Object orientation

Diretrizes gerais sobre projetos em python

Criando uma classe

A definição é muito parecida com a do Java (mas sem frescura). Eis um exemplo:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Onde o método init() é o nosso construtor . Outro exemplo é:

```
class Dog:
    # Class attribute
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Instanciando objetos:

Se uma classe tiver argumentos no seu construtor e seu objeto for declarado sem argumentos acontece o seguinte:

```
>>> Dog()
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
```

```
    Dog()
TypeError: __init__() missing 2 required positional arguments: 'name' and 'age'
```

O correto seria:

```
>>> buddy = Dog("Buddy", 9)
>>> miles = Dog("Miles", 4)
```

e os atributos de um objeto podem ser acessados pela notação de ponto, do mesmo jeito que no Java:

```
>>> buddy.age = 10
>>> buddy.age
10

>>> miles.species = "Felis silvestris"
>>> miles.species
'Felis silvestris'
```

OBS: objetos personalizados são mutáveis(editáveis) por natureza, pois um objeto é mutável se ele puder ser alterado dinamicamente. listas e dicionários são mutáveis, strings e tuplas não são.

Instanciando métodos

continuando com nosso exemplo do cachorro:

```
class Dog:
    # Class attribute
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

Após compilar, as funções são executadas da seguinte forma:

```
>>> miles = Dog("Miles", 4)

>>> miles.description()
'Miles is 4 years old'

>>> miles.speak("Woof Woof")
'Miles says Woof Woof'

>>> miles.speak("Bow Wow")
'Miles says Bow Wow'
```

Entretanto, podemos trocar a função description() para algo melhor:

```
class Dog:
    # Class attribute
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"

    # Replace .description() with __str__()
    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

que resulta em:

```
>>> miles = Dog("Miles", 4)
>>> print(miles)
'Miles is 4 years old'
```

funções como a init() e str() são chamadas de **dunder methods** pelo formato que são escritas (com 2 pares de "_").

Herança

Ao relacionar algumas classes, criamos uma **hierarquia**, onde existem **classes pai** e **classes filho**. Geralmente existem 2 tipos de situações quando se trata de herança:
Override: É quando você sobrepõe um método ou atributo da classe pai;
Extend: É quando você cria atributos ou funções adicionais a uma classe pai.

Continuando com o exemplo anterior, vamos agora criar classes com raças de cachorro, que claramente vão herdar atributos da classe *Dog*:

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def speak(self, sound):
        return f"{self.name} says {sound}

class JackRussellTerrier(Dog):
    pass

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass
```

Até o dado momento, todas as classes filho herdam todos os atributos e métodos da classe pai, por exemplo:

```
>>> miles = JackRussellTerrier("Miles", 4)
>>> buddy = Dachshund("Buddy", 9)
>>> jack = Bulldog("Jack", 3)
>>> jim = Bulldog("Jim", 5)

>>> miles.species
'Canis familiaris'

>>> buddy.name
'Buddy'

>>> print(jack)
Jack is 3 years old

>>> jim.speak("Woof")
'Jim says Woof'
```

Extend e Override

Primeiro vamos mostrar um exemplo de override:

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def speak(self, sound):
        return f"{self.name} says {sound}

class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return f"{self.name} says {sound}"

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass
```

Agora qualquer cachorro da raça *JackRusselTerrier* tem sua própria função *speak()*. A função também possui um argumento padrão na parte da String, então agora não é mais necessário digitar o som que o cachorro faz, mas ainda é possível colocar uma string personalizada. Exemplo:

```
>>> miles = JackRussellTerrier("Miles", 4)
>>> miles.speak()
'Miles says Arf'
>>> miles.speak("Grrr")
'Miles says Grrr'
```

Se alterarmos a classe *Dog* na sua função *speak()* todas as outras classes recebem alteração, com exceção das classes que dão override (no nosso caso, seria a subclasse *JackRusselTerrier*). A classe *Dog* ficaria assim:

```
class Dog:
    # deixa todo o resto igual

    # Troca o print da função .speak()
    def speak(self, sound):
        return f"{self.name} barks: {sound}"
```

e ao chamarmos essa função no nosso objeto *miles* continuamos com a definição feita na subclasse, enquanto todas as outras subclasses herdam a função *speak()* da Classe *Dog*:

```
>>> jim = Bulldog("Jim", 5)
>>> jim.speak("Woof")
'Jim barks: Woof'

>>> miles = JackRussellTerrier("Miles", 4)
>>> miles.speak()
'Miles says Arf'
```

Podemos resolver (pois não é o comportamento esperado pra este caso) usando um método chamado *super()* que é responsável por procurar a classe pai associada a uma subclasse:

```
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return super().speak(sound)
```