Lógica de Programação Orientada a Objetos

Bem-vindos!

Objetivos

- ✔ Orientação a objeto

DECLARAÇÃO DE CLASSE

- class Foo → class Foo(object)
- Classes são modos para definir novos objetos
- Cria uma novo objeto de classe chamado Foo
- Definição de classe cria um novo namespace (escopo)
- Variáveis definidas no corpo da classe são atributos de classe
- Funções definidas no corpo da classe são métodos de instância

```
class Foo:          #python2 Foo(object):
    pass
```

CONSTRUTORES

- __init__ inicializa uma instância da classe
- x = Circle()
  - Cria um objeto do tipo Circle
  - Chama Circle.__init__(self)
  - Liga self com o nome x

```
class Circle:
    def __init__(self, radius):
        self.r = radius
```

MÉTODOS DE INSTÂNCIA

- Definições de método de instância deve usar self como primeiro argumento

```
class Circle:
    def __init__(self, radius=5):
        self.r = radius
    def get_perimter(self):
        return 2 * math.pi * self.r
```

PRIVADO POR CONVENÇÃO

- O primeiro _ significa que uso é por seu risco
- "Todos os adultos aqui": você pode ainda acessar qualquer variável que você quer

```
class Circle:
    _pi = 3.14
    ...

    def get_perimter(self):
        return 2 * self._pi * self.r
```

HERANÇA SIMPLES

- Super classes são argumentos para a declaração class
- objeto é a classe base padrão
- class Circle(Shape): herda de Shape
- Tenha certeza de chamar o __init__ da super classe

```
class Circle(Shape):
    def __init__(self):
        super().__init__()     # python2 super(Circle, self).__init__()
        self.new_var = default
```

EXEMPLO

```
class Animal:
    sound = None
    def make_sound(self):
        print(self.sound)

class Cat(Animal):
    sound = "meow"

class Duck(Animal):
    sound = "quack"

c = Cat()
c.make_sound()

d = Duck()
d.make_sound()
```

LET'S CODE

HERANÇA MÚLTIPLA

- Você pode herdar múltiplas super classes
- Atributos deve ser resolvido via MRO (Ordem de Resolução de Método)
- Circle.mro()

```
class Circle(Shape, Drawable):
    def __init__(self):
        super().__init__(self)
```

MÉTODOS ESTÁTICOS

- Anexa funções para classes (com contexto similar)
- Um método estático não recebe um argumento self
- Métodos estáticos não deve depender de atributos de classe

```
class Circle:
    @staticmethod
    def radius_to_perimeter(r):
        return 2 * math.pi * r
```

MÉTODOS DE CLASSE

- Um método de classe retorna um objeto de classe como self.
- Alternativa ao construtor.
- Chamada o primeiro argumento cls

```
class Circle:
    @classmethod
    def from_circumference(cls, circ):
        return cls(circ/(2 * math.pi))
```

SEM GETTERS E SETTERS???

No python, @property e @attr.setter substitui a necessidade de getters e setters

Métodos decorados com @property, substitui atributos getter

- Retorno chamados em x.attr

Métodos decorados com @attr.setter, substitui atributos setter

- Retorno chamado em x.attr = val

```python
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def area(self):
        return self.radius ** 2 * math.pi

c = Circle(2)
print(c.area)
```

GLOBAL

- Alterando estado global pode ser perigoso, então Python obriga que você declare isso explicitamente
- global pode contornar closures somente leitura
- A palavra-chave global declara certas variáveis em um bloco de código atual para referência ao escopo global
- Variáveis que seguem global não precisam ser limitadas novamente

```
a = 42
def func():
    global a
    a += 1
```

NONLOCAL

- Somente Python 3
- nonlocal declara certas variáveis em um bloco de código atual para referência ao escopo de inclusão mais próximo.
- Se o escopo de inclusão é um escopo global, então o nonlocal levanta um SyntaxError

```
def outer():
    a = 42
    def func():
        nonlocal a
        print(a)
        a += 1
    func()
```

EXERCÍCIO

- Criar classe costs com atributos:
id(contador global), name(nome do custo), description(descrição), value(valor)

- Criar classe person com atributos:
id(contador global), name(nome completo), address(rua/avenida), number(número), city(cidade), state(sigla do estado), list_of_costs;

- Criar função add_cost(person, cost) para adicionar custo a uma lista de custos;

- Criar função get_cost(person) para retornar último custo adicionado;

- Criar função delete_cost(person, cost) para deletar custo pelo id;

- Criar função update_cost(person, cost) para atualizar custo pelo id;