

00 com Python



FATEC 2005

Projetos 00

- **Projetos desenvolvidos utilizando a Orientação a Objetos** são mais **estáveis**, de **fácil manutenção** e sua **reutilização** é mais simples
- Na linguagem Python, quando iniciamos um projeto, apesar dele poder ser desenvolvido utilizando os conceitos de programação procedural, a linguagem já vai pré organizá-lo para a orientação a objetos, pois ele será organizado por meio de estruturas denominadas **Classes** que vão armazenar trechos de códigos relacionados entre si
- Ao utilizar a classe definida não é necessário compreender como os trechos dela foram desenvolvidos, basta utilizá-los por meio de um processo conhecido como **Abstração**

Técnicas de OO no Python

- Classes :
 - Definição do Objeto
 - Conjunto de atributos e funções utilizado como modelo para criação de objetos
- Objetos:
 - Variável que possui todas as características comuns à classe, porém, com valores diferentes em seus atributos
- Abstração:
 - Acesso às utilidades da classe
 - Ação de utilizar mensagens para acessar os recursos de uma classe

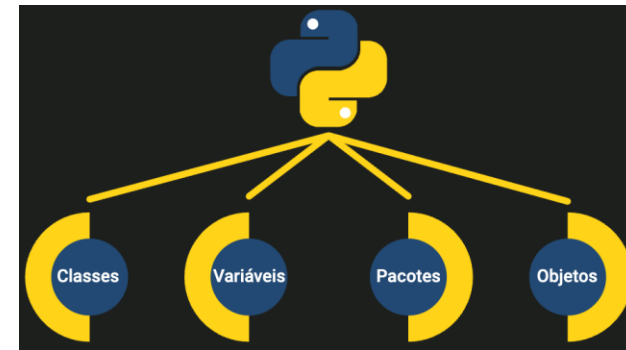
Técnicas de OO no Python

- Atributos :
 - Características do elemento que a classe representa
- Funções:
 - Ações do objeto com retorno
 - Retorna um valor declarado dentro da classe
- Ações do Objeto:
 - Função nativa que não possui retorno declarado dentro de uma classe
- Exceção
 - Tratamento de erros
- Mensagem
 - Chamada a um atributo, método ou função

Técnicas de OO no Python

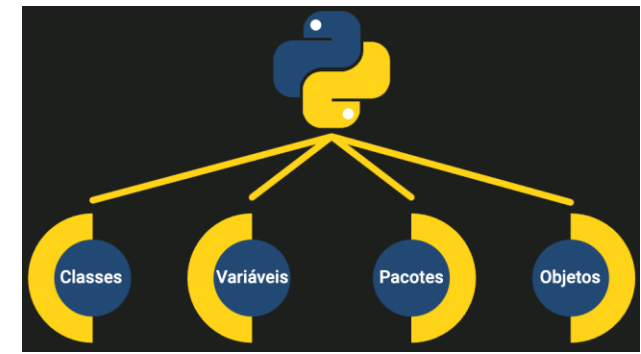
- Herança :
 - Super classe e Sub classe
 - Uma classe pode herdar atributos, métodos e/ou função de outra
- Encapsulamento:
 - O encapsulamento permite que os atributos sejam vistos somente nas classes onde foram declarados, definindo o nível de acesso de atributos, métodos ou funções
- Polimorfismo:
 - Escolher entre os atributos, métodos e/ou funções que sobrescreveram ou que foram sobrescritos

Nomenclatura dos elementos



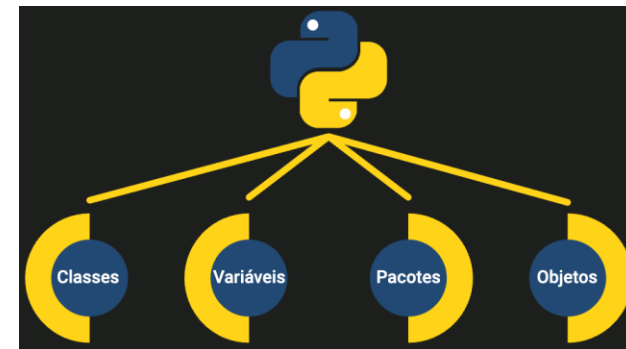
- Referente a caracteres, seguir o mesmo padrão de variáveis e objetos.
- Sempre iniciar as classes com caracteres maiúsculos, inclusive as iniciais de nomes compostos:
Exemplo: MinhaClasse()


Nomenclatura dos elementos



- Utilizar somente caracteres e letras minúsculas.
- No caso de variáveis com nome composto, utilizar o *underline* para separação das palavras.
- Não iniciar o nome com números (podemos utilizar números no nome, mas não devemos iniciar com eles).
- Não utilizar caracteres especiais.
- Não utilizar espaçamento em branco.
- Evitar utilizar os caracteres l e o.

Nomenclatura dos elementos





Pacotes,
Módulos e
Métodos

- Utilizar nomes pequenos.
- Utilizar sempre caracteres minúsculos.
- Utilizar o *underline* para unir nomes compostos.

Classes

- A partir das classes é possível criar objetos, ou seja, uma classe é um “molde” para a criação de objetos
- A classe é um Tipo Abstrato de dados (TAD), ou seja, o código que define e implementa um novo tipo de informação

Objetos

- Classe é o projeto do objeto, contendo o código de programação
- Objeto é a execução do código de uma classe
 - Quando executamos o código de uma classe é criado um novo objeto na memória
 - Uma nova instância de um objeto é um tipo abstrato de informação de um novo tipo de dado
- Instância
 - Objeto sendo executado
 - Ao criar um objeto, cria-se uma instância dele

Declaração dos membros da classe

- Membros da classe são usados para manter uma mesma estrutura de tudo que pertence a determinada informação da classe
- Classes contém valores e esses necessitam de funções específicas para serem manipulados
- Dois tipos de membros:
 - Atributos
 - Armazenam as características de uma classe
 - Declarações de variáveis da classe
 - Métodos
 - Ações da classe, suas funções
 - Representam os estados e ações dos objetos quando instanciados

Método construtor

- Definido de forma implícita ou explícita
- Instanciar é por meio do construtor que ele será inicializado
- Invocado, automaticamente, pela máquina virtual do python todas as vezes que um objeto é criado

#def é usada para a declaração do método
#init() método especial que será chamado
para criar um objeto da classe
parâmetro Self exporta as características
do objeto

```
class Cliente:  
def __init__(self):  
pass
```

Atributos da classe

```
class Cliente:
    def __init__(self, n, fone):
        self.nome = n
        self.telefone = fone
```

O Método Construtor da classe **pode conter um conjunto de parâmetros**

Com isso, podemos determinar os valores para inicialização dos atributos

Isso garante um melhor funcionamento de toda a estrutura do objeto, **obrigando ao programador** determinar **valores default** no momento da inicialização do objeto



Instanciando Objetos

A classe é o código e, para que esse código seja utilizado, precisamos criar os objetos, assim, criamos **instâncias do objeto**

- Na orientação a objetos, **instância e objetos são sinônimos**.

Objetos

- Na linguagem Python, todo objeto criado possui um código de identificação **composto por um número inteiro não negativo**, conhecido como **ID**
- Assim, as instâncias de objetos são diferenciadas

```
a = [1, 2, 3]
b = a
```

```
print(id(a))    # mesmo ID de b
print(id(b))    # aponta para o mesmo
objeto que a
```

```
c = [1, 2, 3]
print(id(c))    # diferente, apesar do
conteúdo igual
```

Projeto controle bancário

```
%%writefile cliente.py
class Cliente:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade

    def apresentar(self):
        print(f"Nome: {self.nome}, Idade: {self.idade}")
```

```
%%writefile conta.py
class Conta:
    def __init__(self, titular, numero, saldo):
        self.numero = numero
        self.saldo = 0
        self.titular=titular
```

```
from cliente import Cliente
from conta import Conta

def main():
    cliente1 = Cliente("João", 30)
    conta1 = Conta(cliente1.nome, 12345, 1000)
    print(conta1.titular, "- Numero ", conta1.numero, "seu saldo:", conta1.saldo)
    cliente1.apresentar()

main()
```


Encapsulamento de dados

- Uma das principais vantagens do conceito de orientação a objetos é a **utilização de estruturas sem a necessidade de conhecer como elas foram implementadas**
- **Encapsulamento de dados** envolve a **proteção dos atributos ou métodos de uma classe**
- Proteger atributos e métodos de uma classe (tornando-os privados), de forma que somente a classe onde as declarações foram feitas tenham acesso
- Esse conceito garante a integridade das informações e também facilita a utilização das implementações

Variáveis e funções que são utilizadas internamente **não devem estar disponíveis externamente**

Modificadores de acesso

- a linguagem Python utiliza o símbolo *underscore* "_"
- Dentro da orientação a objetos temos os modificadores **Public**, **Protected** e **Private**
- Public
 - É o mais comum entre os modificadores
 - Ele permite acesso tanto de dentro, quanto de fora de uma classe
 - Sua implementação se dá por meio do uso do *underline* "_" na frente do nome
- Protect
 - Somente suas classes e subclasses terão acesso ao atributo ou método
 - Para sua implementação adicione um *underline* "_" antes do nome
- Private
 - Permite que somente a sua classe (onde foi definido) tenha acesso a um determinado atributo ou método
 - Para definir o método *private* adicionamos *underline* duplo "__" na frente do nome.

Visibilidade de membros

- Um dos recursos mais importantes da orientação a objetos é o de **restringir o acesso às variáveis de um objeto e a alguns métodos**
- O objetivo principal desta ação é **evitar que variáveis internas sejam acessadas** e recebam valores diretamente ou, ainda, que métodos internos sejam invocados externamente, garantindo, assim, a integridade das informações
- Para modificar a visualização de um membro dentro das IDEs utilizamos as **convenções** apresentadas para a linguagem

Modificando atributos

```
> %%writefile Cliente.py
class Cliente:
    def __init__(Self,n,fone):
        self._nome=n          #privados
        self._telefone = fone
```

Métodos de acesso

- Para permitir o acesso aos atributos de forma controlada, a prática mais comum é a utilização de dois métodos de acesso: um **retornando valor** e outro que **muda valor**
- **Getters** e **Setters** são usados na maioria das linguagens de programação orientada a objetos com o objetivo de garantir o princípio de encapsulamento de dados
- Os métodos são utilizados para implementações que alteram os valores internos da classe ou que retornam valores dela
- Get
 - **Sempre retornam valores**
 - O método **Get** é utilizado para ler os valores internos do objeto e enviá-los como valor de retorno da função
- Set
 - **Recebem valores por parâmetros**
 - Os métodos **Set** recebem argumentos que serão atribuídos a membros internos do objeto

Sintaxe

GET

`get_nome do atributo()`

Exemplo:

- `get_idade(self):return
self._idade`

SET

`set_nome do atributo(valor por parâmetro)`

Exemplo:

```
def set_idade(self, valor):  
    self.idade = valor
```

Interface da classe

Quando os dados estão encapsulados não é necessário mudar as regras de negócio em vários lugares, mas sim em um único lugar, já que essa regra está encapsulada

- O conjunto de métodos públicos de uma classe é conhecido como **Interface da classe**, sendo a única maneira de comunicação com os objetos da classe

a prática o “_” (*underline*) antes do atributo não impede o acesso dele em outra classe, ou seja, **ele não fica privado**.

- Essa forma é somente um indicativo de que os métodos nos quais os nomes iniciam com “_” (*underline*) não devem (mas podem) ser acessados

Convenção de estilo

Em Python, este conceito de “público e privado” não existe na sintaxe da linguagem

- O que temos em Python é a **convenção de estilo** que diz que nomes de atributos, métodos e funções iniciados com “_” (*underscore*) não devem ser usados por usuários de uma classe, só pelos próprios implementadores e que o funcionamento desses métodos e funções pode mudar sem aviso prévio

 Como se convencionou o acesso em Python:

Forma do nome	Significado	Exemplo	Acesso permitido?
nome	Público	self.nome	✅ Sim
_nome	Protegido (por convenção)	self._nome	⚠️ Sim, mas não recomendado para uso externo
__nome	Privado (name mangling)	self.__nome	❌ Não diretamente acessível — o nome é “obfuscado”

```
class Exemplo:
    def __init__(self):
        self.__segredo = 42

e = Exemplo()
print(e.__segredo) # ERRO
print(e._Exemplo__segredo) #
# FUNCIONA, mas é feio!
```


Protocolo de descritores – Decorator

- Um **decorator** é um padrão de projeto de software que permite adicionar comportamento a um objeto já existente, em tempo de execução, ou seja, agrega, de forma dinâmica, responsabilidades adicionais a um objeto
- Na prática, o decorator permite que atributos de uma classe tenham **responsabilidades**
- Um decorator **é um objeto invocável, uma função que aceita outra função como parâmetro (a função decorada)**
- O decorator pode realizar algum processamento com a função decorada e devolvê-la ou substituí-la por outra função

@Property

A função **Property** é um Decorator e é utilizada para obter um valor de um atributo.

- Basicamente, a função **Property** permite que você declare uma função para obter o valor de um atributo
- O decorador `@property` é uma forma simplificada de criar um descritor de leitura e escrita
- Property deve ser utilizada somente se você precisar da funcionalidade de transformar ou verificar um atributo quando ele é atribuído ou lido

```
%writefile Conta.py
class Conta:
    def __init__(self, titular, numero, saldo):
        self.saldo=0
        self.numero = numero
        self.titular=titular

    @property
    def saldo(self):
        return self._saldo

    @saldo.setter
    def saldo(self,saldo):
        if( saldo < 0):
            print("o saldo não pode ser negativo")
        else:
            self._saldo=saldo
```

Exemplo: atributo público

```
class Conta:
    def __init__(self, saldo):
        self.saldo = saldo # qualquer
        valor pode ser atribuído

c = Conta(-100)           # aceito sem erro
print(c.saldo)            # imprime -100
```

Problemas:

- Sem **validação**: permite saldo negativo
- Código que usa a classe pode **quebrar regras de negócio**
- Sem **proteção** da integridade dos dados

Exemplo: Encapsulamento com métodos getters e setters

```
class Conta:
    def __init__(self, saldo):
        self.set_saldo(saldo)

    def get_saldo(self):
        return self._saldo

    def set_saldo(self, valor):
        if valor < 0:
            print("Saldo não pode ser negativo")
            self._saldo = 0
        else:
            self._saldo = valor

c = Conta(-50)           # mensagem de erro, saldo ajustado
print(c.get_saldo())     # imprime 0

c.set_saldo(200)         # aceita novo valor
print(c.get_saldo())     # imprime 200
```

Benefícios:

- Validação de entrada.
- Ocultação de dados via `_saldo`.
- Melhor, mas ainda com uma interface mais verbosa (`get_`, `set_`).

Exemplo: Uso de @property e @setter (descriptor)

```
class Conta:
    def __init__(self, saldo):
        self.saldo = saldo # setter automático

    @property
    def saldo(self):        # getter
        return self._saldo

    @saldo.setter
    def saldo(self, valor): # setter
        if valor < 0:
            print("Saldo não pode ser negativo")
            self._saldo = 0
        else:
            self._saldo = valor

c = Conta(-100) # ajuste automático
print(c.saldo)  # usa getter

c.saldo = 500   # usa setter, sem chamar método
diretamente

print(c.saldo)
```

- Interface limpa: `obj.saldo` e `obj.saldo = x`
- **Validação embutida**, sem expor `get_` e `set_`
- O código cliente **acessa como se fosse atributo**, mas com segurança e controle
- **Mais legível, mais elegante**

Exemplo: leitura de arquivos

```
%%writefile dataset_leitor.py
import pandas as pd

class DatasetLeitor:
    def __init__(self, arquivo_csv):
        self.__caminho = arquivo_csv          # privado
        self._dados = None                    # protegido (por convenção)
        self.__carregar_dados()                # carregamento automático

    def __carregar_dados(self):                # método privado
        try:
            self._dados = pd.read_csv(self.__caminho)
            print("Dados carregados com sucesso.")
        except FileNotFoundError:
            print("Arquivo não encontrado.")
        except Exception as e:
            print("Erro ao carregar dados:", e)
```

```
@property
def dados(self):                                # getter
    return self._dados.copy()                  # protege o original

@property
def colunas(self):                              # interface de acesso
    return list(self._dados.columns)

def filtrar_coluna(self, nome_coluna, valor): # interface de negócio
    if nome_coluna not in self._dados.columns:
        print(f"Coluna '{nome_coluna}' não encontrada.")
        return None
    return self._dados[self._dados[nome_coluna] == valor]

def resumo(self):                              # interface analítica
    return self._dados.describe()

def linhas_nulas(self):                       # utilidade analítica
    return self._dados[self._dados.isnull().any(axis=1)]
```

```
# Simula um CSV |
import pandas as pd

df = pd.DataFrame({
    'Nome': ['Ana', 'Bruno', 'Carlos', 'Ana'],
    'Idade': [23, 31, 19, None],
    'Nota': [8.5, 7.2, 9.1, 6.8]
})
df.to_csv('dados.csv', index=False)
```

```
from dataset_leitor import DatasetLeitor

dh = DatasetLeitor('dados.csv')

print("Colunas:", dh.colunas)
print("Resumo:\n", dh.resumo())
print("Filtro por nome == Ana:\n", dh.filtrar_coluna('Nome', 'Ana'))
print("Linhas com nulos:\n", dh.linhas_nulas())
```


Exemplo

Alterando a classe para:

- Adicionar um método que **salve** os dados tratados em outro CSV
- Criar um método que **normalize** colunas numéricas
- Implementar um filtro com **múltiplas condições**

1. Salvar dados tratados

```
def salvar_csv(self, caminho_saida):  
    try:  
        self._dados.to_csv(caminho_saida, index=False)  
        print(f"Dados salvos em: {caminho_saida}")  
    except Exception as e:  
        print("Erro ao salvar:", e)
```

2. Normalizar colunas numéricas

```
def normalizar_colunas(self):  
    colunas_numericas = self._dados.select_dtypes(include=['number']).columns  
    self._dados[colunas_numericas] = (  
        self._dados[colunas_numericas] - self._dados[colunas_numericas].min()  
    ) / (  
        self._dados[colunas_numericas].max() - self._dados[colunas_numericas].min()  
    )  
    print("Colunas numéricas normalizadas (0 a 1).")
```

3. Filtro com múltiplas condições

```
def filtro_personalizado(self, **condicoes):  
    dados_filtrados = self._dados.copy()  
    for coluna, valor in condicoes.items():  
        if coluna not in dados_filtrados.columns:  
            print(f"Coluna '{coluna}' não encontrada.")  
            continue  
        dados_filtrados = dados_filtrados[dados_filtrados[coluna] == valor]  
    return dados_filtrados
```

```
from dataset_handler import DatasetHandler

dh = DatasetHandler('dados.csv')

# ✓ Resumo antes da normalização
print(dh.resumo())

# ✓ Normalizando colunas numéricas
dh.normalizar_colunas()

# ✓ Resumo após normalização
print(dh.resumo())

# ✓ Salvando novo CSV
dh.salvar_csv("dados_normalizados.csv")

# ✓ Filtro múltiplo: Nome == 'Carlos' e Nota == 1.0 (depois da normalização)
resultado = dh.filtro_personalizado(Nome='Carlos', Nota=1.0)
print("🔍 Filtro personalizado:\n", resultado)
```