**Apresentação - Lógica de Programação**

A disciplina de Lógica de Programação representa a entrada do aluno na área de desenvolvimento de sistemas e correlatos. Seu conteúdo abrange os conceitos e ferramentas básicas de programação, como variáveis e estruturas de controle, que capacitam o aluno a solucionar problemas computacionais e construir programas utilizando essas ferramentas básicas existentes em qualquer linguagem. É uma disciplina de vital importância, pois seu conteúdo serve de alicerce para uma série de outras disciplinas que abordam aspectos mais avançados de programação e que serão vistas na sequência do curso.

Acompanhe a apresentação da disciplina, elaborada pelo Professor Gilberto Aves Pereira:

[Apresentação Lógica de Programação](https://www.youtubeeducation.com/watch?v=L5l2bjfRi4g).

**Objetivos gerais**

Desenvolvimento do Raciocínio Lógico;

Aprender a descrever de forma gráfica uma solução lógica;

Entender os Conceitos Envolvidos em Lógica de Programação;

Ser capacitado a utilizar as ferramentas básicas de lógica de programação abaixo:

* Variáveis e Comandos de Entrada e Saída;
* Estruturas de Controle de Fluxo (Sequencial, Seleção e Repetição);
* Funções;
* Estruturas de Armazenamento - Vetores.

**Parte 1 -** [**Introdução - Algoritmos e Fluxogramas**](https://sites.google.com/faculdadeimpacta.com.br/lo-p1?pli=1&authuser=2) **-** [**Video**](https://www.youtubeeducation.com/watch?v=KWTQJxJ_ZPU)

**Resumo**

Nesse texto discutimos os conceitos básicos envolvidos quando falamos inicialmente sobre lógica de programação: Algoritmos, o funcionamento de um software, Linguagem de Programação e por fim falamos um pouco sobre a ferramenta que usaremos em nosso curso - o Fluxograma.

**Algoritmo**

O termo algoritmo pode ser visto desde o século IX. Foi nesta época que o cientista, astrônomo e matemático persa Abū ‘Abd Allāh Muhammad ibn Mūsā al-Khwārizmī usou pela primeira vez o termo para indicar regras de operações aritméticas utilizando algarismos indo-arábicos (DIERBACH 2012).

No século XII, Adelardo de Bath traduziu o termo para o latim Algorithmi. De lá para cá, o termo evoluiu bastante, incluindo todos os procedimentos definidos para resolver problemas ou realizar tarefas.

DIERBACH (2012) A formalização da noção de algoritmo ocorreu em 1936 com os trabalhos de Alan Turing e Alonzo Church, que desenvolveram independentemente os modelos de Máquinas de Turing e Cálculo Lambda.

Do ponto de vista computacional, um algoritmo pode ser visto como um conjunto de regras e procedimentos lógicos perfeitamente definidos que levam à solução de um problema em um número finito de passos.

DIERBACH (2012) Donald Knuth, um dos pesquisadores mais respeitados em algoritmos, indica uma lista

de cinco propriedades que são requisitos para algoritmos:

**Finitude**: um algoritmo deve sempre terminar após um número finito de etapas (ou passos).

**Definição**: cada passo de um algoritmo deve ser definido com precisão. As ações a serem executadas deverão ser especificadas rigorosamente e sem ambiguidades.

**Entrada**: valores que são dados ao algoritmo antes que ele inicie.

**Saída**: os valores resultantes das ações do algoritmo a partir de uma determinada entrada.

**Eficácia**: todas as operações a serem realizadas pelo algoritmo devem ser suficientemente básicas para poderem, em princípio, ser feitas com precisão e em um período de tempo finito por um homem usando papel e lápis.



A formalidade pode ser conseguida com o uso de lógica. Assim, vamos exigir que um algoritmo seja uma sequência lógica de passos com começo, meio e fim.

Comumente, esta lógica é conhecida como lógica de programação e isto ocupará grande parte de nossa disciplina.

**Funcionamento de um Software**

De uma maneira simplificada, um software para funcionar precisa de um ambiente chamado Sistema Digital. O sistema digital é composto minimamente por uma unidade de processamento de instruções (UCP), memória (volátil - RAM), e de dispositivos de entrada e saída. As informações dentro do sistema digital estão em formato binário (zeros e uns).

O software é um conjunto de instruções para o processador que fica armazenado na memória. Esse conjunto de instruções basicamente fazem o seguinte:

1 - Obtém dados de algum dispositivo de entrada (teclado, mouse, HD, placa de rede,etc) e armazena na memória;

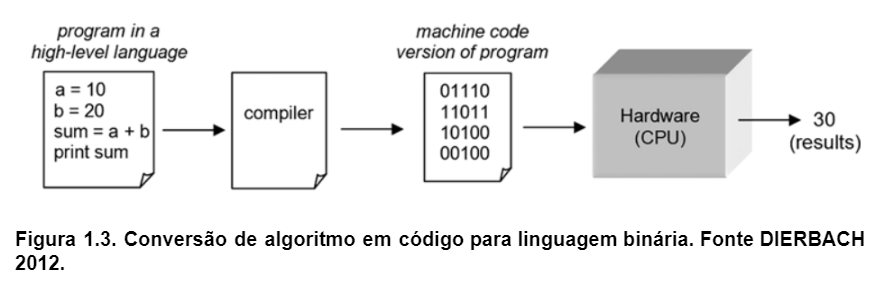
2 - Com as informações na memória efetua cálculos - o que chamamos de processamento, armazenando esses resultados na memória;

3 - Por fim, envia esses resultados desses cálculos da memória para algum dispositivo de saída (impressora, tela, placa de rede,etc).



**Sobre Linguagens de Programação e Algoritmos**

A Linguagem de Programação auxilia no processo de conversão para a linguagem binária do processador, através de uma série de regras que eliminam as ambiguidades.



Algoritmos resolvem qualquer problema. As Linguagens resolvem problemas utilizando os recursos do Sistema Digital (dispositivos de entrada, memória, processamento e dispositivos de saída). Nem todos Algoritmos podem ser implementados por Linguagens.

**O processo para resolver problemas computacionais**

A resolução de um problema computacional não envolve simplesmente o ato de programar um computador. Ele é um processo, sendo que a programação é apenas um dos passos. Antes de escrever um programa é preciso entender e pensar a solução do problema, depois desenvolver o projeto, escrever e testar a solução encontrada.

Uma vez que o algoritmo tenha sido definido, podemos simulá-lo com ferramentas como Scratch, VisuAlg ou Raptor, representá-los em Fluxograma ou implementá-lo diretamente em alguma linguagem de programação (Python, C, C++, Java, Pascal, PHP, dentre outras). Nossa disciplina será focada em Fluxograma.

**Fluxograma**

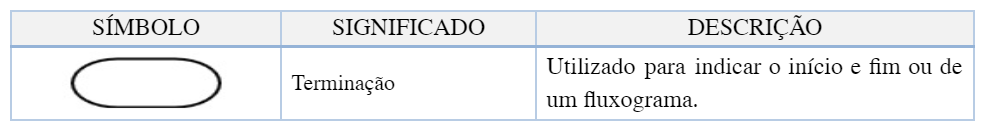
Fluxograma é uma forma padronizada e eficaz para se representar os passos lógicos de um determinado processo (algoritmo), é uma técnica de representação gráfica na qual se utilizam símbolos previamente convencionados, permitindo a descrição clara e precisa do fluxo, ou sequência de um processo, bem como sua análise e redesenho.

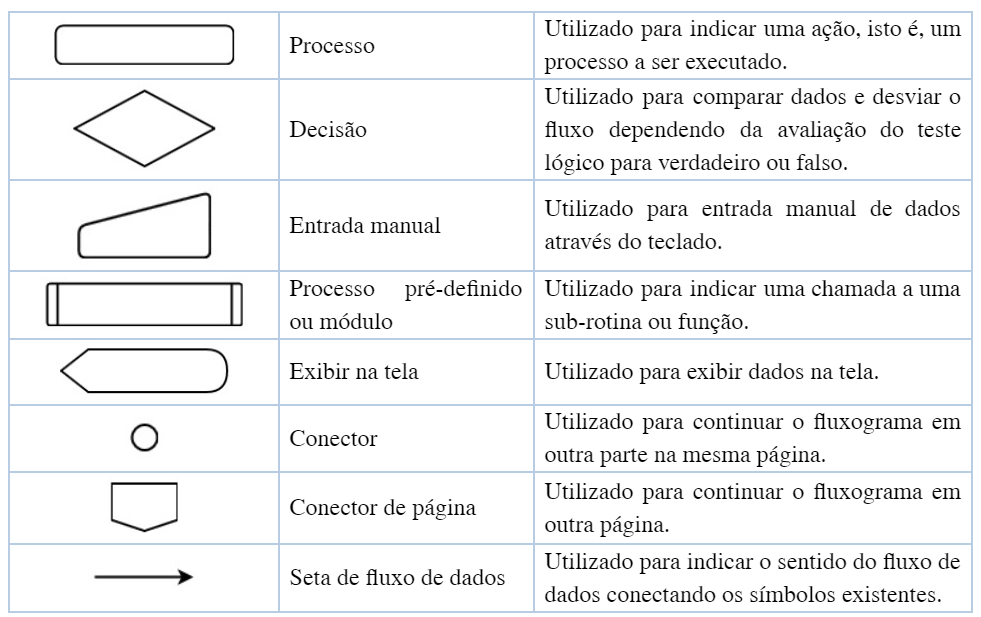
Vantagens: maior clareza na visualização do fluxo de execução (“imagens valem mais do que mil palavras”).

Desvantagens: requer conhecimento de convenções gráficas e dificuldade para fazer correções.

O fluxograma é representado por símbolos e cada um é utilizado para uma finalidade específica, o que torna o fluxograma mais fácil de compreender, pois a representação segue um padrão pré-estabelecido.

Neste curso, usaremos apenas uma pequena parcela dos símbolos disponíveis para uso nos fluxogramas, mostrados na tabela a seguir:

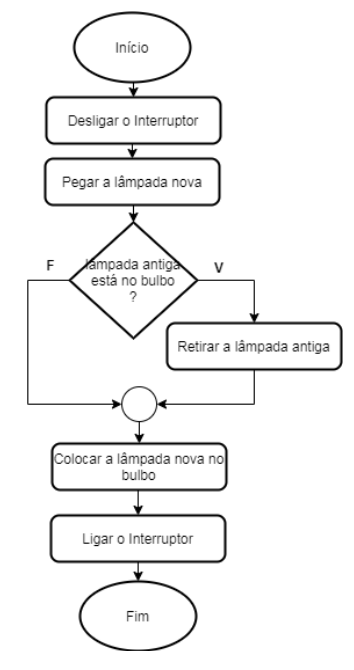




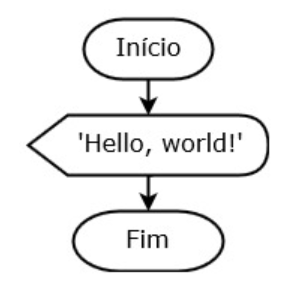
Fluxograma para troca de uma lâmpada. Nesse exemplo, estamos usando um símbolo diferente e novo em relação ao exemplo anterior, o símbolo de decisão.



Este símbolo nos permite criar dois caminhos em nosso fluxograma. Nesse caso, um dos caminhos não fazemos nada e o outro retiramos a lâmpada do bulbo. Tomamos o caminho dependendo da resposta a pergunta - Verdadeiro ou Falso.



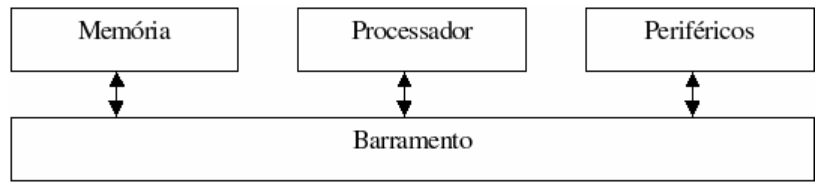
Para exibir uma mensagem na tela, utilizamos o símbolo Exibir. Esse símbolo é uma Saída de Dados, que pode ser utilizado para mostrar o resultado de um processo, um nome ou até mesmo para pedir para que o usuário digite alguma coisa.



Ferramenta para realização dos fluxogramas. - <https://www.drawio.com/>

**Introdução à Arquitetura de Computadores**

**Componentes básicos de um computador**



**Processador (ou microprocessador)**

É um circuito integrado (ou chip). É considerado o cérebro do computador. É ele que executa os programas, faz os cálculos e toma as decisões, de acordo com as instruções armazenadas na memória.

Os microprocessadores formam uma parte importantíssima do computador chamada de UCP (Unidade Central de Processamento), ou em inglês CPU (Central Processing Unit). Antes da existência dos microprocessadores, as CPUs dos computadores eram formadas por um grande número de chips, distribuídos ao longo de uma ou diversas placas. Um microprocessador nada mais é que uma CPU inteira, dentro de um único chip.

Ligando-se um microprocessador a alguns chips de memória e alguns outros chips auxiliares, construiu-se um computador inteiro em uma única placa de circuito, chamada placa mãe dos microcomputadores.

A CPU realiza as seguintes tarefas:

1 - Busca e executa as instruções existentes na memória. Os programas e os dados que ficam gravados no disco (rígido ou disquete) são transferidos para a memória. Uma vez estando na memória a CPU pode executar os programas e processar os dados.

2 - Comanda todos os outros chips do computador

A CPU é composta por:

**Unidade lógica e aritmética (ULA) -**  Assume todas as tarefas relacionadas às operações lógicas (and, or, not, etc.) e aritméticas (adições, subtrações, ...)

**Unidade de controle (UC) -** Assume toda a tarefa de controle das ações a serem realizadas pelo computador, comandando todos os demais componentes de sua arquitetura. É a UC que deve garantir a correta execução dos programas e a utilização dos dados corretos nas operações que as manipulam.

**Os registradores -** A CPU contém internamente uma memória de alta velocidade que permite o armazenamento de valores intermediários ou informação de comando. Esta memória é composta por registradores (ou registros) na qual cada registro possui uma função própria. Um registro memoriza um número limitado de bits, geralmente uma palavra de memória. Os registros mais importantes:

* contador de programa (PC) que aponta para a próxima instrução a executar;
* registro de instrução (IR) que armazena a instrução em execução;
* outros registros que permitem o armazenamento de resultados intermediários.

**Clock**

Clock é um circuito oscilador que tem a função de sincronizar e ditar a medida de velocidade de transferência de dados entre duas partes essenciais de um processamento, por exemplo, entre o processador e a memória principal. Esta frequência é medida em ciclos por segundo, ou hertz .

**Memória**

Constitui de um conjunto de circuitos capazes de armazenar os dados e os programas a serem executados pela máquina. Temos as seguintes categorias de memória:

a ) **A memória principal (ou memória de trabalho)**

É onde normalmente devem estar armazenados os programas e dados a serem manipulados pelo processador. Este tipo de memória aparece como um conjunto de chips que são inseridos na placa mãe do computador.

b) **A memória secundária (ou memória de massa)**

Não é acessada diretamente pela CPU. O acesso é feito através de interfaces ou controladoras especiais. É uma memória do tipo permanente. Possui alta capacidade de armazenamento e um custo menor que o da memória principal. A memória secundária não é formada por chips, e sim por dispositivos que utilizam outras tecnologias de armazenamento (magnética ou óptica). Exemplos: disco rígido, disquete, fita magnética e cd-rom.

**Tipos de memória**

Os chips de memória podem ser divididos em duas grandes categorias:

**RAM (memória de leitura e escrita)**

São chips de memória que podem ser gravados pela CPU a qualquer instante. A CPU usa a RAM para armazenar e executar programas vindos do disco, para ler e gravar os dados que estão sendo processados. É uma memória volátil ( quando o computador é desligado, todos os seus dados são apagados). Por esta razão, os dados e programas devem ficar gravados no disco, que é uma memória permanente.

**Memórias não voláteis**

São memórias cujas informações mantidas não são perdidas caso o computador seja desligado. Exemplo: BIOS (basic input-output system – sistema básico de entrada e saída). Está gravado em uma memória permanente localizada na placa mãe. Tipos de memórias permanentes:

**ROM**

São chips que podem ser lidos pela CPU a qualquer instante, mas não podem ser gravados pela CPU. A gravação é feita pelo fabricante. Este tipo de memória foi usada para armazenar a BIOS.

**PROM**

É uma ROM programável. A gravação pode ser feita apenas uma vez, pois utiliza um processo irreversível.

**EPROM**

É uma ROM programável e apagável. Pode ser programada comportando-se com o uma ROM. A EPROM pode ser apagada com raios ultravioletas de alta potência.

**EEPROM**

É um tipo de memória ROM mais flexível. Pode ser apagada sob controle de software. Utilizada para armazenar as BIOS atuais.

**Memória fora da placa mãe**

A placa mãe contém quase toda a memória de um microcomputador, mas outras placas também podem conter memórias, do tipo RAM e do tipo ROM. As placas de vídeo contém uma ROM com a sua própria BIOS e uma RAM chamada de memória de vídeo

**Placas controladoras**

SCSI (small compact system interface) permite a conexão de diversos periféricos;

IDE (intelligent drive electronics).

**A memória cache**

É uma área reservada de memória que possui duas funções:

* Aumentar o desempenho do computador
* Aumentar o tempo de vida das unidades de disco

Temos dois tipos de memória cache:

* A que vem incorporada à máquina, dessa forma é mais rápida que a memória RAM;
* A que é implementada via software na memória RAM, aumentando o desempenho do acesso ao disco.

**Barramentos**

Um barramento ou bus, é um caminho comum pelo qual os dados trafegam dentro do computador. Este caminho é usado para comunicação e pode ser estabelecido entre dois ou mais elementos do computador.

O tamanho do barramento determina quantos dados podem ser transferidos em uma única vez (16 bits, 32bits, ...)

Um PC possui muitos barramentos, que incluem:

**Barramento do processador**

É o barramento que o chipset (chips de suporte adjacentes contidos na placa mãe) usa para enviar/receber informações do processador.

**Barramento de cache**

É um barramento dedicado para acessar a cache. Usado pelos Pentium Pro e Pentium III.

**Barramento de memória**

Conecta o sub-sistema da memória ao chipset e ao processador.

**Barramento local de entrada/saída (E/S)**

Usado para conectar periféricos de alto desempenho à memória, chipset e processador. Exemplo: placas de vídeo, interface de redes de alta velocidade. Os mais comuns:

* Vesa local bus (VLB)
* Peripheral component inter connect bus (PCI)

**Barramento padrão de entrada/ saída(E/S):**

Usado para periféricos lentos (mouses, placas de som) e também para compatibilidade com dispositivos antigos.

Todos os barramentos possuem duas partes: um barramento de endereçamento (que transfere a informação de onde o dado se encontra) e um barramento de dados (que transfere os dados em si, ou seja, o valor de memória).

**Dispositivos de Entrada e Saída ou Periféricos**

São equipamentos utilizados como portadores das informações que o computador irá processar. Através desses dispositivos, o computador pode armazenar, ler, transferir e receber dados.

Dispositivos de entrada:

Teclado - Mouse - Drive de CD-ROM - Microfone - Scanner

Dispositivos de saída:

Vídeo - Impressora - Alto-falante

Dispositivos de entrada e saída:

Disco rígido - Drive de disquete - Unidade de fita magnética - Modem

**Tipos de comunicação com os dispositivos**

A CPU comunica-se com os periféricos através de circuitos chamados interfaces ou portas de E/S, que implementam a transmissão de dados segundo duas políticas:

* comunicação paralela: impressora;
* comunicação serial: mouse, modem.

**Parte 2 -** [**Estrutura Sequencial - Variáveis, Comandos Básicos e Operadores Aritméticos**](https://sites.google.com/faculdadeimpacta.com.br/lo-p2?pli=1&authuser=2) **-** [**Video**](https://www.youtubeeducation.com/watch?v=cr0xtsc85PI)

**Resumo**

Para elaborar um programa precisamos entender algumas ferramentas básicas. Como vimos na aula 1, um software recebe informações de dispositivos de entrada, armazena na memória, efetua um processamento (cálculo) com as informações que estão na memória, armazena o resultado na memória e por fim envia o resultado desse cálculo que está na memória para um dispositivo de saída. São esses os componentes que iremos ver nesse texto: Variável (memória) e os tipos de informações que são armazenadas na memória, Comandos de Entrada e Saída, processamento (atribuição =). Por fim, como embasamento para se fazer processamento vamos detalhar melhor os operadores aritméticos.

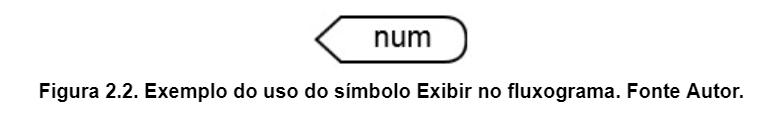
Variáveis e Identificadores

Uma variável tem um nome (identificador) que está associado a um espaço em memória que armazena um valor. Uma variável pode receber diferentes valores durante a execução de um programa, por isso o nome “variável”.

Para atribuir um valor a uma variável, utiliza-se o símbolo de Processo e o operador de atribuição “=”.



Para mostrar um determinado valor na tela, usamos o símbolo de Exibir.

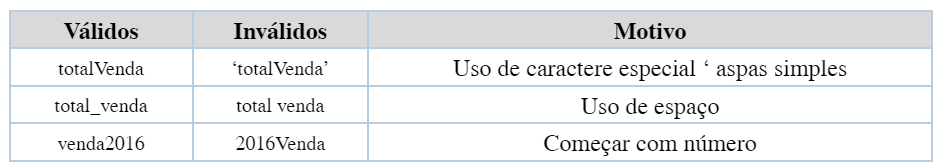


**Como nomear as variáveis**

Um nome ou identificador de uma variável é formado por uma sequência de um ou mais caracteres. Como exemplo, as regras adotadas pelo Python são:

* Pode conter apenas letras, dígitos e underscores (sublinhado);
* Pode começar por uma letra ou underscore;
* Não é permitido o uso de outros caracteres especiais (por exemplo, espaço);
* Não é permitido o uso de palavras reservadas da linguagem;

O identificador deve ser conciso, porém descritivo (idade é melhor que i, tamanho\_nome é melhor que tamanho\_do\_nome\_da\_pessoa).



**Tipos de dados**

**Números**

Um dado numérico é composto por uma sequência de dígitos (0 a 9), um sinal opcional (+ ou –) e um possível ponto decimal (usa-se o ponto e não a vírgula para separar a parte inteira da fracionária). São classificados como inteiro ou de ponto flutuante.

**Texto - letras**

Um texto (string) representa uma sequência de caracteres (letras, dígitos, símbolos especiais). As strings podem ser delimitadas por um par de aspas simples ( ’ ) ou duplas ( ” ).

Exemplos:

“olá”

‘Faculdade Impacta de Tecnologia’

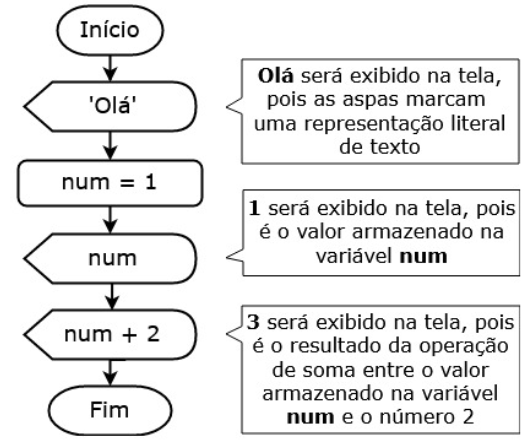
‘Avenida Rudge, 315’

Os dados para serem armazenados na memória devem ser convertidos para números. Com a finalidade de padronizar essa conversão de letras e símbolos para informações em formato numérico para ser armazenada foi criada a tabela ASCII. Veja que por exemplo a letra A (a maiúsculo) é armazenada na memória como o número 65.

**Entrada e saída de dados**

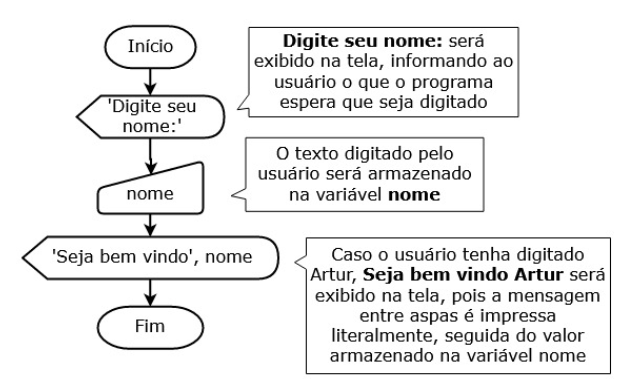
Os programas que iremos escrever poderão obter e apresentar dados aos usuários. Para isso, já vimos em exemplos anteriores o símbolo Exibir (saída de dados).

É possível com isso exibir na tela uma string, o conteúdo de uma variável, o resultado de uma expressão matemática ou booleana, etc.



Se quisermos pedir ao usuário que forneça um valor, via teclado, usaremos o símbolo de Entrada Manual.



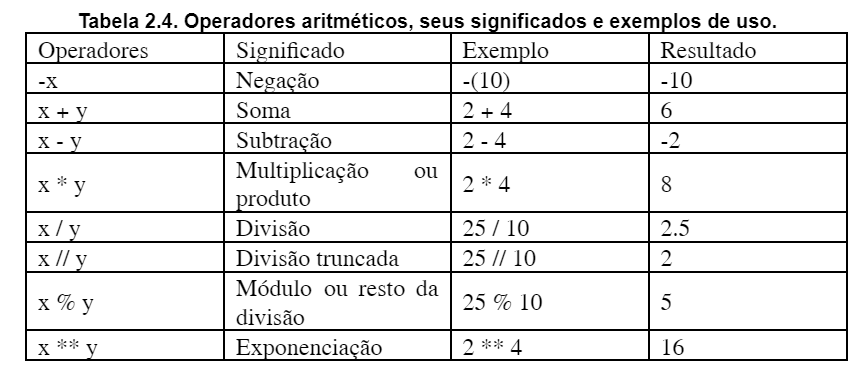


**Operadores e Operadores Aritméticos**

Um operador é um símbolo que representa a operação que pode ser realizada em um ou mais operandos. Operadores que atuam sobre um operando são chamados operadores unários, e operadores que atuam sobre dois operandos são chamados de operadores binários.

**Operadores Aritméticos**

Operadores aritméticos são aqueles cujos operandos são dados de tipo numérico (inteiros ou em ponto flutuante). A tabela a seguir lista os operadores básicos de acordo com a sintaxe do Python, e de maneira geral, a maioria das linguagens de programação tem operadores equivalentes (ou funções que os substituem).



**Expressões e tipos de dados**

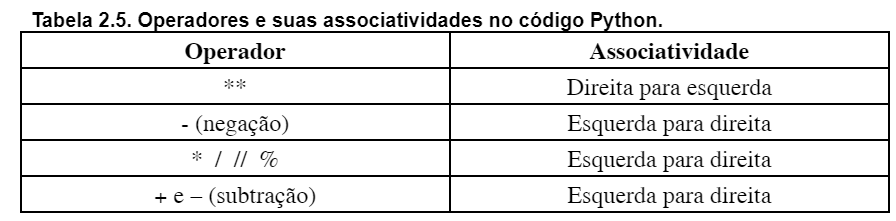
Uma expressão é uma combinação de símbolos (operandos e operadores) que serão avaliados. Assim como na matemática, em programação uma expressão pode conter variáveis, como por exemplo: 4 + 3\*k.

As expressões que avaliam valores do tipo numérico são chamadas de expressões aritméticas.

Para avaliarmos corretamente as expressões acima, é importante sabermos em que ordem devemos realizar as operações, ou seja, é necessário conhecer a precedência dos operadores. O natural seria considerar a precedência da matemática, porém cada linguagem de programação tem sua própria regra, que pode ser igual ou não à da matemática.

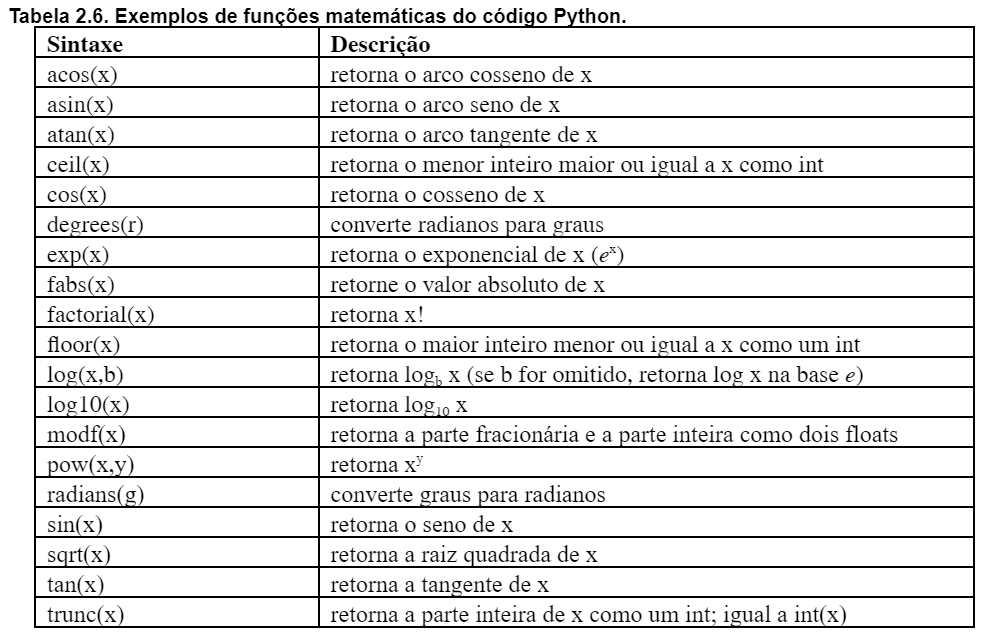
Além da precedência dos operadores, devemos conhecer também como cada linguagem trata os operadores que tenham o mesmo nível de prioridade. À ordem em que isso ocorre damos o nome de associatividade dos operadores.

A tabela a seguir traz a associatividade adotada pelo Python:



**Funções Matemáticas**

Podemos usar um conjunto variado de funções como as trigonométricas (ex: sin, cos, tan, etc) e logarítmicas (ex: log) e também constantes matemáticas (ex: pi). Esse conjunto de funções pode variar dependendo da linguagem de programação utilizada. No nosso curso, utilizaremos as funções disponíveis na biblioteca de funções matemáticas da linguagem Python.

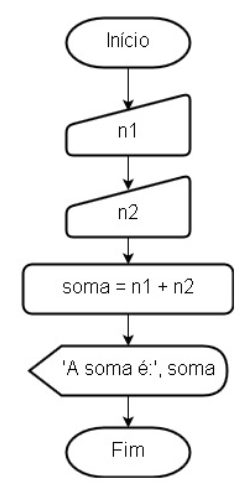


**Estrutura Sequencial**

Nos exercícios construídos anteriormente, sempre foi utilizada a mesma sequência de instruções: receber dados iniciais, realizar operações matemáticas e apresentar resultados finais.

Esta estrutura de controle, caracterizada pela execução das instruções na ordem em que foram escritas, é denominada de Estrutura Sequencial.

Podemos observar esta estrutura no exemplo já trabalhado nas aulas anteriores em que construímos um fluxograma que exibe a soma de dois números informados pelo usuário.



**Exemplo de Fluxograma**

Desenhar um fluxograma que calcula e apresenta o dobro, o triplo e o quadrado de um número digitado pelo usuário.

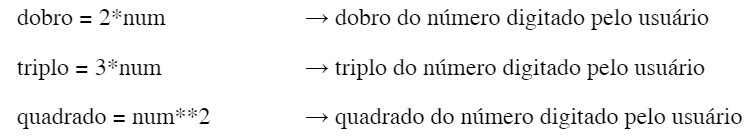
Solução:

Precisamos inicialmente entender o objetivo do exercício. Uma forma interessante de saber onde devemos chegar é identificar no enunciado do exercício o resultado final que está sendo solicitado (saída), os valores que devemos fornecer para conseguirmos fazer o cálculo (entradas) e finalmente elaborar o cálculo necessário (processamento).

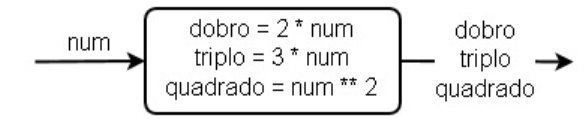
Podemos usar algumas palavras-chave para identificarmos a saída (exibir, apresentar, imprimir). Do texto podemos observar que temos 3 saídas (apresentar dobro, triplo e quadrado). Vamos guardar essas informações em variáveis e chamá-las de dobro, triplo e quadrado.

As entradas também podem ser identificadas por palavras-chave como (dado, digitado, informado, obtido). Nesse caso temos como entrada um único número digitado pelo usuário. Vamos atribuí-lo à variável num.

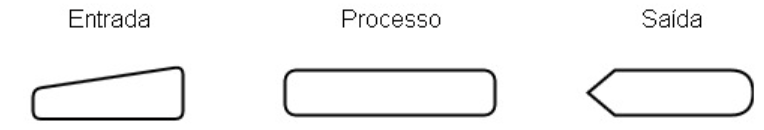
Agora precisamos identificar qual o cálculo que devemos fazer com a entrada (num) para obtermos as saídas pedidas (dobro, triplo e quadrado). Seguem os cálculos:



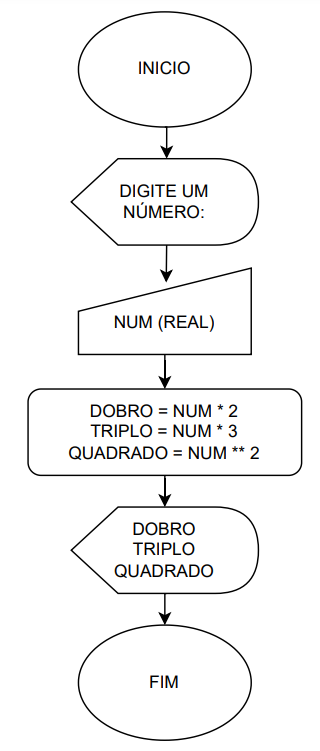
Em seguida, podemos desenhar o seguinte diagrama de entrada/processamento/saída:



E fazendo uso dos seguintes símbolos:

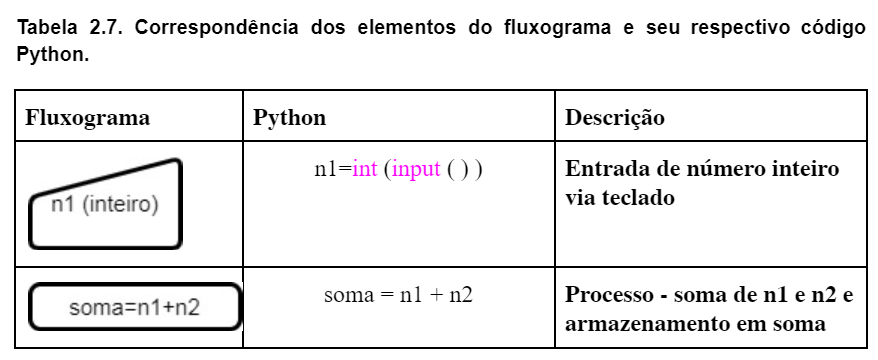


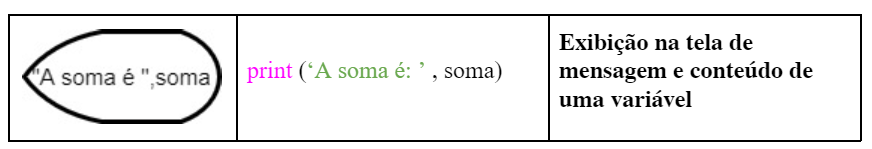
Podemos finalmente desenhar o fluxograma:



Exemplo: Fluxograma e Conversão em Python.

Nesse exemplo, vamos criar um fluxograma para mostrar a soma entre dois números inteiros. Esses números serão digitados pelo usuário e iremos armazená-los nas variáveis n1 e n2. Em seguida, vamos realizar o processo de soma, que consiste em recuperar os valores salvos, somá-los e atribuir o resultado à variável soma. Ao final, este resultado deverá ser apresentado ao usuário.





Faça um fluxograma que:

Defina 3 variáveis com valores que armazenem:

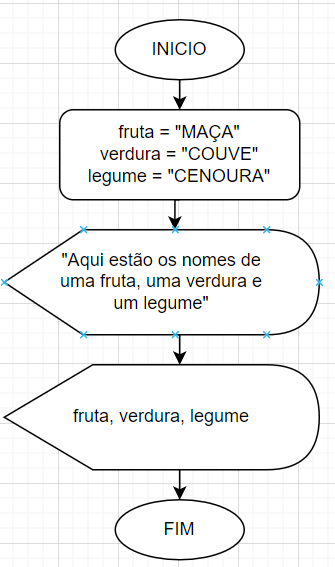
O nome de uma fruta;

O nome de verdura; e

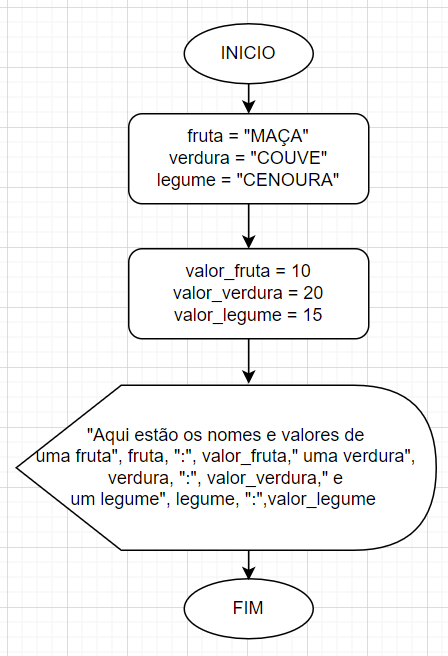
O nome de um legume.

Mostre uma mensagem “Aqui estão os nomes de uma fruta, uma verdura e um legume:”;

Mostre, após essa mensagem, o nome da fruta, da verdura e do legume, respectivamente.



Altere o fluxograma anterior para conter mais três variáveis, com o preço unitário da fruta, verdura e legume. Depois, mostre o preço de cada produto após o nome (nome ao lado do preço, por exemplo: “Maçã: 10”).



**Parte 3 -** [**Como resolver problemas computacionais**](https://sites.google.com/faculdadeimpacta.com.br/lo-p3?pli=1&authuser=2) **-** [**Video**](https://www.youtubeeducation.com/watch?v=v5N9-8Sfpa0)

**Resumo**

Neste texto estão expostas considerações sobre uma técnica simplificada para resolver problemas computacionais salientando os principais pontos: Entendimento, Projeto, Implementação e Testes. Para cada um desses pontos são expostos seus conceitos e exemplificados com vários casos.

**O processo de Resolução de Problemas Computacionais**

Muitos acreditam que para desenvolver um programa basta programar em alguma linguagem. Na verdade, a programação ou construção do programa é apenas uma das etapas desse processo. Antes de um programa ser construído existem passos importantíssimos que se não forem feitos podem levar a um grande prejuízo em termos de tempo e investimentos. Antes da construção precisamos pensar no projeto do programa e antes do projeto precisamos deixar bem claro o que precisa ser feito. Após o programa ser escrito, ele precisa ser testado para verificar o que atende ao que foi proposto. (DIERBACH, 2012)

**As Etapas da Resolução de Problemas Computacionais**

Podemos dividir em quatro as etapas Técnica de Solução de Problemas Computacionais:

1 - Análise e Entendimento

2 - Projeto

3 - Construção ou Implementação

4 - Testes

**Análise e Entendimento**

Esta é a etapa mais importante. Aqui definimos claramente o que deve ser feito, onde devemos chegar, qual o objetivo do programa.

Para exemplificar as Etapas, vamos utilizar o seguinte problema:

Desenhar um fluxograma que receba dois números, calcule e exiba a soma desses números.

Perceba que muitas vezes nos deparamos com um problema incompleto ou mal definido. A primeira e mais importante questão no entendimento é com relação ao objetivo do problema. Está claro o objetivo desse problema? No final das contas, qual a necessidade que esse programa deverá atender?

Para deixar mais claro o entendimento vamos modelar o processo envolvido nesse caso separando as saídas, as entradas e o processamento:

**Saídas:** a soma de dois números. Essa é a informação mais importante. É o nosso objetivo. Uma dica para ajudar a encontrar essa informação, é procurar no texto palavras que tenham a ideia de saída como: exibir, apresentar, mostrar, etc.

**Entradas:** dois números. São os dados que precisamos usar como matéria-prima para conseguir atingir nossa saída, nosso objetivo. Uma dica para ajudar a encontrar essa informação, é procurar no texto palavras que tenham a ideia de entrada como: receba, digite, dados, etc.

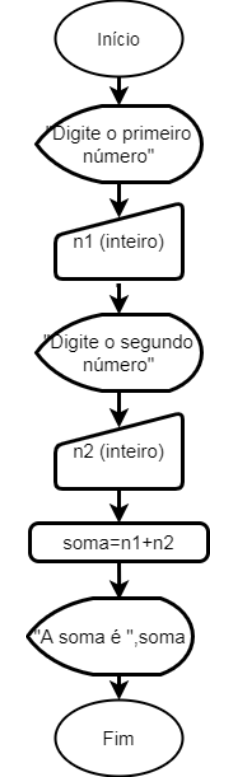
**Processamento:** calcule a soma. O processamento é o que temos que fazer com as entradas para produzir as saídas. Uma dica para ajudar a encontrar essa informação, é procurar no texto verbos de ação envolvendo as entradas e saídas como: calcular, resultar, somar (aqui entra qualquer verbo que indica cálculo ou processamento), etc.

**Projeto**

Nesta etapa, devemos achar uma estratégia de solução de maneira a conseguir efetuar o processamento. Estamos procurando aqui um algoritmo que consiga transformar as entradas nas saídas desejadas. Devemos lembrar que um algoritmo além de envolver processos, passo-a-passo de uma solução também pode envolver uma determinada forma de armazenar os dados para conseguir efetuar o passo-a-passo. No nosso exemplo, esse algoritmo é uma simples soma.

**Implementação ou Construção**

Nesta etapa devemos definir qual ferramenta/linguagem será usada para a construção do algoritmo definido na etapa de projeto. Aqui aplicamos as ferramentas necessárias de lógica de programação para implementar o algoritmo. Essa é a etapa comumente conhecida como “programar”. No nosso exemplo, utilizaremos o fluxograma:



Perceba que a utilização de um fluxograma permite que posteriormente ele possa ser convertido para qualquer linguagem que se queira. O Fluxograma carrega com ele toda a inteligência do entendimento e projeto de software.

**Testes**

Essa é uma etapa costumeiramente esquecida por muitos programadores. Não menos importante que as outras, essa etapa bem feita define a qualidade do programa. Tanto no quesito mais importante de atender ao que foi proposto atingindo seu objetivo com precisão, quanto na eficiência da maneira como foi construído. A disciplina de testes é muito abrangente, vamos mostrar aqui apenas algumas das muitas técnicas de elaboração de testes.

**Como testar um programa**

**Teste da caixa-preta**

Existem formas de testar a corretude de programas. Uma das formas é o teste de caixa-preta, cujos passos resumidamente são:

1 - Escolhe-se uma entrada cuja saída correta correspondente seja conhecida, sem precisar usar o programa que será testado;

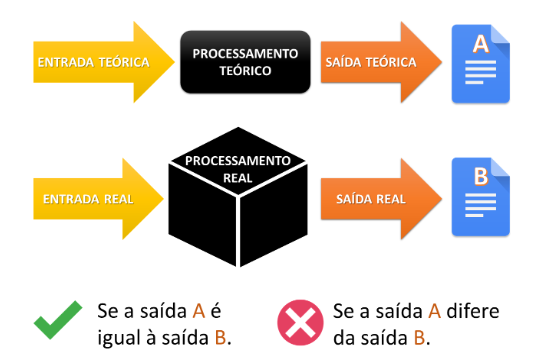
2 - Simula-se a execução do fluxograma usando a entrada escolhida (TESTE DE MESA);

3 - Compara-se a saída simulada com aquela teórica inicialmente esperada;

4 - Caso haja diferença entre a saída teórica e a saída real, muito provavelmente o programa está incorreto e necessita de correções. As devidas modificações são feitas e retorna-se ao passo (b);

5 - Caso não existam divergências entre as saídas, opta-se por: (I) encerrar os testes ou (II) submeter o programa a outros casos de teste.

Note que testes de caixa-preta não consideram a estrutura interna do programa, ou seja, o algoritmo usado para construir o programa, assim como a linguagem de programação em que foi implementado, não importam.



Lembre-se, testes não garantem, necessariamente, que um programa está correto. Um programa que seja aprovado em todos os casos de teste aos quais foi submetido não está obrigatoriamente correto, pois pode haver um caso de teste ausente que geraria uma falha.

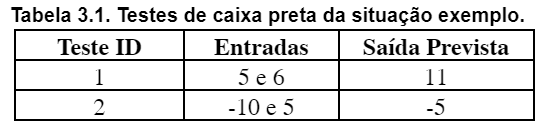
Para garantir a corretude de um programa por meio de testes é necessário fazer um teste exaustivo. Testes exaustivos submetem o programa a todas as entradas esperadas possíveis, o que muitas vezes é impraticável. Imagine quantas possíveis entradas esperadas existem para um programa que soma dois números inteiros e exibe o resultado. Infinitas!

Uma forma de garantir a corretude de um programa sem precisar construir testes, é por meio de prova formal, porém esse método exige maiores conhecimentos matemáticos, técnicas avançadas de análise de algoritmos e criatividade. Provas formais costumeiramente demandam mais recursos financeiros e tempo para serem satisfatoriamente concluídas em programas mais extensos.

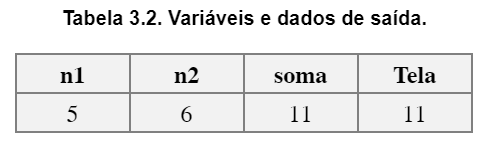
Geralmente, para sistemas não críticos (aqueles que não gerenciam laboratórios com doenças altamente contagiosas; não administram usinas nucleares; não controlam produção de foguetes; não automatizam cirurgias, aviões, trens, etc.) são montadas sequências de testes com boa abrangência, testando principalmente as extremidades das possíveis entradas esperadas. Isso amplia o nível de confiança nas soluções propostas. Em nossas aulas optamos por essa abordagem simplificada para o teste de programas.

O uso de testes de mesa pode auxiliar na execução de testes de caixa-branca, pois permite que toda instrução que implique em mudança nas variáveis seja representada em uma linha-coluna da tabela. No caso de testes de mesa com estruturas condicionais, é facilitador acrescentar uma coluna com a expressão da condição de seleção, sendo útil para perceber erros na definição da condição (frequentes com iniciantes).

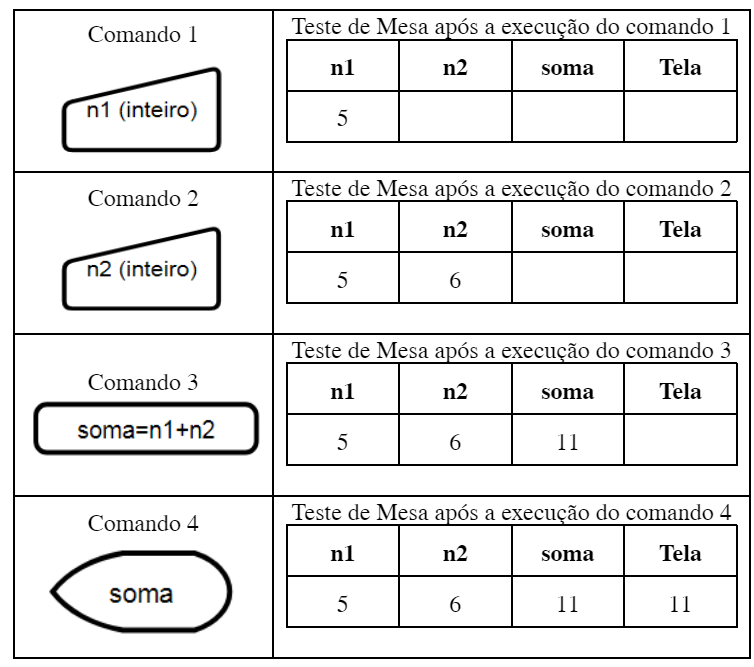
Vamos aplicar a técnica da caixa preta ao exercício da soma de dois números, Vamos pensar em dois testes:



Para a elaboração do teste de mesa, criamos uma tabela com as variáveis usadas e com um coluna (Tela) que mostra o que está sendo exibido na tela:



Em seguida, a cada comando executado no fluxograma, registramos o valor que as variáveis envolvidas vão recebendo, conforme mostrado na tabela a seguir, na qual executamos o teste 1:



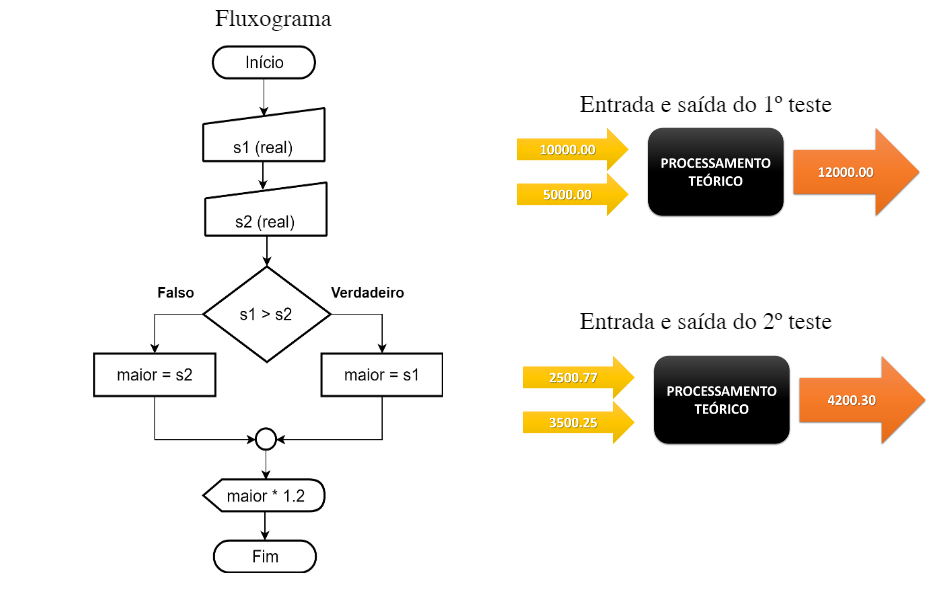
**Teste da caixa-branca**

Também é possível usar testes de caixa-branca, em que a estrutura interna do programa é avaliada, ou seja, cada instrução do algoritmo implementado em uma linguagem de programação será analisada de acordo com a sequência de execução. Neste formato de teste é necessário "abrir a caixa e ver o que está dentro".

O uso de testes de mesa pode auxiliar na execução de testes de caixa-branca, pois permite que toda instrução que implique em mudança nas variáveis seja representada em uma linha-coluna da tabela. No caso de testes de mesa com estruturas condicionais, é facilitador acrescentar uma coluna com a expressão da condição de seleção, sendo útil para perceber erros na definição da condição (frequentes com iniciantes).

Bons testes de caixa-branca para programas com estruturas de seleção buscam passar por todos os caminhos do algoritmo, garantindo que sejam executadas e analisadas tanto as instruções do "bloco do caminho se verdadeiro" quanto do "bloco do caminho se falso", se houver.

Pense no seguinte exemplo: crie um fluxograma que leia como entrada dois salários distintos e exiba uma mensagem com o valor do maior deles, acrescido de 20% de bônus. Vamos construir um fluxograma, fazer dois testes de caixa-preta (sem comparação com as saídas de um programa real, pois não o codificamos) e dois testes de caixa-branca.



Como fazer o teste de mesa? Video: [(#9) Como fazer teste de mesa?](https://youtu.be/Atcfaafvs4M)

**Parte 4 -** [**Estrutura de Seleção - Operadores e Expressões Lógicas**](https://sites.google.com/faculdadeimpacta.com.br/lo-p4?pli=1&authuser=1) **-** [**Video**](https://www.youtubeeducation.com/watch?v=OlG_JnEP1pk)

**Resumo**

Nem todos os comandos de um programa precisam ser sempre executados. Existem situações em que precisamos executar os comandos seletivamente e inclusive às vezes não executar. Vamos aprender nesse texto sobre a ferramenta que permite que isso seja feito - Estrutura de Seleção. Como base para a utilização dessa ferramenta iremos aprender um pouco mais sobre os operadores e as expressões lógicas. Discutimos também os conceitos de estrutura de seleção simples e composta. Tanto a estrutura de seleção simples quanto a estrutura de seleção composta utilizam o símbolo de decisão. Através dessa decisão definimos obrigatoriamente dois caminhos no nosso fluxograma. Abaixo são expostos os conceitos dessas estruturas assim como exemplos de utilização. Ao final é mostrado um exemplo usando fluxograma com sua conversão para Python.

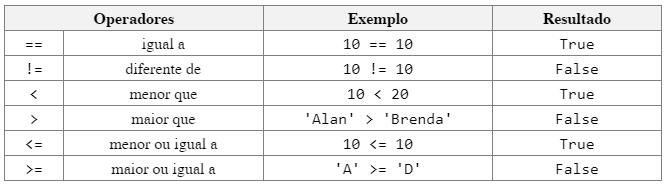
**Expressões lógicas**

Como vimos anteriormente, o resultado de expressões aritméticas são números inteiros ou reais. Em expressões lógicas, o resultado gerado é um valor lógico (também conhecido como booleano), ou seja, Verdadeiro (True) ou Falso (False).

Novamente, como analogia, em expressões aritméticas os operandos costumam ser números e os operadores são aritméticos (adição, subtração, multiplicação, divisão e etc.), já com expressões lógicas os operandos também podem ser números, porém há presença de operadores relacionais e/ou operadores lógicos.

**Operadores relacionais**

Os operadores relacionais são usados em comparações entre valores. O resultado da comparação será um valor lógico, como exposto no quadro a seguir:

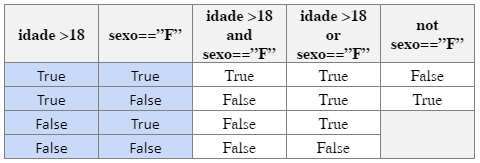


**Operadores lógicos**

Os operadores lógicos podem ser usados para construir expressões lógicas mais complexas, permitindo a combinação de comparações. Consideraremos três operadores: E (and), OU (or) e NÃO (not).

* O operador lógico and resulta verdadeiro somente se os dois operandos forem verdadeiros.
* O operador lógico or resulta verdadeiro se ao menos um dos dois operandos for verdadeiro.
* O operador lógico not resulta verdadeiro se o operando for falso e resulta em falso caso seja verdadeiro.

Veja a tabela verdade dos operadores lógicos:



É preciso ser cauteloso ao usar operadores lógicos. Por exemplo, na matemática, para indicar que um valor está dentro de um determinado intervalo escrevemos:

(1 <= num <= 10)

No entanto, na maioria das linguagens de programação esta expressão não faz sentido. Para entender porque, vamos assumir que a variável num tem o valor 15. A conversão da expressão anterior para uma vasta gama de linguagens de programação seria assim:

(1 <= num <= 10) → (1 <= 15 <= 10) → (True <= 10) → ???

Em diversas linguagens de programação a expressão será avaliada de modo diferente ao tradicionalmente aceito na matemática. Inicialmente se avaliaria a primeira parte da expressão lógica (1 <= num) que geraria um resultado booleano True; na sequência seria avaliada a segunda parte da expressão (True <= 10). Porém, não faz sentido verificar se True é menor ou igual a 10. A forma comumente aceita para a expressão anterior usaria o operador and:

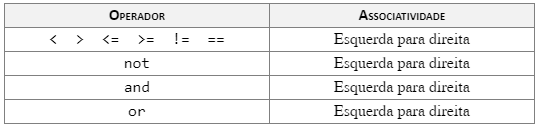
(1 <= num) and (num <= 10)

ou

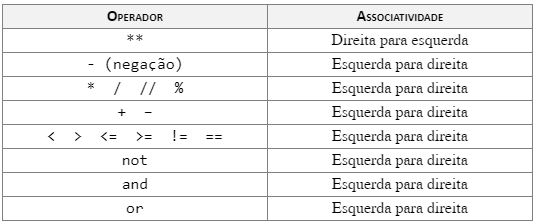
(num >= 1) and (num <= 10)

**Precedência de operadores (prioridade na avaliação)**

A precedência dos operadores relacionais e lógicos, bem como suas associatividades, é definida conforme explicitado na seguinte tabela:

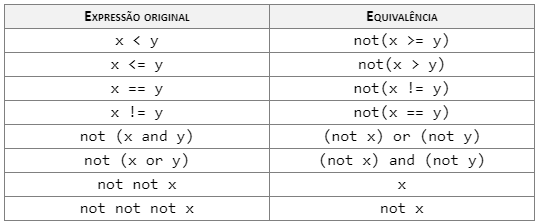


Expressões lógicas também podem conter operadores aritméticos. Agora podemos expandir nossa tabela de operadores conhecidos e suas respectivas precedências: (Aritmético / Relacional / Lógico)



**Expressões lógicas equivalentes**

Assim como há expressões aritméticas equivalentes, como a\*(b+c) é igual a a\*b + a\*c, também existem expressões lógicas equivalentes. É importante reconhecer as equivalências para simplificar expressões e outras finalidades relacionadas.



**Estruturas de seleção (estruturas condicionais)**

A partir de agora vamos estudar as estruturas condicionais que permitem que programas executem sequências diferentes de instruções, dependendo da avaliação de uma expressão lógica.

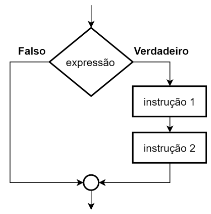
Em programação, o uso de condições para permitir a escolha de executar ou não um trecho de código é muito utilizado, principalmente quando precisamos incluir no programa condições de controle, para evitar situações não permitidas que podem resultar em erros. Por exemplo, para evitar divisões por zero.

Quando há apenas um bloco de instruções cuja execução está condicionada ao resultado de uma expressão lógica, usamos uma estrutura de seleção simples.

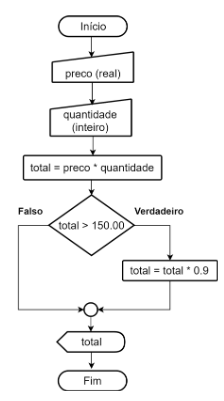
Existe outra estrutura, chamada de estrutura de seleção composta, em que ainda temos apenas uma condição a ser avaliada, porém dois caminhos que podem ser seguidos. Um dos caminhos leva à execução do bloco de instruções condicionado ao resultado verdadeiro e o outro à execução do bloco de instruções condicionado ao resultado falso.

**Estrutura de seleção simples**

Na estrutura de seleção simples ou estrutura condicional simples, um bloco de código só será executado caso a expressão da condição da estrutura resulte em um valor lógico verdadeiro. Caso a condição resulte falsa, o bloco será ignorado e as instruções posteriores, externas à estrutura condicional, serão executadas.



Suponha a seguinte situação: dado o preço de um produto comercializado por uma loja e a quantidade comprada por um cliente, exiba o valor total da compra. Considere que se o total for maior do que R$ 150,00 haverá desconto de 10%. Um fluxograma para esta situação pode ser representado como a seguir:

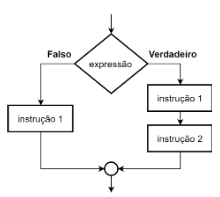


Note que o valor da variável total sempre será exibido, independentemente se houver ou não desconto, porém se o valor de total for superior a 150.00 o trecho de código que o reduz em 10% será executado, pois a condição da seleção resultará verdadeiro.

**Estrutura de seleção composta**

Na estrutura de seleção composta ou estrutura condicional composta, há dois caminhos possíveis, cada um com um conjunto de instruções, mas apenas um deles será executado.

Um bloco de código só será executado caso a condição da estrutura de seleção resulte em um valor lógico verdadeiro. Caso a condição resulte falso, outro bloco de códigos será executado.



Suponha a seguinte situação: dada a nota de um aluno, exibir 'aprovado' caso esta seja maior ou igual a 6.0, e 'reprovado' caso contrário. Um fluxograma para esta situação pode ser representado assim:



Note que neste caso somente uma mensagem será exibida. Se a condição da seleção resultar verdadeiro, será exibido 'aprovado', caso contrário será exibido 'reprovado'. Costumeiramente, chamamos o lado que é executado quando a condição resulta falso de "senão".

**Exemplo 1:**

Elaborar um Fluxograma que recebe uma idade e exibe a mensagem Maior de Idade, caso a idade seja maior ou igual a 18 e exiba a mensagem Menor de Idade, caso contrário.



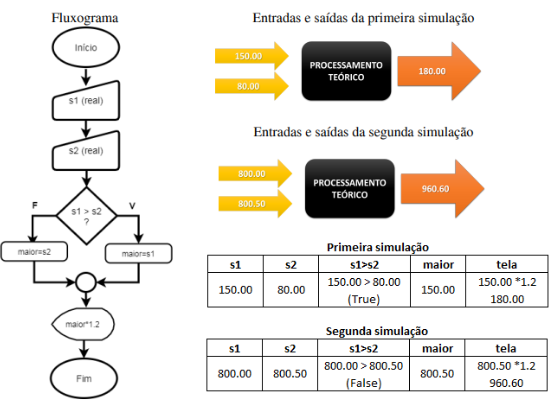
**Saídas:**

Nesse exemplo, veja que a saída são duas mensagens: Maior de Idade (caso a idade seja maior ou igual a 18) e menor de idade (caso a idade seja menor que 18).

**Entradas:**

Temos uma entrada - idade, que é um número inteiro.

Na criação do fluxograma, percebemos que primeiro fizemos a entrada de dados - idade e após isso utilizamos uma estrutura de seleção justamente para definir qual das duas mensagens será exibida dependendo do resultado do teste lógico envolvendo a idade (idade>=18). Caso o teste seja verdadeiro, significa que a idade digitada pelo usuário é maior ou igual a 18. Sendo assim, seguimos com o caminho da direita e será exibida a mensagem “Maior de idade”. Caso o resultado do teste seja falso, será exibida a mensagem “Menor de idade”. Perceba que quando usamos uma estrutura de seleção nunca serão usados os dois caminhos simultaneamente. Se usamos o caminho do verdadeiro, o caminho do falso não é usado e vice-versa.



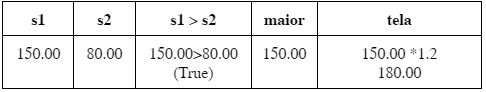
Nesse exemplo temos 2 entradas (os dois salários) que foram armazenadas em duas variáveis, s1 e s2. Criamos uma terceira variável com o objetivo de armazenar o maior salário chamada maior.

Caso o primeiro salário seja maior que o segundo (s1>s2), a variável maior recebe o valor do primeiro salário (maior=s1). Receberá o segundo salário caso esse teste s1>s2 seja falso - (maior=s2). Ao final, após a estrutura de seleção, é apresentado o valor da variável maior multiplicado por 1.2 (aumento de 20%) na tela.



No diagrama acima estamos mostrando também dois testes e seus resultados teóricos.Veja nos teste abaixo temos como entradas 150.00 e 80.00 e veja que na saída é apresentado o valor 180.00 (150\*1.2).

Podemos seguir a simulação desse teste pelo quadro abaixo:



Nesse quadro notamos que são mostrados em colunas as variáveis s1,s2 e maior, a expressão lógica da decisão s1>s2 e o que é exibido na tela.

A simulação ocorre de cima para baixo. Então, temos:

1 - a variável s1 recebe 150.00 (digitado pelo usuário)

2 - a variável s2 recebe 80.00 (digitado pelo usuário)

3 - é feito um teste s1>s2 (150.00>80.00) resultando em Verdadeiro (o fluxo toma o caminho da direita

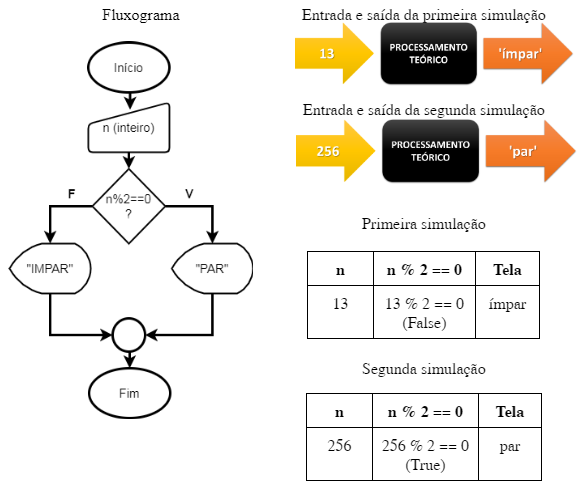
4 - a variável maior recebe 150,00 que é o valor da variável s1

5 - A estrutura de seleção finaliza e é exibo cálculo maior\*1.2 (150.00\*1.2) que apresenta na tela o valor 180.00

O segundo teste com os valores 800.00 e 800.50 ocorre de maneira semelhante. Observe que neste exemplo o objeto de escolha da estrutura de seleção foram comandos de atribuição (cálculos), enquanto que no exemplo anterior foram comandos de saída.

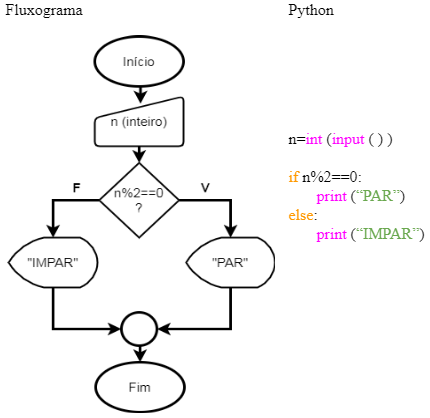
**Exemplo 2**

Vamos elaborar um fluxograma de um programa que recebe um número inteiro e exibe uma mensagem indicando se ele é par ou ímpar (vamos usar o operador % para obter o resto de uma divisão inteira).

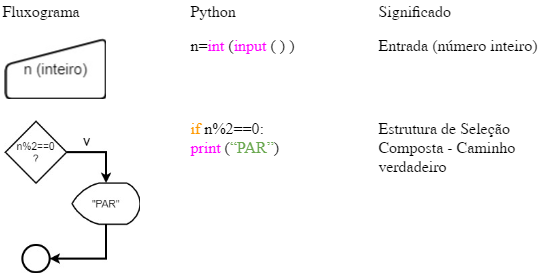


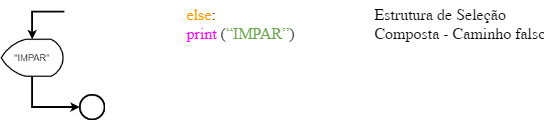
Nesse exemplo, diferentemente aos anteriores, veja que na decisão foi colocado um cálculo (n%2==0). A expressão lógica é avaliada depois do cálculo (n%2). Esse é o resto de n que foi digitado pelo usuário por 2. Quando obtemos o resto da divisão por 2 igual a 0 significa que o número é par. Caso seja 1 é ímpar.

**Conversão para Python**



Veja a equivalência dos comandos:





Estrutura condicional - Fluxograma -[aula09 - Estrutura condicional - Fluxograma](https://youtu.be/kN8Kk1BNrp0)

Fluxograma (estrutura de decisão) + exemplo - <https://youtu.be/PBFeqj1_2tA?si=L3_-vrw1X695TJ1j>

**Parte 5 -** [**Estrutura de Seleção Encadeada**](https://sites.google.com/faculdadeimpacta.com.br/lo-p5?pli=1&authuser=1) **-** [**Video**](https://www.youtubeeducation.com/watch?v=vTWk4RYEy3k)

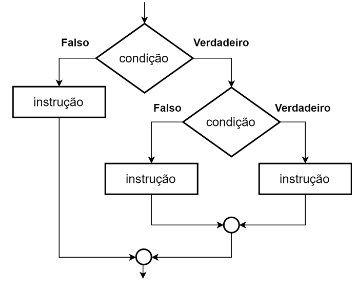
**Resumo**

Neste texto é discutido o conceito de estrutura de seleção encadeada ou aninhada. Como exemplificação mostra-se uma implementação de um exemplo usando fluxogramas, um segundo exemplo usando fluxogramas juntamente com uma simulação. Ao final é mostrado um terceiro exemplo usando fluxograma com sua conversão para Python.

**Estrutura de seleção encadeada**

Tanto na estrutura de seleção simples quanto na seleção composta há apenas uma condição para ser avaliada e que condicionará o fluxo de execução do programa. Portanto, há no máximo uma bifurcação para um caminho, entre dois possíveis.

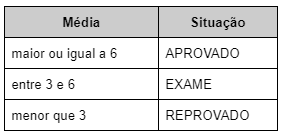
E quando é necessário resolver um problema que possui mais de dois caminhos possíveis? Para lidar com situações como essa, podemos usar uma estrutura de seleção encadeada (também conhecida como estrutura de seleção aninhada).

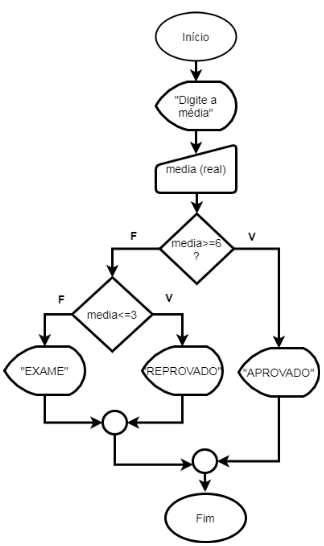


A estrutura de seleção encadeada ocorre quando uma seleção tem como ação uma outra seleção ou quando temos mais de duas possibilidades para uma decisão. A seleção encadeada ou aninhada é o agrupamento de uma ou várias seleções internas a uma seleção externa.

**Exemplo 1:**

Desenhar um fluxograma que obtém a média de um aluno e exibe a sua situação segundo a tabela:





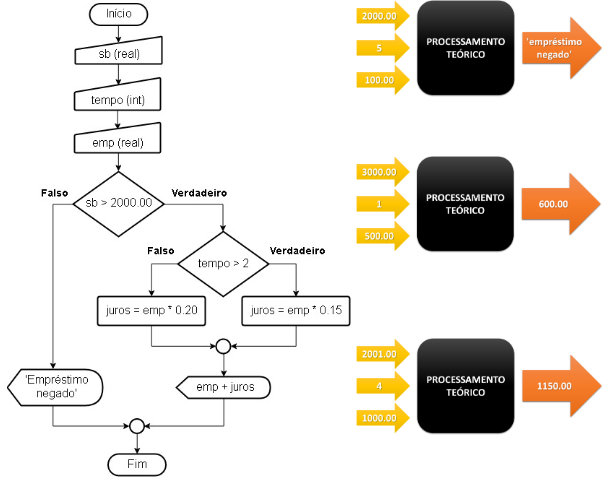
Perceba que nesse fluxograma temos 3 possibilidades de caminhos a serem percorridos, uma para cada status APROVADO, REPROVADO e EXAME. Na primeira decisão media >=6 definimos dois caminhos. No caminho da direita (verdadeiro), temos apenas uma possibilidade (media>=6 portanto aluno aprovado). Já no caminho da esquerda (falso), temos duas possibilidades: média <=3, ou seja, reprovado e média entre 3 e 6 (como ele caiu no caminho da esquerda, o valor da média é menor que 6). Precisamos, portanto, de mais de uma estrutura de seleção para que possamos decidir em qual dessas duas possibilidades a média testada se encontra. Perceba, ainda, que essa outra estrutura de seleção está dentro da primeira (no ramo esquerdo dela) e, portanto, é uma estrutura encadeada ou aninhada (ninho).

**Exemplo 2:**

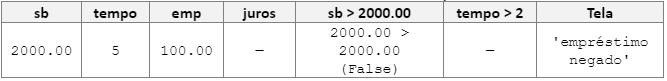
Considere a seguinte situação: um banco oferece uma modalidade de empréstimo de parcela única para qualquer cliente cujo salário bruto ultrapasse R$ 2.000,00. Essa modalidade também considera que, caso o cliente tenha mais de dois anos de contrato, serão cobrados juros de 15% sobre o valor emprestado, caso contrário os juros serão de 20%. Se o salário bruto não atingir o mínimo estipulado, o empréstimo será negado. O gerente do banco quer um programa que dados como entrada o salário bruto (em reais e centavos), o tempo de contrato do cliente com o banco (em anos) e o valor solicitado de empréstimo (em reais e centavos), exibe o valor da dívida do cliente, isto é, o valor emprestado acrescido de juros.

Note que com apenas os recursos das estruturas de seleção simples e composta não seria simples modelar esta situação, pois o enunciado evidencia que existem mais de dois possíveis fluxos de execução no algoritmo, que dependem dos dados passados como entrada.

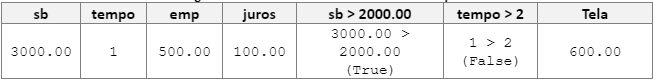
Um fluxograma adequado para esse problema é ilustrado a seguir:



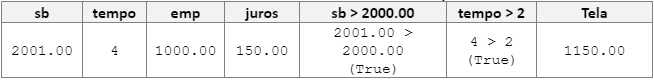
**1º Teste de mesa**



**2º Teste de mesa**



**3º Teste de mesa**

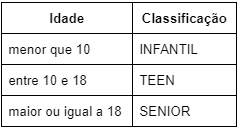


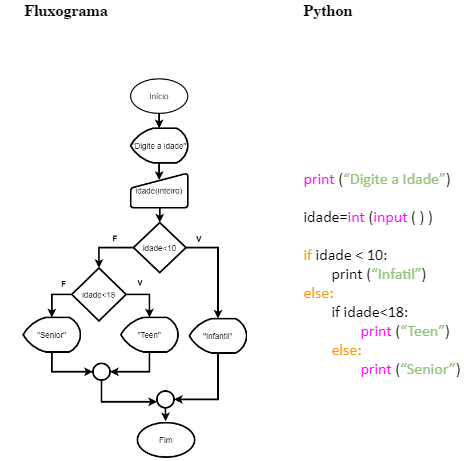
**Comparativo entre fluxograma e Python**

Abaixo podemos identificar um fluxograma e sua conversão para python.

Exemplo 3:

Desenhar um fluxograma que obtém uma idade e exibe uma mensagem segundo a tabela abaixo:





Lógica de Programação - Condicional (ou seleção) encadeada. <https://youtu.be/0k8vpmECZDo>

**Parte 6 -** [**Funções**](https://sites.google.com/faculdadeimpacta.com.br/lo-p6?pli=1&authuser=1) **-** [**Vídeo**](https://www.youtubeeducation.com/watch?v=468pCN5Yk0I)

**Resumo**

Você tem ideia de como três programadores conseguem trabalhar em um mesmo programa? Existe uma maneira simples de reaproveitar um código que resolve um problema em um outro programa?

Uma função nada mais é do que um trecho de código que damos um nome. Os parâmetros e o retorno potencializam muito o uso das funções. Esses são os assuntos discutidos nesse texto.

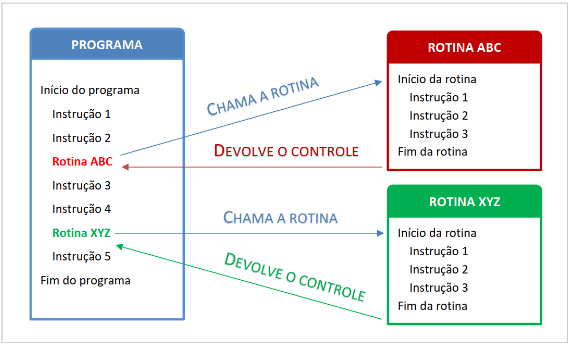
**Introdução**

Até agora vimos conceitos fundamentais, tais como entrada e saída, variáveis, operadores, expressões e estruturas de controle de fluxo de execução (DIERBACH, 2012). Contudo, do ponto de vista prático, essas estruturas sozinhas, ainda não são suficientes. Vamos pensar em problemas mais complexos. Por exemplo, um smartphone contém cerca de 10 milhões de linhas de código, imagine o esforço para desenvolver e depurar softwares dessa dimensão (DIERBACH, 2012).

Para gerenciar a complexidade de grandes problemas, é útil quebrá-los em subproblemas menores. Então, cada subproblema pode ser analisado e resolvido separadamente, permitindo soluções mais simples, legíveis e reutilizáveis. Algo importante para não perder tempo reconstruindo códigos (DIERBACH, 2012). Nesse sentido, funções são blocos de construção fundamentais no desenvolvimento de software e muito usadas no mercado de trabalho.

**Rotinas e Funções**

Uma rotina é definida como um grupo de instruções que executa alguma tarefa bem definida e identificada por um nome (DIERBACH, 2012). Algoritmos maiores, por exemplo programas completos, podem ser compostos por diversas rotinas que são independentes do programa principal (quanto maior a independência melhor).



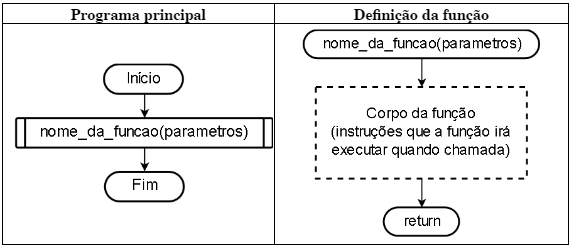
Uma rotina pode ser invocada ou chamada quantas vezes for necessário em um dado programa. No momento em que é chamada (a execução do programa chega na linha do código onde está o nome da rotina), o controle de execução é redirecionado para o código da rotina (DIERBACH, 2012).

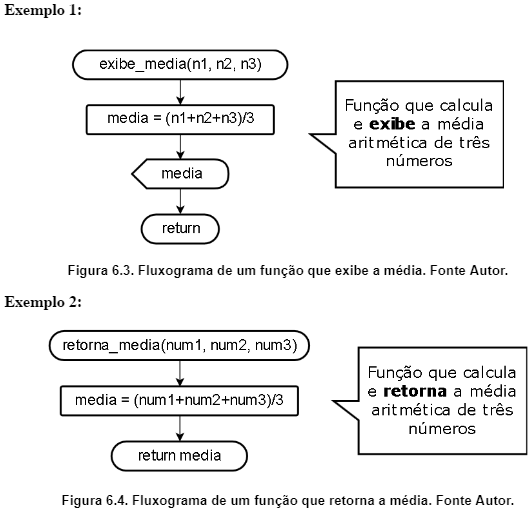
Enquanto a rotina está em execução, o código posterior à linha com o nome da rotina fica em estado de espera, pois não é possível seguir em frente enquanto a rotina não terminar sua tarefa.

Quando uma rotina termina de ser executada, o controle da execução retorna automaticamente para o ponto onde foi chamada e o processamento continua com as instruções sucessoras (DIERBACH, 2012).

As rotinas podem ser pré-definidas da própria linguagem de programação ou construídas pelo programador quando necessário. Existem diversas rotinas prontas em Python, por exemplo, para cálculo de raiz quadrada, para gerar números aleatórios e para exibir mensagens para o usuário.

Usaremos o conceito de função como um tipo de rotina que pode ser criada da seguinte forma:





A primeira linha de definição de uma função é chamada cabeçalho ou assinatura. O cabeçalho da função inicia com um identificador, que é o nome da função (DIERBACH, 2012). No exemplo 1 (Figura 6.3) o identificador da função é exibe\_media, já no exemplo 2 (Figura 6.4) o identificador é retorna\_media.

O nome da função é seguido por um par de parênteses e, entre eles, alguns identificadores chamados de parâmetros formais ou simplesmente parâmetros. No exemplo 1 os parâmetros são n1, n2 e n3, já no exemplo 2 os parâmetros são num1, num2, num3.

Os parâmetros são variáveis locais da função e inicializadas no momento em que a função é chamada. Os parâmetros devem ter um nome que, preferencialmente, faça referência ao dado que será armazenado, assim como variáveis comuns. As funções podem ser construídas com qualquer número de parâmetros, inclusive sem nenhum parâmetro.

Os parâmetros servem, essencialmente, para comunicar algum dado externo à função. Por exemplo, para calcular a média de três números a função precisa saber quais são os três números, por isso os valores são passados como parâmetros para ela.

Após o cabeçalho da função, escreve-se o corpo da função, onde são inseridas as variáveis locais e as demais instruções que a compõem.

Funções são geralmente definidas no início do programa. Contudo, a regra geral é: toda função deve ser definida antes de ser chamada.

**Solicitando a Execução de uma Rotina**

A concepção de uma rotina envolve dois momentos bem diferentes:

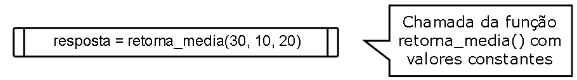
* a criação/declaração e
* a execução da rotina.

Uma rotina é executada quando é chamada, seja no programa principal ou em outra rotina (muitas vezes o programa principal é chamado de rotina-principal).

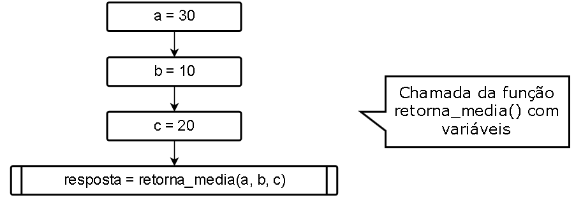
A chamada da rotina é a forma de solicitar sua execução em um determinado passo do algoritmo e, como dito anteriormente, no momento em que é chamada, o código da rotina entra em execução.

Para fazer a chamada de uma rotina devemos especificar seu nome e passar os argumentos necessários (podendo ser valores constantes ou variáveis). Note que os argumentos passados às rotinas são associados aos seus parâmetros.

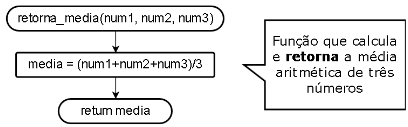
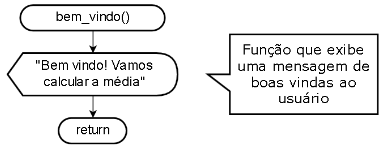
Considerando a função retorna\_media(num1, num2, num3) definida anteriormente, a sua chamada pode ser feita assim:

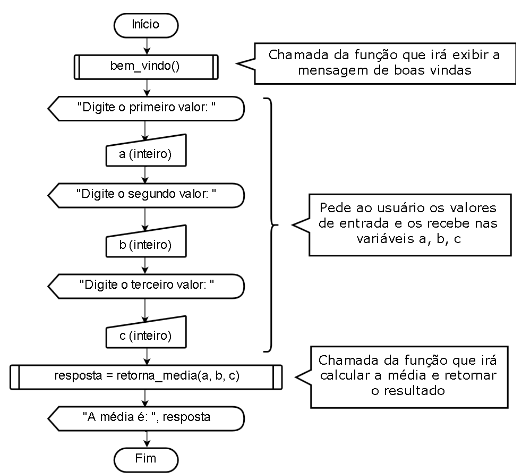


É possível chamar a função retorna\_media(num1, num2, num3) e obter o mesmo resultado da seguinte maneira:



O fluxograma completo que calcula a média aritmética entre 3 números inteiros usando a função retorna\_media(num1, num2, num3), deve ser dividido em duas partes, uma para a definição das funções e outra para o programa principal:





**Retorno/Devolução de valor de funções**

Toda função, ao encerrar, retorna (ou devolve) o controle de execução à instrução seguinte à que foi chamada. Isso nós já vimos, porém, existe outra característica em funções com alta relevância, o retorno (ou devolução) de valor resposta.

Como visto nos exemplos anteriores, foram criadas duas funções: (a) uma chamada exibe\_media e (b) outra chamada retorna\_media. É notável que ambas têm objetivos semelhantes, porém não foram implementadas de forma igual, pois a primeira exibe o valor da média calculada e a outra retorna o valor da média calculada como resposta para a instrução que a chamou.

Portanto, existe uma diferença fundamental entre funções que: (I) retornam um valor como resposta e que (II) não retornam um valor como resposta.

Uma função que não retorna valor, geralmente, se limita às ações ou efeitos colaterais, tal como exibir os dados de saída na tela. Não confunda retornar/devolver valor com exibir algo, esse é um erro comum de iniciantes no uso de funções. Quando um valor é retornado, é possível executar operações sobre ele, como somas, subtrações, etc. Quando um valor é somente exibido em uma função, por exemplo, não é possível armazená-lo em uma variável.

Uma forma clara de se acostumar com essa diferença é associar funções que devolvem valor com perguntas, por exemplo: raiz\_quadrada(n). É fácil perceber que para essa função o que se espera é uma resposta para a pergunta “qual é a raiz quadrada do parâmetro n?”. Note, a resposta não é dada com uma exibição dentro da função, mas sim usando um comando específico, return.

Já as funções sem retorno/devolução de valor resposta podem ser associadas a ordens, por exemplo: exibe\_idade(ano\_nasc, ano\_atual). Note que a própria intuição ao ler o rascunho de cabeçalho da função deixa claro que não se trata de uma pergunta, e sim de uma ordem cuja ação inclusive consta no próprio nome: provavelmente o que se deve fazer é exibir a idade com base nos argumentos passados para a função.

Uma dúvida comum é quando usar cada tipo de função. A resposta direta é: depende. A dependência em âmbito escolar, é essencialmente o enunciado da questão, que provavelmente indicará o formato da função. Porém, em um ambiente profissional, a experiência e/ou a especificação do projeto indicará qual o formato mais adequado. Por isso é importante treinar ambos modelos.

Agora uma revelação: tecnicamente toda função retorna valor. Assim, conceitualmente os pressupostos anteriores continuam válidos (e seguidos rigorosamente por diversas linguagens de programação), porém Python tem funcionamento peculiar.

Existem funções que devolvem valores comuns, quando há um return explicitamente escrito no código e seguido de um valor, e funções que devolvem um valor especial None (em Python) que indica exatamente o que o nome sugere, que a função devolve nada. Contudo, para fins didáticos, consideramos que funções que não executem uma instrução explícita de retorno de valor são funções sem retorno de valor.

**Passagem de Argumentos para Funções**

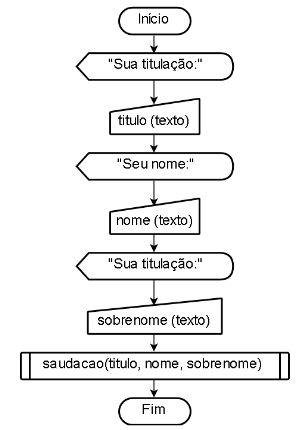
Vimos que funções podem ser definidas contendo parâmetros e que para chamá-las é preciso escrever o nome da função acompanhado dos argumentos correspondentes aos parâmetros.

A correspondência entre os argumentos da chamada e os parâmetros da função é determinada pela ordem em que os argumentos são passados e não pelos seus nomes, como pode-se imaginar de início. Ou seja, por padrão, considera os argumentos como posicionais.

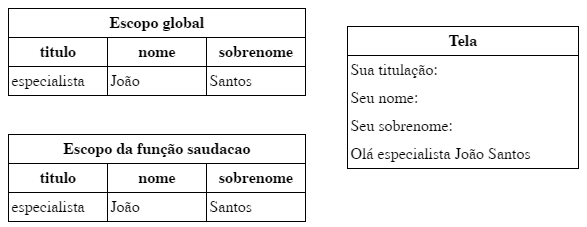
Para exemplificação, considere a Figura 6.13, uma função que recebe como parâmetros um título, um nome e um sobrenome. O objetivo da função é exibir uma frase no formato 'Olá <titulo> <nome> <sobrenome>':



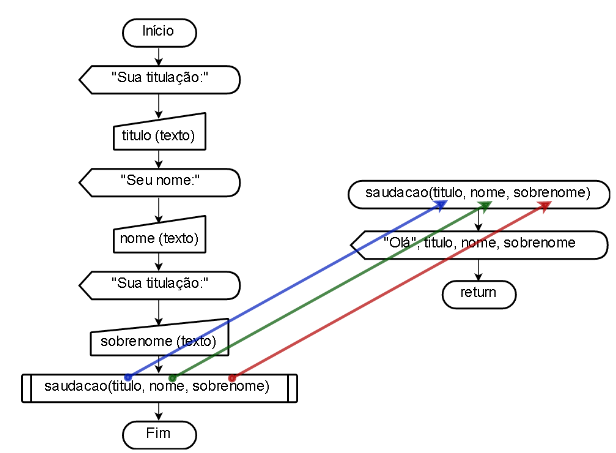
Note que os parâmetros são usados em uma ordem específica que influenciará no resultado da frase exibida. Considere a Figura 6.14 , um fluxograma completo que usa a função saudação definida acima e o seu respectivo teste de mesa com as entradas especialista, João e Santos:



**Teste de Mesa com Funções**



Observe que os valores dos argumentos são copiados para os parâmetros na ordem em que são escritos, ou seja, o primeiro argumento é associado ao primeiro parâmetro, o segundo argumento é associado ao segundo parâmetro e assim sucessivamente, ou seja, um argumento é associado a um parâmetro particular baseado em sua posição na sequência de argumentos que corresponda à mesma posição do parâmetro na sequência de parâmetros.



Portanto não é necessário que os nomes dos argumentos sejam iguais aos dos parâmetros, basta que as posições sejam logicamente equivalentes às dos parâmetros.

**Escopo de variáveis - Global e Local**

Uma variável é chamada de variável local quando é acessível somente dentro de uma única rotina, mais precisamente daquela rotina em que está contida. O seu ciclo de vida, ou seja, o período que a variável existirá, é igual a duração da execução da rotina que a contém.

Quando uma rotina é chamada, ela cria automaticamente as variáveis locais definidas pelos parâmetros formais, atribuindo a elas os valores recebidos através da chamada da função. Conforme a rotina é executada, todo processo em que há a criação de uma nova variável o fará localmente e, ao final da execução da rotina, ou seja, no momento em que ela é encerrada, todas as variáveis locais são automaticamente destruídas.

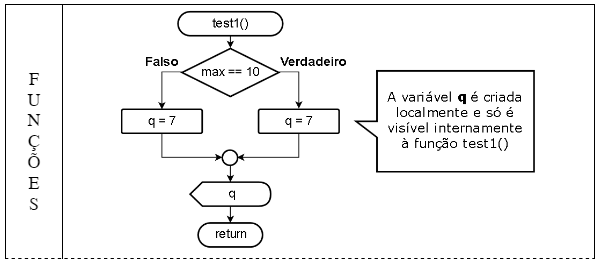
Uma variável global é aquela que é criada no programa principal fora de qualquer função. São ditas terem um escopo global e estarão disponíveis na memória ao longo de toda a execução do programa. Tais variáveis são visíveis a todas as funções a partir do momento que são criadas e só são destruídas quando a execução do programa principal chega ao fim.

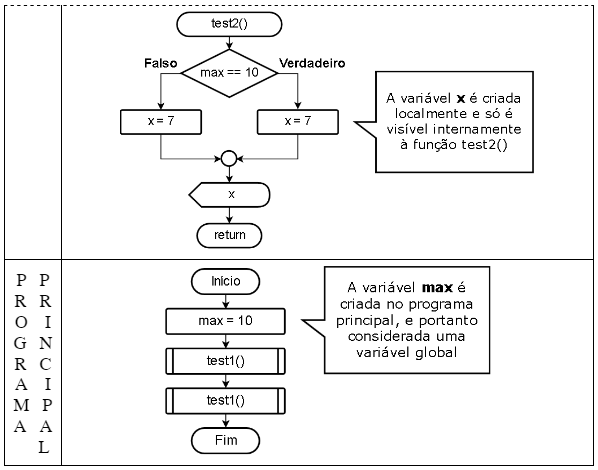
A esse espaço de visibilidade das variáveis dá-se o nome de escopo. Uma forma de entender o significado de escopo é imaginar que cada função tem uma área de trabalho própria, de onde é possível ver a área de trabalho principal mas não a área de trabalho de outras funções.

Todas as funções definidas no escopo global podem acessar e alterar todas as variáveis globais previamente definidas. Por essa razão o uso de variáveis globais é considerado uma má prática de programação em grande parte das situações reais e deve ser evitado.

Para evitar a criação de variáveis globais, uma forma comumente utilizada em diversas linguagens de programação, é a criação de uma função chamada main(), sem parâmetros nem retorno, que irá conter todo o código do programa principal. Para que o programa seja executado então, basta chamarmos a função main().

Veja no exemplo a seguir que a variável max é definida no programa principal, fora do escopo das funções test1() e test2() e, portanto, é considerada global, estando acessível a ambas funções.





Funções - Lógica de Programação. [Funções - Lógica de Programação - AULA 6](https://youtu.be/CvfSxeWcqQA)

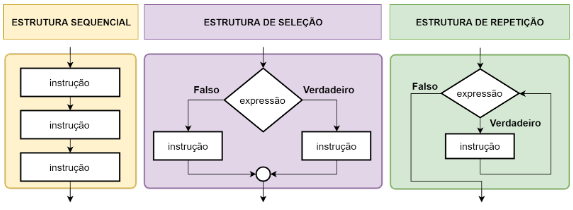
**Parte 7 -** [**Estrutura de Repetição Indefinida e Definida**](https://sites.google.com/faculdadeimpacta.com.br/lo-p7?pli=1&authuser=1) **-** [**Vídeo 1**](https://www.youtubeeducation.com/watch?v=DffXiS0Cx7g) / [**Vídeo 2**](https://www.youtubeeducation.com/watch?v=Uuj2XXWCwb8)

**Resumo**

Neste texto é discutido o conceito de Estrutura de Repetição. Basicamente, a Estrutura de Repetição (também chamada loop ou laço) repete comandos. Podemos classificá-las em dois tipos: Indefinida e Definida. Na Estrutura de Repetição Indefinida, não sabemos o número de vezes que teremos que repeti-los. Na Estrutura de Repetição Definida, tem o objetivo de repetir comandos um determinado número fixo de vezes, ou seja, sabemos o número de vezes que teremos que repeti-los. Abordamos também o uso de laços usados na validação de dados de entrada, laços que utilizam variáveis chamadas Flags e, por fim, as variáveis contadoras usadas em contagens de eventos. Em seguida, abordamos laços que utilizam variáveis chamadas acumuladoras usadas para somar valores e por fim loops infinitos, aqueles que nunca terminam. Ao final é mostrado um exemplo usando fluxograma com sua conversão para Python.

**Estruturas**

Já vimos duas formas de controlar o fluxo de um programa: estrutura de controle sequencial e estrutura de controle de seleção ou condicional.



Vamos iniciar as estruturas de repetição, também conhecidas como iterativas ou de laço (loop) que, junto com as demais estruturas vistas até agora, propiciarão a solução de uma gama muito maior de problemas.

As estruturas de repetição permitem que uma ação seja executada várias vezes sem que tenhamos que executar novamente o programa.

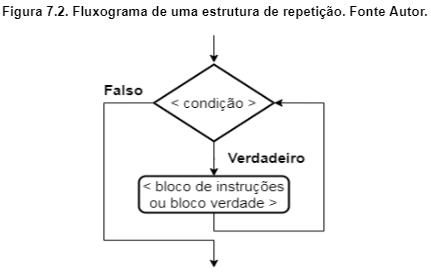
Uma estrutura de controle iterativo, estrutura de repetição, laço de repetição ou simplesmente laço é uma estrutura de controle de fluxo formada por um conjunto de instruções que são executadas um determinado número de vezes ou enquanto uma determinada condição for verdadeira.

**Estruturas de Repetição**

Uma estrutura de repetição, também chamada de loop ou laço, tem o objetivo de repetir comandos. Seu funcionamento depende de uma condição ou expressão lógica. Os comandos são repetidos enquanto a condição for verdadeira. Quando a condição for falsa a repetição termina e o programa segue. Figura 7.2.

Esta estrutura permite executar diversas vezes um bloco de instruções, sempre verificando antes se a <condição> é verdadeira. A condição é uma expressão booleana, semelhante às expressões que usamos nas estruturas de seleção.

A <condição> é primeiro avaliada. Enquanto a <condição> for verdadeira, o <bloco verdade> é executado. Quando a <condição> for falsa, a iteração termina e a execução continua com a instrução após o laço.



Com relação a forma como o laço é montado temos duas possibilidades:

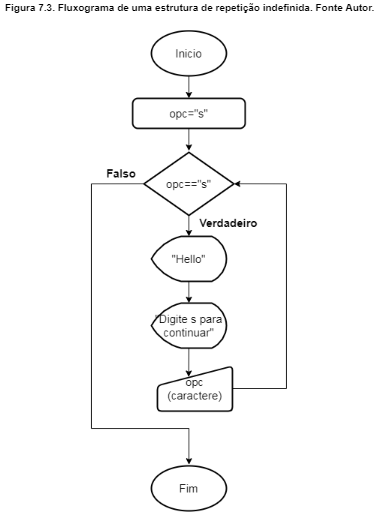
1. Estrutura de Repetição Indefinida: Quando não sabemos quantas vezes serão repetidos os comandos.
2. Estrutura de Repetição Definida: Quando sabemos quantas vezes serão repetidos os comandos.

**Estrutura de Repetição Indefinida**

A Estrutura de Repetição Indefinida ocorre quando não sabemos quantas vezes serão repetidos os comandos. No dia-a-dia temos inúmeros exemplos desse tipo de necessidade. Quando vamos a um caixa eletrônico, por exemplo, a princípio não sabemos exatamente quantas operações vamos fazer. Chegamos no caixa eletrônico, fazemos a nossa operação e ao final nos é perguntado se queremos fazer uma nova operação. Se respondermos que sim, o processo recomeça e efetuamos a nova operação e assim sucessivamente até que resolvemos parar e respondemos não à pergunta sobre a nova operação e o processo finaliza.

Vejamos o exemplo da Figura 7.3 abaixo. Nesse exemplo utilizamos uma variável opc que nos ajuda a controlar o laço. Inicialmente foi armazenado um valor nessa variável “s” de maneira que o teste resulta em verdadeiro. Assim, o bloco verdade é executado (a exibição da palavra “Hello”) e ao final dele o usuário digita um novo valor para essa variável. Caso ele digite “s”, ao retornar ao teste novamente ele resultará em Verdadeiro o que fará executar o bloco verdade novamente. Caso seja digitado qualquer valor diferente de “s” o teste resultará em falso e o laço se encerra.

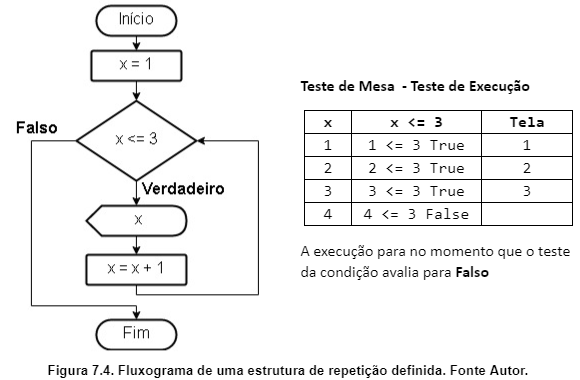
Perceba que nesse exemplo não sabemos quantas vezes o laço será executado. Isso vai depender totalmente do que o usuário digitar na entrada dentro do loop. Devido a esse fato essa montagem é chamada de Laço Indefinido.



**Estrutura de Repetição Definida**

A Estrutura de Repetição Definida ocorre quando sabemos quantas vezes serão repetidos os comandos. Portanto usamos essa construção de laço sempre que sabemos de antemão o número de repetições. Na exibição da tabuada do 5 sabemos que temos que repetir 5 x 1 = 5 e assim sucessivamente por 10 vezes. Quando precisamos coletar as notas dos alunos de uma sala de 60 alunos, também sabemos a quantidade de vezes que precisamos repetir.

Exemplo: Considere o programa que exibe os números inteiros de 1 a 3.



Na figura 7.4, a variável x é inicializada com o valor igual a 1 e a condição é avaliada pela primeira vez. Como a condição x <= 3 é verdadeira, a instrução dentro do laço é executada, exibindo o valor de x e atualizando o valor de x para 2. O controle de execução retorna para o topo do laço na avaliação da condição novamente. Como 2 é menor ou igual a 3, o bloco dentro do laço é executado, exibindo o valor de x e atualizando x para 3. Retoma-se ao topo da estrutura com o teste da condição, como é verdade, o bloco de instruções é executado novamente pela terceira vez. Nesse ponto, a variável x assumiu o valor 4 e a condição x <= 3 resulta em falso (False), terminando assim a repetição do bloco.

É importante observar que, no caso acima descrito, as instruções do laço são executadas 3 vezes, enquanto a condição é avaliada 4 vezes. Isto acontece pois toda vez que a condição é avaliada, o bloco de instruções ou bloco verdade é executado e retorna-se à verificação da condição, até que a condição seja avaliada para Falso.

**Variável Contadora**

Variável contadora é uma variável utilizada para contagem do número de vezes que ocorre um determinado evento.

Recebe um valor inicial (geralmente 0) e é incrementada em algum ponto do algoritmo de um valor constante (geralmente 1). Incrementar uma variável é o mesmo que somar um valor constante a essa variável.

Com o objetivo de entendermos melhor essa variável contadora, vamos considerar a seguinte questão: Desenhar um fluxograma que recebe diversas idades até que seja digitada uma idade negativa. Contar e exibir a quantidade de idades maiores que 65 foram digitadas.



Nesse exemplo a variável que controla nosso laço é a idade. Veja que nesse caso, ao invés de colocarmos um valor inicial para a variável de maneira a entrar no laço, executamos o comando de entrada. Assim, caso o usuário digite um valor de idade negativo o teste lógico resulta em Falso e nem chegamos a entrar no laço. Caso a idade digitada seja maior ou igual a zero, o fluxo segue no caminho do verdadeiro entrando no laço. Dentro do laço é verificada a idade novamente. Agora para ver se consideramos essa idade na contagem das idades maiores que 65. Caso o teste seja verdadeiro (idade>65) efetuamos a contagem (c65=c65+1). Esse comando incrementa de 1 a variável c65 efetuando a contagem. O laço finaliza coletando mais uma idade do usuário e voltando ao teste (idade>=0).

Quando o usuário digitar uma idade negativa o teste do laço resulta em Falso, saímos do laço e executamos o comando seguinte (após o laço) de exibição da quantidade de idades maiores que 65 (variável c65).

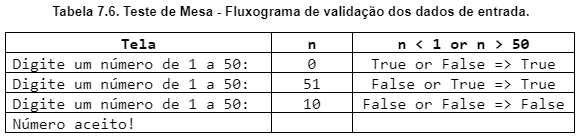
Podemos efetuar várias contagens dentro de um laço, basta que para isso sejam inseridas estruturas de seleção com os testes adequados e que seja criada uma variável contadora para cada evento que desejarmos contar.

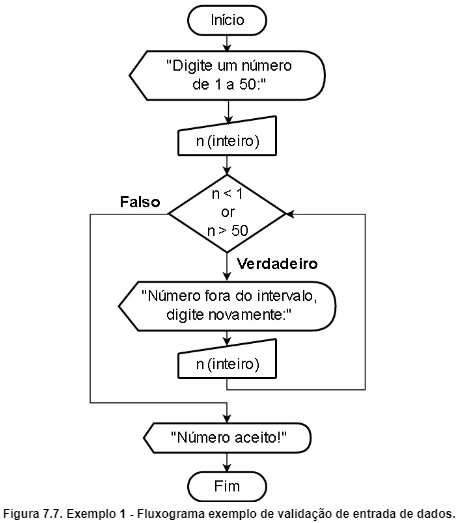
**Validação de Dados de Entrada**

Consiste em verificar se o valor informado pelo usuário está correto ou não. Anteriormente usamos a estrutura condicional para checar informações de entrada e, em caso de erro, encerramos o programa.

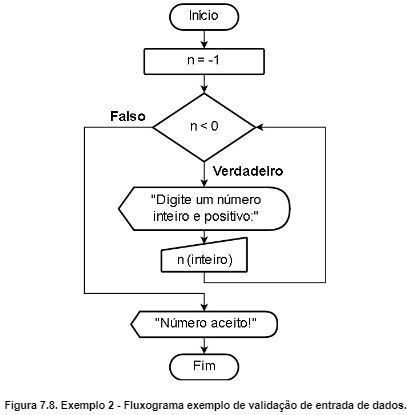
Usando uma estrutura de repetição podemos permitir que o usuário digite o dado enquanto o esteja informando de forma incorreta.

Exemplo 1: Se o usuário precisar digitar um número no intervalo 1 ≥ n ≤ 50, podemos usar a estrutura de repetição que recebe o número e, enquanto este número estiver fora do intervalo permitido, pede novamente que o número seja digitado. Confira na Tabela 7.6.





Exemplo 2 : Desenhe um fluxograma que leia e valide a entrada de um número inteiro e positivo. Usando uma estrutura de repetição podemos permitir que o usuário digite o dado enquanto estiver informando de forma incorreta.

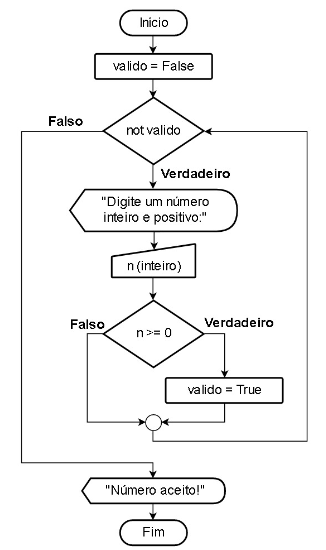


Perceba que nos dois exemplos acima a única forma de finalização do loop ocorre se o usuário digitar um valor válido. O teste é feito de maneira a caso o valor digitado seja incorreto o teste resulta em verdadeiro e assim o fluxo permanece no laço.

**Interrupção do laço. Flag booleana**

O controle do laço pode ser feito por meio do laço infinito de uma única variável booleana, chamada de flag booleana. Dentro do laço verificamos se uma ou mais condições são atendidas através de uma estrutura de seleção e alteramos o valor da flag para que o laço seja encerrado.

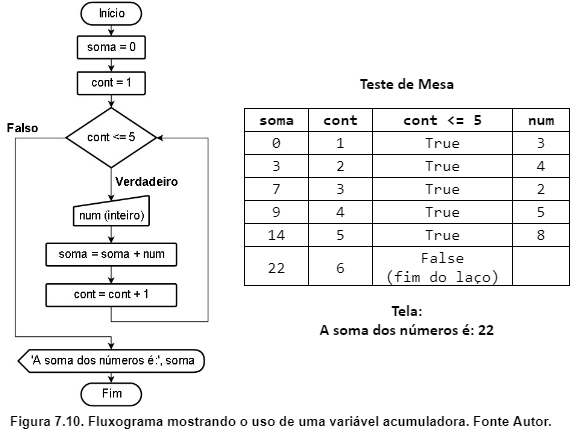
Reescrevendo o exemplo anterior, da entrada de um número inteiro e positivo, com o uso de uma flag booleana, chegamos ao seguinte fluxograma:



É importante notar que o nome da flag e o valor inicial que lhe é atribuído devem ser coerentes com a situação do problema modelado, isto é, iniciar uma flag cujo nome seja VÁLIDO com um valor True não faz sentido, pois teríamos que alterá-la para False justamente no momento em que o número fosse validado. Isso significaria que a flag guardaria o valor True enquanto o número ainda não foi validado (ou seja, é INVÁLIDO) e guardaria o valor False quando o número finalmente fosse VÁLIDO.

**Variável Acumuladora**

Variável acumuladora é uma variável utilizada para controlar a contagem, que recebe um valor inicial (geralmente 0) e é incrementada em algum ponto do algoritmo de um valor variável. Exemplo: Calcular a soma de 5 números digitados pelo usuário.



Perceba que tanto para as variáveis acumuladoras quanto para as contadoras precisamos iniciá-las com zero. No exemplo acima a variável cont é a variável de controle do nosso laço que executa 5 vezes. Para cada execução do laço o usuário digita um valor (num) que é acumulado na nossa variável acumuladora soma. Para acumularmos valores, executamos o comando variável = variável + valor a ser acumulado. No nosso exemplo soma=soma + num. Veja que num é a variável que recebe os diversos valores digitados pelo usuário. Observe no teste de mesa que a variável soma acumula a cada execução do comando soma = soma + num o valor da variável num. Ao final, a variável soma possui a soma de todos os valores digitados pelo usuário 22.

Exemplo 3 - Desenhar um fluxograma que recebe 30 temperaturas de uma cidade do mês de setembro. Exibir a soma dessas temperaturas. Usar uma função para obter a soma.



Para esse exemplo vamos usar uma função chamada somatemp. O objetivo da nossa função será coletar as temperaturas e calcular a soma. Perceba que a exibição do resultado não está na função. Para usarmos essa função vamos criar um fluxograma que invoca (ou chama) a função, obtém o resultado da função através da variável S e após isso exibe a mensagem com o resultado, Observemos que na função No exemplo acima a variável cont é a variável de controle do nosso laço que executa 30 vezes (uma para cada dia do mês) pois o valor dessa variável inicia com 1 no teste, temos cont <= 30 (ou seja quando atingir 31 para o laço), e a cada execução essa variável é incrementada de 1 (cont = cont + 1) - o chamado passo. Portanto esse loop vai ser executado 30 vezes. Colocamos dentro do laço uma entrada de dados na variável temp (temperatura). Finalizando criamos a variável soma. Essa é nossa variável acumuladora das temperaturas. Para cada execução do laço o usuário digita um valor (temp) que é acumulado na nossa variável acumuladora soma. Usando o comando soma = soma + temp. Veja que temp é a variável que recebe as 30 temperaturas do mês de setembro (uma em cada execução do loop), valores esses digitados pelo usuário. Ao final, a variável soma possui a soma de todas as temperaturas digitadas pelo usuário.Vamos usar o conteúdo dessa variável como retorno da função comando return soma.

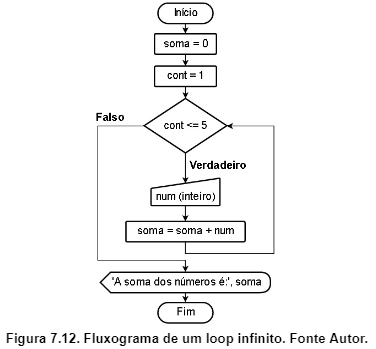
O fluxograma que invoca a nossa função (lado esquerdo da Figura 7.11) possui dois comandos. O primeiro comando invoca a função e obtém o resultado da função (devido ao retorno da função). Como a nossa função retorna da soma das temperaturas, a variável S do nosso fluxograma receberá esse valor. Por fim, é usado o comando de exibição para mostrar a mensagem com o valor da soma das temperaturas.

**Laço Infinito**

Laço ou loop infinito é uma estrutura de controle iterativo que nunca termina (ou eventualmente termina com um erro de sistema).

Exemplo: Calcular e exibir a soma de 5 números digitados pelo usuário.

Observe que no fluxograma abaixo (Figura 7.12) o teste lógico sempre resultará em Verdadeiro porque a variável de controle cont recebe inicialmente o valor 1 e esse valor não é alterado dentro do laço (não existe o comando do passo cont = cont + 1). Dessa forma o teste que será feito cont <= 5 será sempre 1 <= 5:Verdadeiro (o valor da variável cont será sempre 1). Sendo assim esse laço nunca termina, resultando em um loop infinito.



Repetição uso de flag. [Aula 8 repetição parte 3 uso de flag](https://youtu.be/d5g3HryWRtM)

Enquanto\_faça - Utilizando FLAG [Vídeo Aula (Exercício 3 comando enquanto ... faça utilizando FLAG)](https://youtu.be/_b8jM1Xiuxc)

Repetição - Fluxograma. [Aula 10 repetição Fluxograma](https://youtu.be/zOOv-G5L-us)

**Parte 8 -** [**Vetores**](https://sites.google.com/faculdadeimpacta.com.br/lo-p8?pli=1&authuser=1) **-** [**Vídeo**](https://youtu.be/J7gaPoTUr8s)

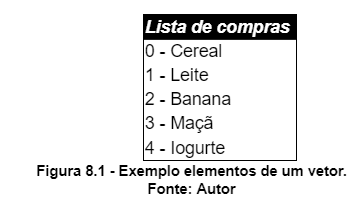
**Resumo**

Usamos vetores no nosso dia-a-dia. Talvez o nome não seja bem esse. Conhecemos mais como listas. Neste texto é discutido o conceito de vetores, seus componentes e a sua forma de utilização. A grande vantagem dos vetores reside justamente na sua capacidade de armazenar uma grande quantidade de informação na memória. É esse justamente o seu objetivo como ferramenta: sempre que tivermos a necessidade de armazenar uma grande quantidade de informação vamos usar esse recurso. Como exemplificação mostra-se uma implementação de um exemplo usando fluxogramas e um segundo exemplo usando fluxogramas juntamente com uma simulação mostrando como percorrer um vetor. Ao final é mostrada a conversão de um fluxograma usando vetores e funções para Python.

**O QUE É UM VETOR?**

Um vetor é uma estrutura de dados linear que mantém a posição dos seus elementos em uma ordem linear. Ou seja, tem o primeiro elemento, o segundo elemento e assim por diante.

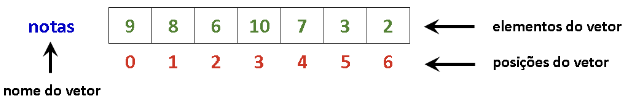
Abaixo na Figura 8.1 nós temos um exemplo de uma lista de compras. Note que o primeiro elemento da lista é o Cereal, o segundo é o Leite, o terceiro é a Banana etc.



A ordem em que os elementos aparecem na lista pode ser a ordem em que o autor planeja pegar os itens no mercado, ou pode ser simplesmente a ordem em que ele anotou os itens à medida que foi se lembrando.

**VETORES EM PROGRAMAÇÃO**

Vetores são variáveis com capacidade de armazenar várias informações. Cada informação é acessada ou armazenada através de sua posição.



No exemplo acima temos que o nome da variável (do tipo vetor) é notas. Esse vetor possui 7 posições. As posições começam de 0 até 6. Os elementos do vetor são os valores que o vetor armazena. Assim:

* O valor do vetor notas na posição 0 (zero) é 9.
* O valor do vetor notas na posição 3 (três) é 10.

**Criando um vetor**

Embora a criação de vetores não seja definida para fluxogramas, vamos usar como padrão a criação de listas em Python 3. Podemos criar um vetor usando um dos comandos a seguir:

notas = 7 \* [0]

ou

notas = [0, 0, 0, 0, 0, 0, 0]

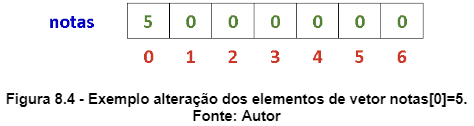
Isso cria a variável notas e atribui a ela um vetor de 7 posições, todas com o conteúdo 0 (zero) (Figura 8.3).



**Atualizando o conteúdo de uma posição do vetor**

Para alterar o conteúdo de uma posição do vetor iremos usar o operador de atribuição “=”, indicando a posição que receberá o novo valor dentro de colchetes “[” e “]”:

notas[0] = 5

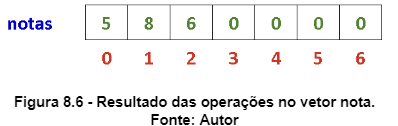


Podemos também executar operações com o conteúdo do vetor em expressões:

notas[2] = notas[1] - notas[0] + 3

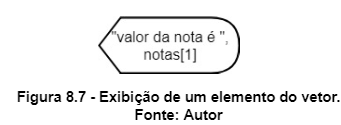
notas[2] = 8 - 5 + 3

notas[2] = 6



A posição 2 do vetor notas recebe o valor 6. Pois notas[1] vale 8 e notas[0] vale 5.

**Exibindo valores de um vetor**

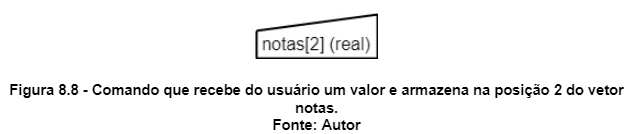


Podemos exibir um elemento de um vetor indicando a posição a ser exibida (Figura 8.7)

Utilizando o vetor acima o resultado desse comando apresentará na tela: **valor da nota é 8**

**Atualizando o vetor com dados digitados pelo usuário**

Podemos executar uma entrada de dados diretamente no vetor, bastando para isso indicar a posição do vetor onde o dado será armazenado:

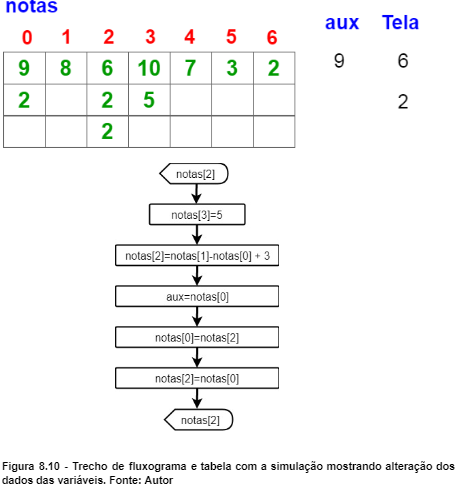


Utilizando o vetor acima, caso o usuário digite o valor 10, o resultado desse comando deixará o nosso vetor como abaixo:



Ou seja, o valor da posição 2 que era 6 foi substituído por 10 (valor digitado pelo usuário)

Exemplo 1 - Nesse exemplo mostramos a simulação de um trecho de fluxograma. A medida em que os comandos são executados as alterações nas posições do vetor notas, na variável aux e na tela são usadas para preencher a tabela abaixo:



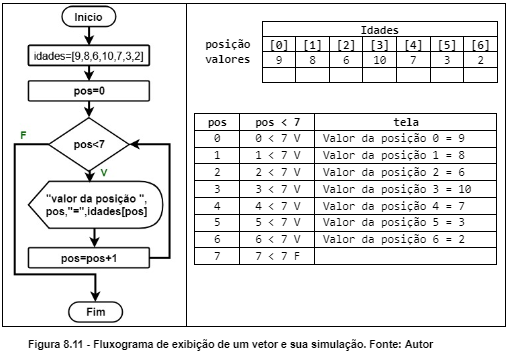
**Executando operações em vetores**

Alteração: Para alterarmos um vetor temos que alterar cada um dos valores em cada posição. Não é possível alterar todos os valores de uma vez. Cada posição do vetor funciona como se fosse uma variável independente e dizemos que vamos percorrer o vetor.

Percorrer os elementos de um vetor é acessar os elementos um a um, começando pelo primeiro e terminando no último elemento do vetor, posição a posição.

Esse percurso pelos elementos de uma sequência pode ser feito através de um laço de repetição.

Exibição: Veja abaixo um exemplo para a exibição dos valores de um vetor:

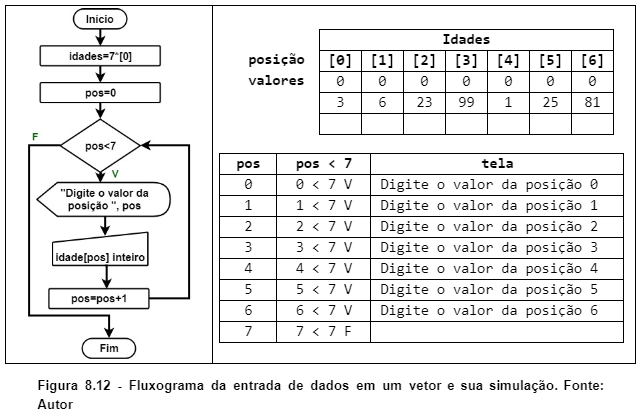


A execução deste bloco irá fazer com que a variável pos (que é a variável de controle do laço) assuma o valor da primeira posição do vetor (no caso 0 - zero) e, a cada execução do bloco de instruções do laço, assuma o valor da posição seguinte até que todas as posições sejam percorridas. Ao final, pos irá assumir o valor 7 e o laço será encerrado, pois o vetor não possui uma posição 7.

O comando de exibição na tela será repetido 7 vezes e para cada execução o valor da variável pos será diferente, começando em 0 e terminando em 6, e será impresso na tela o valor correspondente a cada posição. Dessa forma, quando pos vale 0 (zero) o valor exibido será aquele correspondente à primeira posição (posição zero) do vetor idades, e assim sucessivamente até a última posição.

Atribuição: Veja abaixo um exemplo para a atribuição de valores a um vetor a partir da leitura de tais valores do teclado:

Vamos supor que o usuário digite os seguintes valores: 3, 6, 23, 99, 1, 25, 81



De uma maneira análoga a exibição, a execução deste bloco irá fazer com que a variável pos (que é a variável de controle do laço) assuma, a cada execução do bloco de instruções do laço, o valor da posição seguinte até que todas as posições sejam percorridas.

O comando será repetido 7 vezes e, para cada execução, o valor da variável pos será diferente, começando em 0 e terminando em 6. Dessa forma quando pos vale 0 o valor digitado será armazenado na posição 0 do vetor idade e assim sucessivamente até a posição 6, pois ao receber o valor 7, o laço é encerrado pois não é possível acessar a posição 7 do vetor (ela não existe).

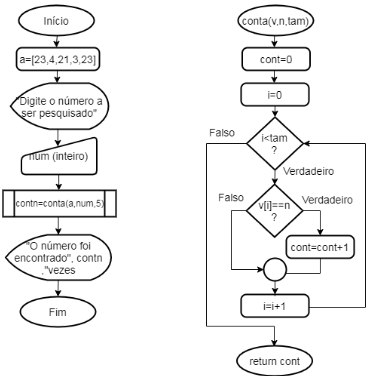
Exemplo 2: Desenhar um fluxograma de uma função que recebe como parâmetro um vetor e um valor. A função deve retornar a quantidade de vezes que o valor aparece no vetor. Desenhar também um fluxograma que crie o vetor v=[23,4,21,3,23] , solicite um valor para o usuário e chame a função e exiba o valor retornado pela função.

Quando trabalhamos com funções é importante deixar bem claro qual parte do código fica na função e qual não fica. Nesse caso, vamos criar uma função que tem o objetivo de contar quantas vezes um determinado valor aparece em um vetor. Vamos passar por parâmetro o vetor V e o valor n. A grande vantagem de se fazer isso é que essa mesma função pode ser usada para outros valores e outros valores. Basta para isso passarmos essas informações como parâmetros da função. A reusabilidade é uma característica importante quando criamos uma função. O fato da função poder ser reutilizada em várias outras situações é um predicado importante para as funções. Pensando nisso vamos utilizar mais um parâmetro tam. Ele será usado para passarmos para a função a informação da quantidade de posições do vetor que estamos considerando para a nossa procura. Apesar de um vetor ter 100 posições, por exemplo, não precisamos utilizar todas as suas posições, podemos utilizar apenas 10 posições das 100 disponíveis.

Vamos primeiramente entender o funcionamento da nossa função conta. Nessa função vamos usar uma variável contadora cont para armazenar a quantidade de valores n encontrados no vetor V. Como todas as variáveis contadoras fazemos inicialmente sua inicialização em 0 (cont=0). Em seguida implementamos um loop utilizando uma variável de controle i de maneira que inicie valendo 0 - que é a primeira posição do vetor, até n-1 (usando i<tam) - que é a última posição do vetor.

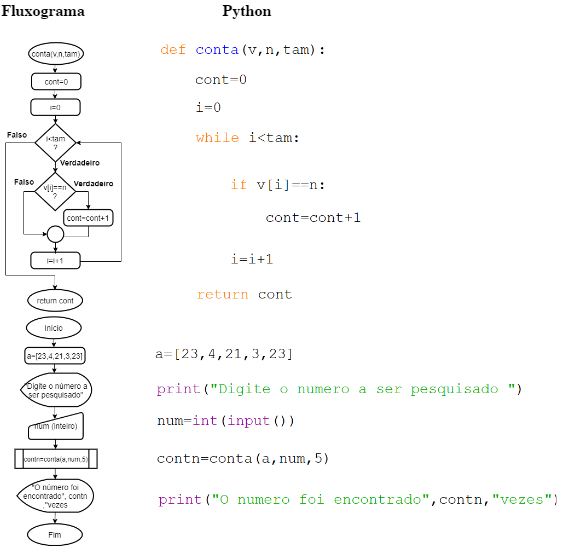
Dentro desse loop usamos uma estrutura de seleção simples que tem a função de verificar se o valor da posição do vetor é igual ao valor procurado V[i]==n para cada uma das posições do vetor. Temos que lembrar que i é uma variável que na primeira iteração (passo) do loop vale 0 depois 1 e assim sucessivamente até atingir seu valor máximo (tam). Dessa forma, na primeira iteração fazemos o teste V[0]==n, pois i vale 0, depois V[1]==n e assim por diante. Assim, em cada iteração do loop testamos uma posição diferente. Caso o teste seja verdadeiro, incrementamos a variável cont através do comando cont=cont+1 indicando e contabilizando que encontramos o valor na posição i. Assim ao final do loop a variável cont armazenará a quantidade de vezes que o valor foi encontrado no vetor. Por fim, o comando return cont devolve o valor cont ao comando que invocou a função no programa.

Perceba que na função não criamos o vetor e nem pedimos nenhuma informação ao usuário. Veja que isso é feito no programa. Observe que no fluxograma do programa criamos o vetor a=[23,4,21,3,23] (que será usado na função) e solicitamos a digitação do valor a ser encontrado num. Em seguida a função conta é invocada e são passados por parâmetro o vetor, o valor a ser procurado e o tamanho do vetor contn=conta(a,num,5). Perceba que nessa instrução a variável contn recebe o valor retornado pela função conta (a frequência de vezes que o valor foi encontrado no vetor). Finalmente a variável contn é exibida juntamente com uma mensagem.



**Conversão de Fluxograma para Python**

Abaixo podemos ver o exemplo 2 acima implementado em Python:



Vetores - Definição. Disponível em: <https://youtu.be/0ZNWX9ho69c?si=allFq2Bqlk-tVrQY>

**Parte 9 -** [**Passagem de Parâmetros por Valores e por Referência**](https://sites.google.com/faculdadeimpacta.com.br/lo-p9?pli=1&authuser=1) **-** [**Vídeo**](https://youtu.be/Tj61Bl-7IUE)

**Resumo**

Sabemos que os parâmetros de uma função tem o importante papel de fornecer os insumos (matéria-prima) da função. Utilizando corretamente os parâmetros de uma função podemos deixá-la muito mais reutilizável. Sabemos também que as funções conseguem alterar apenas as variáveis do seu escopo, as variáveis locais a função. Vamos aprender neste capítulo as duas formas que a função tem de passar parâmetros para uma função: por valor e por referência. Vamos ver que na passagem por valor a variável utilizada como parâmetro não consegue ser alterada pela função mas na passagem por referência a função consegue alterar essa variável.

**Introdução**

A utilização de parâmetros em funções é um mecanismo muito eficiente para permitir a reutilização da função. Esses parâmetros são considerados as matérias-primas das funções, seus insumos. Representam os dados necessários para que a função consiga cumprir seu objetivo. Sabemos também que os parâmetros e as variáveis criadas na função são variáveis locais e portanto seus valores não conseguem ser acessados externamente às funções. Seria possível então criarmos uma função que teria como objetivo o preenchimento de uma variável como por exemplo um vetor? A seguir vamos entender mais detalhadamente como isso é possível em algumas situações.

**Passagem de Parâmetros para funções**

Os parâmetros carregam as informações para a função. Existem duas formas de passarmos essas informações para as funções:

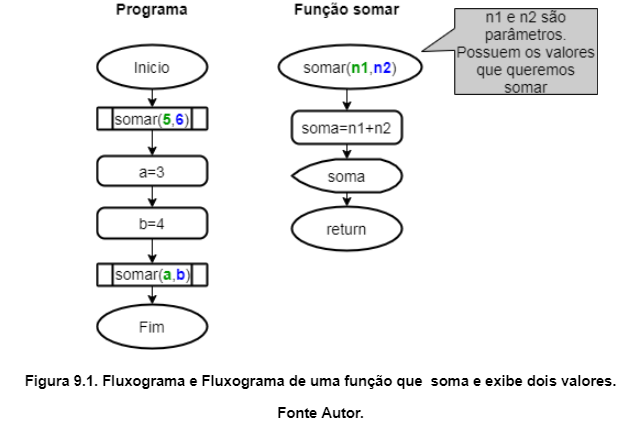
**por Valor :** O valor passado como parâmetro é copiado para a função

**por Referência:** Não é criada uma cópia. Usa-se a própria variável do parâmetro

**Passagem por valor**

Esse é o tipo mais comum utilizado. Nesse caso os a função recebe os valores passados na chamada da função. Observemos o exemplo abaixo:

**Exemplo 1** - Criar uma função que recebe como parâmetro 2 números e exibe a soma desses números.



No Fluxograma acima temos a direita o fluxograma da função, Perceba que temos dois parâmetros n1 e n2 utilizados para receber os valores a serem somados. À esquerda temos um fluxograma que invoca a função soma duas vezes. Na primeira vez usamos o comando soma(5,6) . Quando executamos esse comando, a função soma é chamada e usamos como parâmetros os valores 5 e 6. Perceba que na invocação da função não estamos usando variáveis e sim números. Nesse caso o que ocorre quando a função é chamada é que o número 5 é copiado para a variável local n1 e o número 6 é copiado para a variável local e também parâmetro n2. O primeiro valor 5 é copiado para o primeiro parâmetro n1 e o segundo valor 6 é copiado para o segundo parâmetro n2. Chamamos isso de passagem posicional pois dependendo da posição do valor passado ele será copiado para um parâmetro. O primeiro valor para o primeiro parâmetro e assim por diante. A partir daí os comandos da função são executados considerando-se esses valores para as variáveis n1 e n2 até a exibição e o comando return.

Devemos lembrar que a execução do comando return numa função finaliza a função naquele ponto. Caso existam outros comandos após o comando return eles não serão executados. Outra coisa que acontece após a execução do comando return é que todas as variáveis locais criadas na função são desalocadas da memória, ou seja, elas não existem mais. Portanto quando saímos da função todas aquelas informações usadas na função, os parâmetros e os cálculos são perdidos. Dizemos que essas variáveis são efêmeras e seu tempo de vida se resume ao tempo que a função demora para ser executada. A única coisa que podemos aproveitar de uma função é o valor que ela retorna, o valor usado juntamente com o comando return. Mesmo assim, esse valor que pode ser retornado pelo comando return não é obrigatório. Que é o caso da nossa função soma. Ela não retorna nada. Ela exibe o valor da soma calculado na função. Perceba que nessa função usamos apenas o comando return, sem nenhum valor na frente do comando.

Seguindo o fluxograma, na segunda chamada da função soma perceba que usamos o comando soma(a,b). Nesse caso, os valores que serão passados como parâmetro serão 3 e 4 que são os valores das variáveis a e b respectivamente. Assim, na invocação da função o valor da variável a (3) será copiado para a variável n1 da função e o valor da variável b (4) será copiado para a variável n2 da função soma. Observe o Quadro 9.1 na linha segunda chamada. A partir daí ocorre a execução da função de maneira semelhante a primeira invocação da função, a única diferença ocorre em relação aos valores dos parâmetros, 5 e 6 na primeira execução e 3 e 4 na segunda. Note que apesar de termos usado números como parâmetro na primeira chamada e variáveis na segunda chamada, o funcionamento da função foi semelhante. Isso ocorre porque, sob o ponto de vista da função, o que ela recebe são valores e não variáveis. O que é passado para a função como parâmetro são os valores das variáveis ou valores que passamos diretamente nos parâmetros como na primeira chamada desse exemplo. Esse é o motivo pelo qual esse tipo de passagem de parâmetro é chamado de passagem por valor. É o tipo mais comum de passagem de parâmetros.



**Passagem por referência**

Nesse tipo de passagem de parâmetro conseguimos alterar o conteúdo da variável usada na invocação da função. Nesse caso o que é passado para a função é o endereço de memória da variável, não o valor da variável. Observemos o exemplo 2 abaixo:

**Exemplo 2** - Criar uma função que recebe um vetor como parâmetro e multiplica seus elementos por 5.

Primeiramente perceba que o objetivo dessa função é alterar o conteúdo de um vetor passado como parâmetro. Assim criamos o vetor a no fluxograma do programa a=[1,2,3] e passamos esse vetor juntamente com 2 outros parâmetros: n - o tamanho do vetor e m o valor que vamos multiplicar cada elemento do vetor. Assim se invocarmos a função passando m sendo 5 esperamos que ao final da execução o vetor a seja alterado para [5,10,15].

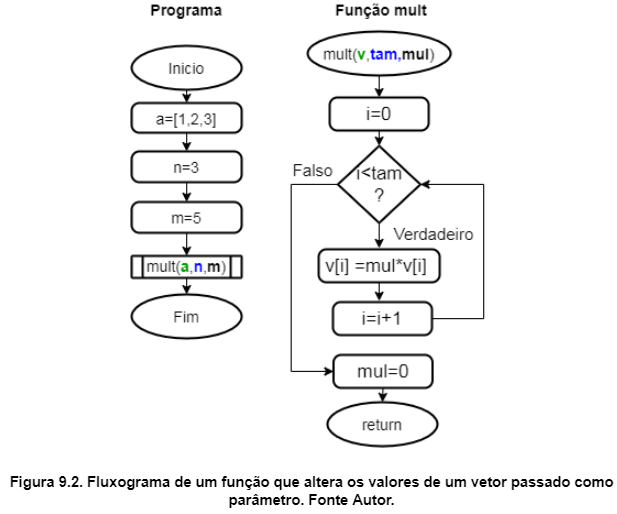
Podemos observar na função mult que temos três parâmetros (variáveis locais): v,tam e mult, onde v é o vetor que queremos alterar, tam é o tamanho do vetor e mult é o valor que iremos usar para multiplicar o conteúdo do vetor v. Perceba que a multiplicação do vetor é feita através de um loop que vai alterar o conteúdo de cada uma das posições do vetor através do comando v[i]=mult\*v[i]. Quando i vale 0 o conteúdo da posição 0 do vetor v é alterada, depois o mesmo ocorre para 1 e assim por diante.

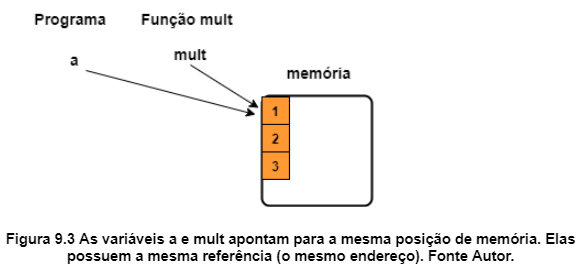
Vimos na passagem por valor que os parâmetros de uma função são variáveis locais que são destruídas logo após o término da execução da função. Se é assim, o vetor v também é destruído quando saímos da função. Então como ocorre a alteração do vetor a?

Em Python assim como em várias outras linguagens quando passamos vetores e alguns outros tipos de variáveis como parâmetro a passagem ocorre de uma forma um pouco diferente.

Vamos entender melhor como ocorre. No nosso exemplo, quando passamos o vetor a como parâmetro o que ocorre é o parâmetro v da função não recebe os valores do vetor a. Ele funciona como um apelido do vetor a. Ou seja, o vetor v recebe a referência de a e não seu conteúdo como ocorre na passagem por valor. Dessa forma, tanto v quanto a estarão acessando o mesmo local na memória onde estão as informações do vetor a (Figura 9.3). Então qualquer alteração em v na verdade estará alterando o vetor a (ambas acessam as mesmas posições na memória). Qualquer alteração feita no parâmetro correspondente afetará o vetor original.

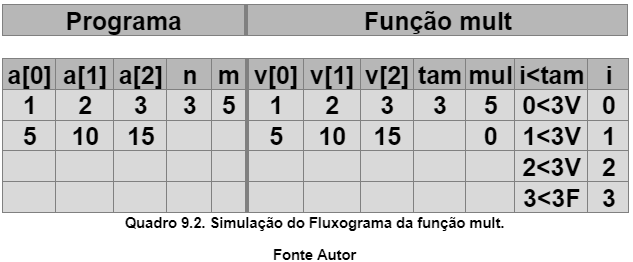
Outra observação que podemos fazer é que numa mesma chamada podemos ter tipos de passagem diferentes. Assim, no nosso exemplo, a variável a é passada por referência e as outras variáveis n e m são passadas por valor.





No quadro 9.2 podemos observar 2 coisas:

1. Perceba que as alterações na variável v da função mult refletem na variável a do programa.
2. As alterações da variável mul da função mult não refletem na variável m do programa



Em resumo, em Python uma variável dos tipos primitivos (int, float, str) sempre é passada por valor. Não conseguimos em Python passar essas variáveis por referência.

Existem alguns tipos de variáveis como listas, tuplas, dicionários ou objetos (Classes) que são passados sempre por referência mesmo sem nenhuma indicação. Em Python, variáveis desses tipos não conseguimos passar por valor.

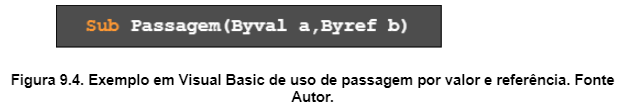
Ou seja, em Python o tipo de variável define como será a sua passagem por valor ou referência.

**Exemplos de outras linguagens**

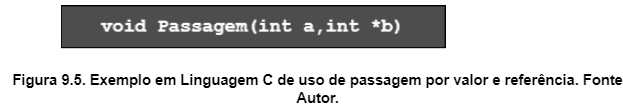
Os exemplos que vimos acima são de códigos em Python. O tipo da variável define o tipo de passagem. O mesmo acontece com algumas linguagens como Java. Em outras linguagens podemos escolher como será o tipo da passagem. O tipo de passagem fica explícito.

Seguem alguns exemplos:

**Visual Basic** - Veja abaixo que os parâmetros passados por valor são precedidos por Byval e por referência por Byref. Conseguimos passar tipos primitivos (inteiro, real) por referência ou valor.Vetores são passados sempre por referência.



**Linguagem C -** Veja abaixo que os parâmetros passados por valor não possuem nenhuma indicação e por referência indicamos por \*. Conseguimos passar tipos primitivos por referência ou valor. Vetores são passados sempre por referência.

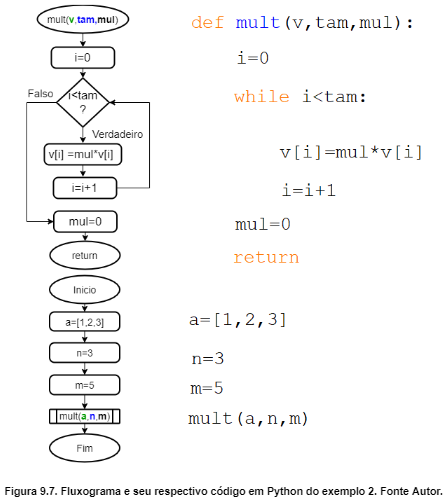


**Pascal -** Veja abaixo que os parâmetros passados por valor não possuem nenhuma indicação e por referência indicamos por var. Conseguimos passar tipos primitivos por referência ou valor. Vetores são passados sempre por referência.



**Comparativo entre fluxograma e Python**

Abaixo podemos identificar um fluxograma do Exemplo 2 e sua conversão para Python.



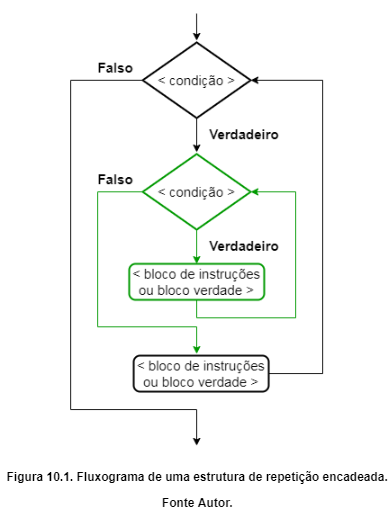
**Parte 10 -** [**Estrutura de Repetição Encadeada**](https://sites.google.com/faculdadeimpacta.com.br/lo-p10?pli=1&authuser=1) **-** [**Vídeo**](https://youtu.be/sfts0dWQwUw)

**Resumo**

Neste texto é discutido o conceito de estrutura de repetição encadeada ou aninhada. Nessa construção um laço é inserido dentro de outro laço. Forma-se assim uma repetição de uma repetição. Como exemplificação mostra-se uma implementação de um exemplo usando fluxogramas, um segundo exemplo usando fluxogramas juntamente com uma simulação. Ao final é mostrada a conversão do fluxograma para Python.

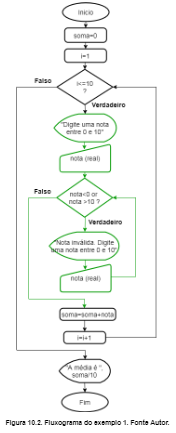
**Estruturas de Repetição Encadeada ou Aninhada**

Na Estrutura de Repetição Encadeada um laço é inserido dentro de um outro laço. Observando a Figura 10.1 percebemos que existe um laço em verde completamente inserido em outro laço. O que ocorre neste caso é que para cada repetição do laço externo (para cada passo do laço externo) o laço interno (em verde) é executado por completo (todos os seus passos).



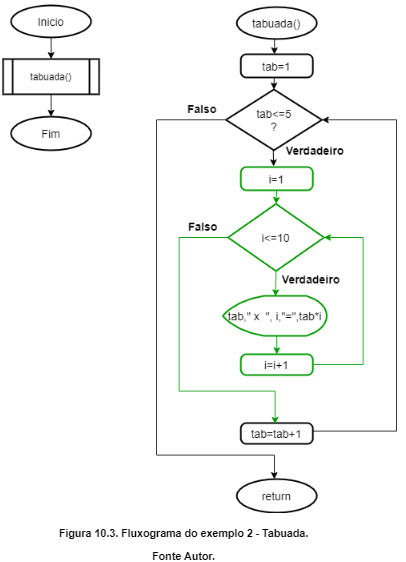
Veja que não fizemos nenhuma consideração em relação ao tipo de laço (definido ou indefinido). Essa construção pode ser usada para qualquer combinação desses tipos de laços. Vejamos o exemplo abaixo:

Exemplo 1: Desenhe um fluxograma que receba 10 notas de uma sala e exiba a média dessas notas. OBS.: A entrada da nota deve ser validada por um loop.



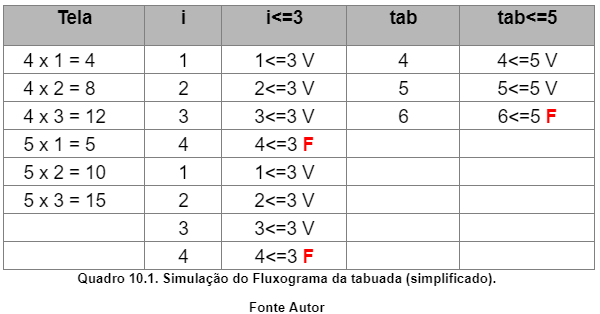
Perceba que nesse exemplo temos o loop da validação em verde (loop indefinido), dentro de um loop definido (que executa 10 vezes).

Exemplo 2: Desenhe um fluxograma que exiba as tabuadas do 1 até 5.



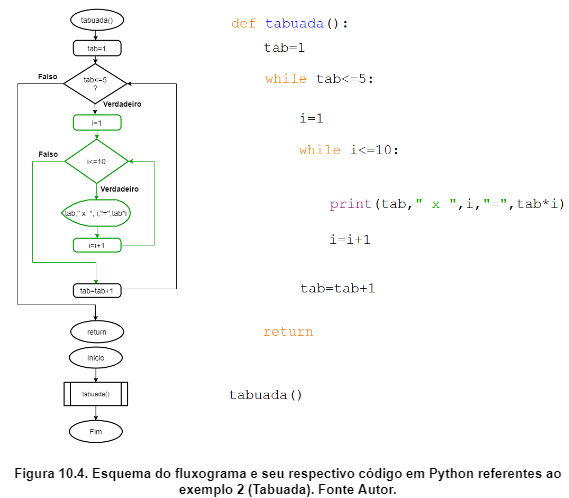
Criamos uma função chamada tabuada. Essa função tem o objetivo de exibir as tabuadas do 1 ao 5. Não utilizaremos parâmetros nessa função. Perceba que o loop em verde executa a exibição de uma única tabuada. Como queremos exibir várias tabuadas (do 1 ao 5) colocaremos esse loop dentro de outro loop executado 5 vezes, uma vez para a tabuada do 1, e até a tabuada do 5. Para cada execução do loop externo é exibida uma tabuada inteira (loop verde). O valor da variável tab define qual a tabuada será exibida. Foi criado também um fluxograma que chama a função tabuada. Veja que como a função tabuada não utiliza parâmetros e também não retorna nenhum valor (ele exibe na tela a tabuada) não foi criada nenhuma variável no programa. É apenas invocada a função tabuada.

Segue uma simulação simplificada (Tabuadas do 4 ao 5 e ao invés de ir do 1 ao 10 para cada tabuada a simulação vai do 1 ao 3).



**Comparativo entre fluxograma e Python**

Abaixo podemos identificar um fluxograma do Exemplo 2 e sua conversão para Python. Na Figura 10.5 pode ser observado um trecho da saída do programa em Python.



**Parte 11 -** [**Busca e Ordenação**](https://sites.google.com/faculdadeimpacta.com.br/lo-p11?pli=1&authuser=1) **-** [**Vídeo**](https://youtu.be/3uRsS6SUqB4)

**Resumo**

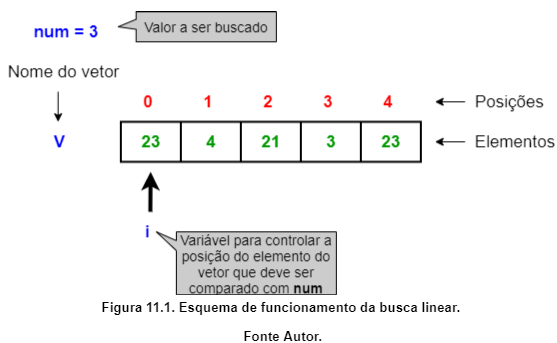
Neste capítulo vamos abordar algoritmos um pouco mais complexos. Primeiramente vamos discutir dois tipos de busca: a busca linear e uma busca otimizada que é a busca binária. Em seguida vamos discutir um algoritmo de ordenação pouco eficiente mas de fácil implementação: o Bubble Sort. Ao final, mostramos a conversão do Bubble Sort para Python.

**Busca Linear**

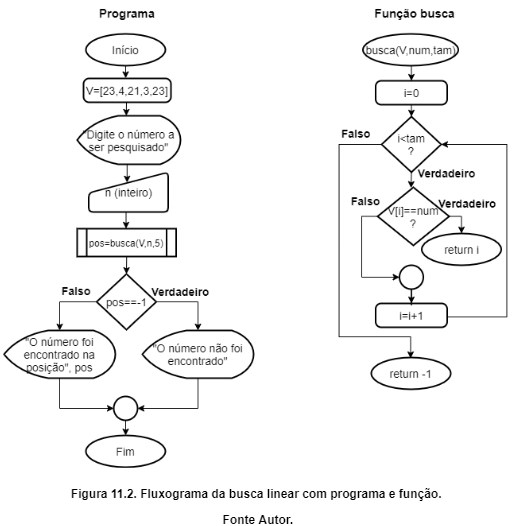
É muito comum no desenvolvimento de software nos defrontarmos com a necessidade de buscar, pesquisar. Basta entrar em qualquer site que sempre iremos encontrar o desenho de uma lupa. Sem dúvida a pesquisa nos poupa de um gasto enorme de tempo.

A busca linear é um algoritmo básico de busca. Ela é feita a partir de um local onde as informações (normalmente uma grande quantidade) é armazenada. Vamos usar como local de armazenamento vetores. O algoritmo consiste em verificarmos todos os elementos até encontrarmos o que procuramos, ou até verificarmos todos os elementos para constatarmos que ele não foi encontrado. Vejamos o exemplo 1 abaixo:

**Exemplo 1** - Criar uma função que receba como parâmetro um vetor com números e um número a ser procurado. A função deve retornar a posição do vetor onde o número foi encontrado ou -1 caso não encontre o número.



Basicamente o algoritmo usa uma variável auxiliar i para ajudar a percorrer todas as posições do vetor (Figura 11.1) Ou seja haverá uma comparação para cada posição do vetor. Usaremos um loop para isso e uma estrutura de seleção para fazer as comparações. No exemplo da figura 11.1 estamos procurando o número 3 no vetor V. A busca inicia-se pela posição 0 (elemento 23) e vai percorrendo o vetor (através do conteúdo da variável i ) até encontrar o elemento 3 na posição 3 do vetor. Uma vez encontrado o valor que buscamos não precisamos continuar procurando. Podemos encerrar a busca e retornar o valor 3 (a posição onde o elemento 3 foi encontrado).



Na Figura 11.2 podemos observar o fluxograma da função e do programa que chama a função. No programa devemos criar o vetor V com os elementos da base de valores em que será feita a busca e obter do usuário o valor n que iremos buscar no vetor. Observe que na chamada da função pos=busca(V,n,5) estamos passando três parâmetros: V o vetor que possui os dados, n o número a ser procurado no vetor e 5 o tamanho do vetor). Além disso, estamos usando a variável pos que vai receber o retorno da nossa função busca. Caso o retorno seja -1, esse será o valor de pos indicando que não encontramos o valor n no vetor V. Caso contrário, o valor de pos será exatamente a primeira posição onde o valor procurado foi encontrado no vetor V. Veja que ao final do programa temos uma estrutura de seleção para tratar essas situações e emitir a mensagem adequada a cada caso.

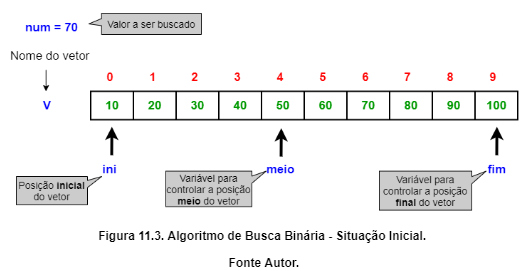
No lado direito da nossa figura 11.2 podemos verificar a função busca. Basicamente foi implementado um loop para fazer com que a variável i receba a cada passo do loop os valores das posições do vetor de 0 até a última posição que é tam-1. Dentro do loop usamos uma estrutura de seleção de forma que a cada passo do loop verificamos se o conteúdo de uma posição do vetor é igual ao valor que procuramos V[i]==num. Quando i é igual a 0 verificamos para a posição 0 do vetor e assim por diante. Caso o teste seja verdadeiro, encerramos a nossa busca através do comando return i. Lembrando que o comando return faz com que a função seja encerrada imediatamente. Nesse caso a função é encerrada e o valor i é retornado (o valor da posição onde o numero que estavamos procurando foi encontrado. Caso o valor não seja encontrado em nenhuma posição do vetor, então o loop é encerrado e retornamos -1 indicando que não encontramos o valor. Notamos que nessa função temos dois comandos return mas apenas um deles é executado uma vez que esse comando encerra a função e não mais é executado.

Perceba que nesse algoritmo o melhor desempenho ocorre quando o elemento que buscamos aparece na primeira posição do vetor (1 teste). Já o pior desempenho ocorre quando o elemento não é encontrado e temos que testar todas as posições do vetor (n testes).

**Busca Binária**

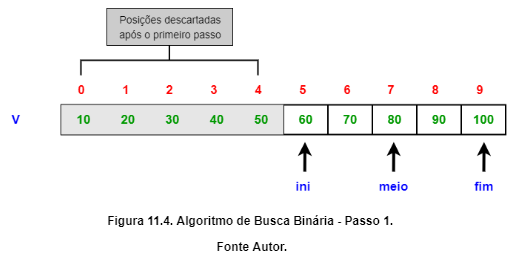
Um outro método bastante eficiente de busca é a busca binária. Vamos verificar que esse método precisa de um número bem inferior de comandos executados mesmo considerando-se o pior caso. Esse algoritmo tem como pré-requisito que o vetor onde a busca irá ocorrer deve estar previamente ordenado. Portanto temos que ter em mente que ele só poderá ser usado nessas condições.

A busca binária é um algoritmo clássico de busca. O algoritmo se aproveita do fato dos valores do vetor estarem ordenados. Vamos entender seu funcionamento através de um exemplo Figura 11.3:

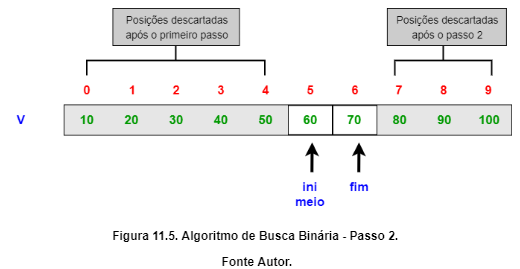


Situação Inicial - No início temos um vetor ordenado V, o número a ser buscado num, três variáveis indicando a posição de início, meio e fim do vetor. início começa com 0, fim com o valor da última posição do vetor 9 e meio com a média dessas posições (0+9)/2 = 4. Como as posições são números inteiros, desprezamos os valores após a vírgula. Figura 11.3.

**Passo 1** - Verificamos se o valor que buscamos está na posição do meio (50). Como não está, descartamos a metade onde não está o número. Nesse passo como vetor está ordenado e o valor que se encontra no meio é 50, podemos descartar o lado esquerdo desse vetor pois temos certeza que o número buscado 70 não está desse lado. Fazemos então o acerto das variáveis inicio, meio e fim para: inicio = 5, fim = 9 e meio = (5+9)/2 = 7 Figura 11.4. Perceba que apenas nesse passo descartamos metade dos elementos do vetor. Vamos seguir para o passo 2.



**Passo 2** - Verificamos se o valor que buscamos está na posição do meio (80). Como não está, descartamos a metade onde não está o número. Efetuamos o descarte de maneira semelhante ao passo 1. Descartando a metade onde não está o número considerando que o início agora é a posição 5. Como o valor que se encontra no meio é 80, podemos descartar o lado direito desse vetor pois temos certeza que o número buscado 70 não está desse lado (é menor que 80). Fazemos então o acerto das variáveis inicio, meio e fim para: inicio = 5, fim = 6 e meio = (5+6)/2 = 5 Figura 11.5.Vamos seguir para o passo 3.



**Passo 3 -** Verificamos se o valor que buscamos está na posição do meio (60). Como não está, descartamos a metade onde não está o número.. Como o valor que se encontra no meio é 60, podemos descartar o lado esquerdo desse vetor pois temos certeza que o número buscado 70 não está desse lado (é maior que 60). Fazemos então o acerto das variáveis inicio, meio e fim para: inicio = 6, fim = 6 e meio = (6+6)/2 = 6 Figura 11.6. Vamos seguir para o passo 4.



**Passo 4 -** Verificamos se o valor que buscamos está na posição do meio (70). Como é o valor que buscamos, encerramos a busca retornando a posição onde foi encontrado (6). Como sobrou apenas 1 elemento, caso esse não fosse o valor encontrado retornaríamos -1 indicando que o valor não foi encontrado. Perceba que quando inicio=meio=fim significa que temos apenas um elemento,

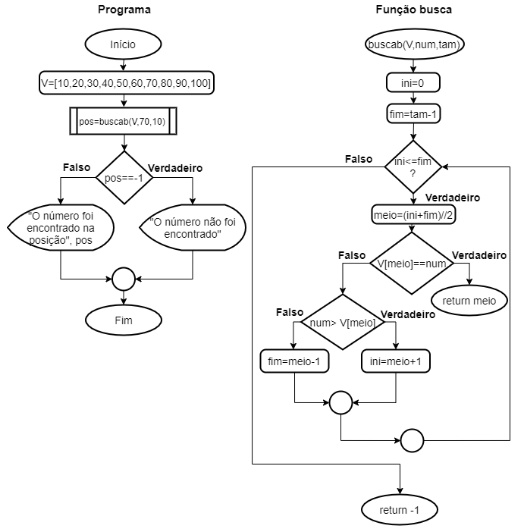
Perceba então que nesse algoritmo a cada passo o número de posições que temos que procurar se reduz à metade. Pode-se provar que o número de buscas em função do tamanho do vetor tam pode ser calculado como log2 tam. Caso tam = 1024, =10, ou seja para um vetor de 1024 posições a busca precisa de no máximo em 10 passos. Comparando com a busca linear, o mesmo vetor no pior caso demoraria 1024 passos ou elementos verificados (o tamanho do vetor). É portanto 100 vezes mais rápido esse algoritmo para um vetor de 1024 posições. Vamos verificar esse algoritmo usando fluxogramas:

**Exemplo 2** - Criar uma função que receba como parâmetro um vetor com números e um número a ser procurado. A função deve retornar a posição do vetor onde o número foi encontrado ou -1 caso não encontre o número. Usar o algoritmo de busca binária e considere que o vetor está ordenado em ordem crescente.

Podemos observar na Figura 11.7, do lado esquerdo temos a criação do vetor de 10 posições e a chamada da função pos=buscab(V,70,10). Na chamada estamos passando o vetor V, o número a ser pesquisado 70 e o tamanho do vetor 10. Em seguida temos o mesmo tratamento que foi feito na busca linear.Do lado direito temos a função que executa a busca binária. Na parte inicial da função armazenamos os valores iniciais das variáveis inicio e fim (primeira e última posição do vetor).

Monta-se em seguida um loop que finaliza quando inicio < fim (a medida em que o algoritmo é executado o valor dessas variáveis vai se aproximando). No loop efetuamos os comandos vistos acima para cada passo. Calcula-se a posição do meio meio=(ini+fim)//2, verifica-se se o valor é encontrado nessa posição (V[meio]==num). Se for encontrado, encerramos o algoritmo e retornamos a posição meio através do comando return meio. Caso contrário, ajustamos as variáveis inicio e fim.

Caso o valor seja maior do que o valor que está na posição do meio num> v[meio] desprezamos o lado esquerdo do vetor. Para isso ajustamos a variável inicio=meio +1. Caso contrário ajustamos desprezamos o lado direito ajustando a variável fim=meio-1. Se o valor for encontrado a função é encerrada. Caso o loop chegue ao final significa que o valor não foi encontrado e o comando return -1 é executado, retornando o valor -1 o que significa que o valor buscado não foi encontrado.



**Um algoritmo de ordenação Bubble Sort**

Como vimos informações ordenadas facilitam e muito as buscas. Um exemplo disso é a busca de uma palavra no dicionário. O fato do dicionário estar ordenado agiliza é muito a nossa busca pelas cerca de 300.000 palavras existentes,

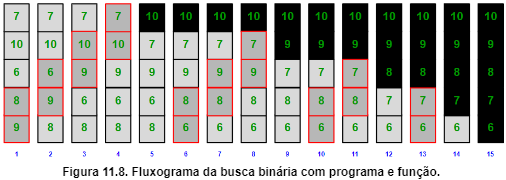
Vamos entender o funcionamento do algoritmo Bubble Sort ou método da bolha. Ele é chamado assim porque a medida em que o algoritmo é executado o valor ordenado vai subindo como bolhas. Veja na Figura 11.8 a sequência de passos na ordenação de um vetor. Ele foi colocado na vertical para sugerir exatamente o comportamento das bolhas. Seu funcionamento é o seguinte:

Fazemos a comparação entre duas posições vizinhas. Caso a posição anterior seja maior que a seguinte, efetuamos a troca dos valores dessas posições. Veja no passo 1 da Figura 11.8 efetuamos a comparação entre os valores 9 e 8. Como o 9 é menor que 8 efetuamos a troca (veja no passo 2). Essa troca será feita pela função troca(v,p1,p2) Figura 11.9 abaixo. No passo 2 efetuamos a troca das 2 próximas posições 9 e 6. Como o 9 é maior do que o 6 efetuamos a troca desses valores.

E assim sucessivamente passos 3, 4 e 5 até a última posição do vetor. Perceba que sempre o maior valor de todos termina na posição mais acima. Veja que o valor 10 em preto ficou na última posição. Como ele é o maior, já está na sua posição final. Portanto ele não fará parte das próximas iterações. A cada iteração é chamada a função empurra(v,n) Figura 11.9 abaixo.

Após o passo 5, já percorremos todas as posições do vetor efetuando as comparações entre as posições vizinhas e já movimentamos o maior valor para a sua posição final.Vamos agora iniciar uma nova iteração que vai varrer novamente as posições do vetor menos a última pois já está na sua posição correta. Executamos então os passos 6,7,8 e 9 onde posicionamos o 9 que é o maior valor na sua posição correta. Perceba que a medida que o número fica na sua posição correta estamos colorindo a posição de preto.

E assim é feito sucessivamente até que o vetor esteja ordenado. Perceba que a cada iteração um número fica na posição correta e deixamos de percorrer uma posição, ou seja, na primeira iteração percorremos 5 posições, depois 4, 3 até 1.



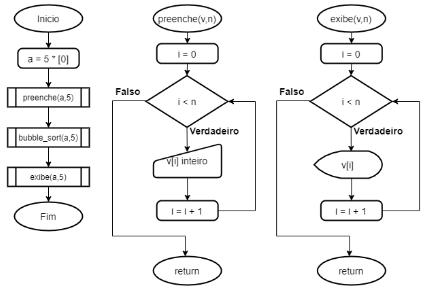
Para implementar o algoritmo vamos dividi-los em várias funções:

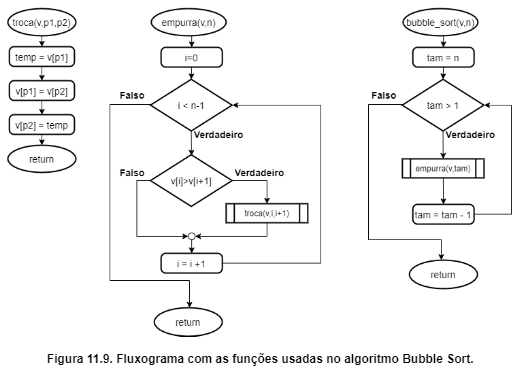
**Programa, funções preenche e exibe**: O programa cria o vetor, chama a função preenche(a,5) que preenche os valores do vetor, chama a função de ordenação usando o vetor preenchido bubble(a,5) e depois chama a função exibe(a,5) que vai exibir o vetor após a execução da função de ordenação. Espera-se que o vetor seja exibido ordenado. Todas essas funções possuem um segundo parâmetro com o tamanho do vetor.

**troca(v,p1,p2):** Essa função vai efetuar a troca entre duas posições p1 e p2 do vetor v. Após a execução dessa função o valor de v[p1] estará trocado com o valor de v[p2]

empurra(v,n): Executa o loop até o final do vetor (apenas 1 vez) verificando as posições vizinhas v[i]>v[i+1] e trocando (utilizando a função troca) caso o valor da posição anterior v[i] seja maior do que o valor da posição seguinte v[i+1]. A cada chamada da função empurra, um valor é ordenado. O objetivo dessa função é empurrar o maior valor para o final o vetor.

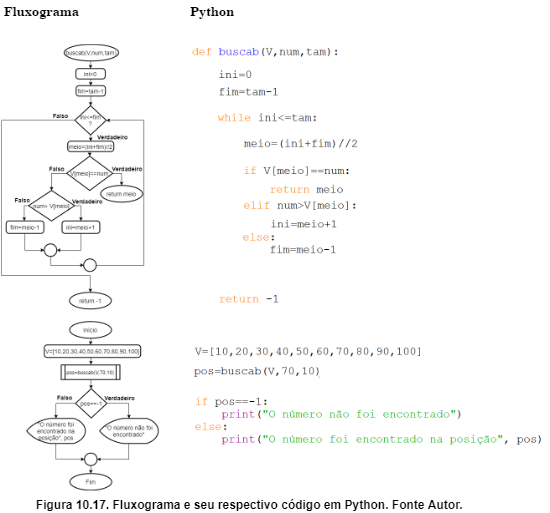
**bubble\_sort(v,n)**: Essa função ao final de sua execução deixa o vetor v ordenado. Ela chama a função empurra n-1 vezes. A cada vez que chamamos a função empurra, o parâmetro de tamanho do vetor é reduzido, pois a cada vez que a função empurra é invocada, um elemento acaba ficando em sua posição final ordenada no final do vetor.





**Comparativo entre fluxograma e Python**

Abaixo podemos identificar um fluxograma da função do exemplo 2 busca binária e sua conversão para python.



Bubble Sort - Ordenação em Bolha. [Bubble Sort - Ordenação em Bolha - Canal do Código](https://www.youtube.com/watch?v=IQYvXHVROws)

BUBBLE SORT ANIMATION. [BUBBLE SORT ANIMATION](https://youtu.be/JP5KkzdUEYI)

**Parte 12 -** [**Exercícios**](https://sites.google.com/faculdadeimpacta.com.br/lo-p12?pli=1&authuser=1) **-** [**Vídeo**](https://youtu.be/ZK0Rge87Vp0)

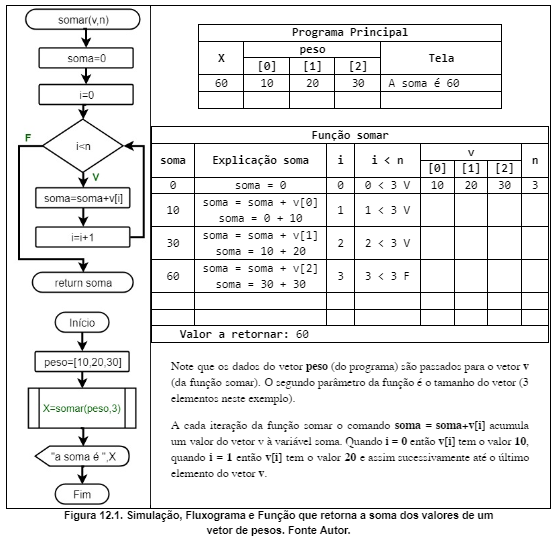
**Resumo**

Sabemos que para absorver os conceitos envolvendo matemática ou lógica necessitam de prática. Sem dúvida alguma o exercício da manipulação das ferramentas envolvendo a lógica de programação é de fundamental importância no aprendizado dessa disciplina. Dedicamos este capítulo na discussão de 5 exercícios resolvidos, nos quais utilizamos as ferramentas aprendidas neste curso. Ao final é mostrado um deles convertido para Python.

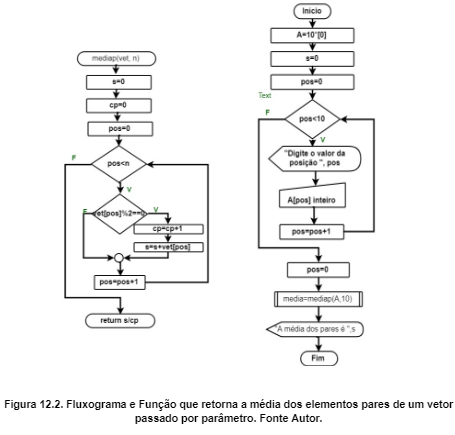
**Introdução**

Variáveis, comandos de atribuição, precedência de operadores aritméticos e lógicos, estruturas de seleção e repetição, contadores, acumuladores, vetores, funções, parâmetros, retorno de funções, algoritmos. Existem muitas ferramentas à nossa disposição para desenvolvermos nossos algoritmos. Vamos usá-las em nossos exercícios a seguir.

Exemplo 1- Escrever um fluxograma de uma função que retorna a soma um vetor de pesos. Criar também o programa que cria o vetor e chama a função.

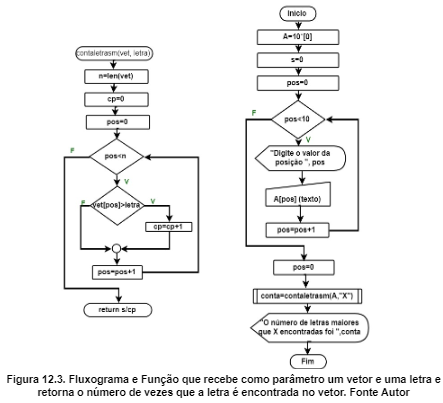


Exemplo 2 -Escrever um fluxograma que lê um vetor de inteiros com 10 elementos. Elaborar também uma função que calcula e retorna a média dos elementos pares de um vetor passado por parâmetro. Exibir a média.



Observe que nesse exemplo foram passados como parâmetros para a função um vetor e o tamanho do vetor (n). Embora existam em muitas linguagens de programação funções que devolvem o tamanho do vetor (listas em Python), a utilização de um parâmetro com a quantidade de elementos é muito interessante pois quando usamos essas funções que devolvem o tamanho do vetor isso nos obriga a varrer todo o vetor e muitas vezes estamos utilizando apenas um pedaço do vetor, não ele por completo. Essa alternativa de passar a quantidade de elementos usados do vetor por parâmetro é muito válida.

Exemplo 3 - Escrever um fluxograma que lê um vetor de caracteres com 10 elementos. Elaborar também uma função contaletrasm(vet,letra) que calcula e retorna o número de letras do vetor vet que são maiores que letra (também passado por parâmetro). Exibir o retorno da função contaletrasm.

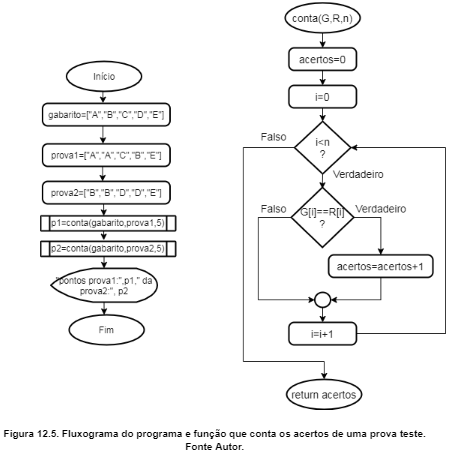


Veja que nesse exemplo utilizamos a função len() para obter o tamanho do vetor.

Exemplo 4 - Criar uma função que recebe como parâmetro 2 vetores G gabarito e R resposta relativos a questões de uma prova e retorna a quantidade de acertos.

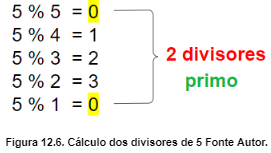
Nesse exemplo cada elemento do vetor representa a resposta de uma questão. A contagem consiste na seguinte verificação: cada posição dos vetores corresponde a uma pergunta. Portanto a posição 0 corresponde à primeira pergunta. O valor dessa posição no vetor Gabarito corresponde a resposta correta e o valor dessa posição no vetor resposta corresponde a resposta do aluno. Caso seus conteúdos sejam iguais, a resposta está correta (Figura 12.4). O Fluxograma deve fazer esses teste para todas as posições do vetor e a cada acerto efetuar a contagem (Figura 12.5).



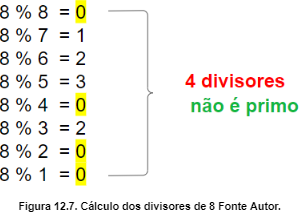


Exemplo 5 - Desenhar um Fluxograma de uma função que receba um número inteiro e retorne 1 caso o número seja primo e zero caso contrário

Primeiramente vamos entender como podemos calcular os divisores de um número e verificar se o número é primo. Veja a figura 12.6. Para verificarmos quais os divisores de 5 vamos calcular o resto da divisão de 5 por todos os números inferiores ou iguais a 5 até o número 1. Verificamos que teremos 5 divisões a serem feitas para os números 5,4,3,2 e 1. Percebemos que de todas essas divisões apenas as divisões pelo próprio número (5) e por 1 são exatas. Essa é a definição de número primo. Todo número primo é divisível apenas por ele mesmo e por 1 (todos os números são divisíveis por 1). Todos os números primos possuem apenas 2 divisores, logo 5 é primo porque possui e divisores.



Vejamos o mesmo cálculo para o número 8. Figura 12.7. Nesse caso percebemos que 8 possui 4 números que dividem 8 de forma exata (resto 0) 4 divisores. Como ele tem mais de 2 divisores portanto ele não é primo. Esse será o algoritmo que iremos utilizar. Vamos verificar para todos os números iguais ou inferiores ao número que queremos efetuar o teste se é primo, quantos dividem esse número de forma exata. Caso seja 2 é primo. Caso o número de divisores seja maior que 2 o número não é primo.



Na Figura 12.8 podemos observar os fluxogramas do programa e da função primo(n). A esquerda, no fluxograma do programa, a função primo é chamada x=primo(5). Perceba que o parâmetro da função é exatamente o número que queremos verificar se é primo. A variável x recebe o retorno dessa função. Se após a execução da função o valor recebido pela variável x for 0 significa que o número passado como parâmetro para a função não é primo. Caso contrário o número é primo. Ao final do programa foi colocada uma estrutura de seleção para exibir uma mensagem adequada para cada caso.

Vamos observar a função primo(n) agora. Nesta função usamos a variável c para contar a quantidade de divisores de n. Usamos também uma variável de controle i de um loop. Nesse loop, a variável de controle i começa valendo n e a cada passo é decrementada até atingir o valor 0 quando o loop é interrompido. A cada valor de i verifica-se se esse valor é divisor de n usando o teste: n%i==0. Se o teste for verdadeiro, c é incrementada indicando que encontramos mais um divisor de n. Ao final, o valor da variável c indica quantos divisores foram encontrados para n. É feito então um tratamento usando a estrutura de seleção de maneira que caso o número seja primo (isso ocorre quando c é igual a 2) o retorno será 1, caso contrário o retorno será 0.

