando una clase para representar rectángulos en un espacio bidimensional. ¿Qué atributos utilizarías para especificar la ubicación y tamaño de un rectángulo? Para simplificar la tarea, ignoraremos el ángulo, supongamos que el rectángulo es vertical u horizontal. Hay, al menos, dos posibilidades:

- Podríamos especificar la esquina inferior izquierda, la anchura y la altura.

- Podríamos especificar dos esquinas opuestas.

Para ejemplificar, implementaremos la primera opción:

```
class Rectangle:
    """Representa un rectangulo.
    atributos: anchura, altura, esquina
    """
    def __init__(self, anchura, altura, esquina):
        self.anchura = anchura
        self.altura = altura
        self.esquina = esquina
```

Los atributos `anchura` y `altura` son números de punto flotante, mientras que `esquina` es un objeto de tipo `Point`. Decimos que hay **composición** porque para construir los rectangulos necesitamos construir otro objeto primero, el punto.

Para representar un rectangulo, debemos instanciar la clase

```
pt = Point(0, 0)
r = Rectanle(4, 3, pt)
```

Podemos hacer referencia a los atributos a varios niveles. Por ejemplo, la expresión `r.esquina.x` significa "Ve al objeto `r`, obtén el valor del atributo `esquina`; luego ve a ese objeto y obtén el valor del atributo `x`".

5. Herencia

La *herencia* es un mecanismo muy simple, pero muy poderoso, para alcanzar el objetivo de modularidad y reusabilidad del código. A partir de ella, podemos crear nuevas clases basándonos en una clase anterior que ya teníamos definida, evitando así redefinir y reimplementar comportamiento comunes a ambas clases.
Por ejemplo, si bien conseguimos abstraer las propiedades y comportamientos comunes a un gato a una clase `Cat`, puede que esto no sea necesario, pues también necesitamos representar perros en nuestro dominio. Al implementar la clase para perros, deberíamos volver a implementar un constructor con los mismos atributos, el método `play` también es el mismo... en realidad la única diferencia es que en lugar de un método para maullar deberíamos tener un método para ladrar. Entonces lo que hacemos es abstraer las propiedades y comportamientos comunes a una clase `Pet` (Mascota) y luego heredamos de ella para implementar las clases `Cat` y `Dog`

5.1. Herencia en Python

```
class Pet:
    def __init__(
        self, name, weight,
        owner, sex, hair_color,
    ):
        self.name = name
        self.weight = weight
        self.owner = owner
        self.sex = sex
        self.hair_color = hair_color

    def play(self, toy):
        print(f"{self.name} is playing with {toy}!")

    def eat(self):
```

```
        print(f"{self.name} is eating")
        self.weight += 0.1

class Cat(Pet):
    def meow(self):
        print(f"{self.name} says 'meow'")

class Dog(Pet):
    def woof(self):
        print(f"{self.name} says 'woof'")

bobo = Cat("Bobo", 5, "Mela", "Macho", ["Gris"])
leon = Dog("León", 26, "Romi", "Macho", ["Marron"])

bobo.meow()
leon.woof()

bobo.eat()
leon.eat()
```

Para declarar una herencia, simplemente aclaramos en el encabezado de la clase de que otra clase estamos heredando, como si fuera un argumento.

Nota: En Python una clase puede heredar de varias clases, en un esquema que se conoce como *herencia múltiple*, pero no lo veremos en este curso.

Decimos que la clase `Cat` *hereda de* la clase `Pet` o también que `Cat` es una clase hija de `Pet`. A su vez, `Pet` es la clase padre o clase base de `Cat`.

Como podemos ver en el ejemplo, tanto las instancias de `Dog` como de `Cat` tomaron todos los atributos y métodos que tenía la clase `Pet` y además fueron capaces de agregar métodos propios.

Tengamos en cuenta además, que ahora `bobo` es una instancia tanto de `Cat` como de `Pet`, pero no de `Dog`

```
>>> isinstance(bobo, Pet)
True
>>> isinstance(bobo, Cat)
True
>>> isinstance(bobo, Dog)
False
```

La *jerarquía de clases* podría ser mas complicada, tanto por abstracción (es decir, definir clases más generales que `Pet`) como por especialización (es decir, definir clases más específicas que `Dog` o `Cat`).

Definimos *clase ancestra*, *clase descendiente* y *clase hermana* de la misma forma que si las clases estuvieran formando un árbol genealógico con las relaciones de herencia.

Del mismo modo, se suele utilizar *superclase* como sinónimo de *clase padre* y *subclase* como sinónimo de *clase hija*

Nota: En realidad, en Python todas las clases heredan de una clase base común que es la clase `object`. Esta clase implementa cierto comportamiento básico y esencial sin el cuál Python no podría funcionar. Cuando escribimos una clase sin aclarar ninguna herencia, Python entiende que queremos heredar de `object` y no de nada más específico.

5.2. *Override* de métodos

Muchas veces ocurrirá que los métodos son casi los mismos, pero no exactamente iguales. Por ejemplo, suponiendo que un perro come más que un gato, el método `eat` de la clase `Pet` es válido para ambos, pero para perros es más difícil de usar, puesto que hay que llamarlo más veces.

El *override* [3] de métodos es una habilidad de cualquier lenguaje orientado a objetos que permite que una clase hija provea una implementación más específica de un método que ya provee una clase ancestro. Es importante que, para que el override del método este bien hecho, el método tenga el mismo nombre, tome los mismos argumentos y tenga el mismo tipo de retorno que el método en su clase ancestro.

La versión o sabor del método que será ejecutado será determinado por la clase más específica que provea el método.

```
class Dog(Pet):
    def woof(self):
        print(f"{self.name} says 'woof'")

    def eat(self):
        print(f"{self.name} is eating!")
        self.weigth += 0.2

    bobo = Cat("Bobo", 5, "Mela", "Macho", ["Gris"])
    leon = Dog("León", 26, "Romi", "Macho", ["Marron"])

    print(f"Antes de comer, bobo pesa: {bobo.weigth}")
    # Antes de comer, bobo pesa 5
    print(f"Antes de comer, leon pesa: {leon.weigth}")
    #Antes de comer, leon pesa 26
    bobo.eat()
    leon.eat()
    print(f"Despues de comer, bobo pesa: {bobo.weigth}")
    # Despues de comer, bobo pesa 5.1
    print(f"Despues de comer, leon pesa: {leon.weigth}")
    # Despues de comer, leon pesa 26.2
```

5.3. La función `super`

¿Que pasa si en lugar de querer agregar un nuevo método, queremos agregar un nuevo atributo de instancia?

Deberíamos hacer override del constructor, pero eso nos llevaría a repetir el código para todos los atributos compartidos.

Para evitar esto, Python nos provee una función llamada `super` con la cual podemos utilizar dentro de una definición de clase, los métodos de la clase padre correspondiente y así replicar el comportamiento del método en la clase padre.

```
class Pet:
    def __init__(
        self, name, weight,
        owner, sex,
    ):
        self.name = name
        self.weight = weight
```

[3] En español, *override* puede traducirse como anulación, supresión, superación o sustitución; pero ninguna resulta del todo adecuada. Así, seguiremos la convención general y nos referiremos a este concepto directamente con su nombre en inglés.

```
        self.owner = owner
        self.sex = sex

class Cat(Pet):
    def __init__(self, character, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.character = character

bobo = Cat("friendly", "Bobo", 5, "Mela", "Macho", )
print(bobo.character) # friendly
print(bobo.name) # Bobo
```

6. Métodos especiales en Python

Los métodos especiales, *métodos mágicos*, o *dunders* son métodos que, si están definidos para un objeto, Python se encargara de invocarlos ante ciertas circunstancias. El nombre dunder es un anglicismo proveniente de la contracción de *double underscore*, que significa "doble guión bajo", dado que muchos de estos métodos se escriben empezando y terminando en doble guión bajo.

6.1. El constructor `__init__`

El método constructor `__init__`, como ya dijimos, será llamado internamente por Python cada vez que instanciemos un nuevo objeto de una clase. El constructor hace explícito que atributos maneja la clase. Se considera una buena práctica definir un constructor para cada clase que escribamos, ya que facilitan la lectura del código y documenta la forma apropiada de utilizar una clase.

6.2. Imprimiendo objetos: El método `__str__`

Ya vimos que cuando imprimimos un objeto, el resultado no es muy agradable a la vista. Para corregir esto podemos utilizar el método mágico `__str__` que será llamado por Python cada vez que quiera mostrarnos por pantalla un objeto. Mediante este método esperamos obtener una descripción amigable y legible del objeto, que contenga todos los datos o al menos aquellos que consideremos más relevantes.

```
class Cat:
    number_of_legs = 4

    def __init__(
        self, name, weight,
        owner, sex,
    ):
        self.name = name
        self.weight = weight
        self.owner = owner
        self.sex = sex

    def __str__(self):
        return f"Soy {self.name} y mi duenia es {self.owner}"

bobo = Cat("Bobo", 5, "Mela", "Macho", )
print(bobo) # Soy Bobo y mi duenia es Mela
```

6.3. Igualdad de objetos: El método `__eq__`

Ya hablamos de la identidad de objetos, y vimos que Python la representa con la dirección de memoria del objeto. Sin embargo, es importante hacer una distinción entre las palabras *identidad* e *igualdad*. Por

ejemplo, cuando decimos que Juan y María tienen el mismo auto, seguramente nos referimos a que cada uno tiene un auto propio, pero que a efectos de la conversación donde haya surgido la frase, ambos autos son indistinguibles entre sí. Diremos que dos objetos son iguales cuando representen el mismo valor, más allá de si realmente son el mismo objeto o no.

```
bobo1 = Cat("Bobo", 5, "Mela", "Macho", [])
bobo2 = Cat("Bobo", 5, "Mela", "Macho", [])
print(bobo1 is bobo2) # False
print(bobo1 == bobo2) # False
```

¿Como decide Python si `bobo1` y `bobo2` son el mismo objeto? ¿Por su nombre? ¿Por todas las características? En general, Python no puede decidir esto automáticamente, así que en caso de duda, prefiere asociar la igualdad de los objetos con su identidad. Podemos cambiar este comportamiento utilizando el método `__eq__`. Por ejemplo, si suponemos que un mismo dueño nunca va a tener dos gatos distintos con el mismo nombre, una implementación razonable del método `__eq__` es la siguiente:

```
class Cat:
    ...
    def __eq__(self, other):
        return self.owner == other.owner and self.name == other.name


bobo1 = Cat("Bobo", 5, "Mela", "Macho", )
bobo2 = Cat("Bobo", 5, "Mela", "Macho", )
print(bobo1 is bobo2) # False
print(bobo1 == bobo2) # True
```

6.4. Sobrecarga de operadores

Esto se conoce como *sobrecarga de operadores* [4] ya que le estamos dando al operador == un nuevo significado que se adapta mas para el contexto en el que estamos trabajando. Casi todos los operadores que conocemos en Python se pueden sobrecargar, implementando en la clase un método mágico apropiado. Se deja como referencia la siguiente tabla con los operadores matemáticos y de comparación y el método mágico que se debe implementar para sobrecargarlo.

Operador	Método Mágico
+	__add__(self, other)
-	__sub__(self, other)
*	__mul__(self, other)
/	__truediv__(self, other)
%	__mod__(self, other)
**	__pow__(self, other)
<	__lt__(self, other)
<	__gt__(self, other)
<=	__le__(self, other)
>=	__ge__(self, other)
==	__eq__(self, other)
!=	__ne__(self, other)

7. Copiando objetos

Supongamos que tengo un objeto tipo `Point` y uno de tipo `Rectangle`.

[4] *operator overloading*, en inglés.

```
>>> p = Point(3, 4)
>>> r = Rectangle(10, 2, p)
```

Si quisiera crear un objeto de tipo punto igual al objeto `p`, intuitivamente podríamos suponer que tenemos que hacer lo siguiente:

```
>>> q = p
```

El problema es que aquí no estamos realmente instanciando un nuevo objeto, si no que hay un solo objeto y dos nombres distintos para acceder a él. Eso se comprueba sencillamente cambiando los valores en un objeto y viendo que los cambios se reflejan en ambas variables:

```
>>> q.x = 2
>>> print(p.x)
2
```

Podemos usar el operador `is` para confirmar si dos variables se refieren al mismo objeto:

```
>>> p is q
True
```

Usualmente, necesitamos crear objetos que sean iguales a uno que tenemos, pero que sean *independientes* de este. Podemos usar para ello las funciones del modulo de python `copy`.

```
>>> import copy
>>> q = copy.copy(p)
>>> p is q
False
```

El módulo `copy` es genérico y lo podemos utilizar con cualquier objeto. Por ejemplo, podemos usarlo para copiar rectangulos:

```
>>> import copy
>>> r2 = copy.copy(r)
>>> r is r2
False
>>> r.esquina is r2.esquina
True
```

Algo importante es que cuando tenemos un objeto dentro de otro (composición), los objetos embebidos no se copian, si no que siguen siendo referencias al mismo objeto subyacente. Por eso decimos que `copy.copy` es una **copia superficial** (o *shallow copy*, en inglés).

Si necesitamos copiar el objeto y todos los objetos contenidos dentro de forma recursiva, hay que utilizar una **copia profunda** o *deep copy*.

```
>>> import copy
>>> r2 = copy.deepcopy(r)
>>> r is r2
False
>>> r.esquina is r2.esquina
False
```

8. Errores comunes

Cuando comienzas a trabajar con objetos, es probable que encuentres algunas excepciones nuevas. Si intentas acceder a un atributo que no existe, obtienes un `AttributeError`:

```
>>> p = Point(3, 4)
>>> p.z
AttributeError: Point instance has no attribute 'z'
```

Si no sabes bien de que tipo es un objeto, puedes utilizar la función `type` para verificarlo:

```
>>> type(p)
<class '__main__.Point'>
```

Python también nos provee la función `isinstance` para verificar si un objeto es instancia de una clase. Por ejemplo:

```
>>> isinstance(5, int)
True
>>> isinstance(2.3, int)
False
>>> isinstance("foo", str)
True
>>> isinstance([1,2,3,4,5], list)
True
```

Si no sabes bien si un objeto tiene un atributo en particular, puedes utilizar la función incorporada `hasattr`[5]:

```
>>> hasattr(p, 'x')
True
>>> hasattr(p, 'z')
False
```

Para la función `hasattr`, el primer argumento puede ser cualquier objeto, el segundo argumento es un string que contiene el nombre del atributo.

Referencias

[1] A. Downey et al, 2002. *How to Think Like a Computer Scientist. Learning with Python.* Capitulos 12 a 16

[2] https://docs.python.org/es/3/tutorial/classes.html

[5] `hasattr` proviene del inglés *has attribute*.