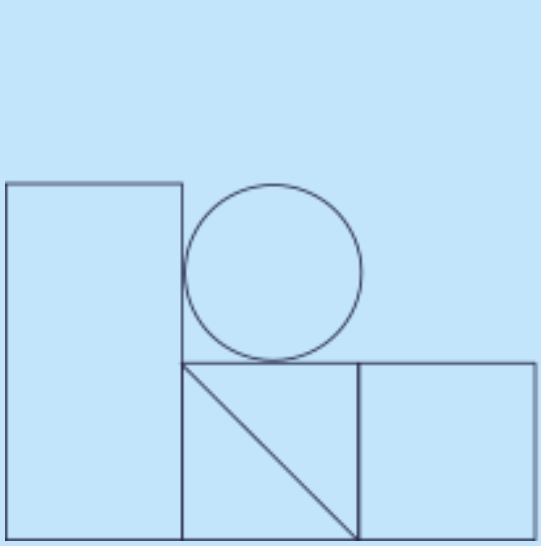
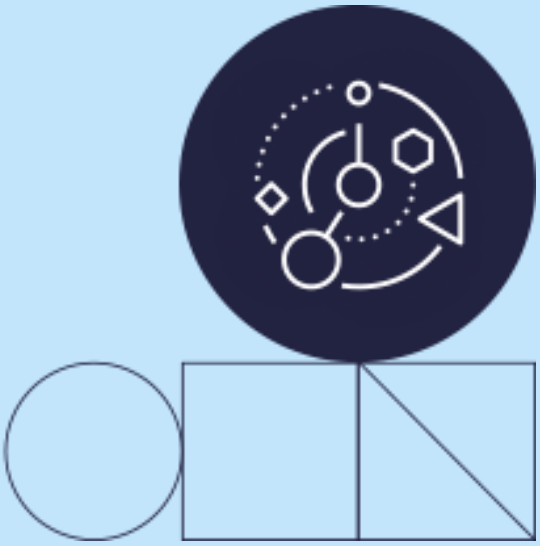
IBM SkillsBuild

IBM

# OOP com Python

## O que é OOP em Python?





# Índice

# Introdução

Como a maioria das atividades que realizamos diariamente, a programação também tem diferentes formas de ser feita. Estas formas de programação são chamadas de *paradigmas de programação* e entre as mais importantes estão a programação orientada a objetos (OOP) e a programação estruturada.

Ao longo da história, diferentes paradigmas de programação têm surgido. Linguagens seqüenciais como COBOL ou linguagens de procedimento como Basic ou C se concentraram mais na lógica do que nos dados. Linguagens mais modernas como Java, C# e Python usam paradigmas para definir programas, sendo a Programação Orientada a Objetos a mais popular.

# OOP Vs programação estruturada

Com o paradigma da Programação Orientada a Objetos, o que procuramos é deixar de nos concentrar na lógica pura dos programas e começar a pensar em objetos, que é a base deste paradigma. Isto nos ajuda muito em grandes sistemas, já que, em vez de pensarmos em funções, pensamos nas relações ou interações dos diferentes componentes do sistema.

Um programador projeta um programa de software organizando partes de informação e comportamentos relacionados em um modelo chamado classe. Então, objetos individuais são criados a partir do modelo de classe. Todo o programa de software é executado fazendo com que vários objetos interajam uns com os outros para criar um programa maior.

Quando começamos a usar linguagens como Java, C#, Python e outras que permitem o paradigma orientado a objetos, é comum cometermos erros e aplicarmos o pensamento de programação estruturada de que estamos usando recursos orientados a objetos.

Na programação estruturada, um programa consiste em três tipos básicos de estruturas:

- **Seqüências**: são os comandos a serem executados.
- **Condições**: seqüências que só devem ser executadas se uma condição for cumprida (exemplos: if...else, switch e comandos similares).
- **Repetições**: seqüências que devem ser realizadas repetidamente até que uma condição seja cumprida (for, while, do...while, etc.).

Estas estruturas são usadas para processar a entrada do programa, alterando os dados até que a saída esperada seja gerada.

A principal diferença entre este paradigma e o OOP é que, na programação estruturada, um programa é geralmente escrito em uma única rotina (ou função) e pode, naturalmente, ser dividido em sub-rotinas. Mas o fluxo do programa permanece o mesmo, como se você pudesse copiar e colar o código das sub-rotinas diretamente nas rotinas que as chamam, de modo que, no final, há apenas uma grande rotina que executa o programa inteiro.

Além disso, o acesso às variáveis não é muito restrito na programação estruturada. Em idiomas fortemente baseados neste paradigma, restringir o acesso a uma variável se limita a dizer se ela é visível ou não dentro de uma função (ou módulo, como no uso da palavra-chave static em C), mas não é possível dizer nativamente que uma variável só pode ser acessada por algumas rotinas de programa. O esquema para situações como estas envolve práticas de programação prejudiciais ao desenvolvimento do sistema, tais como o uso excessivo de variáveis globais. Vale lembrar que variáveis globais são tipicamente utilizadas para manter estados no programa, marcando onde eles estão na execução.

Programas feitos com programação estruturada acabam como um único código, em um único arquivo e com uma extensão que torna a depuração e a escala muito difícil. Em princípio, as instruções são executadas linha por linha, uma após a outra, mas há inúmeros saltos que nos levam de uma parte do código para outra, o que o torna ainda mais complicado. Isto é chamado de código spaghetti.

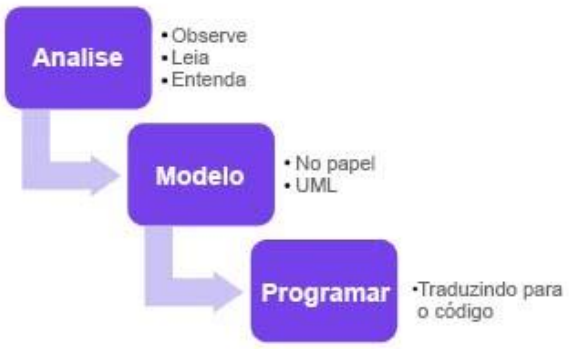

Eram programas complexos, com muitas linhas de código que somente o programador (e às vezes nem mesmo o programador) entendia corretamente, difíceis de manter e quase impossíveis de serem escalados. Além disso, no caso de falhas de execução, a localização dessas falhas foi uma tarefa extremamente complicada.

A programação orientada a objetos surgiu como uma alternativa a estas características da programação estruturada. O objetivo de sua criação foi aproximar o manuseio das estruturas de um programa do manuseio das coisas no mundo real, daí o nome "objeto" como algo genérico, que pode representar qualquer coisa tangível.

Este novo paradigma se baseia principalmente em dois conceitos-chave: classes e objetos. Todos os outros conceitos, igualmente importantes, se baseiam nestes dois.

Na vida real, temos objetos; uma cadeira, uma mesa... etc. E esses objetos têm uma série de características e são capazes de realizar uma série de ações. Um carro pode ter uma cor, um peso, uma altura... etc., e pode arrancar, acelerar, virar... etc. O objetivo da programação orientada a objetos é assimilar a programação ao que temos na vida real, ou seja, objetos com qualidades e capacidades, que na programação são conhecidos como atributos (como eles são) e métodos (o que eles podem fazer).

O OOP também nos ajuda a saber como estruturar nosso trabalho, a saber por onde começar:




Atualmente, quase todas as linguagens de programação de alto nível são, em maior ou menor grau, baseadas no OOP. Alguns são parcialmente orientados a objetos, ou seja, têm programação orientada a objetos, mas também programação estruturada, e outros são totalmente orientados a objetos. No caso do Python, trata-se de uma linguagem de programação totalmente orientada a objetos. Isto significa que tudo em Python é considerado um objeto; variáveis, dicionários, tuplos, funções, modificadores de fluxo, erros... etc. Python é 100% orientado a objetos. Todos os elementos de um programa Python são considerados objetos e, como objetos, terão propriedades e métodos. Esta é a base da filosofia de Python.

# Benefícios da Programação Orientada a Objetos

- **Reutilização** do código.
- Ele transforma coisas complexas em **estruturas simples e reprodutíveis**.
- Evita a **duplicação do código**.
- Permite **trabalhando em equipe** graças ao encapsulamento, pois ele minimiza a possibilidade de duplicação de funções quando várias pessoas estão trabalhando no mesmo objeto ao mesmo tempo.
- Por estar la classe bem estruturada permite a **correção de erros em vários lugares no código**.
- **Protege as informações** através de encapsulamento, pois os dados do objeto só podem ser acessados através de propriedades e métodos privados.
- A abstração nos permite **construir sistemas mais complexos** e de uma forma mais simples e organizada.

Aqui estão dois exemplos de scripts Python que desempenham a mesma função. O primeiro é escrito com programação estruturada e o segundo com OOP. No momento, o estudante ainda não possui os conhecimentos necessários para compreender estes códigos. Recomenda-se que, uma vez concluído o programa, o estudante retorne a este ponto para poder apreciar as diferenças entre os dois códigos.

## Exemplo de programação estruturada

```
# 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'}]
```

## Exemplo 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 podemos ver, o código orientado a objetos é mais auto-explicativo de se usar. Além disso, com programação estruturada temos que enviar a lista que queremos consultar o tempo todo, enquanto que com OOP temos estas "estruturas" como a empresa que contém os clientes, tudo é mais ordenado.

## Princípios da Programação Orientada a Objetos

Todas as linguagens de programação orientadas a objetos têm uma série de elementos em comum, estes são:

- Classe
- Objeto
- Atributo
- Método
- Abstração
- Herança
- Encapsulamento
- Polimorfismo

Nos tópicos a seguir, explicaremos cada um desses conceitos em detalhes.