Server-Side – POO

Católica de Santa Catarina – Centro Universitário
Jaraguá do Sul - SC, Brasil

Professor: Ph.D. Andrei Carniel (Prof. Andrei)
Contato: andrei.carniel@gmail.com
linktr.ee/andrei.carniel

CatólicaSC
Centro Universitário

Classes

Contexto

- Classes estão diretamente ligadas à programação orientada à objetos, que nada mais é do que uma abstração de como lidamos com certos tipos de problemas em nosso código.
- Criando e usando objetos de uma forma mais complexa, a partir de estruturas “molde”.
- Uma classe é um objeto que ficará reservado ao sistema tanto para indexação quanto para uso de sua estrutura.
- É um molde de onde criaremos uma série de objetos.
- Para o Python, toda variável é um objeto. Objetos baseados em classes são como variáveis compostas.
- A forma como instanciamos um objeto faz com que o interpretador o trate com mais ou menos privilégios.

Resumindo

Podemos dizer então que uma classe é uma estrutura lógica modelável e reutilizável

Classes

- Uma classe fará com que uma variável se torne uma categoria reservada ao sistema para que possamos atribuir dados, valores e parâmetros de maior complexidade.

- É como se transformássemos uma variável em uma super variável de maior possibilidade de recursos e uso.

- A classe então servirá de molde para a criação de outras variáveis compostas (objetos).

- Lembre-se da atribuição do tipo de variável.

Definindo uma classe

- Para tal, existe uma sintaxe adequada que fará com que o interpretador entenda a estrutura criada como sendo uma classe.

- Quando temos uma função dentro de uma classe ela é chamada de método dessa classe.

- Quando definimos uma classe começamos criando um construtor para ela, isto é, uma função que ficará reservada para o sistema e será chamada sempre que uma instância dessa classe for criada para que o interpretador crie a instância corretamente.

Sintaxe

```
class Usuario:
        def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

        def boas_vindas(self):
                print(f'Usuário: {self.nome}, Idade: {self.idade}')

usuario1 = Usuario(nome= 'Andrei', idade='23')
usuario1.boas_vindas()
print(usuario1.nome)
```

Entendendo...

- Palavra **class** diz ao interpretador que ali começa uma classe.
- Pela sintaxe, o nome da classe deve começar com letra maiúscula.
- A palavra __**init**__ (com dois “_” em cada lado) define o construtor, que terá self como parâmetro (instância padrão), seguido dos parâmetros que criarmos, separados por vírgula.
- Métodos seguem o padrão das funções, porém é necessário adicionar **self** como primeiro parâmetro.
- Ao executar a linha usuario1 = Usuario(nome= 'Andrei ', idade='23'), o interpretador irá criar o objeto na memória e preencher os atributos dele, de acordo com o que está descrito no construtor da classe.

Alterando dados/valores de uma instância

- Feito por atribuição direta

```
usuario1 = Usuario(nome='Andrei', idade='22')
usuario1.nome = 'Andrei Carniel'
```

- Ou via funções

```
usuario1 = Usuario(nome='Andrei', idade='22')
setattr(usuario1, 'nome', 'Andrei Carniel')
delattr(usuario1, 'idade')
setattr(usuario1, 'idade', '22')
```

Ou ainda via métodos

- Getter & Setter

Encapsulamento em Python

- Entre as várias frases de efeito que ditam o rumo do desenvolvimento de Python está que nós todos somos adultos consentidos.
- Isso significa que, segundo as crenças do criador do Python Guido Van Rossum e o grupo de desenvolvedores do CPython, não há um motivo realmente vantajoso para limitar as possibilidades dos outros programadores — todo mundo sabe o que está fazendo com seu próprio script.
- Em Python, graças à filosofia de design de Guido Van Rossum & cia., não existe encapsulamento, bem como não existem as declarações *private* e *protected*
- Se um desenvolvedor quiser importar minha classe e alterar todos os métodos internos, os efeitos colaterais estarão por conta dele.

Mas e se eu quiser evitar o acesso?

- O Python não utiliza o termo *private*, que é um modificador de acesso e também chamado de modificador de visibilidade. No Python inserimos dois underscores (“__”) ao atributo para adicionar esta característica

```
class Pessoa:
        def __init__(self, idade):
                self.__idade = idade
```

```
pessoa = Pessoa(20)
pessoa.__idade
```

```
Traceback (most recent call last):
  File "exemplo.py", line 8, in <module>
    pessoa.__idade
AttributeError: 'Pessoa' object has no attribute '__idade'
```

Explicação

- O interpretador acusa que o atributo idade não existe na classe Pessoa. Mas isso não garante que ninguém possa acessá-lo.
- No Python não existem atributos realmente privados, ele apenas alerta que você não deveria tentar acessar este atributo, ou modificá-lo. Para acessá-lo, fazemos:
  - p._Pessoa__idade
- Ao colocar o prefixo __ no atributo da classe, o Python apenas renomeia **“__nome_do_atributo”** para **“_nomeda_Classe_nome_do_atributo”**, como fez em __idade para _Pessoa__idade.
- Qualquer pessoa que saiba que os atributos privados não são realmente privados, mas "desconfigurados", pode ler e atribuir um valor ao atributo "privado“ diretamente.
- Mas fazer pessoa._Pessoa__idade = 20 é considerado má prática e pode acarretar em erros.

Getters e Setters no Python

- Como vimos, atributos podem ser acessados de fora de uma classe em Python.
- Sendo assim, pouco adianta criar métodos *get* e *set* se sempre será possível alterar a propriedade por trás. No entanto, isso dificulta o processo quando queremos adicionar efeitos colaterais conforme conversamos anteriormente, certo?
  - Errado!

- Não precisamos nos preocupar com isso porque podemos contar com

  @property.

Exemplo

```python
class Person:
  def __init__(self, name):
    self._name = name
    self._times_updated = 0

  @property
  def name(self):
    return self._name

  @name.setter
  def name(self, name):
    self._times_updated += 1
    self._name = name

  @property
  def times_updated(self):
    return self._times_updated

  def say_hi(self):
    print('Hello, {0}!'.format(self.name))
    print('{0} has had their name changed {1} times.'.format(self.name, self.times_updated))


joao = Person('João')
joao.say_hi()
# Hello, João!
# João has had their name changed 0 times.

joao.name = 'Jo'
joao.say_hi()
# Hello, Jo!
# Jo has had their name changed 1 times.
```