pythondrops.com

Curso Básico de Python

Lição 6: Objetos com "classe"

Sumário

OOP

Orientação a objetos é uma forma de modularizar seu programa, muito popular hoje em dia. Se você não conhece OOP (Object-oriented programming) em nenhuma outra linguagem, é melhor estudar o assunto antes de continuar:

https://www.devmedia.com.br/principais-conceitos-da-programacao-orientada-a-objetos/32285

Se você já possui experiência com OOP em outras linguagens, como Java ou C#, pode ter algumas surpresas. Em Python, a declaração e instanciamento de classes é bastante diferente.

Objetos

Geralmente falando, em Python, todos os tipos de dados são classes, portanto, suas variáveis são objetos:

```
>>> a = 1
>>> type(a)
<class 'int'>
>>> b = True
>>> type(b)
<class 'bool'>
>>> c = 'Teste'
>>> type(c)
<class 'str'>
>>> print(a.bit_length())
1
>>> print(a.to_bytes(2, byteorder='big'))
b'\x00\x01'
```

Como pode ver, os tipos das variáveis "a", "b" e "c" são classes. E as variáveis são objetos, possuindo métodos, como pode ver nos dois últimos comandos.

O que acontece quando atribuímos um valor a um objeto? Se for um objeto imutável, o python verificará se já existe aquele objeto em memória e, neste caso, apenas atribui o mesmo endereço à variável:

```
>>> nome="fulano"
>>> id(nome)
140098638964640
>>> nome2="fulano"
>>> id(nome2)
```

```
140098638964640
```

A função "id()" retorna o endereço de um objeto na memória. Note que o endereço das duas variáveis é o mesmo, pois ambas apontam para o mesmo objeto imutável. Se criarmos outro objeto string, este endereço muda:

```
>>> nome3="beltrano"
>>> id(nome3)
140098669530800
```

Bom, se o objeto é o mesmo e se duas variáveis apontam para o mesmo objeto, é de se esperar que, ao alterarmos o objeto, ambas reflitam o mesmo valor, certo?

```
>>> nome="fulano"
>>> id(nome)
140098638964640
>>> nome2="fulano"
>>> id(nome2)
140098638964640

>>> nome="cicrano"
>>> nome2
'fulano'
```

Algo errado? Era de se esperar que, ao alterarmos o objeto apontado por ***nome***, a variável ***nome2*** fosse afetada. Na verdade, strings são objetos imutáveis, logo, se alterarmos o conteúdo de uma variável string, estamos, na verdade, criando outro objeto e apontando para este. Se havia outra variável apontando para o objeto original, não será afetada.

Isto também acontece com inteiros:

```
>>> numero = 5
>>> outro_numero = numero
>>> id(numero)
94711138497440
>>> id(outro_numero)
94711138497440
>>> numero = 6
>>> outro_numero
5
>>> id(numero)
94711138497472
>>> id(outro_numero)
94711138497440
```

Com outros tipos de variáveis, como números reais, este comportamento pode ser diferente:

```
>>> numero_real = 50.05
>>> id(numero_real)
140098670363296
>>> outro_ainda = 50.05
>>> id(outro_ainda)
140098670362792
>>> a = 5
>>> b = 5
>>> id(a)
94711138497440
>>> id(b)
94711138497440
```

Note que o comportamento com objetos que são números reais é diferente do comportamento com objetos que são inteiros.

Este comportamento de objetos imutáveis é diferente para cada implementação de Python (CPython, PyPy ou Jython).

Com objetos mútaveis, podemos criar cópias independentes. Por exemplo, vamos supor uma ***list***:

```
>>> numeros = [1,2,3,4,5]
>>> outros_numeros = [1,2,3,4,5]
>>> id(numeros)
140098669581512
>>> id(outros_numeros)
140098638878472
```

Note que, apesar do valor atribuído ser o mesmo, dois objetos diferentes foram criados na memória. E o que acontece se atribuirmos outra variável a um objeto mutável?

```
>>> numeros = [1,2,3,4,5]
>>> id(numeros)
140098669581512
>>> lista1 = numeros
>>> id(lista1)
140098669581512

>>> numeros[-1]=10
>>> lista1
[1, 2, 3, 4, 10]
```

Criamos uma variável ***lista1***, atribuindo a ela o objeto ***numeros***. O que o Python fez? Usou o mesmo objeto! Tanto é verdade, que, ao alterarmos o último elemento da lista ***numeros***, o último elemento da lista ***lista1*** também foi alterado! Quando o objeto é mutável, o Python não faz uma cópia, apenas adiciona um "apelido" para o mesmo objeto!

Lembre-se sempre disto!

Classes

Podemos criar nossas próprias classes (ou tipos de dados), e instanciarmos objetos a partir delas. Por exemplo:

```
>>> class MinhaClasse:
...     pass
...
>>> var1 = MinhaClasse()
>>> type(var1)
<class '__main__.MinhaClasse'>
```

A sintaxe é simples:

```
class <nome da classe>:
      <variáveis da classe>
      <métodos>
```

Você sabe a diferença entre classe e instância, certo? Sabe o que são variáveis de classe e variáveis de instância, certo? Vou mostrar uma coisa:

```
>>> class Carro:
...     eixos=2
...
>>> fusca = Carro()
>>> fusca.marca = "Volkswagen"
>>> fusca.eixos
2
>>> fusca.marca
'Volkswagen'
>>> focus = Carro()
>>> focus.marca = "Ford"
>>> focus.eixos
2
>>> focus.marca
'Ford'
>>> Carro.eixos
2
```

Toda variável declarada dentro de uma classe é considerada como variável de classe (ou "static" em Java). No exemplo, a variável ***eixos*** é da classe ***Carro***, portanto, só existe uma única cópia dela em memória.

As variáveis atribuídas aos objetos individuais (***fusca***, ***focus***) são variáveis de instância, e cada objeto tem seu próprio valor para elas. Na verdade, objetos diferentes da mesma classe, podem possuir propriedades com nomes diferentes.

Eu atribuí a nova propriedade marca aos dois objetos que criei, com valores diferentes. Cada objeto possui sua marca, mas a classe Carro não possui marca:

```
>>> fusca.marca
'Volkswagen'
>>> focus.marca
'Ford'
>>> Carro.marca
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Carro' has no attribute 'marca'
```

Como já deve ter visto, instanciamos classes atribuindo uma variável ao nome da classe:

```
>>> fusca = Carro()
>>> id(fusca)
140098638965896
>>> focus = Carro()
>>> id(focus)
140098638965672
>>> id(Carro)
94711172186904
>>> volks = fusca
>>> id(volks)
140098638965896
```

Não há necessidade de utilizar um operador ***new***, como em Java. Se estivermos instanciando um objeto mutável, uma nova instância será criada na memória. Mas, se estivermos atribuindo um objeto já existente, então estamos apenas criando um novo "apelido" para ele, que é o caso das variáveis ***fusca*** e ***volks***. Note que a própria classe ***Carro*** tem seu endereço de memória separado.

Posso estar sendo chato e redundante, mas é importantíssimo que você entenda como funciona atribuição de objetos em Python e quais as diferenças para outras linguagens de programação, como Java, por exemplo.

Cópias

E se quisermos criar uma cópia de um objeto em memória?

```
>>> fusca = Carro()
```

```
>>> id(fusca)
140098638965896
>>> volks = fusca
>>> id(volks)
140098638965896
```

Neste exemplo, a variável ***volks*** aponta para o mesmo objeto da variável ***fusca***. E se eu quisesse criar uma cópia, ou seja um novo objeto, mas com os mesmos valores do ***fusca***? Uma opção é importar o módulo ***copy*** e utilizar o seu método ***copy***:

```
>>> fusca.marca = "Volkswagen"
>>> volks = copy.copy(fusca)
>>> id(fusca)
140098638965896
>>> id(volks)
140098638965952
>>> fusca.marca
'Volkswagen'
>>> volks.marca
'Volkswagen'
>>> fusca.marca = "New beetle"
>>> volks.marca
'Volkswagen'
```

Note que eu fiz uma cópia do objeto, tanto é que posso alterar a propriedade ***marca*** do ***fusca***, que o objeto ***volks*** permanecerá sem alterações.

Shallow e Deep copy

Cara, eu disse que Python tem suas idiossincrasias, e não estou brincando! Há dois tipos de cópias de objetos: ***Shallow*** e ***Deep***. O método ***copy.copy()*** executa uma shallow copy. Isso é melhor demonstrado com objetos que possuem outros objetos, como listas:

```
>>> lista1 = [[1,2,3],[4,5,6]]
>>> lista2 = copy.copy(lista1)
>>> id(lista1)
140098639081800
>>> id(lista2)
140098639081992
>>> lista2[0][0]=33
>>> lista2
[[33, 2, 3], [4, 5, 6]]
>>> lista1
[[33, 2, 3], [4, 5, 6]]
```

Ué? Eu usei o método ***copy.copy()*** e criei dois objetos diferentes! Os ***ids*** são diferentes! Como é possível eu alterar a ***lista2*** e a ***lista1*** ser afetada?

O método ***copy.copy()*** executa uma shallow copy (cópia rasa), procurando reutilizar os objetos internos de uma lista. Se quisermos realmente criar uma cópia profunda, temos que usar o método ***copy.deepcopy()***:

```
>>> lista1 = [[1,2,3],[4,5,6]]
>>> lista2 = copy.deepcopy(lista1)
>>> lista2[0][0]=33
>>> lista2
[[33, 2, 3], [4, 5, 6]]
>>> lista1
[[1, 2, 3], [4, 5, 6]]
```

Variáveis de instância

Se quisermos criar uma classe com variáveis de instância, precisamos de um método construtor. Isto é feito com o método ***__init__*** (script "carro.py"):

```
class Carro:
    eixos = 2
    def __init__(self,marca=None):
        self.marca = marca
```

E podemos importar e instanciar este tipo:

```
>>> from carro import Carro
>>> volks = Carro("Volkswagen")
>>> print('Carro: {} eixos {}'.format(volks.marca, volks.eixos))
Carro: Volkswagen eixos 2
```

O método ***__init__*** é invocado sempre que instanciamos uma classe. Note que ele possui um primeiro parâmetro posicional, tradicionamente nomeado ***self***, que é uma referência ao próprio objeto que está sendo criado/acessado. Usando esta referência, podemos atribuir variáveis a esta instância. Assim, criamos variáveis de instância em classes Python.

O parâmetro ***self*** é declarado no método, mas não é passado quando instanciamos o objeto! O Python passa automaticamente! O que acontece se não especificarmos o parâmetro self, ao declaramos o método? Bom, se ele contiver um parâmetro, receberá automaticamente a referência ao objeto. Caso contrário, ele será considerado um método da classe Carro:

```
class Carro:
    eixos = 2
    def __init__(self,marca=None):
        self.marca = marca
    def metodo():
        print('Chamou')
```

```
>>> from carro import Carro
>>> volks = Carro()
>>> volks = Carro('volkswagen')
>>> volks.metodo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: metodo() takes 0 positional arguments but 1 was given
>>> Carro.metodo()
Chamou
```

Note que lambança! Se tentarmos invocar o método "metodo()" a partir do objeto, ele reclama que está faltando o parâmetro ***self***. Mas podemos invoca-lo a partir da classe.

Métodos de instância

Um método de instância pertence à instância da classe e pode acessar as variáveis de instância usando o parâmetro ***self:***

```
class Carro:
    eixos = 2
    def __init__(self,marca=None):
        self.marca = marca
    def dict(self):
        return {'marca':self.marca}
```

É um método que possui o perimeiro argumento, que pode ou não ser chamado ***self***, mas que sempre receberá o objeto a partir do qual o método foi invocado. Ele pode acessar as variáveis de instância. Eis o resultado:

```
>>> from carro_1 import Carro
>>> var1 = Carro("Ford")
>>> type(var1.dict())
<class 'dict'>
>>> print(var1.dict())
{'marca': 'Ford'}
```

O método dict retorna um ***dictionary*** com a propriedade ***marca*** da instância.

Métodos estáticos e de classe

O Python faz distinção entre métodos estáticos e métodos de classe. Nós os declaramos utilizando um ***decorator*** (uma declaração iniciada por "@"):

```
class Carro:
    eixos = 2
    def __init__(self,marca=None):
        self.marca = marca
    @staticmethod
    def comentario():
        return "Esta é a classe carro."
    @classmethod
    def tipoVeiculo(cls):
        return "Este veículo tem {} eixos".format(cls.eixos)
```

O decorator @staticmethod declara um método estático, independente, que não tem acesso a nada da classe ou da instância. O decorator @classmethod recebe um parâmetro para a classe e pode ter acesso a qualquer variável ou método desta classe.

Controle de acesso

Python não tem um controle de acesso muito forte, como Java (*default*, private, protected e public). A princípio, todas as variáveis e métodos são públicos, ou seja: Podem ser acessados a partir de qualquer código.

Há uma convenção em Python para tratar variáveis e métodos como privados: Devem começar com um único caractere sublinha ("_"):

```
class Carro:
    eixos = 2
    def __init__(self,marca=None, chassi=None):
        self._chassi = chassi
        self.marca = marca
    def _mostrar(self):
        return self._chassi
```

Vamos ver como utilizar esta classe:

```
>>> from carro_3 import Carro
>>> volks = Carro("Volkswagen","DABB01")
>>> volks.marca
'Volkswagen'
>>> volks._chassi
'DABB01'
>>> volks._mostrar()
'DABB01'
```

A variável ***_chassi*** e o método ***_mostrar()*** não fazem parte da API pública da classe, logo, não deveriam ser acessada por código externo. Nós os acessamos sem nenhuma limitação. Como eu

disse, é apenas uma convenção, ou seja, você não deveria utilizar membros cujo nome inicia por sublinha, pois não fazem parte da API pública da classe.

Quando pensamos em variáveis de classe privadas, podemos ter problemas, pois é preciso separar as variáveis de cada subclasse. Uma maneira de esconder variáveis de classe privadas é prefixando-as com dois caracteres sublinha ("__"):

```
class Carro:
    eixos = 2
    __quantidade = 0
    def __init__(self,marca=None):
        Carro._Carro__quantidade+=1
        self.marca = marca
    @classmethod
    def contagem(cls):
        return "existem {} veiculos".format(cls._Carro__quantidade)
```

Se prefixarmos um membro com dois caracteres sublinha, então a variável é renomeada como:

```
_<nome da classe>__<nome da variável>
```

Veja só como utilizamos:

```
>>> from carro_4 import Carro
>>> fusca = Carro("Volkswagen")
>>> Carro.contagem()
'existem 1 veiculos'
>>> fusca.__quantidade
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Carro' object has no attribute '__quantidade'
>>> Carro.__quantidade
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Carro' has no attribute '__quantidade'
>>> Carro._Carro__quantidade
1
```

Continuamos a poder acessar a variável, se a prefixarmos com _Carro. O objetivo não é esconder a variável, mas evitar conflitos de nomes com subclasses.

Herança e polimorfismo

Como toda linguagem orientada a objetos, Python oferece esses dois mecanismos. Vamos ver o script "veiculo.py":

```
class Veiculo:
    rodas = 4
    __contagem = 0
    def __init__(self,marca=None):
        self.marca = marca
    @classmethod
    def quemSou(cls):
        return(cls)
    def mostrar(self):
        return {'marca': self.marca}

class Carro(Veiculo):
    pass
```

A classe Veiculo tem variáveis de classe públicas, privadas, métodos de classe etc. A classe Carro herda de Veiculo. São herdados: Construtor, variáveis públicas de classe, variáveis privadas de instância e os métodos.

```
>>> from veiculo import Veiculo, Carro
>>> v = Veiculo("Volkswagen")
>>> c = Carro("fusca")
>>> c.quemSou()
<class 'veiculo.Carro'>
>>> c._Veiculo__contagem
0
>>> c._Carro__contagem
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Carro' object has no attribute '_Carro__contagem'
```

Veja que a classe ***Carro*** não declarou um construtor, porém, ele existe, pois foi herdado. A princípio, tudo o que você declarar na classe derivada, não será herdado da classe original. Como ***Carro*** não tem um ***__init__*** ele herda o da classe ***Veiculo*** (junto com quaisquer variáveis que instancie com ***self***, dentro do ***__init__***).

Carro não tem uma variável privada de classe chamada ***__contagem***. Mas ele herda a de ***Veiculo***. Vamos modificar um pouco para mostrar como ficaria:

```
class Veiculo:
    rodas = 4
    __contagem = 100
```

```
    def __init__(self,marca=None):
        self.marca = marca
    @classmethod
    def quemSou(cls):
        return(cls)
    def mostrar(self):
        return {'marca': self.marca}

class Carro(Veiculo):
    __contagem = 50
```

Veja só agora:

```
>>> from veiculo import Veiculo, Carro
>>> c = Carro("fusca")
>>> v = Veiculo("Volkswagen")
>>> c._Carro__contagem
50
>>> v._Veiculo__contagem
100
```

Confuso, não? Mas o que vai te confundir mesmo é a Herança múltipla. Vou mostrar apenas um pequeno exemplo:

```
class Veiculo():
    rodas = 4
    def __init__(self, marca=None):
        self.marca = marca

class Automotor():
    cilindradas = 1000

class Carro(Veiculo,Automotor):
    def mostrar(self):
        return {'marca':self.marca, 'cilindradas':self.cilindradas}
```

Como pode ver, a classe ***Carro*** deriva de duas classes bases: ***Veiculo*** e ***Automotor***, portanto herda membros das duas:

```
>>> from veiculo2 import *
>>> c = Carro()
>>> c.mostrar()
{'marca': None, 'cilindradas': 1000}
```

Usei a sintaxe * para importar tudo do módulo veiculo2, e a classe carro herdou as propriedades ***marca*** e ***cilindradas***.

Polimorfismo

É a capacidade de diferenciar a invocação de métodos de acordo com a classe especifica do objeto. Vamos supor este código:

```
class Veiculo():
    def ligar(self):
        raise NotImplementedError("A subclasse deve implementar este método")

class Carro(Veiculo):
    def ligar(self):
        return('ligando o carro')

class Moto(Veiculo):
    def ligar(self):
        return('ligando a moto')
```

Temos uma estrutura de classes aqui. Note que a classe base, ***Veiculo***, levanta um erro, caso invoquemos o método ***ligar***, pois ela não sabe que tipo de veículo ela é, portanto, não sabe como liga-lo. Ao executar, temos este comportamento:

```
>>> from poly import *
>>> v = Veiculo()
>>> c = Carro()
>>> m = Moto()
>>> v.ligar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/cleuton/Documents/projetos/python/curso/licao6/poly.py", line 3,
in ligar
    raise NotImplementedError("A subclasse deve implementar este método")
NotImplementedError: A subclasse deve implementar este método
>>> m.ligar()
'ligando a moto'
>>> c.ligar()
'ligando o carro'
```

Se tentarmos invocar o método ligar a partir de um objeto ***Veiculo***, o erro acontecerá. E cada objeto invoca o seu próprio método ligar, independentemente da classe base.

Classes abstratas e interfaces

Python não possui um mecanismo para isto, mas oferece um módulo, chamado ***ABC***, que permite definir ***Abstract Base Classes*** ou classes básicas abstratas.

Uma classe abstrata ou mesmo uma interface, servem para criarmos um modelo para classes derivadas, que devem oferecer os métodos espeficiados. Por exemplo, uma classe derivada de veículo deve, obrigatóriamente, oferecer um método ligar. Em Java ou C# podemos fazer isto através de classes abstratas ou interfaces. Mas Python não tem esse recurso.

Podemos criar uma classe abstrata, com métodos abstratos, utilizando o módulo ABC:

```
from abc import ABC, abstractmethod
class Veiculo(ABC):
    @abstractmethod
    def ligar(self):
        pass

class Carro(Veiculo):
    pass
```

A classe ***Veiculo*** herda de ***ABC*** (abstract base class) e possui um método marcado com o decorator ***@abstractmethod***, portanto, não pode ser instanciada. Ela serve de base para criarmos outras classes, como a classe ***Carro***.

Só que a classe ***Carro*** não declarou o método abstrato ligar, portanto, também não pode ser instanciada:

```
>>> from veiculo3 import *
>>> c = Carro()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Carro with abstract methods ligar
>>> v = Veiculo()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Veiculo with abstract methods ligar
```

Dunders

Eu já falei sobre ***dunders*** (nomes que ficam entre dois caracteres sublinha, como ***__init__***) e o Python tem vários métodos dunders para você declarar em sua classe.

Abra o programa exemplo do curso, maze/model/labirinto.py. Na classe Labirinto há dois métodos dunders:

```
def __init__(self,linhas=10,colunas=10):
...
def __str__(self):
...
```

O ***__init__*** já conhecemos: Ele faz o papel de Construtor de instâncias, e o ***__str__*** retorna uma representação informal em string do objeto, funcionando aproximadamente como o método ***Object.toString()***, do Java.

O programa ***maze/maze.py*** simplesmente passa uma instância de ***Labirinto*** para o ***print*** e ela é impressa direitinho, porque o método ***__str__*** é invocado, de modo a retornar uma representação string para o ***print*** mostrar:

```
from model.labirinto import Labirinto
from solver import Solver

labirinto = Labirinto(5,50)
print(labirinto)

solver = Solver()
solver.solve(labirinto)
print(labirinto)
```

Existem vários métodos dunder (ou mágicos) que você pode declarar em sua classe:

__eq__, __lt__ : Comparam instâncias da classe. O primeiro verifica igualdade e o segundo, se a instância é menor que outra. Por exemplo:

```
class Carro:
    def __init__(self,marca=None,comprimento=0.0):
        self.marca = marca
        self.comprimento = comprimento
    def __eq__(self,other):
        return self.marca==other.marca
    def __lt__(self,other):
        return self.comprimento<other.comprimento
```

Exemplo:

```
>>> from dunders import Carro
>>> a = Carro("Volkswagen",3.50)
>>> b = Carro("Ford",4.10)
>>> a==b
False
>>> c = a
>>> a==c
True
>>> a<b
True
>>> b<a
False
```

Aqui você vai achar vários tipos de dunder methods:

https://dbader.org/blog/python-dunder-methods

O que não falei...

Bom, o curso básico é esse. Acabou... c'est fini!

Muita coisa ficou de fora, por exemplo: Generators, List comprehension, Meta class / Meta programming, WSGI etc.

A ideia era te ensinhar o básico do Python, com muitos exemplos e acredito que este objetivo foi alcançado. Prossiga estudando e visite sempre o site http://pythondrops.com para ver os tutoriais e extras.

Boa sorte em sua carreira,

Cleuton Sampaio, M.Sc.