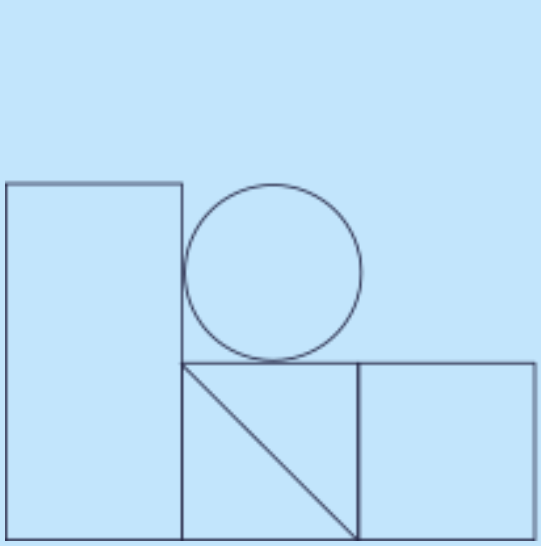
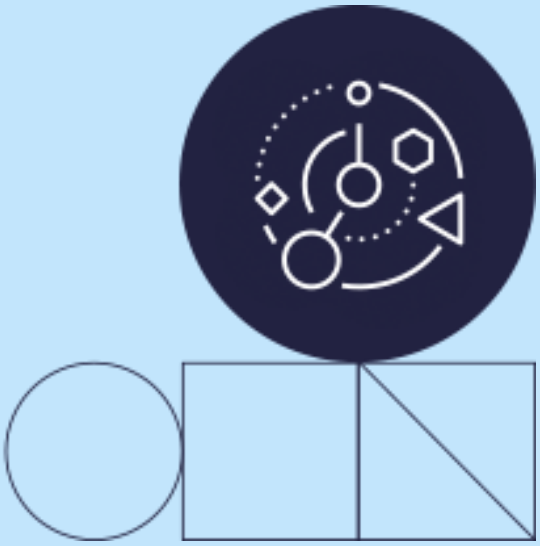
IBM SkillsBuild

IBM

POO con Python

¿Qué es la POO en Python?





Índice

Introducción

Como la mayoría de las actividades que hacemos a diario, la programación también tiene diferentes formas de realizarse. Estas formas de programar se llaman *paradigmas de programación* y entre las más importantes están la programación orientada a objetos (POO) y la programación estructurada.

A lo largo de la historia, han ido apareciendo diferentes paradigmas de programación. Lenguajes secuenciales como COBOL o procedimentales como Basic o C, se centraban más en la lógica que en los datos. Otros más modernos como Java, C# y Python, utilizan paradigmas para definir los programas, siendo la Programación Orientada a Objetos la más popular.

POO *vs.* programación estructurada

Con el paradigma de Programación Orientado a Objetos lo que buscamos es dejar de centrarnos en la lógica pura de los programas, para empezar a pensar en objetos, lo que constituye la base de este paradigma. Esto nos ayuda muchísimo en sistemas grandes, ya que, en vez de pensar en funciones, pensamos en las relaciones o interacciones de los diferentes componentes del sistema.

Un programador diseña un programa de software organizando piezas de información y comportamientos relacionados en una plantilla llamada clase. Luego, se crean objetos individuales a partir de la plantilla de clase. Todo el programa de software se ejecuta haciendo que varios objetos interactúen entre sí para crear un programa más grande.

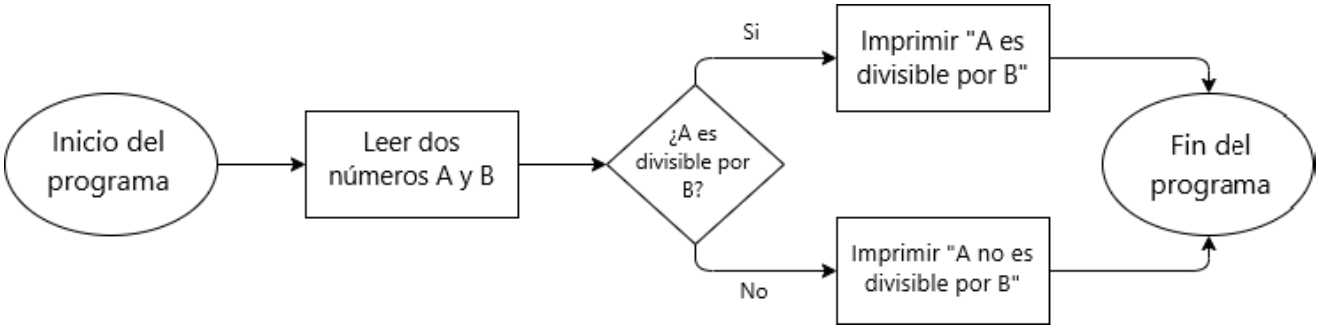
Cuando comenzamos a usar lenguajes como Java, C#, Python y otros que posibilitan el paradigma orientado a objetos, es común cometer errores y aplicar la programación estructurada pensando que estamos usando recursos de la orientación a objetos.

En la programación estructurada, un programa consta de tres tipos básicos de estructuras:

- **Secuencias**: son los comandos a ejecutar.
- **Condiciones**: secuencias que sólo deben ejecutarse si se cumple una condición (ejemplos: if...else, switch y comandos similares).
- **Repeticiones**: secuencias que deben realizarse repetidamente hasta que se cumpla una condición (for, while, do...while, etc.).

Estas estructuras se utilizan para procesar la entrada del programa, cambiando los datos hasta que se genera la salida esperada.

La principal diferencia entre este paradigma y la POO es que, en la programación estructurada, un programa generalmente se escribe en una sola rutina (o función) y, por supuesto, puede dividirse en subrutinas. Pero el flujo del programa sigue siendo el mismo, como si se pudiese copiar y pegar el código de las subrutinas directamente en las rutinas que las llaman, de tal forma que, al final, solo existiese una gran rutina que ejecute todo el programa.



Además, el acceso a las variables no tiene muchas restricciones en la programación estructurada. En lenguajes fuertemente basados en este paradigma, restringir el acceso a una variable se limita a decir si es visible o no dentro de una función (o módulo, como en el uso de la palabra clave static, en lenguaje C), pero no es posible decir de forma nativa que sólo se puede acceder a una variable mediante unas pocas rutinas del programa. El esquema para situaciones como estas implica prácticas de programación perjudiciales para el desarrollo del sistema, como el uso excesivo de variables globales. Vale la pena recordar que las variables globales se usan típicamente para mantener estados en el programa, marcando en qué parte de la ejecución se encuentran.

Los programas hechos con programación estructurada acaban siendo un solo código, en un solo archivo y con una extensión que dificulta enormemente su depuración y escalado. En principio las instrucciones se van ejecutando línea a línea, una tras otra, pero presentan numerosos saltos que nos remiten de una parte del código a otra, lo que lo complica aún más. Es lo que se denomina código spaguetti.

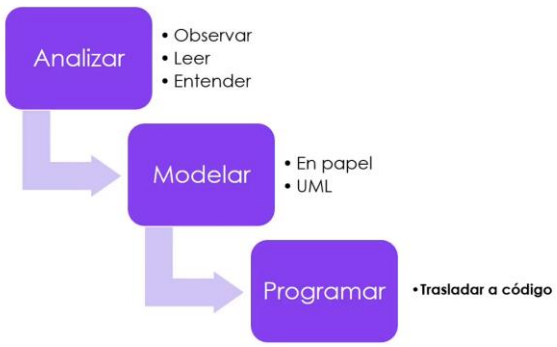
Eran programas complejos, con muchas líneas de código que sólo entendía correctamente aquel que lo había programado (y en ocasiones ni siquiera él), difícil de mantener y casi imposible de escalar. Además, en el caso de que la ejecución diese fallos, localizar dichos fallos era una tarea sumamente complicada.

La programación orientada a objetos surgió como una alternativa a estas características de la programación estructurada. El propósito de su creación fue acercar el manejo de las estructuras de un programa al manejo de las cosas en el mundo real, de ahí el nombre "objeto" como algo genérico, que puede representar cualquier cosa tangible.

Este nuevo paradigma se basa principalmente en dos conceptos clave: clases y objetos. Todos los demás conceptos, igualmente importantes, se basan en estos dos.

En la vida real tenemos objetos; una silla, una mesa, etc. Y dichos objetos tienen una serie de características y son capaces de desempeñar una serie de acciones. Un coche puede tener un color, un peso, un alto, etc, y puede arrancar, acelerar, girar, etc. Lo que se pretende con la programación orientada a objetos es asimilar la programación a lo que tenemos en la vida real, es decir, objetos con cualidades y capacidades, lo que en programación se conoce como atributos (cómo son) y métodos (qué pueden hacer).

La POO además nos ayuda a saber estructurar nuestro trabajo, a saber por dónde empezar:



Actualmente casi todos los lenguajes de programación de alto nivel se basan en mayor o menor medida en la POO. Algunos están parcialmente orientados a objetos, es decir, tienen programación orientada a objetos, pero también programación estructurada, y otros son totalmente orientados a objetos. En el caso de Python, es un lenguaje de programación totalmente orientado a objetos. Esto significa que todo en Python se considera un objeto; las variables, los diccionarios, tuplas, las funciones, los modificadores de flujo, los errores, etc. Python es 100% orientado a objetos. Todos los elementos de un programa de Python se consideran objetos y como objetos que son, tendrán propiedades y métodos. Esta es la base de la filosofía de Python.

Beneficios de Programación Orientada a Objetos

- **Reutilización** del código.
- Convierte cosas complejas en **estructuras simples reproducibles**.
- Evita la **duplicación de código**.
- Permite **trabajar en equipo** gracias al encapsulamiento ya que minimiza la posibilidad de duplicar funciones cuando varias personas trabajan sobre un mismo objeto al mismo tiempo.
- Al estar la clase bien estructurada permite la **corrección de errores** en varios lugares del código.
- **Protege la información** a través de la encapsulación, ya que solo se puede acceder a los datos del objeto a través de propiedades y métodos privados.
- La abstracción nos permite **construir sistemas más complejos** y de una forma más sencilla y organizada.

A continuación, pondremos dos ejemplos de script de Python que realizan la misma función. El primero está escrito con programación estructurada y el segundo con POO. De momento el alumno no dispone aún de los conocimientos necesarios para entender estos códigos. Se recomienda que, una vez terminado el temario se vuelva a este punto para poder, ya sí, apreciar las diferencias entre ambos códigos.

Ejemplo de programación estructurada

```
# Definimos unos cuantos clientes
clientes= [
    {'Nombre': 'Hector',
'Apellidos':'Costa Guzman',
'dni':'11111111A'},
    {'Nombre': 'Juan',
'Apellidos':'González Márquez',
'dni':'22222222B'}
]

# Creamos una función que muestra un
cliente en una lista a partir del DNI
def mostrar_cliente(clientes, dni):
    for c in clientes:
        if (dni == c['dni']):
            print('{}
{}'.format(c['Nombre'],c['Apellidos']))
            return
    print('Cliente no encontrado')
```

```
# Creamos una función que borra un
cliente en una lista a partir del DNI
def borrar_cliente(clientes, dni):
    for i,c in enumerate(clientes):
        if (dni == c['dni']):
            del( clientes[i] )
            print(str(c),"> BORRADO")
            return

    print('Cliente no encontrado')

### Fíjate muy bien cómo se utiliza el
código estructurado

print("==LISTADO DE CLIENTES==")
print(clientes)

print("\n==MOSTRAR CLIENTES POR DNI==")
mostrar_cliente(clientes, '11111111A')
mostrar_cliente(clientes, '11111111Z')

print("\n==BORRAR CLIENTES POR DNI==")
borrar_cliente(clientes, '22222222V')
borrar_cliente(clientes, '22222222B')

print("\n==LISTADO DE CLIENTES==")
print(clientes)
```

```
==LISTADO DE CLIENTES==
[{'Nombre': 'Hector', 'Apellidos':
'Costa Guzman', 'dni': '11111111A'},
{'Nombre': 'Juan', 'Apellidos':
'González Márquez', 'dni': '22222222B'}]

==MOSTRAR CLIENTES POR DNI==
Hector Costa Guzman
Cliente no encontrado

==BORRAR CLIENTES POR DNI==
Cliente no encontrado
{'Nombre': 'Juan', 'Apellidos':
'González Márquez', 'dni': '22222222B'}
> BORRADO

==LISTADO DE CLIENTES==
[{'Nombre': 'Hector', 'Apellidos':
'Costa Guzman', 'dni': '11111111A'}]
```

Ejemplo orientado a objetos

```
### No intentes entender este código,
sólo fíjate en cómo se utiliza abajo

# Creo una estructura para los clientes
class Cliente:

    def __init__(self, dni, nombre,
apellidos):
        self.dni = dni
        self.nombre = nombre
        self.apellidos = apellidos

    def __str__(self):
        return '{}
{}'.format(self.nombre,self.apellidos)

# Y otra para las empresas
class Empresa:

    def __init__(self, clientes=[]):
        self.clientes = clientes

    def mostrar_cliente(self, dni=None):
        for c in self.clientes:
            if c.dni == dni:
                print(c)
                return
        print("Cliente no encontrado")

    def borrar_cliente(self, dni=None):
        for i,c in
enumerate(self.clientes):
            if c.dni == dni:
                del(self.clientes[i])
                print(str(c),"> BORRADO")
                return
        print("Cliente no encontrado")

### Ahora utilizaré ambas estructuras

# Creo un par de clientes
hector = Cliente(nombre="Hector",
apellidos="Costa Guzman",
dni="11111111A")
juan = Cliente("22222222B", "Juan",
"Gonzalez Marquez")
```

```
# Creo una empresa con los clientes
iniciales
empresa = Empresa(clientes=[hector,
juan])

# Muestro todos los clientes
print("==LISTADO DE CLIENTES==")
print(empresa.clientes)

print("\n==MOSTRAR CLIENTES POR DNI==")
# Consulto clientes por DNI
empresa.mostrar_cliente("11111111A")
empresa.mostrar_cliente("11111111Z")

print("\n==BORRAR CLIENTES POR DNI==")
# Borro un cliente por DNI
empresa.borrar_cliente("22222222V")
empresa.borrar_cliente("22222222B")

# Muestro de nuevo todos los clientes
print("\n==LISTADO DE CLIENTES==")
print(empresa.clientes)
```

```
==LISTADO DE CLIENTES==
[<__main__.Cliente object at
0x0000023F567B42E8>,
<__main__.Cliente object at
0x0000023F567B4320>]

==MOSTRAR CLIENTES POR DNI==
Hector Costa Guzman
Cliente no encontrado

==BORRAR CLIENTES POR DNI==
Cliente no encontrado
Juan Gonzalez Marquez > BORRADO

==LISTADO DE CLIENTES==
[<__main__.Cliente object at
0x0000023F567B42E8>]
```

Como vemos, el código orientado a objetos es más autoexplicativo a la hora de utilizarlo.

Además, con programación estructurada tenemos que enviar la lista que queremos consultar todo el rato, mientras que con la POO tenemos esas "estructuras" como la empresa que contienen los clientes, todo queda más ordenado.

Principios de la Programación Orientada a Objetos

Todos los lenguajes de programación orientados a objetos tienen una serie de elementos comunes, estos son:

- Clase
- Objeto
- Atributo
- Método
- Abstracción
- Herencia
- Encapsulación
- Polimorfismo

A lo largo de los siguientes temas iremos explicando cada uno de estos conceptos detalladamente.