ICP225 - Computação II

Prof. Gabriel Rodrigues Caldas de Aquino

gabrielaquino@ic.ufrj.br
Instituto de Computação - Universidade Federal do Rio de Janeiro

Compilado em:
October 20, 2025

Método de Avaliação

Componentes da Nota

- **Provas (P1 e P2):** Média das provas (MP) $\rightarrow$ 80% da nota
- **Trabalhos:** Média dos trabalhos (MT) $\rightarrow$ 20% da nota

Critério de Aprovação Direta

- Média ponderada de MP e MT ≥ 7 $\rightarrow$ Aprovado direto
- Se $3 <=$ Média ponderada de MP e MT $<= 7$ $\rightarrow$ Prova final (PF)

Cálculo da Média Final

$$\text{Média final} = \frac{\text{Média anterior} + \text{PF}}{2}$$

- Se Média final $\geq 5 \Rightarrow$ Aprovado
- Caso contrário $\Rightarrow$ Reprovado

Revisão Inicial

Prof. Gabriel Rodrigues Caldas de Aquino

gabrielaquino@ic.ufrj.br
Instituto de Computação - Universidade Federal do Rio de Janeiro

Compilado em:
October 20, 2025

Material de Referência

Site de Computação 1 - Python

Acesse o conteúdo, exemplos e exercícios do curso de Computação 1 em:

https://python.ic.ufrj.br/index.html

Explore o material complementar, slides, desafios e mais recursos úteis para aprender Python.

Paradigma de Programação

- **Padrão** para estruturação do programa.
- Organização da **solução algorítmica**.
- Exemplo em Python (imperativo estruturado):

```
def calcular_media(notas):
    soma = 0
    for n in notas:
        soma += n
    return soma / len(notas)
```

Estilos de Programação

Estilo de Programação

- **Organização do código** (legibilidade, manutenção).
- Influência no **raciocínio do programador**.
- Exemplo modular em Python:

```
def quadrado(x):
    return x ** 2

print(quadrado(5))  # Saída: 25
```

- Em Comp2: Evolução para POO (classes, herança) e outros paradigmas

Paradigma Imperativo

Paradigma Imperativo Estruturado

- Código sequencial
- "Faça isso, depois isso, depois aquilo..."

```
valor = int(input("Digite um número: "))
if valor > 0:
    print("Positivo")
    resultado = valor * 2
else:
    print("Negativo")
    resultado = valor + 10
print("Resultado:", resultado)
```

Estilo Modular

- Construção de módulos
- Pequenos códigos (módulos) que tem uma função única
- Evitar fazer códigos grandes
- A combinação dos módulos realiza uma grande tarefa
- Os módulos são as funções

```
# Módulo de processamento
def calcular(x):
    return x * 2 if x > 0 else x + 10
# Módulo de entrada e saída
def ler_imprimir():
    valor = int(input("Digite um número: "))
    print("Resultado:", calcular(valor))
ler_imprimir()
```

Tipos de Dados Básicos

Numéricos

```
# Inteiro (int)
idade = 25

# Ponto flutuante (float)
peso = 68.5
```

Texto (string)

```
# String (str)
nome = "Maria"
msg = 'Olá, mundo!'
```

Características

- `int`: Valores inteiros (ex: -3, 0, 42)
- `float`: Decimais (ex: 3.14, -0.5)
- `str`: Texto (entre aspas simples ou duplas)

Operações com Tipos Básicos

Numéricas

```
soma = 5 + 3.2          #Result: 8.2
subtracao = 10 - 4      #Result: 6
multiplicacao = 7 * 2   #Result: 14
divisao = 15 / 2        #Result: 7.5
modulo = 10 % 3         #Result: 1
exponenciacao = 2 ** 3 #Result: 8
```

Strings

```
"a" + "b"    # "ab" (concatenação)
"oi" * 3     # "oioioi" (repetição)
len("abc")   # 3 (tamanho)
```

Importante

- Não misture tipos! `"10" + 5` → Erro!
- Converta com `int()`, `str()`, etc.

Variáveis e Atribuição

O que são variáveis?

- **Armazenam informações** para uso futuro e são mantidas na **memória RAM**
- Acessadas através de um **identificador** (nome)

Comando de Atribuição

```
# Sintaxe: nome_variavel = valor
idade = 25              # Criando variável 'idade'
pi = 3.1415             # Atribuição de valor decimal
mensagem = "Olá!"       # Variável do tipo string
```

Regras Importantes

- Nomes à **esquerda** do =, valores à **direita**
- Podem ser **reescritas**: `contador = 10` → `contador = 20`
- Tipos são **inferidos** automaticamente

Funções em Python

O que são funções?

- **Trechos de código** com um objetivo específico
- **Organizam** e **dividem** problemas complexos
- Podem **retornar valores** ou executar ações

Por que usar funções?

- **Evita repetição** de código
- Facilita **manutenção**
- Permite **reuso** em diferentes partes do programa

Sintaxe Básica

```
def nome_funcao(argumentos):
    # Corpo da função
    return resultado
# Exemplo real:
def calcular_media(nota1, nota2):
    media = (nota1 + nota2) / 2
    return media
```

Exemplo Prático de Funções com Reuso

Código em Python

```
def calcular_area(largura, altura):
    """Calcula área de um retângulo"""
    return largura * altura

# Reuso da função
parede = calcular_area(3, 2.5)
janela = calcular_area(1.2, 1.5)
porta = calcular_area(0.8, 2.1)

area_total = parede - janela - porta

print(f"Área para pintar: {area_total}m²")
```

Escopo de Variáveis em Funções - Cuidado!

Exemplo Prático

```
def calcular():
    x = 10  # Variável local
    print(x)

calcular()
print(x)  # ERRO! x não existe aqui
```

Escopo: Conceito chave que determina

- **Tempo de vida**:
  Quando a variável existe
- **Visibilidade**:
  Onde a variável pode ser acessada

Regras Importantes

- Variáveis **dentro** da função:
  - Nascem quando a função é chamada
  - Morrem quando a função termina
  - São **invisíveis** fora da função
- **Cuidado**:
  Tentar acessar fora causa erro

Strings em Python - Definição e Índices

Definição e Criação

- Sequência imutável de caracteres
- Podem usar ’ ’ ou " "
- Exemplos:

```
disciplina = ’Computação 1’
nome = "Maria"
```

Acesso por Índice

- Posições começam em 0
- Índices negativos acessam do final

```
print(nome[0])    # ’M’
print(nome[-1])   # ’a’
```

Tabela de Índices

Índice +	0	1	2	3	4
Caractere	M	a	r	i	a
Índice -	-5	-4	-3	-2	-1

Imutabilidade de string

- Strings são **imutáveis**
- Tentar modificar gera erro:

```
nome[0] = ’m’   # TypeError
```

Strings em Python - Operações Comuns

Operações com Strings

- **Concatenação (+):** Junta duas strings.
- **Repetição (*):** Repete a string várias vezes.
- **Tamanho (`len`):** Retorna o número de caracteres.
- **Pertencimento (`in`):** Verifica se uma substring está contida.
- **Fatiamento (`:`):** Retorna partes da string (substrings).

Exemplos

```
len(nome)          # 5
nome + ' Silva'    # 'Maria Silva'
'ri' in nome       # True
'-' * 10           # '----------'
nome[1:4]          # 'ari'
```

Tudo sobre Strings em Python - Demonstração Prática

```
nome = "Ana Clara"
idade = 22

print(nome[0], nome[-1])  # A a

print(nome[4:9])   # Clara
print(nome[:3])    # Ana

print(len(nome))           # 8
print("Ana" in nome)       # True
print(nome + " Silva")     # Ana Clara Silva

print(nome.split())        # ['Ana', 'Clara']

print(f"{nome} tem {idade} anos")  # Ana Clara tem 22 anos
```

Tuplas em Python

Características das Tuplas

- Sequência de dados **ordenados**, representadas por **parênteses**.
- Podem conter **diferentes tipos** de dados: inteiros, floats, strings, outras tuplas etc.
- São **imutáveis**: não é possível alterar seus valores após a criação.
- São **indexadas**: acessamos os elementos por índices, como em listas ou strings.
- Suportam **fatiamento** (slicing).

Exemplo

```
tuplanum = (1, 2, 3)
mistura = ('Maria', 10, 3.14, (1, 2))

print(tuplanum[0])    # 1
print(mistura[-1])    # (1, 2)
print(tuplanum[1:])   # (2, 3)
```

Listas em Python

Características das Listas

- Sequência de valores **ordenados**, representada por **colchetes**.
- Suporta **diferentes tipos** de dados: inteiros, floats, strings, outras listas etc.
- São **mutáveis**: é possível alterar, adicionar e remover elementos.
- São **indexadas**, como strings e tuplas.

Exemplo

```
listanum = [1, 2, 3]
listanum[0] = 10
```

Tabela de Índices

Índice +	0	1	2
Elemento	1	2	3
Índice -	-3	-2	-1

Listas em Python - Operações Comuns

Operações com Listas

- **Tamanho** (`len`): número de elementos da lista.
- **Adicionar** (`append`, `insert`): adiciona elementos.
- **Remover** (`remove`, `pop`): exclui elementos.
- **Concatenação** (`+`): junta duas listas.
- **Repetição** (`*`): repete os elementos da lista.
- **Pertencimento** (`in`): verifica se um valor está na lista.
- **Fatiamento** (`:`): retorna sublistas.

Exemplos

```
lista = [1, 2, 3]
len(lista)      # 3
lista.append(4)# [1, 2, 3, 4]
lista.pop()     # [1, 2, 3]
[0] + lista     # [0, 1, 2, 3]
[1] * 3         # [1, 1, 1]
2 in lista      # True
lista[1:3]      # [2, 3]
```

Dicionários em Python

Características dos Dicionários

- **Coleção não ordenada** de dados.
- Representada por { } com pares **chave: valor**.
- Acesso aos valores é feito pelas **chaves**, e não por índices.
- Cada **chave é única** dentro do dicionário.
- Os dicionários são **mutáveis**: é possível alterar, adicionar e remover pares.

Exemplo

```
produtos = {
    'farinha': 3.00,
    'feijão': 5.00,
    'leite': 4.25,
    'açúcar': 2.49 }
print(produtos['leite'])  # 4.25
```

Dicionários em Python - Operações Comuns

Principais Operações

- **Adicionar ou atualizar elementos**: atribuindo um valor a uma nova chave.
- **Pertencimento** (`in`): verifica se uma **chave** está no dicionário.
- **Verificar existência de chave**: com `in` ou usando `get()`.

Exemplos

```
produtos['arroz'] = 4.99
'feijão' in produtos       # True
'macarrão' in produtos     # False
preco = produtos.get('leite')
```

Booleanos em Python

O que são Booleanos?

- Tipo `bool` representa apenas dois valores: `True` e `False`.
- Usado para expressar condições lógicas.
- Muito comum em estruturas de decisão (como `if`).

Operadores de Comparação

- `==` : Igual a
- `!=` : Diferente de
- `>` : Maior que
- `>=` : Maior ou igual a
- `<` : Menor que
- `<=` : Menor ou igual a

Exemplos

```
5 > 3          # True
10 == 20       # False
7 != 4         # True
2 <= 2         # True
```

Booleanos em Python - Operadores Lógicos

Conectando Expressões Booleanas

- `not` – **Não lógico**: inverte o valor lógico.
- `and` – **E lógico**: verdadeiro apenas se as duas expressões forem verdadeiras.
- `or` – **Ou lógico**: verdadeiro se pelo menos uma das expressões for verdadeira.

Exemplos

```
not True        # False
not False       # True

True and True   # True
True and False  # False
```

Mais Exemplos

```
True or False   # True
False or False  # False

x = 5
x > 0 and x < 10      # True
not (x == 5)          # False
```

Estrutura Condicional em Python

Decisões com `if` e `else`

- Em muitas situações, partes do código devem ser executadas apenas se certas condições forem verdadeiras.
- Para isso, usamos expressões booleanas com as estruturas `if` e `else`.
- O bloco `if` só será executado se a condição for `True`.
- Caso contrário, o bloco `else` (se presente) será executado.

Sintaxe

```
if <condição>:
    <bloco if>
else:
    <bloco else>
```

Exemplo de Estrutura Condicional

Problema

Queremos verificar se uma pessoa pode votar. Se a idade for maior ou igual a 18 anos, ela pode votar. Caso contrário, ela não pode votar.

Código em Python

```
idade = int(input("Digite sua idade: "))

if idade >= 18:
    print("Você pode votar.")
else:
    print("Você não pode votar.")
```

Estrutura de Repetição em Python

O que são Loops?

- Permitem executar um trecho de código várias vezes durante a mesma execução.
- Úteis para automatizar tarefas repetitivas.

Tipos de Estruturas de Repetição

- **While:** repete enquanto uma condição booleana for verdadeira.
- **For:** itera sobre elementos de uma coleção (strings, listas, tuplas, etc.).

Exemplos

```
while condicao:
    # bloco de código
for elemento in iteravel:
    # bloco de código
```

Exemplo de Loop `while`

Contando de 1 a 5 com `while`

```
contador = 1

while contador <= 5:
    print(contador)
    contador += 1
```

Exemplo de Loop `for`

Iterando sobre uma lista com `for`

```
nomes = ['Ana', 'Bruno', 'Carlos']

for nome in nomes:
    print(nome)
```

Entrada de Dados em Python

Função `input()`

- Usada para ler dados digitados pelo usuário.
- Sempre retorna uma **string**.
- É necessário converter o valor se quisermos um número.

Exemplos

```
nome = input("Digite seu nome: ")
idade = input("Digite sua idade: ")

# Convertendo para inteiro
idade = int(idade)

# Ou diretamente
idade = int(input("Digite sua idade: "))
```

Saída de Dados em Python

Função `print()`

- Utilizada para **exibir informações** ao usuário.
- Pode mostrar textos, resultados de expressões, valores de variáveis, etc.
- Aceita múltiplos argumentos separados por vírgula.

Exemplos

```
print("Olá, mundo!")
nome = "Maria"
print("Olá,", nome)

idade = 20
print("Idade:", idade)
```

Programação Orientada a Objetos

Prof. Gabriel Rodrigues Caldas de Aquino

gabrielaquino@ic.ufrj.br
Instituto de Computação - Universidade Federal do Rio de Janeiro

Compilado em:
October 20, 2025

Programação Orientada a Objetos

O que é um Objeto?

- Unidade de software que **encapsula dados e algoritmos**.
- Representa conceitos ou entidades do mundo real ou conceitual.
- Relaciona-se com outras entidades, assim como na vida cotidiana.

Por que usar?

- Estruturação clara e organizada do código.
- Especialmente útil para **projetos extensos**.
- Facilita o desenvolvimento de sistemas:
  - Modulares
  - Reutilizáveis
  - De fácil manutenção

Classe em Python

O que é uma Classe?

- Unidade inicial e mínima para código orientado a objetos.
- Abstrai conceitos, definições e comportamentos.
- Descreve atributos (dados) e métodos (ações) dos objetos.
- Objetos são **instanciados** a partir de classes.

Exemplo em Python

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
    def apresentar(self):
        print(f"Olá, meu nome é {self.nome} e tenho {self.idade} anos.")
p1 = Pessoa("Maria", 20)
p1.apresentar()
```

Exemplo em Python - Personagem de RPG

```
class Personagem:
    def __init__(self, vida, forca, inteligencia):
        self.vida = vida
        self.forca = forca
        self.inteligencia = inteligencia
    def atacar(self):
        print("O personagem atacou!")
    def defender(self):
        print("O personagem defendeu!")
    def usar_magia(self):
        print("O personagem lançou uma magia!")
# Criando um objeto (instância) da classe
heroi = Personagem(100, 20, 15)
heroi.atacar()
print("Vida:", heroi.vida)
```

Exemplo com Conceitos de POO

- **Classe**: Projeto de um personagem de RPG.
- **Objeto**: Cada personagem criado com base nesse projeto.
- **Atributos**
  - Vida
  - Força
  - Inteligência
- **Métodos**
  - Atacar
  - Defender
  - Usar magia

Trecho de Código

```
heroi = Personagem(100, 20, 15)
heroi.atacar()
```

Atributos em POO

Definição

- Representam as características do objeto.
- Definidos dentro da classe.
- Armazenam valores que caracterizam o objeto.

Exemplo: Classe `Personagem`

- **vida** – Pontos de vida do personagem.
- **força** – Poder de ataque físico.
- **inteligência** – Capacidade para magia ou estratégias.

Observação

Cada objeto criado a partir da classe `Personagem` terá seus próprios valores para esses atributos.

Métodos em POO

Definição

- São as ações que o objeto pode executar.
- Definidos como funções dentro da classe.
- Podem usar e alterar os atributos do objeto.

Exemplo: Classe `Personagem`

- **atacar()** – Realiza um ataque.
- **defender()** – Executa defesa.
- **usar_magia()** – Lança um feitiço.

Trecho de Código

```
heroi.defender()
heroi.usar_magia()
```

Método Construtor em Python

Definição

- Método especial chamado automaticamente ao criar um objeto.
- Usado para inicializar os atributos da nova instância.
- Em Python, o construtor é o método `__init__()`.
- O construtor:
  - Facilita a inicialização de atributos obrigatórios.
  - Se não for definido, Python usa um construtor default que não faz nada.

Exemplo simples

```
class Personagem:
    def __init__(self, vida, forca):
        self.vida = vida
        self.forca = forca

heroi = Personagem(100, 20)
```

O parâmetro `self` em Métodos

Definição

- Deve ser o primeiro parâmetro em métodos de instância.
- Representa o próprio objeto que chama o método.
- Permite acessar atributos e outros métodos do objeto.
- É obrigatório explicitar no código, mas não na chamada do método.

Exemplo simples

```
class Personagem:
    def apresentar(self):
        print("Eu sou um personagem!")

p = Personagem()
p.apresentar()  # 'self' é passado automaticamente
```

Classe Personagem - Método defender

```
class Personagem:
    def __init__(self, vida, forca, inteligencia):
        self.vida = vida
        self.forca = forca
        self.inteligencia = inteligencia
    def defender(self, dano_recebido):
        dano_final = dano_recebido / 2
        self.vida -= dano_final
        print(f"{self} defendeu e recebeu {dano_final} de dano. Vida restant
    def __str__(self):
        return "Personagem"

heroi = Personagem(100, 20, 15)
heroi.defender(30)
print("Vida do heroi:", heroi.vida)
```

Classe Personagem - Método ganhar_healthpack

```
class Personagem:
    def __init__(self, vida, forca, inteligencia):
        self.vida = vida
        self.forca = forca
        self.inteligencia = inteligencia
    def defender(self, dano_recebido):
        dano_final = dano_recebido / 2
        self.vida -= dano_final
        print(f"{self} defendeu e recebeu {dano_final} de dano. Vida restant
    def ganhar_healthpack(self, vida_restaurada):
        self.vida += vida_restaurada
        print(f"{self} recebeu {vida_restaurada} de Healthpack. Vida restant
    def __str__(self):
        return "Personagem"
heroi = Personagem(100, 20, 15)
heroi.ganhar_healthpack(50)
```

Deixando mais pessoal - Personagem com nome

```
class Personagem:
    def __init__(self, nome, vida, forca, inteligencia):
        self.nome = nome
        self.vida = vida
        self.forca = forca
        self.inteligencia = inteligencia

    def __str__(self):
        return self.nome

heroi = Personagem("Arthur", 100, 20, 15)
```

Por que o `print` mostra o nome do personagem?

O método `__str__()`

- O Python usa o método especial `__str__()` para definir como representar um objeto como texto.
- Quando usamos `print(objeto)`, o Python chama `objeto.__str__()` automaticamente.
- No exemplo, `__str__()` foi definido para retornar o nome do personagem.
- Por isso, o `print(heroi)` exibe o nome ao invés do endereço de memória.

Trecho do código

```
def __str__(self):
    return self.nome
```

Método atacar com interação entre objetos

```
class Personagem:
    def __init__(self, nome, vida, forca, inteligencia):
        self.nome = nome
        self.vida = vida
        self.forca = forca
        self.inteligencia = inteligencia
    def atacar(self, alvo):
        dano = self.forca * 2
        alvo.vida -= dano
        print(f"{self} atacou {alvo} causando {dano} de dano!")
        print(f"Vida restante de {alvo}: {alvo.vida}")
    def __str__(self):
        return self.nome
heroi = Personagem("Arthur", 100, 20, 15)
inimigo = Personagem("Goblin", 80, 15, 10)
heroi.atacar(inimigo)
```

Discussão em Grupo: Modelando Tipos de Dano

Desafio

Como podemos implementar diferentes tipos de dano em nosso jogo?

- Dano pode ser igual à força do personagem.
- Pode ser o dobro (exemplo: fogo contra planta).
- Pode ser metade (exemplo: água contra planta).
- Ou igual para ataques contra o mesmo tipo.

Pontos para pensar

- Como representar os tipos dos personagens e dos ataques?
- Como aplicar multiplicadores diferentes para cada tipo de dano?
- Qual estrutura de dados usar para armazenar essas regras?
- Como modificar o método `atacar` para considerar isso?

Tipos de dano

```
class Personagem:
    def __init__(self, nome, vida, forca, inteligencia, tipo):
        (...)
        self.tipo = tipo  # 'fogo', 'planta', 'agua'
    def atacar(self, alvo):
        if self.tipo == "fogo" and alvo.tipo == "planta":
            dano = self.forca * 2
        elif self.tipo == "agua" and alvo.tipo == "planta":
            dano = self.forca * 0.5
        else:
            dano = self.forca
        alvo.vida -= dano
        print(f"{self} atacou {alvo} causando {dano} de dano!")
        print(f"Vida restante de {alvo}: {alvo.vida}")
heroi = Personagem("Arthur", 100, 20, 15, "fogo")
inimigo = Personagem("Treant", 80, 15, 10, "planta")
```

Exercício: Modelando Formas Geométricas

Desafio: Crie classes em Python para representar formas geométricas básicas.

- Comece com duas formas: `Retangulo` e `Circulo`.
- Cada classe deve ter:
  - Atributos para as dimensões (ex: base e altura para retângulo, raio para círculo).
  - Método para calcular a área.
  - Método para calcular o perímetro (ou circunferência no caso do círculo).
- Crie objetos de cada classe e imprima a área e perímetro.

Exemplo: Classe Retângulo

Classe `Retangulo`

```
class Retangulo:
def init(self, base, altura):
self.base = base
self.altura = altura
def area(self):
    return self.base * self.altura
def perimetro(self):
    return 2 * (self.base + self.altura)
```

Exemplo: Classe Círculo

Classe Circulo

```
class Circulo:
def init(self, raio):
self.raio = raio
def area(self):
    return 3.1416 * self.raio ** 2

def perimetro(self):
    return 2 * 3.1416 * self.raio
```

POO - Herança e Polimorfismo

Prof. Gabriel Rodrigues Caldas de Aquino

gabrielaquino@ic.ufrj.br
Instituto de Computação - Universidade Federal do Rio de Janeiro

Compilado em:
October 20, 2025

Encapsulamento

- **Objetivo**: Esconder os detalhes internos de uma classe e expor apenas o que for necessário por meio de uma interface controlada
- **Ideia**: Os atributos (dados) de um objeto fiquem protegidos contra acessos e modificações indevidas, e que a interação externa aconteça de forma controlada através de métodos

Então, basicamente:

- Você coloca os dados e os métodos que operam sobre esses dados dentro da mesma classe.
- Controla quem pode acessar ou modificar esses dados usando modificadores de acesso (como público, protegido, privado).
- Com isso, aumenta a segurança, reutilização e manutenibilidade do código.

Encapsulamento

Então, na aula passada criamos métodos e os atributos...

- Precisamos agora definir a visibilidade para atributos e métodos.
- Ver como podemos controlar o acesso e a manipulação dos dados da classe.
- Com isso vamos expor apenas o necessário para o uso correto da classe.
- O Fim é proteger a integridade dos dados, escondendo detalhes internos.
  - Usuários sabem o que a classe faz, mas não como ela faz.
  - Permite alterar a implementação sem impactar código externo.

Níveis de acesso (**Convenção do Python!**)

- **Público**: acessível de qualquer lugar.
- **Protegido** (_): acessível na classe e subclasses.
- **Privado** (__): acessível apenas dentro da própria classe.

Métodos Protegidos e Privados em Python

- **Métodos Protegidos (`_método`)**
  - Prefixados com `_`.
  - Indicativo para programadores: não usar fora da classe.
  - **Não impedem** o acesso externo — podem ser chamados normalmente.
  - São uma **convenção** para organizar o código.
- **Métodos Privados (`__método`)**
  - Prefixados com `__`.
  - Python faz **name mangling** para dificultar o acesso externo.
  - Ainda podem ser acessados, mas não diretamente — é desencorajado.
  - Oferecem um nível maior de "proteção" que métodos protegidos.

Atributos e Métodos Públicos

- **Atributos Públicos** Podem ser acessados e modificados diretamente fora da classe.
- **Métodos Públicos** Também podem ser chamados livremente de fora da classe.

Exemplo em Python

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome      # atributo público
        self.idade = idade    # atributo público
    def apresentar(self):     # método público
        print(f"Meu nome é {self.nome} e tenho {self.idade} anos.")
p = Pessoa("João", 30)
print(p.nome)       # acesso direto ao atributo público
p.idade = 31       # modificação direta
p.apresentar()      # chamada do método público
```

Métodos Protegidos

Definição

Métodos protegidos não devem ser acessados ou modificados diretamente fora da classe. **Servem para detalhes internos**. Na **prática, é possível**, mas não foi feito para ser chamado diretamente por código externo.

Exemplo em Python

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

    def _falar(self):   # método protegido (convenção)
        print(f"{self.nome} está falando...")

p = Pessoa("Ana")
p._falar()   # possível, mas não recomendado acessar diretamente
```

Exemplo de Método Protegido

```
class Documento:
    def __init__(self, conteudo):
        self.conteudo = conteudo

    def _validar_tamanho(self):   # protegido
        return len(self.conteudo) > 0

    def salvar(self):             # público
        if self._validar_tamanho():
            print("Documento salvo.")
        else:
            print("Erro: documento vazio.")

    def exportar_pdf(self):        # público
        if self._validar_tamanho():
            print("Relatório exportado em PDF.")
```

Métodos Privados

Definição

Métodos privados são usados para **detalhes internos da classe** que não devem ser acessados ou modificados fora dela. No Python, começam com __ (dois underlines) e sofrem **name mangling**, dificultando o acesso externo.

```
class Conta:
    def __init__(self, saldo):
        self.__saldo = saldo
    def __log_operacao(self, msg):   # método privado
        print(f"[LOG] {msg}")
    def depositar(self, valor):      # método público
        self.__saldo += valor
        self.__log_operacao(f"Depósito de {valor}")
c = Conta(100)
c.depositar(50)
# c.__log_operacao("teste")  # Erro: acesso direto proibido
```

Exemplo de Método Privado em Python

```
class ContaBancaria:
    def __init__(self, saldo):
        self.__saldo = saldo  # atributo privado
    def depositar(self, valor):
        if valor > 0:
            self.__adicionar_saldo(valor)
            print(f"Depositado: {valor}")
    def __adicionar_saldo(self, valor):
        self.__saldo += valor
    def mostrar_saldo(self):
        print(f"Saldo atual: {self.__saldo}")
conta = ContaBancaria(100)
conta.depositar(50)
conta.mostrar_saldo()
# conta.__adicionar_saldo(100)  # Isso gera erro!
conta._ContaBancaria__adicionar_saldo(100)
conta.mostrar_saldo()
```

Name Mangling em Python

O que é Name Mangling?

- Técnica do Python para "esconder" atributos/métodos privados.
- Atributos que começam com `__` (dois underlines) são renomeados internamente.
- Exemplo: `__saldo` vira `_NomeDaClasse__saldo`.
- Evita acesso direto acidental e conflitos em herança.

Importante

- Não é proteção total, apenas dificulta o acesso direto.
- Ainda é possível acessar via `_NomeDaClasse__atributo`, mas não recomendado.
- Indica que o membro é "privado" e não deve ser acessado externamente.

Name Mangling em Python

Exemplo

```
class Conta:
def init(self):
self.__saldo = 100

c = Conta()
print(c._Conta__saldo) # Acesso via name mangling
#(funciona, mas não é recomendado)
```

Herança

Definição

- Herança é o relacionamento entre classes em que uma classe chamada de **subclasse** (ou classe filha) é uma **extensão** ou **subtipo** de outra classe chamada de **superclasse** (ou classe pai/mãe).
- A subclasse consegue **reaproveitar os atributos e métodos** da superclasse. Além do que for herdado, a subclasse pode definir seus próprios membros (atributos e métodos).

Herança e o uso do super()

Relação de Herança

Estabelecemos a relação de herança ao indicar a **superclasse** entre parênteses na definição da subclasse:

```
class Subclasse(Superclasse):
    ...
```

Uso do super()

Utilizamos o método `super()` para acessar a superclasse e, então, **reaproveitar seu construtor ou outros métodos**:

```
class Aluno(Pessoa):
    def __init__(self, nome, cpf, dre):
        super().__init__(nome, cpf)  # chama __init__ de Pessoa
        self.dre = dre
```

O que é `super()`

- `super()` retorna um objeto temporário da superclasse.
- Permite chamar métodos da superclasse sem repetir código.
- Fundamental para reaproveitar atributos e lógica já definida.

Exemplo de Herança: Classe Aluno

```
class Pessoa:
    def __init__(self, nome):
        self.nome = nome
    def print_nome(self):
        print(self.nome)
class Aluno(Pessoa):   # Aluno herda de Pessoa
    def __init__(self, nome, matricula):
        super().__init__(nome)    # chama o construtor da superclasse
        self.matricula = matricula
    def inscrever_disciplina(self, disciplina):
        print(f"{self.nome} inscrito em {disciplina}")
    def exibir_informacoes(self):
        print(f"Nome: {self.nome}")
        print(f"Matrícula: {self.matricula}")
a = Aluno("Maria", "2025001")
a.inscrever_disciplina("Comp 2")
a.exibir_informacoes()
```

Exemplo 2: Pessoa, aluno e professor

```
class Pessoa:
    def __init__(self, nome, cpf):
        self.nome = nome
        self.cpf = cpf
    def print_dados_pessoais(self):
        print(f"Nome: {self.nome}, CPF: {self.cpf}")
class Aluno(Pessoa):
    def __init__(self, nome, cpf, dre):
        super().__init__(nome, cpf)
        self.dre = dre
    def print_dados_aluno(self):
        print(f"Nome: {self.nome}, CPF: {self.cpf}, DRE: {self.dre}")
class Professor(Pessoa):
    def __init__(self, nome, cpf, siape):
        super().__init__(nome, cpf)
        self.siape = siape
    def print_dados_professor(self):
        print(f"Nome: {self.nome}, CPF: {self.cpf}, Siape: {self.siape}")
```

Função isinstance()

Definição

A função `isinstance(objeto, classe)` verifica se um objeto é uma instância de uma classe específica ou de suas subclasses.

- Retorna `True` se o objeto for da classe ou de uma subclasse.
- Retorna `False` caso contrário.

Exemplo em Python

```
class Pessoa:
    pass
class Aluno(Pessoa):
    pass
a = Aluno()
print(isinstance(a, Aluno))   # True
print(isinstance(a, Pessoa))  # True, pois Aluno herda Pessoa
print(isinstance(a, dict))    # False
```

Polimorfismo em POO

Definição

Polimorfismo significa que o **significado de uma operação depende do objeto** em que ela é aplicada. Ou seja, o código não precisa se importar com o tipo exato do objeto, apenas com o que ele faz.

Polimorfismo e Sobrescrita

Na prática, o polimorfismo ocorre principalmente através da **sobrescrita de métodos**:

- Subclasses podem redefinir métodos da superclasse.
- Estabelece uma **interface comum** para todas as classes que herdam da mesma superclasse.

Benefício

Podemos manipular objetos da superclasse de forma genérica, sem nos preocupar com a subclasse exata, sabendo que todos implementam os métodos comuns. O comportamento dos métodos pode variar dependendo da subclasse do objeto.

Exemplo de Polimorfismo: Animais

Definição de classes e métodos

```
class Animal:
    def fazer_som(self):
        print("arrrww")   # som genérico

class Gato(Animal):
    def fazer_som(self):
        print("miau")     # sobrescreve o som

class Cachorro(Animal):
    def fazer_som(self):
        print("auau")    # sobrescreve o som

# Lista de animais
animais = [Animal(), Gato(), Cachorro()]
```

Exemplo de Polimorfismo: Aprovação em Disciplina

```
class Aluno:
    def __init__(self, nome):
        self.nome = nome
    def verifica_aprovacao(self):
        pass  # método genérico, será sobrescrito
class Graduacao(Aluno):
    def verifica_aprovacao(self, nota, frequencia):
        return nota >= 5 and frequencia >= 75
class PosGraduacao(Aluno):
    def verifica_aprovacao(self, conceito, frequencia):
        return conceito in ["A", "B", "C"] and frequencia >= 75
a1 = Graduacao("Maria")
a2 = PosGraduacao("João")
print(a1.verifica_aprovacao(6, 80))
print(a2.verifica_aprovacao("B", 90))
```

POO - Métodos Mágicos

Prof. Gabriel Rodrigues Caldas de Aquino

gabrielaquino@ic.ufrj.br
Instituto de Computação - Universidade Federal do Rio de Janeiro

Compilado em:
October 20, 2025

Métodos Mágicos (Dunder)

Definição

Métodos Mágicos são métodos pré-definidos que podem ser sobrescritos para definir como os objetos de uma classe irão interagir com operadores, funções específicas e instruções da linguagem Python.

Características

- O nome dos métodos mágicos **inicia e termina com dois underlines** (__).
- Também são chamados de **métodos Dunder** ("Double Underscore").
- Permitem personalizar comportamento de operadores e funções nativas, como:
  - __str__ → define como o objeto é convertido em string
  - __add__ → define o comportamento do operador +

Método Mágico __str__

Relembrando

O método `__str__` é um dos métodos mágicos que já vimos e é usado para definir como um objeto será convertido em string. Ele é chamado automaticamente quando usamos o `print` com um objeto.

Exemplo

```
class Pessoa:
    def __init__(self, nome, idade):
        self.nome = nome
        self.idade = idade
    def __str__(self):
        return f"{self.nome}, {self.idade} anos"
p = Pessoa("Maria", 20)
print(p)   # chama automaticamente p.__str__()
# Saída: Maria, 20 anos
```

Alguns Métodos Mágicos (Dunder)

Operadores Aritméticos

`__add__(self, other)`	+
`__sub__(self, other)`	-
`__mul__(self, other)`	*
`__truediv__(self, other)`	/
`__floordiv__(self, other)`	//
`__mod__(self, other)`	%
`__pow__(self, other)`	**

Operadores de Comparação e Condicionais

`__eq__(self, other)`	==
`__ne__(self, other)`	!=
`__lt__(self, other)`	<
`__le__(self, other)`	<=
`__gt__(self, other)`	>
`__ge__(self, other)`	>=
`__contains__(self, other)`	in

Sobrescrita do operador + (__add__)

```
class ContaBancaria:
    def __init__(self, banco, saldo):
        self.banco = banco
        self.saldo = saldo
    def __add__(self, outra):
        return self.saldo + outra.saldo
    def __str__(self):
        return f"{self.banco}: R$ {self.saldo}"
c1 = ContaBancaria("Santander", 1000)
c2 = ContaBancaria("Itau", 500)
print(c1 + c2)   # 1500
```

Sobrescrita do operador > (__gt__)

```
class ContaBancaria:
    def __init__(self, banco, saldo):
        self.banco = banco
        self.saldo = saldo
    def __gt__(self, outra):
        return self.saldo > outra.saldo
    def __str__(self):
        return f"{self.banco}: R$ {self.saldo}"
c1 = ContaBancaria("Bradesco", 1000)
c2 = ContaBancaria("Banco do Brasil", 500)
if c1 > c2:
    print(f"A conta {c1.banco} tem mais dinheiro")
else:
    print(f"A conta {c2.banco} tem mais dinheiro")
```

Alguns Outros Métodos Mágicos (Dunder)

Funções Comuns

`__repr__(self)`	`repr()`
`__str__(self)`	`str()`
`__len__(self)`	`len()`
`__abs__(self)`	`abs()`

Operadores em Objetos Indexáveis

`__getitem__(self, key)`	`x = obj[key]`
`__setitem__(self, key, value)`	`obj[key] = value`
`__delitem__(self, key)`	`del obj[key]`

Instruções Especiais

`__iter__(self)`, `__next__(self)`	`for`
`__enter__(self)`, `__exit__(self, exc_type, exc_value, traceback)`	`with`

Exemplo de __getitem__

Classe que armazena notas de uma disciplina

```
class  Notas_disciplina:
    def __init__(self, disciplina, notas):
        self.disciplina = disciplina
        self.notas = notas  # lista de notas

    def __getitem__(self, indice):
        # permite acessar notas como se fosse uma lista
        return self.notas[indice]

Comp2 = Notas_disciplina("Comp2", [7, 8, 9, 10])

print(Comp2[0])  # 7
print(Comp2[2])  # 9
```

POO - Classe Abstrata

Prof. Gabriel Rodrigues Caldas de Aquino

gabrielaquino@ic.ufrj.br
Instituto de Computação - Universidade Federal do Rio de Janeiro

Compilado em:
October 20, 2025

Classes Abstratas na POO

Um componente do desenvolvimento moderno de software é a **Programação Orientada a Objetos (POO)**, que permite organizar o código de forma **escalável**, **modular** e **reutilizável**.

As **classes abstratas** são um dos conceitos centrais da POO, fundamentais para criar um **modelo** (template) que outras classes podem usar.

Pergunta

O que vocês acham que é uma classe abstrata?

Classe Abstrata em POO

Definição

- Uma **classe abstrata** é uma classe que não pode ser instanciada diretamente.
- Atua apenas como base (superclasse) para outras classes.
- Pode conter:
  - **Métodos implementados**: já possuem código definido.
  - **Métodos abstratos**: devem obrigatoriamente ser sobrescritos nas subclasses concretas.

Objetivo

Garantir que certas operações sejam obrigatoriamente implementadas pelas subclasses, definindo uma interface comum.

O que é uma Classe Abstrata?

Uma **classe abstrata** é como um **molde** (ou **template**) para outras classes. Ela define métodos que devem estar presentes em qualquer classe que herde dela, mas não fornece o código específico desses métodos.

Pense nela como um **esqueleto mas sem dizer como preencher o resto**.

Por que usar Classes Abstratas?

- Definem um **molde** para outras classes.
- Obrigam a implementação de métodos específicos nas subclasses.
- Melhoram a **organização** e a **reutilização** do código.
- Permitem **polimorfismo**, ou seja, diferentes objetos compartilhando uma interface comum.

Exemplo de Classe Abstrata

Suponha que você está construindo um programa para calcular a área de diferentes formas geométricas.

- Você cria uma **classe abstrata** chamada `Forma`, que define que toda forma deve ter um método `area()`.
- Mas `Forma` não especifica como `area()` funciona, pois a fórmula depende do tipo de forma.
- Cada forma específica (como `Circulo` ou `Retangulo`) herda de `Forma` e fornece sua própria implementação de `area()`.

Antes das Classes Abstratas

```
class Forma:
    pass
    def area(self):
      pass
class Retangulo(Forma):
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura
    def area(self):
        return self.largura * self.altura
class Circulo(Forma):
    def __init__(self, raio):
        self.raio = raio
    def area(self):
        import math
        return 3.14 * self.raio**2
```

Depois das classes abstratas

```
from abc import ABC, abstractmethod
# Classe abstrata
class Forma(ABC):
    @abstractmethod
    def area(self):
        pass
    @abstractmethod
    def perimetro(self):
        pass
# Subclasse concreta
class Circulo(Forma):
    def __init__(self, raio):
        self.raio = raio
    def area(self):
        return 3.14159 * self.raio ** 2
    def perimetro(self):
        return 2 * 3.14159 * self.raio
```

Exemplo Completo de Classe Abstrata

```
from abc import ABC, abstractmethod
class Forma(ABC):
    @abstractmethod
    def area(self):
        pass
    @abstractmethod
    def perimetro(self):
        pass
class Circulo(Forma):
    def __init__(self, raio):
        self.raio = raio
    def area(self):
        return 3.14159 * self.raio ** 2
    def perimetro(self):
        return 2 * 3.14159 * self.raio
```

```
class Retangulo(Forma):
    def __init__(self, largura,
    altura):
        self.largura = largura
        self.altura = altura

    def area(self):
        return self.largura
        * self.altura

    def perimetro(self):
        return 2 * (self.largura
        + self.altura)
```

Por que usar Classes Abstratas em Python?

As classes abstratas são úteis quando queremos:

- **Garantir a implementação de métodos**: os métodos abstratos funcionam como um **contrato**, exigindo que cada subclasse forneça sua própria implementação, evitando inconsistências.
- **Estimular a reutilização de código**: classes abstratas podem ter métodos concretos que reduzem duplicação e promovem o princípio DRY (*Do Not Repeat Yourself*).
- **Melhorar legibilidade e manutenibilidade**: fornecem uma estrutura consistente e transparente, facilitando a compreensão e manutenção do código.
- **Permitir polimorfismo**: possibilitam escrever código genérico que funciona com diferentes subclasses, aumentando a extensibilidade e adaptabilidade do software.

Passos para Criar e Usar Classes Abstratas em Python

1. Importar `ABC` e `abstractmethod` do módulo `abc`.
2. Definir a classe abstrata como subclasse de `ABC`.
3. Definir métodos abstratos usando o decorador `@abstractmethod`.
4. Sobrescrever os métodos abstratos nas subclasses não abstratas.

Objetivo

Garantir que todas as subclasses concretas implementem os métodos essenciais definidos na classe abstrata, criando uma interface comum.

Como o módulo abc garante a implementação de métodos

- Qualquer subclasse de uma classe abstrata deve implementar todos os métodos decorados com `@abstractmethod`.
- Se uma subclasse não implementar todos os métodos abstratos, Python impede sua instanciação.
- Um `TypeError` é gerado, ajudando a identificar falhas de implementação cedo.
- Isso garante que todas as subclasses concretas sigam o comportamento e design esperados da classe abstrata.

Especificação da classe abstrata Animal

```
[48] from abc import ABC, abstractmethod
     # Classe abstrata
     class Forma(ABC):
         @abstractmethod
         def area(self):
             pass
     # Subclasse concreta
     class Circulo(Forma):
         def __init__(self, raio):
             self.raio = raio
```

```
0s  c = Circulo(1)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipython-input-2633766872.py in <cell line: 0>()
----> 1 c = Circulo(1)

TypeError: Can't instantiate abstract class Circulo without an implementation for abstract method 'area'
```

Exemplo: Erro ao não implementar métodos abstratos

```
class Retangulo(Forma):
    def __init__(self, largura, altura):
        self.largura = largura
        self.altura = altura

    # Métodos area() e perimetro() não implementados

# Tentativa de instanciar
r = Retangulo(5, 10)
# TypeError:
# Can't instantiate abstract class Retangulo
# with abstract methods area, perimetro
```

Resumo do Módulo ABC

O módulo **abc** oferece suporte nativo para classes abstratas.

- "ABC" significa **Abstract Base Classes**.
- Fornece ferramentas como a classe `ABC` e o decorador `@abstractmethod`.
- Permite definir métodos abstratos que obrigam a implementação nas subclasses.
- Possibilita implementar um esqueleto de funcionalidades comuns.

Classe Abstrata: Papel do `@abstractmethod`

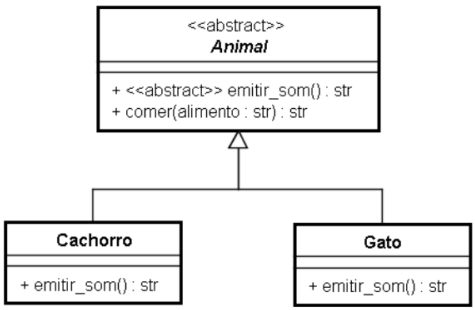
Função do `@abstractmethod`

- Indica que o método é **abstrato** e deve ser implementado em subclasses concretas.
- Garante que todas as subclasses forneçam sua própria implementação.
- Permite que a classe abstrata defina uma **interface comum**.

Consequência

Se uma subclasse de `Animal` não implementar `emitir_som()`, o Python gera um erro de instância: `TypeError: Can't instantiate abstract class X with abstract methods emitir_som`

Especificação da classe abstrata Animal



Classe Abstrata: Animais - Definição

```
from abc import ABC, abstractmethod
class Animal(ABC):
    def __init__(self, nome):
        self.nome = nome
    @abstractmethod
    def emitir_som(self):
        pass
class Cachorro(Animal):
    def emitir_som(self):
        print(f"{self.nome} faz Au Au!")
class Gato(Animal):
    def emitir_som(self):
        print(f"{self.nome} faz Miau!")
class Pato(Animal):
    def emitir_som(self):
        print(f"{self.nome} faz Arrrww!")
```

Classe Abstrata: Animais - Uso e Saída

```
c = Cachorro("Rex")
g = Gato("Mimi")
p = Pato("Donald")

c.emitir_som()  # Rex faz Au Au!
g.emitir_som()  # Mimi faz Miau!
p.emitir_som()  # Donald faz Arrrww!
```

Saída Esperada

Rex faz Au Au!
Mimi faz Miau!
Donald faz Arrrww!

POO - Herança Múltipla

Prof. Gabriel Rodrigues Caldas de Aquino

gabrielaquino@ic.ufrj.br
Instituto de Computação - Universidade Federal do Rio de Janeiro

Compilado em:
October 20, 2025

Herança Múltipla em Python - Conceito

Definição

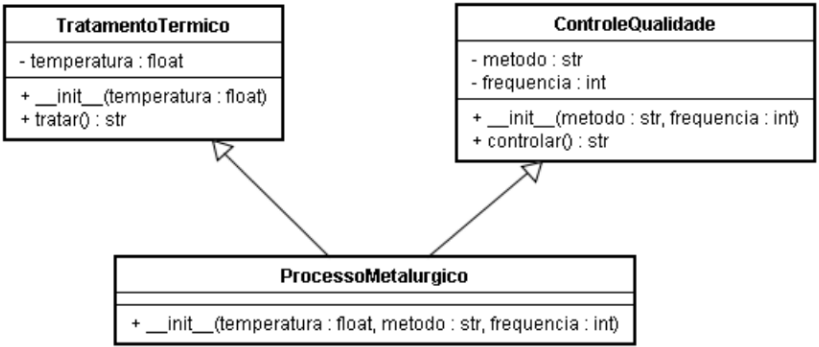
Em Python, uma classe pode herdar de mais de uma classe base: `class Cachorro(Animal, Mamifero):`
A classe derivada herda todos os atributos e métodos (públicos e protegidos) de todas as classes base.

Herança Múltipla em Python - Conceito

Ordem de Resolução de Métodos (MRO)

- O método `mro()`, herdado de `object`, mostra a Ordem de Resolução de Métodos (Method Resolution Order - MRO) de uma classe.
- Se múltiplas classes base possuem um método ou atributo com o mesmo nome, o da classe mais à esquerda na herança prevalece.

Herança Múltipla



Herança Múltipla - Cenário: Processos Metalúrgicos

Um processo metalúrgico envolve várias etapas, e duas delas são:

- **Tratamento térmico:** define como o material é aquecido e resfriado para alterar suas propriedades (dureza, resistência, etc.).
- **Controle de qualidade:** garante que o produto final atenda aos padrões, fazendo medições, testes e inspeções.

`TratamentoTermico` e `ControleDeQualidade` são classes independentes:

- Representam aspectos diferentes e podem ser aplicados em outros contextos.

`ProcessoMetalurgico` herda dessas duas:

- Processo metalúrgico envolve tanto tratamento térmico quanto validação de qualidade.

A herança múltipla:

- Permite que a classe final tenha os métodos e atributos de ambas, compondo o comportamento necessário.

Implementação do exemplo - Processos Metalúrgicos

```python
class TratamentoTermico:
    """Classe base 1"""
    def __init__(self, temperatura: float):
        self.__temperatura = temperatura

    def tratar(self) -> str:
        return f"Executando tratamento térmico a {self.__temperatura:.1f}°C ..."

class ControleQualidade:
    """Classe base 2"""
    def __init__(self, metodo: str, frequencia: int):
        self.__metodo = metodo
        self.__frequencia = frequencia

    def controlar(self) -> str:
        return f"Controle de qualidade, usando o método {self.__metodo}, " \
               f"{self.__frequencia} vezes por dia."

class ProcessoMetalurgico(TratamentoTermico, ControleQualidade):
    """Classe derivada que usa herança múltipla"""
    def __init__(self, temperatura: float, metodo: str, frequencia: int):
        # Inicializa a classe TratamentoTermico
        TratamentoTermico.__init__(self, temperatura)
        # Inicializa a classe ControleQualidade
        ControleQualidade.__init__(self, metodo, frequencia)
```

Herança Múltipla em Python - Como Funciona

Funcionamento

- Uma classe derivada pode herdar de mais de uma classe base.
- Todos os atributos e métodos (públicos e protegidos) das classes base são herdados.
- Quando várias classes base possuem métodos ou atributos com o mesmo nome:
  - O método ou atributo da classe mais à esquerda na definição da herança é usado.
- O Python utiliza a **Ordem de Resolução de Métodos (MRO)** para determinar qual método será chamado.
- O método `mro()` da **classe** mostra essa ordem.

Herança Múltipla em Python - Resumo

- Herança múltipla permite reaproveitar código de diversas classes
- **Importante**: entender a MRO para evitar comportamentos inesperados quando houver métodos com o mesmo nome.

Herança Múltipla em Python - Exemplo Prático

```
class A:
    def metodo1(self):
        print("Metodo1 - Classe A")
    def metodoA2(self):
        print("MetodoA2 - Classe A")
class B:
    def metodo1(self):
        print("Metodo1 - Classe B")
    def metodoB2(self):
        print("MetodoB2 - Classe B")
class C(A, B):
    pass
c = C()
c.metodo1()
c.metodoA2()
c.metodoB2()
print(C.mro())  # Mostra a ordem de herança
```

Entendendo a saída

```
Metodo1 - Classe A
MetodoA2 - Classe A
MetodoB2 - Classe B
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```

Explicação detalhada:

- Python segue a ordem do MRO: C → A → B
- Encontra metodo1() primeiro na Classe A
- c.metodoA2() → só existe na Classe A, então o Python encontra e executa normalmente
- c.metodoB2() → só existe na Classe B, então o Python encontra e executa normalmente
- C.mro() → Mostra a ordem de resolução de métodos

E se eu quiser executar em uma ordem diferente?

```
class CLASS_A:
    nome = "Classe A"
    def metodo1(self):
        print(f"Classe A fala: {self.nome}")
class CLASS_B:
    nome = "Classe B"
    def metodo1(self):
        print(f"Classe B fala: {self.nome}")
class CLASS_C(CLASS_A, CLASS_B):
    nome = "Classe C"
    pass
var_a = A()
var_b = B()
var_c = C()
var_a.metodo1()
var_b.metodo1()
var_c.metodo1()
print(CLASS_C.mro())
CLASS_B.metodo1(var_c)
```

Herança Múltipla – método `super()`

Uso de `super()`

- O método `super()` permite chamar métodos ou construtores da superclasse.
- Em herança múltipla, o `super()` segue a Ordem de Resolução de Métodos (MRO).

Recomendação em herança múltipla

- Se os métodos das superclasses tiverem parâmetros diferentes, pode haver conflito.
- Nesses casos, é mais seguro usar a chamada explícita: `NomeDaSuperClasse.metodo(self, ...)` ao invés de `super().metodo(...)`.
- Isso garante que cada classe receba os parâmetros corretos e evita erros de inicialização.
- `super()` é útil para reutilizar métodos da superclasse.
- Em herança múltipla complexa, chamadas explícitas podem ser mais claras e seguras.

Herança Múltipla - Exemplo Prático: Hidrocarro

```
class Carro:
    def __init__(self, rodas):
        self.rodas = rodas
    def dirigir(self):
        print(f"Dirigindo o carro com {self.rodas} rodas.")
class Barco:
    def __init__(self, tamanho):
        self.tamanho = tamanho
    def navegar(self):
        print(f"Navegando no barco de {self.tamanho} metros.")
class Hidrocarro(Carro, Barco):
    def __init__(self, rodas, tamanho):
        Carro.__init__(self, rodas)
        Barco.__init__(self, tamanho)
    def mostrar_info(self):
        print(f"Hidrocarro com {self.rodas} rodas e {self.tamanho} metros de comprimento.")
h = Hidrocarro(4, 7)
h.dirigir()
h.navegar()
h.mostrar_info()
```

Tratamento de Exceções

Prof. Gabriel Rodrigues Caldas de Aquino

gabrielaquino@ic.ufrj.br
Instituto de Computação - Universidade Federal do Rio de Janeiro

Compilado em:
October 20, 2025

Afinal qual o motivo de tratarmos exceções?

Antes de começarmos o nosso assunto, uma reflexão...

Qual o efeito de um código que não funciona corretamente?

Veja o caso to THERAC-25

Erro de Sintaxe Vs. Exceção

Pontos de observação

- **Erros de Sintaxe**:
  - Detectados antes da execução
  - Exemplo: esquecer dois-pontos (`:`) em um `if`
  - Mensagem mostra o local aproximado do erro antes de executar o código
- **Exceções**:
  - Ocorrem durante a execução do código
  - Representam erros lógicos ou condições inesperadas
    - Exemplos: divisão por zero, tipo incorreto
  - Mostram tipo da exceção e (ex: `ZeroDivisionError`) encerram a execução do código

Erro de Sintaxe Vs. Exceção

Principais Diferenças

- **Erro de Sintaxe**:
  - Impede a execução do código
  - Ou seja o código não roda
  - Precisa ser corrigido antes de executar
- **Exceção**:
  - Ocorre **durante** a execução do código
  - Pode ser tratada com `try-except`
  - Caso seja tratada, o funcionamento do código pode ser recuperado

Exemplos de Erros de Sintaxe

Exemplos de Erros de Sintaxe

- **Esquecer os dois-pontos**:

```
if x > 5   # ERRO: falta ':'
    print("Maior que 5")
```

- **Parênteses não fechados**:

```
print("Olá, mundo"   # ERRO: falta ')'
```

Exemplos de Exceções em Python

Exemplos Práticos

- **ZeroDivisionError**:

```
10 / 0   # Tenta dividir por zero
```

- **NameError**:

```
print(var_inexistente)  # Variável não definida
```

- **TypeError**:

```
"2" + 2  # Concatenação de tipos incompatíveis
```

- **IndexError**:

```
lista = [1, 2]
lista[3]  # Acesso a índice inexistente
```

Mas o que é uma exceção?

Definição

Uma exceção é um acontecimento inesperado ou incomum no fluxo normal do código.

Características principais

- Situações que fogem do comportamento esperado do código
  - Podemos prever ou não
- Códigos podem lançar exceções intencionalmente
- `ZeroDivisionError` e `ValueError` são exemplos de exceções

Então, exceção é isso!

Pontos principais

- **Interrompem** no fluxo normal do programa
- Ocorrem quando algo **inesperado** acontece
- Exemplos:
  - Tentar abrir um arquivo que não existe
  - Dividir um número por zero
  - Acessar uma posição inválida em uma lista

Por que tratar exceção é importante?

- Permitem **recuperar** o programa de erros
- Evitam que o programa **trave** completamente
- Oferecem **feedback** útil para depuração

Entendendo as Mensagens de Erro

Mensagem de Erro

- **Traceback**:
  - Histórico que levou ao erro
- **Localização**:
  - `path/to/file.py`
  - Linha 3: `c=a/b`
- **Tipo da Exceção**:
  - Classe do erro
  - ex: `ZeroDivisionError`
- **Descrição**:
  - Explicação legível do problema
  - ex: "division by zero"

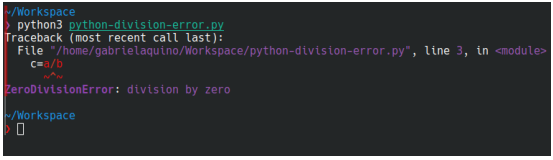


Figure: Exemplo de mensagem

Try/Except

Código Original

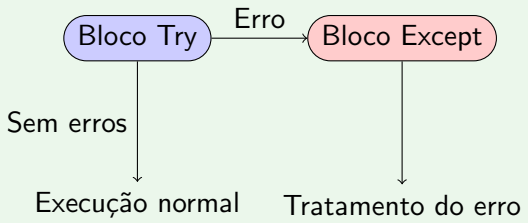
```
a = 10
b = 0
c = a / b  # Problema aqui
print(c)
```

Versão Corrigida com Try/Except

```
a = 10
b = 0

try:
    c = a / b
except ZeroDivisionError:
    print("Erro: Divisão por zero")
    c = float('inf') # Valor padrão
```

Fluxo de Execução

Fluxo de Execução



Como podemos fazer o tratamento?

- Especificar o tipo de exceção
- Registrar que ocorreu um problema
- Definir valores padrão caso tenha um problema

Exemplo de tratamento

Exemplo de tratamento

```
try:
    print("Início do bloco try")
    x = 10 / 0  # O problema ocorre aqui
    print("Esta linha NUNCA será executada")
except ZeroDivisionError:
    print("Erro capturado - divisão por zero")
```

Qual problema temos aqui?

Código Vulnerável

```
idade = int(input("Digite sua idade: "))
if idade >= 18:
    print("Maior de idade")
else:
    print("Menor de idade")
```

O que pode dar errado?

Porque esse código é vulnerável ?

Problema com Entrada do Usuário

Código Vulnerável

```
idade = int(input("Digite sua idade: "))
if idade >= 18:
    print("Maior de idade")
else:
    print("Menor de idade")
```

O que pode dar errado?

- ValueError: Se usuário digitar "dez" em vez de 10

Solução com Tratamento de Exceções

Código com o tratamento de exceção

```
try:
    idade = int(input("Digite sua idade: "))
    if idade >= 18:
        print("Maior de idade")
    else:
        print("Menor de idade")
except ValueError:
    print("Por favor, digite apenas números!")
```

O que pode dar errado neste codigo?

Discutam o código abaixo

```
num = int(input("Número: "))
print(f"Resultado: {100 / num}")
```

Riscos levantados

Pontos de risco

```
num = int(input("Número: "))            # Risco 1: Valor não inteiro
print(f"Resultado: {100 / num}")        # Risco 2: Divisão por zero
```

- ValueError:
  - Entradas como "vinte", "a"...
- ZeroDivisionError:
  - Divisão por zero caso num=0

Solução de Tratamento

Código com Tratamento de Erros

```
try:
    num = int(input("Número: "))
    print(f"Resultado: {100 / num}")
except ValueError:
    print("Digite um número inteiro válido!")
except ZeroDivisionError:
    print("Não pode ser zero!")
```

Exemplo de Execuções

- Entrada "10" → Resultado: 10.0
- Entrada "0" → "Não pode ser zero!"
- Entrada "abc" → "Digite um número inteiro válido!"

O Bloco `else` no Tratamento de Exceções

Quando usar?

O bloco `else` executa **somente se**:

- O bloco `try` for concluído **sem erros**
- Nenhuma exceção foi levantada

Diferença entre fluxos

```
try:
    # Código que pode falhar
except MinhaExcecao:
    # Executa SE ocorrer erro
else:
    # Executa SE NÃO ocorrer erro
finally:
    # Executa SEMPRE
```

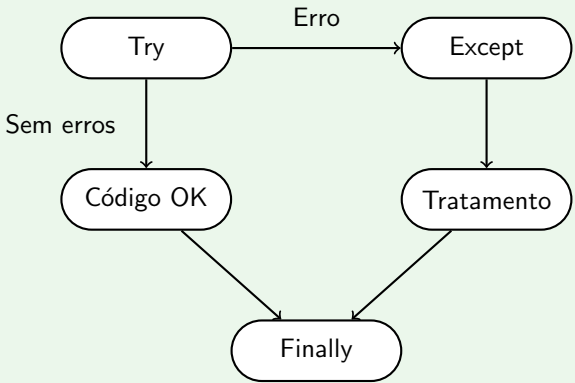
O bloco `finally`

Características do `finally`

- Sempre executa, independentemente:
  - Se ocorrer erro **ou não**
  - Se o erro foi tratado **ou não**
  - Se houve `return` no bloco
- Uso típico para:
  - Liberar recursos (arquivos, conexões)
  - Fazer limpeza
  - Registrar/logging de operações

Fluxo Try-Except-Finally

Fluxo de Execução com Tratamento de Exceções com o finally



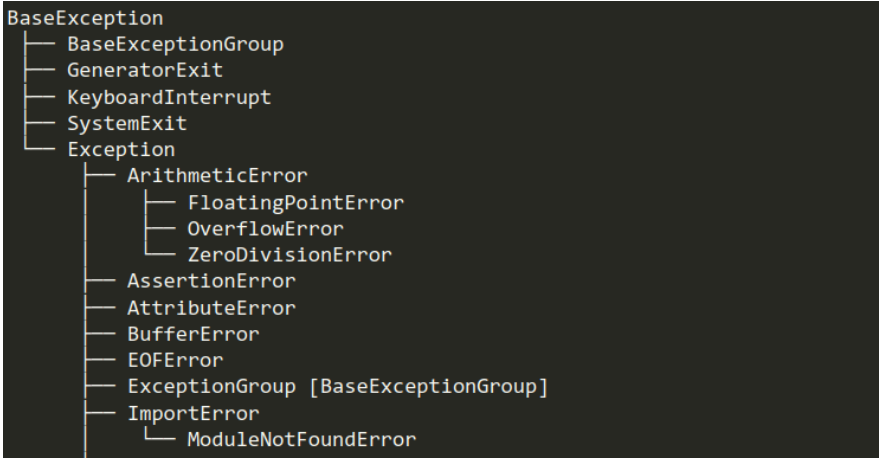
O Bloco `finally` em Python

Funcionamento Básico

```
arquivo = None
try:
    arquivo = open("dados2.txt", "r")
    # Operações com o arquivo
except FileNotFoundError:
    print("Arquivo não encontrado!")
finally:
    print("Sempre executa")
    if arquivo != None:
        arquivo.close()  # Garante o fechamento
    else:
        print(arquivo)
```

Hierarquia de Exceções em Python

Todas as exceções herdam de `BaseException`. Veja "Exception hierarchy" em `https://docs.python.org/3/library/exceptions.html`



Capturando Exceções Genéricas

Exemplo Prático

```
try:
    num = int(input("Digite um número: "))
    resultado = 100 / num
    print(f"Resultado: {resultado}")

except Exception as err:
    print(f"Ocorreu um erro: {type(err).__name__}")
    print(f"Mensagem: {str(err)}")
    print(f"Detalhes completo: {err}")
```

Tratando Diferentes Tipos de Exceções

Exemplo Completo

```
try:
    num = int(input("Digite um número (não zero): "))
    resultado = 100 / num
    print(f"Resultado: {resultado:.2f}")

except ZeroDivisionError:
    print("Erro: Não é possível dividir por zero!")
except ValueError:
    print("Erro: Digite apenas números inteiros!")

except BaseException as err:
    print()
    print(f"Ops! {type(err).__name__}")
    print(f"Fim!")
```

Exceções Personalizadas

Como criar uma exceção básica

```
class MeuErroCustomizado(Exception):
    pass

raise MeuErroCustomizado("Mensagem de erro especial")
```

Exemplo Prático

```
try:
    raise MeuErroCustomizado("Algo deu errado!")
except MeuErroCustomizado as erro:
    print(f"Erro capturado: {erro}")
```

Saída:

```
Erro capturado:  Algo deu errado!
```

Verificação de Triângulo Equilátero

Código que verifica se os lados formam um triângulo equilátero

```
def verifica_equilatero(triangulo):
    if triangulo[0] == triangulo[1] == triangulo[2]:
        return True
    else:
        raise ValueError("Não é um triângulo equilátero!")

try:
    lados = [5, 5, 5]
    if verifica_equilatero(lados):
        print("É um triângulo equilátero!")
except ValueError as e:
    print(f"Erro: {e}")
```

Exceções - Criando exceções personalizadas

Quando usar?

Usada quando precisamos definir nossos próprios tipos de erro para tornar o código mais legível e facilitar o tratamento de erros específicos.

Passo a passo para criação

1. Criar uma classe que herda de `Exception`
2. Definir um construtor (`__init__`) para personalizar a exceção
3. Levantar a exceção (`raise`) no código
4. Capturar a exceção (`except`) e tratá-la

Exceções - Criando exceções personalizadas

1. Criar uma classe que herda de Exception

Cada exceção personalizada deve ser uma classe que herda da classe Exception. Isso garante que ela tenha o comportamento de uma exceção normal.

Exemplo Básico

```
class SaldoInsuficienteError(Exception):
    """Exceção para indicar que o saldo da conta é insuficiente."""
    pass
```

`SaldoInsuficienteError` já funciona como uma exceção, mas ainda não tem uma mensagem personalizada.

Exceções - Criando exceções personalizadas

2. Definir um construtor (__init__) para personalizar a exceção

Podemos adicionar um construtor para aceitar parâmetros e definir uma mensagem de erro.

Exemplo com construtor personalizado

```
class SaldoInsuficienteError(Exception):
    """Exceção para saldo insuficiente."""
    def __init__(self, saldo, valor,
                 mensagem="Saldo insuficiente para a operação."):
        self.saldo = saldo
        self.valor = valor
        self.mensagem = f"{mensagem} Saldo atual: R${saldo:.2f}, valor
solicitado: R${valor:.2f}."
        super().__init__(self.mensagem)
```

3. Levantar a exceção (`raise`) no código

Agora podemos usar `raise` para lançar essa exceção em uma função.

Exemplo de uso com `raise`

```
def sacar(saldo, valor):
    if valor > saldo:
        raise SaldoInsuficienteError(saldo, valor)
    saldo -= valor
    return saldo
```

4. Capturar a exceção (except) e tratá-la no código

Como qualquer outra exceção, podemos capturá-la com `try-except`

Exemplo completo de tratamento

```
try:
    saldo_atual = 100.0
    novo_saldo = sacar(saldo_atual, 200.0)
    print(f"Saque realizado! Novo saldo: R${novo_saldo:.2f}")
except SaldoInsuficienteError as e:
    print(f"Erro: {e}")
```

Criando Exceções Personalizadas

```
class SaldoInsuficienteError(Exception):
    def __init__(self, saldo, valor, mensagem="Saldo insuficiente para a
    operação."):
        self.saldo = saldo
        self.valor = valor
        self.mensagem = f"{mensagem} Saldo atual: R${saldo:.2f},
        valor solicitado: R${valor:.2f}."
        super().__init__(self.mensagem)
def sacar(saldo, valor):
    if valor > saldo:
        raise SaldoInsuficienteError(saldo, valor)
    saldo -= valor
    return saldo
try:
    saldo_atual = 100.0
    novo_saldo = sacar(saldo_atual, 20.0)
    print(f"Saque realizado! Novo saldo: R${novo_saldo:.2f}")
except SaldoInsuficienteError as e:
```

Dicas e melhores práticas

- Não use *except Exception* de forma genérica.
- Não use *except* sem nenhuma exceção.
- Para pegar uma exceção e tratar, precisa que o código que você queira testar esteja dentro do *try*
- Não sabe qual exceção pegar? Leia a documentação do python
  - Google é seu amigo, procure por *"python exception list"*
  - ou vá na URL: `https://docs.python.org/3/library/exceptions.html`

Persistência de Dados

Prof. Gabriel Rodrigues Caldas de Aquino

gabrielaquino@ic.ufrj.br
Instituto de Computação - Universidade Federal do Rio de Janeiro

Compilado em:
October 20, 2025

Persistência de Dados

Problema

Os dados gerados em uma execução do código são perdidos quando o programa é encerrado.

Solução

Se precisamos usar os dados no futuro, é necessário armazená-los de forma persistente.

Mecanismos de Persistência de Dados:

- Arquivos (TXT, CSV, JSON, XML)
- Bancos de dados (SQLite, PostgreSQL, MySQL)

Persistência em Arquivos de Texto

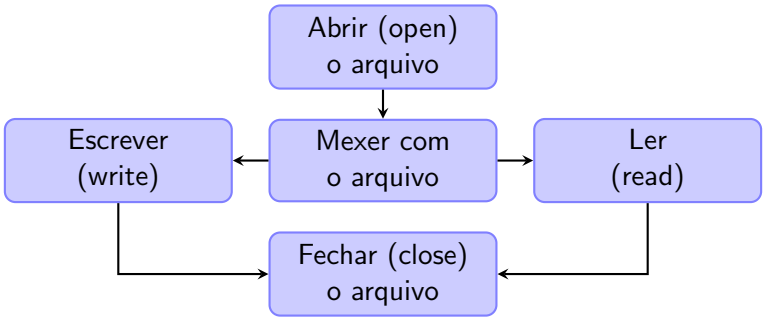
Características

- Formatos comuns: `.txt`, `.csv`, `.json`
- Armazenamento disco rígido
- Manipulação via objetos da classe `File`

Para manipular temos um "protocolo" de Uso

Três etapas: **Abrir**, **Manipular** e **Fechar**

Fluxo para se fazer a manipulação de arquivos



Abrindo Arquivos em Python

Método `open()`

Usado para abrir um arquivo. Requer dois parâmetros principais:

```
arquivo = open("nome_do_arquivo.txt", "modo_de_abertura")
```

Parâmetros

- **Nome do arquivo**:
  - Caminho completo ou relativo
- **Modo de abertura**:
  - `"r"` - Leitura
  - `"w"` - Escrita

Exemplos

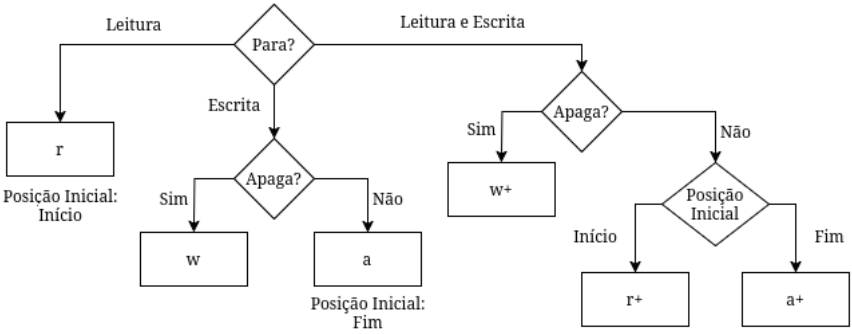
```
# Para escrita
arq = open("dados.txt", "w")

# Para leitura
arq = open("dados.txt", "r")
```

Modos de Abertura de Arquivos

O modo de abertura define como interagiremos com o arquivo:

Modo	**Descrição**	**Existência do Arquivo**
`r`	Leitura apenas	Deve existir
`w`	Escrita	Cria se não existir
`a`	Escrita no final	Cria se não existir
`r+`	Leitura e escrita	Deve existir
`w+`	Leitura e escrita	Cria se não existir
`a+`	Leitura e escrita no final	Cria se não existir

Modos de Abertura de Arquivo



Abrindo Arquivos - Modo de abertura 'r'

- Método read: ler dados de um arquivo que foi previamente aberto para leitura
- Para abrir um arquivo para leitura:
  - Precisamos dar o nome de um arquivo existente
  - Em seguida indicar o modo de leitura *r* no momento da abertura

Abrindo o arquivo dados.txt

```
arquivo = open("dados.txt", "r")
conteudo = arquivo.read()
print(conteudo)
arquivo.close()
```

Lendo linha por linha

- Em arquivos com múltiplas linhas podemos usar for loop para facilitar o tratamento linha por linha

Exemplo de leitura linha por linha

```
arquivo="dados.txt"
arq = open(arquivo, "r")
numero_da_linha = 1
for linha in arq:
    print(f"Linha {numero_da_linha}: {linha}")
    numero_da_linha = numero_da_linha + 1
arq.close()
```

Escrever em Arquivo: write() - Modo de abertura 'w'

Método write()

```
arq.write("conteúdo que será escrito no arquivo")
```

Passos para escrita em arquivo

1. Abrir o arquivo em modo de escrita (`'w'`)
2. Passar o conteúdo como string para `write()`
3. Fechar o arquivo

Escrita Básica em Arquivos

Exemplo de escrita em arquivo

```
arq = open("nomes.txt", "w")
arq.write("Gabriel, Pedro, Manoel")
arq.close()
```

Importante!

- `'w'` = write (escrita)
- Sempre feche o arquivo após escrever
- Cada `write()` grava o conteúdo exato
- Se o arquivo "nomes.txt" **já existe**, ele será **sobrescrito**

Arquivos com mais de uma linha

Leitura de Arquivos Linha por Linha

- Arquivos com múltiplas linhas contêm `\n` escondido

Exemplo Prático

Pedro\n
Antonio\n
Maria\n
Jose\n

Pulando linha na escrita em arquivos

Código de exemplo

```
arq = open("nomes.txt", "w")
arq.write("Gabriel\nPedro\nManoel")
arq.close()
```

O que acontece?

- `\n` é o **caractere especial** para quebra de linha
- Quando escrito no arquivo, ele:
  - Finaliza a linha atual
  - Move o cursor para a próxima linha

Fechando o Arquivo: close()

Método close()

```
arq.close()  # Fecha o arquivo após uso
```

Por que fechar arquivos?

- **Libera recursos do sistema**: Arquivos abertos consomem memória
- **Garante a escrita completa**: Dados podem ficar em buffer
- **Evita corrupção**: Previne acesso concorrente indevido
- **Libera o arquivo**: Permite que outros programas o acessem

Modo de Abertura 'a' (Append)

Funcionamento do modo "a"

```
arquivo = open("dados.txt", "a")  # Modo append
arquivo.write("Novo conteúdo\n")
arquivo.close()
```

Características

- **Abre para escrita** no final do arquivo
- Ponteiro no **fim do arquivo** (só escreve no final)

Gerenciamento de Arquivos com `with open`

Sintaxe Básica

```
with open("arquivo.txt", "modo") as arquivo:
    # Bloco de código
```

Exemplo Prático

```
with open("dados.txt", "r") as arq:
    conteudo = arq.read()
    print(conteudo)
```

Facilidade

- O `with` deixa arquivo aberto.
- Ao sair do bloco, o `close()` é automático.

Controlando a Posição com `seek()`

O que é `seek()`?

Método que permite mover o "cursor" de leitura/escrita para qualquer posição no arquivo

```
arquivo.seek(offset)
```

Parâmetros

- **offset**: Número de bytes para mover

Exemplos

```
# Ir para o byte 10
arquivo.seek(10)
```

Encontrando um conteúdo no arquivo

Como fazer para achar um conteúdo no arquivo?

Podemos achar um conteúdo com usando o **find**

```
f = open("vinyl_sales.txt", "r+")
conteudo = f.read()
pos = conteudo.find("achado")
print(pos)
f.seek(pos)
f.write("Encontrei")
f.close()
```

Nesse caso

Nós lemos o arquivo, depois encontramos a palavra desejada com o find e temos a posição correta na variável **pos**

Modo de Abertura r+

Funcionamento do modo "r+"

```
with open("arquivo.txt", "r+") as arquivo:
    # Operações de leitura E escrita
    conteudo = arquivo.read()  # Lê
    arquivo.write("novo texto")  # Escreve
```

Características

- Permite **leitura e escrita** no mesmo arquivo
- Não apaga o arquivo na hora de abrir
- Posição inicial: **início do arquivo**

Modo de Abertura w+ (Escrita e Leitura)

Funcionamento do modo "w+"

```
with open("arquivo.txt", "w+") as arquivo:
    arquivo.write("Conteúdo inicial\n")
    arquivo.seek(0)  # Volta ao início para leitura
    conteudo = arquivo.read()
    print(conteudo)
```

Características Principais

- **Apaga conteúdo existente**
- Posição inicial: **início do arquivo**

Cuidado!

Sempre use `seek()` antes de ler após escrever

Modo de Abertura a+ (Append e Leitura)

Funcionamento do modo "a+"

```
with open("arquivo-teste.txt", "a+") as arquivo:
    arquivo.write("Nova entrada\n")  # Escreve no FINAL
    arquivo.seek(0)                  # Volta ao início
    texto = arquivo.read()       # Lê todo conteúdo
    print(texto)
```

Características

- **Não apaga** conteúdo existente
- Posição inicial: **Final do arquivo**

Tratamento de Exceções com Arquivos

Por que tratar exceções?

Lidar com arquivos pode lançar exceções inesperadas que devem ser gerenciadas.

Principais exceções com arquivos

- `FileNotFoundError`: Arquivo não existe ou caminho incorreto
- `IOError`: Erros gerais de entrada/saída (disco cheio, permissões)

Tratamento de Exceções com Arquivos

Código de Exemplo

```
try:
    f = open("arquivo-inexistente.txt", "r")
    conteudo = f.read()
    f.close()
    print(conteudo)
except FileNotFoundError:
    print("Arquivo não existente")
```

Demonstração: Tratando Arquivo Corrompido

Código de Tratamento

```
try:
    f = open("arquivo-corrompido.txt", "r")
    conteudo = f.read()
    f.close()
    print(conteudo)
except IOError:
    print("Erro: Problema ao ler o arquivo")
```

Biblioteca Numpy

Prof. Gabriel Rodrigues Caldas de Aquino

gabrielaquino@ic.ufrj.br
Instituto de Computação - Universidade Federal do Rio de Janeiro

Compilado em:
October 20, 2025

NumPy - Introdução

O que é NumPy?

- Pacote fundamental para **computação científica** em Python

Objeto principal: `numpy.ndarray`

- Vetor **n-dimensional** (arrays multidimensionais)
- Características fundamentais:
  - **Tamanho fixo** (definido na criação)
  - **Indexado** por tuplas de inteiros positivos
  - **Homogêneo** - todos elementos do mesmo tipo

Documentação Oficial

Há diversas métodos e atributos disponíveis no NumPy, os quais podem ser consultados na documentação oficial no endereço: `https://numpy.org/doc/stable/index.html`

NumPy - Arrays

Importe o módulo e crie arrays a partir de listas

```
import numpy as np
np.array(lista)
```

Exemplos

```
# Array 1D (vetor)
x = np.array([1, 2, 3])


# Array 2D (matriz)
y = np.array([[1., 0., 0.],
              [0., 1., 0.]])
```

Saída dos Exemplos

```
#Saída:
>>> x
array([1, 2, 3])

>>> y
array([[1., 0., 0.],
       [0., 1., 0.]])
```

Tipos de Numpy array

```
>>> type(x)
numpy.ndarray

>>> type(y)
 numpy.ndarray
```

Características

- `x` é um array **1-dimensional** (vetor)
- `y` é um array **2-dimensional** (matriz)
- Ambos são do tipo `numpy.ndarray`

NumPy - O Atributo dtype

Definição

O dtype define o tipo dos elementos armazenados no array NumPy

Exemplo Inicial

```
minhalista = [1, 2, 3, 4, 5]
arr = np.array(minhalista)
print(arr)          # array([1, 2, 3, 4, 5])
print(type(arr))    # <class 'numpy.ndarray'>
print(arr.dtype)    # dtype('int64')
```

Com Número Decimal

```
minhalista = [1, 2, 3, 4, 5.5]
arr = np.array(minhalista)
print(arr.dtype)  # dtype('float64')
```

Com String

```
minhalista = [1, 2, 3, 4, "ola"]
arr = np.array(minhalista)
print(arr.dtype)  # dtype('<U21')
```

NumPy - Propriedades de Arrays

Criação de Array 2D

```
arr2 = np.array([[1, 2],
                 [3, 4]])
```

Propriedades Fundamentais

- `.ndim` - Número de dimensões
- `.shape` - Tupla com tamanho em cada dimensão
- `.size` - Número total de elementos
- `.dtype` - Tipo dos dados

Aplicado ao Exemplo

```
>>> arr2.ndim
2
>>> arr2.shape
(2, 2)
>>> arr2.size
4
>>> arr2.dtype
dtype('int64')
```

NumPy - Propriedades Básicas de Arrays

Propriedade	**Descrição**
`ndarray.ndim`	Número de eixos (dimensões) do array.
`ndarray.shape`	Dimensões do array. Uma tupla de inteiros indicando o tamanho em cada dimensão. Para uma matriz com n linhas e m colunas, shape será (n, m). O comprimento da tupla shape é igual ao número de dimensões (`ndim`).
`ndarray.size`	Número total de elementos do array.
`ndarray.dtype`	Objeto que descreve o tipo dos elementos no array. Podem ser usados tipos padrão do Python ou tipos específicos do NumPy como `numpy.int32`, `numpy.int16` e `numpy.float64`.

NumPy – Arrays com Valores Pré-definidos

Por que usar valores pré-definidos?

- Facilita a **inicialização** de matrizes antes do preenchimento com dados
- Evita **erros** na alocação de memória para grandes arrays
- Útil para **cálculos numéricos** e simulações

Funções de Criação

Função	**Descrição**
`np.zeros(shape)`	Cria array preenchido com **zeros**
`np.ones(shape)`	Cria array preenchido com **uns**
`np.empty(shape)`	Cria array com **valores aleatórios**

NumPy – Arrays 1D com Valores Pré-definidos

Exemplos 1D

```
# Array de 5 zeros
np.zeros(5)

# Array de 3 uns
np.ones(3)

# Array vazio 4 posições
np.empty(4)
```

Dica Importante

Especifique sempre o `dtype` para controle preciso do tipo numérico:
`np.zeros(5, dtype=np.float32)`

NumPy – Arrays 2D com Valores Pré-definidos

Exemplos 2D

```
# Matriz 2x3 de zeros
np.zeros((2,3))

# Matriz 3x3 de uns
np.ones((3,3))

# Matriz 2x2 vazia
np.empty((2,2))
```

Dica Importante

Ao criar arrays multidimensionais, lembre-se de usar dois parênteses:
o primeiro envolve a tupla com as dimensões, o segundo é da chamada da função.

Exemplo correto: `np.ones((3,3), dtype=int)`

Exemplo: Criação de Arrays com Zeros

```
# importando o módulo
import numpy as np

# criando um array com cinco elementos sendo zeros
array_zeros = np.zeros(5)
print(array_zeros)
# Saída: [0. 0. 0. 0. 0.]

# criando uma matriz 3x4 preenchida com zeros
matriz_zeros = np.zeros((3, 4))
print(matriz_zeros)
# Saída: [[0. 0. 0. 0.]
          [0. 0. 0. 0.]
          [0. 0. 0. 0.]]
```

Exemplo: Criação de Arrays com np.empty

```
# importando o módulo
import numpy as np

# criando uma matriz 3x3 sem inicialização garantida
array_empty = np.empty((3, 3))
print(array_empty)
# Saída: [[4.67296746e-307 1.69121096e-306 1.37959131e-306]
          [1.11261162e-306 1.11260619e-306 9.34609790e-307]
          [8.45559303e-307 9.34600963e-307 1.37959740e-306]]
```

Observação

Os valores podem ser aleatórios, pois `np.empty` não inicializa os elementos, apenas aloca espaço na memória.

Criando Arrays com `np.arange()`

Array 1D (Vetor)

```
# Vetor de 0 a 8
v = np.arange(9)
print(v)
# [0 1 2 3 4 5 6 7 8]

# Vetor de 0 a 8 pulando de 2 em 2
v2 = np.arange(0, 9, 2)
print(v2)
# [0 2 4 6 8]
```

Dica

`np.arange(início, fim, passo)` cria um vetor com espaçamento definido.
O valor final **não é incluído**. O passo pode ser positivo ou negativo!

Criando Matrizes com `reshape()`

Matriz 2x2

```
# Criar e redimensionar
m2x2 = np.arange(1,5).reshape(2,2)
print(m2x2)
# [[1 2]
#  [3 4]]
```

Matriz 3x3

```
# Criar e redimensionar
m3x3 = np.arange(9).reshape(3,3)
print(m3x3)
# [[0 1 2]
#  [3 4 5]
#  [6 7 8]]
```

Dica

- `reshape(linhas, colunas)` para transformar um array 1D em matriz.
- **O número total de elementos deve ser compatível**

NumPy – Entendendo Eixos (Axes)

O que são eixos em NumPy?

Em arrays multidimensionais, os eixos representam as direções ao longo das quais as operações podem ser aplicadas.

- `axis=0`: opera coluna por coluna
  Saída: `[coluna1, coluna2, coluna3]`
- `axis=1`: opera linha por linha
  Saída: `[linha1, linha2, linha3]`

Dica Visual

Pense que o `axis` é a dimensão que será "reduzida".
`axis=0` colapsa cada coluna.
`axis=1` colapsa cada linha.

NumPy – Exemplo Prático com Eixos

Soma com axis=0 e axis=1

```
import numpy as np
matriz = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Soma por colunas (axis=0)
print(np.sum(matriz, axis=0))
# Saída: [12 15 18]

# Soma por linhas (axis=1)
print(np.sum(matriz, axis=1))
# Saída: [ 6 15 24]
```

Operações ao Longo dos Eixos em NumPy

Operação	**axis = 0 (colunas)**	**axis = 1 (linhas)**
`np.sum()`	Soma por coluna	Soma por linha
`np.mean()`	Média por coluna	Média por linha
`np.max()`	Máximo por coluna	Máximo por linha
`np.min()`	Mínimo por coluna	Mínimo por linha

Exemplo: Matriz de vendas

- Linhas representam lojas
- Colunas representam produtos (Lápis, borracha, caderno)

Codigo

```
vendas = np.array([
    [10, 20, 30],  # Loja 1
    [15, 25, 35],  # Loja 2
    [12, 18, 28],  # Loja 3
    [8, 22, 26],   # Loja 4
])
np.sum(vendas, axis=0) # Soma por coluna (produtos)
# [45 85 119]
np.sum(vendas, axis=1) # Soma por linha (lojas)
# [60 75 58 56]
np.mean(vendas, axis=0) # Média por produto (em todas as lojas)
# [11.25 21.25 29.75]
```

Operações Aritméticas em NumPy – Adição Escalar (Vetor)

Adição Escalar

```
import numpy as np

A = np.arange(1, 10)
print(A)
# [1 2 3 4 5 6 7 8 9]

print(A + 5)  # Adição escalar
# [ 6  7  8  9 10 11 12 13 14]
```

Operações Aritméticas em NumPy – Matriz + Escalar

Adição Escalar em Matrizes

```
import numpy as np

B = np.random.randint(0, 10, (3,3))
print(B)
# [[4 3 1]
#  [6 0 7]
#  [9 3 3]]

print(B + 5)
# [[ 9  8  6]
#  [11  5 12]
#  [14  8  8]]
```

Exemplo: Estoque em Papelarias

- Cada linha representa uma loja;
- Cada coluna representa um item: lápis, borracha e caderno.

Código

```
import numpy as np
# Quantidade atual em 3 lojas
estoque = np.array([[10, 5, 2],
                    [3,  8, 4],
                    [6,  2, 7]])

novo_estoque = estoque + 5 # Reposição de 5 unidades em cada item

print(novo_estoque)
# [[15 10  7]
#  [ 8 13  9]
#  [11  7 12]]
```

Operações entre Vetores em NumPy - Soma

Vetor A

```
import numpy as np

A = np.array([1, 2, 3, 4, 5])
print(A)
# [1 2 3 4 5]
```

Vetor B

```
B = np.array([10, 20, 30, 40, 50])
print(B)
# [10 20 30 40 50]
```

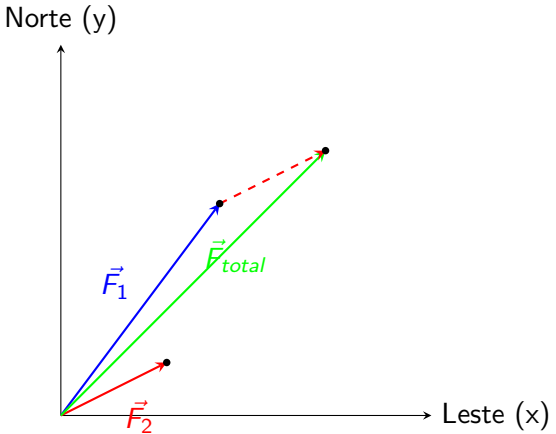
Soma A + B

```
print(A + B)
# [11 22 33 44 55]
```

Soma de Vetores com NumPy – Exemplo

Forças

```
import numpy as np

F1 = np.array([3, 4])
F2 = np.array([2, 1])

F_total = F1 + F2
print(F_total)
# [5 5]
```



Operações entre Matrizes em NumPy - Soma

Matriz B

```
B = np.random.randint(0, 10, (3,3))
print(B)
# [[4 3 1]
#  [6 0 7]
#  [9 3 3]]
```

Matriz C

```
C = np.random.randint(0, 10, (3,3))
print(C)
# [[3 7 1]
#  [6 4 5]
#  [0 1 7]]
```

Soma B + C

```
print(B + C)
# [[ 7 10  2]
#  [12  4 12]
#  [ 9  4 10]]
```

Operações entre Matrizes – Compras em uma Loja

Cada linha é um cliente e cada coluna um item comprado (lápis, caneta, caderno).

Compras no Dia 1

```
dia1 = np.array([
    [1, 2, 0],  # Cliente 1
    [0, 1, 3],  # Cliente 2
    [2, 0, 1]   # Cliente 3
])
```

Compras no Dia 2

```
dia2 = np.array([
    [0, 1, 2],
    [1, 0, 1],
    [1, 2, 1]
])
```

Total de Compras (Dia 1 + Dia 2)

```
total = dia1 + dia2
print(total)
# [[1 3 2]
#  [1 1 4]
#  [3 2 2]]
```

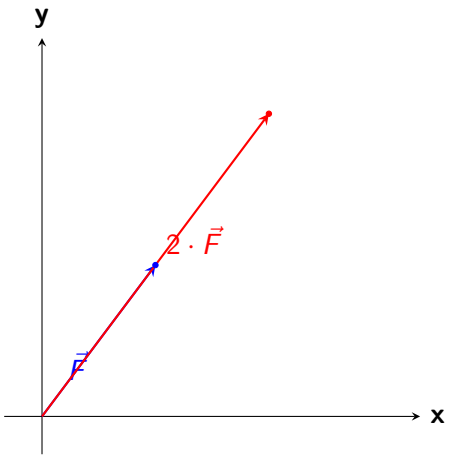
Multiplicação Escalar de Vetor

Código NumPy – Multiplicação Escalar

```
import numpy as np

F = np.array([3, 4])
print(F)
#[3 4]

# Multiplicando por 2
print(2 * F)
#[6 8]
```

Vetores

$$\vec{F} = \begin{bmatrix} 3 \\ 4 \end{bmatrix} \quad 2 \cdot \vec{F} = \begin{bmatrix} 6 \\ 8 \end{bmatrix}$$



NumPy - Multiplicação Element-wise

Exemplo: Receita por produto (quantidade × preço)

```
import numpy as np

Quantidade_produtos = np.array([10, 5, 2])
Preco_produtos = np.array([1, 0.5, 20])

# Gasto por produto
print(Quantidade_produtos * Preco_produtos)
# [10.   2.5 40. ]
```

Multiplicação de cada elemento

Cenário

Cada linha representa uma loja.Cada coluna representa um produto (Lápis, Borracha, Caderno). Queremos saber a receita por produto em cada loja (quantidade $\times$ preço unitário).

Quantidade Vendida (B)

```
B = np.array([[4, 3, 1],
              [6, 0, 7],
              [9, 3, 3]])
```

Preço Unitário (C)

```
C = np.array([[3, 7, 1],
              [6, 4, 5],
              [0, 1, 7]])
```

Receita por Loja e Produto (B * C)

```
print(B * C)
# [[12 21  1]
#  [36  0 35]
#  [ 0  3 21]]
```

Multiplicação Matricial com `np.dot()`

Matriz B (3x3)

```
B = np.array([[4, 3, 1],
              [6, 0, 7],
              [9, 3, 3]])
```

Matriz C (3x3)

```
C = np.array([[3, 7, 1],
              [6, 4, 5],
              [0, 1, 7]])
```

Resultado `np.dot(B, C)`

```
print(np.dot(B, C))
# [[ 30  41  26]
#  [ 18  55  41]
#  [ 45  78  45]]
```

Diferença Fundamental

- `B * C`: Multiplicação elemento a elemento
- `np.dot(B,C)`: Multiplicação de matrizes

Explicação: Multiplicação Matricial com `np.dot()`

Dado que temos duas matrizes quadradas **B** e **C**, ambas de dimensão 3×3. Estamos comparando duas formas distintas de multiplicação em NumPy: a multiplicação elemento a elemento (`*`) e a multiplicação matricial (`np.dot()`).

A diferença das duas operações:
Diferente da multiplicação elemento a elemento (que faz $B[i][j] \times C[i][j]$), a multiplicação matricial segue a regra:
Para cada elemento da matriz resultante, fazemos o produto escalar da linha i de B pela coluna j de C.

Exemplo prático:
Para calcular o valor na posição (0,0) do resultado:

$$4 \times 3 + 3 \times 6 + 1 \times 0 = 12 + 18 + 0 = 30$$

Esse processo se repete para cada posição da matriz resultante.

Transposição de Matrizes em NumPy

Matriz Original

```
B = np.array([[4, 3, 1],
              [6, 0, 7],
              [9, 3, 3]])
```

Método 1: Atributo .T

```
print(B.T)
# [[4 6 9]
#  [3 0 3]
#  [1 7 3]]
```

Método 2: Função transpose()

```
print(B.transpose())
# [[4 6 9]
#  [3 0 3]
#  [1 7 3]]
```

Resolver Sistema Linear (np.linalg.solve(a,b))

Considere o sistema linear:

$$\begin{cases} 2x + 3y - z = 5 \\ 4x + y + 2z = 6 \\ -3x + 2y + z = -4 \end{cases}$$

Representamos na forma matricial $Ax = b$:

$$A = \begin{bmatrix} 2 & 3 & -1 \\ 4 & 1 & 2 \\ -3 & 2 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} 5 \\ 6 \\ -4 \end{bmatrix}$$

Código Python para resolver usando NumPy:

```
import numpy as np
A = np.array([[2, 3, -1],
              [4, 1, 2],
              [-3, 2, 1]])
b = np.array([5, 6, -4])
x = np.linalg.solve(A, b)
print("Solução: ", x)
```

Resolva

Considere o seguinte sistema linear:

$$\begin{cases} 3x_1 - 2x_2 + 4x_3 + x_4 - x_5 + 2x_6 = 7 \\ -2x_1 + x_2 - x_3 + 3x_4 + 5x_5 - x_6 = -3 \\ x_1 + x_2 + x_3 + x_4 + x_5 + x_6 = 10 \\ 4x_1 - x_2 + 2x_3 - x_4 + 3x_5 - 2x_6 = 2 \\ -3x_1 + 5x_2 - x_3 + 2x_4 - 4x_5 + x_6 = 5 \\ 2x_1 + 3x_2 - 2x_3 + x_4 + x_5 - 3x_6 = -1 \end{cases}$$

Forma Matricial

Representamos o sistema como $Ax = b$:

$$A = \begin{bmatrix} 3 & -2 & 4 & 1 & -1 & 2 \\ -2 & 1 & -1 & 3 & 5 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 4 & -1 & 2 & -1 & 3 & -2 \\ -3 & 5 & -1 & 2 & -4 & 1 \\ 2 & 3 & -2 & 1 & 1 & -3 \end{bmatrix}, \quad b = \begin{bmatrix} 7 \\ -3 \\ 10 \\ 2 \\ 5 \\ -1 \end{bmatrix}$$

Resolvendo com NumPy

Código Python para resolver o sistema com NumPy:

```
import numpy as np

A = np.array([
    [3, -2, 4, 1, -1, 2],
    [-2, 1, -1, 3, 5, -1],
    [1, 1, 1, 1, 1, 1],
    [4, -1, 2, -1, 3, -2],
    [-3, 5, -1, 2, -4, 1],
    [2, 3, -2, 1, 1, -3]
])

b = np.array([7, -3, 10, 2, 5, -1])

x = np.linalg.solve(A, b)
print("Solução:", x)
```