# 03c.Mas_sorbre_programacion_orientada_objetos

October 21, 2022

## 1 Programación orientada a objetos (POO) en Python

La programación orientada a objetos (POO) es una forma de organizar el código. Así como un algoritmo suele estar asociado a una estructura de datos particular, la programación orientada a objetos "empaqueta" los datos junto con los métodos usados para tratarlos.

Python también permite la POO, que es un paradigma de programación en la que los datos y las operaciones que pueden realizarse con esos datos se agrupan en unidades lógicas llamadas objetos.

Cada uno de esos *objetos* consiste en:

- Datos (atributos de los objetos).
- Comportamiento (métodos de los objetos: son funciones que actúan sobre los atributos del objeto).

Por ejemplo, al manipular una lista:

[ ]:
```
nums = [1, 2, 3]    # Lista
nums.append(4)      # Esto es un método de la lista
nums.insert(1,10)   # Otro método de la lista
nums
[1, 10, 2, 3, 4]    # Estos son los datos modificados por los métodos
```

[ ]:
```
[1, 10, 2, 3, 4]
```

Miremos un poco más en detalle este fragmento de código. Sabemos que `nums` es una variable de tipo lista. Equivalentemente, podemos decir que `nums` es una *instancia* de la clase *list*. Cada variable de tipo lista es una instancia de la misma clase. Al hablar de 'instancia' nos referimos a un 'objeto': un objeto es una instancia de una clase.

Un objeto de tipo lista tiene atributos (datos) y métodos. Los métodos, como `append()` o `insert()`, se definen cuando se define la clase, pero se usan para manipular los datos de un objeto concreto (`nums` en este caso).

### 1.1 La instrucción `class`

Para definir un nuevo tipo de objeto en Python, se usá la instrucción `class`.

```
class Nombre_del_objeto:
    def __init__(self, atributo1, atributo2, etc):
        self.atributo1 = x
        self.atributo1 = y
```

```
        self.etc     = 100

    def Metodo1(self, dx, dy):
        self.atributo1 = self.atributo1 + dy
        self.atributo2 = self.atributo2 + dy

    def Metodo2(self, pts):
        self.etc = self.etc - pts
```

Un objeto de tipo `Nombre_del_objeto` tiene como atributos `atributo1`, `atributo2`, `etc`. Sus métodos son `Metodo1` y `Metodo2`.

Puede decirse que una clase es la definición formal de las relaciones entre los datos y los métodos que los manipulan. Un objeto es una instancia particular de la clase a la cual pertenece, con datos propios pero los mismos métodos que los demás objetos de esa clase. Este concepto te va a quedar más claro cuando lo veas funcionar y lo uses.

## 1.2 Instancias

Los programas manipulan instancias individuales de las clases. Cada instancia es un objeto, y es en cada objeto que uno puede manipular los datos y llamar a sus métodos.

Podemos crear un objeto mediante un llamado a la clase como si fuera una función.

Primero creemos nuestra clase `Jugador`:

[ ]:
```
class Jugador:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.salud = 100

    def mover(self, dx, dy):
        self.x += dx
        self.y += dy

    def lastimar(self, pts):
        self.salud -= pts
```

Ahora podemos defininir nuestras clases:

[ ]:
```
a = Jugador(2, 3)
b = Jugador(10, 20)
```

`a` y `b` son instancias de `Jugador` definida más arriba. Es decir, a y b son objetos de la clase `Jugador`.

### 1.3 Datos de una instancia

[ ]:
```
print(a.x)
print(a.y)
```

```
2
3
```

Estos datos locales se inicializan, para cada instancia, durante la ejecución del método `__init__()` de la clase.

```
class Jugador:
    def __init__(self, x, y):
        # Todo dato guardado en `self` es propio de esa instancia
        self.x = x
        self.y = y
        self.salud = 100
```

No hay restricciones en la cantidad o el tipo de atributos que puede tener una clase.

### 1.4 Métodos de una instancia

Los métodos de una instancia son los métodos y las funciones que actúan sobre los datos almacenados en esa instancia.

```
class Jugador:
    ...
    # `mover` es un método
    def mover(self, dx, dy):
        self.x += dx
        self.y += dy
```

Siempre se recibe la instancia misma como primer argumento: "self" significa "mismo" como en "mi mismo" ó "en sí misma". Es como decir "yo".

[ ]:
```
a.mover(1, 2)
print(a.x)
print(a.y)
```

```
3
5
```

Por convención siempre llamamos `self` a la instancia actual, y ésta es siempre pasada como primer argumento a todos los métodos. En realidad el nombre real de la variable no importa, pero es una convención en Python llamar al primer argumento `self`.

```
>>> a.mover(1, 2)

# `self` refiere a `a`
# `dx` refiere a `1`
# `dy` refiere a `2`
```

```
def mover(self, dx, dy):
    ...
```

Podríamos usar `mismo`, por ejemplo, en lugar de `self` y todo va a funcionar igual, pero no respeta las convenciones de la comunidad:

```
class Jugador:
    ...
    # `mover` es un método
    def mover(mismo, dx, dy):
        mismo.x += dx
        mismo.y += dy
```

### 1.5 Ejercicios: Objetos como estructura de datos.

Durante las primeras clases trabajamos con datos en forma de tuplas y diccionarios. Un lote con cajones de frutas, por ejemplo, estaba representado por una tupla, como ésta:

```
s = ('Pera', 100, 490.10)
```

```
s = {
    'nombre'  : 'Pera',
    'cajones' : 100,
    'precio'  : 490.10
     }
```

Incluso podemos escribir funciones para manipular datos almacenados de ese modo:

```
def costo(registro):
    return registro['cajones'] * registro['precio']
```

Otra forma de representar los datos con los que estás trabajando es definir una clase. Creá una clase llamada `Lote` que represente un lote de cajones de una misma fruta. Definila de modo que cada instancia de la clase `Lote` (es decir, cada objeto lote) tenga los atributos `nombre`, `cajones`, y `precio`. Éste es un ejemplo del comportamiento buscado:

Vamos a crear más objetos de tipo `Lote` para manipularlos. Por ejemplo:

```
>>> b = Lote('Manzana', 50, 122.34)
>>> c = Lote('Naranja', 75, 91.75)
>>> b.cajones * b.precio
6117.0
>>> c.cajones * c.precio
6881.25
>>> lotes = [a, b, c]
>>> lotes
[<lote.Lote object at 0x37d0b0>, <lote.Lote object at 0x37d110>, <lote.Lote object at 0x37d050
>>> for c in lotes:
     print(f'{c.nombre:>10s} {c.cajones:>10d} {c.precio:>10.2f}')

... mirá el resultado ...
>>>
```

Fijate que la clase `Lote` funciona como una "fábrica" para crear objetos que son instancias de esa clase. Nosotros la llamás como si fuera una función y te crea una nueva instancia de sí misma. Más aún, cada instancia es única y tiene sus propios datos que son independientes de las demás instancias de la misma clase.

Una instancia definida por una clase puede tener cierta similitud con un diccionario, pero usa una sintaxis algo diferente. Por ejemplo, en lugar de escribir `c['nombre']` ó `c['precio']` en objetos escribís `c.nombre` o `c.precio`.

[ ]:
```
class Lote:
    def __init__(self, nombre, cajones, precio):
        self.nombre  = nombre
        self.cajones = cajones
        self.precio  = precio
```

[ ]:
```
# Creamos un objeto

a = Lote('Pera', 100, 490.10)
```

[ ]:
```
a.nombre
```

[ ]:
```
'Pera'
```

[ ]:
```
a.cajones
```

[ ]:
```
100
```

[ ]:
```
a.precio
```

[ ]:
```
490.1
```

[ ]:
```
b = Lote('Manzana', 50, 122.34)
c = Lote('Naranja', 75, 91.75)
```

[ ]:
```
lotes = [a, b, c]
lotes
```

[ ]:
```
[<__main__.Lote at 0x7ff2900798b0>,
 <__main__.Lote at 0x7ff29007e3d0>,
 <__main__.Lote at 0x7ff29007e1f0>]
```

[ ]:
```
for c in lotes:
    print(f'{c.nombre:>10s} {c.cajones:>10d} {c.precio:>10.2f}')
```

```
      Pera         100     490.10
   Manzana          50     122.34
   Naranja          75      91.75
```

### 1.6 Agregámos algunos métodos a nuestra clase `Lote`

[ ]:
```
class Lote:
    def __init__(self, nombre, cajones, precio):
        self.nombre  = nombre
        self.cajones = cajones
        self.precio  = precio

    def costo(self):
        return self.cajones * self.precio

    def vender(self, ncajones):
        self.cajones -= ncajones
```

[ ]:
```
s = Lote('Pera', 100, 490.10)
```

[ ]:
```
s.costo()
```

[ ]:
```
49010.0
```

[ ]:
```
s.vender(25)
```

[ ]:
```
s.cajones
```

[ ]:
```
75
```

[ ]:
```
s.costo()
```

[ ]:
```
36757.5
```

## 2 Métodos especiales

Podemos modificar muchos comportamientos de Python definiendo lo que se conoce como "métodos especiales". Acá vamos a ver cómo usar estos métodos y a discutir brevemente otras herramientas relacionadas.

Una clase puede tener definidos métodos especiales. Estos métodos tienen un significado particular para el intérprete de Python. Sus nombres empiezan y terminan en `__` (doble guión bajo). Por ejemplo `__init__`.

```
class Lote(object):
    def __init__(self):
        ...
    def __repr__(self):
        ...
```

Hay decenas de métodos especiales pero sólo vamos a tratar algunos ejemplos específicos acá.

### 2.0.1 Métodos especiales para convertir a strings

Las funciones `str()` y `repr()` llaman a métodos especiales de la clase para generar la cadena de caracteres que se va a mostrar.

**Ejemplo:**

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

    def __str__(self):
        return f'({self.x}, {self.y})'

    # Used with `repr()`
    def __repr__(self):
        return f'Punto({self.x}, {self.y})'
```

### 2.0.2 Métodos matemáticos especiales

Las operaciones matemáticas sobre los objetos involucran llamados a los siguientes métodos.

```
a + b       a.__add__(b)
a - b       a.__sub__(b)
a * b       a.__mul__(b)
a / b       a.__truediv__(b)
a // b      a.__floordiv__(b)
a % b       a.__mod__(b)
a << b      a.__lshift__(b)
a >> b      a.__rshift__(b)
a & b       a.__and__(b)
a | b       a.__or__(b)
a ^ b       a.__xor__(b)
a ** b      a.__pow__(b)
-a          a.__neg__()
~a          a.__invert__()
abs(a)      a.__abs__()
```

Así, al definir un método `__add__(b)` en la clase `Punto`, por ejemplo, nos permitirá sumar dos instancias de esta clase usando el operador `+`.

**Ejemplo:**

[ ]:
```
class Punto():
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __repr__(self):
        return f'Punto({self.x}, {self.y})'
```

```
    def __add__(self, b):
        return Punto(self.x + b.x, self.y + b.y)
```

[ ]:
```
a = Punto(1,2)
b = Punto(3,4)
repr(a + b)
```

[ ]:
```
'Punto(4, 6)'
```

# 3 Herencia

La herencia entre clases es una herramienta muy usada para escribir programas extensibles.

### 3.0.1 Introducción

Se usa herencia para crear objetos más especializados a partir de objetos existentes.

```
class Padre:
    ...

class Hijo(Padre):
    ...
```

Se dice que `Hijo` es una clase derivada o subclase. La clase `Padre` es conocida como la clase base, o superclase. La expresión `class Hijo(Padre):` significa que estamos creando una clase llamada `Hijo` que es derivada de la clase `Padre`.

### 3.0.2 Extensiones

Al usar herencia podemos tomar una clase existente y ...

- Agregarle métodos
- Redifinir métodos existentes
- Agregar nuevos atributos

Podemos verlo como una forma de **extender** un codigo existente. Darle nuevos comportamientos, abarcar un abanico más amplio de posibilidades ó aumentar su compatibilidad.

**Ejemplo:**

[ ]:
```
class Lote:
    def __init__(self, nombre, cajones, precio):
        self.nombre = nombre
        self.cajones = cajones
        self.precio = precio

    def costo(self):
        return self.cajones * self.precio
```

```
    def vender(self, ncajones):
        self.cajones -= ncajones
```

Podemos modificar lo que necesites mediante herencia.

### 3.0.3 Agregar un nuevo método

[ ]:
```
class MiLote(Lote):
    def rematar(self):
        self.vender(self.cajones)
```

[ ]:
```
c = MiLote('Pera', 100, 490.1)
```

[ ]:
```
c.vender(25)
```

[ ]:
```
c.cajones
```

[ ]:
```
75
```

[ ]:
```
c.rematar()
```

[ ]:
```
c.cajones
```

[ ]:
```
0
```

Esta clase heredó los atributos y métodos de `Lote` y la extendío con un nuevo método (`rematar()`).

### 3.0.4 Redefinir un método existente

[ ]:
```
class MiLote(Lote):
    def costo(self):
        return 1.25 * self.cajones * self.precio
```

[ ]:
```
c = MiLote('Pera', 100, 490.1)
```

[ ]:
```
c.costo()
```

[ ]:
```
61262.5
```

El método nuevo simplemente reemplaza al definido en la clase base. Los demás métodos y atributos no son afectados. ¿No es buenísimo?

### 3.0.5 Utilizar un método prevalente

Hay veces en que una clase extiende el método de la superclase a la que pertenece, pero necesita ejecutar el método original como parte de la redefinición del método nuevo. Para referirte a la superclase, usamos `super()`:

**Ejemplo:**

[ ]:
```
class Lote:
    ...
    def costo(self):
        return self.cajones * self.precio
    ...

class MiLote(Lote):
    def costo(self):
        # Fijate cómo usamos `super`
        costo_orig = super().costo()
        return 1.25 * costo_orig
```

Usamos `super()` para llamar al método de la clase base (del la cual ésta es heredera).

### 3.0.6 El método `__init__` y herencia.

Al crear cada instancia se ejecuta `__init__`. Ahí reside el código importante para la creación de una instancia nueva. Si redefinimos `__init__` siempre incluimos un llamado al método `__init__` de la clase base para inicializarla también.

**Ejemplo:**

[ ]:
```
class Lote:
    def __init__(self, nombre, cajones, precio):
        self.nombre = nombre
        self.cajones = cajones
        self.precio = precio
    ...


class MiLote(Lote):
    def __init__(self, nombre, cajones, precio, factor):
        # Fijate como es el llamado a `super().__init__()`
        super().__init__(nombre, cajones, precio)
        self.factor = factor

    def costo(self):
        return self.factor * super().costo()
```

Es necesario llamar al método `__init__()` en la clase base. Es una forma de ejecutar la versión previa del método que estamos redefiniendo.

### 3.0.7 Usos de herencia

Uno de los usos de definir una clase como heredera de otra es organizar jerárquicamente objetos que están relacionados.

Un ejemplo: Las figuras geométricas pueden tener ciertos métodos y atributos que luego son refinados en casos concretos como círculos o rectángulos.

**Ejemplo:**

[ ]:
```
import numpy as np
class Punto():
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def dist_origen(self):
        return np.sqrt(self.x**2 + self.y**2)

    def mover(self, dx, dy):
        self.x += dx
        self.y += dy

    def __str__(self):
        return f'({self.x}, {self.y})'

    def __repr__(self):
        return f'Punto({self.x}, {self.y})'

    def __add__(self, b):
        return Punto(self.x + b.x, self.y + b.y)

class Circulo(Punto):

    def __init__(self, x, y, r):
        self.x = x
        self.y = y
        self.r = r

    def radio(self):
        return self.r

    def area(self):
        return np.pi * self.r * self.r
```

[ ]:
```
c = Circulo(1,2,3)
print(c)
print(c.area())
c.mover(1,1)
print(c)
```

```
(1, 2)
28.274333882308138
(2, 3)
```

### 3.1 Retornar al índice