# Programação Orientada a Objetos

Classe, objetos, namespace, sombreamento de atributos e self.

Parte 3

# Programação Orientada a Objetos

## Um pouco mais sobre classes e objetos:

```
class Classe_simples(): # Quando vazios, as chaves são opcionais, class Classe_simples:
    pass

print(type(Classe_simples)) # Qual o tipo do objeto? => <class 'type'>

obj = Classe_simples() # obj é uma instância da classe simples
print(type(obj)) # <class '__main__.Classe_simples'>

print(type(obj) == Classe_simples) # True, obj é do tipo Classe_simples
```

# Programação Orientada a Objetos

## Um pouco mais sobre classes e objetos:

```
class Pessoa():
    especie = 'Humano'

print(Pessoa.especie) # Humano
Pessoa.vivo = True # Adicionado dinamicamente
print(Pessoa.vivo) # True
homem = Pessoa()
print(homem.especie) # Humano (herdado)
print(homem.vivo) # True (herdado)
Pessoa.vivo = False
print(homem.vivo) # False (herdado)
homem.nome = 'Evaldo'
homem.sobrenome = 'Wolkers'
print(homem.nome, homem.sobrenome) # Evaldo Wolkers
```

# Programação Orientada a Objetos

**Escopo e espaço de nome (Namespace):**

Namespace é um espaço ou região dentro do programa, onde um nome (seja uma variável, uma função, etc.) é considerado válido.

Depois que o objeto de classe é criado, ele basicamente representa um namespace. Podemos chamar essa classe para criar suas instâncias. Cada instância herda os atributos e métodos da classe e recebe seu próprio namespace.

# Programação Orientada a Objetos

**Escopo e espaço de nome (Namespace):**

No Python temos basicamente 3 escopos:

1 – Escopo local: que contém nomes locais (função atual).

2 – Escopo global: escopo do módulo que contém nomes globais, acessado por todas funções do módulo.

3 – Built-in names (nomes embutidos): que é o namespace que contém as funções built-in do Python (funções padrões como abs(), cmp(), etc.) e built-in exception names (usados para tratar erros específicos nas cláusulas *except* de blocos *try*).

# Programação Orientada a Objetos

## Escopo e espaço de nome (Namespace):

```
def funcao_externa():
    b = 20
    a = 80
    print(f"Imprimindo 'b' em funcao_externa: {b}")
    print(f"Imprimindo 'a' em funcao_externa: {a}")
    def funcao_interna():
        c = 30
        b = 25
        a = 70
        print(f"Imprimindo 'c' em funcao_interna: {c}")
        print(f"Imprimindo 'b' em funcao_interna: {b}")
        print(f"Imprimindo 'a' em funcao_interna: {a}")

    funcao_interna()
    print(f"Imprimindo 'b' de novo em funcao_externa: {b}")

a = 10

print(f"Imprimindo 'a' no escopo global: {a}")
funcao_externa()
print(f"Imprimindo 'a' de novo no escopo global: {a}")
```

A variável “a” está no namespace global.
A variável “b” está no namespace local da função “funcao_externa”.
A variável “c” está no namespace local da função “funcao_interna”.
Quando estamos na funcao_interna, “c” é local, “a” é global e “b” é nonlocal. Se tentarmos atribuir um valor a “b”, será criada uma variável local para funcao_interna diferente da b nonlocal.
A mesma coisa acontece ao atribuirmos um valor a variável global “a”.

```
Imprimindo 'a' no escopo global: 10
Imprimindo 'b' em funcao_externa: 20
Imprimindo 'a' em funcao_externa: 80
Imprimindo 'c' em funcao_interna: 30
Imprimindo 'b' em funcao_interna: 25
Imprimindo 'a' em funcao_interna: 70
Imprimindo 'b' de novo em funcao_externa: 20
Imprimindo 'a' de novo no escopo global: 10
```

# Programação Orientada a Objetos

## Escopo e espaço de nome (Namespace):

```
def funcao_externa():
    b = 20
    global a
    a = 80
    print(f"Imprimindo 'b' em funcao_externa: {b}")
    print(f"Imprimindo 'a' em funcao_externa: {a}")
    def funcao_interna():
        c = 30
        b = 25
        global a
        a = 70
        print(f"Imprimindo 'c' em funcao_interna: {c}")
        print(f"Imprimindo 'b' em funcao_interna: {b}")
        print(f"Imprimindo 'a' em funcao_interna: {a}")

    funcao_interna()
    print(f"Imprimindo 'b' de novo em funcao_externa: {b}")

a = 10

print(f"Imprimindo 'a' no escopo global: {a}")
funcao_externa()
print(f"Imprimindo 'a' de novo no escopo global: {a}")
```

Ao definir a variável “a” dentro das funções especificando “*global*” será referenciada a variável “a” do escopo global, ao alterar seu valor, será alterado o valor da variável “a” global.

```
Imprimindo 'a' no escopo global: 10
Imprimindo 'b' em funcao_externa: 20
Imprimindo 'a' em funcao_externa: 80
Imprimindo 'c' em funcao_interna: 30
Imprimindo 'b' em funcao_interna: 25
Imprimindo 'a' em funcao_interna: 70
Imprimindo 'b' de novo em funcao_externa: 20
Imprimindo 'a' de novo no escopo global: 70
```

# Programação Orientada a Objetos

**Sombreamento de atributos:**

Quando um atributo em um objeto não é encontrado, o Python continua buscando na classe que foi usada para criar esse objeto (e continua pesquisando até que seja encontrado ou o fim da cadeia de herança seja alcançado). Isso leva a um comportamento de sombreamento.

# Programação Orientada a Objetos

## Sombreamento de atributos:

```
class Ponto():
    x = 10
    y = 7

p = Ponto()

print(p.x) # 10 (do atributo da classe)
print(p.y) # 7 (do atributo da classe)
p.x = 12 # p obtém seu próprio atributo "x"
print(p.x) # 12 (encontrado na instância)
print(Ponto.x) # 10 (O atributo da classe ainda é o mesmo)
del p.x # Apagando o atributo da instância
print(p.x) # 10 (Agoa que não existe "x" na instância, será retornado da classe)
p.z = 3
print(p.z) # 3
print(Ponto.z) # O objeto Ponto não tem o atributo "z"
# AttributeError: type object 'Ponto' has no attribute 'z'
```

**O que é o *self*?**

Dentro de um método de classe, podemos nos referir a uma instância por meio de um argumento especial, chamado *self* por convenção. *Self* é sempre o primeiro atributo de um método de instância.

# Programação Orientada a Objetos

## O que é o *self*?

```
class Quadrado():
    lados = 8
    def area(self): # self é uma referencia a uma instância
        return self.lados ** 2

quadrado = Quadrado()
print(quadrado.area()) # 64 ('lados' foi encontrado na classe)
print(Quadrado.area(quadrado)) # 64 (equivalente a quadrado.area())
quadrado.lados = 10
print(quadrado.area()) # 100 ('lados' foi encontrado na instância)
```

## O que é o *self*?

```
class Calculo():
    def calcular_total(self, quantidade, desconto):
        return (self.preco * quantidade - desconto)

calc = Calculo()
calc.preco = 15

print(calc.calcular_total(15, 10))
print(Calculo.calcular_total(calc, 15, 10))
```

# CONTINUA...