OPERADORES ARITIMÉRICOS

+ ADIÇÃO

- SUBTRAÇÃO

***** MULTIPLICAÇÃO

/ DIVISÃO

// DIVISÃO INTEIRA

% RESTO DA DIVISÃO

****** POTÊNCIA

== IGUAL

ORDEM DE PROCEDÊNCIA

1ª: **()** 2ª: ****** 3ª: *****, **/**, **%**, **//** 4ª: **+**, **-**

CONTROLE DE *STRING*

\n Quebra de linha(adicionar ao começo de uma string).

, end='' Junta o próximo texto ao anterior, separando-os com a mensagem entre as aspas.

> Fatiamento:

O fatiamento serve para que em um texto, por exemplo, seja mostrado apenas uma parte específica, como exemplo, o texto:

Código de exemplo:

```
texto = ('Eu te amo, só que não!')
print(texto[0:9])
```

Resultado:

Eu te amo

Para fatiar é necessário utilizar os dois pontos (:) entre dois números, sendo o primeiro o começo do fatiamento, e o segundo o número da posição do número posterior ao do fim do fatiamento. Lembrando que as posições começam a partir do 0, sendo assim o 1 é a segunda posição. Se colocado apenas o começo, e depois deixar vazio ao final dos dois pontos (:), ele pegará do começo até o fim do texto, e vice-versa. Caso colocado apenas os dois pontos (:), irá fatiar do começo ao fim, uma cópia perfeita.

É possível fazer um fatiamento saltando posições, para isso é necessário colocar mais um dois pontos (:) e mais um número, sendo esse número a quantidade de saltos de uma posição à outra.

Código de exemplo:

```
texto = 'Eu te amo, só que não!'
print(texto[::2])
```

Resultado:

E eao óqeno

> Formatação de Uma *String*:

Formatar uma *string* é fundamental caso queira utilizar variáveis no texto a ser mostrado com o comando *print*, há duas formas principais de fazer isso em Python, com as *F string 's* ou com o *format.*

- ***F String 's :***

Utilizando a letra 'f' antes das aspas de uma *string* em um comando *print*, a string já estará formatado, assim, variáveis podem ser colocadas dentro de chaves ({ }) para serem mostradas ao executar o comando.

Código de exemplo:

```
numero1 = 2
numero2 = 3
soma = numero1 + numero2
print(f'{numero1} + {numero2} = {soma}')
```

Resultado:

2 + 3 = 5

- ***.format() :***

Deixando as chaves os pontos em que vão ser escritas as variáveis ou comandos desejados, é só colocar, à um espaço antes do último parêntese do comando *print*, o comando *.format()*, e entre esses parênteses, colocar, em ordem, as variáveis/comandos que deverão aparecer.

Código de exemplo:

```
numero1 = 2
numero2 = 3
soma = numero1 + numero2
print('A soma entre {} e {} é igual a {}!'.format(numero1,
numero2, soma))
```

Resultado:

A soma entre 2 e 3 é igual a 5!

Para formatar uma variável, para por exemplo se centralizar ao meio de uma quantidade de caracteres, é possível utilizar os dois pontos (:), mesmo não sendo um fatiamento. Funciona da seguinte maneira, ao final de uma variável/comando em uma F String, coloque *:10*, sendo o 10 o número de caracteres em que a mensagem deve ser centralizada, podendo ser personalizada de acordo com a necessidade.

< começo da linha

> final da linha

^ centro da linha

Ordem para formatação: : 1ª *caracteres para preencher os espaços (não colocar nada será apenas espaço*

2ª *se a mensagem será ao começo, ao fim, ou ao centro dos espaços*

3ª *número de espaços para se preencher*

Código de exemplo:

```
print(f'{" Dinamite!!! ":-^25}')
```

Resultado:

------ Dinamite!!! ------

Para o formatar basta colocar a formatação dentro das chaves vazias.

> Manipulação de *String* :

Existem vários comandos úteis que ajudam a manipular uma *string*:

- ***.count() *contador de caracteres/trechos*:***

Usado para contar uma palavra ou caractere específico dentro de uma *String*. Ele também pode ser fatiado, mas com vírgula (,) ao invés dos dois pontos (:), e sem a função de passo.

Código de exemplo:

```
frase = 'a sombra das ruínas paira acima...'
```

Resultado:

```
8
```

O resultado 8 é a soma das letras 'a' na *string*. Esse comando é sensível a letras maiúsculas, minúsculas e acentos.

- ***.find() / rfind() *encontrar na string*:***

Usado para encontrar um trecho específico em um texto.

Código de exemplo:

```
frase = 'Eu amo chocolate'
print(frase.find('c'))
```

Resultado:

```
7
```

O resultado é a primeira posição em que a letra 'c' aparece na *string*. Caso queira a busca começando pelo lado direito, utilize o ***.rfind()***.

- ***.replace() *substituição*:***

Código de exemplo:

```
frase = 'Você está com problemas, que pena...'
print(frase.replace('Você', 'Ele'))
```

Resultado:

Ele está com problemas, que pena...

O primeiro argumento é o que está para ser substituído e o segundo é o novo (substitui para toda a string).

- ***.upper()* *tornar tudo maiúsculo*:**

Transforma todos os caracteres da *string* em maiúsculo.

Comando:

****string*.upper()***

- ***.lower()* *tonar tudo em minúsculo*:**

Transforma todos os caracteres da *string* em minúsculo.

Comando:

****string*.lower()***

- ***.capitalize()* *primeira letra da *string* em maiúsculo*:**

Transforma apenas a primeira letra de uma *string* em maiúsculo.

Comando:

****string*.capitalize()***

- ***.title()* *primeira letra de cada frase em maiúsculo*:**

Transforma a primeira letra de todas as palavras em maiúsculo, como em nomes próprios.

Comando:

****string*.title()***

- ***.strip()* *remover espaços ao começo e/ou ao fim de uma *string*:***.

> ****frase*.strip()***: exclui espaços ao **começo e ao fim** de uma *string*.

> ****frase*.rstrip()***: exclui espaços apenas do lado **direito** de uma *string*.

> ****frase*.rstrip()***: exclui espaços apenas do lado **esquerdo** de uma *string*.

- ***.split()* *separa cada palavra dentro de uma única lista*:**

Irá dividir uma *string* colocando-a dentro de uma única lista.

Comando:

frase.split()

- ***.join()* *juntar trechos separados de uma lista*:**

Irá fazer o oposto do *split()*, juntando os trechos de uma lista ou tupla. Deverá colocar aspas antes do ponto (.), e entre elas será o divisor dos trechos, onde você escolhe, podendo ser qualquer caractere, mas geralmente é usado o espaço. Entre os parênteses deve ficar a lista/tupla.

Comando:

' '.join()

- ***len() *número de caracteres na string*:***

A resposta dessa função irá retornar o número de caracteres de uma *string*. Dentro dos parênteses é o lugar para a *string*.

Comando:

len(*variável*)

- ***in *existe dentro da string*:***

A resposta deste operador será ***True***, caso o trecho exista dentro da *string*, ou ***False***, em caso contrário. Entre as primeiras aspas deve conter o trecho a ser achado na *string*, em seguida o operador e por último a *string* onde deverá procurar o trecho.

Comando:

' ' in *string*

CONDIÇÕES

As condições são basicamente possibilidades. Em uma condição, ou o bloco verdadeiro é executado ou o bloco falso. Sendo assim, apenas um bloco pode ser executado, o que estiver em condição.

- ***.if, elif, else :***

O ***if***, ou "se", é a primeira condição, se sua condição for verdadeira, ele executará tudo que está em seu bloco, caso seja falso, ele passará para uma segunda condição, caso exista no código.

O ***elif***, ou "se não", funciona da mesma forma que o *if*, mas executando apenas se o *if* for falso inicialmente.

O ***else***, ou "senão", é o que sobra, seu bloco executará caso o *if* e, caso exista, o *elif* forem falsos.

Código de exemplo:

```
nota = float(input('Qual foi a sua nota? '))
if nota <5:
    print('Nada bom, melhor estudar!')
else:
    print('Boa, continue assim!')
```

Resultado 1:

Qual foi a sua nota? 7

Boa, continue assim!

Resultado 2:

Qual foi a sua nota? 3

Nada bom, melhor estudar!

Outro exemplo, mas também com o ***elif***:

```
nota = float(input('Qual foi a sua nota? '))
if nota <5:
    print('Nada bom, melhor estudar!')
elif nota >= 5 and nota < 7:
    print('Boa, continue assim!')
else:
    print('Excelente! Sua nota está ótima!')
```

Resultado 1:

Qual foi a sua nota? 2

Nada bom, melhor estudar!

Resultado 2:

Qual foi a sua nota? 6

Boa, continue assim!

Resultado 3:

Qual foi a sua nota? 10

Excelente! Sua nota está ótima!

Como pode ver acima, existe alguns sinais como **>** (maior que), **<** (menor que) e **=** (igual), usados para condições que tenham número como base, como as dos exemplos anteriores. Caso tenha dois sinais desses juntos, como **>=** ou **<=**, significam, respectivamente, maior ou igual e menor ou igual.

Também foi visto o operador ***and***, ou "e", que faz com precise de mais de uma condição para que o bloco seja executado.

Existe também o operador ***or***, ou "ou", que faz com que o bloco execute com uma condição ou outra.

REPETIÇÕES

- ***for :***

As repetições servem para repetir (obviamente) tudo dentro de seu alinhamento em uma quantidade específica de vezes.

Código de exemplo:

```
for c in range(1, 10):
    print('Olá!', end=' ')
print()
```

Resultado:

Olá! Olá! Olá! Olá! Olá! Olá! Olá! Olá! Olá! Olá!

Com o código acima, o programa irá escrever "Olá!" 9 vezes, pois o primeiro número é o início, o segundo é o fim, sendo que ele é desconsiderado e contado o seu antecessor, ou seja, 10 será 9, 11 será 10 etc. E se caso o terceiro número, o de passo, for negativo, ele contará de forma decrescente, ou seja, -1 ele contará para trás pulando de 1 em 1, -2 contará para trás pulando de 2 em 2, e assim por diante.

Código de exemplo:

```
for c in range(1, 11, 2):
    print(c, end=' ')
print()
```

Resultado:

1 3 5 7 9

- ***range():***

O ***range*** foi usado nos exemplos anteriores, para repetições, mas é importante lembrar que ele pode ser usado para contagem, mesmo fora de uma repetição. O resultado será os números, separados por um espaço entre eles. Sendo assim, o ***range*** pode ser usado para gerar listas ou tuplas com vários números.

- ***for in *lista/tupla*:***

Caso usado o Caso usado o ***for*** em uma lista, ele fará uma repetição para cada item dentro de uma lista/tupla.

Código de exemplo:

```
l = list(['a', 'b', 'c', 'd'])
for v in l:
    print(v, end=' ')
print()
```

Resultado:

a b c d

- ***while:***

O ***while*** bem mais versátil que a repetição *for*, pois é possível utilizá-la em situações onde não se sabe a quantidade de repetições, ao contrário do *for*, que só pode ser usado em situações em que o limite de repetições é evidente. O ***while*** funciona como as condições.

Código de exemplo:

```
c = 0
while c < 10:
    c = c + 1
    print(c, end=' ')
print()
```

Resultado:

1 2 3 4 5 6 7 8 9 10

A cada repetição a variável **c** irá somar mais 1, ou seja, na segunda repetição a variável irá se tornar 2, na terceira, será 3, e assim por diante até o limite dado ao ***while***, que é 10, nesse caso.

Outro exemplo é:

Código de exemplo:

```
condicao = True
c = 0
while condicao == True:
    c = c + 1
    print(c, end=' ')
    if c == 10:
        condicao = False
print()
```

Resultado:

1 2 3 4 5 6 7 8 9 10

No exemplo acima, a repetição só será interrompida quando o valor da variável *condicao* se tornar falso.

- **Repetições Infinitas*:***

Código de exemplo:

```
c = 0
while True:
    c = c + 1
    print(c, end=' ')
    if c == 10:
        break
print()
```

Resultado:

1 2 3 4 5 6 7 8 9 10

Com o comando ***while True***, o bloco irá se repetir infinitamente e a única forma de interromper é usando o comando *break*, mostrado no exemplo acima.

O comando ***continue*** é usado para retornar ao começo de uma repetição *while*, independentemente de onde estiver no momento. (Nota: ao pesquisar esse comando, vi que o mesmo tem a função de apenas continuar a execução do bloco, mas utilizei ele várias vezes para voltar ao começo da repetição.)

TUPLAS

As **tuplas** servem para armazenar mais de um item em uma única variável. As **tuplas** são representadas por parenteses "**()**", ou seja, os itens devem estar dentro deles. Pode ser declarada por ***tuple()***.

Exemplo:

```
tupla = ('Dinossauro', 'Abelha', 'Cachorro')
          0            1          2
```

Para acessar um item dentro de uma **tupla** é bem simples. Cada item da **tupla** é numerado, começando do 0, assim como mostrado acima.

Para acessar o terceiro item dessa **tupla**, por exemplo:

Código de exemplo:

```
tupla = ('Dinossauro', 'Abelha', 'Cachorro')
print(tupla[2])
```

Resultado:

```
Cachorro
```

As **tuplas** são **IMUTÁVEIS**, ou seja, não é possível substituir um item dentro dela, ou excluí-lo. Para fazer isso seria melhor utilizar listas, que são mutáveis.

Mesmo que as **tuplas** não sejam mutáveis, é possível organizá-las para mostrar na tela, utilizando o comando *sorted*. Essa organização terá como base a ordem alfabética dos item da lista, ou sua ordem numérica.

Código de exemplo:

```
tupla = ('Dinossauro', 'Abelha', 'Cachorro')
print(sorted(tupla))
```

Resultado:

['Abelha', 'Cachorro', 'Dinossauro']

É importante lembrar que o resultado é uma lista, como já mostrado acima, em que os itens estão entre colchetes "[]".

Para organizá-la ao contrário, utilize o parâmetro ***reverse=True*** após a váriável e uma vírgula.

Código de exemplo:

```
tupla = ('Dinossauro', 'Abelha', 'Cachorro')
print(sorted(tupla, reverse=True))
```

Resultado:

['Dinossauro', 'Cachorro', 'Abelha']

LISTAS

Assim como as tuplas podem ser usadas para armazenar mais de um item em uma única variável, com a diferença de que as listas **SÃO MUTÁVEIS**. Deve ser usado colchetes **[]** para declarar uma lista, ou ***list()***.

> Adicionar item na lista:

- .append()

No caso acima, o "Gato" será incluso na lista na última posição da **lista**.

- .insert()

O *insert* serve para inserir um item a uma posição específica.

Código de exemplo:

```
lista = ['Futebol', 'Basquete', 'Vôlei', 'Futsal', 'Natação']
lista.insert(2, 'Videogame')
print(lista)
```

Resultado:

['Futebol', 'Basquete', 'Videogame', 'Vôlei', 'Futsal', 'Natação']

0 1 2 3 4 5

Como visto acima, ele inseriu o "Videogame" na posição 2.

> Remover item da lista:

- ***.remove():***

Para remover um item específico pelo nome, use o ***lista*.remove(*nome do item*)**.

Código de exemplo:

```
lista = ['Batata', 'Limão', 'Maçã', 'Cenoura', 'Acerola']
lista.remove('Limão')
print(lista)
```

Resultado:

['Batata', 'Maçã', 'Cenoura', 'Acerola']

Caso tente remover um item que não existe dentro de uma lista, o programa será encerrado com um erro. Nesse caso utilize o operador ***in*** para verificar se existe ou não o item dentro da lista.

Código de exemplo:

```
lista = ['Batata', 'Limão', 'Maçã', 'Cenoura', 'Acerola']
if 'Limão' in lista:
    lista.remove('Limão')
print(lista)
```

Resultado:

['Batata', 'Maçã', 'Cenoura', 'Acerola']

O ***remove*** deleta o primeiro valor digitado na lista, ou seja, se na lista existir um mesmo item 2 vezes ou mais, ele irá remover apenas o primeiro.

- ***del:***

O operador ***del*** exclui um item de uma lista a partir de sua posição.

Código de exemplo:

```
lista = ['Batata', 'Limão', 'Maçã', 'Cenoura', 'Acerola']
del lista[3]
print(lista)
```

Resultado:

['Batata', 'Limão', 'Maçã', 'Acerola']

O programa será encerrado com um erro caso seja especificado o valor de uma posição inexistente dentro da lista ou a própria lista, caso a mesma tenha sido excluída com o comando.

- ***.pop():***

Utilizando o função ***pop***, poderá apagar uma item específico a partir de sua posição na lista, assim como o *del*. Se utilizar apenas ***pop()***, irá apagar o último item na lista, caso utilize ***pop(pos)***, sendo "*pos*" a posição do item, irá apagar o item especificado.

> Limpar uma lista:

- .clear()

Utilizando a função ***clear()***, poderá limpar uma lista, deixando-a em branco.

> Organizar uma lista:

- .sort()

Usando a função ***sort()***, é possível organizar uma lista em ordem numérica ou alfabética.

Código de exemplo:

```
lista = [4, 5, 2, 1, 3]
lista.sort()
print(lista)
```

Resultado:

[1, 2, 3, 4, 5]

Para colocar em ordem contrária, use o ***sorted(reverse=True)***.

Código de exemplo:

```
lista = [4, 5, 2, 1, 3]
lista.sort(reverse=True)
print(lista)
```

Resultado:

[5, 4, 3, 2, 1]

> Declarar uma lista com *list()*:

Uma lista pode ser declarado utilizando o *range*, da seguinte forma:

Código de exemplo:

```
lista = list(range(1, 11))
print(lista)
```

Resultado:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

> Duplicar uma lista:

Caso queira duplicar uma lista, usar apenas *copia = lista* não irá funcionar. A "nova" lista ficaria ligada à primeira e por isso elas são a mesma lista, só que com duas variáveis. Para duplicar corretamente, use o fatiamento.

Código de exemplo:

```
l1 = [0, 2, 4, 6, 8]
l2 = l1[:]
l2[2] = 5
print(f'Lista 1: {l1}')
print(f'Lista 2: {l2}')
```

Resultado:

Lista 1: [0, 2, 4, 6, 8]

Lista 2: [0, 2, 5, 6, 8]

> Lista dentro de outra lista:

É possível colocar uma ou mais listas dentro de uma outra lista, o exemplo é uma perfeita explicação:

```
pessoas = [['Carlos', 39], ['Luiz', 21], ['Luana', 19]]
```

Sendo: [['Carlos', 39], ['Luiz', 21], ['Luana', 19]]

0 1 0 1 0 1

0 1 2

***Prints*:**

print(pessoas[0][0]) > Carlos

print(pessoas[1][1]) > 21

print(pessoas[2][0]) > Luana

print(pessoas[2]) > ['Luana', 19]

print(pessoas) > [['Carlos', 39], ['Luiz', 21], ['Luana', 19]]

> enumerate:

Use o ***enumerate*** para numerar as posições de itens em uma **tupla** ou **lista**, por exemplo.

Código de exemplo:

```
tupla = ('League of Legends', 'Terraria', 'Minecraft', 'RimWorld',
'GTA')
for pos, c in enumerate(tupla):
    if pos == len(tupla) - 1:
        print(f'{pos+1} > {c}', end='')
    else:
        print(f'{pos+1} > {c}', end=' | ')
print()
```

Resultado:

1 > League of Legends | 2 > Terraria | 3 > Minecraft | 4 > RimWorld | 5 > GTA

> Maior e menor valor (Tupla e Lista):

- ***max():***

Mostra o **maior** valor de uma lista ou tupla.

- ***min():***

Mostra o **menor** valor de uma lista ou tupla

DICIONÁRIOS

Os dicionários são representados por chaves **{}** . Um dicionário pode ser declarado como **{}** ou como ***dict()***. Os dicionários podem fazer parte de listas ou tuplas.

Em um dicionário existe as chaves(*keys*), os valores(*values*) e os itens(*items*). Para mostrar o que é cada um deles, considere o dicionário:

```
dicionario = {'nome': 'Luana', 'idade': 22, 'sexo': 'Feminino'}
```

- ***Chaves(keys):***

As **chaves** são os nomes relacionados à seus valores. No caso acima, são "nome", "idade" e "sexo". Para mostrá-los em Python use o ***dicionario*.*keys()*.**

Resultado:

dict_keys(['nome', 'idade', 'sexo'])

- ***Valores(values):***

Os **valores** são "Luana", 22 e "Feminino". Para mostrá-los em Python use o ***dicionario*.*values()*.**

dict_values(['Luana', 22, 'Feminino'])

Resultado:

dict_values(['Luana', 22, 'Feminino'])

- ***Itens(items):***

São todos os itens dentro de um dicionário, cada um separado em uma tupla. Para mostrá-los em Python use o ***dicionario*.*items()*.**

Resultado:

dict_items([('nome', 'Luana'), ('idade', 22), ('sexo', 'Feminino')])

O comando também pode ser utilizado para repetições *for*, de forma parecida com o *enumerate*.

Código de exemplo:

```
dicionario = {'nome': 'Luana', 'idade': 22, 'sexo': 'Feminino'}
for k, v in dicionario.items():
    print(f'{k} = {v}')
```

Resultado:

nome = Luana

idade = 22

sexo = Feminino

> Apagar chave:

Para apagar chaves, use o comando já visto ***del***. Como exemplo:

```
dicionario = {'nome': 'Luana', 'idade': 22, 'sexo': 'Feminino'}
del dicionario['sexo']
for k, v in dicionario.items():
    print(f'{k} = {v}')
```

> Adicionar/Substituiar uma chave:

Para **adicionar**, basta criar uma variável do dicionário com uma chave inexistente.

Código de exemplo:

```
dicionario = {'nome': 'Luana', 'idade': 22, 'sexo': 'Feminino'}
dicionario['emprego'] = 'Programadora'
for k, v in dicionario.items():
    print(f'{k} = {v}')
```

Resultado:

nome = Luana

idade = 22

sexo = Feminino

emprego = Programadora

Para **substituir** basta trocar o valor declarando uma nova variável.

Código de exemplo:

```
dicionario = {'nome': 'Luana', 'idade': 22, 'sexo': 'Feminino'}
dicionario['nome'] = 'Layla'
for k, v in dicionario.items():
    print(f'{k} = {v}')
```

Resultado:

nome = Layla

idade = 22

sexo = Feminino

> Copiar Dicionário:

Para cópia, existe um método interno, o ***.copy()***.

FUNÇÕES

Funções são aqueles comandos, sejam internos ou externos, que terminam com parênteses fechados, como exemplo, os comandos *print()*, *int()*, *str()*.

- ***def* (sem parâmetro):**

Resumindo, ***def***'s são rotinas. Caso precise utilizar o mesmo comando várias vezes em um único código, você pode criar um "atalho" para ele utilizando o ***def***.

Código de exemplo:

```
def linha():
    print('=-~=' * 6)


linha()
print(' Cadastro de Pessoas ')
linha()
```

Resultado:

```
=-~=-~=-~=-~=-~=-~=

 Cadastro de Pessoas

=-~=-~=-~=-~=-~=-~=
```

Entre o fim do ***def*** e o começo do programa ativo têm duas linhas separando-os. Isso é uma questão de estética e não é fundamental, mas é bem útil para organizar o programa.

- ***def* (com parâmetro):**

Mas supondo que fosse necessário escrever várias coisas ao meio do código, como além de 'Cadastro de Pessoas', também escrever 'Validação de Dados' e 'Idantificação', o ***def*** poderia ser utilizado de outra maneira.

Código de exemplo:

```
def texto(txt):
    print('<' * 30)
    print(txt)
    print('>' * 30)


texto('     Cadastro de Pessoas')
texto('     Validação de Dados')
texto('     Idantificação')
```

Resultado:

```
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
   Cadastro de Pessoas
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
   Validação de Dados
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
   Idantificação
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
```

Como visto, o ***txt*** recebe a mensagem atribuída abaixo, podendo ser pelo próprio usuário com o ***input()***. O parâmetro nessa função é justamente o ***txt***, atribuído no comando principal abaixo das duas linhas de separação do ***def***.

- ***Múltiplos parâmetros:***

É possível que você queira mais de um **parâmetro** em uma função, pensando em uma situação de multiplicação, você precisa multiplicar dois números dados pelo usuário. Veja como seria:

Código de exemplo:

```
def mult(a, b):
    multiplicacao = a * b
    print(f'--- {a} x {b}: {multiplicacao} ---')


num1 = int(input('Primeiro valor: '))
num2 = int(input('Segundo valor:  '))
mult(num1, num2)
```

Resultado:

Primeiro valor: 5

Segundo valor: 10

--- 5 x 10: 50 ---

Você pode especificar cada **parâmetro**, colocando antes da variável/valor o **parâmetro** desejado, por exemplo: (**a** = nums2, **b** = num1). Nesse caso é necessário especificar todos os parâmetros, ou ocorrerá um erro ao executar o programa.

Contudo, isso tem um problema... E se precisássemos utilizar mais de um **parâmetro**, sendo assim, multiplicar 3 valores entre si. Da forma abaixo por exemplo:

mult(2, 5)

mult(3, 2, 7)

mult(6)

Utilizando o mesmo ***def*** anterior, o programa se encerraria com um ERRO, e o motivo é que existem apenas 2 **parâmetros** a serem multiplicados, limitando assim, o programa. Nesse caso, existe uma ferramenta chamada **Empacotamento** (empacotar).

> Empacotar (*def*):

No mesmo exemplo acima, se encontram **parâmetros** de tamanhos diferentes, nesse caso podemos utilizar o símbolo " * " antes do parâmetro, dizendo assim ao Python, que o número de **parâmetros** será indeterminado, podendo ser um ou vários.

Código de exemplo:

```
def mult(*num):
    s = 1
    for n in num:
        s = s * n
    print(f'--- Multiplição dos valores {num}: {s}')


mult(2, 5)
mult(3, 2, 7)
mult(6)
```

Resultado:

--- Multiplição dos valores (2, 5): 10

--- Multiplição dos valores (3, 2, 7): 42

--- Multiplição dos valores (6,): 6

> Parâmetros Opcionais:

Caso um **parâmetro** tenha, por exemplo: **c** = 0, significa que caso não seja especificado o valor de **c**, o programa vai continuar e o valor de **c** será 0.

Código de exemplo:

```
def somar(a, b, c = 0):
    s = a + b + c
    print(f' - Valores: {a}, {b}, {c} | Soma: {s} - ')


somar(2, 5, 7)
somar(1, 22)
```

Resultado:

- Valores: 2, 5, 7 | Soma: 14 -

- Valores: 1, 22, 0 | Soma: 23 –

> Escopo de Variáveis:

- ### Variável Local*:*

Uma variável local é uma variável que funciona apenas na função(**def**) em que ela foi declarada, não funcionando fora a essa função.

- ### Variável Global*:*

Uma variável global pode ser usada em todo o programa, em uma função ou fora dela. Uma variável global é declarada no código externo, fora de uma função, mas também é possível declarar uma variável global dentro de uma função utilizando o comando ***global*** ***variável*** antes de declarar uma variável.

Código de exemplo:

```
def text():
    global txt
    txt = 'FIM!'
for c in range(1, 10):
    print(f'{c}', end=' ')
text()
print(txt)
```

Resultado:

1 2 3 4 5 6 7 8 9 FIM!

➢ Retorno de Valores:

Com o ***return*** ***variável***, é possível retornar o valor de uma **variável** presente dentro de uma função para fora dela. Ele só retornará o valor se a função tiver sido executada em uma variável, como abaixo no exemplo abaixo:

Código de exemplo:

```
def funcao(a=0, b=0, c=0):
    soma = a + b + c
    return soma


r1 = funcao(2, 5, 7)
r2 = funcao(1, 3)
r3 = funcao(9)
print(f'As somas dos valores, respectivamente, são: {r1}, {r2},
{r3}')
```

Resultado:

As somas dos valores, respectivamente, são: 14, 4, 9

No caso acima, o ***return*** retornou à variável **r1** o valor da soma dos 3 **parâmetros**, e o mesmo com o **r2** e o **r3**.

Como o ***return*** é possível retornar qualquer valor declarado a uma variável, inclusive verdadeiros(**True**) ou falsos(**False**).

MÓDULOS E PACORES/BIBLIOTECA

Módulos são funções internas ou externas ao Python, por padrão várias já vêm com a instalação, mas há muitos outros que podem ser obtidos através de downloads.

➢ Importar Módulos:

Para importar módulos, basta utilizar o comando:

import *módulo*

Caso queira utilizar apenas uma das ferramentas de um módulo, use:

from *módulo* import *comando*

➢ Usar Comando Importado:

Caso tenha sido importado a biblioteca(módulo) **inteira**, é necessário colocar o nome do módulo antes do comando:

módulo*.*comando()

Se tiver importado apenas um comando, pode usar ele como qualquer outra função normalmente.

➢ Modularização:

Caso crie uma ou mais funções em um arquivo **.py,** ele poderá ser considerado um módulo, podendo importá-lo a qualquer momento em seu projeto(caso esteja no mesmo projeto).

➢ Pacotes(bibliotecas):

Pacotes ou bibliotecas são vários módulos em um, eles ficam em uma pasta principal e nessa pasta existem várias subpastas, cada uma com um módulo, com uma ou várias funções. Os **pacotes** são recomendados apenas para projetos enormes, já que ele armazena muitas funções.

Dentro de uma desses pacotes, deve conter um arquivo chamado **__init__.py**. Esse arquivo é onde deve conter as funções (*def* 's). Também podem haver outras pastas dessa dentro dessa pasta, elas seriam as subpastas(também contendo o arquivo para o módulo).

INTERACTIVE HELP

Utilizando a função ***help()***, você pode descobrir as funcionalidades de alguma ferramenta específica. Basta escrever o comando dentro dos parênteses da função.

Código de exemplo:

```
help(input)
help(print)
help(len)
```

Para outras informações, pode utilizar:

- ***print(*função)*.__doc__)***

➢ Docstring ’s:

Para caso de você criar uma **função** e queira deixar o manual dos comandos e funcionalidades, use uma **Docstring**. Basta colocar três aspas duplas abrindo e mais três fechando abaixo da linha de declaração da função, logo abaixo do ***def***. Tudo escrito entre as aspas será de instrução para o usuário, aparecendo quando o mesmo digitar ***help(*função*)***.

Código de exemplo:

```
def maiorMenor(lista):
    """
    :param lista:
    :return: Sem retorno
    Calcula o maior e o menor valor de uma lista.
        Para utilizar, é necessário uma lista apenas com números.
    """
    c = 0
    for v in lista:
        c = c + 1
        if c == 1:
            maior = v
```

```
            menor = v
        else:
            if v > maior:
                maior = v
            if v < menor:
                menor = v
    print('=-~=' * 7)
    print(f' - Maior valor : \033[34m{f"{maior}":>2}\033[m')
    print(f' - Menor valor : \033[34m{f"{menor}":>2}\033[m')
    print('=-~=' * 7)


l = [0, 5, 2, 7, 9, 8]
maiorMenor(l)
print(help(maiorMenor))
```

Resultado:

=-~==-~==-~==-~==-~==-~==-~=

- Maior valor : 9

- Menor valor : 0

=-~==-~==-~==-~==-~==-~==-~=

Help on function maiorMenor in module __main__:

maiorMenor(lista)

:param lista:

:return: Sem retorno

Calcula o maior e o menor valor de uma lista.

Para utilizar, é necessário uma lista apenas com números.

TRATAMENTO DE ERRO / EXCEÇÕES

Erros acontecem o tempo inteiro na produção de um programa, mas existe uma solução para tratá-los. -> ***try***: ->***except***: -> ***else***: >***finally***:

➢ Teoria:

Erros de escrita, como primt ou imput acontecem, mas erros do programa, quando tudo está digitado de forma "certa" mas ainda resulta em um erro, é chamado de **exceção**. **Exceções** podem ocorrer por diversos motivos, como tentar ler um valor inteiro e o usuário digitar um valor *float* ou até mesmo alfabéticos, o que seriam *strings*, o programa se encerraria com um erro de valor (***ValueError***).

Mas se você "pedir" ao invés de "mandar" o computador, esse problema pode ser resolvido.

- ***try e except:***

Abaixo do ***try***, deve ficar seu bloco, esse bloco será a **operação**. O computador

Irá tentar executar esse bloco, se no fim ele estiver com um erro, ele passará para o bloco do ***except***, que deverá estar logo abaixo, como um *if* e *else*.

Abaixo do ***except***, também deve conter um bloco, esse será o bloco de **falha**, ou seja, caso a execução do ***try*** conter uma **exceção**, esse bloco será executado.

Código de exemplo:

```
try:
    a = int(input(' > Valor 1: '))
    b = int(input(' > Valor 2: '))
    resultado = a / b
except:
    print('ERRO de DIGITAÇÂO, digite apenas números')
print(f'Resultado: {resultado:.1f}')
```

Considerando o código acima, o primeiro bloco será uma execução, e só irá partir para o segundo caso essa execução tenha um erro(exceção). Existe mais de um tipo de erro aqui.

Caso tente dividir 3 por 0 (3 / 0), por exemplo, ele mostrará o erro ***ZeroDivisionError***, ou se pedir para usar uma variável não declarada: ***NameError***, enfim, existe diversos erros, mas sabendo qual o ocorrido você pode criar uma forma de prevenilo.

Comando de exemplo:

```
except Exception as erro:
    print(erro.__class__)
```

Exception as é o comando para descobrir o motivo de uma exceção, sendo mais útil em uma situação de teste. A última palavra "erro" é uma variável, podendo ser qual achar melhor. Abaixo no comando *print*, está a chamada para a exceção para mostrá-la na tela.

Comando de exemplo:

```
try:
    a = int(input(' > Valor 1: '))
    b = int(input(' > Valor 2: '))
    resultado = a / b
except ZeroDivisionError:
    print('ERRO | Não pode dividir por zero! | ERRO')
except Exception as erro:
    print(erro.__class__)
```

Ao lado de um ***except***, pode ser colocado o nome de uma exceção, e o bloco desse ***except*** só irá executar caso o ***try*** acima tenha ocorrido esse erro(exceção), além disso, abaixo do ***try*** podem existir vários ***except's***.

No caso, ele irá executar uma mensagem de erro personalizada caso o valor que seja o denominador da divisão seja **zero**. Para a mesma execução desse bloco para mais de um erro, basta colocar vírgula após os nomes ao lado do ***except***, e também colocando-os entre parênteses.

Código de exemplo:

```
except (ZeroDivisionError, NameError):
```

- ***else* (Para tratamento de erro)*:***

O bloco do ***else*** será a continuação caso o ***try*** não tenha ocorrido nenhum erro. Não é obrigatório e deve existir apenas um em uma única estrutura de tratamento.

- ***finally:***

Será executado ao fim de tudo acima, com ou sem erro. Não é obrigatório, mas só poderá conter um em uma única estrutura de tratamento.

ABRIR ARQUIVOS

- ***with:***

Utilizado para garantir a finalização de recursos adquiridos. Sendo assim, o with é usado para abrir arquivos.

➢ **Arquivos de texto:**

➢ ***Editar arquivo:***

(encoding='utf-8')

Write > ADICIONA/SUBSTITUI (arquivo.write('*mensagem*')

Read > LER DOCUMENTO INTEIRO (arquivo.read())

Readlines > LER LINHA POR LINHA (FOR) (arquivo.readlines())

Readline > LER APENAS UMA LINHA (FOI BOM, NA VERDADE) (arquivo.readline())

CORES

Para utilizar cores, pelo menos de forma mais simplificada, é utilizado o **ANSI**, e todo comando em **ANSI** começa com \033[style; text; back m

- **Style (estilo):**

Os códigos de **estilo** são os estilos das fontes, como sublinhado e negrito. Os códigos são:

0 - None (padrão)

1 - Bold (negrito)

2 - Underline (sublinhado)

3 - Negative (negativo) > Utilizado para inverção de cores.

- **Text (texto):**

O **text** é as cores das letras, e sua lista de códigos de cores é de 30 a 37. Veja a lista abaixo:

30 - Branco

31 - Vermelho

32 - Verde

33 - Amarelo

34 - Azul

35 - Roxo

36 - Ciano

37 – Cinza

- **Back (background ou fundo):**

O **back** é a cor de fundo, e sua lista de código é de 40 a 47, na mesma ordem das cores do **text**, veja a lista:

40 - Branco

41 - Vermelho

42 - Verde

43 - Amarelo

44 - Azul

45 - Roxo

46 - Ciano

47 - Cinza

ORIENTAÇÃO A OBJETO

VALIDAÇÃO DE DADOS