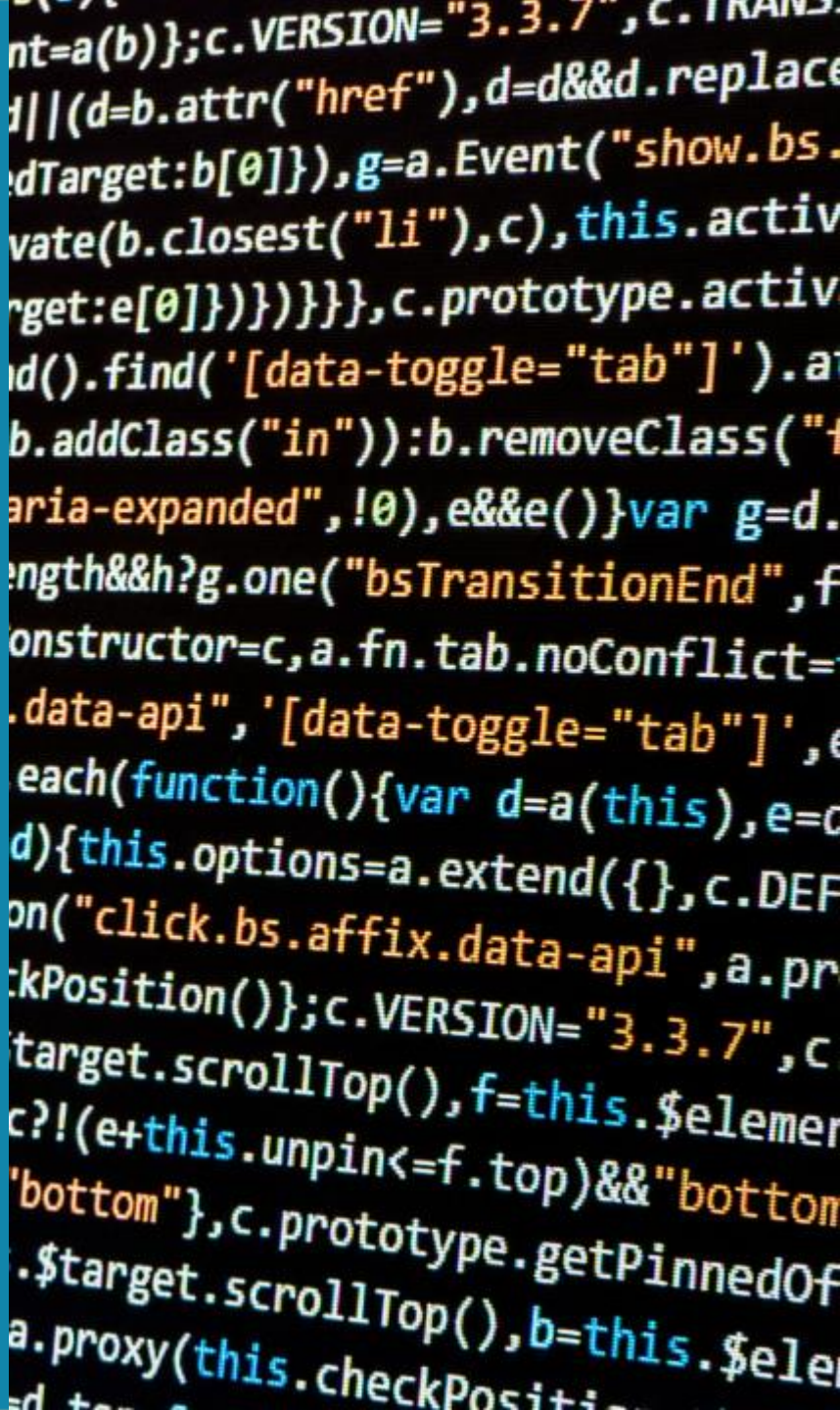
Olá, Professor!
Peço que realize a **avaliação do conteúdo** com o intuito de manter o nosso material sempre atualizado.

Avalie este conteúdo!

**https://bit.ly/451BDqS**

# Paradigmas de linguagens de programação em python

# Paradigmas e linguagem Python

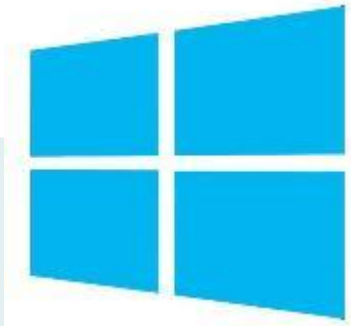


# Razões para estudarmos linguagens de programação

## Linguagem de programação e produtividade do programador

Um programa de computador, ou software, é um conjunto de instruções a fim de orientar o hardware do computador para o que deve ser feito. O software pode ser classificado em:

**Software aplicativo**



**Software básico**

# Razões para estudarmos linguagens de programação

## O papel da abstração nas linguagens de programação

- Linguagem de máquina é composta por 1s e 0s, difícil para humanos.
- Linguagem Assembly introduziu nomes simbólicos, mas ainda era rígida.
- Linguagens de alto nível surgiram para oferecer abstração e facilitar a programação.
- Estas linguagens se aproximam da linguagem humana, tornando a programação mais eficiente.

# Razões para estudarmos linguagens de programação

## O papel da abstração nas linguagens de programação

Tabela exibe código Python, Assembly (Linux) e linguagem de máquina de um processador.

| Linguagem Python | Assembly | Linguagem de máquina |
|---|---|---|
| `def swap(self, v, k):` | `swap:` | `00000000001111111111100000000001` |
| `  temp = self.v[k];` | `Muli $2,$5,4` | `00011111111000000111000011111101` |
| `  self.v[k] =` | `Add $2,$4,$2` | `11111000001100000111111110000000` |
| `  self.v[k+1];` | `Lw $15,0($2)` | `10000000100000001000000010000000` |
| `  self.v[k+1]= temp;` | `Lw $16,4($2)` | `00000000010000000001000000000010` |
| | `Sw $16,0($2)` | `00000000000000001111000010010101` |
| | `Sw $15,4($2)` | `00000000111000111111001111111111` |
| | `Jr $31` | |

# Razões para estudarmos linguagens de programação

## O papel da abstração nas linguagens de programação

# Classificação das linguagens de programação

## Classificação por nível

A classificação por nível considera a proximidade da linguagem de programação com as características da arquitetura do computador ou com a comunicação com o homem.

# Classificação das linguagens de programação

## Classificação por nível


**Linguagem de baixo nível**

- Aproximam-se da linguagem de máquina e interagem diretamente com hardware.
- Usam conjuntos de instruções específicos de processadores.
- Programação em linguagem de máquina é complexa, levando ao desenvolvimento do Assembly.
- O Assembly usa instruções reais do processador e precisa ser convertido para linguagem de máquina.

# Classificação das linguagens de programação

## Classificação por nível

**Linguagem de alto nível**

- Distanciam-se da linguagem de máquina, aproximando-se da linguagem humana.
- Abstraem detalhes de hardware e componentes do computador.
- Incluem linguagens como C, C++, Java, JavaScript e outras com instruções abstratas.
- Facilitam a programação, tornando-a mais legível e compreensível para desenvolvedores.

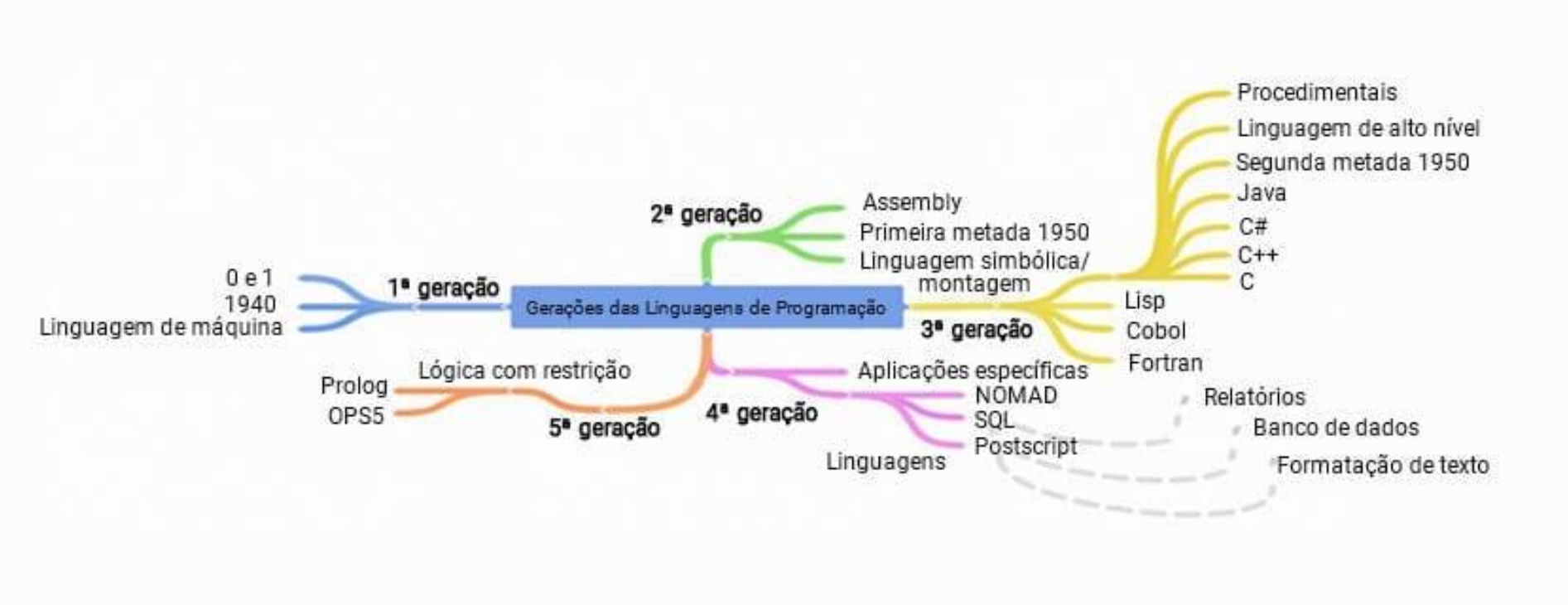
# Classificação por gerações

Linguagens de programação são frequentemente classificadas em gerações, variando de 5 a 6 gerações, à medida que novos recursos são adicionados.

# Classificação por gerações

- LINGUAGENS DE 1ª GERAÇÃO (LINGUAGEM DE MÁQUINA)
- LINGUAGENS DE 2ª GERAÇÃO (LINGUAGEM DE MONTAGEM – ASSEMBLY)
- LINGUAGENS DE 3ª GERAÇÃO (LINGUAGENS PROCEDURAIS)
- LINGUAGENS DE 4ª GERAÇÃO (LINGUAGENS APLICATIVAS)
- LINGUAGENS DE 5ª GERAÇÃO (VOLTADAS À INTELIGÊNCIA ARTIFICIAL)

# Classificação por gerações




Características das gerações das Linguagens de Programação.

# Domínios da programação

As principais áreas de programação:

- Aplicações científicas (máquinas de calcular com alta precisão)
- Aplicações comerciais
- Aplicações com inteligência artificial
- Programação de sistemas
- Programação para web
- Programação mobile

# Avaliação de linguagens de programação

Existem quatro grandes critérios para avaliação das linguagens de programação. Cada critério é influenciado por algumas características da linguagem.

**Legibilidade**

**Facilidade de escrita**

**Confiabilidade**

**Custo**

# Legibilidade

As características que influenciam a legibilidade de uma linguagem de programação são:

- **Simplicidade**
- **Ortogonalidade**
- **Instruções de controle**
- **Tipos e estruturas de dados**
- **Sintaxe**

# Legibilidade

## Facilidade de escrita (redigibilidade)

Facilidade de escrita refere-se à medida de quão simples é criar programas em uma linguagem.

Simplicidade e ortogonalidade ajudam, quanto menos construções primitivas, melhor.

Expressividade contribui para escrever códigos de forma mais conveniente.

Suporte para abstração permite criar representações mais próximas do domínio do problema.

# Legibilidade

## Confiabilidade

Confiabilidade em programação refere-se à capacidade de um programa se comportar conforme especificado. Alguns fatores que afetam a confiabilidade incluem:

- Verificação de tipos em tempo de compilação ou execução, reduzindo erros de tipo.
- Tratamento de exceções para lidar com eventos indesejados.
- Restrição de "aliasing", onde vários nomes se referem à mesma memória.
- Legibilidade e facilidade de escrita, influenciando a confiabilidade do código.

# Legibilidade

## Custo

O custo de uma linguagem envolve treinamento, escrita do programa, compilação, execução, implementação, confiabilidade e manutenção.

Facilidade de escrita e legibilidade afetam o custo de treinamento e escrita de programas.

Python é uma linguagem com baixo custo, devido à sua legibilidade, facilidade de escrita e confiabilidade.

Portabilidade, reusabilidade e facilidade de aprendizado são critérios adicionais de avaliação de linguagens.

# Paradigma imperativo

## Paradigma estruturado

Caracteriza as principais linguagens de programação da década de 1970 e 1980 que seguiram os princípios da programação estruturada:

- Não usar desvios incondicionais (Goto, característico de linguagens como BASIC e versões iniciais do COBOL).
- Desenvolver programas por refinamentos sucessivos (metodologia top down), motivando o desenvolvimento de rotinas
- Desenvolver programas usando três tipos de estruturas: sequenciais, condicionais e repetição.
- O paradigma estruturado baseia-se nos princípios da arquitetura de Von Neumann

# Paradigma imperativo

## Paradigma orientado a objetos

Paradigma orientado a objetos surgiu para lidar com complexidade e permitir organização e reuso de código.

Classes definem estruturas com atributos e métodos, e objetos são instâncias dessas classes.

Python é uma linguagem orientada a objetos com suporte a herança e polimorfismo.

Traz maior organização e extensibilidade, permitindo desenvolvimento rápido e confiável.

# Paradigma imperativo

## Paradigma concorrente

- Envolve a execução simultânea de processos, competindo pelos recursos de hardware.
- Pode ocorrer em um ou vários processadores, localizados em uma única máquina ou em máquinas distribuídas.

# Paradigma declarativo

## Paradigma funcional

- Paradigma declarativo enfoca o que o programa deve fazer, não como fazer.
- Inclui o paradigma funcional, que se baseia em funções, gerando programas concisos.
- Exemplos de linguagens declarativas incluem LISP, Haskell e ML.
- Amplamente utilizado em Inteligência Artificial, devido à interpretação recursiva.

# Paradigma declarativo

## Paradigma lógico

O paradigma lógico expressa a solução como raciocínio baseado em fatos e inferência.

Usa uma máquina de inferência para deduzir novos fatos a partir dos existentes.

Prolog é a linguagem de programação lógica mais conhecida.

Usado em sistemas de banco de dados, sistemas especialistas e tutores inteligentes. Python não é adequado para esse paradigma.

# Tradução

- Tradução converte linguagem de alto nível em linguagem de máquina.
- O processo de tradução ocorre em várias fases simultaneamente.
- Fases do tradutor: Compilação, Montagem, Carga e Ligação.
- Comum chamar todo o processo de "compilação", mas compilação é uma de suas fases.

# Tradução

## Compilador

- O compilador é central no processo de tradução.
- Muitas vezes, erroneamente, o processo todo é chamado de compilação.
- Compilador é figura central no projeto de linguagem.


Componentes na compilação de um programa fonte.

# Tradução

## Compilador

**Fases da compilação:**

| | | |
|---|---|---|
| Análise léxica | Análise sintática | Análise semântica |
| Gerador de código intermediário, otimizador de código e gerador de código | Tratador de erros | Gerenciador da tabela de símbolos |

# Interpretação

- Interpretação traduz e executa comandos do programa-fonte imediatamente.
- O interpretador funciona com três etapas: Obter instrução, Interpretar e Executar.
- Processo similar ao de máquinas de Von Neumann: Obter, Decodificar e Executar instruções.


Processo de interpretação.

# Interpretação

**Tradução x interpretação:**

| | Vantagens | Desvantagens |
|---|---|---|
| Tradutores | 1. Execução mais rápida | 1. Várias etapas de conversão. |
| | 2. Permite estruturas de programas mais complexas. | 2. Programação final é maior, necessitando de mais memória para sua execução. |
| | 3. Permite a otimização de código. | 3. Processo de correção de erros e depuração é mais demorado. |
| Interpretadores | 1. Depuração mais simples. | 1. Execução do programa é mais lenta. |
| | 2. Consome menos memória. | 2. Estruturas de dados demasiado simples. |
| | 3. Resultado imediato do programa (ou parte dele). | 3. Necessário fornecer o código fonte ao utilizador. |

Fonte: http://codemastersufs.blogspot.com/2015/10/compiladores-versus-interpretadores.html. Adaptado pelo autor.

# Sistemas híbridos

- Combinam a velocidade de compiladores com a portabilidade de interpretadores.
- Usam código intermediário facilmente interpretável e não específico de plataforma.
- Isso permite portar programas para diferentes plataformas sem alterações.
- Exemplos notáveis incluem Python e Java com suas máquinas virtuais.

# Sistemas híbridos

## Sistema de implementação de Python

Python utiliza um sistema híbrido com compilador e interpretador.

O compilador gera um código intermediário executado na PVM (Python Virtual Machine).

É possível traduzir código Python para Bytecode Java com Jython.

O interpretador converte para código de máquina em tempo de execução, enquanto o compilador traduz o programa inteiro.

# Python básico

# Visão geral

- Criada por Guido van Rossum em 1990, Python destaca-se pela clareza, eficiência e legibilidade do código.
- Python é uma linguagem de alto nível que simplifica a programação com instruções intuitivas.
- Em contraste, linguagens de baixo nível exigem comunicação mais próxima do dispositivo.
- Adequada para projetos de diferentes escalas, enfatizando espaços em branco significativos.

# Visão geral

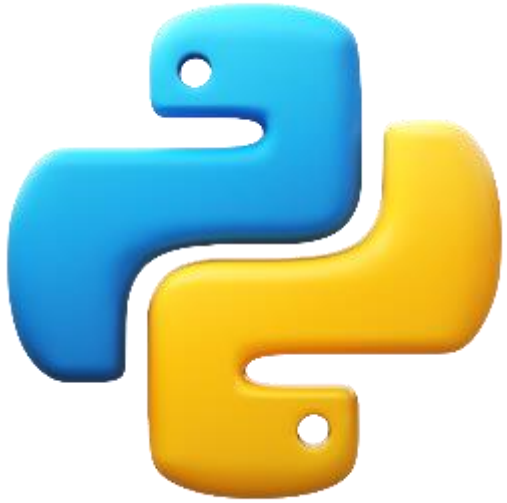
## Características da linguagem Python

| | | | |
|---|---|---|---|
| É multiparadigma | É interativa | É híbrida quanto ao método de implementação | É portável |
| É extensível | Suporta bancos de dados | Suporta interface com usuário | Pode ser usado como linguagem de script |
| | Permite desenvolvimento de aplicações Web | Permite criação de aplicações comerciais | |

# Instalando o Python

## Instalação

- Python é uma linguagem em crescimento no ensino de programação.
- Para usá-la, é necessário baixar o Python e uma IDE, como o PyCharm.



# Instalando o Python

## Utilitários e módulos

Pode-se usar utilitários como ***dir*** e ***help*** para explorar atributos e documentação de tipos de dados.

No Python Console, digite ***dir(int)*** para ver os atributos disponíveis e ***help(int)*** para acessar a documentação do tipo int.

Pressione Enter após digitar cada comando no console Python.

Esses utilitários são úteis para entender e explorar tipos de dados em Python.

# Principais Características

## Blocos

Em Python, blocos são definidos pela indentação, não por chaves.

Um bloco começa com ‘:’ na linha superior e é representado por 4 espaços à esquerda.

O código mostra blocos aninhados com for, if, else, e suas relações de indentação.

As linhas no mesmo nível de indentação pertencem ao mesmo bloco.

# Principais Características

## Comentários

Em Python, os comentários podem ser de uma linha ou de várias linhas.

| | Python | C |
|---|---|---|
| Comentários com uma linha | Iniciados com # | Iniciados com // |
| Comentários com várias linhas | Limitados por """ (três aspas duplas) no início e no fim | Iniciados com /* e encerrados com */ |

Tabela: Comentários.
Humberto Henriques de Arruda.

# Principais Características

## Boas práticas de programação

Uma prática muito importante é utilizar comentários no seu programa, explicando o que aquele trecho resolve.

Uma característica marcante da comunidade de desenvolvedores Python é manter uma lista de propostas de melhorias, chamadas PEP.

# Variáveis

## Conceitos

Variáveis são abstrações para endereços de memória, tornando a programação mais compreensível.

Em Python, o operador de atribuição é o símbolo **=.**

**A instrução x = 10 atribui o valor 10 à variável x.**

Pode-se usar print(x) para exibir o valor de x no console Python.

# Variáveis

## Identificadores de variáveis

Os identificadores das variáveis podem ser compostos por letras, o underline (_) e, com exceção do primeiro caractere, números de 0 a 9.

**MinhaVariavel, _variavel, salario1 e salario1_2**

São válidos.

**1variavel e salario-1**

Não são válidos.

**MinhaVariavel e minhavariavel**

São identificadores de duas variáveis distintas.

# Conceito de Amarração *(Binding)*

## Amarrações

Chama-se de amarração (binding) a associação entre entidades de programação. Exemplos:

- Variável amarrada a um valor;
- Operador amarrado a um símbolo;
- Identificador amarrado a um tipo.

# Conceito de Amarração *(Binding)*

## Amarrações

O tempo em que a amarração ocorre é chamado de **tempo de amarração**. Cada linguagem pode ter os seguintes tempos de amarração:

**Tempo de projeto da linguagem**

**Tempo de implementação**

**Tempo de Compilação**

**Tempo de ligação**

**Tempo de carga**

**Tempo de execução**

# Conceito de Amarração *(Binding)*

## Amarração de tipo

As amarrações de tipo vinculam a variável ao tipo do dado. Elas podem ser:

**Estáticas**

Ocorrem antes da execução e permanecem inalteradas. Em C, declaramos int a.

**Dinâmicas**

Ocorrem durante a execução e podem ser alteradas. É o caso do Python.

# Escopo e Tempo de Vida

## Escopo de visibilidade

- Escopo define onde variáveis são visíveis no programa.
- Cada variável em Python tem seu escopo.
- Fora do escopo, o nome da variável não existe.
- Variáveis podem ser globais ou locais, dependendo do escopo.

# Escopo e Tempo de Vida

## Variáveis globais

- São nomes atribuídos no prompt interativo ou fora de funções são globais. Exemplo: Variável x fora de funções é global.
- Podem ser acessadas em todo o programa.
- Escopo global é útil para compartilhar dados entre funções.

# Escopo e Tempo de Vida

## Variáveis locais

Variáveis definidas dentro de funções são de escopo local.

Exemplo: Variável "a" fora da função é global, dentro da função é local.

O Python procura variáveis em ordem: escopo da função, globais, módulo builtins.

Para modificar uma variável global dentro de uma função, use a palavra reservada "global".

# Escopo e Tempo de Vida

## Escopos

Os tipos de escopo são:

### Estático

O escopo é baseado na descrição textual do programa e as amarrações são feitas em tempo de compilação. É o caso de C, C++ e Java, por exemplo.

### Dinâmico

O escopo é baseado na sequência de chamada dos módulos (ou funções). Por isso, as amarrações são feitas em tempo de execução. É o caso do Python.

# Escopo e Tempo de Vida

## Tempo de vida

Embora escopo e tempo de vida tenham uma relação próxima, eles são conceitos diferentes.

| **Escopo** | **Tempo de vida** |
| --- | --- |
| É um conceito textual. | É um conceito temporal. |

# Constantes

## Definição

Em Python, não existem constantes nativas. Para criar constantes:

1. Atribua um valor a uma variável com prefixo "c_" ou em letras maiúsculas.
2. Exemplo: c_PI = 3.141592 ou PRECISION = 0.001.
3. Evite alterar o valor para manter a constância.
4. Cuidado com escopo e visibilidade para evitar resultados inesperados.

# Tipos de dados padrão

## Conceitos

Tipos de dados padrão incorporados ao interpretador Python. Os principais tipos internos são:

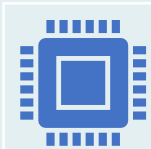
**Numéricos**

**Sequenciais**

**Dicionários**

# Tipos de dados padrão

## Tipos numéricos



Python possui três tipos numéricos principais: inteiros, ponto flutuante e complexos.

Inteiros em Python não têm limites definidos, dependem da memória disponível.

Ponto flutuante é usado para números reais, incluindo frações e decimais.

Python suporta números complexos com parte real e imaginária (por exemplo, 3 + 4j).

# Tipos de dados padrão

## O tipo float

Manipula números com parte inteira e parte decimal (números de ponto flutuante).

Usado para representar elementos do conjunto dos números reais (R). Use um ponto como separador decimal, não uma vírgula.

Operações com inteiros resultam em inteiros, mas um float em qualquer operação gera um resultado float.

A divisão de inteiros pode gerar um float. Para obter o quociente inteiro e o resto, use // e %, respectivamente.

# Tipos de dados padrão

## O tipo complex

- Usado para números complexos na forma x + yj (onde x é a parte real e y é a parte imaginária).
- Exemplo: Complexo 1 é 2 + 5j.
- O método r.conjugate() retorna o conjugado do número complexo r, mantendo a parte real e invertendo o sinal da parte imaginária.

# Tipos de dados padrão

## O tipo bool

Usado para avaliar expressões booleanas. Expressões booleanas podem ser avaliadas como True ou False.

Por exemplo, 2 < 3 avalia para True, enquanto 2 > 3 avalia para False.

O operador not inverte o valor booleano; por exemplo, not(2 < 3) é False.

Python permite criar expressões booleanas compostas usando conectivos como E (and) e OU (or).

# Tipos de dados padrão

## Operadores numéricos

| Operação matemática | Símbolo usado | Exemplo | |
|---|---|---|---|
| | | Equação | Resultado |
| Soma | + | 2.5 + 1.3 | 3.8 |
| Subtração | - | 2.5 - 1.3 | 1.2 |
| Multiplicação | * | 2.5 * 1.3 | 3.25 |
| Divisão | / | 2.5/1.3 | 1.9230769230769 |
| Divisão inteira | // | 9/2 | 4 |
| Resto na divisão inteira | % | 9%2 | 1 |
| Valor absoluto | abs(parâmetro) | abs(-2.5) | 2.5 |
| Exponenciação | ** | 2**4 | 16 |

Operadores matemáticos.

# Tipos de dados padrão

## Operadores numéricos

Além das operações algébricas, é possível realizar operações de comparação. Os operadores de comparação têm como resultado um valor booleano (**True** ou **False**).

| Símbolo usado | Descrição |
|---|---|
| < | Menor que |
| <= | Menor ou igual a |
| > | Maior que |
| >= | Maior ou igual a |
| == | Igual |
| != | Não igual |

Operadores de comparação.

# Tipos de dados padrão

## Operadores booleanos

- As expressões booleanas são aquelas que podem ter como resultado um dos valores booleanos: **True** ou **False**.
- É possível escrever expressões algébricas complexas concatenando diversas expressões menores e escrever expressões booleanas grandes, com os operadores **and**, **or** e **not**.

# Tipos de dados padrão

## Tipos sequenciais

**Listas:** Sequência mutável de elementos. A indexação começa em 0 e pode ser negativa para contar do final.

**Tuplas:** Sequência imutável de elementos. A indexação é semelhante às listas.

**Objetos range:** Usados para criar sequências numéricas. São imutáveis e economizam memória.

**Strings (str):** Tipo especial para dados textuais. Também é uma sequência imutável e pode ser indexada de maneira semelhante às listas.

# Tipos de dados padrão

## Strings

Armazenam texto e caracteres usando aspas simples ou duplas.

Métodos úteis incluem upper (maiúsculas), lower (minúsculas) e split (dividir em substrings).

As strings podem conter letras, números, espaços e símbolos.

São flexíveis e amplamente usadas para manipulação de texto em Python.

# Tipos de dados padrão

## Listas

Listas são sequências mutáveis, normalmente usadas para armazenar coleções de itens homogêneos.

**[]**

Usando um par de colchetes para denotar uma lista vazia.

**[a], [a, b, c]**

Usando colchetes, separando os itens por vírgulas.

**[x for x in iterable]**

Usando a compreensão de lista.

**list() ou list(iterable)**

Usando o construtor do tipo list.

# Tipos de dados padrão

## Tuplas

Sequências imutáveis usadas para armazenar coleções de itens, geralmente heterogêneos.

Criadas usando parênteses vazios () para uma tupla vazia ou separando itens por vírgulas (a, b, c).

Também podem ser criadas usando o construtor tuple(iterable), onde o iterable pode ser uma sequência, um objeto iterador ou um contêiner iterável.

As vírgulas geram a tupla, enquanto os parênteses são opcionais, exceto para tuplas vazias.

# Tipos de dados padrão

## Range

- Representa uma sequência imutável de números.
- Com um único argumento, a sequência começa em 0 e vai até (exclusive) o limite do argumento.
- Com dois argumentos, é possível definir o início e o fim da sequência, com incremento padrão de 1.
- Três argumentos permitem definir início, fim e passo (incremento).

# Tipos de dados padrão

## Operadores sequenciais comuns

| Uso | Resultado |
|---|---|
| x in s | **True** se **x** for um subconjunto de **s** |
| x not in s | **False** se **x** for um subconjunto de **s** |
| s + t | Concatenação de **s** e **t** |
| n*s | Concatenação de **n** cópias de **s** |
| s[i] | Caractere de índice **i** em **s** |
| len(s) | Comprimento de **s** |
| min(s) | Menor item de **s** |
| max(s) | Maior item de **s** |

Operadores sequenciais.

# Tipos de dados padrão

## Dicionários

Estrutura de dados que associa pares de chave-valor.

Criados com chaves {} e pares chave-valor separados por **:**.

Exemplo: {'chave1': 'valor1', 'chave2': 'valor2’}.

Chaves são únicas e imutáveis (strings, números, tuplas), usadas para acessar valores.

# Relação de precedência entre operadores

Os parênteses têm a maior prioridade e forçam a avaliação interna primeiro.

Exponenciação, multiplicação, divisão, adição e subtração são avaliados em ordem.

Comparação, operadores de pertencimento, de identidade e lógicos têm a menor prioridade.

Parênteses podem ser usados para alterar a ordem de avaliação quando necessário.

# Relação de precedência entre operadores

| Operador | Descrição |
|---|---|
| [expressões ...] | Definição de lista |
| x[ ], x[índice : índice] | Operador de indexação |
| ** | Exponenciação |
| +x, -x | Sinal de positivo e negativo |
| *, /, //, % | Produto, divisão, divisão inteira, resto |
| +, - | Soma, subtração |
| in, not in, <, <=,>, >=, <>, !=, == | Comparações, inclusive a ocorrência em listas |
| not x | Booleano NOT (não) |
| and | Booleano AND (e) |
| or | Booleano OR (ou) |

Operadores sequenciais.

# Conversões de tipos de dados

Python realiza conversões implícitas entre tipos, convertendo operadores para o tipo mais abrangente na expressão.

Por exemplo, int é convertido em float, mas o inverso não é verdadeiro.

Valores booleanos True e False são convertidos em inteiros 1 e 0, respectivamente.

Conversões explícitas podem ser realizadas usando os construtores de tipos desejados, como int(), float(), etc.

# Variáveis: formas de atribuição

## Sentenças de atribuição

**Atribuição simples**

Como x = 10, onde uma variável (neste caso, x) recebe um único valor.

**Atribuição múltipla**

Permite atribuir valores a várias variáveis em uma única instrução, como x, y = 2, 5, onde as variáveis x e y recebem valores 2 e 5, respectivamente.

# Variáveis: formas de atribuição

## Operadores de atribuição compostos

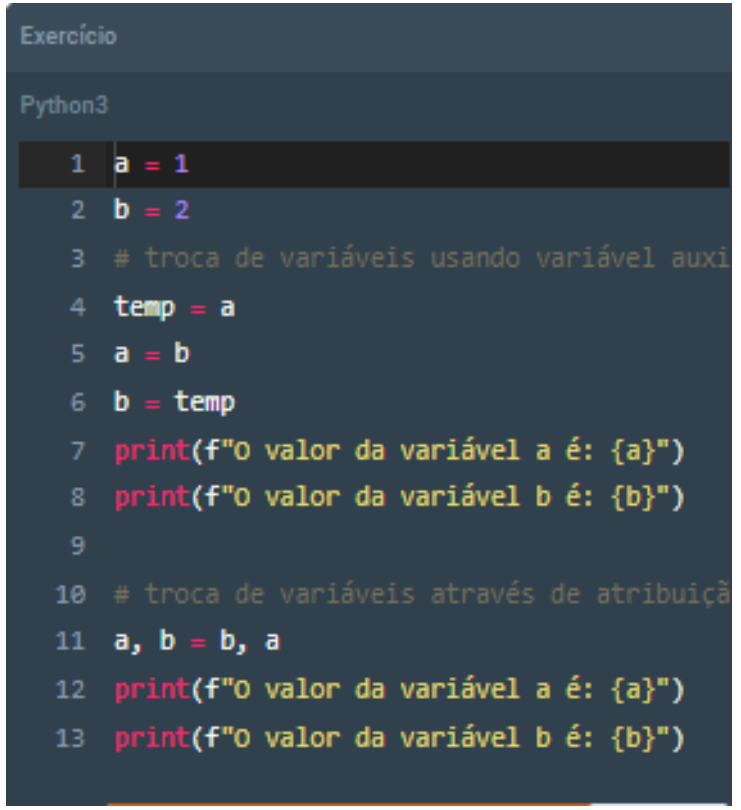
| Nome | Símbolo usado | Exemplo | |
|---|---|---|---|
| | | Instrução | Resultado |
| Mais igual | += | x += 2 | x passa a valer 12 |
| Menos igual | -= | x -= 2 | x passa a valer 8 |
| Vezes igual | *= | x *= 2 | x passa a valer 20 |
| Dividido igual | /= | x /= 2 | x passa a valer 5 |
| Módulo igual | %= | x %= 3 | x passa a valer 1 |

Operadores compostos.

# Variáveis: formas de atribuição

## Troca de variáveis

- Pode ser feita de forma simples com atribuição múltipla, como a, b = b, a.
- Não é necessária uma variável auxiliar.
- Torna o código mais simples e legível.



# Programação em Python

## O primeiro programa em Python

- Utilize uma IDE como o PyCharm.
- Crie um novo arquivo com a extensão .py, como "primeiro_programa.py".
- Digite suas instruções no arquivo para executar seu programa.

# Programação em Python

## Saída de dados com a função print()

Imprime na tela em linhas diferentes sem a necessidade do caractere \n para pular a linha.

Permite imprimir valores de variáveis sem especificar um formato, ao contrário de printf() em C.

Para imprimir "Hello World" em Python, use print("Hello World").

Pode imprimir strings ou valores de variáveis previamente definidas.

# Programação em Python

## Entrada de dados com a função Input()

- Solicita a entrada do usuário.
- Armazena a entrada como uma string.
- Use conversões como int() ou float() para tipos diferentes.
- Verifique o tipo da variável com print(type(variavel)).

# Programação em Python

## A função eval()

- Recebe uma string como parâmetro.
- Avalia a string como uma expressão numérica.
- Permite que você trate a entrada do usuário como um número e execute operações aritméticas.
- Pode ser usada com ***input()*** para obter entrada do usuário e realizar cálculos com ela.

# Formatação de dados de saída

Formate a saída de dados usando concatenação de strings, método format(), ou f-strings.

Especifique a largura de campo para números inteiros na formatação.

Ajuste a precisão ao exibir números de ponto flutuante.

Use format() com chaves {} para inserir valores em uma string de forma flexível.

# Formatação de dados de saída

## Impressão de sequências

Python facilita a impressão de sequências, incluindo strings e vetores.

Use colchetes para imprimir uma substring indicando o intervalo de índices.

Para imprimir uma string invertida, use [::-1] no parâmetro da função print().

Lembre-se de que os limites do intervalo devem respeitar o sentido reverso quando for usado.

# Python estruturado

# Estruturas de decisão

## Tratamento das condições

Python possui estruturas de decisão e repetição semelhantes a C.

Python tem o tipo bool, que permite condições verdadeiras ou falsas.

Em Python, True representa verdadeiro, e False representa falso.

Em C, qualquer valor diferente de 0 é verdadeiro, e 0 ou vazio é falso.

# Estruturas de decisão

## As estruturas de decisão if, if-else e elif

Em Python, é possível utilizar as estruturas de decisão if e if-else da mesma forma que em C. A diferença principal é o modo de delimitar os blocos de instruções relativos a cada parte da estrutura.

| Python | C |
| --- | --- |
| if <condição>: | if <condição> { |
| Instruções com 4 espaços de indentação | A indentação não é exigida |
| Instrução fora do if | } |

Estruturas de decisão simples.

# Estruturas de decisão

## As estruturas de decisão if, if-else e elif

| Python | C |
| --- | --- |
| if <condição>: | if <condição> { |
| Instruções com 4 espaços de indentação (caso a condição seja verdadeira) | Bloco 1 (caso a condição seja verdadeira, a indentação não é exigida) |
| else: | } else { |
| Instruções com 4 espaços de indentação (caso a condição seja falsa) | Bloco 2 (caso a condição seja falsa). A indentação não é exigida |
| Instrução fora do if | } |

Estruturas de decisão compostas.

# Estruturas de repetição

## Estruturas de repetição for

**As listas do tipo range()**

Nas estruturas de repetição for em Python:


- Python oferece maior flexibilidade em relação a C.
- A função range() gera sequências de números.
- **range(x)** cria sequência de **0 a x (exclusive).**
- **range(start, end)** cria de start a **end (exclusive).**
- **range(start, end, step)** cria com passo de **step.**

# Estruturas de repetição

## Estruturas de repetição for

**A sintaxe da estrutura for**

A estrutura for tem a seguinte sintaxe em Python:

```python
for <variável> in <sequência>:
    Bloco que será repetido para todos os itens da sequência
Instrução fora do for
```

# Estruturas de repetição

## Estruturas de repetição for

**A sintaxe da estrutura for**

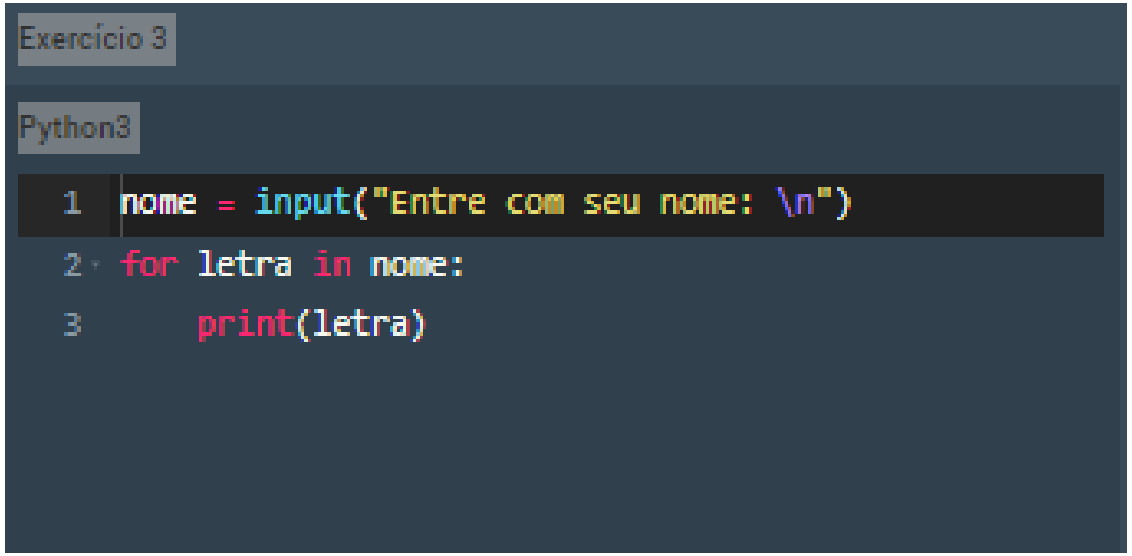
Diferença de sintaxe entre as linguagens C e Python.

| Python | C |
| --- | --- |
| for <variável> in <sequência>: | for (inicialização; condição; incremento ou decremento){ |
| Instruções com 4 espaços de indentação | Bloco de instruções a ser repetido. A indentação não é exigida |
| Instrução fora do for | } |

# Estruturas de repetição

## O laço for com uma string

Em Python, é possível usar um loop for para percorrer os caracteres de uma string, facilitando operações como soletrar palavras.



# Estruturas de repetição

## Uso do laço for com qualquer sequência

O laço for em Python não está limitado a sequências numéricas ou de caracteres; ele pode ser usado com qualquer tipo de sequência.

Você pode percorrer sequências heterogêneas, como listas, para executar operações específicas em cada item.

Veja um exemplo disso no código fornecido no exercício 4.

O for é uma ferramenta flexível para iterar sobre uma ampla variedade de sequências em Python.

# Estruturas de repetição

## Estrutura de repetição while

A estrutura de repetição **while** tem funcionamento e sintaxe muito semelhantes nas linguagens C e Python.

| Python | C |
| --- | --- |
| while <condição>: | while <condição>{ |
| Instruções com 4 espaços de indentação | Bloco de instruções a ser repetido. A indentação não é exigida. |
| Instrução fora do while | } |

Comparação do while em Python e em C.

# Estruturas de repetição

## O laço while infinito

O laço while infinito é útil para executar um bloco de instruções indefinidamente.

Pode ser necessário encerrar um laço infinito com a instrução break.

A instrução continue interrompe a iteração atual de um laço, passando para a próxima.

A instrução pass permite criar um bloco condicional com um placeholder para futuras instruções.

# Visão geral

## Definições básicas

Um subprograma é definido quando sua interface e ações são descritas pelo desenvolvedor.

Um subprograma é chamado quando uma instrução solicita explicitamente sua execução.

Um subprograma está ativo a partir do momento de sua chamada até a conclusão de sua execução.

O cabeçalho de um subprograma inclui seu nome, parâmetros e tipo de retorno.

# Parâmetros, procedimentos e funções

## Parâmetros

Subprogramas podem receber dados através de parâmetros ou acessando variáveis não locais visíveis a eles.

O acesso sistemático a variáveis não locais pode prejudicar a confiabilidade do programa.

Parâmetros formais são definidos no cabeçalho do subprograma, enquanto os parâmetros reais são fornecidos ao chamar o subprograma.

Python permite definir valores padrão para parâmetros formais, que são usados quando nenhum parâmetro real é fornecido.

# Parâmetros, procedimentos e funções

## Procedimentos e funções

- Procedimentos não retornam valores.
- Funções retornam valores.
- Em algumas linguagens, como Python, as funções podem ser definidas sem retornar valores, comportando-se como procedimentos.

# Parâmetros, procedimentos e funções

## Ambientes de referenciamento local

**Variáveis locais**

Subprogramas estabelecem ambientes de referenciamento local com variáveis locais.

Variáveis locais têm escopo restrito ao corpo do subprograma.

Em Python, todas as variáveis locais são dinâmicas da pilha, enquanto em C/C++, algumas podem ser declaradas como estáticas.

Variáveis globais podem ser implicitamente declaradas como locais se possuírem o mesmo nome que uma variável local em um subprograma Python.

# Parâmetros, procedimentos e funções

## Ambientes de referenciamento local

**Subprogramas aninhados**

Python e a maioria das linguagens funcionais permitem subprogramas aninhados.

Em Python, é possível definir uma função dentro de outra função, como no exemplo da função taximetro() que contém a função calculaMult().

A função aninhada calculaMult() é chamada na linha 7, e seu retorno é atribuído à variável multiplicador.

Em linguagens como C e C++, a aninhamento de funções não é permitido.

# Parâmetros, procedimentos e funções

## Ambientes de referenciamento local

**Métodos de passagens de parâmetros**

Métodos de passagem de parâmetros: valor (cópia do valor do parâmetro real) e referência (transmissão de um caminho de acesso ao valor).

Em C, ponteiros são usados para passagem por referência; caso contrário, é passagem por valor.

Python usa passagem por atribuição, onde todas as variáveis são referências a objetos.

A passagem por atribuição é semelhante à passagem por referência, já que os parâmetros reais são referências a objetos.

# Funções recursivas

## Recursividade

Funções recursivas são aquelas que chamam a si mesmas.

Para evitar recursão infinita, é essencial definir uma condição de parada.

Funções recursivas bem definidas possuem casos básicos como condições de parada e chamadas recursivas com parâmetros mais próximos dos casos básicos.

Exemplos clássicos de funções recursivas incluem o cálculo do fatorial de um número inteiro não negativo e a sequência de Fibonacci.

# Funções recursivas

## A função recursiva fatorial

A função matemática fatorial de um inteiro não negativo n é calculada por:

$$n! = \begin{cases} 1, \text{ se } n = 0 \text{ ou } n = 1 \\ n.(n-1)\cdot(n-2)\ldots 3.2.1, \text{ se } n \geq 2 \end{cases}$$

Além disso, ela pode ser definida recursivamente por:

$$n! = \begin{cases} 1, \text{ se } n = 0 \text{ ou } n = 1 \\ n \cdot [(n-1)!], \text{ se } n \geq 2 \end{cases}$$

# Funções recursivas

## A sequência de Fibonacci

A sequência de Fibonacci é: 1, 1, 2, 3, 5, 8, 13, 21... Os dois primeiros termos são 1; a partir do 3º termo, cada termo é a soma dos dois anteriores.

Python

```python
def fibo(n):
    if n == 1 or n == 2:
        return 1
    else:
        return fibo(n - 1) + fibo(n - 2)
```

# Documentação de funções – Docstrings

## Docstrings

- Em Python, é possível definir docstrings para documentar funções, usando uma string como descrição.
- O utilitário help() pode ser usado para exibir a documentação da função, baseada na docstring.
- A docstring é inserida como uma string no início da definição da função e pode ser acessada usando help(nome_da_funcao).

# Importação de funções e módulos

## Biblioteca padrão Python

A biblioteca padrão Python fornece funções, métodos e classes para diversas finalidades.

Ela é organizada em módulos, com mais de 200 módulos para operações matemáticas, GUI, funções matemáticas e mais.

Isso estende a funcionalidade da linguagem e facilita o desenvolvimento de aplicativos.

Compreender classes e objetos é essencial, pois são centrais na programação orientada a objetos.

# Importação de funções e módulos

## Como usar uma função de módulo importado

Para usar funções de um módulo importado em Python:

- Importe o módulo desejado com **import nome_modulo.**
- Chame a função desejada precedida pelo nome do módulo com **nome_modulo.nome_funcao(paramêtros**).

```
Python3
1  import math
2
3  x = math.sqrt(5)
4  print(x)
```

# Módulos nativos do Python

## Módulo math

| Função | Retorno |
| --- | --- |
| sqrt(x) | Raiz quadrada de x |
| ceil(x) | Menor inteiro maior ou igual a x |
| floor(x) | Maior inteiro menor ou igual a x |
| cos(x) | Cosseno de x |
| sin(x) | Seno de x |
| log(x, b) | Logaritmo de x na base b |
| pi | Valor de Pi (3.141592...) |
| e | Valor de e (2.718281...) |

Principais funções do módulo math.

# Módulos nativos do Python

## Módulo random

Esse módulo implementa geradores de números pseudoaleatórios para várias distribuições.

### Números inteiros

Para inteiros, existe:

- Uma seleção uniforme a partir de um intervalo.

### Sequências

Para sequências, existem:

- Uma seleção uniforme de um elemento aleatório; - Uma função para gerar uma permutação aleatória das posições na lista; - Uma função para escolher aleatoriamente sem substituição.

# Módulos nativos do Python

## Módulo random

O módulo random em Python oferece funções para trabalhar com distribuições de valores reais, números inteiros e sequências.

**Para valores reais**, as funções incluem **random()** para valores aleatórios entre 0 e 1 e **gauss()** para distribuição gaussiana.

**Para números inteiros, randrange()** permite escolher um elemento dentro de um intervalo, e **randint()** gera inteiros em um intervalo especificado.

**Para sequências, choice()** seleciona elementos aleatórios, **shuffle()** embaralha a sequência e **sample()** gera uma amostra sem repetição.

# Módulos nativos do Python

## Módulo SMTPLIB

- O módulo smtplib em Python permite enviar e-mails através do protocolo SMTP.
- Para usar o Gmail, ative a opção "Permitir aplicativos menos seguros" nas configurações da conta.
- O código envolve criar uma mensagem de e-mail, configurar um servidor SMTP seguro, fazer login, enviar a mensagem e encerrar o servidor.

# Módulos nativos do Python

## Módulo time

| Função | Retorno |
| --- | --- |
| time() | Número de segundos passados desde o início da contagem (epoch). Por padrão, o início é 00:00:00 do dia 1 de janeiro de 1970. |
| ctime(segundos) | Uma **string** representando o horário local, calculado a partir do número de segundos passado como parâmetro. |
| gmtime(segundos) | Converte o número de segundos em um objeto **struct_time** descrito a seguir. |
| localtime(segundos) | Semelhante à **gmtime()**, mas converte para o horário local. |
| sleep(segundos) | A função suspende a execução por determinado número de segundos. |

Principais funções do módulo time.

# Módulos nativos do Python

## Módulo tkinter

O módulo tkinter é a interface gráfica padrão do Tk para Python.

Para criar uma GUI, importe o módulo tkinter, crie uma janela usando Tk(), e use elementos como Label e Button.

Funções podem ser associadas a botões para ações.

Exemplo: criação de uma janela com texto, imagem e um botão associado a uma função.

# Pacotes externos

## Usando pacotes externos

Os pacotes externos são módulos desenvolvidos por terceiros que estendem as funcionalidades da linguagem Python.

O Python Package Index (PyPI) é um repositório de pacotes externos mantido pela comunidade.

O pip é uma ferramenta para instalar pacotes externos do PyPI.

Para instalar um pacote externo, use o comando pip install nome_do_pacote.

# Pacotes externos

## Criação do próprio módulo

- Os desenvolvedores podem criar os próprios módulos de forma a reutilizar as funções que já escreveram e organizar melhor seu trabalho.
- Para isso, basta criar um arquivo .py e escrever nele suas funções.
- O arquivo do módulo precisa estar na mesma pasta do arquivo para onde ele será importado.

# Erros em um programa Python

## Erros e exceções

Erros em Python podem ser de dois tipos: erros de sintaxe, resultantes de instruções mal formadas.

Erros em tempo de execução ocorrem devido a estados inválidos no programa.

Exceções são criadas quando o programa encontra um erro em tempo de execução.

Exceções são objetos que contêm informações sobre o erro e podem ser tratadas no código.

# Erros em um programa Python

## Erros e exceções

| Exceção | Explicação |
|---|---|
| KeyboardInterrupt | Levantado quando o usuário pressiona CTRL+C, a combinação de interrupção. |
| OverflowError | Levantado quando uma expressão de ponto flutuante é avaliada como um valor muito grande. |
| ZeroDivisionError | Levantado quando se tenta dividir por 0. |
| IOError | Levantado quando uma operação de entrada/saída falha por um motivo relacionado a isso. |
| IndexError | Levantado quando um índice sequencial está fora do intervalo de índices válidos. |
| NameError | Levantado quando se tenta avaliar um identificador (nome) não atribuído. |
| TypeError | Levantado quando uma operação da função é aplicada a um objeto do tipo errado. |
| ValueError | Levantado quando a operação ou função tem um argumento com o tipo correto, mas valor incorreto. |

Tipos comuns de exceção.

# Tratamento de exceções e eventos

## Captura e manipulação de exceções

Captura e manipulação de exceções permitem que programas continuem a ser executados mesmo após um erro.

O bloco try contém as instruções de execução normal do programa.

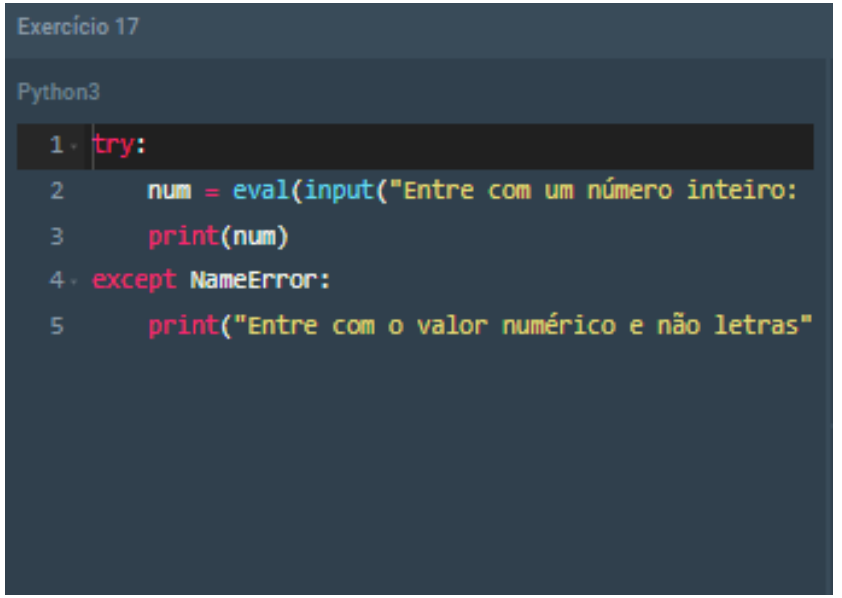
O bloco except é executado caso uma exceção seja levantada, permitindo o tratamento adequado do erro.

Em Python, o manipulador de exceção padrão exibe mensagens de erro no console se não houver tratamento explícito.

# Tratamento de exceções e eventos

## Captura de exceções de determinado tipo

- Python permite que o bloco relativo ao except só seja executado caso a exceção levantada seja de determinado tipo.
- Para isso, o except precisa trazer o tipo de exceção que se deseja capturar.



# Tratamento de exceções e eventos

## Captura de exceções de múltiplos tipos

O Python permite que haja diversos tratamentos para diferentes tipos possíveis de exceção. Isso pode ser feito com mais de uma cláusula except vinculada à mesma cláusula try.

Python3

```
1 try:
2     num = eval(input("Entre com um número inteiro:
3     print(num)
4 except ValueError:
5     print("Mensagem 1")
6 except IndexError:
7     print("Mensagem 2")
8 except:
9     print("Mensagem 3")
```

# Tratamento de exceções e eventos

## O tratamento completo das exceções

A forma geral completa para lidar com as exceções em Python é:

```
try:
  Bloco 1
except Exception1:
  Bloco tratador para Exception1
except Exception2:
  Bloco tratador para Exception1
...
else:
  Bloco 2 - executado caso nenhuma exceção seja levantada
finally:
  Bloco 3 - executado independente do que ocorrer
Instrução fora do try/except
```

# Tratamento de exceções e eventos

## Tratamento de eventos

- O tratamento de eventos é semelhante ao tratamento de exceções.
- Os eventos são notificações de ações, como cliques de mouse em botões, em resposta às interações do usuário.
- O tratador de eventos é um segmento de código que é executado em resposta à ocorrência de um evento.

# Python orientado a objeto

# Conceitos e pilares de programação orientada a objetos

## Conceitos de programação orientada a objetos (POO)

A Programação Orientada a Objetos (POO) revolucionou a estruturação de programas de computador.

Ela se baseia em pensar abstratamente problemas do mundo real usando conceitos reais em vez de ideias puramente computacionais.

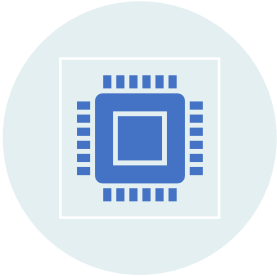
A abordagem baseada em objetos permite que os mesmos conceitos e notações sejam usados desde a análise até a implementação de um projeto de software.

O software na POO é organizado como uma coleção de objetos que incorporam tanto a estrutura quanto o comportamento dos dados.

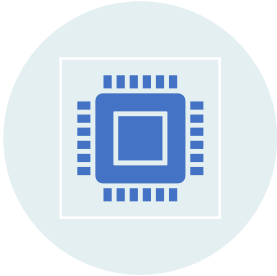
# Conceitos e pilares de programação orientada a objetos

## Pilares da orientação a objetos

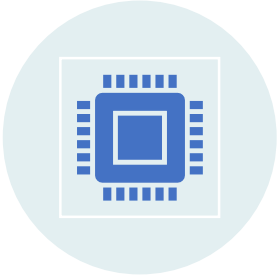
**Objetos**



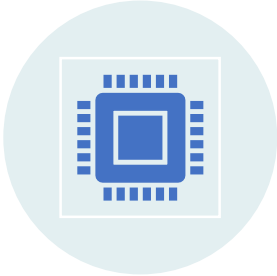
Um objeto na programação orientada a objetos é a representação computacional de um elemento ou processo do mundo real.



Cada objeto possui características (informações) e operações (comportamento) que podem alterar suas características (estado).



Os objetos no mundo real e na programação são variados, como Aluno, Professor, Livro, Empréstimo, e assim por diante.



Na análise de objetos, é importante focar apenas nos objetos relevantes e suas características essenciais para resolver o problema em questão.

# Conceitos e pilares de programação orientada a objetos

## Pilares da orientação a objetos

**Atributos**

São propriedades do mundo real que descrevem um objeto. Cada objeto possui as respectivas propriedades desse mundo, as quais, por sua vez, possuem valores.

Nome - Maria

Idade - 35

Peso - 63kg

Altura - 1,70m

# Conceitos e pilares de programação orientada a objetos

## Pilares da orientação a objetos

**Operações**

Operações são funções que podem ser aplicadas a objetos ou seus dados.

Cada objeto tem um conjunto de operações definidas em sua interface.

Objetos colaboram entre si por meio de suas interfaces, permitindo que operações sejam realizadas.

Desenvolver sistemas orientados a objetos envolve identificar objetos, seus atributos e operações, e mapear isso para resolver problemas.

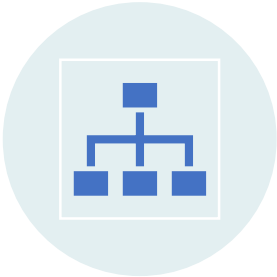
# Conceitos e pilares de programação orientada a objetos

## O conceito de classe

- Uma classe descreve as características e o comportamento de um conjunto de objetos.
- Objetos são instâncias de uma classe.
- As classes são os blocos básicos para a construção de programas orientados a objetos.
- Os objetos colaboram entre si, enviando mensagens e chamando operações para resolver problemas computacionais.

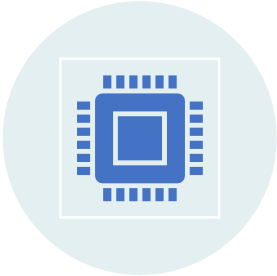

Colaboração entre motorista e carro.

# O conceito de encapsulamento



O encapsulamento separa os aspectos externos de um objeto de seus detalhes internos de implementação.

Isso permite que a interface de comunicação de um objeto revele o mínimo possível sobre seu funcionamento interno.



Os objetos clientes só precisam conhecer as operações disponíveis, sem necessidade de entender a implementação.

O encapsulamento facilita a manutenção e evolução do sistema, isolando alterações internas dos objetos.

# Os mecanismos de herança e polimorfismo

## Herança

A herança é um mecanismo na programação orientada a objetos onde as classes compartilham atributos e comportamentos.

Permite criar hierarquias de classes, com classes "filhas" herdando características de classes "pais".

Classes "filhas" podem reutilizar e especializar comportamentos da classe "pai".

Facilita a evolução de sistemas, permitindo a adição de novas classes à hierarquia.

# Os mecanismos de herança e polimorfismo

## Polimorfismo

Polimorfismo é a capacidade de comportamentos diferentes em classes diferentes.

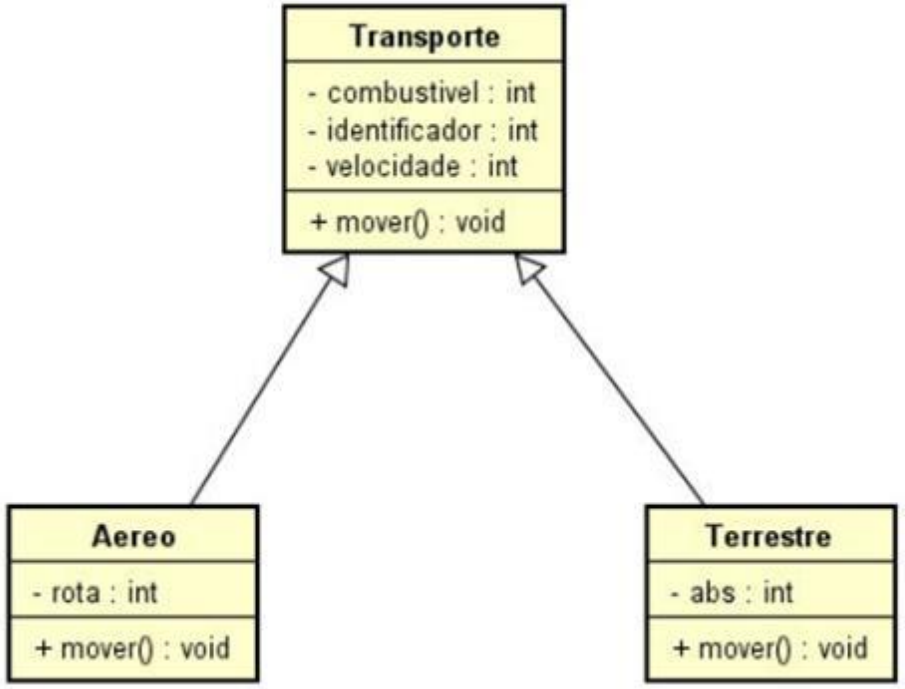
Permite que uma mesma mensagem seja executada de maneira diversa, dependendo do objeto receptor.

Isso ocorre ao reimplementar um método nas subclasses de uma herança.

Exemplo: o método "mover()" pode ter diferentes implementações em classes distintas.

# Os mecanismos de herança e polimorfismo

## Polimorfismo




mover() – método polimórfico.

# Classes e objetos

## Definição de classe

Uma classe é uma declaração de tipo que encapsula variáveis, constantes e métodos.

Cada classe deve ser única em um sistema orientado a objetos.

É boa prática manter uma classe em um arquivo .py com um nome correspondente, como "Conta.py".

As classes são definidas com a palavra-chave "class" seguida de ":".

# Classes e objetos

## Construtores e self

Em Python, o construtor de uma classe é definido pelo método **__init__().**

O construtor é usado para inicializar os atributos da classe quando um objeto é instanciado.

O construtor tem um parâmetro chamado self, que é uma referência ao próprio objeto sendo criado.

A palavra-chave self é usada para se referir aos atributos do objeto dentro do construtor.

# Classes e objetos

## Métodos

As classes em Python possuem métodos que permitem manipular os atributos da classe.

Os métodos são funções definidas dentro da classe e operam no objeto instanciado.

Os métodos podem receber o parâmetro self, que é uma referência ao próprio objeto.

O self é usado para acessar e modificar os atributos do objeto dentro dos métodos.

# Classes e objetos

## Métodos com retorno

- Métodos em Python podem retornar valores.
- A palavra-chave return é usada para especificar o valor de retorno.
- O valor de retorno fornece informações de volta ao chamador do método.
- No exemplo, o método sacar retorna False se o saque for maior que o saldo disponível.

# Classes e objetos

## Referências entre objetos na memória

Instâncias de uma classe representam objetos distintos, com referências separadas na memória.

Referências diferentes para objetos permitem operações independentes e não afetam uns aos outros.

Para interações entre objetos, é possível passar referências de um objeto como parâmetros para métodos.

Esse conceito de referências entre objetos é fundamental para a programação orientada a objetos, permitindo comunicação controlada.

# Tipos de associação entre objetos

## Agregação

A agregação é um relacionamento "todo-parte" entre classes.

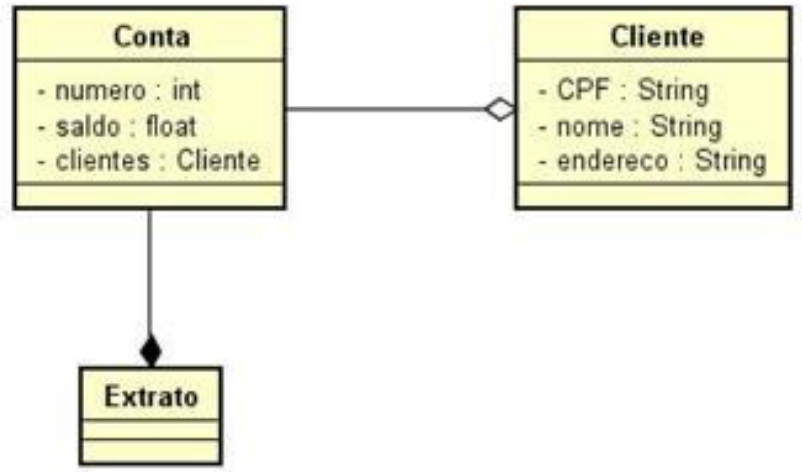
Ela indica que uma classe contém ou é composta por instâncias de outra classe.

Os objetos contidos podem existir independentemente da classe que os contém.

É representada visualmente por um losango e modela relações complexas, como uma conta corrente com vários clientes.

# Tipos de associação entre objetos

## Composição




Classe Conta composta de 1 ou mais extratos.

- Composição é um relacionamento "todo-parte" entre classes, onde uma classe contém uma ou mais instâncias de outra classe.
- Na composição, os objetos contidos só podem existir como parte do objeto que os contém.
- A classe Conta agora é composta por vários objetos Extrato, permitindo que ela armazene transações realizadas e gere extratos.
- As transações realizadas são adicionadas como linhas ao objeto Extrato, que é referenciado pela classe Conta.

# Integridade dos objetos: encapsulamento

## Encapsulamento

Encapsulamento: Protege a integridade dos objetos, evitando alterações indevidas nos atributos.

Reunião de dados e funções: A orientação a objetos reúne dados e métodos em uma única entidade.

Exemplo de encapsulamento: Impede alterações diretas nos atributos que violam as regras de negócio.

Método sacar: Não permite que o saldo fique negativo, mantendo o estado válido do sistema.

# Integridade dos objetos: encapsulamento

## Atributos públicos e privados

Atributos privados: Em Python, são definidos iniciando com dois underscores (ex: __atributo).

Acesso: O interpretador renomeia o atributo privado para _nomedaClasse__nomedoatributo.

Prática não recomendada: Embora seja possível acessar atributos privados, isso viola o encapsulamento da orientação a objetos.

Boas práticas: A disciplina deve ser mantida para não acessar diretamente atributos com __ ou _ definidos nas classes.

# Integridade dos objetos: encapsulamento

## Decorator @property

Decorator @property: Utilizado para criar propriedades em métodos, permitindo o acesso controlado a atributos privados.

Método @<nomedometodo>.setter: Usado para controlar alterações de valores dos atributos privados, garantindo o encapsulamento.

Boa prática: Definir métodos property e setter somente quando há regras de negócio associadas ao atributo.

Propriedades em Python: Contribuem para manter a integridade dos objetos e aplicar o princípio de encapsulamento.

# Integridade dos objetos: encapsulamento

## Atributos de classe

- Atributos de Classe: São usados para controlar valores associados à classe, não às instâncias.
- Exemplo: Ao criar uma classe "Círculo," podemos adicionar um atributo de classe "total_circulos" para contar o número de círculos criados.
- Declaração: A variável de classe é criada na classe e pode ser atualizada para rastrear informações compartilhadas entre todas as instâncias da classe.
- Privacidade: É uma boa prática tornar o atributo de classe privado usando um underscore (ex: _total_circulos) para evitar acesso direto.

# Integridade dos objetos: encapsulamento

## Métodos de classe

Métodos de Classe: São usados para acessar os atributos de classe e têm acesso direto à área de memória que contém esses atributos.

Decorator @classmethod: Utilizado para definir um método como estático, permitindo o acesso aos atributos de classe.

Exemplo: Na classe "Círculo," o método @classmethod permite acessar o atributo de classe _total_circulos.

Parâmetro cls: É criado como uma referência para a classe e é usado dentro do método de classe para acessar os atributos da classe.

# Integridade dos objetos: encapsulamento

## Métodos públicos e privados

- Métodos públicos e privados seguem as mesmas regras dos atributos.
- Métodos privados são marcados com dois underscores antes do nome (ex: __metodo_privado).
- Embora métodos privados possam ser chamados diretamente, a boa prática é usá-los apenas internamente para manter o encapsulamento.

# Integridade dos objetos: encapsulamento

## Métodos estáticos

Métodos Estáticos: São métodos que podem ser chamados sem a necessidade de criar uma instância da classe.

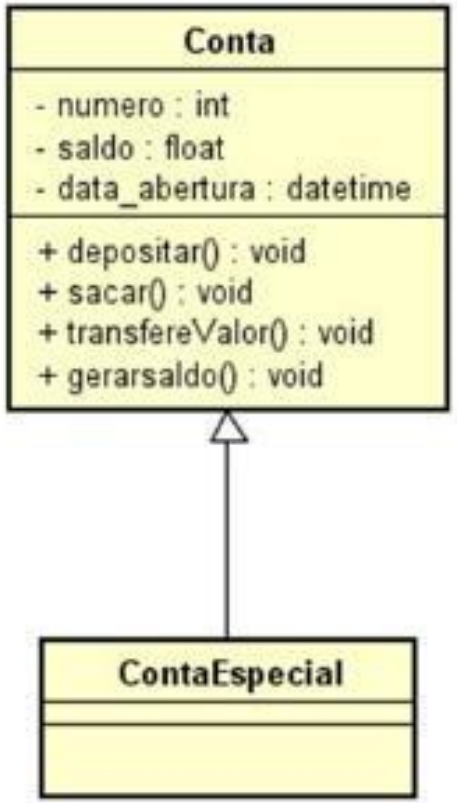
Exemplo: O método sqrt da classe Math pode ser chamado diretamente, sem instanciar um objeto da classe.

Boa Prática: Em programação orientada a objetos, métodos estáticos não são recomendados e devem ser usados apenas em casos especiais, como em classes de log em sistemas.

# Herança e polimorfismo

- Herança: Permite que uma classe filho herde métodos e atributos da classe pai, reduzindo repetições de código.
- Polimorfismo: Permite que objetos tenham comportamentos específicos em situações diferentes.
- Tratamento de Exceções: Facilita a manipulação de erros durante a execução do código, tornando-o mais robusto e controlado.

# Implementando herança




Herança Conta -> ContaEspecial.

- Herança em POO: Facilita a reutilização de código e a extensão para atender a diferentes regras de negócio.
- Exemplo de Conta Especial: Uma classe ContaEspecial herda da classe Conta, permitindo saques com limite.
- Vantagens: Redução de duplicação de código e manutenção mais simples.
- Código Demonstrado: A implementação mostra a criação e operação de objetos ContaEspecial derivados de Conta.

# Herança múltipla

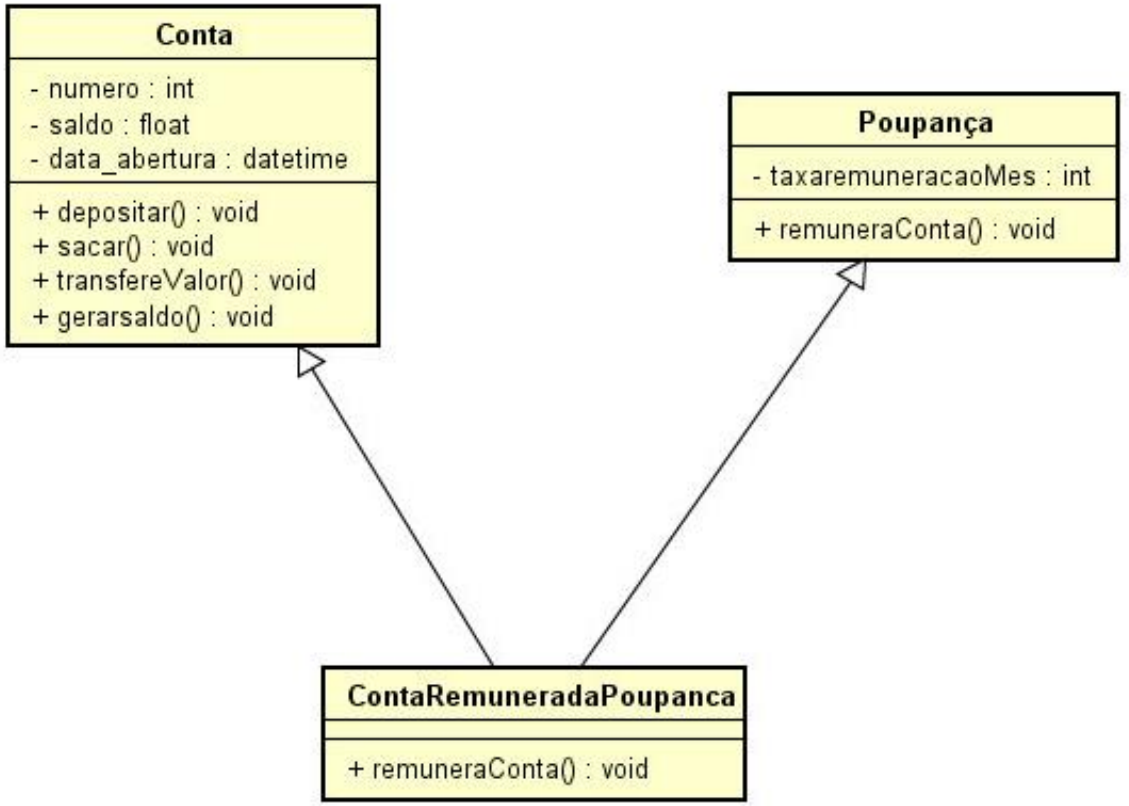
Herança Múltipla: Permite que uma classe herde o código de duas ou mais superclasses, combinando suas funcionalidades.

Exemplo: No sistema bancário, é criada uma classe ContaRemuneradaPoupanca que herda funcionalidades de Conta e Poupanca.

Nova Funcionalidade: A classe ContaRemuneradaPoupanca tem um rendimento diário baseado na poupança e cobra uma taxa de manutenção mensal.

Implementação: O código demonstra a criação de objetos e operações em classes derivadas de várias superclasses.

# Herança múltipla




Herança múltipla.

# Implementando polimorfismo

Polimorfismo: Permite métodos com o mesmo nome se comportarem de maneira diferente conforme o objeto que os chama.

Exemplo Bancário: Classes ContaCliente, ContaComum e ContaRenumerada possuem regras específicas para calcular rendimento.

Uso de Herança: Todas as classes são do tipo ContaCliente, permitindo que o Banco adicione contas de tipos diferentes.

Benefícios: Polimorfismo simplifica o código, evita verificações de tipos e torna a chamada de métodos transparente em sistemas com muitas subclasses.

# Classes abstratas

Classes Abstratas: Não podem ser instanciadas e servem como modelos para subclasses concretas.

Herança e Abstração: Utiliza-se o módulo abc em Python para criar classes abstratas herdando da classe ABC.

Método Abstrato: Uma classe abstrata deve conter pelo menos um método abstrato, indicado com @abstractmethod.

Subclasses Concretas: Subclasses como ContaComum e ContaVIP devem fornecer a implementação concreta do método abstrato da superclasse ContaCliente.

# Classes abstratas

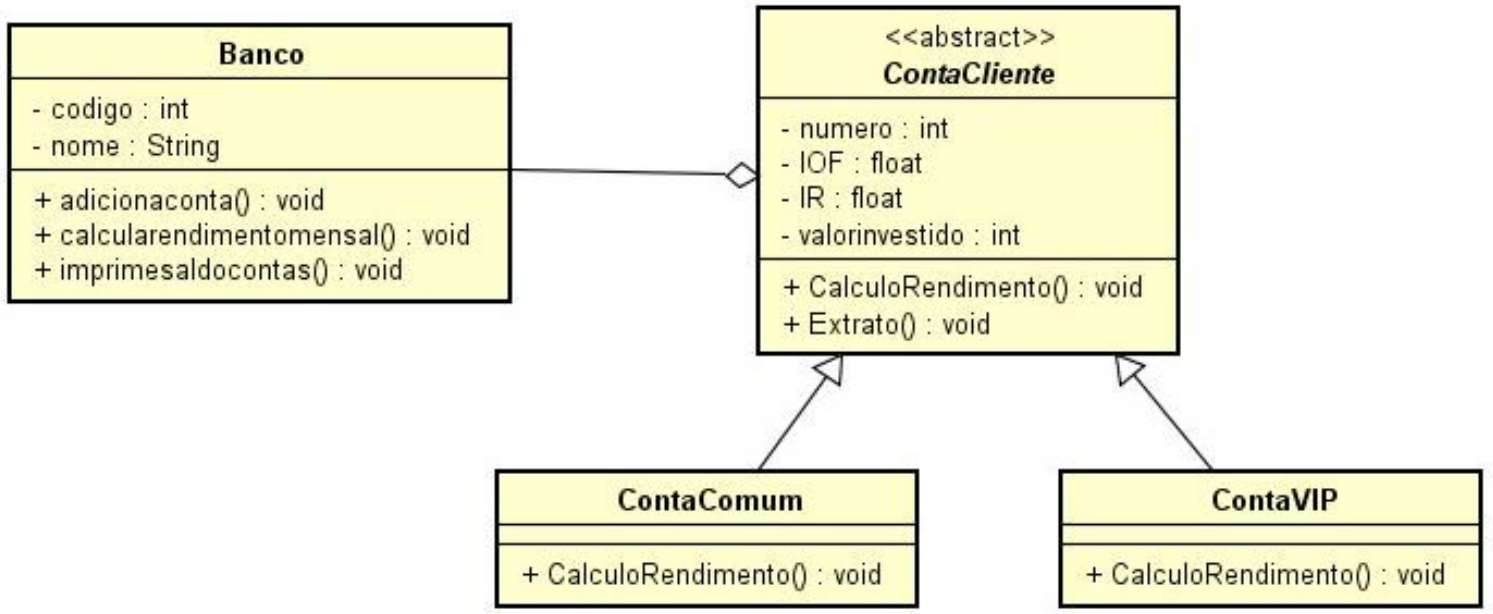



Diagrama de classes abstratas.

# Tratamento de exceções

- Tratamento de Exceções em Python: Permite criar exceções personalizadas herdando de Exception.
- Exemplo: O código define a exceção ExcecaoCustomizada e a captura usando try...except.
- Herança de Exceções: É uma prática comum herdar exceções da classe Exception para distinguir erros da aplicação dos erros padrão da linguagem.

```
1  class ExcecaoCustomizada(exception):
2      pass
3
4      def throws(): (2)
5          raise ExcecaoCustomizada
6          try:
7              throws()
8          except ExcecaoCustomizada as ex:
9              print ("Excecao lançada")
```

# Orientação a objetos e suas linguagens de programação

## Comparação com C++ e JAVA

**Instanciação de objetos**

As linguagens Java e C++ nos obrigam a utilizar a palavra reservada new e a indicar o tipo do objeto. A linguagem C++ referencia todos os objetos explicitamente por intermédio de ponteiros.

| Java | C++ | Python |
| --- | --- | --- |
| Conta c1 = new Conta() | Conta *c1 = new Conta() | c1 = Conta() |

# Orientação a objetos e suas linguagens de programação

## Comparação com C++ e JAVA

**Construtores de objetos**

As linguagens Java e C++ exigem um método definido como público e com o mesmo nome da classe. O Python obriga a ser um método privado __init__, conforme demonstra a tabela a seguir.

| Java | C++ | Python |
| --- | --- | --- |
| Utilizado um método público com o mesmo nome da classe sem tipo de retorno: Public Conta() | Utilizado um método com o mesmo nome da classe sem tipo de retorno: Conta::Conta(void) | Utilizando um método público com a obrigatoriedade da passagem do objeto com self: def __init__(self) |

# Orientação a objetos e suas linguagens de programação

## Comparação com C++ e JAVA

**Modificadores de acesso a atributos**

- C++ e Python têm modificadores público e privado para atributos.
- Em C++ e Java, atributos privados são acessados somente pela classe.
- Python permite acessar atributos privados, o que C++ e Java evitam.
- Python não garante o encapsulamento estrito de atributos privados.

# Orientação a objetos e suas linguagens de programação

## Comparação com C++ e JAVA

**Modificadores de acesso a atributos**

| Java | C++ | Python |
| --- | --- | --- |
| Possui os seguintes modificadores:<br>public<br>private<br>protected<br>default | Possui os seguintes modificadores:<br>public<br>private | Possui os seguintes modificadores:<br>público<br>privado – iniciado com "_"<br>Modificador privado pode ser burlado e acessado diretamente |

# Orientação a objetos e suas linguagens de programação

## Comparação com C++ e JAVA

**Herança múltipla de classes**

As linguagens C++ e Python implementam a herança múltipla diretamente por meio de classes, enquanto Java implementa graças à herança múltipla de interfaces.

| Java | C++ | Python |
| --- | --- | --- |
| Não implementa. Utiliza herança múltipla de interfaces para substituir essa característica. | Implementa com a referência das classes herdadas na declaração da classe: Class ContaRemunerada: public Conta, public Poupanca) | Implementa com a referência das classes herdadas na declaração da classe: class ContaRemuneradaPoupanca(Conta, Poupanca): |

# Orientação a objetos e suas linguagens de programação

## Comparação com C++ e JAVA

**Classes sem definição formal**

| Java | C++ | Python |
| --- | --- | --- |
| protected void Calcular(){<br>class Calculo{<br>private int soma;<br>public void setSoma(int soma) {<br>this.soma = soma;<br>}<br>public int getSoma() {<br>return soma;<br>}<br>}<br>} | Existe o conceito de classes locais – internos a funções:<br>void func() {<br>class LocalClass {<br>}<br>} | Existe o conceito de classes locais:<br>def scope_test():<br>def do_local():<br>spam = "local spam" |

# Orientação a objetos e suas linguagens de programação

## Comparação com C++ e JAVA

**Tipos primitivos**

| Java | C++ | Python |
|---|---|---|
| Possui vários tipos primitivos: | Possui vários tipos primitivos: | Todas as variáveis são consideradas objetos: |
| int | bool | >>>5.__add__(3) |
| byte | char | 8 |
| short | int | |
| long | float | |
| float | double | |
| double | void | |
| boolean | wchar_t | |

# Orientação a objetos e suas linguagens de programação

## Comparação com C++ e JAVA

**Interfaces**

| Java | C++ | Python |
| --- | --- | --- |
| Implementa interfaces simples e múltiplas:<br><br>Simples - Class Funcionario implements Autenticavel<br><br>Múltipla - Class Funcionario implements Autenticavel, Descontavel | Implementa interfaces simples e múltiplas:<br><br>Simples - Class Funcionario: public Autenticavel<br><br>Múltipla - Class Funcionario: public Autenticavel, public Descontavel | Não implementa |

# Orientação a objetos e suas linguagens de programação

## Comparação com C++ e JAVA

**Sobrecarga de métodos**

| Java | C++ | Python |
| --- | --- | --- |
| Implementa sobrecarga de métodos:<br>calculaimposto(salario)<br>calculaImposto(salario,IR) | Implementa sobrecarga de métodos:<br>calculaimposto(salario)<br>calculaImposto(salario,IR) | Não implementa |

# Orientação a objetos e suas linguagens de programação

## Comparação com C++ e JAVA

**Tabela comparativa**

Comparação de Linguagens: Java e C++ possuem atributos privados, interfaces e sobrecarga de métodos, ideais para sistemas orientados a objetos robustos.

Python não tem atributos privados nem sobrecarga de métodos.

Vantagens Java e C++: Mais adequadas para sistemas puramente orientados a objetos.

Python: Cresce como primeira linguagem de programação ensinada e é usado em Data Science e desenvolvimento web devido a bibliotecas e simplicidade.

# Python em outros paradigmas

# Visão geral

## Introdução

Programação funcional começou nos anos 1950 com LISP e é um paradigma de programação.

Na programação funcional, as funções devem ser puras, os dados são imutáveis e loops são evitados.

Isso garante previsibilidade e evita efeitos indesejados.

Linguagens como Python, Java e C++ suportam programação funcional, enquanto Haskell, Clojure e Elixir são predominantemente funcionais.

# Relação entre funções puras e dados imutáveis

## Funções puras

Funções puras dependem apenas dos parâmetros de entrada para produzir saídas.

- Consistência: Ao chamar uma função pura com os mesmos parâmetros, o resultado é sempre o mesmo.
- Ausência de Efeitos Colaterais: Funções puras não modificam variáveis ou causam efeitos indesejados fora de seu escopo.
- Retorno de Valor: Elas sempre retornam um valor, objeto ou outra função, tornando seu comportamento previsível e isolado.

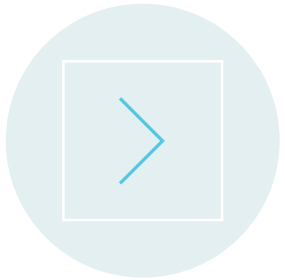
# Relação entre funções puras e dados imutáveis

## Dados imutáveis

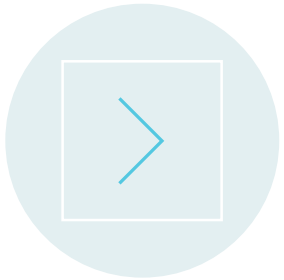
- Dados imutáveis na programação funcional não podem ser alterados após a criação.
- Em Python, ao passar uma lista como argumento, modificações afetam a lista original.
- Modificar dados diretamente pode levar a efeitos indesejados e saídas inconsistentes.
- Criar cópias de dados evita tais problemas e mantém resultados consistentes.

# Efeitos indesejáveis

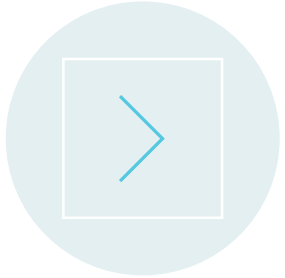
## Efeito colateral e estado da aplicação



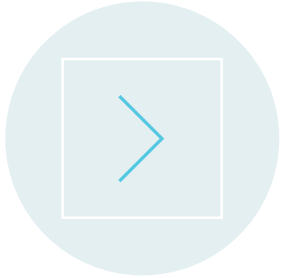
Efeito Colateral: Programação funcional visa evitar efeitos colaterais, como o ocorrido no script funcao3.py.



Independência do Estado: Funções puras dependem apenas dos parâmetros, garantindo resultados consistentes independentemente do estado da aplicação.



Evitar Comportamentos Variáveis: A programação funcional evita comportamentos diferentes para uma função com base no estado atual da aplicação.



Objetivo Principal: O objetivo principal da programação funcional é evitar o efeito colateral e garantir a previsibilidade do código.

# Outros tipos de função

## Funções de ordem superior

Funções de ordem superior: em programação funcional, funções que aceitam ou retornam outras funções.

Exemplo python: script funcao5.Py demonstra como criar funções que geram e retornam funções personalizadas.

Criação de funções internas: função "multiplicar_por" cria funções internas para executar operações específicas.

Flexibilidade e reutilização: esse conceito permite a criação de funções personalizadas e reutilizáveis para tarefas específicas.

# Outros tipos de função

## Funções lambda

Funções Lambda em Python: São funções anônimas definidas sem nome, usadas frequentemente como argumentos em funções de ordem superior.

Sintaxe Lambda: Utilizam a sintaxe lambda, seguida por argumentos separados por vírgula e uma expressão que é automaticamente retornada.

Exemplo de Uso: Por exemplo, uma função lambda para multiplicar dois números pode ser escrita como lambda a, b: a * b.

Armazenamento e Uso: Funções lambda podem ser atribuídas a variáveis e usadas como funções regulares, oferecendo flexibilidade e concisão no código.

# Boa prática de programação

## Não utilizar loops

Outra regra, ou boa prática, da programação funcional é não utilizar laços (for e while), mas sim composição de funções ou recursividade.

A função lambda exerce um papel fundamental nisso, como veremos a seguir.

Para facilitar a composição de funções e evitar loops, o Python disponibiliza diversas funções e operadores.

As funções internas mais comuns são map e filter.

# Boa prática de programação

## Maps

Função Map: Aplica uma função a cada elemento de um iterável, gerando um novo iterável com valores modificados.

Pureza e Ordem Superior: Função map é pura e de ordem superior, dependendo apenas de seus parâmetros e recebendo uma função como argumento.

Sintaxe: map(função, iterável1, iterável2...), onde a função é aplicada a cada item do iterável.

Exemplos: Mostram como triplicar os elementos de uma lista usando a função map.

# Boa prática de programação

## Filter

Função Filter: Filtra elementos de um iterável com base em uma função que retorna verdadeiro ou falso.

- Pureza e Ordem Superior: A função filter é pura e de ordem superior, depende apenas dos parâmetros e recebe uma função como argumento.
- Sintaxe: filter(função, iterável), retornando um novo iterável com os elementos que atendem ao critério da função.
- Exemplos: Demonstram como filtrar elementos ímpares de uma lista usando a função filter, gerando uma nova lista com os elementos desejados.

# Visão geral

## Introdução

- Avanço Tecnológico: Antigamente, a maioria dos dispositivos tinha apenas um processador, mas agora a maioria possui múltiplos núcleos.
- Necessidade de Desempenho: Aprender concorrência e paralelismo é crucial para otimizar programas e tirar proveito dos dispositivos modernos.

# Conceitos de programa e processo

- **Programa:** É estático e permanente, contendo um conjunto de instruções, percebido como passivo pelo sistema operacional.
- **Processo:** É dinâmico e efêmero, com seu estado em constante mudança durante a execução. Composto por código, dados e contexto.
- **Múltiplos Processos:** Executar o mesmo programa várias vezes gera processos distintos, cada um com dados e contexto de execução diferentes.

# Conceitos de concorrência e paralelismo

- **Concorrência:** Permite que vários processos compartilhem um único núcleo, tornando programas mais usáveis, ideal para sistemas multitarefa.

- **Paralelismo:** Acelera a execução de programas, executando múltiplas operações ao mesmo tempo, requerendo vários núcleos ou máquinas. Pode ser usado em GPUs.

# Conceito de Threads

## Threads e processos

- **Processos:** São instâncias de programas com espaço de memória próprio e podem executar em núcleos diferentes para melhorar o desempenho.
- **Threads:** Unidades menores de execução dentro de um processo, compartilham memória e acessam dados transparentemente.
- **Global Interpreter Lock (GIL):** No CPython, permite que apenas uma thread execute código Python por vez, protegendo objetos da linguagem.

# Conceito de Threads

## Criação de threads e processos

Criação de Threads e Processos em Python: Podemos criar threads e processos para executar funções simultaneamente.

Classe thread: Para criar uma thread, usamos a classe thread do módulo threading, definindo a função a ser executada como target e os argumentos como uma tupla em args.

Iniciando Threads: Threads são iniciadas chamando o método start() em seus objetos.

Classe process para Processos: Para criar processos, usamos a classe process do módulo multiprocessing, seguindo um processo semelhante à criação de threads.

# Conceito de Threads

## Múltiplas threads e processos

Múltiplas Threads e Processos: Os exemplos criam várias threads e processos para demonstrar diferenças de comportamento.

Compartilhamento de Variáveis: Threads compartilham variáveis globais, enquanto processos criam cópias independentes.

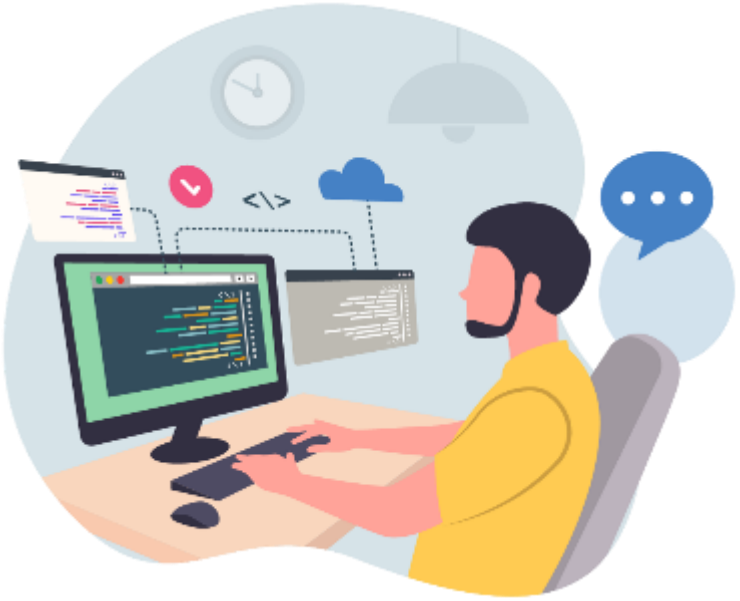
Aguardando Término: Para garantir que todos terminem, armazena-se referências e usa-se o método "join()".

Resultados: Threads compartilham variáveis, processos não.

# Conceito de Threads

## Travas (Lock)

- Travas (Lock): Evitam conflitos em operações compartilhadas por threads.
- Condição de Corrida: Problemas quando várias threads disputam recursos.
- Uso de Lock: Protege seções críticas, garantindo acesso exclusivo.



# Conceito de Threads

## Compartilhando variáveis entre processos

Variáveis Compartilhadas entre Processos: Podem ser criadas com a classe Value do módulo multiprocessing.

Evitar Condição de Corrida: Utiliza-se uma trava (lock) para proteger a operação em variáveis compartilhadas.

Uso do Método get_lock(): Disponível em objetos do tipo Value para garantir acesso seguro às variáveis compartilhadas.

Resultado Consistente: Evita problemas de condição de corrida e mantém a consistência dos dados.

# Visão geral

## Introdução

Diversas opções de linguagem de programação para aplicações web: PHP, ASP.NET, Ruby e Java.

PHP é a linguagem mais utilizada nos servidores, com 79% de participação, devido a CMS como WordPress e Joomla.

Python pode ser usado para desenvolver aplicações web com diversos frameworks disponíveis.

Frameworks em Python podem ser full-stack ou não full-stack.

# Visão geral

## Conceito

Flask é um framework micro para desenvolvimento web, oferecendo simplicidade e extensibilidade.

Desenvolvedores têm liberdade para escolher suporte a modelos, bibliotecas de banco de dados e manipulação de formulários.

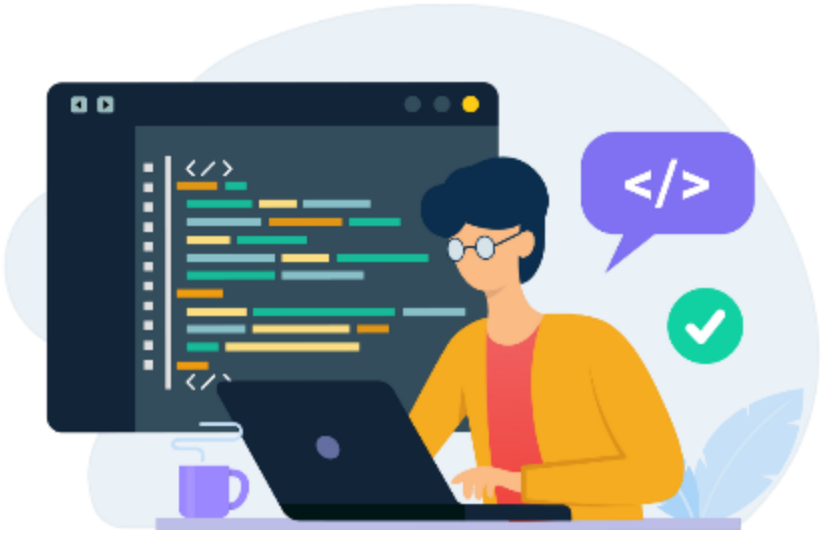
O Flask disponibiliza um servidor web interno para desenvolvimento, com configuração padrão na porta 5000.

O uso de rotas direciona as requisições do navegador para funções específicas, que retornam conteúdo para o usuário.

# Rotas e parâmetros

## Aprimorando rotas

- Flask é um framework que usa rotas para direcionar URLs para funções, permitindo respostas específicas.
- Rotas definem URLs para funções, como '/ola', e podem ser estabelecidas usando decoradores.
- As funções associadas a rotas retornam conteúdo, como "Página principal" ou "Olá, mundo," quando acessadas pelo navegador.



# Rotas e parâmetros

## Recebendo parâmetros

No Flask, o decorador de rota permite a passagem de parâmetros para funções, indicando-os entre < e > na URL.

O parâmetro da URL é capturado e passado para a função correspondente, como 'nome' na função 'ola_mundo’.

Pode-se definir um valor padrão para parâmetros na função, tornando-os opcionais na URL.

O mesmo ponto de entrada pode ter várias rotas diferentes para lidar com URLs variadas.

# Métodos HTTP e modelos

## Métodos HTTP

- O Flask permite especificar quais métodos HTTP uma rota aceitará com o parâmetro 'methods' no decorador @app.route().
- Os métodos HTTP comuns incluem GET, POST, PUT e DELETE, com o Flask respondendo a GET por padrão.
- O objeto 'request' fornece informações sobre a requisição, incluindo o método HTTP utilizado.
- A função associada à rota pode fornecer respostas diferentes com base no método HTTP, como mostrado no exemplo.

# Métodos HTTP e modelos

## Utilizando modelos

Flask utiliza templates (modelos) para simplificar a criação de páginas HTML.

Os modelos são arquivos de texto com recursos, incluindo escape automático, facilitando a segurança.

Marcadores especiais permitem injetar variáveis, criar laços e condicionantes nos modelos.

Os modelos devem ser colocados na pasta "templates" e podem ser renderizados com a função "render_template".

# Visão geral

## Introdução

A ciência computacional é o terceiro paradigma de pesquisa, complementando experimentação e teoria.

A Ciência de Dados é o quarto paradigma que utiliza dados para resolver problemas.

O processo KDD (Knowledge Discovery in Databases) envolve a identificação de padrões úteis em grandes conjuntos de dados.

Mineração de dados é uma técnica que extrai informações úteis e implícitas de bases de dados.

# Visão geral

## Introdução

O processo de KDD é basicamente composto por três grandes etapas:

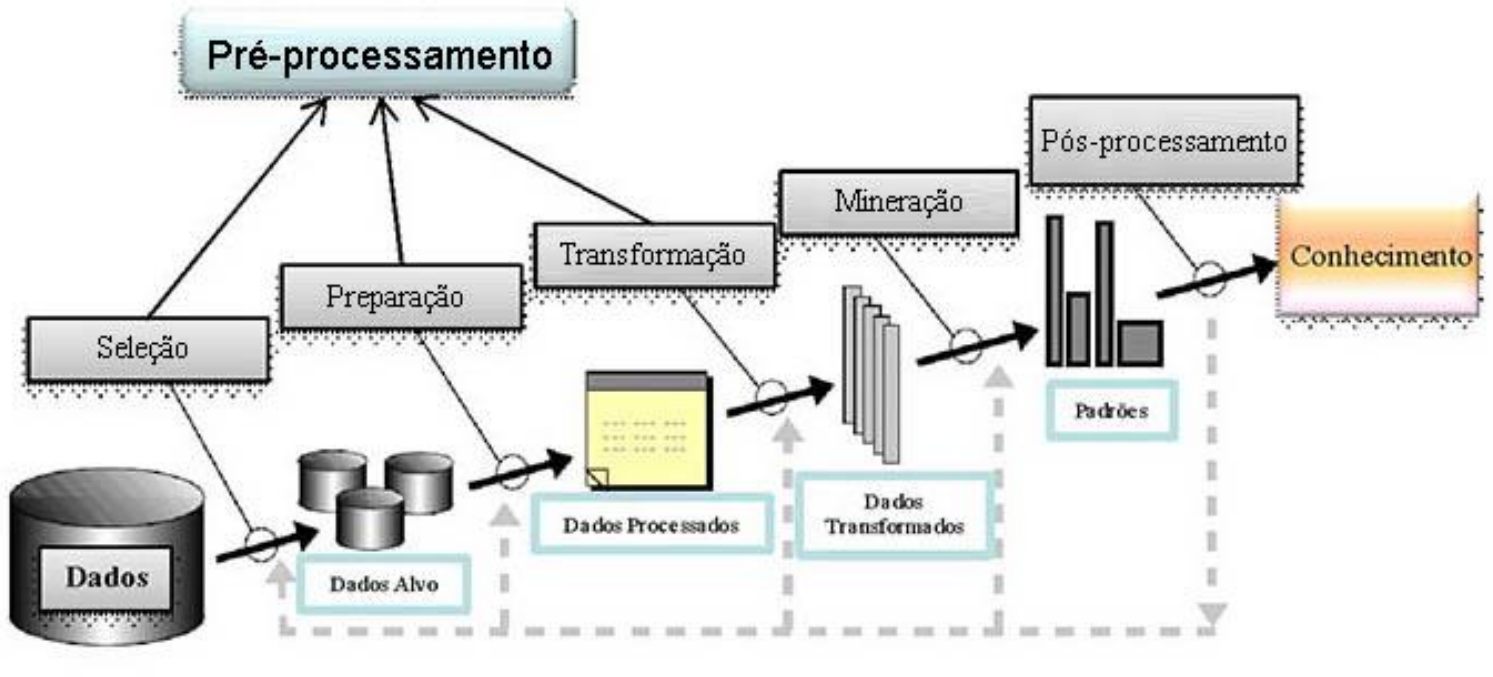
**Pré-processamento**

**Mineração de dados**

**Pós-processamento**

# Visão geral

## Introdução




Visão geral dos passos que compõem o processo de KDD.

# Visão geral

## Introdução

Algumas atividades envolvidas no pré-processamento são:

**Coleta e integração**

**Codificação**

**Construção de atributos**

**Limpeza de dados**

**A partição dos dados**

# Visão geral

## Introdução

Mineração de dados extrai padrões ocultos em grandes conjuntos de dados, tornando-os inteligíveis e úteis.

Algoritmos de mineração podem ser supervisionados (com aprendizado com dados rotulados) ou não supervisionados (descobrindo padrões por conta própria).

Algoritmos não supervisionados incluem regras de associação (para recomendações) e agrupamento (para agrupar objetos semelhantes).

Algoritmos supervisionados envolvem classificação (atribuir a uma classe) e regressão linear (estimar variáveis a partir de funções).

# Algoritmos supervisionados

## Supervisionado – regressão linear

- Supervisionado: Usando regressão linear para prever casos de dengue em uma série histórica.
- Dados de entrada: Planilha CSV com anos e casos (2001-2017).
- Processo: Aplicação da regressão linear com Pandas e Scikit-Learn.
- Resultado: Previsão do número de casos para 2018 e visualização em um gráfico.

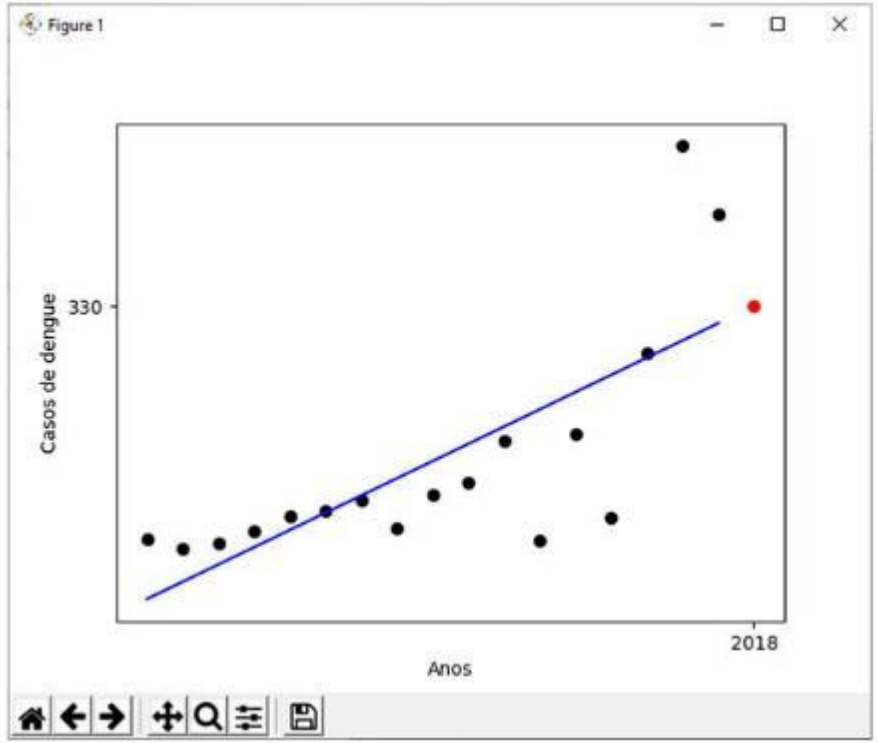



Gráfico da série dos casos de dengue.

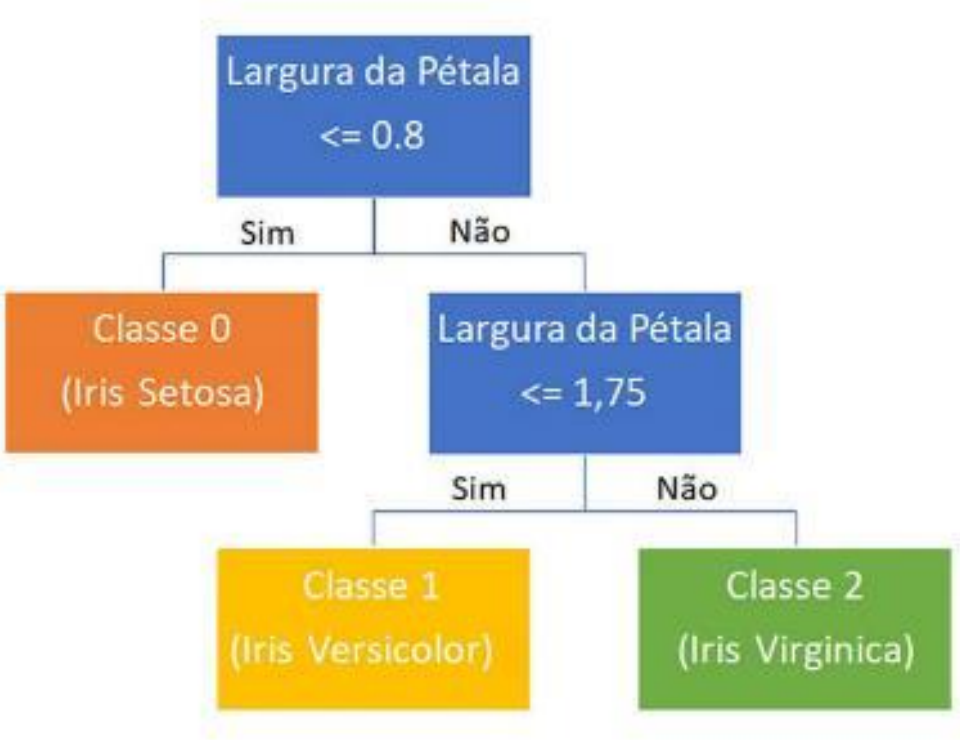
# Algoritmos supervisionados

## Supervisionado – classificação

- **Iris Dataset**: Conjunto de dados com informações de flores de três classes (setosa, versicolor, virginica).
- **Pré-processamento**: Coleta de dados, separação de características e rótulos, divisão em treino e teste.
- **Classificação**: Treinamento de algoritmos (árvore de decisão e SVM) para classificar flores e cálculo de acurácia.

# Algoritmos supervisionados

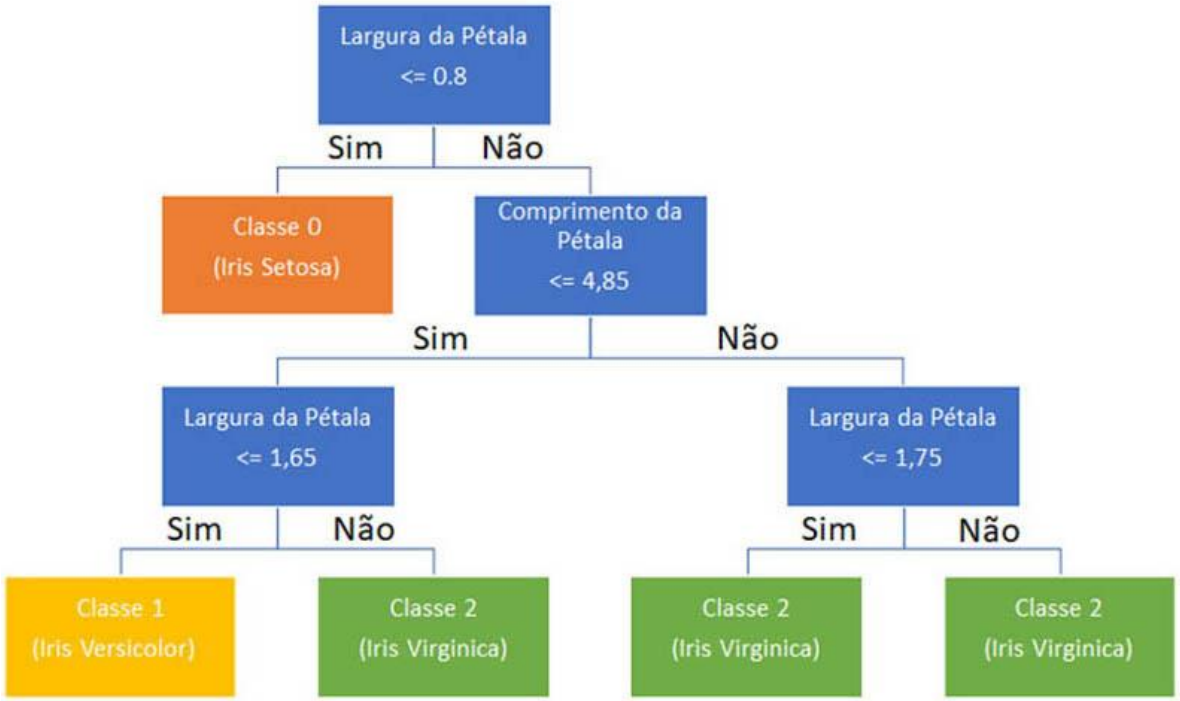
## Supervisionado – classificação




Representação da árvore de decisão com profundidade 2.

# Algoritmos supervisionados

## Supervisionado – classificação




Representação da árvore de decisão com profundidade 3.

# Algoritmos não supervisionados

## Não supervisionado – agrupamento

Algoritmos não supervisionados: Modelos que não requerem rótulos para aprender.

Agrupamento: Objetivo é reunir objetos com afinidade, maximizando a similaridade interna e minimizando a entre grupos.

Exemplos: K-means e Mean-shift são algoritmos de agrupamento.

K-medias: Algoritmo não supervisionado para agrupamento; separa amostras em grupos automaticamente.