

# Programação em Python

---

<https://advancedinstitute.ai>



# Documentação de Código Python

---

*Docstrings*

## Referências e Fontes das Imagens

- ❑ [Google Python Style Guide](#)
- ❑ [Docstrings Styles](#)
- ❑ [The Hitchhiker's Guide to Python: Best Practices for Development](#) (Book)

## Docstrings

- ❑ Importante escrever a documentação da API que resuma claramente o que **cada objeto e método faz**.
- ❑ Manter a documentação atualizada é difícil;
- ❑ A melhor maneira de fazer isso é escrever **direto em nosso código**;
- ❑ Cada classe, função ou cabeçalho de método pode ter uma *string* na primeira linha após a definição;
- ❑ Essa linha deve estar no mesmo nível de indentação do conteúdo interno à definição;

```
1 >>> class Point:
2     ...     'Essa classe representa um ponto no espaço bi-dimensional
           utilizando coordenadas geométricas'
```

## Docstrings - Visualizando a documentação

- A função `help()` e a variável especial `__doc__`

```
1 >>> help(Point)
2 Help on class Point in module __main__:
3
4 class Point(builtins.object)
5 |     Essa classe representa um ponto no espaço bi-dimensional utilizando
6 |     coordenadas geométricas
7
8 >>> Point.__doc__
9 'Essa classe representa um ponto no espaço bi-dimensional utilizando
10  coordenadas geométricas'
```

## Docstrings - Comentários de Múltiplas Linhas

□ Vários estilos, e.g., [Estilo do Google](#)

```
1  def move(self, x, y):
2      """Altera a posição do ponto no espaço
3
4      Verifica se a nova posição é diferente da atual. Caso seja, move o
5          ponto para nova posição
6
7      Args:
8          x (float): Coordenada X.
9          y (float): Coordenada y.
10
11      Returns:
12          bool: Um Booleano informando se o movimento foi feito ou não
13      """
```



# Programação Python

---

*Herança e Polimorfismo*

## Referências e Fontes das Imagens

- ❑ Object-Oriented Programming and Python Tutorial
- ❑ Inheritance and Composition: A Python OOP Guide
- ❑ Python 3 Object Oriented Programming (Book)
- ❑ Learning Python: Powerful Object-Oriented Programming (Book)



## Herança

- ❑ **Toda classe que criamos já usa herança**
- ❑ Todas as classes são subclasses da classe especial chamada `object`
- ❑ Relação de Super-Sub Classe;
- ❑ Mecanismo de Reuso de Código;
- ❑ Definição de Comportamentos e Estados compartilhados entre instâncias;
- ❑ Requer uma **quantidade mínima de sintaxe extra** sobre uma definição de classe básica;
- ❑ O uso mais simples e óbvio de herança é **adicionar funcionalidade a uma classe existente**.
- ❑ Em geral, as classes podem **herdar, personalizar ou estender** o código existente em *superclasses*;

## Herança

### □ Princípio de Substituição de Liskov

- *Em um programa de computador, se  $S$  é um subtipo de  $T$ , então objetos do tipo  $T$  podem ser substituídos por objetos do tipo  $S$  sem alterar nenhuma das propriedades desejadas do programa.*

### □ E.g., uma classe `Eletronico` possui `TV` como subclasse

- A classe `Eletronico` possui três operações: `ligar()`, `desligar()` e `esta_ligado()`
- Instâncias da classe `TV` herdam as funcionalidade da classe `Eletronico`, evitando a duplicidade de código;

## Importante

O ponto principal por trás de OOP em geral: programamos personalizando (especializando) o que já foi feito, em vez de copiar ou alterar o código existente;

## Herança

```
1 In [2]: class Eletronico:
2     ...:     def __init__(self):
3     ...:         self.ligado = False
4     ...:     def esta_ligado(self):
5     ...:         return self.ligado
6     ...:     def ligar(self):
7     ...:         if not self.esta_ligado():
8     ...:             self.ligado = True
9     ...:     def desligar(self):
10    ...:         if self.esta_ligado():
11    ...:             self.ligado = False
12
13 In [3]: class TV(Eletronico):
14    ...:     def mudar_canal(self, canal):
15    ...:         self.canal = canal
```

## Herança

```
1 In [4]: e1 = Eletronico()
2 In [5]: tv1 = TV()
3 In [7]: tv1.ligar()
4 In [8]: tv1.esta_ligado()
5 Out[8]: True
6 In [9]: tv1.mudar_canal(10)
7 In [10]: tv1.canal
8 Out[10]: 10
9 In [11]: e1.mudar_canal(5)
10 -----
11 AttributeError                                Traceback (most recent call last)
12 <ipython-input-11-4028bef25be5> in <module>
13 ----> 1 e1.mudar_canal(5)
14 AttributeError: 'Eletronico' object has no attribute 'mudar_canal'
```

## Estendendo métodos

- Acrescentar novas funcionalidades a um método definido na superclasse;
  - Função especial `super()` que faz referência a superclasse;

```
1 In [12]: class FormaGeometrica:
2         ...:     def __init__(self, x, y):
3         ...:         self.x, self.y = x, y
4
5 In [13]: class Retangulo(FormaGeometrica):
6         ...:     def __init__(self, x, y, largura, altura):
7         ...:         super().__init__(x, y)
8         ...:         self.largura, self.altura = largura, altura
9
10 In [14]: ret = Retangulo(10, 20, 5, 5)
11 In [15]: ret.x, ret.y
12 Out[15]: (10, 20)
```

## Sobrescrevendo Métodos

```
1  class FormaGeometrica:
2      def desenha(self):
3          print(f"Iniciando Desenho no ponto ({self.x}, {self.y})")
4
5  class Retangulo(FormaGeometrica):
6      def desenha(self):
7          print(f"Desenhando Retangulo com início no ponto ({self.x}, {self.y}), com L x A: {self.largura} x {self.altura}")
8
9  >>> ret1 = Retangulo(10, 20, 5, 5)
10 >>> ret1.desenha()
11 'Desenhando Retangulo com início no ponto (10, 20), com L x A: 5 x 5'
```

## Polimorfismo

- ❑ Comportamentos diferentes para mesma operação dependendo de qual subclasse é usada para instanciar o objeto.
- ❑ **Métodos com mesmos nomes** em classes e subclasses;
- ❑ Uma das razões principais para usar herança;
  - *Python duck typing*
    - O tipo do objeto é determinado por seu comportamento e estado;
- ❑ Fornecimento de uma **única interface** para entidades de diferentes tipos;

## Polimorfismo

```
1 >>> f1 = FormaGeometrica(0,0)
2 >>> for forma in (f1, ret1):
3     ...     forma.desenhe()
4
5 'Iniciando Desenho no ponto (0, 0)'
6 'Desenhando Retangulo com início no ponto (100, 200), com L x A: 10 x 10'
```



## Encapsulamento

- ❑ Esconder detalhes de implementação de quem vai utilizar uma dada classe
- ❑ Variáveis “*privadas*” em Python:

```
1  class Ponto(FormaGeometrica):
2      def __init__(self, x=0, y=0):
3          self.__x, self.__y = x, y
4  >>> p1 = Ponto()
5  >>> p1.desenhe()
6  'Ponto localizado em (0, 0)'
7  >>> p1.__x
8  -----
9      -
10  AttributeError
11  ...
11  AttributeError: 'Ponto' object has no attribute '__x'
```

Dúvidas?