LÓGICA DE PROGRAMAÇÃO

PROF.: JOÃO VICTOR PEREIRA

LÓGICA

Segundo o filósofo Copi, “o estudo da lógica é o estudo dos métodos e princípios usados para distinguir o raciocínio correto do incorreto”. Em outras palavras, pode-se dizer que a lógica consiste no uso de princípios e conhecimentos para que o ser humano possa distinguir um raciocínio correto.

Todos possuem este conhecimento chamado lógica e todos utilizam-na no dia a dia, mesmo sem perceber, pois torna-se algo trivial e natural para o ser humano. Este conceito é necessário para formular uma sequência de raciocínio que permita o ser humano usar de premissas e informações para obter uma conclusão.

LÓGICA

Ex.:

a) Porto Alegre pertence ao estado do Rio Grande do Sul.

b) Rio Grande do Sul é uma unidade federativa do Brasil.

c) Logo, Porto Alegre pertence ao Brasil.

Neste exemplo foram apresentadas duas premissas iniciais (a) e (b), que servem como evidências que sustentam uma conclusão (c).

LÓGICA

Um segundo exemplo da utilização da Lógica irá explorar os passos necessários para passar por uma porta que se encontra trancada, para tanto, pode-se executar a sequência a seguir:

a) colocar a chave na fechadura;
b) girar a chave no sentido anti-horário para destrancar a porta;
c) retirar a chave da fechadura;
d) girar a maçaneta;
e) puxar a porta para abri-la;
f) passar pela entrada;
g) puxar a porta novamente para fecha-la;
h) colocar novamente a chave na fechadura;
i) girar a chave no sentido horário para trancar a porta;
j) retirar a chave da fechadura.

LÓGICA

A sequência de passos do exemplo anterior envolve um raciocínio lógico que pode ser executado para passar por uma porta trancada, um procedimento comum, realizado no dia a dia de um ser humano, porém, provavelmente, nunca analisado desse ponto de vista.

Este é um exemplo simples de uma atividade que envolve raciocínio lógico, porém, sabe-se que todas as atividades envolvem lógica, portanto, é necessário buscar a percepção das sequências lógicas que são executadas no dia a dia, desde atividades como tomar banho e escovar os dentes até atividades como pegar um ônibus ou dirigir um carro.

Os conceitos de lógica são extremamente necessários para o desenvolvimento de códigos a serem produzidos usando a Lógica de Programação.

LÓGICA

CASOS E RELATOS

Trocando o pneu

Um motorista dirigia solitário, quando percebeu que o pneu do seu carro estava furado e que precisava encostar para realizar a troca. Sem pensar duas vezes, seguiu o seu instinto e, antes de qualquer coisa, montou e posicionou o triângulo a fim de sinalização. Na sequência, observou o pneu que estava furado, montou o macaco e levantou o carro. Na etapa seguinte, o motorista se deparou com um problema: ao posicionar a chave de roda e colocar força para desparafusar a roda, percebeu que ela começou a girar, não permitindo afrouxar o parafuso. Neste momento, percebeu que algo estava errado, e que precisaria repensar os seus passos até aquele ponto. Foi quando percebeu que a sequência lógica aplicada para a resolução deste problema estava resultando em uma conclusão incorreta e que precisaria retornar algumas etapas. Neste momento, desceu o carro novamente e aí sim conseguiu desparafusar as rodas, na sequência, levantou o carro novamente, trocou a roda, e como já havia aprendido com a situação anterior, desceu o carro para posteriormente parafusar a roda novamente. Neste contexto, é possível observar que um passo errado dentro da sequência lógica levou a uma conclusão falha, sendo necessário refazer o trabalho para atingir uma conclusão correta.

LÓGICA DE PROGRAMAÇÃO

A lógica de programação surgiu a partir dos princípios da lógica e consiste em uma técnica de encadear pensamento para atingir um determinado objetivo. Para isso, é necessário fazer uso de uma sequência lógica, que é um conjunto de passos a serem executados para atingir um objetivo ou a solução de um problema.

Quando se fala em sequência lógica, fala-se de **algoritmos**.

Um **algoritmo** é uma sequência de instruções que resolve uma determinada tarefa, de forma que indiquem as ações a serem executadas *(o passo a passo)*.

Todo algoritmo segue um princípio básico de execução que consiste em **entrada, processamento e saída**. É necessário observar as entradas do algoritmo, depois realizar o processamento necessário (com base nas entradas obtidas) e, por fim, apresentar o resultado obtido como a saída do problema.

LÓGICA DE PROGRAMAÇÃO

Ex.: tem-se um problema no qual serão obtidas as três notas de um aluno, para informar, na sequência, se o aluno foi aprovado, ficou de recuperação ou foi reprovado, com base na média final. Para a resolução deste problema, serão necessários dois algoritmos, sendo que cada um resolverá um pequeno problema desta situação.

1º: calcular a média do aluno. Como entrada tem-se as três notas obtidas durante o semestre; como processamento tem-se o cálculo da média, realizando a soma das três notas e, posteriormente, dividindo a soma por três; por fim, a saída será a média obtida por este aluno.

2º: verificar a situação final do aluno. A entrada será a média obtida com o 1º algoritmo; o processamento será a verificação da média para descobrir se foi maior ou igual a 7, menor que 7 e maior ou igual a cinco, ou menor que cinco; com esta verificação, é possível definir a saída, informando se o aluno foi aprovado, está em recuperação ou se foi reprovado.

LÓGICA DE PROGRAMAÇÃO: Formas de Representação

Até o momento sabe-se o que é lógica e lógica de programação, e que um programa é formado por um ou mais algoritmos, que representam uma sequência lógica de passos que levam à resolução de um determinado problema. Para realizar a representação destes algoritmos, existem três formas tradicionais que são usadas, umas para fins mais didáticos e outras para representações mais próximas da linguagem de programação.

LÓGICA DE PROGRAMAÇÃO: Formas de Representação

Descrição Narrativa: utilizada estritamente para fins didáticos para um primeiro contato com a elaboração de um algoritmo. Utiliza linguagem natural para expressão dos algoritmos, permitindo a definição da sequência lógica, passo a passo, em busca da resolução do problema.

Uma das vantagens da utilização desta forma de representação é que fica aberto para que sejam usados mais ou menos detalhes a fim de resolver o algoritmo. Enquanto uma das desvantagens é que neste formato é dada uma oportunidade para más interpretações, justamente pela linguagem usada ser a linguagem natural.

O exemplo a seguir apresentará um algoritmo que partirá de quatro notas de um aluno, calculará a média e informará se o aluno está aprovado ou não. Para isto, será necessária a adição de uma tomada de decisão para obter uma saída.

LÓGICA DE PROGRAMAÇÃO: Formas de Representação

a) obter as quatro notas do aluno;

b) somar as quatro notas obtidas;

c) dividir o resultado da adição por quatro;

d) se a média obtida for maior ou igual a 7(sete), o aluno foi aprovado; caso contrário, o aluno foi reprovado.

LÓGICA DE PROGRAMAÇÃO: Formas de Representação

Fluxogramas: representação feita a partir de figuras geométricas que descrevem diferentes ações a serem realizadas durante a execução do algoritmo.

O fluxograma auxilia na elaboração do raciocínio lógico a ser seguido para a resolução de um problema. Por usar representação gráfica, ajuda a visualizar melhor um processo, compreendê-lo mais facilmente e encontrar falhas ou problemas de eficiência. Toda a execução do algoritmo segue as setas de fluxo, que por ser a representação de um algoritmo, deve seguir apenas um caminho, obedecendo uma sequência lógica, salvo quando existe a divisão do algoritmo por conta de um processo de decisão.

Processo

Início/Fim

Decisão

Setas de Fluxo

Leitura

Escrita

LÓGICA DE PROGRAMAÇÃO: Formas de Representação

a) Início/Fim: todo fluxograma deve iniciar e encerrar com este símbolo. Deve conter apenas um início, porém poderá possuir mais de um fim, caso seja dividido durante o processo.

b) Decisão: divide a execução do fluxograma em dois caminhos. Sempre que usado, uma pergunta deve ser feita: caso a resposta seja verdadeira, o fluxograma segue por um caminho; caso contrário, segue por outro. Esta é a única situação em que devem ser usadas duas setas de fluxo a partir de uma figura geométrica.

c) Leitura: representa uma entrada do usuário, quando o programa fará uma leitura de uma informação digitada pelo usuário.

d) Escrita: representa a impressão de alguma informação na tela pelo programa, a fim de informar algo ao usuário.

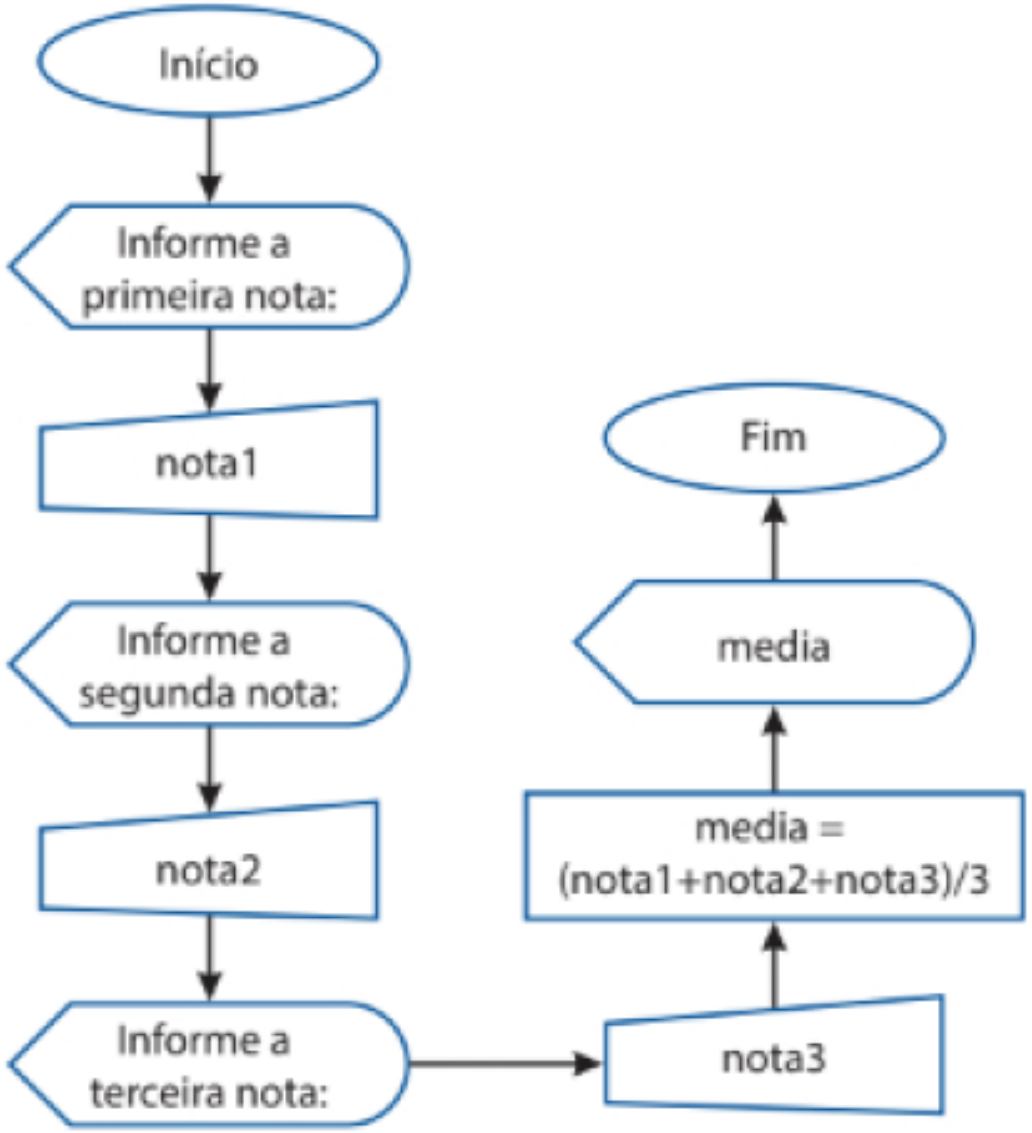
LÓGICA DE PROGRAMAÇÃO: Formas de Representação

e) Setas de Fluxo: representam o caminho do fluxograma, a partir do início. A leitura do fluxograma é feita seguindo as setas de fluxo.

f) Processo: utilizado quando algo deve ser processado pelo programa, por exemplo, para cálculos matemáticos.

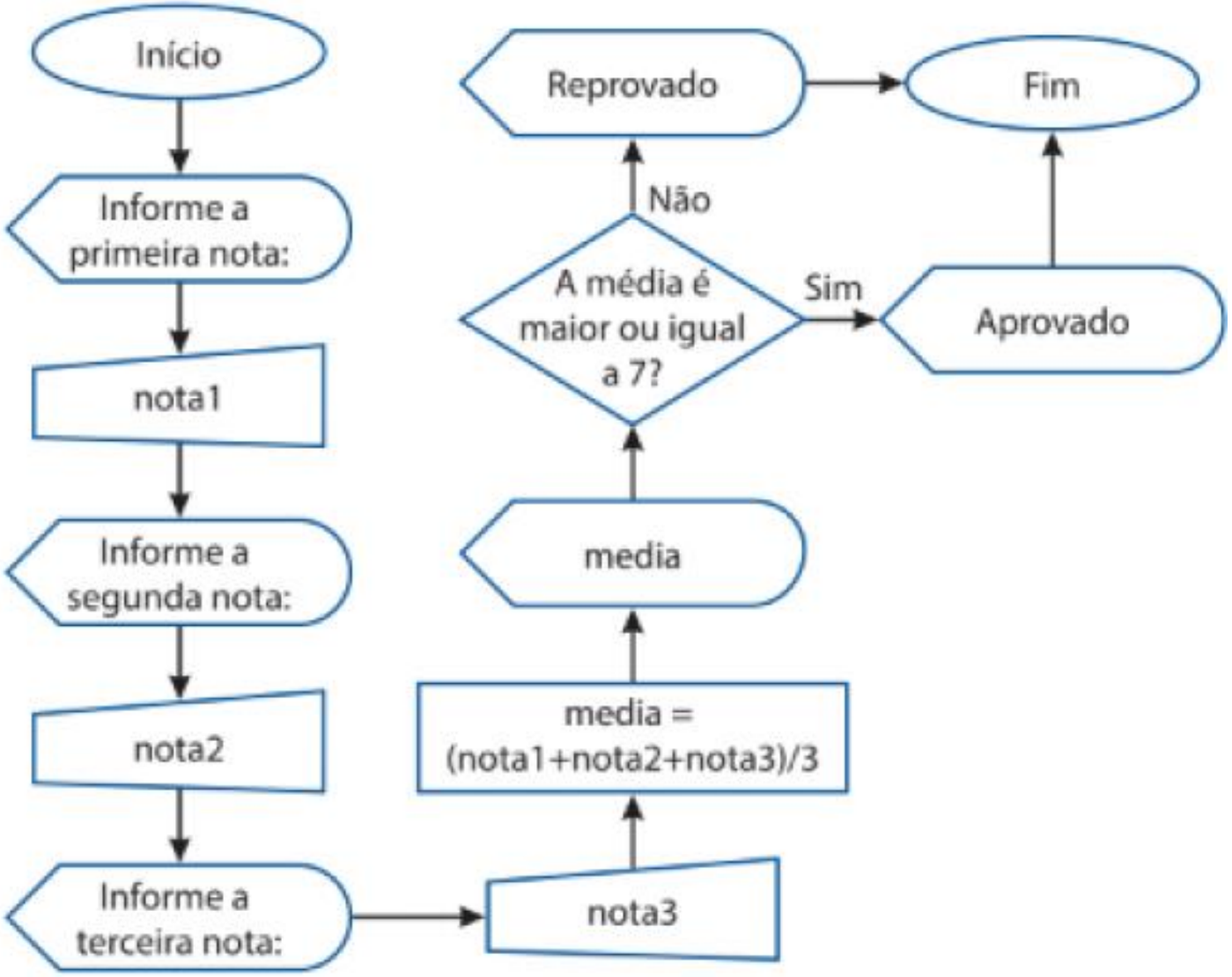
Para exemplificar, a imagem a seguir apresenta um fluxograma de um algoritmo que solicita três notas, calcula e apresenta a média.

LÓGICA DE PROGRAMAÇÃO: Formas de Representação



Após o início, tem-se a solicitação das três notas do aluno. Para isto é usada a escrita para apresentar ao usuário que este deve informar a nota, na sequência de cada escrita é feita a leitura da nota informada. Para as leituras, usa-se um nome genérico, como **nota1**, **nota2** e **nota3**. A ideia do fluxograma é poder ser executado para qualquer situação que se adeque a este problema, portanto, se utilizar valores fixos, se adequaria apenas àquela situação. Após a solicitação das três notas, é realizado um processo para calcular a média, e o resultado é considerado o nome genérico “média”. Por fim, escreve-se para o usuário a média calculada e finaliza o programa.

LÓGICA DE PROGRAMAÇÃO: Formas de Representação



Neste exemplo foi inserida uma decisão após o cálculo e apresentação da média do aluno. Como para toda decisão é necessário ter uma pergunta, neste caso a pergunta foi "A média é maior ou igual a 7?", caso a resposta seja sim, seguirá a seta do fluxo que irá imprimir para o usuário a palavra **Aprovado**; caso contrário, seguirá o caminho que irá imprimir a palavra **Reprovado**. Por fim, pode-se observar que, qualquer um dos dois caminhos seguidos pelo algoritmo, ele sempre terminará na figura que representa o fim.

LÓGICA DE PROGRAMAÇÃO: Formas de Representação

Pseudocódigo: forma de representação do algoritmo que mantém uma proximidade entre uma linguagem de programação e a linguagem natural. Assim como o fluxograma, o pseudocódigo auxilia na elaboração do raciocínio lógico a ser seguido para a resolução de um problema, porém, ao invés de usar figuras geométricas para esta representação, utiliza uma linguagem estruturada, que pode ser escrita em qualquer idioma.

O pseudocódigo é representado por três etapas: a identificação do algoritmo, a declaração das variáveis e o corpo do algoritmo, que contém a sequência lógica para a resolução dele. Uma variável, resumidamente, é um espaço na memória para armazenar dado. A este espaço é dado um nome e definido o tipo de dado na sua declaração.

O exemplo a seguir realiza o cálculo da média e apresenta se o aluno está aprovado ou não. Mesmo exemplo anterior, porém agora usando pseudocódigo.

LÓGICA DE PROGRAMAÇÃO: Formas de Representação

```
1. Algoritmo CalcularMedia
2. var
3.  nota1,nota2,nota3,media:real
4.
5. início
6.  escrever("Digite a primeira nota: ")
7.  ler(nota1)
8.  escrever("Digite a segunda nota: ")
9.  ler(nota2)
10. escrever("Digite a terceira nota: ")
11. ler(nota3)
12. media <- (nota1+nota2+nota3)/3
13. escrever(media)
14. se media >= 7 então
15.     escrever("Aprovado")
16. senão
17.     escrever("Reprovado")
18. fimse
19. fim
```

Na linha 1, tem-se a etapa de identificação do algoritmo, que define o nome "CalcularMedia" para o algoritmo. Nas linhas 2 e 3, tem-se a etapa de declaração das variáveis, que define as variáveis *nota1*, *nota2*, *nota3* e *media* com o tipo **real**, ou seja, estes nomes genéricos podem ter números com vírgulas relacionados a eles. Por fim, é definido o corpo do algoritmo entre as linhas 5 e 19, onde são solicitadas as três notas (linhas 6 a 11), é realizado o cálculo da média (linha 12) e a decisão (linhas 14 a 18) que definirá se o aluno está aprovado ou não com base na verificação da média.

LÓGICA DE PROGRAMAÇÃO: Linguagem de Programação

Ao decorrer dos anos, com a evolução tecnológica, as linguagens de programação foram sendo aprimoradas, facilitando a interação entre o humano e o computador. As linguagens de programação são divididas em duas: as de baixo nível (mais próximas da linguagem de máquina e mais complexas), e as de alto nível (mais próximas da linguagem natural e largamente utilizadas atualmente).

Existem duas formas de execução de um algoritmo desenvolvido em linguagem de programação, chamadas de linguagens interpretadas e compiladas. Em ambas acontece a tradução da linguagem de alto nível para a linguagem de máquina, porém de formas diferentes.

As linguagens interpretadas executam o código do programa à medida em que ele vai sendo traduzido, o que torna a sua execução mais flexível. Ex.: JavaScript e Python.

As linguagens compiladas primeiramente traduzem todo o código do programa para depois executar. Para isto, é necessária a existência de um compilador, que é responsável por esta tradução para a linguagem de máquina. E.: Java, C, C++ e C# *(lê-se C Sharp)*.

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

Variáveis: ao desenvolver um algoritmo, é necessário armazenar dados para posterior utilização. Por exemplo: ao implementar um algoritmo que solicita dois números para, em seguida, mostrar a soma entre eles, será necessário solicitar o valor do primeiro número e armazená-lo em algum lugar. Mesma coisa acontece para o segundo número.

Para armazenar esses dados, utiliza-se de **variáveis**. Uma variável é um dado que será armazenado na memória RAM do computador, possibilitando a recuperação deste dado posteriormente.

Toda variável é composta por três informações bases: um identificador, um valor e um tipo de dado.

O identificador é o nome que definirá a variável, o qual será chamado quando for necessário recuperar o dado. É uma boa prática que o programador use nomes que correspondam com o objetivo da criação da variável. Por exemplo, se for criada uma variável para definir o nome de uma pessoa, um bom identificador para essa variável seria “nomePessoa”.

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

Algumas regras devem ser observadas ao definir o nome de uma variável, obedecendo os caracteres válidos para a maioria das linguagens de programação, que são:

a) palavras: palavras com letras maiúsculas ou minúsculas, sem espaço e sem acento;

Obs.: como boa prática, o nome da variável deverá iniciar sempre com letra minúscula. Caso a variável receba um nome com 2 ou mais termos, o ideal é que o primeiro termo inicie com letra minúscula e os demais termos com letra maiúscula. Ex.: vrMediaAluno

b) números: podem ser utilizados desde que apareçam após uma ou mais letras.

c) *underline (_)* e cifrão ($): são considerados caracteres válidos e aceitos em qualquer local do identificador. Esses caracteres não são muito utilizados, porém algumas poucas linguagens usam o *underline* para identificar variáveis que tenham espaço no nome.

Alguns nomes não são aceitos como identificadores de variáveis por serem consideradas palavras reservadas da linguagem, identificam outras operações já definidas pela linguagem de programação.

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

Além do identificador, uma variável também possui um tipo de dado, que representa que tipo de dado poderá ser armazenado na variável.

O terceiro conceito que compõe uma variável é o seu valor, que deverá estar de acordo com o seu tipo de dado. O valor sempre será a informação que o usuário deseja guardar, que possuirá um tipo de dado específico e um identificados para auxiliar na recuperação do dado.

Além do conceito de variáveis, deve-se compreender que existem dois tipos de variáveis que podem ser declaradas: **variáveis globais e variáveis locais**.

Uma variável global consiste em uma variável declarada de forma que todo o código poderá acessá-la.

A variável local possui um acesso mais restrito. É sempre declarada dentro de um bloco de código específico, que pode pertencer a uma estrutura de controle ou um procedimento.

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

Constantes: consideradas variáveis, porém trabalham com valores fixos, imutáveis, que devem ser definidos no início do algoritmo e não podem ter seus valores alterados durante a execução do algoritmo.

Ex.: PI <- 3.14159265359
MESES <- 12
POLEGADA <- 2.54

Obs.: Por padrão de nomenclatura, todas as constantes devem ser definidas usando apenas letras maiúsculas.

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

Tipos de Dados: cada linguagem de programação tem os seus tipos de dados, porém a maioria segue uma padronização quanto aos nomes. Todos os tipos de dados primitivos possuem como base quatro tipos:

a) inteiro (int): tipo numérico que define números inteiros negativos e positivos;

b) real (float): tipo numérico que define números decimais com vírgula, negativos e positivos;

c) lógico (bool): tipo de dado que aceita apenas dois valores, verdadeiro (1) *[true]* ou falso (0) *[false]*;

d) literal (str): tipo de dado que define um conjunto de caracteres, aceitando letras, números e símbolos.

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

Operadores: usados para representar expressões de cálculo, comparação, condição e expressão. São divididos em quatro tipos principais: de atribuição, aritméticos, relacionais e lógicos.

a) de atribuição: responsável por atribuir uma valor a uma variável. Na pseudolinguagem é definido pelo símbolo **<-** ou **:=**

Ex.:

```
v1 <- 5        v1 := 5
v2 <- 7        v2 := 7
```

Este algoritmo pode ser lido como uma atribuição do número 5 à variável **v1**, assim como está atribuindo o número 7 à variável **v2**.

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

b) aritméticos: usados para representar cálculos matemáticos mais clássicos, tendo como base cinco operações: adição, subtração, multiplicação, divisão e módulo.

OPERADOR	SÍMBOLO	EXEMPLO
Adição	+	2 + 2
Subtração	-	2 – 2
Multiplicação	*	2 * 2
Divisão de inteiro	DIV	2 DIV 2
Divisão de real	/	2 / 2
Módulo	MOD	4 % 3

Ex.:

adição: soma1 <- 5 + 10

soma2 := 10 + 15 + 20

soma3 = 30 + 40 + 70 *(python usa = para atribuição)*

subtração: subtração = 10 – 3

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

b) aritméticos: usados para representar cálculos matemáticos mais clássicos, tendo como base cinco operações: adição, subtração, multiplicação, divisão e módulo.

OPERADOR	SÍMBOLO	EXEMPLO
Adição	+	2 + 2
Subtração	-	2 – 2
Multiplicação	*	2 * 2
Divisão de inteiro	DIV	2 DIV 2
Divisão de real	/	2 / 2
Módulo	MOD	4 % 3

Ex.:

multiplicação: multiplicacao1 <- 10 * 4

multiplicacao2 = 25 * 3 *(python)*

divisão de inteiro: divisao1 := 36 div 4

divisão2 = 360 // 9 *(python usa // para divisão de inteiros)*

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

b) aritméticos: usados para representar cálculos matemáticos mais clássicos, tendo como base cinco operações: adição, subtração, multiplicação, divisão e módulo.

OPERADOR	SÍMBOLO	EXEMPLO
Adição	+	2 + 2
Subtração	-	2 – 2
Multiplicação	*	2 * 2
Divisão de inteiro	DIV	2 DIV 2
Divisão de real	/	2 / 2
Módulo	MOD	4 % 3

Ex.:

<u>divisão de real:</u> divReal1 <- 36 / 4

divReal2 = 10 / 2 *(python também usa / para divisão de real (float))*

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

b) operadores aritméticos: usados para representar cálculos matemáticos mais clássicos, tendo como base cinco operações: adição, subtração, multiplicação, divisão e módulo.

OPERADOR	SÍMBOLO	EXEMPLO
Adição	+	2 + 2
Subtração	-	2 – 2
Multiplicação	*	2 * 2
Divisão de inteiro	DIV	2 DIV 2
Divisão de real	/	2 / 2
Módulo	MOD	4 % 3

MOD: retorna o resto de uma divisão. Pode dividir tanto inteiro como real.

Ex.:

módulo: resto <- 35 MOD 4 *(terá resto 3, pois a divisão não é exata, sobrando um resto 3)*

resto2 = 10 % 2 *(no python, MOD é representado pelo sinal de %) [resto 0, div exata]*

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

Para usar os operadores aritméticos, deve-se lembrar das regras de prioridade da matemática, ou seja, qual operação deve ser feita primeiro. A seguinte ordem de prioridade deve ser seguida:

1ª: parênteses
2ª: módulo, divisão e multiplicação
3ª: adição e subtração

Primeiro devem ser executadas as operações que estiverem dentro de parênteses, partindo dos parênteses mais internos para os mais externos. Na sequência, deverão ser realizados os cálculos referentes ao módulo, divisão e multiplicação, todos no mesmo nível de prioridade; e, por fim, os cálculos de adição e subtração.

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

Ex.: tendo como base a seguinte expressão, quais seriam as ordens de prioridade?

5 * (3 + 4) + 4

Inicialmente deve-se resolver a adição dentro dos parênteses, na sequência a multiplicação e, por fim, a adição.

5 * (3 + 4) + 4
5 * 7 + 4
35 + 4
39

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

E se a expressão for **5 * 4 / 2 MOD 6**, qual seria a ordem de prioridade?

As operações da expressão pertencem a mesma classe de prioridade, portanto deve-se começar da esquerda para a direita.

Ex.:

5 * 4 / 2 MOD 6
20 / 2 MOD 6
10 MOD 6
4

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

CASOS E RELATOS

Uma empresa de desenvolvimento de software recebeu uma demanda de uma escola para o desenvolvimento de um sistema de gerenciamento escolar, o qual deve, entre as suas funções, registrar as notas dos alunos e realizar o cálculo da média final. Toda a implementação foi realizada de acordo com as solicitações da escola, porém, na hora dos testes, estava acontecendo uma inconsistência na geração das médias, as quais, por algum motivo, geralmente ficavam acima de 10 em valores absurdos. Após uma profunda análise no código que realiza o cálculo da média, descobriu-se que o motivo desse erro consistia na errada utilização das prioridades junto aos operadores aritméticos, estando o cálculo da seguinte forma:

n1+n2+n3/3

Desta forma, este cálculo primeiro estava dividindo o valor da nota 3 pelo número 3, para posteriormente adicionar os valores referentes as notas 1 e 2, o que estava ocasionando as médias absurdas. Para solucionar o problema, adicionaram os parênteses nas adições para dar prioridade a elas e posteriormente dividir, ficando da seguinte forma:

(n1+n2+n3)/3

Após isso, o sistema passou em todos os testes e foi entregue para a escola conforme solicitado.

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

c) operadores relacionais: são usados para realizar a comparação entre dois valores. Assim como os operadores aritméticos, estes também possuem como base operadores existentes na matemática, que consistem em:

- Igualdade;
- Diferença;
- Maior que;
- Maior ou igual que;
- Menor que;
- Menor ou igual que;

Esses operadores sempre retornarão como resultado um valor lógico: verdadeiro ou falso.

A tabela a seguir aponta o nome, símbolo e exemplo referente aos operadores relacionais supracitados. No python, alguns operadores são diferentes, estes serão pontuados em seguida.

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

OPERADOR	SÍMBOLO	EXEMPLO
Igualdade	=	5 = 5
Diferença	<>	5 <> 4
Menor que	<	4 < 5
Menor ou igual que	<=	5 <= 5
Maior que	>	5 > 4
Maior ou igual que	>=	5 >= 5

No Python, os operadores de Igualdade e Diferença são diferentes:

OPERADOR	SÍMBOLO	EXEMPLO
Igualdade	==	5 == 5
Diferença	!=	5 != 4

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

A igualdade é representada pelo sinal de = ***(no python, ==)***, e pode ser utilizada para verificar a igualdade entre valores de uma variável.

A diferença, representada pelo símbolo <> ***(no python, !=)***, realiza a comparação inversa ao da igualdade, realizando a comparação entre dois valores e retornando verdadeiro caso sejam diferentes, e falso caso sejam iguais.

O operador "menor que", representado por <, representa a verificação se um valor é menor que o outro. É importante atentar-se para o fato de que o retorno será verdadeiro caso o número localizado à esquerda for menor que o da direita; caso for maior ou igual ao outro número, então o retorno será falso.

Semelhante ao operador anterior, o operador "menor ou igual que", definido por <=, representa a verificação considerando também se os números são iguais. Neste caso, o valor localizado à esquerda deve ser menor ou igual ao valor localizado à direita para retornar verdadeiro, caso contrário retornará falso.

LÓGICA DE PROGRAMAÇÃO: Tipos de Dados e Operadores

Assim como existem os operadores para verificar se um valor é menor que o outro, também há os operadores para verificar se um valor é maior que o outro. Começando pelo operador “maior que”, representado por >, verifica se o valor localizado à esquerda é maior que o valor da direita, caso seja, retorna verdadeiro, caso contrário, falso.

O operador “maior ou igual que”, representado por >=, considera também como verdadeiro caso os dois valores sejam iguais.

O resultado de uma comparação é um valor lógico, ou seja **True** (verdadeiro) ou **False** (falso).

LÓGICA DE PROGRAMAÇÃO: Exemplo de Uso de Operadores Relacionais

a = 1	a recebe 1
b = 5	b recebe 5
c = 2	c recebe 2
d = 1	d recebe 1

a == b	a é igual a b?	**False**
b > a	b é maior que a?	**True**
a < b	a é menor que b?	**True**
a == d	a é igual a d?	**True**
b >= a	b é maior ou igual a a?	**True**
c <= b	c é menor ou igual a b?	**True**
d != a	d é diferente de a?	**False**
d != b	d é diferente de b?	**True**

LÓGICA DE PROGRAMAÇÃO: Operadores Lógicos

Para agrupar operações com lógica booleana, usa-se operadores lógicos. O Python suporta três operadores básicos: **not** (não), **and** (e) e **or** (ou). Esses operadores podem ser traduzidos como **não** (negação), **e** (conjunção) e **ou** (disjunção).

Operador Python	Operação
not	não
and	e
or	ou

Cada operador obedece a um conjunto simples de regras, expresso pela tabela verdade desse operador. A tabela verdade demonstra o resultado de uma operação com um ou dois valores lógicos ou operandos. Quando o operador utiliza apenas um operando, dizemos que é um operador unário. Ao utilizar dois operandos, é chamado operador binário. O operador de negação (**not**) é um operador unário, **or** (ou) e **and** (e) são operadores binários, precisando, assim, de dois operandos.

LÓGICA DE PROGRAMAÇÃO: Operadores Lógicos - NOT

O operador **not** (não) é o mais simples, pois precisa apenas de um operador. A operação de negação também é chamada de inversão, pois um valor verdadeiro negado se torna falso e vice-versa. A tabela verdade do operador **not** (não) é apresentada abaixo:

V_1	not V_1
V	F
F	V

Ex.:

```
>>> not True
False

>>> not False
True
```

LÓGICA DE PROGRAMAÇÃO: Operadores Lógicos - AND

O operador **and** (e) tem sua tabela verdade representada abaixo. O operador **and** (e) resulta verdadeiro apenas quando seus dois operadores forem verdadeiros:

V_1	V_2	V_1 and V_2
V	V	V
V	F	F
F	V	F
F	F	F

Ex.:

```
>>> True and True
True

>>> True and False
False

>>> False and True
False

>>>False and False
False
```

LÓGICA DE PROGRAMAÇÃO: Operadores Lógicos - OR

A tabela verdade do operador **or** (ou) é apresentada abaixo. A regra fundamental do operador **or** (ou) é que ele resulta em falso apenas se seus dois operadores também forem falsos. Se apenas um de seus operadores for verdadeiro, ou se os dois forem, o resultado da operação será verdadeiro.

V_1	V_2	V_1 or V_2
V	V	V
V	F	V
F	V	V
F	F	F

Ex.:

>>> True or True
True

>>> False or True
True

>>> True or False
True

>>>False or False
False

LÓGICA DE PROGRAMAÇÃO: Expressões Lógicas

Os operadores lógicos podem ser combinados em expressões lógicas mais complexas. Quando uma expressão tiver mais de um operador lógico, avalia-se o operador **not** (não) primeiramente, seguido do operador **and** (e) e, finalmente, **or** (ou). A seguir, está detalhada a ordem de avaliação da expressão, onde a operação sendo avaliada é sublinhada; e o resultado, mostrado na linha seguinte.

```
True or False and not True
True or False and False
True or False
True
```

LÓGICA DE PROGRAMAÇÃO: Expressões Lógicas

Os operadores relacionais podem ser utilizados em expressões com operadores lógicos.

```
salário > 1000 and idade > 18
```

Nesses casos, os operadores relacionais devem ser avaliados primeiramente. Supõe-se que **salário = 100** e **idade = 20:**

```
salário > 1000 and idade > 18
   100 > 1000 and 20 > 18
      False and True
         False
```

A grande vantagem desse tipo de expressão é representar condições que podem ser avaliadas com valores diferentes. Por exemplo: imagine que **salário > 1000** and **idade > 18** seja uma condição para um empréstimo de compra de um carro novo. Quando **salário = 100** e **idade = 20**, sabe-se que o resultado da expressão é falso, e pode-se interpretar que, nesse caso, a pessoa não receberia o empréstimo.

LÓGICA DE PROGRAMAÇÃO: Variáveis String

Variáveis do tipo **string** armazenam cadeias de caracteres como nomes e textos em geral. Chamamos cadeia de caracteres uma sequência de símbolos como letras, números, sinais de pontuação etc. Exemplo: *João e Maria comem pão*. Nesse caso, João é uma sequência com as letras J, o, ã, o. Para simplificar o texto, será usado o nome **string** para mencionar cadeias de caracteres. Podemos imaginar uma string como uma sequência de blocos, onde cada letra, número ou espaço em branco ocupa uma posição, como mostra a figura abaixo:

String																				
J	o	ã	o		e		M	a	r	i	a		c	o	m	e	m		p	ã	o

LÓGICA DE PROGRAMAÇÃO: Variáveis String

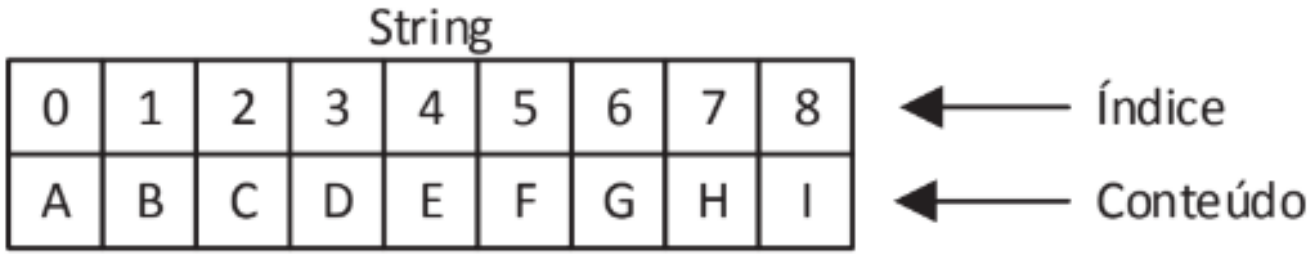
Uma string em Python tem um tamanho associado, assim como um conteúdo que pode ser acessado caractere a caractere. O tamanho de uma string pode ser obtido utilizando-se a função **len**. Essa função retorna o número de caracteres na string. Dizemos que uma função retorna um valor quando podemos substituir o texto da função por seu resultado. A função **len** retorna um valor do tipo inteiro, representando a quantidade de caracteres contidos na string. Se a string é vazia (representada simplesmente por "", ou seja, duas aspas sem nada entre elas, nem mesmo espaços em branco), seu tamanho é igual a zero.

Ex.:

```
>>> print (len("A"))
1

>>> print (len(""))
0

>>> print (len("AB"))
2

>>> print (len("Olá, mundo!"))
11
```

LÓGICA DE PROGRAMAÇÃO: Variáveis String

Como dito anteriormente, outra característica de strings é poder acessar seu conteúdo caractere a caractere. Sabendo que uma string tem um determinado tamanho, pode-se acessar seus caracteres utilizando um número inteiro para representar sua posição. Esse número é chamado de **índice**, e começamos a contar de zero. Isso quer dizer que o primeiro caractere da string é de **posição** ou **índice 0**.



LÓGICA DE PROGRAMAÇÃO: Variáveis String

Para acessar os caracteres de uma string, deve-se informar o índice ou posição do caractere entre colchetes ([ ]). Como o primeiro caractere de uma string é o de índice 0, pode-se acessar valores de 0 até o tamanho da string menos 1. Logo, se a string contiver 9 caracteres, pode-se acessar os caracteres de 0 a 8. Se tentarmos acessar um índice maior que a quantidade de caracteres da string, o interpretador emitirá uma mensagem de erro.

Ex.:

```
>>> a = “ABCDEF”
>>> print(a[0])
 A

>>> print(a[1])
B

>>> print(a[5])
F
```

```
>>> print(a[6])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range

>>> print(len(a))
6
```

LÓGICA DE PROGRAMAÇÃO: Variáveis String - Concatenação

O conteúdo de variáveis string podem ser somados, ou melhor, **concatenados**. Para concatenar duas strings, utilizamos o operador de adição (+). Assim, "AB" + "C" é igual a "ABC". Um caso especial de concatenação é a repetição de uma string várias vezes. Para isso, utilizamos o operador de multiplicação (*): "A" * 3 é igual a "AAA".

Ex.:

```
>>> s = “ABC”
>>> print (s + “C”)
ABCC

>>> print (s + “D” * 4)
ABCDDDD

>>> print (“X” + “-”*10 + “X”)
X----------X
```

```
>>> print (s+“x4 = ”+s*4)
ABCx4 = ABCABCABCABC
```

INICIANDO UM ALGORITMO

PYTHON

INICIANDO UM ALGORITMO (PYTHON)

Para iniciar o desenvolvimento de um algoritmo em Python, será utilizado o seguinte exemplo: um algoritmo que receba 2(duas) notas de um aluno, calcule e mostre a média das duas notas.

```
1  nota1 = float(input('Digite a primeira nota: '))
2  nota2 = float(input('Digite a segunda nota: '))
3
4  vrMedia = (nota1 + nota2)/2
5
6  print(vrMedia)
```

Nas linhas 1 e 2 são definidas as variáveis que receberão os valores das duas notas.

nota1 (*indica o nome da variável (identificador)*)

= (*sinal de atribuição (nota1 recebe ...)*)

float (*tipo de dado*)

input (*comando que indica a inserção de um valor digitado pelo usuário atribuindo-o à variável* ***nota1***)

'Digite .... ' (*mensagem a ser exibida para o usuário pelo programa*)

INICIANDO UM ALGORITMO (PYTHON)

Para iniciar o desenvolvimento de um algoritmo em Python, será utilizado o seguinte exemplo: um algoritmo que receba 2(duas) notas de um aluno, calcule e mostre a média das duas notas.

```
1  nota1 = float(input('Digite a primeira nota: '))
2  nota2 = float(input('Digite a segunda nota: '))
3
4  vrMedia = (nota1 + nota2)/2
5
6  print(vrMedia)
```

Na linha 4 é realizada a atribuição do valor da soma de [(*nota1 + nota2)/2]* à variável **vrMedia**.

Na linha 6 é usado o comando **print** (usado para impressão em tela), passando um parâmetro que é a variável que guarda o valor que o usuário quer exibir em tela (neste caso, vrMedia).

INICIANDO UM ALGORITMO (PYTHON)

Na execução do código abaixo:

```python
nota1 = float(input('Digite a primeira nota: '))
nota2 = float(input('Digite a segunda nota: '))

vrMedia = (nota1 + nota2)/2

print(vrMedia)
```

Tem-se:

```
Digite a primeira nota: 8.5
Digite a segunda nota: 7
7.75

Process finished with exit code 0
```

INICIANDO UM ALGORITMO (PYTHON)

Pode-se melhorar o código, fazendo-o exibir uma mensagem final para o usuário, com a média:

```
1  nota1 = float(input('Digite a primeira nota: '))
2  nota2 = float(input('Digite a segunda nota: '))
3
4  vrMedia = (nota1 + nota2)/2
5
6  print('A média é: ', vrMedia)
```

Ao executar o programa acima, tem-se:

```
Digite a primeira nota: 9
Digite a segunda nota: 7.5
A média é:  8.25
```

INICIANDO UM ALGORITMO (PYTHON)

A seguir serão apresentados mais 3 algoritmos resolvidos em Python.

ALGORITMO (PYTHON)

1. Faça um programa que receba duas notas, calcule e mostre a média ponderada dessas notas, considerando peso 2 para a primeira nota e peso 3 para a segunda nota.

```
1   VRPESONOTA1 = 2
2   VRPESONOTA2 = 3
3   VRSOMAPESOS = VRPESONOTA1 + VRPESONOTA2
4
5   nota1 = float(input('Digite a primeira nota: '))
6   nota2 = float(input('Digite a segunda nota: '))
7
8   vrMediaPonderada = ((nota1*VRPESONOTA1) + (nota2*VRPESONOTA2))/VRSOMAPESOS
9
10  print('A média é: ', vrMediaPonderada)
```

Foi usado o conceito de Constante nas linhas 1 a 3.

Na linha 8 foi trabalhado o conceito das prioridades de cálculos matemáticos, separando os termos em parênteses, para que o cálculo não seja feito incorretamente.

ALGORITMO (PYTHON)

Ao executar o exercício 1, tem-se:

```
Digite a primeira nota: 7.5
Digite a segunda nota: 6
A média é:  6.6
```

ALGORITMO (PYTHON)

2. Um funcionário recebe um salário fixo mais 4% de comissão sobre as vendas. Faça um programa que receba o salário fixo de um funcionário e o valor de suas vendas, calcule e mostre a comissão e o salário final do funcionário.

```
1  salFixo = float(input('Digite o salário fixo: '))
2  vrVendas = float(input('Digite o valor das vendas: '))
3
4  vrComissao = vrVendas * 0.04
5  vrSalFinal = salFixo + vrComissao
6
7  print('\nValor da Comissão: R$', vrComissao)
8  print('Salário Final: R$', vrSalFinal)
```

Na linha 4 foi executada a lógica para extrair 4% do vrVendas *(existem outras formas de fazer)*

Na linha 7 foi usado ‘\n’ antes da frase a ser exibida para o usuário. Este comando indica uma quebra de linha, *neste caso antes da mensagem a ser exibida.*

ALGORITMO (PYTHON)

Ao executar o exercício 2, tem-se:

```
Digite o salário fixo: 2750.00
Digite o valor das vendas: 6787.54

Valor da Comissão: R$ 271.5016
Salário Final: R$ 3021.5016
```

Note que os valores de Comissão e Salário Final exibiram 4 casas decimais após a vírgula. Vamos tratar isso? Por padrão, utiliza-se apenas duas casas decimais à direita da vírgula para exibir os centavos. Então será necessário tratar isto no código fonte.

ALGORITMO (PYTHON)

Serão alteradas as linhas 7 e 8:

```
1   salFixo = float(input('Digite o salário fixo: '))
2   vrVendas = float(input('Digite o valor das vendas: '))
3
4   vrComissao = vrVendas * 0.04
5   vrSalFinal = salFixo + vrComissao
6
7   print('\nValor da Comissão: R$ %.2f' %vrComissao)
8   print('Salário Final: R$ %.2f' %vrSalFinal)
```

Foi adicionado ao código, no local onde o valor da variável será exibido, a instrução **%.2f**. Isto indica para o sistema que ele deverá exibir apenas 2 casas decimais após a vírgula. Ao executar, tem-se:

```
Digite o salário fixo: 2750.00
Digite o valor das vendas: 6787.54

Valor da Comissão: R$ 271.50
Salário Final: R$ 3021.50
```

ALGORITMO (PYTHON)

3. Faça um programa que receba o número de horas trabalhadas, o valor do salário mínimo e o número de horas extras trabalhadas. Calcule e mostre o valor total de horas extras e o salário a receber seguindo as regras a seguir:

a) a hora trabalhada vale 1/8 do salário mínimo;
b) a hora extra vale ¼ do salário mínimo;
c) o salário bruto equivale ao número de horas trabalhadas multiplicado pelo valor da hora trabalhada;
d) a quantia a receber pelas horas extras equivale ao número de horas extras trabalhadas multiplicado pelo valor da hora extra;
e) o salário a receber equivale ao salário bruto mais a quantia a receber pelas horas extras.

ALGORITMO (PYTHON)

```python
numHorasTrabalhadas = float(input('Digite o número de horas trabalhadas: '))
vrSalarioMinimo = float(input('Digite o valor do Salário Mínimo: '))
numHorasExtrasTrabalhadas = float(input('Digite o número de horas extras trabalhadas: '))

vrHoraTrabalhada = vrSalarioMinimo/8
vrHoraExtra = vrSalarioMinimo/8
vrSalarioBruto = numHorasTrabalhadas * vrHoraTrabalhada
vrReceberHorasExtras = numHorasExtrasTrabalhadas * vrHoraExtra
vrSalarioReceber = vrSalarioBruto + vrReceberHorasExtras

print('\nValor das Horas Extras: R$ %.2f' %vrReceberHorasExtras)
print('O salário a receber será de: R$ %.2f' %vrSalarioReceber)
```

ALGORITMO (PYTHON)

Ao executar, tem-se:

```
Digite o número de horas trabalhadas: 40
Digite o valor do Salário Mínimo: 500
Digite o número de horas extras trabalhadas: 10

Valor das Horas Extras: R$ 625.00
O salário a receber será de: R$ 3125.00
```

Prof.: João Victor Pereira

ESTRUTURAS DE CONTROLE E REPETIÇÃO

APLICADOS AO PYTHON

COMANDOS DE DECISÃO (PYTHON)

Os comandos de decisão são usados para permitir que o algoritmo possa decidir entre dois ou mais caminhos para continuar a execução. Muitas vezes, serão necessárias diferentes execuções de acordo com resultados obtidos a partir de processamentos anteriores. Neste ponto, os comandos de decisão são primordiais.

Para exemplificar, imagine a situação em que é necessário definir se um aluno foi aprovado ou reprovado em uma matéria, tendo como base a sua média. Neste caso são possíveis duas situações:

a) O aluno está aprovado caso obtenha média igual ou superior a 7(sete);
b) O aluno está reprovado caso obtenha média inferior a 7(sete);

Como é possível observar, dependendo do valor da média, o algoritmo pode realizar uma ação para o caso de aprovação e outra para o caso de reprovação.

Caso seja aprovado, o sistema poderá emitir uma mensagem: “Aluno Aprovado”, caso contrário “Aluno Reprovado”.

Serão apresentados os quatro tipos de decisão possíveis: Seleção Simples, Seleção Composta, Seleção Composta Encadeada e Escolha-Caso.

SELEÇÃO SIMPLES

Uma seleção é definida pelo comando “se” *(if, em python)*, o qual decidirá sobre a execução ou não de um determinado bloco de código com base em uma expressão que deve, obrigatoriamente, retornar um valor lógico.

if *expressão lógica*:
    #Bloco de código

Esta é a estrutura básica de um comando de seleção simples, o qual apenas decidirá se executará o bloco de código ou não. Para isto, é verificado o resultado da expressão lógica, caso seja verdadeiro, executa o bloco de código, caso seja falso, apenas ignora o bloco do **if** e continua a execução do algoritmo.

```
1    valor = int(input('Digite um valor: '))
2
3    if valor > 10:
4        print('O valor informado é maior que 10')
```

SELEÇÃO COMPOSTA

Também trabalha com uma verificação que tem como base uma expressão lógica, porém é adicionada a esta seleção uma opção para o caso de a expressão ser falsa.

if *expressão lógica*:
    #Bloco de código caso a expressão seja verdadeira
else:
    #Bloco de código caso a expressão seja falsa

Na seleção composta, um dos dois blocos de código deverá, obrigatoriamente, ser executado.

```
1   valor = int(input('Digite um valor: '))
2
3   if valor > 10:
4       print('O valor informado é maior que 10')
5
6   else:
7       print('O valor informado é menor ou igual a 10')
```

SELEÇÃO COMPOSTA (OUTRO EXEMPLO)

if *expressão lógica*:
    #Bloco de código caso a 1ª expressão seja verdadeira
elif *expressão lógica:*
    #Bloco de código caso a 2ª expressão seja verdadeira
else:
    #Bloco de código caso as duas expressões anteriores sejam falsas

```
1   valor = int(input('Digite um valor: '))
2
3   if valor < 10:
4       print('O valor informado é menor que 10')
5
6   elif valor < 20:
7       print('O valor informado é maior que 10 e menor que 20')
8
9   else:
10      print('O valor é maior ou igual a 20')
```

SELEÇÃO COMPOSTA ENCADEADA

Usada quando se possui um grande número de verificações, as quais devem ser testadas em sequência até que encontre uma expressão verdadeira. Segue o conceito da seleção composta, porém o encadeamento de seleção se dá com a criação de outros comandos **if** dentro do bloco de código referente ao comando **else**.

```
if expressão lógica:
    #Executa este bloco de código
else:
    if expressão lógica:
        #Executa este bloco de código
    else:
        #Executa este bloco de código
```

Pode-se enxergar uma seleção composta encadeada como uma sequência de verificações à procura da expressão que se encaixa na necessidade do algoritmo, pois sempre haverá uma expressão lógica que será verificada e, caso seja falsa, testará a próxima, e assim por diante.

SELEÇÃO COMPOSTA ENCADEADA

```python
valor = int(input('Digite um valor: '))

if valor < 10:
    print('O valor informado é menor que 10')

else:
    if valor < 20:
        print('O valor informado é maior que 10 e menor que 20')

    else:
        if valor < 30:
            print('O valor informado é maior que 20 e menor que 30')

        else:
            print('O valor é maior ou igual a 30')
```

SELEÇÃO COMPOSTA ENCADEADA

Explicando o exemplo anterior:

No exemplo é solicitado um valor e são realizadas algumas verificações a fim de definir em que intervalo este valor pertence. Para isso, após solicitar o valor, primeiramente verifica-se se o valor é menor que 10(dez). Caso o resultado desta expressão seja verdadeiro, apresenta a mensagem *"O valor informado é menor que 10"* e finaliza a execução da seleção composta encadeada. Verifique que as outras seleções não serão realizadas, pois estão no comando **else *(senão)***, o qual não será atingido caso a verificação do comando **if** seja verdadeira.

Agora, caso o valor não seja menor que 10(dez), entrará no comando **else *(senão)*** e realizará a próxima verificação, que consiste em saber se o valor é menor do que 20. Observe que, se chegar nessa verificação, já sabe-se que o valor não é menor que 10, ou seja, caso a verificação retorne verdadeiro, significa que o valor é menor que 20 e maior ou igual a 10, portanto apresentará a mensagem *"O valor é maior ou igual a 10 e menor que 20"*.

Caso a verificação resulte em falso, o próximo passo será executar o comando **else** desta seleção, o qual possui outra seleção para verificar se o número é menor que 30. Se esta verificação for verdadeira, apresentará a mensagem *"O valor é maior ou igual a 20 e menor que 30"*.

Por fim, pode-se observar a existência apenas de um comando **else**, ou seja, se nenhuma das expressões lógicas executadas até o momento for verdadeira, então entrará neste último comando **else** que não possui mais nenhuma verificação no seu corpo, apenas apresentará a mensagem *"O valor é maior ou igual a 30"*.

ESCOLHA-CASO

Este comando referente à estrutura de seleção é diferente dos anteriores no que diz respeito à nomenclatura. Foram vistos, até o momento, os comandos **if**, **elif** e **else**, já o comando escolha-caso *(**match-case**, no python)*, é usado para situações onde para um determinado valor tem-se um conjunto de opções para escolha.

Também pode ser representado usando a seleção composta encadeada, porém sugere-se que, a partir de uma sequência maior que três seleções encadeadas, utilize-se o comando escolha-caso *(match case)*.

```
match variável:
    case 1:
        #Comandos caso o valor da variável for 1

    case 2:
        #Comandos caso o valor da variável for 2

    case _:
        #Comandos caso nenhuma das opções acima correspondam ao valor da variável
```

ESCOLHA-CASO (MATCH-CASE)

```python
valor = int(input('Digite um valor: '))

match valor:
    case 1:
        print('Um')

    case 2:
        print('Dois')

    case 3:
        print('Três')

    case 4:
        print('Quatro')

    case _:
        print('Valor inválido!')
```

Este exemplo faz a leitura de um valor inteiro e apresenta, como saída, a escrita por extenso, validando os números de 1 a 4. Caso nenhum dos casos seja correspondente, será apresentada a mensagem *"Valor Inválido"*.

Prof.: João Victor Pereira

ESTRUTURAS DE REPETIÇÃO

APLICADAS AO PYTHON

ESTRUTURA DE REPETIÇÃO

A estrutura de repetição é um recurso para o desenvolvimento de tarefas repetitivas em um loop contínuo. O loop funciona até que uma certa condição seja atendida. O loop faz uma iteração *(repetição que realiza uma análise da estrutura aplicada).*

Esta estrutura funciona como um bloco de código que executa uma única operação em todos os dados por n vezes, até que a condição seja satisfeita. Os loops são codificados com as estruturas **FOR** e **WHILE**.

FOR

```
1  num = int(input('Digite um número: '))
2
3  print(f'\nTABUADA DE {num}:')
4
5  for i in range (1, 11): #range(x, y) gera e retorna uma lista de números de x até y, sem incluir o y
6      print(f'{num} x {i} = {num*(i)}')
```

No exemplo acima, foi necessário criar um algoritmo que receba um número e mostre a tabuada deste número. Visto que, em uma tabuada, os números são multiplicados por 1 até 10, será necessário realizar 10 vezes o mesmo processo. Para simplificar, foi usada a estrutura **FOR**.

Na linha 5, *for* ***i*** *(i é o nosso contador, que sempre iniciará por 0 caso não seja definido anteriormente)*, in range *(ou seja, na faixa), (1, 11)* de 1 a < 11. O range retorna uma lista de números de x a y, sem incluir o y. Então, neste caso, o for conta de 1 a 10. *Caso não tenha faixa (início e fim, apenas tamanho), o contador inicia como 0.*

Na 1ª execução, o nosso contador **i** será igual a **1**. Sendo assim, no print (linha 6) chamamos **num*i** *(o número digitado pelo usuário * o valor do contador (que na primeira execução é **1**)*. Após o término, o for é repetido, porém agora o contador recebe incremento *(antes tinha valor 1, agora tem valor 2)* e assim sucessivamente, até que a faixa definida seja totalmente atendida.

FOR

```
Digite um número: 4

TABUADA DE 4:
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
4 x 6 = 24
4 x 7 = 28
4 x 8 = 32
4 x 9 = 36
4 x 10 = 40
```

Como resultado, pode-se ver que o que é exibido em tela, nada mais é que:

print(f'{num} x {i} = {num*(i)}')

... para cada loop da estrutura de repetição (definida com um range (1, 11)) – *lembrando que o range x,y retorna uma lista de x a y, porém não inclui o y*

RANGE

A função **range** pode ser usada para gerar listas simples. Esta função não retorna uma lista propriamente dita, mas um gerador ou *generator*. Por enquanto, basta entender como podemos usá-la. Imagine um programa simples que imprime de 0 a 9 na tela:

```
for i in range(10):
   print(i)
```

A função range gerará números de 0 a 9 porque foi passado 10 como parâmetro. Ela normalmente gera valores a partir de 0, logo, ao especificarmos apenas 10, estamos apenas informando onde parar.

RANGE – FUNÇÃO COM INTERVALOS

A função **range** pode também ser usada indicando qual é o primeiro número a gerar. Para isso, são usados dois parâmetros: início e fim.

```
for i in range(5, 8):
   print(i)
```

Usando 5 como início e 8 como fim, serão impressos os números 5, 6 e 7. A notação aqui para o fim é a mesma utilizada com fatias, ou seja, o fim é um intervalo aberto, isto é, não incluso na faixa de valores.

RANGE – FUNÇÃO COM SALTOS

A função **range** pode também ser usada acrescentando um terceiro parâmetro, onde teremos como saltar entre os valores gerados, por exemplo, ***range(0,10,2)*** gera os pares entre 0 e 10, pois começa de 0 e adiciona 2 a cada elemento. Vejamos um exemplo onde geramos os 10 primeiros múltiplos de 3:

```
for v in range(3, 33, 3):
    print(v, end=" ")
    print()
```

Observe que um gerador como o retornado pela função range não é exatamente uma lista. Embora seja usado de forma parecida, é, na realidade, um objeto de outro tipo.

RANGE – Transformando resultado da função em uma lista

Para transformar um gerador em lista, utilize a função **list**:

```
L = list(range(100, 1100, 50))
print(L)
```

WHILE

A instrução WHILE permite executar um bloco de códigos enquanto uma determinada condição for verdadeira. A instrução contém uma expressão logo no início, que é avaliada a cada loop, e um bloco de códigos que é executado enquanto a condição for verdadeira.

while <condição>:
    <<bloco de código>>

else:
    <<outro bloco de código>>

Normalmente, o laço **while** é utilizado para repetir uma ou várias instruções por determinado número de vezes. Para isso, usa-se uma variável com a função de contador para controlar o número de repetições dos comandos, ou uma expressão para determinar quando a condição for verdadeira ou falsa.

WHILE

```
1  i = 1
2
3  while (i <= 10):
4      print(i)
5      i += 1
```

```
1
2
3
4
5
6
7
8
9
10

Process finished with exit code 0
```

Pode-se notar que foi utilizado um contador **i** para controlar a repetição do loop **while**. Este contador sofre um incremento dentro das instruções do loop. Isso faz com que a condição de avaliação encontre o ponto de parada e encerre o loop. Caso não haja o incremento, o programa seguirá em um loop infinito, buscando sempre pelo **i** até chegar a **10**, porém nunca chegará, visto que sem o incremento, **i** será sempre igual a 1.

LISTAS

Listas são um tipo de variável que permite o armazenamento de vários valores, acessados por um índice. Uma lista pode conter zero ou mais elementos de um mesmo tipo ou de tipos diversos, podendo inclusive conter outras listas. O tamanho de uma lista é igual à quantidade de elementos que ela contém.

Pode-se imaginar uma lista como um edifício de apartamentos, onde o térreo é o andar zero, o primeiro andar é o andar 1 e assim por diante. O índice é utilizado para especificar o “apartamento” onde serão guardados os dados. Em um prédio de seis andares, teremos números de andar variando entre 0 e 5. Se chamarmos nosso prédio de P, teremos P[0] como o endereço do térreo, P[1] como endereço do primeiro andar, continuando assim até P[5]. Em Python, P seria o nome da lista; e o número entre colchetes, o índice.

Listas são mais flexíveis que prédios e podem crescer ou diminuir com o tempo.

LISTAS – Lista Vazia

L = [ ]

Essa linha cria uma lista chamada L com zero elemento, ou seja, uma lista vazia. Os colchetes ([ ]) após o símbolo de igualdade servem para indicar que L é uma lista.

LISTAS – Lista com Três Elementos

Z = [15 , 8 , 9]

A lista Z foi criada com três elementos: 15, 8 e 9. Dizemos que o tamanho da lista Z é 3. Como o primeiro elemento tem índice 0, temos que o último elemento é Z[2].

LISTAS – Acesso a uma Lista

```
>>> Z = [ 15 ,  8 ,  9 ]
>>> Z[0]
15

>>> Z[1]
8

>>> Z[2]
9
```

LISTAS – Modificação de uma Lista

Utilizando o nome da lista e um índice, pode-se mudar o conteúdo de um elemento.

```
>>> Z = [ 15 ,   8 ,   9 ]
>>> Z[0]
15

>>> Z[0] = 7
>>> Z[0]
7

>>> Z
[7,  8,  9]
```

Quando a lista Z foi criada, o primeiro elemento era o número 15. Por isso, Z[0] era 15. Quando executamos Z[0]=7, alteramos o conteúdo do primeiro elemento para 7. Isso pode ser verificado quando pedimos para exibir Z, agora com 7, 8 e 9 como elementos.

LISTAS – Adição de Elementos

Uma das principais vantagens de trabalharmos com listas é poder adicionar novos elementos durante a execução do programa. Para adicionar um elemento ao fim da lista, utiliza-se o método **append**. Em Python, chamamos um método escrevendo o nome dele após o nome do objeto.

Como listas são objetos, sendo L a lista, teremos **L.append(valor)**.

```
>>> L = [ ]
>>> L.append(“a”)
>>> L
[‘a’]
>>> L.append(“b”)
>>> L
[‘a’, ‘b’]
>>> L.append(“c”)
>>> L
[‘a’, ‘b’, ‘c’]
>>> len(L)
3
```

LISTAS – Remoção de Elementos

Como o tamanho da lista pode variar, permitindo a adição de novos elementos, podemos também retirar alguns elementos da lista, ou mesmo todos eles. Para isso, utilizaremos a instrução **del**.

```
>>> L = [“a”, “b”, “c”]
>>> del L[1]
>>> L
[‘a’, ‘c’]
>>> del L[0]
>>> L
[‘c’]
```

É importante notar que o elemento excluído não ocupa mais lugar na lista, fazendo com que os índices sejam reorganizados, ou melhor, que passem a ser calculados sem esse elemento.

Podemos também apagar fatias inteiras de uma só vez.

```
>>> L=list(range(101))
>>> del L[1:99]
```

```
>>> L
[0, 99, 100]
```

DICIONÁRIOS

Dicionários consistem em uma estrutura de dados similar às listas, mas com propriedades de acesso diferentes. Um dicionário é composto por **um conjunto de chaves e valores**. O dicionário em si consiste em relacionar uma chave a um valor específico.

Em Python, os dicionário são criados utilizando chaves ({}). Cada elemento do dicionário é uma combinação de chave e valor.

Produto	Preço
Alface	R$ 0,45
Batata	R$ 1,20
Tomate	R$ 2,30
Feijão	R$ 1,50

```
tabela = { "Alface": 0.45,
           "Batata": 1.20,
           "Tomate": 2.30,
           "Feijão": 1.50 }
```

DICIONÁRIOS - Criação

Um dicionário é acessado por suas chaves. Para obter o preço da alface, digite no interpretador, depois de ter criado a tabela, **`tabela["Alface"]`**, onde tabela é o nome da variável do tipo dicionário, e "Alface" é nossa chave. O valor retornado é o mesmo que associamos na tabela, ou seja, *0.45*.

Diferentemente de listas, onde o índice é um número, dicionários utilizam suas chaves como índice. Quando atribuímos um valor a uma chave, duas coisas podem ocorrer:

1. Se a chave já existe: o valor associado é alterado para o novo valor.
2. Se a chave não existe: a nova chave será adicionada ao dicionário.

DICIONÁRIOS - Funcionamento

```
>>> tabela = { "Alface": 0.45,
               "Batata": 1.20,
               "Tomate": 2.30,
               "Feijão": 1.50 }

>>> print(tabela["Tomate"])
2.3
>>> print(tabela)
{'Batata': 1.2, 'Alface': 0.45, 'Tomate': 2.3, 'Feijão': 1.5}

>>> tabela["Tomate"] = 2.50
>>> print(tabela["Tomate"])
2.5

>>> tabela["Cebola"] = 1.20
>>> print(tabela)
{'Batata': 1.2, 'Alface': 0.45, 'Tomate': 2.5, 'Cebola': 1.2, 'Feijão': 1.5}
```

❶ ***acessando o valor associado à chave "Tomate"***

❷ ***alterando o valor associado à chave "Tomate" para um novo valor***

❸ ***criando uma nova chave "Cebola", que é adicionada ao dicionário***

DICIONÁRIOS – Acesso a uma chave inexistente

```
>>> tabela = { "Alface": 0.45,
               "Batata": 1.20,
               "Tomate": 2.30,
               "Feijão": 1.50 }

>>> print(tabela[“Manga"])
Traceback (most recente call last):
    File “<stdin>”, line 1, in <module>
KeyError: ‘Manga’
```

Se a chave não existir, uma exceção do tipo KeyError será ativada.

DICIONÁRIOS – Verificação de existência de uma chave

```
>>> tabela = { "Alface": 0.45,
               "Batata": 1.20,
               "Tomate": 2.30,
               "Feijão": 1.50 }

>>> print("Manga‟ in tabela)
False

>>> print("Batata" in tabela)
True
```

Se a chave não existir, uma exceção do tipo KeyError será ativada.

DICIONÁRIOS – Aninhamento

Há casos onde será necessário aplicar o conceito do que chamamos de **Dicionários Aninhados**, que nada mais é que um dicionário dentro de outro dicionário, ou seja, um dicionário onde cada valor é, por sua vez, outro dicionário, partindo de um dicionário principal, onde **cada chave está associada a um valor que é um dicionário interno**.

Ex. prático apenas do dicionário:

ESPECIFICAÇÃO	CÓDIGO	PREÇO
Cachorro quente	100	R$ 1,20
Bauru simples	101	R$ 1,30
Bauru com ovo	102	R$ 1,50
Hambúrguer	103	R$ 1,20
Cheeseburguer	104	R$ 1,30
Refrigerante	105	R$ 1,00

Faça um programa que leia o código dos itens pedidos e as quantidades desejadas. Calcule e mostre o valor a ser pago por item (preço * quantidade) e o total geral do pedido. Considere que o cliente deve informar quando o pedido deve ser encerrado.

DICIONÁRIOS – Exemplo de Dicionário Aninhado

```
cardapio = {
    100: {"especificacao": "Cachorro quente", "preco": 1.20},
    101: {"especificacao": "Bauru Simples", "preco": 1.30},
    102: {"especificacao": "Bauru com Ovo", "preco": 1.50},
    103: {"especificacao": "Hambúrguer", "preco": 1.20},
    104: {"especificacao": "Cheeseburguer", "preco": 1.30},
    105: {"especificacao": "Refrigerante", "preco": 1.00}
}
```

Ex.: a chave **100** está associada a um dicionário interno que contém:

```
"especificação": "Cachorro quente"
"preco": 1.20
```