INTRODUÇÃO AO PYTHON
SECRETARIA DE ESTADO DE SEGURANÇA PÚBLICA DO PARANÁ
CENTRO INTEGRADO DE COMANDO E CONTROLE EM CRISE REGIONAL

NIVELAMENTO EM *PYTHON*[1]

[1]*EQUIPE TÉCNICA CICCR*

1 de junho de 2023

1 Objetivos

O Nivelamento de ***Conceitos de Programação - Introdução ao Python*** tem como objetivo introduzir os participantes à linguagem de programação *Python*, fornecendo as habilidades básicas para escrever programas funcionais, compreender conceitos fundamentais e promover a resolução de problemas. Ao final do nivelamento, os alunos estarão preparados para aplicar seus conhecimentos em projetos pessoais ou profissionais.

Ao fim deste módulo, o aluno deverá dominar os seguintes tópicos:

- **Do que se trata programar**: Programar é o ato de escrever código de computador para instruir um computador a executar determinadas tarefas. Envolve a criação de algoritmos, o uso de linguagens de programação e a aplicação de lógica e resolução de problemas para desenvolver soluções computacionais.
- **O que significa o termo algoritmo**: Um algoritmo é uma sequência de passos ou instruções lógicas bem definidas para resolver um problema ou realizar uma tarefa. É um conjunto de regras que descreve uma série de etapas para obter um resultado desejado.

- **Onde encontrar informações sobre como instalar o *Python***: Para obter informações sobre como instalar o ***Python***, você pode visitar o site oficial do ***Python*** em *python.org*. Lá você encontrará documentação detalhada, tutoriais e guias de instalação para diferentes plataformas. Para um guia de consulta rápida, acesse **Guia de Instalação do *Python***.

- **Sintaxe e Estrutura de Dados**: A sintaxe em programação se refere às regras e convenções que determinam como o código fonte deve ser escrito em uma determinada linguagem de programação. A estrutura de dados se refere às formas de organizar e armazenar dados em um programa, como listas, tuplas, conjuntos, dicionários, entre outros.

- **Funções em *Python***: Em ***Python***, as funções são blocos de código reutilizáveis que realizam uma tarefa específica. Elas permitem dividir um programa em partes menores e mais gerenciáveis, facilitando a organização e a reutilização de código. As funções podem receber argumentos, executar um conjunto de instruções e retornar um resultado, se necessário.

- **Primeira Aplicação em Python**: A primeira aplicação em Python refere-se à criação do seu primeiro programa usando a linguagem ***Python***. Geralmente, é um programa simples que mostra uma saída na tela ou realiza uma tarefa básica para familiarizá-lo com a sintaxe e as estruturas básicas da linguagem.

- **Comentários na sua primeira aplicação**: Comentários são trechos de texto adicionados ao código fonte para fornecer explicações e informações adicionais sobre o código. Na sua primeira aplicação em ***Python***, os comentários podem ser usados para descrever o propósito do programa, fornecer instruções ou explicar partes específicas do código. Eles são ignorados pelo interpretador ***Python*** durante a execução do programa.

2 Introdução Ao Python

2.1 Programação e Algoritmos

O que é programar Programar é o ato de escrever um conjunto de instruções ou algoritmos que um computador pode executar para realizar uma tarefa específica. Envolve a criação de código em uma linguagem de programação que segue uma sintaxe e estrutura definidas. A programação permite que os desenvolvedores criem software, aplicativos, sites e sistemas que automatizam processos, resolvam problemas e forneçam funcionalidades úteis. Programar requer habilidades lógicas e analíticas para que as instruções sejam escritas de forma clara e precisa, permitindo que o computador execute as

tarefas conforme especificado. É uma habilidade essencial no mundo da tecnologia e desempenha um papel fundamental no avanço da computação e da inovação tecnológica.

Programar refere-se ao processo de criar um conjunto de instruções ou algoritmos para serem executados por um computador ou outro dispositivo eletrônico. É a arte de escrever código, que consiste em uma série de comandos precisos e lógicos, para que o computador execute tarefas específicas.

A programação permite que os desenvolvedores criem programas de software, aplicativos móveis, sites, jogos e uma ampla gama de soluções tecnológicas. É por meio da programação que os computadores são capazes de realizar diversas tarefas, desde cálculos complexos até a execução de operações de processamento de dados.

Ao programar, os desenvolvedores utilizam linguagens de programação, como ***Python***, *Java*, *C++*, *JavaScript*, entre outras. Essas linguagens fornecem uma sintaxe específica e um conjunto de regras que permitem aos programadores expressar suas instruções de forma clara e precisa para que o computador as execute corretamente.

A programação requer habilidades lógicas, criatividade e resolução de problemas. Os programadores devem entender o problema que desejam resolver, projetar uma solução eficiente e implementar o código necessário para alcançar o resultado desejado. Além disso, a depuração e o teste do código são partes essenciais do processo de programação para garantir que o programa funcione corretamente e produza os resultados esperados.

A programação é uma área em constante evolução, impulsionada pelo avanço da tecnologia e das demandas da sociedade. A capacidade de programar é uma habilidade valiosa em muitos setores, como desenvolvimento de software, ciência de dados, inteligência artificial, engenharia de software e muitos outros. Ela permite que as pessoas criem soluções inovadoras e automatizem tarefas para melhorar a eficiência e a produtividade em diversos campos de atuação.

O que é uma linguagem de Programação Uma linguagem de programação é uma forma de comunicação entre um programador e um computador. É um conjunto de regras e símbolos que permitem escrever instruções para que o computador execute tarefas específicas. Essas linguagens são projetadas para serem compreensíveis tanto para os seres humanos quanto para as máquinas.

As linguagens de programação fornecem um conjunto de palavras-chave, símbolos, estruturas gramaticais e regras semânticas que permitem aos programadores expressar suas instruções de forma estruturada e lógica. Elas permitem que os programadores escrevam algoritmos, que são sequências de passos que o computador deve seguir para resolver um determinado problema.

Existem muitas linguagens de programação disponíveis, cada uma com suas próprias características e finalidades. Algumas linguagens são de uso geral, como ***Python***, *Java*, *C++* e *JavaScript*,

e podem ser usadas para uma ampla variedade de aplicações. Outras são mais especializadas e foram projetadas para fins específicos, como linguagens para desenvolvimento *web*, análise de dados, inteligência artificial, entre outros.

As linguagens de programação podem ser classificadas em diferentes níveis de abstração. Linguagens de baixo nível, como ***Assembly***, estão mais próximas da linguagem de máquina e requerem um conhecimento mais detalhado do hardware do computador. Por outro lado, linguagens de alto nível, como ***Python***, permitem que os programadores expressem suas instruções de maneira mais próxima da linguagem humana, sendo menos dependentes dos detalhes de implementação.

Cada linguagem de programação tem suas próprias regras e características, mas todas compartilham o objetivo de permitir que os programadores desenvolvam soluções e criem programas que possam ser executados em um computador. O conhecimento de diferentes linguagens de programação permite aos programadores escolher a linguagem mais adequada para o problema em questão e expandir suas habilidades no desenvolvimento de software.

Estrutura de uma linguagem de programação A estrutura de uma linguagem de programação é composta por diversos elementos que definem como o código deve ser organizado e como as instruções devem ser escritas. Os principais elementos de estrutura de uma linguagem de programação são:

- ***Sintaxe:*** A sintaxe é o conjunto de regras que define a estrutura correta das instruções na linguagem. Ela determina como as palavras-chave, operadores, variáveis e outros elementos devem ser combinados para formar instruções válidas. A sintaxe define a ordem e a forma correta de escrever o código.

- ***Palavras-chave:*** As palavras-chave são termos reservados pela linguagem de programação que possuem um significado específico. Elas são usadas para definir estruturas de controle, declaração de variáveis, operações matemáticas e outras funcionalidades da linguagem. Exemplos de palavras-chave incluem `if`, `for`, `while`, `function`, `class`, entre outras.

- ***Tipos de dados:*** As linguagens de programação possuem diferentes tipos de dados, como números, texto, booleanos, `arrays`, objetos, entre outros. Os tipos de dados definem o tipo de valor que uma variável pode armazenar e as operações que podem ser realizadas com esses valores.

- ***Variáveis:*** As variáveis são utilizadas para armazenar valores na memória durante a execução do programa. Elas possuem um nome e um tipo de dados associado. As variáveis podem ser

usadas para armazenar informações temporárias, realizar cálculos e representar dados em geral.

- ***Estruturas de controle:*** As estruturas de controle permitem que o fluxo de execução do programa seja controlado. Elas incluem estruturas condicionais (como `if` e `else`) que permitem executar diferentes blocos de código dependendo de uma condição, e estruturas de repetição (como `for` e `while`) que permitem executar um bloco de código várias vezes.

- ***Funções e procedimentos:*** As funções e procedimentos são blocos de código que podem ser definidos e chamados em diferentes partes do programa. Eles permitem agrupar um conjunto de instruções em uma unidade lógica e reutilizável. As funções podem receber parâmetros e retornar valores.

- ***Bibliotecas e módulos:*** Muitas linguagens de programação possuem bibliotecas ou módulos que são conjuntos de código pré-existente que fornecem funcionalidades adicionais. Essas bibliotecas podem conter funções, classes e outras estruturas que podem ser importadas e utilizadas em um programa. Elas permitem estender as capacidades da linguagem de programação e facilitam o desenvolvimento de soluções complexas.

- ***Estruturas de dados:*** As estruturas de dados são formas organizadas de armazenar e manipular conjuntos de dados. Elas incluem `arrays`, listas, conjuntos, dicionários, entre outros. Cada estrutura de dados tem suas próprias características e métodos para realizar operações específicas, como adicionar, remover, buscar ou modificar elementos.

- ***Orientação a objetos:*** Muitas linguagens de programação são orientadas a objetos, o que significa que elas permitem a criação de classes e objetos. A programação orientada a objetos organiza o código em torno de objetos que possuem atributos (dados) e métodos (ações). Essa abordagem permite encapsular a lógica do programa e promover a reutilização de código.

- ***Tratamento de erros:*** As linguagens de programação têm mecanismos para lidar com erros e exceções que podem ocorrer durante a execução do programa. Isso permite que o desenvolvedor capture e trate essas situações inesperadas, evitando falhas ou comportamentos indesejados do programa.

Paradigmas de Programação Um paradigma, no contexto da programação de computadores, refere-se a um conjunto de conceitos, princípios e abordagens que definem a forma como os programas são estruturados, organizados e desenvolvidos. É uma forma de pensar e abordar a resolução de problemas na programação.

Cada paradigma de programação possui suas próprias regras, diretrizes e estilo de escrita de código. Ele define a estrutura básica do programa, as técnicas de solução de problemas e os padrões de design a serem seguidos.

Os paradigmas de programação fornecem diferentes formas de pensar sobre a estrutura do programa, o fluxo de controle, o gerenciamento de estado e a interação entre os componentes do sistema. Eles influenciam a maneira como os programadores abordam a resolução de problemas e as ferramentas e técnicas que eles utilizam.

É importante mencionar que não existe um único paradigma ***"melhor"*** ou ***"correto"***. Cada paradigma tem suas vantagens e desvantagens, e a escolha do paradigma adequado depende do tipo de problema a ser resolvido, das restrições e das preferências pessoais. Além disso, muitas linguagens de programação permitem a combinação de múltiplos paradigmas, permitindo maior flexibilidade na forma como os programas são escritos.

Os paradigmas de programação mais comuns incluem a programação imperativa, orientada a objetos, funcional, lógica, estruturada, entre outros. Cada um deles oferece uma abordagem única para a construção de programas e tem suas próprias técnicas e conceitos específicos.

Alguns paradigmas de programação são:

- ***Programação Imperativa:*** É o paradigma mais tradicional e amplamente utilizado. Nesse paradigma, os programas são estruturados em sequências de instruções que modificam o estado do programa. Ele se baseia na ideia de que um programa é uma série de comandos que são executados em ordem.

- ***Programação Orientada a Objetos (POO)***: Nesse paradigma, os programas são organizados em torno de objetos, que são instâncias de classes. Os objetos possuem atributos (dados) e métodos (ações) que podem interagir uns com os outros. A POO enfatiza a reutilização de código, encapsulamento e modularidade.

- ***Programação Funcional:*** Nesse paradigma, o foco está nas funções. Os programas são escritos em termos de funções puras, que não possuem efeitos colaterais e retornam um valor com base em seus argumentos. A programação funcional enfatiza a imutabilidade dos dados e o uso de funções de ordem superior.

- ***Programação Lógica:*** Nesse paradigma, os programas são escritos em termos de regras lógicas. A programação lógica se baseia no uso de predicados e inferência lógica para resolver problemas. A linguagem de programação Prolog é um exemplo comum de programação lógica.

- ***Programação Estruturada:*** É um paradigma que enfatiza a organização do código em estruturas bem definidas, como sequências, laços e condicionais. A programação estruturada busca evitar o uso de desvios incondicionais (como o "goto") e promover a legibilidade e a manutenibilidade do código.

Além desses, existem outros paradigmas de programação, como programação procedural, programação orientada a eventos, programação concorrente, entre outros. Cada paradigma tem suas próprias vantagens e é mais adequado para determinados tipos de problemas. Muitas linguagens de programação permitem a combinação de paradigmas, permitindo ao programador escolher a abordagem mais adequada para cada situação.

Programação Estruturada A programação estruturada é um paradigma de programação que se baseia na organização do código em estruturas lógicas bem definidas. Nesse estilo de programação, o programa é dividido em blocos de código chamados de procedimentos ou funções, que contêm uma sequência lógica de instruções para executar uma determinada tarefa.

A programação estruturada segue alguns princípios-chave:

- ***Sequência:*** As instruções são executadas em ordem sequencial, de cima para baixo, sem desvios ou saltos incondicionais.

- ***Seleção:*** A seleção é realizada por meio de estruturas de controle condicionais, como o ‘‘`if-else`’’ ou ‘‘`switch-case`’’, permitindo que diferentes blocos de código sejam executados com base em condições específicas.

- ***Repetição:*** A repetição é realizada por meio de estruturas de controle de *loop*, como ‘‘`for`’’ e ‘‘`while`’’, permitindo que um bloco de código seja executado várias vezes com base em uma condição específica.

- ***Modularidade:*** O código é dividido em procedimentos ou funções independentes, que podem ser chamados de outros locais do programa. Essa abordagem promove a reutilização de código, facilita a compreensão e a manutenção do programa.

A programação estruturada tem como objetivo principal tornar o código mais legível, organizado e fácil de entender. Ela evita o uso excessivo de desvios incondicionais, como "goto", que podem tornar o código confuso e difícil de dar manutenção. Além disso, a programação estruturada facilita a identificação de erros e a depuração do código.

Ao seguir os princípios da programação estruturada, é possível escrever programas mais eficientes, modulares e robustos. No entanto, ela possui algumas limitações em lidar com problemas complexos,

especialmente quando se trata de lidar com grandes volumes de dados ou lógicas mais avançadas. Nesses casos, paradigmas como a programação orientada a objetos podem ser mais adequados.

Programação Procedural A programação procedural é um paradigma de programação que se concentra em estruturar o código em procedimentos ou funções. Ela se baseia na ideia de que um programa é composto por uma sequência de instruções que são executadas em ordem, e o controle do fluxo do programa é realizado por meio de estruturas de controle, como loops e condicionais.

Na programação procedural, o foco está na decomposição do programa em procedimentos ou funções menores e mais gerenciáveis, que podem ser reutilizados em diferentes partes do programa. Cada procedimento é uma sequência de instruções que realiza uma tarefa específica e pode receber argumentos (parâmetros) e retornar um valor.

Uma das principais características da programação procedural é o uso de variáveis, que armazenam dados temporários e podem ser manipuladas por meio de operações como atribuição, cálculos e comparações.

A programação procedural segue uma abordagem top-down, onde o programa principal é dividido em procedimentos e subprocedimentos hierarquicamente. Essa divisão modular facilita a compreensão, a manutenção e a reutilização do código.

Linguagens de programação como C, Pascal e Fortran são exemplos de linguagens que suportam a programação procedural. No entanto, muitas linguagens modernas combinam elementos da programação procedural com outros paradigmas, como a programação orientada a objetos, permitindo uma abordagem mais flexível e modular na construção de programas.

Programação Orientada a Objetos A programação orientada a objetos (*POO*) é um paradigma de programação que organiza o código em torno de objetos, que são instâncias de classes. Na *POO*, um objeto é uma entidade que contém dados e comportamentos relacionados.

O paradigma da programação orientada a objetos baseia-se em quatro princípios fundamentais:

- ***Abstração:*** Permite representar objetos do mundo real no código, identificando suas características relevantes e ignorando detalhes irrelevantes. Isso ajuda a modelar e compreender o problema a ser resolvido.

  Encapsulamento: Envolve os dados e os comportamentos relacionados em uma única unidade chamada classe. A classe define a estrutura e o comportamento de um objeto, ocultando os detalhes internos e fornecendo interfaces para interagir com o objeto.

  Herança: Permite criar novas classes a partir de classes existentes, herdeiras das características

e comportamentos da classe pai. A herança facilita a reutilização de código e a criação de hierarquias de classes.

Polimorfismo: Permite que objetos de diferentes classes sejam tratados de maneira uniforme, por meio do uso de métodos com o mesmo nome, mas com comportamentos específicos para cada classe. Isso aumenta a flexibilidade e a extensibilidade do código.

Na programação orientada a objetos, as classes são utilizadas para definir objetos, especificando suas propriedades (atributos) e comportamentos (métodos). Os objetos são criados a partir das classes e podem interagir uns com os outros por meio de troca de mensagens.

O estado de um objeto é definido pelos valores dos seus atributos ou propriedades. Os atributos representam as características ou informações que um objeto possui. Por exemplo, um objeto ***"Carro"*** pode ter atributos como ***"marca"***, ***"modelo"***, ***"cor"*** e ***"velocidade"***. O estado do objeto seria determinado pelos valores desses atributos, como $marca =' Toyota'$, $modelo =' Corolla'$, $cor =' vermelho'$ e $velocidade = 60$.

O comportamento de um objeto é definido pelos métodos associados a ele. Os métodos são as ações ou operações que um objeto pode realizar. Eles representam o comportamento do objeto e podem ser usados para modificar o estado do objeto ou realizar cálculos. Continuando com o exemplo do objeto ***"Carro"***, ele pode ter métodos como $acelerar()$, $frear()$, $ligar()$ e $desligar()$. Esses métodos definem as ações que o carro pode executar, afetando seu estado, como aumentar a velocidade, diminuir a velocidade, ligar o motor e desligar o motor.

Linguagens de programação como *Java*, *C++*, *Python* e *C#* são exemplos de linguagens que suportam a programação orientada a objetos. A *POO* é amplamente utilizada devido à sua capacidade de modelar problemas complexos de forma mais clara, modular e reutilizável, facilitando o desenvolvimento e a manutenção de software.

O que são algoritmos O estudo e uso de algoritmos remonta a milhares de anos. A palavra *"algoritmo"* deriva do nome de um matemático persa do século IX, *Al-Khwarizmi*, que foi um dos primeiros a sistematizar métodos de resolução de equações lineares e quadráticas. No entanto, a ideia de algoritmo e seu uso prático são anteriores a *Al-Khwarizmi*.

Antes do advento dos computadores, os algoritmos eram resolvidos manualmente por matemáticos e cientistas. Grandes avanços foram feitos por matemáticos notáveis, como ***Euclides***, que desenvolveu o algoritmo para encontrar o maior divisor comum de dois números (Algoritmo de Euclides), e ***Isaac Newton***, que desenvolveu algoritmos para cálculo diferencial e integral.

Com o avanço da tecnologia e a invenção dos computadores, o estudo e desenvolvimento de

algoritmos expandiu-se significativamente. Durante a Segunda Guerra Mundial, os primeiros computadores foram construídos para auxiliar nos cálculos e criptografia. Esse período marcou o início da programação de computadores e do desenvolvimento de algoritmos para resolver problemas complexos.

Nos anos seguintes, a ciência da computação emergiu como uma disciplina acadêmica, e muitos pesquisadores contribuíram para o desenvolvimento de algoritmos eficientes e otimizados. Diversos algoritmos famosos foram criados nessa época, como o algoritmo de classificação rápida (*quicksort*), o algoritmo de busca binária e o algoritmo de *Dijkstra* para encontrar o caminho mais curto em um grafo.

À medida que os computadores se tornaram mais poderosos e acessíveis, o campo da ciência da computação se expandiu rapidamente, levando ao desenvolvimento de algoritmos mais avançados e sofisticados. Algoritmos de aprendizado de máquina, algoritmos de criptografia, algoritmos de compressão de dados e algoritmos de otimização são apenas alguns exemplos das áreas em que os algoritmos têm sido amplamente utilizados.

Hoje em dia, os algoritmos estão presentes em quase todos os aspectos de nossas vidas, desde a pesquisa na web até as transações financeiras. O estudo contínuo e a evolução dos algoritmos são fundamentais para acompanhar os avanços tecnológicos e resolver problemas cada vez mais complexos.

Algoritmo é uma sequência de passos usada para resolver um problema. A sequência apresenta um método único de abordar uma questão, fornecendo uma solução específica. Um algoritmo não precisa representar conceitos matemáticos ou lógicos, embora as apresentações de algoritmos frequentemente se enquadrem nessa categoria. Algumas fórmulas especiais também são algoritmos, como a fórmula quadrática. Para que um processo represente um algoritmo, ele deve ser:

- ***Finito:*** O algoritmo deve eventualmente resolver o problema. Este livro discute problemas com uma solução conhecida para que você possa avaliar se um algoritmo resolve o problema corretamente.

- ***Bem-definido:*** A série de passos deve ser precisa e apresentar etapas compreensíveis. Especialmente porque os computadores estão envolvidos no uso de algoritmos, o computador deve ser capaz de entender as etapas para criar um algoritmo utilizável.

- ***Efetivo:*** Um algoritmo deve resolver todos os casos do problema para o qual alguém o definiu. Um algoritmo deve sempre resolver o problema que precisa ser resolvido. Embora se deva antecipar algumas falhas, a ocorrência de falhas é rara e ocorre apenas em situações aceitáveis para o uso pretendido do algoritmo.

Para criar um algoritmo, siga estas etapas:

- ***Compreenda o problema:*** Analise e compreenda claramente o problema que deseja resolver. Identifique os requisitos, restrições e objetivos do problema.
- ***Divida o problema em etapas menores:*** Quebre o problema em etapas menores e mais gerenciáveis. Isso ajuda a simplificar o problema e facilita a resolução passo a passo.
- ***Identifique as entradas e saídas:*** Determine quais informações são necessárias como entrada para o algoritmo e qual é a saída esperada após a execução do algoritmo.
- ***Projete a lógica do algoritmo:*** Desenvolva a lógica do algoritmo, definindo a sequência de passos que devem ser seguidos para resolver o problema. Use estruturas de controle, como loops e condicionais, para controlar o fluxo do algoritmo.
- ***Teste e revise o algoritmo:*** Teste o algoritmo com diferentes conjuntos de dados de entrada para garantir que ele esteja funcionando corretamente. Faça ajustes e revisões conforme necessário para melhorar a eficiência e corrigir erros.
- ***Documente o algoritmo:*** Escreva o algoritmo de forma clara e organizada. Utilize comentários para explicar o propósito de cada etapa e fornecer informações adicionais para facilitar a compreensão.
- ***Implemente o algoritmo:*** Traduza o algoritmo para a linguagem de programação escolhida. Escreva o código correspondente para cada etapa do algoritmo.
- ***Teste e depure o código:*** Execute o código implementado, fornecendo diferentes dados de entrada e verificando se a saída corresponde ao esperado. Identifique e corrija quaisquer erros ou problemas de execução.
- ***Otimize o algoritmo:*** Analise o desempenho do algoritmo e faça melhorias para torná-lo mais eficiente, reduzindo o tempo de execução ou a utilização de recursos.

Lembre-se de que a criação de um algoritmo requer prática e experiência. À medida que você ganha mais familiaridade com a programação e a resolução de problemas, se tornará mais fácil criar algoritmos eficazes e eficientes.

Representação de Algoritmos Um algoritmo pode ser apresentado de diversas formas, dependendo do contexto em que está sendo utilizado. Aqui estão algumas das maneiras mais comuns de representar um algoritmo:

- ***Descrição Narrativa:*** Nesse formato, o algoritmo é descrito em linguagem natural, usando frases e parágrafos para explicar os passos necessários para resolver um problema. Essa descrição pode incluir exemplos e detalhes adicionais para facilitar o entendimento.
- ***Fluxograma:*** Um fluxograma é uma representação gráfica do algoritmo. Nele, são usadas formas geométricas, como retângulos, losangos e setas, para representar os diferentes passos do algoritmo. As setas indicam a sequência de execução, e as formas geométricas contêm as instruções ou ações a serem realizadas.
- ***Pseudocódigo***: O pseudocódigo é uma forma intermediária entre a descrição narrativa e a linguagem de programação real. Ele usa uma mistura de linguagem natural e elementos de programação para descrever os passos do algoritmo. O pseudocódigo não segue a sintaxe de nenhuma linguagem específica, mas é uma forma mais estruturada e próxima da programação real.
- ***Linguagem de Programação:*** Um algoritmo também pode ser apresentado diretamente em uma linguagem de programação real. Nesse caso, o algoritmo é escrito usando a sintaxe e as estruturas da linguagem escolhida. Essa forma de apresentação é mais adequada para programadores experientes ou quando se deseja implementar o algoritmo em um ambiente específico.

Independentemente da forma de apresentação escolhida, um algoritmo deve ser claro, preciso e completo. Deve descrever em detalhes os passos necessários para resolver um problema, indicando a sequência de ações, as condições de controle e os dados envolvidos. A escolha da forma de apresentação depende do público-alvo e do contexto em que o algoritmo será utilizado.

Fluxogramas Fluxogramas são diagramas visuais que representam a sequência de passos ou fluxo de um processo, algoritmo ou sistema. Eles são amplamente utilizados na programação e em outras áreas para representar de forma clara e visual as etapas e decisões envolvidas em um processo.

Os fluxogramas permitem visualizar a lógica e a estrutura de um processo, tornando mais fácil entender e comunicar o fluxo das operações. Eles são ferramentas valiosas para projetar, documentar e depurar algoritmos e sistemas complexos, além de facilitar a colaboração entre membros de uma equipe.

Um fluxograma usa símbolos gráficos para representar diferentes elementos e estruturas do algoritmo. Aqui estão alguns dos símbolos comumente usados em um fluxograma:

- ***Início/Finalização:*** Representado por um oval, indica o início e o término do fluxograma.
- ***Processamento:*** Representado por um retângulo, indica uma ação ou operação a ser executada. Pode ser uma atribuição de valor, cálculo matemático, chamada de função, etc.
- ***Decisão:*** Representado por um losango, é usado para tomar uma decisão com base em uma condição. O fluxo se divide em diferentes caminhos, dependendo do resultado da condição.
- ***Entrada/Saída:*** Representado por um paralelogramo, indica a entrada de dados ou a exibição de resultados.
- ***Conector:*** Representado por um círculo pequeno, é usado para conectar diferentes partes do fluxograma quando há um desvio de fluxo.
- ***Setas:*** Usadas para indicar a direção do fluxo, conectando os diferentes símbolos.

Além desses símbolos básicos, existem variações e símbolos adicionais que podem ser usados em fluxogramas mais complexos, dependendo da notação adotada. Esses símbolos fornecem uma representação visual clara e compreensível do fluxo de um algoritmo, facilitando a compreensão e análise do mesmo.

Além dos símbolos mencionados anteriormente, aqui estão alguns símbolos adicionais comumente encontrados em fluxogramas:

- ***Loop (Laço):*** Representado por um retângulo com duas linhas diagonais internas, indica a repetição de um conjunto de instruções até que uma condição seja atendida.
- ***Sub-rotina:*** Representado por um retângulo com cantos arredondados, indica uma sequência de instruções que são agrupadas e podem ser chamadas várias vezes de diferentes partes do fluxograma.
- ***Armazenamento de Dados:*** Representado por um paralelogramo com uma linha curva na parte superior, indica o armazenamento ou recuperação de dados em um meio físico, como um banco de dados ou arquivo.
- ***Conexões:*** São setas que conectam os diferentes símbolos para indicar o fluxo de execução, seja sequencialmente, por decisões ou ciclos.

Pseudocódigo Pseudocódigo é uma forma de representação textual de um algoritmo, utilizando uma linguagem simplificada e próxima da linguagem natural. É uma forma de escrever instruções de forma mais clara e compreensível, antes de serem traduzidas para uma linguagem de programação específica.

O pseudocódigo não segue uma estrutura rígida como uma linguagem de programação real, mas geralmente utiliza palavras-chave simples e convenções para indicar as instruções e estruturas de controle. Ele é usado para expressar a lógica de um algoritmo de forma genérica, sem se preocupar com a sintaxe de uma linguagem de programação específica.

Um exemplo simples de pseudocódigo seria:

```
Início
        Ler valor1
        Ler valor2
        Soma <- valor1 + valor2
        Escrever Soma
Fim
```

Nesse exemplo, temos um pseudocódigo que lê dois valores, realiza a soma e exibe o resultado. As instruções são escritas em uma sequência lógica, utilizando palavras-chave como ***"Ler"***, ***"Escrever"*** e o operador de atribuição **"<-"** para indicar a atribuição de valores a variáveis.

O pseudocódigo é uma ferramenta útil para planejar e estruturar algoritmos antes de implementá-los em uma linguagem de programação real. Ele permite a expressão de ideias de forma mais simples e clara, facilitando o entendimento e a comunicação entre desenvolvedores. Além disso, o pseudocódigo é independente de qualquer linguagem de programação específica, tornando-o mais flexível e acessível para diferentes pessoas e contextos.

Outro exemplo de pseudocódigo, comumente encontrado em artigos, é dado a seguir:

3 Linguagem Python

Python é uma linguagem de programação de alto nível, interpretada e de propósito geral. Foi criada por *Guido van Rossum* e lançada pela primeira vez em 1991. ***Python*** se destaca por sua simplicidade e legibilidade de código, tornando-a uma linguagem acessível para iniciantes, ao mesmo tempo em que oferece recursos avançados para desenvolvedores experientes.

Algumas características distintas do ***Python*** incluem:

Algorithm 1 An algorithm with caption

Require: $n \geq 0$
Ensure: $y = x^n$
  $y \leftarrow 1$
  $X \leftarrow x$
  $N \leftarrow n$
  while $N \neq 0$ **do**
    if N is even **then**
      $X \leftarrow X \times X$
      $N \leftarrow \frac{N}{2}$ ▷ This is a comment
    else if N is odd **then**
      $y \leftarrow y \times X$
      $N \leftarrow N - 1$
    end if
  end while

- ***Sintaxe clara e concisa: Python*** enfatiza a legibilidade do código, utilizando uma sintaxe limpa e de fácil compreensão. Isso torna o desenvolvimento mais rápido e menos propenso a erros.

- ***Tipagem dinâmica: Python*** é uma linguagem de tipagem dinâmica, o que significa que as variáveis não precisam ser declaradas com um tipo específico. Os tipos são inferidos em tempo de execução, proporcionando flexibilidade ao programador.

- ***Amplas bibliotecas e módulos: Python*** possui uma vasta biblioteca padrão que abrange uma ampla gama de áreas, desde processamento de texto e manipulação de arquivos até desenvolvimento web e científico. Além disso, existem inúmeras bibliotecas de terceiros disponíveis, como o NumPy, Pandas e TensorFlow, que estendem ainda mais as funcionalidades do Python.

- ***Suporte multiplataforma: Python*** é executado em diferentes plataformas, incluindo *Windows*, *macOS* e várias distribuições de Linux. Isso permite que os programas escritos em *Python* sejam facilmente portados entre diferentes sistemas operacionais.

- ***Orientação a objetos: Python*** suporta programação orientada a objetos, permitindo a definição de classes e objetos, encapsulamento de dados e reutilização de código.

O ***Python*** é uma linguagem de programação em alta principalmente porque possui todos os elementos certos para o tipo de desenvolvimento de software que impulsiona o mundo do desenvolvimento de software nos dias de hoje. A aprendizagem de máquina, a robótica, a inteligência artificial e a ciência de dados são as principais tecnologias atualmente e para o futuro previsível. O ***Python*** é popular principalmente porque já possui muitas capacidades nessas áreas, enquanto muitas linguagens mais antigas ficam para trás nessas tecnologias.

Assim como existem diferentes marcas de pasta de dente, xampu, carros e praticamente qualquer outro produto que você possa comprar, existem diferentes marcas de linguagens de programação com nomes como *Java*, *C*, *C++* (pronunciado C plus plus) e *C#* (pronunciado C sharp). Todas são linguagens de programação, assim como todas as marcas de pasta de dente são pastas de dente. As principais razões citadas para a atual popularidade do ***Python*** são:

1. O ***Python*** é relativamente fácil de aprender.
2. Tudo o que você precisa aprender (e fazer) em ***Python*** é gratuito.
3. O ***Python*** oferece mais ferramentas prontas para as tecnologias atuais mais populares, como ciência de dados, aprendizagem de máquina, inteligência artificial e robótica, do que a maioria das outras linguagens.

3.1 Sintaxe do *Python*

Sintaxe é um conjunto de regras e estruturas gramaticais que definem a forma correta de escrever um determinado código ou linguagem. Na programação, a sintaxe é fundamental para que o código seja compreensível e interpretado corretamente pelo compilador ou interpretador.

Cada linguagem de programação tem sua própria sintaxe, com regras específicas que determinam como as instruções devem ser escritas e organizadas. Essas regras geralmente incluem elementos como palavras-chave, operadores, símbolos, estruturas de controle e convenções de formatação.

A sintaxe define a estrutura básica do código, como a ordem das instruções, a maneira como os blocos de código são delimitados e a forma como os elementos do código são combinados. Ela também define como os elementos individuais do código devem ser escritos, como a sintaxe correta para declarar variáveis, chamar funções, criar loops, realizar operações matemáticas, entre outros.

Quando a sintaxe de um código está incorreta, ocorrem erros de sintaxe que impedem o compilador ou interpretador de entender o código. Esses erros são geralmente indicados por mensagens de erro que informam onde ocorreu o problema e qual é a natureza do erro.

A sintaxe do ***Python*** é conhecida por ser clara e legível. Algumas características principais da sintaxe do Python incluem:

Exemplo:

- ***Indentação significativa: Python*** utiliza a indentação para delimitar blocos de código em vez de usar chaves ou palavras-chave especiais. Isso significa que a consistência na indentação é fundamental para a correta estruturação do código.

```
1
2  if x > 5:
3          print("x é maior que 5")
4  else:
5          print("x é menor ou igual a 5")
6
```

- ***Comentários:*** Comentários são trechos de texto que explicam o código e são ignorados pelo interpretador do ***Python***. Eles são precedidos pelo caractere "#" e ajudam a documentar o código para facilitar a compreensão. Nos comentários multilinha, o comentário deve estar entre três aspas simples ou duplas.

  Exemplo:

```
1
2  """
3  Introdução ao Python - CICCR - Exemplo de Comentário Multilinha
4  @author Augusto Mathias Adams <augusto.mathias@sesp.pr.gov.br>
5  MIT License
6  ..........
7  """
8
9  # Esta é uma linha de comentário
10
```

- ***Variáveis e Atribuição:*** As variáveis em ***Python*** são usadas para armazenar valores. Em Python, as variáveis são criadas atribuindo valores a elas. Não é necessário declarar explicitamente o tipo da variável, pois a tipagem é dinâmica. A atribuição de valores às variáveis é feita usando o sinal de igual '='.

  Exemplo:

```
1
2  x = 10
3  nome = "Maria"
4
```

- ***Estruturas de Controle:*** As estruturas de controle em ***Python*** são usadas para controlar o fluxo de execução de um programa. Elas permitem que você tome decisões com base em condições e repita a execução de um bloco de código várias vezes. As estruturas de controle em ***Python*** são:

  – Estruturas Condicionais:

  if: Executa um bloco de código se uma condição for verdadeira.

  elif: Permite testar condições adicionais se as condições anteriores forem falsas.

  else: Executa um bloco de código se todas as condições anteriores forem falsas.

  Exemplo:

```
x = 5
if x > 0:
        print("x é positivo")
elif x == 0:
        print("x é igual a zero")
else:
        print("x é negativo")

```

  – ***Estruturas de Repetição:***

  for: Itera sobre uma sequência (como uma lista, tupla ou string) ou um objeto iterável por um número específico de vezes.

  while: Executa um bloco de código repetidamente enquanto uma condição for verdadeira. Exemplo:

```

# Exemplo Simples de Loop for - itera sob uma lista de números
# consecutivos fornecida pela função range()
for i in range(5):
        print(i)

# Exemplo Simples de loop while
# utilizado somente quando a variável da condição é alterada dentro do loop
x = 10
while x > 0:
        print(x)
        x -= 1


```

– ***Controle de Loop:***

break: Encerra imediatamente o loop em que está sendo executado.

continue: Pula para a próxima iteração do loop, ignorando o restante do código dentro do bloco do loop.

Exemplo:

```
1
2  # Exemplo Simples de loop for com break
3
4  for i in range(10):
5          if i == 5:
6                  break
7          print(i)
8
9  # Exemplo Simples de loop for com continue
10 for i in range(10):
11         if i % 2 == 0:
12                 continue
13         print(i)
14
```

- ***Funções:*** Como mencionado anteriormente, as funções são blocos de código reutilizáveis que realizam tarefas específicas. A sintaxe básica para definir uma função é usando a palavra-chave `def`, seguida pelo nome da função, parênteses contendo os parâmetros (opcional) e dois pontos (:). O corpo da função é indentado.

  Exemplo:

```
1
2  # definição de uma função simples
3  def saudacao():
4          # o corpo da função é sempre indentado
5          print("Olá, mundo!")
6
7  # Chamando a função
8  saudacao()
9
```

3.1.1 Estruturas de Controle

Estruturas Condicionais As estruturas condicionais em ***Python*** são usadas para controlar o fluxo de execução do programa com base em certas condições. Elas permitem que um trecho específico de código seja executado somente se uma condição for atendida. Existem duas estruturas condicionais principais em ***Python***: o `if` e o `if-else`.

Cláusula `if`: O `if` é a estrutura condicional básica em Python. Ela verifica uma condição e, se a condição for avaliada como verdadeira, o bloco de código indentado abaixo do `if` é executado. Caso contrário, o bloco de código é ignorado.

Exemplo:

```
1
2  # conferindo se a idade é maior ou igual a 18
3
4  if idade >= 18:
5          print("Você é maior de idade.")
6
```

Nesse exemplo, se a variável `idade` for igual ou maior que 18, a mensagem *"Você é maior de idade"* será exibida.

Cláusula `if-else`: O `if-else` é uma estrutura condicional que permite especificar um bloco de código para ser executado quando a condição é verdadeira e outro bloco de código para ser executado quando a condição é falsa.

```
1
2  # conferindo se a idade é maior ou igual a 18
3
4  if idade >= 18:
5          # sim, você é maior de idade
6          print("Você é maior de idade.")
7  else:
8          # não, te peguei!!!
9          print("Você é menor de idade.")
10
11
```

Nesse exemplo, se a variável `idade` for igual ou maior que 18, a mensagem *"Você é maior de idade"* será exibida. Caso contrário, a mensagem *"Você é menor de idade"* será exibida.

Cláusula `elif` O `elif` é uma abreviação de `else if` e permite verificar múltiplas condições em sequência. Ele é usado quando existem mais de duas possibilidades de resultados.

Exemplo:

```
1
2  if nota >= 90:
3          print("Você obteve uma nota A.")
4  elif nota >= 80:
5          print("Você obteve uma nota B.")
6  elif nota >= 70:
7          print("Você obteve uma nota C.")
8  else:
9          print("Você obteve uma nota abaixo de C.")
10
```

Nesse exemplo, dependendo do valor da variável `nota`, uma mensagem correspondente será exibida.

É importante observar que a indentação é fundamental na estruturação correta das estruturas condicionais em ***Python***. A indentação define os blocos de código que devem ser executados em determinadas condições.

Bônus: cláusula `match` Uma declaração `match` recebe uma expressão e compara o seu valor com padrões sucessivos fornecidos em um ou mais blocos de casos (`case blocks`). Isso é superficialmente semelhante a uma declaração `switch` em *C*, *Java* ou *JavaScript* (e muitas outras linguagens), mas é mais semelhante a correspondência de padrões em linguagens como *Rust* ou *Haskell*. Apenas o primeiro padrão que corresponde é executado e também é possível extrair componentes (elementos de sequência ou atributos de objeto) do valor em variáveis.

Exemplo:

```
1
2  # função exemplo que usa match como estrutura condicional
3
4  def process_data(data):
5          match data:
6                  case 1:
7                          print("O valor é 1")
8                  case 2:
9                          print("O valor é 2")
10                 case 3:
11                         print("O valor é 3")
```

```
                case _:
                        print("O valor não corresponde a nenhum caso")

# Exemplo de uso
process_data(2)


```

Neste exemplo, a função `process_data` recebe um argumento `data`. A declaração de correspondência é usada para comparar o valor de `data` com diferentes casos. Se `data` corresponder a algum caso específico, a instrução correspondente será executada. Se não houver correspondência, o caso padrão (`case _:`) será executado.

No exemplo acima, se chamarmos `process_data(2)`, a saída será "O valor é 2", pois o valor fornecido corresponde ao caso 2. Se chamarmos `process_data(5)`, a saída será "O valor não corresponde a nenhum caso", pois não há um caso correspondente para o valor 5.

Estruturas de Repetição As estruturas de repetição em ***Python*** são usadas para executar um bloco de código repetidamente com base em determinadas condições. Elas permitem que você execute uma sequência de comandos várias vezes até que uma condição seja atendida ou enquanto uma condição for verdadeira. Existem duas principais estruturas de repetição em ***Python***: `while` e `for`.

Loop `while`: O `while` é uma estrutura de repetição que executa um bloco de código repetidamente enquanto uma condição específica for verdadeira. A condição é verificada antes de cada iteração do loop.

Exemplo:

```

# fazendo um contador com loop while
contador = 0
while contador < 5:
        print("Contador:", contador)
        contador += 1


```

Nesse exemplo, o bloco de código dentro do *loop* `while` será repetido enquanto a variável `contador` for menor que 5. A cada iteração, o valor do contador é incrementado em 1 e impresso na tela.

Loop `for`: O `for` é uma estrutura de repetição que percorre um iterável, como uma lista, uma string ou um intervalo de números. Ele executa um bloco de código para cada elemento no iterável.

Exemplo:

```
1
2  # usando o for pra dar um olá para o pessoal da Técnica
3
4  nomes = ["Sargento Franco",
5           "Soldado Prisse",
6           "Soldado Aguayo",
7           "Soldado Erivelton",
8           "Jean",
9           "Augusto"]
10
11 for nome in nomes:
12         print("Olá,", nome)
13
```

Nesse exemplo, o bloco de código dentro do *loop* `for` será executado para cada elemento da lista `nomes`. A cada iteração, o valor do elemento é atribuído à variável `nome` e uma saudação é impressa na tela.

Além disso, a função embutida `range()` é frequentemente utilizada para gerar uma sequência de números a serem percorridos no *loop* `for`.

Exemplo:

```
1
2  # contando de 1 a 4 usando o loop for e range()
3
4  for i in range(1, 5):
5          print(i)
6
```

Nesse exemplo, o *loop* `for` percorre os números de 1 a 4 (o último valor não é incluído) e imprime cada número na tela.

As estruturas de repetição são úteis quando você precisa executar um bloco de código repetidamente até que uma condição seja atendida ou enquanto uma condição for verdadeira. Elas permitem automatizar tarefas e processar coleções de dados de forma eficiente. A escolha entre o uso de um *loop* `while` ou `for` depende da situação específica e das necessidades do seu programa.

Controle de *Loop* As estruturas de controle de *loop* são recursos em programação que permitem controlar o fluxo de execução dentro de um *loop*. Elas fornecem mecanismos para interromper a execução do loop antes de sua conclusão normal, avançar para a próxima iteração ou pular para uma parte específica do *loop*.

Existem três principais estruturas de controle de loop que podem ser usadas em ***Python***: `break`, `continue` e `pass`.

Cláusula `break`: A declaração `break` é usada para interromper a execução do *loop* imediatamente, mesmo que a condição de repetição ainda seja verdadeira. Quando o programa encontra a instrução `break`, ele sai do *loop* e continua a execução do código após o *loop*.

Exemplo:

```
1
2 for i in range(1, 10):
3         if i == 5:
4                 break
5         print(i)
6
```

Nesse exemplo, o *loop* `for` é interrompido quando a variável `i` é igual a 5. Portanto, apenas os números de 1 a 4 serão impressos na tela.

Cláusula `continue`: A declaração `continue` é usada para pular para a próxima iteração do *loop*, ignorando o restante do bloco de código para aquela iteração específica. Quando o programa encontra a instrução `continue`, ele volta para o início do *loop* e verifica a condição de repetição novamente.

Exemplo:

```
1
2 for i in range(1, 6):
3         if i == 3:
4                 continue
5         print(i)
6
```

Nesse exemplo, o número 3 será pulado e não será impresso na tela. O *loop* continuará para as próximas iterações e imprimirá os números restantes.

Cláusula `pass`: A declaração `pass` é uma instrução vazia que não faz nada. Ela é usada quando é necessário ter uma instrução válida sintaticamente, mas não se deseja executar nenhum código. Geralmente é usado como um espaço reservado temporário ou para evitar erros de sintaxe quando um bloco de código está vazio.

Exemplo:

```
1
2 for i in range(1, 5):
3         if i == 3:
4                 pass
5         else:
6                 print(i)
7
```

Nesse exemplo, a instrução `pass` é usada para indicar que não há ação específica a ser tomada quando o valor de `i` é igual a 3. O *loop* continuará normalmente e imprimirá os outros números.

As estruturas de controle de *loop* oferecem flexibilidade e controle adicional sobre a execução dos *loops* em um programa ***Python***. Elas permitem que você manipule o comportamento padrão do loop e faça decisões com base em determinadas condições. A escolha adequada entre `break`, `continue` e `pass` depende dos requisitos específicos do seu programa e da lógica desejada para controlar o *loop*.

3.1.2 Operadores da Linguagem

Operadores em ***Python*** são símbolos especiais que permitem realizar operações entre valores ou variáveis. Eles são usados para manipular e combinar dados, realizar cálculos matemáticos, comparar valores e realizar operações lógicas. Existem vários tipos de operadores em ***Python***, incluindo operadores aritméticos, operadores de atribuição, operadores de comparação, operadores lógicos e operadores de pertinência.

Operadores Aritméticos Os operadores aritméticos são usados para realizar operações matemáticas básicas. Os operadores aritméticos em ***Python*** incluem:

- **+** $\Rightarrow$ Soma dois valores.
- **-** $\Rightarrow$ Subtrai o segundo valor do primeiro.
- ***** $\Rightarrow$ Multiplica dois valores.
- **/** $\Rightarrow$ Divide o primeiro valor pelo segundo.

- **%** ⇒ Retorna o resto da divisão inteira entre dois valores.
- ****** ⇒ Realiza a exponenciação do primeiro valor pelo segundo.
- **//** ⇒ Realiza a divisão inteira do primeiro valor pelo segundo.

Exemplos de uso:

```

a = 10
b = 3

# Soma
soma = a + b
print(soma)  # Output: 13

# Subtração
subtracao = a - b
print(subtracao)  # Output: 7

# Multiplicação
multiplicacao = a * b
print(multiplicacao)  # Output: 30

# Divisão
divisao = a / b
print(divisao)  # Output: 3.3333333333333335

# Resto da divisão
resto = a % b
print(resto)  # Output: 1

# Exponenciação
exponenciacao = a ** b
print(exponenciacao)  # Output: 1000

# Divisão inteira
divisao_inteira = a // b
print(divisao_inteira)  # Output: 3

```

Operadores de Atribuição Os operadores de atribuição são usados para atribuir valores a variáveis. O operador de atribuição mais comum é o sinal de igual (=), que atribui o valor à variável. Os operadores de atribuição incluem:

- **=** ⇒ Atribui um valor à variável.

- **+=** ⇒ Incrementa o valor da variável com outro valor.
- **-=** ⇒ Decrementa o valor da variável com outro valor.
- ***=** ⇒ Multiplica o valor da variável por outro valor.
- **/=** ⇒ Divide o valor da variável por outro valor.
- **%=** ⇒ Calcula o resto da divisão da variável por outro valor.

Exemplos de uso:

```
1
2  a = 5
3
4  # Atribuição simples
5  b = a
6  print(b)  # Output: 5
7
8  # Atribuição com soma
9  a += 2  # Equivalente a: a = a + 2
10 print(a)  # Output: 7
11
12 # Atribuição com subtração
13 a -= 3  # Equivalente a: a = a - 3
14 print(a)  # Output: 4
15
16 # Atribuição com multiplicação
17 a *= 2  # Equivalente a: a = a * 2
18 print(a)  # Output: 8
19
20 # Atribuição com divisão
21 a /= 4  # Equivalente a: a = a / 4
22 print(a)  # Output: 2.0
23
24 # Atribuição com resto da divisão
25 a %= 3  # Equivalente a: a = a % 3
26 print(a)  # Output: 2.0
27
28 # Atribuição com exponenciação
29 a **= 3  # Equivalente a: a = a ** 3
30 print(a)  # Output: 8.0
31
32 # Atribuição com divisão inteira
33 a //= 2  # Equivalente a: a = a // 2
34 print(a)  # Output: 4.0
35
```

Operadores de Comparação Os operadores de comparação são usados para comparar dois valores e retornar um valor *booleano* (Verdadeiro ou Falso) com base na comparação. Os operadores de comparação em ***Python*** incluem:

- **==** $\Rightarrow$ Verifica se dois valores são iguais.
- **!=** $\Rightarrow$ Verifica se dois valores são diferentes.
- $>$ $\Rightarrow$ Verifica se o primeiro valor é maior que o segundo.
- $<$ $\Rightarrow$ Verifica se o primeiro valor é menor que o segundo.
- $>=$ $\Rightarrow$ Verifica se o primeiro valor é maior ou igual ao segundo.
- $<=$ $\Rightarrow$ Verifica se o primeiro valor é menor ou igual ao segundo.

Exemplos de Uso:

```

a = 5
b = 3

# Igual a
igual = (a == b)
print(igual)  # Output: False

# Diferente de
diferente = (a != b)
print(diferente)  # Output: True

# Maior que
maior = (a > b)
print(maior)  # Output: True

# Menor que
menor = (a < b)
print(menor)  # Output: False

# Maior ou igual a
maior_igual = (a >= b)
print(maior_igual)  # Output: True

# Menor ou igual a
menor_igual = (a <= b)
print(menor_igual)  # Output: False

```

Operadores Lógicos Os operadores lógicos são usados para combinar condições e realizar operações lógicas. Os operadores lógicos em ***Python*** incluem:

- **and** ⇒ Retorna `True` se ambos os operandos são verdadeiros.
- **or** ⇒ Retorna `True` se pelo menos um dos operandos é verdadeiro.
- **not** ⇒ Inverte o valor de verdade de um operando.

Exemplos de Uso:

```

a = True
b = False

# Operador AND
resultado_and = a and b
print(resultado_and)  # Output: False

# Operador OR
resultado_or = a or b
print(resultado_or)  # Output: True

# Operador NOT
resultado_not_a = not a
print(resultado_not_a)  # Output: False

resultado_not_b = not b
print(resultado_not_b)  # Output: True

```

Operadores de Pertencimento Os operadores de pertencimento são usados para verificar se um valor está presente em uma sequência. Os operadores de pertencimento em ***Python*** incluem:

- **in** ⇒ Verifica se um valor está presente em uma sequência.
- **not in** ⇒ Verifica se um valor não está presente em uma sequência.

Exemplos de Uso:

```

lista = [1, 2, 3, 4, 5]

# Operador in
```

```
resultado_in = 3 in lista
print(resultado_in)  # Output: True

resultado_not_in = 6 in lista
print(resultado_not_in)  # Output: False

```

Operadores de Identidade Em ***Python***, os operadores de identidade são usados para comparar a identidade de dois objetos. Eles avaliam se os objetos se referem ao mesmo local de memória. Os operadores de identidade em ***Python*** são os seguintes:

- **is** ⇒ Verifica se dois objetos são exatamente o mesmo objeto, ou seja, se possuem a mesma identidade.
- **is not** ⇒ Verifica se dois objetos não são o mesmo objeto, ou seja, se possuem identidades diferentes.

Os operadores de identidade são frequentemente utilizados para comparar variáveis ou objetos e verificar se eles apontam para o mesmo objeto na memória. Esses operadores são úteis quando queremos saber se duas variáveis estão se referindo ao mesmo objeto ou não.

Exemplos de Uso:

```

x = [1, 2, 3]
y = [1, 2, 3]
z = x

# Operador is
resultado_is = x is y
print(resultado_is)  # Output: False

resultado_is_not = x is not y
print(resultado_is_not)  # Output: True

# Operador is
resultado_is_z = x is z
print(resultado_is_z)  # Output: True

```

Lembre-se: Os operadores em ***Python*** são fundamentais para realizar diversas operações e manipulações de dados. Eles permitem que você realize cálculos, compare valores, atribua valores a variáveis e controle o fluxo do programa com base em condições. Combinando diferentes operadores, você pode criar expressões complexas e efetuar diversas tarefas em seus programas ***Python***.

3.1.3 Estruturas De Dados

Em ***Python***, as estruturas de dados são formas de armazenar e organizar dados de maneira eficiente e conveniente. Existem várias estruturas de dados embutidas na linguagem, cada uma com suas próprias características e finalidades. Abaixo, descrevo algumas das principais estruturas de dados em ***Python***:

Listas (*list*) coleção ordenada e mutável de elementos, que podem ser de diferentes tipos. Os elementos de uma lista são acessados através de índices, onde o primeiro elemento tem índice 0.

Pontos importantes:

- Uma lista é uma sequência ordenada de elementos.
- Os elementos podem ser de diferentes tipos (números, strings, objetos, etc.).
- Os elementos são acessados através de índices, começando em 0.
- As listas são mutáveis, ou seja, os elementos podem ser adicionados, removidos ou modificados.

Exemplos de Uso:

```
1
2  # lista de frutas - deixei em inglês pois é mais merchant
3
4  fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
5
6  # contagem de elementos - maçã
7  print(fruits.count('apple')) # output: 2
8
9
10 print(fruits.count('tangerine')) # output: 0
11
12
13 print(fruits.index('banana')) # output: 3
14
15 print(fruits.index('banana', 4))  # Find next banana starting at position 4
16 # output: 6
17 # reverte a lista
18
19 fruits.reverse()
20
21 print(fruits)
22 #: output: ['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
23
24 # adicionando elementos
```

```
fruits.append('grape')

print(fruits)
# output: ['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']

# ordenando
fruits.sort()

print(fruits)
# output: ['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']

# tirando um elemento da lista
print(fruits.pop()) # output: 'pear'

```

O exemplo manipula uma lista de frutas em Python. Ele conta o número de ocorrências de determinada fruta, encontra o índice da primeira ocorrência de uma fruta, inverte a ordem dos elementos na lista, adiciona novos elementos, ordena a lista e remove o último elemento. As operações demonstram funcionalidades básicas de manipulação de listas em Python.

Tuplas (*tuple*) Semelhante a uma lista, uma tupla é uma coleção ordenada de elementos. A diferença é que as tuplas são imutáveis, ou seja, seus elementos não podem ser alterados após a criação.

Pontos importantes:

- Uma tupla é semelhante a uma lista, mas é imutável, ou seja, não pode ser modificada após a criação.
- Os elementos são acessados através de índices, começando em 0.
- As tuplas são úteis quando se deseja armazenar um conjunto de valores que não devem ser alterados.

Exemplos de Uso:

```

# Criando uma tupla
tupla = (1, 2, 3, 4, 5)

# Acessando elementos da tupla
print(tupla[0])  # Saída: 1
print(tupla[2])  # Saída: 3

```

```python
# Iterando sobre uma tupla
for elemento in tupla:
        print(elemento)

# Obtendo o comprimento da tupla
comprimento = len(tupla)
print(comprimento)  # Saída: 5

# Verificando se um elemento está presente na tupla
if 3 in tupla:
        print("O elemento 3 está presente na tupla")

# Concatenando tuplas
tupla2 = (6, 7, 8)
tupla_concatenada = tupla + tupla2
print(tupla_concatenada)  # Saída: (1, 2, 3, 4, 5, 6, 7, 8)

# Desempacotando uma tupla
a, b, c = tupla
print(a)  # Saída: 1
print(b)  # Saída: 2
print(c)  # Saída: 3

# Usando uma tupla como chave de dicionário
dicionario = {(1, 2): "Valor da tupla"}
print(dicionario[(1, 2)])  # Saída: Valor da tupla

# Retornando múltiplos valores de uma função usando tupla
def retorna_valores():
        return 1, 2, 3

valores = retorna_valores()
print(valores)  # Saída: (1, 2, 3)

# essa linha vai dar erro
tupla = (1,2)

tupla[1] = 3

```

Neste exemplo, criamos uma tupla contendo os números de 1 a 5. Demonstramos como acessar elementos da tupla usando índices, iterar sobre a tupla com um *loop*, obter o comprimento da tupla, verificar a presença de um elemento na tupla e concatenar duas tuplas.

Também mostramos como desempacotar os elementos de uma tupla em variáveis individuais, usar uma tupla como chave de um dicionário, e retornar múltiplos valores de uma função usando uma tupla.

Dicionários (*dict*) coleção de pares chave-valor, onde cada chave é única e mapeada a um valor correspondente. Os dicionários são úteis para armazenar informações relacionadas e acessá-las de forma eficiente através das chaves.

Pontos importantes:

- Um dicionário é uma estrutura de dados que armazena pares de chave-valor.
- As chaves são únicas e usadas para acessar os valores correspondentes.
- Os valores podem ser de diferentes tipos.
- Os dicionários são úteis para mapear informações ou associar valores a identificadores.

Exemplos de Uso:

```
1
2  # Criando um dicionário
3  dicionario = {"nome": "João", "idade": 30, "cidade": "São Paulo"}
4
5  # Acessando elementos do dicionário
6  print(dicionario["nome"])   # Saída: João
7  print(dicionario["idade"])  # Saída: 30
8
9  # Iterando sobre um dicionário
10 for chave, valor in dicionario.items():
11 print(chave, valor)
12
13 # Verificando se uma chave está presente no dicionário
14 if "cidade" in dicionario:
15 print("A chave 'cidade' está presente no dicionário")
16
17 # Adicionando um novo par chave-valor ao dicionário
18 dicionario["profissao"] = "Engenheiro"
19
20 # Atualizando o valor de uma chave no dicionário
21 dicionario["idade"] = 31
22
23 # Removendo um par chave-valor do dicionário
24 del dicionario["cidade"]
25
26 # Obtendo o número de pares chave-valor no dicionário
27 tamanho = len(dicionario)
28 print(tamanho)  # Saída: 3
29
30 # Obtendo uma lista das chaves do dicionário
31 chaves = list(dicionario.keys())
32 print(chaves)  # Saída: ['nome', 'idade', 'profissao']
```

```

# Obtendo uma lista dos valores do dicionário
valores = list(dicionario.values())
print(valores)  # Saída: ['João', 31, 'Engenheiro']

```

Neste exemplo, criamos um dicionário contendo informações sobre uma pessoa, como nome, idade e cidade. Demonstramos como acessar elementos do dicionário usando as chaves, iterar sobre o dicionário com um *loop*, verificar a presença de uma chave no dicionário e adicionar, atualizar e remover pares chave-valor do dicionário.

Também mostramos como obter o número de pares chave-valor no dicionário, obter uma lista das chaves do dicionário e obter uma lista dos valores do dicionário.

Conjuntos (*sets*) coleção não ordenada de elementos únicos. Os conjuntos não permitem elementos duplicados e suportam operações como união, interseção e diferença.

Pontos Importantes:

- Um conjunto é uma coleção de elementos únicos e não ordenados.
- Os elementos em um conjunto não possuem índices.
- Os conjuntos são úteis para remover elementos duplicados e realizar operações de conjunto, como união, interseção e diferença.

Exemplos de Uso:

```

# Criando um conjunto vazio
my_set = set()

# Adicionando elementos ao conjunto
my_set.add(1)
my_set.add(2)
my_set.add(3)
my_set.add(3)  # Não adiciona duplicatas

print(my_set)  # output: {1, 2, 3}

# Criando um conjunto com elementos iniciais
my_set = {4, 5, 6, 6}  # Não adiciona duplicatas

print(my_set)  # output: {4, 5, 6}

# Verificando o pertencimento de um elemento
```

```
print(3 in my_set)  # output: True
print(7 in my_set)  # output: False

# Removendo um elemento do conjunto
my_set.remove(4)

print(my_set)  # output: {5, 6}

# Realizando operações de conjunto
set1 = {1, 2, 3, 4, 5}
set2 = {4, 5, 6, 7, 8}

# União de conjuntos
union_set = set1.union(set2)
print(union_set)  # output: {1, 2, 3, 4, 5, 6, 7, 8}

# Interseção de conjuntos
intersection_set = set1.intersection(set2)
print(intersection_set)  # output: {4, 5}

# Diferença entre conjuntos
difference_set = set1.difference(set2)
print(difference_set)  # output: {1, 2, 3}

# Verificando se um conjunto é subconjunto de outro
subset = {1, 2}
print(subset.issubset(set1))  # output: True

# Verificando se um conjunto é superconjunto de outro
superset = {1, 2, 3, 4, 5, 6, 7, 8, 9}
print(superset.issuperset(set1))  # output: True

```

Nesse exemplo, criamos conjuntos, adicionamos elementos, verificamos a pertinência de elementos, removemos elementos, realizamos operações de conjunto como união, interseção e diferença, e verificamos se um conjunto é subconjunto ou superconjunto de outro.

Além dessas estruturas de dados, ***Python*** também possui bibliotecas adicionais que oferecem estruturas de dados mais especializadas, como pilhas, filas, árvores, grafos, entre outros. Essas estruturas podem ser importadas e usadas quando necessário.

Ao escolher uma estrutura de dados em ***Python***, é importante considerar a eficiência, a facilidade de uso e a adequação aos requisitos do problema. Cada estrutura de dados tem suas próprias vantagens e desvantagens, e a escolha correta pode influenciar no desempenho e na eficiência do código.

3.1.4 Funções

Em ***Python***, as funções são blocos de código reutilizáveis que executam uma tarefa específica. Elas são usadas para agrupar um conjunto de instruções relacionadas e podem receber argumentos (valores de entrada) e retornar resultados (valores de saída).

As funções em ***Python*** possuem as seguintes características:

- ***Definição:*** Uma função é definida usando a palavra-chave `def`, seguida pelo nome da função e parênteses contendo os argumentos, se houver.
- ***Parâmetros:*** Os parâmetros são variáveis que recebem os valores passados para a função quando ela é chamada. Eles são opcionais e podem ser de qualquer tipo de dado válido em Python.
- ***Corpo da função:*** O corpo da função é um bloco de código indentado que contém as instruções a serem executadas quando a função é chamada. Pode conter qualquer número de instruções ou até mesmo outras chamadas de função.
- ***Retorno de valores:*** Uma função pode retornar um valor usando a palavra-chave ‘‘`return`’’. Isso permite que a função forneça um resultado para o código que a chamou.

Para executar o código contido em uma função, você precisa chamá-la pelo nome, seguido de parênteses contendo os argumentos, se houver. A chamada da função faz com que o código dentro dela seja executado.

As funções em ***Python*** são uma maneira eficiente de organizar e reutilizar o código, pois permitem que você divida um programa em tarefas menores e mais gerenciáveis. Além disso, elas ajudam a melhorar a legibilidade do código e facilitam a manutenção e depuração. ***Python*** também fornece várias funções embutidas, como `print()`, `len()`, `range()`, entre outras, que podem ser usadas diretamente sem a necessidade de definição.

Em ***Python***, assim como em outras linguagens, utilizar funções em programação traz várias vantagens, que incluem:

- **Modularidade:** As funções permitem dividir um programa em blocos de código independentes e reutilizáveis. Isso facilita a compreensão e organização do código, além de promover a reutilização de código em diferentes partes do programa.
- **Reutilização de código:** Ao definir uma função, você pode chamá-la quantas vezes for necessário em diferentes partes do programa. Isso evita a duplicação de código e torna as atualizações e correções mais fáceis, já que você só precisa fazer as alterações em um único lugar.

- **Legibilidade e manutenção:** Utilizar funções ajuda a melhorar a legibilidade do código, pois as funções podem ter nomes descritivos que indicam sua funcionalidade. Além disso, ao dividir o código em funções menores e mais focadas, é mais fácil entender, testar e corrigir problemas específicos.

- **Abstração:** Funções permitem abstrair detalhes de implementação complexos em um nível mais alto de abstração. Isso significa que você pode usar uma função sem precisar saber todos os detalhes internos de como ela funciona, tornando o código mais fácil de entender e usar.

- **Testabilidade:** Funções isoladas podem ser testadas de forma independente, o que facilita a identificação de erros e o desenvolvimento de testes automatizados. Isso também contribui para a qualidade e confiabilidade do código.

- **Encapsulamento:** As funções permitem encapsular um conjunto de instruções em um bloco único, fornecendo um contexto claro para a execução das ações contidas. Isso ajuda a evitar interferências indesejadas entre partes diferentes do código.

Além do mais, o uso de funções é largamente empregado quando utilizamos o paradigma de *programação estruturada*, conforme visto na seção 2.1.

Exemplo:

```
# Definindo uma função para calcular a área de um círculo
def calcular_area_circulo(raio):
        area = 3.14159 * raio ** 2
        return area

# Chamando a função e armazenando o resultado em uma variável
resultado = calcular_area_circulo(5)

# Imprimindo o resultado
print("A área do círculo é:", resultado)

```

Nesse exemplo, temos uma função chamada `calcular_area_circulo` que recebe o raio de um círculo como parâmetro. Dentro da função, calculamos a área do círculo utilizando a fórmula matemática $\pi \times r^2$ e armazenamos o resultado na variável `area`. Em seguida, utilizamos a instrução `return` para retornar o valor da área.

Na parte principal do código, chamamos a função `calcular_area_circulo` passando o valor 5 como argumento e armazenamos o resultado retornado pela função na variável resultado. Por fim, imprimimos o resultado utilizando a função `print`.

O resultado final será a área do círculo com raio 5.

Funções aninhadas Funções aninhadas são funções definidas dentro de outras funções. Elas permitem que você crie funções que são localmente acessíveis e relevantes apenas dentro do escopo da função externa. Essas funções internas podem acessar variáveis e parâmetros da função externa, tornando-as úteis para encapsular lógica específica e reutilizá-la apenas onde for necessário.

As funções aninhadas podem ser definidas dentro de qualquer função em ***Python***, e elas têm acesso a todas as variáveis locais e argumentos da função externa. Isso significa que as funções internas podem usar e manipular os valores dessas variáveis e até mesmo retornar valores.

Uma vantagem das funções aninhadas é que elas podem ajudar a evitar a poluição do espaço global de nomes, já que estão limitadas ao escopo da função externa. Além disso, as funções aninhadas também podem ser usadas para fornecer um encapsulamento lógico mais limpo e organizado, tornando o código mais legível e fácil de manter.

Aqui está um exemplo simples de uma função aninhada em ***Python***:

```
1
2  # definição da função principal - no caso, a função externa
3  def funcao_externa():
4          # definição da função aninhada - só vale dentro do escopo da função
5          def funcao_interna():
6                  print("Esta é uma função interna.")
7
8          print("Esta é uma função externa.")
9          funcao_interna()
10
11 funcao_externa()
12
```

Neste exemplo, temos uma função externa chamada `funcao_externa` que define uma função interna chamada `funcao_interna`. Quando chamamos a função externa `funcao_externa`, ela imprime uma mensagem e, em seguida, chama a função interna `funcao_interna`, que também imprime uma mensagem. Dessa forma, as funções internas estão aninhadas dentro da função externa e podem ser acessadas apenas dentro do escopo dessa função externa.

Um exemplo mais prático:

```
1
2  def funcao_externa(x):
3          def funcao_interna(y):
4                  return y * 2
5
```

```
        resultado = funcao_interna(x)
        return resultado

valor = 5
resultado_final = funcao_externa(valor)

print("O resultado final é:", resultado_final)

```

Neste exemplo, temos uma função chamada `funcao_externa` que recebe um parâmetro `x`. Dentro dessa função, definimos uma função interna chamada `funcao_interna`, que recebe um parâmetro y e retorna o dobro desse valor.

Em seguida, chamamos a função `funcao_interna` dentro da função `funcao_externa`, passando o parâmetro `x` como argumento, e armazenamos o resultado na variável resultado.

Por fim, fora das funções, atribuímos um valor de 5 à variável valor e chamamos a função *funcao_externa* passando esse valor como argumento. O resultado retornado pela função é armazenado na variável *resultado_final*.

O resultado final será o valor 10, que é o resultado de chamar a função interna `funcao_interna` com o argumento 5 dentro da função externa `funcao_externa`.

Funções anônimas Funções anônimas, também conhecidas como funções `lambda`, são funções sem nome que podem ser definidas de forma concisa em uma única linha de código. Elas são usadas quando você precisa de uma função simples e de curta duração, sem a necessidade de definir uma função separada com um nome específico.

As funções anônimas são definidas usando a palavra-chave *lambda* seguida dos parâmetros da função, dois pontos (:) e a expressão que será retornada pela função. A sintaxe básica de uma função *lambda* é a seguinte:

```
lambda argumentos: expressão
```

A principal diferença entre uma função `lambda` e uma função normal é que as funções `lambda` não precisam de um bloco de código separado com instruções ou declarações adicionais. Elas são expressões compactas e retornam automaticamente o resultado da expressão.

As funções anônimas são frequentemente usadas como argumentos de outras funções, como `map()`, `filter()` e `reduce()`, ou em situações em que você precisa de uma função rápida para uma operação simples.

Aqui está um exemplo de uma função lambda que calcula o quadrado de um número:

```
quadrado = lambda x: x**2
```

Neste exemplo, a função `lambda` recebe um argumento `x` e retorna o valor de `x` elevado ao quadrado. Podemos chamar essa função `lambda` da seguinte forma:

```
resultado = quadrado(5)
print(resultado)  # Output: 25
```

A função `lambda` pode ser usada em qualquer lugar onde você normalmente usaria uma função definida explicitamente. Elas são especialmente úteis quando você precisa de uma função simples e de curta duração, evitando a necessidade de definir uma função separada com um nome específico.

No entanto, é importante notar que as funções anônimas têm suas limitações. Como são expressões curtas e não têm um nome associado, podem ser menos legíveis em comparação com funções nomeadas tradicionais. Além disso, as funções `lambda` não podem realizar tarefas complexas que exigem múltiplas instruções ou estruturas de controle.

Decoradores (*decorators*) `decorators` são recursos avançados da linguagem ***Python*** que permitem modificar ou estender o comportamento de uma função ou classe sem precisar alterar o código fonte original. Eles fornecem uma maneira elegante de envolver ou decorar uma função, adicionando funcionalidades extras antes, depois ou em torno da execução da função.

Em termos simples, um `decorator` é uma função que recebe outra função como argumento e retorna uma nova função modificada. Ele é usado para envolver a função original com lógica adicional, sem precisar modificar diretamente a implementação da função original.

A sintaxe básica para definir e usar um `decorator` é a seguinte:

```
def decorator(funcao):
        def funcao_decorada(*args, **kwargs):
                # Código adicional antes da execução da função
                resultado = funcao(*args, **kwargs)
                # Código adicional depois da execução da função
                return resultado
        return funcao_decorada
```

```
@decorator
def funcao_original():
        # Implementação da função original
        pass
```

Nesse exemplo, o `decorator` é definido como uma função chamada `decorator`. Ele recebe a função original como argumento e define uma nova função chamada `funcao_decorada`. Dentro da função `funcao_decorada`, você pode adicionar código adicional antes e depois da chamada da função original. Em seguida, o `decorator` retorna a função *funcao_decorada*.

Ao usar o `decorator`, você simplesmente coloca o símbolo `@` seguido pelo nome do `decorator` antes da definição da função original. Isso informa ao ***Python*** que a função original deve ser envolvida pelo `decorator` durante a execução.

Os `decorators` são amplamente usados para implementar recursos como `logging`, controle de acesso, medição de tempo, tratamento de erros e muito mais. Eles ajudam a manter o código limpo, modular e reutilizável, separando as preocupações adicionais do código principal da função.

É importante destacar que os `decorators` podem ser empilhados, ou seja, você pode aplicar vários `decorators` em uma mesma função, resultando em um encadeamento de funcionalidades extras.

Em resumo, os `decorators` são recursos avançados do ***Python*** que permitem modificar o comportamento de uma função ou classe sem alterar o código fonte original. Eles envolvem a função original com lógica adicional, adicionando funcionalidades extras antes, depois ou em torno da execução da função. Os `decorators` são amplamente usados para implementar recursos adicionais de forma modular e reutilizável.

Um exemplo prático (não se preocupe com o django, veremos detalhadamente do que se trata este *framework* em outro módulo):

```
1  from django.http import HttpResponse
2
3  # Definição do decorator
4  def verifica_autenticacao(view_func):
5          def wrapper(request, *args, **kwargs):
6                  # Verifica se o usuário está autenticado
7                  if request.user.is_authenticated:
8                          # Se estiver autenticado, chama a view original
9                          return view_func(request, *args, **kwargs)
10                 else:
11                         # Caso contrário, redireciona para a página de login
12                         return HttpResponse("Acesso negado. Faça login para continuar.")
```

```
        return wrapper

# Exemplo de uso do decorator em uma view
@verifica_autenticacao
def minha_view(request):
        return HttpResponse("Conteúdo da view protegida")
```

Neste exemplo, temos um `decorator` chamado `verifica_autenticacao` que verifica se o usuário está autenticado antes de permitir o acesso a uma determinada `view`. O `decorator` envolve a função `wrapper`, que realiza a verificação de autenticação e redireciona o usuário para a página de *login* se não estiver autenticado. Caso contrário, a `view` original é chamada.

No exemplo de uso do `decorator`, a função `minha_view` é decorada com `@verifica_autenticacao`. Isso significa que, sempre que essa `view` for acessada, o `decorator` será aplicado automaticamente, verificando a autenticação do usuário antes de executar o código da `view`.

Dessa forma, o `decorator` permite adicionar funcionalidades extras a uma `view`, como a verificação de autenticação, sem modificar diretamente o código da `view` em si. Isso ajuda a manter o código organizado, reutilizável e facilita a implementação de recursos comuns em várias `views`.

3.1.5 Tratamento de Exceções - bloco `try ... except`

O tratamento de exceções no Python é feito utilizando os blocos `try` (tentar) e `except` (capturar). O objetivo do tratamento de exceções é lidar com possíveis erros ou situações excepcionais que possam ocorrer durante a execução do código.

O formato básico do tratamento de exceções é o seguinte:

```
try:
        # Código que pode gerar uma exceção
        # ...
except TipoDeExcecao:
        # Código a ser executado em caso de exceção do tipo TipoDeExcecao
        # ...
```

Vamos analisar os principais componentes do tratamento de exceções:

- O bloco `try` contém o código onde pode ocorrer uma exceção. Esse bloco é executado normalmente até que uma exceção seja lançada.

- O bloco `except` é utilizado para capturar e tratar uma exceção específica. É dentro desse bloco que você pode escrever o código que será executado caso a exceção ocorra. O tipo de exceção que você deseja capturar é especificado após a palavra-chave `except`.
- É possível ter múltiplos blocos `except` para lidar com diferentes tipos de exceções. Você pode capturar exceções específicas e tratá-las de forma adequada em cada bloco `except` correspondente.
- Além do bloco `except`, você também pode usar os blocos `else` e `finally`:
  - O bloco `else` é opcional e é executado somente se nenhum erro ocorrer dentro do bloco `try`. É útil para executar código adicional quando não há exceções.
  - O bloco `finally` também é opcional e é executado sempre, independentemente se ocorreu uma exceção ou não. É usado para realizar ações de limpeza, como fechar arquivos ou liberar recursos, independentemente de qualquer exceção que tenha ocorrido.

Para capturar exceções genéricas, você pode usar a palavra-chave `except` sem especificar o tipo de exceção. No entanto, é recomendável capturar exceções específicas sempre que possível, para que você possa tratar cada tipo de exceção adequadamente.

Aqui está um exemplo de tratamento de exceção no ***Python***:

```
try:
        # Código que pode gerar uma exceção
        resultado = dividir(10, 0)
except ZeroDivisionError:
        # Código a ser executado caso ocorra uma divisão por zero
        print("Não é possível divisão por zero")
```

Neste exemplo, o código dentro do bloco `try` chama uma função `dividir(10, 0)` que realiza uma divisão por zero. Como isso gera uma exceção do tipo `ZeroDivisionError`, a exceção é capturada pelo bloco `except ZeroDivisionError` e o código dentro desse bloco é executado, exibindo a mensagem *"Não é possível divisão por zero"*.

É importante tratar exceções de forma apropriada em seu código para garantir que erros inesperados sejam tratados de maneira adequada e que seu programa continue funcionando corretamente, mesmo em situações excepcionais.

4 Mão na Massa

Desafio: Tendo como base o conhecimento adquirido até agora, vamos implementar uma calculadora com as 4 operações básicas:

- Implemente uma função para cada uma das 4 operações da calculadora (soma, subtração, multiplicação e divisão)
- implemente o *loop* principal utilizando a função `input` nativa do ***Python*** e um *loop* `while` com a condição sempre verdadeira. É um *loop* infinito, porém, não se preocupe com isto agora.

Não sabe como fazer? temos uma sugestão. Continue lendo.

A primeira coisa a se fazer é ter uma ideia de como estruturar o programa de forma clara e precisa (seguindo os passos de criação de algoritmos descrito na seção 2.1):

- ***Compreenda o problema:*** O problema se trata de fazer uma calculadora simples de 4 operações em ***Python***.
- ***Divida o problema em etapas menores:*** o próprio enunciado sugere que você implemente 4 funções para executar as operações da calculadora, mais uma função contendo o *loop* principal. Vamos ser bons meninos, não inventar moda e atribuir a cada uma destas funcionalidades os nomes de `soma`, `subtrai`, `multiplica`, `divide` e `main`.
- ***Identifique as entradas e saídas:*** as funções que executam as operações aritméticas recebem dois operandos (entradas) e devolvem um resultado (saída). A função principal não recebe nem devolve nada (sem entrada e saída), porém lê do teclado 2 operandos e uma operação, imprimindo o resultado da operação na tela.
  - função *soma*: deve somar os operandos e devolver o resultado da soma.
  - função *subtrai*: deve subtrair os operandos e devolver o resultado da subtração.
  - função `multiplica`: deve multiplicar os operandos e devolver o resultado da multiplicação.
  - função `divide`: deve dividir os operandos e devolver o resultado da divisão.
  - função `main`: deve requisitar ao usuário dois operandos numéricos, em ordem (`a` e `b`), requisitar ao usuário o tipo de operação (soma, subtração, multiplicação ou divisão - por economia de tempo utilizei notação polonesa [operando, operando, operação]), deve analisar qual o tipo de operação e executar a função correspondente.

Nota: podem ocorrer exceções ou erros na entrada do usuário. Por padrão, o programa ***NUNCA*** deve confiar cegamente na entrada do usuário. Faremos uma estratégia para evitar estes erros e ter somente operandos e operação essencialmente numéricos.

A princípio, o pseudocódigo define as operações da calculadora como funções. A seguir, uma explicação exaustiva, porém necessária, para os iniciantes na *arte de transformar café em código*, ou seja, programar.

Função `soma` A primeira função definida é a função `soma`, que recebe dois parâmetros, a e b, e retorna a soma desses dois valores:

```
Função soma(a, b)
        s <- a + b
        Retornar s
```

- A função `soma` é definida com os parâmetros a e b.
- A variável s é inicializada com o valor da soma de a e b, ou seja, s recebe o resultado da operação de adição de a e b.
- O valor de s é retornado como resultado da função.
- O fluxo de execução retorna para o ponto em que a função foi chamada, e o resultado da soma pode ser usado conforme necessário.

Função `subtrai` A função `subtrai` recebe dois parâmetros, a e b, e retorna a subtração $(a - b)$ desses dois valores:

```
Função subtrai(a, b)
        s <- a - b
        Retornar s
```

- A função `subtrai` é definida com os parâmetros a e b.
- A variável s é inicializada com o valor da subtração de a e b, ou seja, s recebe o resultado da operação de subtração de a por b.

- O valor de s é retornado como resultado da função.
- O fluxo de execução retorna para o ponto em que a função foi chamada, e o resultado da subtração pode ser usado conforme necessário.

Função `multiplica` A função `multiplica` recebe dois parâmetros, a e b, e retorna a multiplicação ($a \times b$) desses dois valores:

```
Função multiplica(a, b)
        s <- a * b
        Retornar s
```

- A função `multiplica` é definida com dois parâmetros, a e b.
- A variável s recebe o resultado da multiplicação de a e b, ou seja, s é atribuído o valor da expressão $a \times b$.
- O valor de s é retornado como resultado da função.

Função `divide` A função `divide` recebe dois parâmetros, a e b, e retorna a divisão (a/b) desses dois valores:

```
Função divide(a, b)
        Tentar
                s <- a / b
                Retornar s
        Capturar erro de divisão por zero
                Imprimir "Não é possível divisão por zero"
```

- A função **divide** é definida com os parâmetros a e b.
- Tenta-se executar o bloco de código dentro do comando **Tentar**.
- Dentro do bloco, a variável s é inicializada com o resultado da divisão de a por b.
- O valor de s é retornado como resultado da função.

- Se ocorrer um erro de divisão por zero, o fluxo de execução será interrompido e passará para o bloco **Capturar erro de divisão por zero**.
- Dentro desse bloco, a mensagem "Não é possível divisão por zero"é impressa.

Função `main` a função `main` é um tanto extensa, o que torna necessário sua explicação passo a passo:

- A função `main()` é definida, que será responsável pela execução principal do programa.

```
Função main()
```

- A variável `q` é inicializada com o valor 'a'. Essa variável será usada como uma condição de controle para o *loop* principal.

```
# Variável de controle do loop - quando o usuário quiser sair, basta digitar 'q'
# após o loop exibir o resultado da operação
q <- 'a' # atribuição inicial com valor diferente de 'q'
```

- Entra em um *loop* enquanto `q` for diferente de 'q', o que significa que o usuário ainda não digitou 'q' para sair do programa.

```
Enquanto q diferente de 'q' faça
# Entrada de operandos e operação - Notação Polonesa
```

- Solicita ao usuário que digite o primeiro operando e tenta ler o valor como um número decimal (`float`).
  - Se o valor digitado for válido, ele é armazenado na variável `opa`.
  - Caso contrário, captura o erro de valor inválido e imprime a mensagem "Somente números são permitidos". Em seguida, continua para a próxima iteração do *loop*.

```
Imprimir "Digite o primeiro operando: "
Tentar
        Ler opa como float
Capturar erro de valor inválido
        Imprimir "Somente números são permitidos"
        Continuar para próxima iteração
```

- Solicita ao usuário que digite o segundo operando e tenta ler o valor como um número decimal (`float`).
  - Se o valor digitado for válido, ele é armazenado na variável `opb`.
  - Caso contrário, captura o erro de valor inválido e imprime a mensagem "Somente números são permitidos". Em seguida, continua para a próxima iteração do *loop*.

```
Imprimir "Digite o segundo operando: "
Tentar
        Ler opb como float
Capturar erro de valor inválido
        Imprimir "Somente números são permitidos"
        Continuar para próxima iteração
```

- Solicita ao usuário que informe o tipo de operação desejada: 0 para soma, 1 para subtração, 2 para multiplicação ou 3 para divisão. Tenta ler o valor como um número inteiro.
  - Se o valor digitado for válido, ele é armazenado na variável `op`.
  - Caso contrário, captura o erro de valor inválido e imprime a mensagem "Somente números inteiros são permitidos". Em seguida, continua para a próxima iteração do *loop*.

```
Imprimir "Informe o tipo de operação (0 - Soma; 1 - Subtração;
        2 - Multiplicação; 3 - Divisão): "
Tentar
        Ler op como inteiro
Capturar erro de valor inválido
        Imprimir "Somente números inteiros são permitidos"
        Continuar para próxima iteração
```

- Analisa o valor de `op` para determinar a operação a ser executada.
  - Se `op` for igual a 0, chama a função `soma(opa, opb)` e imprime o resultado.
  - Se `op` for igual a 1, chama a função `subtrai(opa, opb)` e imprime o resultado.
  - Se `op` for igual a 2, chama a função `multiplica(opa, opb)` e imprime o resultado.
  - Se `op` for igual a 3, chama a função `divide(opa, opb)` e imprime o resultado.
  - Se `op` não corresponder a nenhum dos valores anteriores, imprime a mensagem "Operação sem suporte".

```
# Análise e execução da operação escolhida
Se op igual a 0 então
        Imprimir soma(opa, opb)
Senão, se op igual a 1 então
        Imprimir subtrai(opa, opb)
Senão, se op igual a 2 então
        Imprimir multiplica(opa, opb)
Senão, se op igual a 3 então
        Imprimir divide(opa, opb)
Senão
        Imprimir "Operação sem suporte"
```

- Solicita ao usuário que pressione qualquer tecla para continuar ou 'q' para sair.
  - Lê o valor digitado e o armazena em `q`. O `loop` continuará se `q` for diferente de 'q', caso contrário, o programa irá sair do `loop` e encerrar.

```
Imprimir "Pressione qualquer tecla para continuar, q para sair"
Ler q
```

- Após sair do `loop`, o programa termina sua execução.

Chamada à função `main` por fim, o pseudocódigo chama a função `main` para executar o loop principal do programa:

```
# Chamada da função principal para iniciar a execução da calculadora
main()

Fim do programa
```

O algoritmo geral do programa *calculadora* em pseudocódigo é apresentado a seguir:

```
Algoritmo Calculadora Simples

# Definição das funções de operação
Função soma(a, b)
        s <- a + b
        Retornar s

Função subtrai(a, b)
        s <- a - b
        Retornar s

Função multiplica(a, b)
        s <- a * b
        Retornar s

Função divide(a, b)
        Tentar
                s <- a / b
                Retornar s
        Capturar erro de divisão por zero
                Imprimir "Não é possível divisão por zero"

Função main()
        # Variável de controle do loop - quando o usuário quiser sair, basta digitar 'q'
        # após o loop exibir o resultado da operação
        q <- 'a' # atribuição inicial com valor diferente de q

        Enquanto q diferente de 'q' faça
                # Entrada de operandos e operação - Notação Polonesa

                Imprimir "Digite o primeiro operando: "
                Tentar
                        Ler opa como float
                Capturar erro de valor inválido
                        Imprimir "Somente números são permitidos"
                        Continuar para próxima iteração

                Imprimir "Digite o segundo operando: "
                Tentar
                        Ler opb como float
                Capturar erro de valor inválido
```

```
                        Imprimir "Somente números são permitidos"
                        Continuar para próxima iteração

                Imprimir "Informe o tipo de operação (0 - Soma; 1 - Subtração;
                2 - Multiplicação; 3 - Divisão): "
                Tentar
                        Ler op como inteiro
                Capturar erro de valor inválido
                        Imprimir "Somente números inteiros são permitidos"
                        Continuar para próxima iteração

                # Análise e execução da operação escolhida
                Se op igual a 0 então
                        Imprimir soma(opa, opb)
                Senão, se op igual a 1 então
                        Imprimir subtrai(opa, opb)
                Senão, se op igual a 2 então
                        Imprimir multiplica(opa, opb)
                Senão, se op igual a 3 então
                        Imprimir divide(opa, opb)
                Senão
                        Imprimir "Operação sem suporte"

                Imprimir "Pressione qualquer tecla para continuar, q para sair"
                Ler q

# Chamada da função principal para iniciar a execução da calculadora
main()

Fim do programa

```

Enfim, começaremos a codificar o código em *Python*, utilizando as convenções estudadas nesta apostila. Pe tradicionalmente aceito que acima de qualquer código tenha um grande comentário apresentando o software, o que ele faz, a autoria e o *copyright*:

```
"""
Calculadora simples para exemplificar os conceitos vistos no módulo 01
@author Augusto Mathias Adams <augusto.mathias@sesp.pr.gov.br>

MIT License

Copyright (c) 2023 Augusto Mathias Adams

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
```

```
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN
"""
```

Isto vai do gosto pessoal ou de normas da empresa na qual se trabalha. Para fins educativos, todo o código exposto nesta apostila está sob a licença *MIT*.

As funções são implementadas logo em seguida. Ainda não falamos de *pacotes*, mas seria logo após a apresentação do software que importaríamos os pacotes necessários para o funcionamento de nossas funções. Mas como não precisamos de nada de um pacotes específico, vamos logo para a definição das funções. Pacotes é assunto do módulo 2.

As funções ficam assim definidas em *Python*:

```
# definição de operações da calculadora

def soma(a, b) -> float:
        """
        Efetua a Soma dos Números a e b
        Args:
                a (real): número a
                b (real): número b
        Returns:
                real: a soma de a e b
"""
        s = a + b
        return s

def subtrai(a, b) -> float:
        """
        Efetua a Subtração dos Números a e b
        Args:
                a (real): número a
                b (real): número b
        Returns:
                real: a subtração de a e b
```

```
        """
        s = a - b
        return s

def multiplica(a, b) -> float:
        """
        Efetua a multiplicação dos Números a e b
        Args:
                a (real): número a
                b (real): número b
        Returns:
                real: a multiplicação de a e b
        """
        s = a * b
        return s

def divide(a, b) -> float:
        """
        Efetua a divisão dos Números a e b
        Args:
                a (real): número a
                b (real): número b
        Returns:
                real: a multiplicação de a e b
        """
        try:
                s = a / b
                return s
        except ZeroDivisionError as e:
                print("Não é possível divisão por zero")
```

Note a tradução do pseudocódigo para o *Python*: é praticamente uma tradução do português para o inglês, além da inserção de elementos de sintaxe e uso de uma função nativa do *Python* (`print`). Não é à toa que ***Python*** é considerada uma linguagem compreensivel e fácil. Um pouco de prática e dedicação elevam você à categoria de programador mirim.

O *loop* principal, obviamente, é bastante semelhante ao pseudocódigo:

```
def main() -> None:
        """
        Executa o loop principal da calculadora de açougueiro
        Args:
                None
        Returns:
                None
        """
```

```
        # variável  de controle do loop - quando o usuário quiser sair
        # basta digitar q após o loop exibir o resultado da operação
        q = ord('a')

        while q != 'q':
        # entrada de operandos e a operação - Notação Polonesa

        print("Digite o primeiro operando: ")
        try:
                opa = float(input())
        except ValueError as e:
                print("Somente números são permitidos")
                continue

        print("Digite o segundo operando: ")
        try:
                opb = float(input())
        except ValueError as e:
                print("Somente números são permitidos")
                continue

        print("Informe o tipo de operação(0 - Soma; 1 - Subtração. " +\
        "2 - Multiplicação; 3 - Divisão): ")
        try:
                op = int(input())
        except ValueError as e:
                print("Somente números inmteiros são permitidos")
                continue

        # análise e execução de operação

        if op == 0:
                print(soma(opa, opb))
        elif op == 1:
                print(subtrai(opa, opb))
        elif op == 2:
                print(multiplica(opa, opb))
        elif op == 3:
                print(divide(opa, opb))
        else:
                print("Operação sem Suporte")

        print("Pressione qualquer tecla para continuar, q para sair")

        q = input()

# usando a função main para controlar a execução da calculadora

if __name__ == "__main__":
```

```
    main()
```

Fácil não? Relaxa, é brincadeira. Todo mundo que é programador e analista de sistemas já teve dificuldade (e ainda muitos tem) de elaborar algoritmos até mesmo simples de forma procedural como este da calculadora por não entender os princípios básicos da programação estruturada, descrito na seção 2.1 e da programação procedural, descrito na seção 2.1. Contudo, vale lembrar que:

- Facilite sua vida entendendo o problema. Parece tempo perdido, mas não é. Elabore e pense em detalhes que parecem escondidos: o processador que irá executar pode ser o mais rápido do mundo, porém, é uma máquina desprovida de inteligência própria e é por este motivo que descrever algoritmos deve ser tão detalhado quanto humanamente possível. Não se furte de pegar uma folha de papel e anotar os requisitos necessários para o bom funcionamento do que quer que esteja fazendo em software.

- Quebre o problema em pequenas partes (análise): novamente, parece perda de tempo. Somente parece. Contudo, vale lembrar que muitos problemas são insolúveis em sua forma integral. Lembre-se de que até uma grande guerra é composta por pequenas batalhas.

- Resolva um pequeno problema por vez. Teste. Reteste. Corrija os erros. Teste novamente, até que esteja funcionando para todos os casos de interesse.

- Por fim, faça a síntese de tudo que você resolveu até agora em um programa principal. Não ache que você não precisa de nada disto que foi feito até agora e comece escrevendo código *"macarrônico"* em um *script* somente. Dependendo do código, nem compensa fazer grande coisa, tal como o código simples de *ping* a seguir:

```
import os
import csv

def ping(file):
        content = ""
        with open(file) as file:
                reader = csv.reader(file)

        next(reader)  # Advance past the header

        for row in reader:
                print(row[0])
                response = os.system("ping -c 1 " + row[0])
```

```
                # and then check the response...
                if response == 0:
                        content += ",".join(row) + ",UP\n"
                        print(row[0] + " UP")
                else:

                        content += ",".join(row) + ",DOWN\n"
                        print(row[0] + " DOWN")

        with open("data/pingados.csv", "w") as f:
                f.write(content)

ping("data/ipping.csv")
```

Este é um exemplo de um código feito às pressas para resolver rapidamente um problema. Embora funcional e tenha elementos de programação estruturada, é um tanto complicado adicionar funcionalidades pois, embora tenha uma função (`ping`), praticamente todo o *loop* principal do *script* está contido nela e se precisarmos de algo complexo, teremos problemas para colocar tudo neste *loop* da mesma forma que colocamos tudo que precisávamos para resolver um pequeno, porém importante, problema. Defina quantas funções forem necessárias, desenhe um fluxograma (ou melhor vários) para todo e qualquer parte do algoritmo, refaça algoritmos se for necessário, inclusive os que estão descritos aqui.

- Vale lembrar também que:
  - Nenhum problema admite solução única.
  - O debate sobre soluções é sempre válido.
  - Programar é um mundo vasto, multidisciplinar, interessante, apaixonante e (dependendo do nível) bastante promissor.
  - Programar é uma arte que exige esforço, pensamento e muita fé: em várias situações, não se sabe qual o melhor caminho da solução. E para isto não há remédio, somente a experiência lhe indica o melhor a fazer. Mas confie em si, pois é você quem vai resolver o pepino.
  - O estudo e o esforço são os investimentos que rendem os melhores juros. ***Lembre-se disto!!!***