Modulo 3

Contents

1 Classes

```
# class - Classes são moldes para criar novos objetos
# As classes geram novos objetos (instancias) que
# podem ter seus proprios atributos e metodos.
# Os objetos gerados pela classe podem usar seus dados
# internos para realizar varias açoes.
# Por convenção, usamos PascalCase para nomes de
# classes.
# string = 'Luiz'  # str
# print(string.upper())
# print(isinstance(string, str))
class Pessoa:
    ...


p1 = Pessoa('Luiz', 'Otávio')
p1.nome = 'Luiz'
p1.sobrenome = 'Otávio'

p2 = Pessoa('Maria', 'Joana')
p2.nome = 'Maria'
p2.sobrenome = 'Joana'

print(p1.nome)
print(p1.sobrenome)

print(p2.nome)
print(p2.sobrenome)
```

2 init

```
  # Introdução ao metodo __init__ (inicializador de atributos)
# As classes geram novos objetos (instancias) que
# podem ter seus proprios atributos e metodos.
# Os objetos gerados pela classe podem usar seus dados
# internos para realizar varias açoes.
# Por convenção, usamos PascalCase para nomes de
# classes.
# string = 'Luiz'  # str
# print(string.upper())
# print(isinstance(string, str))
class Pessoa:
    def __init__(self, nome, sobrenome):
        self.nome = nome
        self.sobrenome = sobrenome


p1 = Pessoa('Luiz', 'Otávio')
# p1.nome = 'Luiz'
# p1.sobrenome = 'Otávio'

p2 = Pessoa('Maria', 'Joana')
# p2.nome = 'Maria'
# p2.sobrenome = 'Joana'

print(p1.nome)
print(p1.sobrenome)

print(p2.nome)
print(p2.sobrenome)
```

3 Métodos em instancias de classes python

```
  # Metodos em instancias de classes Python
# Hard coded - e algo que foi escrito diretamente no codigo
class Carro:
    def __init__(self, nome):
        self.nome = nome

    def acelerar(self):
        print(f'{self.nome} está acelerando...')


string = 'Luiz'
print(string.upper())

fusca = Carro('Fusca')
```

```
print(fusca.nome)
fusca.acelerar()

celta = Carro(nome='Celta')
print(celta.nome)
celta.acelerar()
```

3.1 Entendendo self em classes Python

```
# Classe - Molde (geralmente sem dados)
# Instancia da class (objeto) - Tem os dados
# Uma classe pode gerar varias instancias.
# Na classe o self e a propria instancia.
class Carro:
    def __init__(self, nome):
        self.nome = nome

    def acelerar(self):
        print(f'{self.nome} está acelerando...')


fusca = Carro('Fusca')
fusca.acelerar()
Carro.acelerar(fusca)
# print(fusca.nome)
# fusca.acelerar()

celta = Carro(nome='Celta')
celta.acelerar()
Carro.acelerar(celta)
# print(celta.nome)
```

4 Escopo da classe e de métodos da classe

```
class Animal:
    # nome = 'Leão'

    def __init__(self, nome):
        self.nome = nome

        variavel = 'valor'
        print(variavel)

    def comendo(self, alimento):
        return f'{self.nome} está comando {alimento}'

    def executar(self, *args, **kwargs):
        return self.comendo(*args, **kwargs)


leao = Animal(nome='Leão')
print(leao.nome)
print(leao.executar('maçã'))
```

5 Mantendo estados dentro da classe

```
    # Mantendo estados dentro da classe
class Camera:
    def __init__(self, nome, filmando=False):
        self.nome = nome
        self.filmando = filmando

    def filmar(self):
        if self.filmando:
            print(f'{self.nome} Ja está filmando...')
            return

        print(f'{self.nome} está filmando...')
        self.filmando = True

    def parar_filmar(self):
        if not self.filmando:
            print(f'{self.nome} NAO esta filmando...')
            return

        print(f'{self.nome} está parando de filmar...')
        self.filmando = False
```

```
    def fotografar(self):
        if self.filmando:
            print(f'{self.nome} não pode fotografar filmando')
            return

        print(f'{self.nome} está fotografando...')


c1 = Camera('Canon')
c2 = Camera('Sony')

c1.filmar()
c1.filmar()
c1.fotografar()
c1.parar_filmar()
c1.fotografar()

print()

c2.parar_filmar()
c2.filmar()
c2.filmar()
c2.fotografar()
c2.parar_filmar()
c2.fotografar()
```

6 Atributos de classe

```
class Pessoa:
    ano_atual = 2022

    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def get_ano_nascimento(self):
        return Pessoa.ano_atual - self.idade


p1 = Pessoa('João', 35)
p2 = Pessoa('Helena', 12)

print(Pessoa.ano_atual)
# Pessoa.ano_atual = 1

print(p1.get_ano_nascimento())
print(p2.get_ano_nascimento())
```

7 dict e vars para atributos de instancia

```
class Pessoa:
    ano_atual = 2022

    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def get_ano_nascimento(self):
        return Pessoa.ano_atual - self.idade


dados = {'nome': 'João', 'idade': 35}
p1 = Pessoa(**dados)
# p1.nome = 'EITA'
# print(p1.idade)
# p1.__dict__['outra'] = 'coisa'
# p1.__dict__['nome'] = 'EITA'
# del p1.__dict__['nome']
# print(p1.__dict__)
# print(vars(p1))
# print(p1.outra)
# print(p1.nome)
print(vars(p1))
print(p1.nome)
```

8 Métodos de classe + factories (fábricas)

```python
# Sao metodos onde "self" sera "cls", ou seja,
# ao inves de receber a instancia no primeiro
# parametro, receberemos a propria classe.
class Pessoa:
    ano = 2023  # atributo de classe

    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    @classmethod
    def metodo_de_classe(cls):
        print('Hey')

    @classmethod
    def criar_com_50_anos(cls, nome):
        return cls(nome, 50)

    @classmethod
    def criar_sem_nome(cls, idade):
        return cls('Anonima', idade)


p1 = Pessoa('João', 34)
p2 = Pessoa.criar_com_50_anos('Helena')
p3 = Pessoa('Anonima', 23)
p4 = Pessoa.criar_sem_nome(25)
print(p2.nome, p2.idade)
print(p3.nome, p3.idade)
print(p4.nome, p4.idade)
# print(Pessoa.ano)
# Pessoa.metodo_de_classe()
```

9 @staticmethod (métodos estáticos)

```python
     # sao inuteis em Python
    # Metodos estaticos sao metodos que estao dentro da
    # classe, mas nao tem acesso ao self nem ao cls.
    # Em resumo, sao funçoes que existem dentro da sua
    # classe.
    class Classe:
        @staticmethod
        def funcao_que_esta_na_classe(*args, **kwargs):
            print('Oi', args, kwargs)


    def funcao(*args, **kwargs):
        print('Oi', args, kwargs)


    c1 = Classe()
    c1.funcao_que_esta_na_classe(1, 2, 3)
    funcao(1, 2, 3)
    Classe.funcao_que_esta_na_classe(nomeado=1)
    funcao(nomeado=1)
```

10 method vs @classmethod vs @staticmethod

```python
# method - self, metodo de instancia
# @classmethod - cls, metodo de classe
# @staticmethod - metodo estatico (self, cls)
# não tem acesso ao self ou cls
class Connection:
    def __init__(self, host='localhost'):
        self.host = host
        self.user = None
        self.password = None

    def set_user(self, user):
        self.user = user

    def set_password(self, password):
        self.password = password

    @classmethod
    def create_with_auth(cls, user, password):
        connection = cls()
        connection.user = user
```

```
        connection.password = password
        return connection

    @staticmethod
    def log(msg):
        print('LOG:', msg)


def connection_log(msg):
    print('LOG:', msg)


# c1 = Connection()
c1 = Connection.create_with_auth('luiz', '1234')
# c1.set_user('luiz')
# c1.set_password('123')
print(Connection.log('Essa e a mensagem de log'))
print(c1.user)
print(c1.password)
```

11 @property - um getter no modo Pythônico

```
# getter - um metodo para obter um atributo
# cor -> get_cor()
# modo pythonico - modo do Python de fazer coisas
# @property e uma propriedade do objeto, ela
# e um metodo que se comporta como um
# atributo
# Geralmente e usada nas seguintes situaçoes:
# - como getter
# - p/ evitar quebrar codigo cliente
# - p/ habilitar setter
# - p/ executar açoes ao obter um atributo
# Codigo cliente - e o codigo que usa seu codigo
class Caneta:
    def __init__(self, cor):
        self.cor_tinta = cor

    @property
    def cor(self):
        print('PROPERTY')
        return self.cor_tinta

    @property
    def cor_tampa(self):
        return 123456

############################


caneta = Caneta('Azul')
print(caneta.cor)
print(caneta.cor)
print(caneta.cor)
print(caneta.cor)
print(caneta.cor)
print(caneta.cor)
print(caneta.cor_tampa)

####################################

# class Caneta:
#     def __init__(self, cor):
#         self.cor_tinta = cor

#     def get_cor(self):
#         print('GET COR')
#         return self.cor_tinta

# ############################


# caneta = Caneta('Azul')
# print(caneta.get_cor())
# print(caneta.get_cor())
# print(caneta.get_cor())
# print(caneta.get_cor())
# print(caneta.get_cor())
```

12 @property + @setter - getter e setter no modo Pythônico

```
# - como getter
# - p/ evitar quebrar codigo cliente
# - p/ habilitar setter
# - p/ executar acoes ao obter um atributo
# Atributos que começar com um ou dois underlines
# não devem ser usados fora da classe.
class Caneta:
    def __init__(self, cor):
        # private protected
        self.cor = cor
        self._cor_tampa = None

    @property
    def cor(self):
        print('ESTOU NO GETTER')
        return self._cor

    @cor.setter
    def cor(self, valor):
        print('ESTOU NO SETTER')
        self._cor = valor

    @property
    def cor_tampa(self):
        return self._cor_tampa

    @cor_tampa.setter
    def cor_tampa(self, valor):
        self._cor_tampa = valor


caneta = Caneta('Azul')
caneta.cor = 'Rosa'
caneta.cor_tampa = 'Azul'
print(caneta.cor)
print(caneta.cor_tampa)
```

13 Encapsulamento (modificadores de acesso: public, protected, private)

```
# Python NAO TEM modificadores de acesso
# Mas podemos seguir as seguintes convencoes
#   (sem underline) = public
#       pode ser usado em qualquer lugar
# _ (um underline) = protected
#       não DEVE ser usado fora da classe
#       ou suas subclasses.
# __ (dois underlines) = private
#       "name mangling" (desfiguração de nomes) em Python
#       _NomeClasse__nome_attr_ou_method
#       so DEVE ser usado na classe em que foi
#       declarado.
from functools import partial


class Foo:
    def __init__(self):
        self.public = 'isso e publico'
        self._protected = 'isso e protegido'
        self.__exemplo = 'isso e private'

    def metodo_publico(self):
        # self._metodo_protected()
        # print(self._protected)
        print(self.__exemplo)
        self.__metodo_private()
        return 'metodo_publico'

    def _metodo_protected(self):
        print('_metodo_protected')
        return '_metodo_protected'

    def __metodo_private(self):
        print('__metodo_private')
        return '__metodo_private'
```

```
f = Foo()
# print(f.public)
print(f.metodo_publico())
```

14 Relações entre classes: associação, agregação e composição

```
# Associaçao e um tipo de relacao onde os objetos
# estão ligados dentro do sistema.
# Essa e a relação mais comum entre objetos e tem subconjuntos
# como agregação e composição (que veremos depois).
# Geralmente, temos uma associação quando um objeto tem
# um atributo que referencia outro objeto.
# A associação não especifica como um objeto controla
# o ciclo de vida de outro objeto.
class Escritor:
    def __init__(self, nome) -> None:
        self.nome = nome
        self._ferramenta = None

    @property
    def ferramenta(self):
        return self._ferramenta

    @ferramenta.setter
    def ferramenta(self, ferramenta):
        self._ferramenta = ferramenta


class FerramentaDeEscrever:
    def __init__(self, nome):
        self.nome = nome

    def escrever(self):
        return f'{self.nome} está escrevendo'


escritor = Escritor('Luiz')
caneta = FerramentaDeEscrever('Caneta Bic')
maquina_de_escrever = FerramentaDeEscrever('Máquina')
escritor.ferramenta = maquina_de_escrever

print(caneta.escrever())
print(maquina_de_escrever.escrever())
print(escritor.ferramenta.escrever())
```

15 Agregação

```
# Agregacao e uma forma mais especializada de associacao
# entre dois ou mais objetos. Cada objeto terá
# seu ciclo de vida independente.
# Geralmente e uma relacao de um para muitos, onde um
# objeto tem um ou muitos objetos.
# Os objetos podem viver separadamente, mas pode
# se tratar de uma relação onde um objeto precisa de
# outro para fazer determinada tarefa.
# (existem controversias sobre as definicoes de agregacao).
class Carrinho:
    def __init__(self):
        self._produtos = []

    def total(self):
        return sum([p.preco for p in self._produtos])

    def inserir_produtos(self, *produtos):
        # self._produtos.extend(produtos)
        # self._produtos += produtos
        for produto in produtos:
            self._produtos.append(produto)

    def listar_produtos(self):
        print()
        for produto in self._produtos:
            print(produto.nome, produto.preco)
        print()


class Produto:
    def __init__(self, nome, preco):
```

```
        self.nome = nome
        self.preco = preco


carrinho = Carrinho()
p1, p2 = Produto('Caneta', 1.20), Produto('Camiseta', 20)
carrinho.inserir_produtos(p1, p2)
carrinho.listar_produtos()
print(carrinho.total())
```

16 Composição

```
 e uma especializacao da agregacao.
# Mas nela, quando o objeto "pai" for apagado, todas
# as referencias dos objetos filhos tambem sao
# apagadas.
class Cliente:
    def __init__(self, nome):
        self.nome = nome
        self.enderecos = []

    def inserir_endereco(self, rua, numero):
        self.enderecos.append(Endereco(rua, numero))

    def inserir_endereco_externo(self, endereco):
        self.enderecos.append(endereco)

    def listar_enderecos(self):
        for endereco in self.enderecos:
            print(endereco.rua, endereco.numero)

    def __del__(self):
        print('APAGANDO,', self.nome)


class Endereco:
    def __init__(self, rua, numero):
        self.rua = rua
        self.numero = numero

    def __del__(self):
        print('APAGANDO,', self.rua, self.numero)


cliente1 = Cliente('Maria')
cliente1.inserir_endereco('Av Brasil', 54)
cliente1.inserir_endereco('Rua B', 6745)
endereco_externo = Endereco('Av Saudade', 123213)
cliente1.inserir_endereco_externo(endereco_externo)
cliente1.listar_enderecos()

del cliente1


print(endereco_externo.rua, endereco_externo.numero)
print('########################## AQUI TERMINA MEU CODIGO')
```

17 Herança simples Relações entre classes

```
    # Associacao - usa, Agregacao - tem
    # Composicao - e dono de, Herança - e um
    # Herança vs Composição
    # Classe principal (Pessoa)
    #   -> super class, base class, parent class
    # Classes filhas (Cliente)
    #   -> sub class, child class, derived class
```

17.1 primeiro exemplo

```
# Classe principal (Pessoa)
#   -> super class, base class, parent class
# Classes filhas (Cliente)
#   -> sub class, child class, derived class
class Pessoa:
    cpf = '1234'
```

```
    def __init__(self, nome, sobrenome):
        self.nome = nome
        self.sobrenome = sobrenome

    def falar_nome_classe(self):
        print('Classe PESSOA')
        print(self.nome, self.sobrenome, self.__class__.__name__)


class Cliente(Pessoa):
    def falar_nome_classe(self):
        print('EITA, nem sai da classe CLIENTE')
        print(self.nome, self.sobrenome, self.__class__.__name__)


class Aluno(Pessoa):
    cpf = 'cpf aluno'
    ...


c1 = Cliente('Luiz', 'Otávio')
c1.falar_nome_classe()
a1 = Aluno('Maria', 'Helena')
a1.falar_nome_classe()
print(a1.cpf)
# help(Cliente)
```

17.2 super() e a sobreposição de membros – Python Orientado a Objetos

```
# Classe principal (Pessoa)
#   -> super class, base class, parent class
# Classes filhas (Cliente)
#   -> sub class, child class, derived class
# class MinhaString(str):
#     def upper(self):
#         print('CHAMOU UPPER')
#         retorno = super(MinhaString, self).upper()
#         print('DEPOIS DO UPPER')
#         return retorno


# string = MinhaString('Luiz')
# print(string.upper())

class A(object):
    atributo_a = 'valor a'

    def __init__(self, atributo):
        self.atributo = atributo

    def metodo(self):
        print('A')


class B(A):
    atributo_b = 'valor b'

    def __init__(self, atributo, outra_coisa):
        super().__init__(atributo)
        self.outra_coisa = outra_coisa

    def metodo(self):
        print('B')


class C(B):
    atributo_c = 'valor c'

    def __init__(self, *args, **kwargs):
        # print('EI, burlei o sistema.')
        super().__init__(*args, **kwargs)

    def metodo(self):
        # super().metodo()  # B
        # super(B, self).metodo()  # A
        # super(A, self).metodo()  # object
        A.metodo(self)
        B.metodo(self)
        print('C')
```

```
# print(C.mro())
# print(B.mro())
# print(A.mro())
c = C('Atributo', 'Qualquer')
# print(c.atributo)
# print(c.outra_coisa)
c.metodo()
# print(c.atributo_a)
# print(c.atributo_b)
# print(c.atributo_c)
# c.metodo()
```

17.3 continuação

```
    # super() e a sobreposição de membros - Python Orientado a Objetos
# Classe principal (Pessoa)
#   -> super class, base class, parent class
# Classes filhas (Cliente)
#   -> sub class, child class, derived class
# class MinhaString(str):
#     def upper(self):
#         print('CHAMOU UPPER')
#         retorno = super(MinhaString, self).upper()
#         print('DEPOIS DO UPPER')
#         return retorno


# string = MinhaString('Luiz')
# print(string.upper())

class A(object):
    atributo_a = 'valor a'

    def __init__(self, atributo):
        self.atributo = atributo

    def metodo(self):
        print('A')


class B(A):
    atributo_b = 'valor b'

    def __init__(self, atributo, outra_coisa):
        super().__init__(atributo)
        self.outra_coisa = outra_coisa

    def metodo(self):
        print('B')


class C(B):
    atributo_c = 'valor c'

    def __init__(self, *args, **kwargs):
        # print('EI, burlei o sistema.')
        super().__init__(*args, **kwargs)

    def metodo(self):
        # super().metodo()  # B
        # super(B, self).metodo()  # A
        # super(A, self).metodo()  # object
        A.metodo(self)
        B.metodo(self)
        print('C')


# print(C.mro())# metodo resolution order
# print(B.mro())
# print(A.mro())
c = C('Atributo', 'Qualquer')
# print(c.atributo)
# print(c.outra_coisa)
c.metodo()
# print(c.atributo_a)
# print(c.atributo_b)
# print(c.atributo_c)
# c.metodo()
```

18

19