

Live de Python #62

Programação orientada a objetos #2

Ajude a Live de Python

apoia.se/livedepython

picPay: @livedepython

Roteiro

- cls x self
- Atributos
 - Classe
 - Instância
- Métodos
 - Classe
 - Instância
 - Estáticos
- Herança
- Polimorfismo

self x cls

Usamos **self** sempre que queremos falar com o
exemplo (instância)

x

Usamos **cls** quando queremos falar com a
classe

```
class Fila:
    c_filas = []

    @classmethod
    def c_entrar(cls, obj):
        cls.c_filas.append(obj)
        print(cls.c_filas)

    def __init__(self):
        self.s_filas = []

    def s_entrar(self, obj):
        self.s_filas.append(obj)
        print(self.s_filas)
```

1

Criação de uma abstração de
dado.

No caso uma **Fila**

```
class Fila:
    c_fila = []

    @classmethod
    def c_entrar(cls, obj):
        cls.c_fila.append(obj)
        print(cls.c_fila)

def __init__(self):
    self.s_fila = []

def s_entrar(self, obj):
    self.s_fila.append(obj)
    print(self.s_fila)
```

**Manipulação da abstração de
dado**

**Manipulação do exemplo, ou
instância, representada pelo tipo
de dado**

```
class Fila:
    c_filas = []

    @classmethod
    def c_entrar(cls, obj):
        cls.c_filas.append(obj)
        print(cls.c_filas)
```

2

Um atributo da classe.

```
class Fila:
    c_fila = []

    @classmethod
    def entrar(cls, obj):
        cls.c_fila.append(obj)
        return cls.c_fila
```

2

2

Um atributo da classe.

Característica em comum que será mantida, ainda se alterada, junto com os exemplos


```
class Fila:
    c_fila = []

    @classmethod
    def c_entrar(cls, obj):
        cls.c_fila.append(obj)
        print(cls.c_fila)
```

3

3

Método que manipula um atributo de classe.

Vamos ver isso depois.

Recebe **CLS**, pois a referência é da própria classe, o decorador **@classmethod** faz isso, deixa explícito que é um método de classe

```
def __init__(self):  
    self.s_filas = []
```

```
def s_entrar(self, obj):  
    self.s_filas.append(obj)  
    print(self.s_filas)
```

4

4

Método inicializador da classe,
NÃO É CONSTRUTOR.

Inicia o exemplo, ou instância. Com isso podemos tratar dinamicamente os atributos das instâncias.

```
def __init__(self):  
    self.s_fila = []  
  
def s_entrar(self, obj):  
    self.s_fila.append(obj)  
    print(self.s_fila)
```

4

Método inicializador da classe,
NÃO É CONSTRUTOR.

Trabalha com **SELF**, pois está trabalhando com o exemplo da classe

```
def __init__(self):  
    self.s_fila = []  
  
def s_entrar(self, obj):  
    self.s_fila.append(obj)  
    print(self.s_fila)
```

4

4

Método inicializador da classe,
NÃO É CONSTRUTOR.

```
In [1]: f = Fila()
```

```
In [2]: f.s_entrar('Eduardo')  
['Eduardo']
```

```
In [3]: f.s_entrar('João')  
['Eduardo', 'João']
```

5

Atributo da instância, que nascerá e morrerá com ela.

```
def __init__(self):  
    self.s_fila = []
```

```
def s_entrar(self, obj):  
    self.s_fila.append(obj)  
    print(self.s_fila)
```

5

6

Método que manipula um atributo de instância

```
class Fila:
    c_fila = []

    @classmethod
    def c_entrar(cls, obj):
        cls.c_fila.append(obj)
        print(cls.c_fila)

def __init__(self):
    self.s_fila = []

def s_entrar(self, obj):
    self.s_fila.append(obj)
    print(self.s_fila)
```

**Manipulação da abstração de
dado**

**Manipulação do exemplo, ou
instância, representada pelo tipo
de dado**

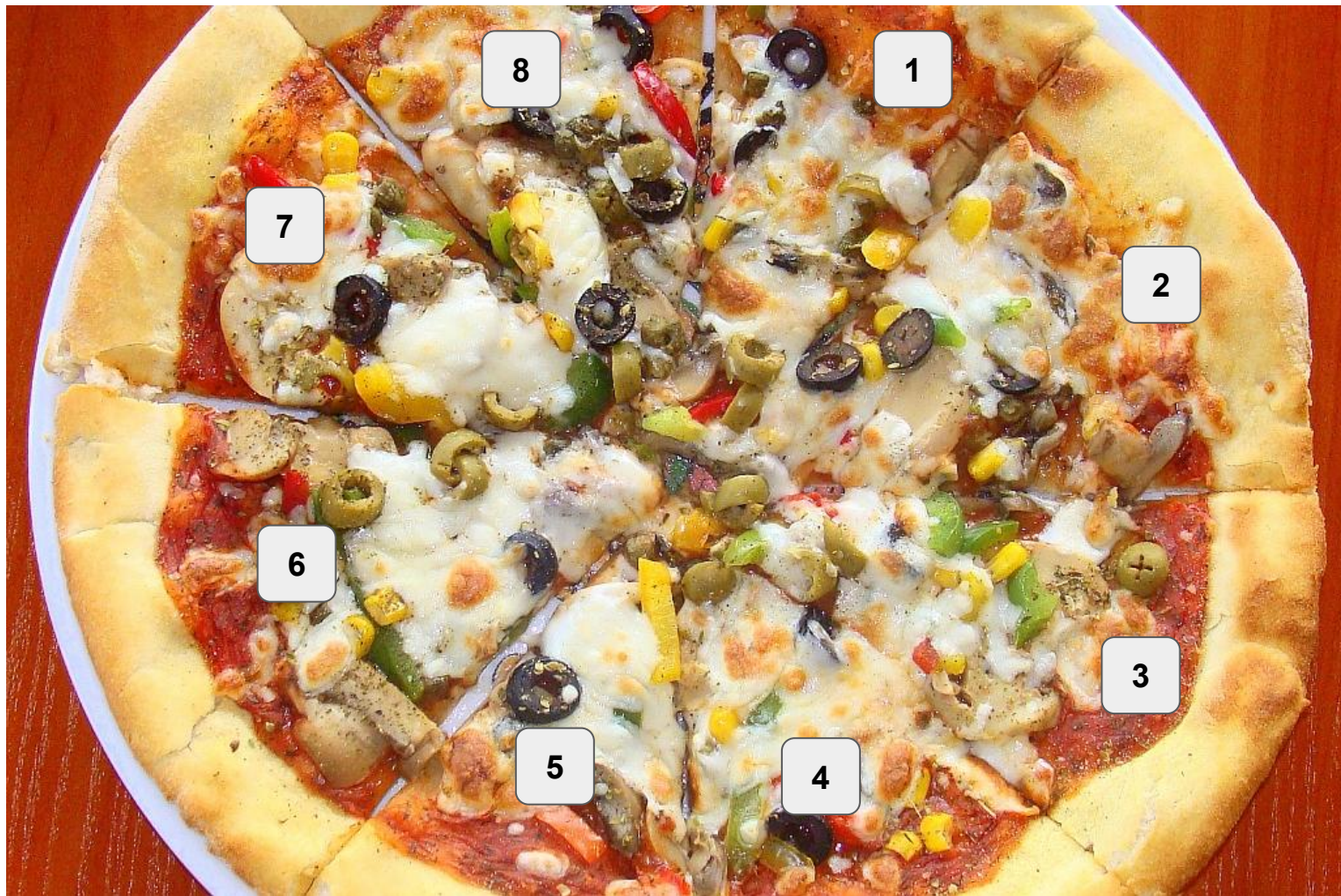
Vamos exemplificar
isso

Métodos de fato

[Julien Danjou](#)

Tipos de métodos

- Métodos de instância:
 - Só funcionam com a classe instanciada
 - Manipulam atributos da instância
- Métodos de classe:
 - Funcionam a todo momento, até mesmo na instância
 - Manipulam atributos de classes
- Métodos estáticos:
 - Funcionam a todo momento
 - Não interagem com atributos
- Métodos abstratos (Assunto pra outra hora):
 - Dizem a subclasse o que ela deve implementar



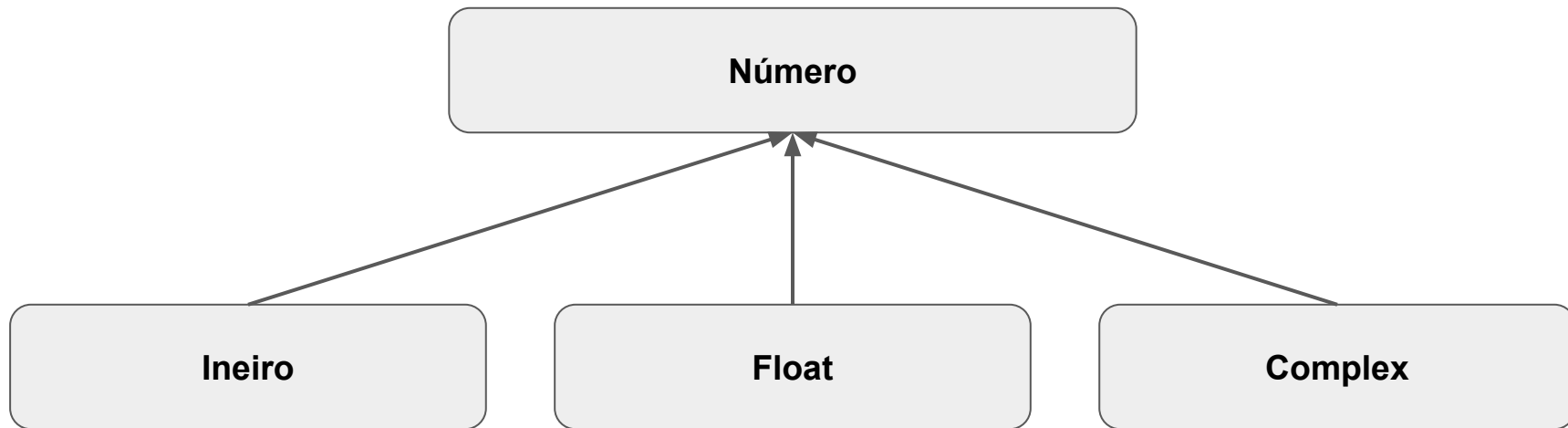
Vamos pensar em pizza

- Todas as pizza (NORMAIS) são de 8 pedaços. Ou seja, isso é indiferente ao nosso exemplo de uma pizza. Logo isso pode ser um atributo de classe.
- Já a quantidade de pedaços disponíveis são referentes a nossa instância. Pois da pizza “real” eu posso pegar um pedaço.
- Os ingredientes da pizza não fazem referência a nenhum momento, vocês não concordam? Eu posso pensar em queijo e molho de tomate sem pensar na pizza de fato.

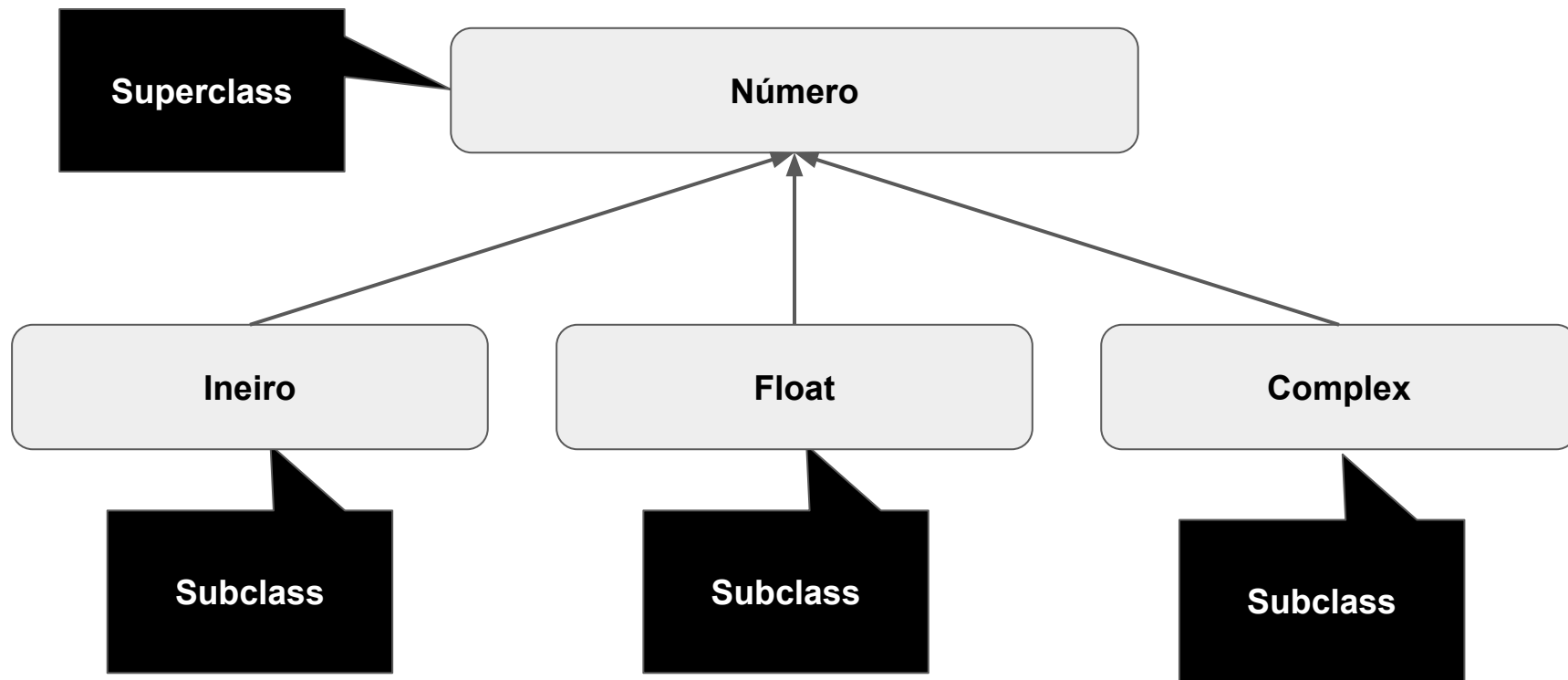
Vamos “cozgrammar”

Herança

Exemplificar antes de explicar



Exemplificar antes de explicar



É mesmo?

```
In [1]: from numbers import Number
```

```
In [2]: isinstance(int, Number)
```

```
Out[2]: True
```

```
In [3]: isinstance(float, Number)
```

```
Out[3]: True
```

```
In [4]: isinstance(complex, Number)
```

```
Out[4]: True
```


Herança

Segundo Sebesta, herança é um concepção para resolver dois problemas:

- O reuso de tipos abstratos de dados
 - Eles eram definidos para resolver um problema muito específico. Depois de usado em um contexto, mesmo um tipo abstrato ficava sem uso durante todo o resto do programa
 - Geralmente quem usava os tipos, por abstração não sabiam exatamente como tipo era implementado
- Todos os tipos têm a mesma hierarquia e são independentes
 - Em muitos problemas do mundo real, entidades era muito parecidas, quase “irmãs” e não havia um mecanismo para unificar coisas parecidas
 - Era praticamente impossível ter que construir objetos quase iguais, todas as vezes

Herança

A herança surge como uma solução para os dois tipos de problema.

Se um tipo de dados abstratos puder herdar a abstração de tipo e as abstrações de operações de um tipo já existente e também for permitido mudar/adicionar pequenas coisas a reutilização será facilitada e a classe inicial não precisará ser modificada

Herança

Então, com isso, a herança permite com que

- Tipos de tipos abstratos possam ser construídos para solucionar problemas
 - Com isso atender novos requisitos
- Criar hierarquia de tipos
 - Tipos co-dependentes
- **Reutilização de código**

Herdando pizzas

Herança

```
class Pizza:
    pedaços = 8

    @classmethod
    def mudar_tamanho(cls, pedaços):
        cls.pedaços = pedaços

class Mussarela(Pizza):
    ...
```

Pizza representa a abstração total de uma pizza.

Ou seja, sabores diferentes não diferem de ser uma pizza.

Herança

```
class Mussarela(Pizza):  
    ...  
  
class Calabresa(Pizza):  
    ...  
  
class QuatroQueijos(Pizza):  
    ...  
  
class FrangoComRqueijão(Pizza):  
    ...
```

Pizza representa a abstração total de uma pizza.

Ou seja, sabores diferentes não diferem de ser uma pizza.

Herança

Com isso podemos construir pizzas de múltiplos sabores também

```
class Mussarela(Pizza):  
    ...  
  
class Calabresa(Pizza):  
    ...  
  
class MeioAMeio(Pizza, Mussarela, Calabresa):  
    ...
```

Herança

Com isso podemos construir pizzas de múltiplos sabores também

```
class Mussarela(Pizza):
```

```
...
```

```
class Calabresa(Pizza):
```

```
...
```

```
class MeioAMeio(Pizza, Mussarela, Calabresa):
```

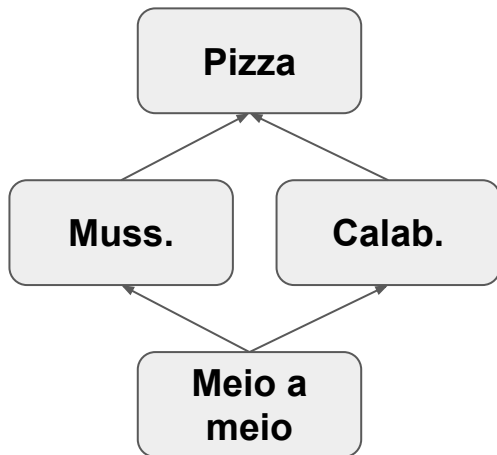
```
...
```

Herança Múltipla

Herança

Pizza já é herdado por associação

```
class MeioAMeio(Pizza, Mussarela, Calabresa):  
    ...
```

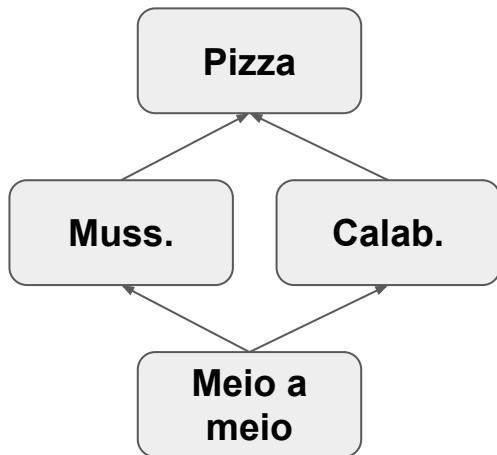


```
class MeioAMeio(Mussarela, Calabresa):  
    ...
```

Herança

Pizza já é herdado por associação

```
class MeioAMeio(Mussarela, Calabresa):  
    ...
```



```
In [1]: m = MeioAMeio()
```

```
In [2]: isinstance(m, Pizza)
```

```
Out[2]: True
```

```
In [3]: isinstance(m, Mussarela)
```

```
Out[3]: True
```

Polimorfismo

Polimorfismo

Vamos pensar que temos um método, ou abstração de processo, que nos mostre quais são os ingredientes de uma pizza.

Vamos pensar por um momento:

- Todas as pizza tem ingredientes
- Todas as pizza tem ingredientes diferentes (se não seriam a mesma pizza)


Porém, o método *ingredientes* tem que mudar em todas as pizza.

Polimorfismo

O nome dado a esse tipo de comportamento, “sobrescrever” um método de uma classe é **‘Polimorfismo’**

```
class Pizza:  
    def ingredientes(self):  
        return 'Ingredientes'
```

```
class Mussarela(Pizza):  
    def ingredientes(self):  
        return ['quejo mussarela',  
                'molho de tomate',  
                'oregano']
```



A diagram illustrating inheritance. A vertical arrow points upwards from the `Mussarela` class box to the `Pizza` class box, indicating that `Mussarela` inherits from `Pizza`.

Polimorfismo

Porém existe um problema nessa implementação. O criador da subclasse não é “obrigado” a sobrescrever esse método. Ele pode usar a implementação original. Ou seja, Você pode fazer uma subclasse que retorna “ingredientes”.

Para isso existem as metaclasses, MMMMMMMMAAAAASSSSSSSSSS

