



```

4     else:
5         print(f"Status: {response.status_code}\n")
6
7         # using BeautifulSoup to parse the response object
8         soup = BeautifulSoup(response.content, "html.parser")
9
10        # find all images in the soup
11        images = soup.find_all("img", alt="Post image")

```

PYTHON (UDEMY)

<https://www.udemy.com/course/python-3-do-zero-ao-avancado/learn/lecture/15099532#overview>

● Seção 01: Introdução

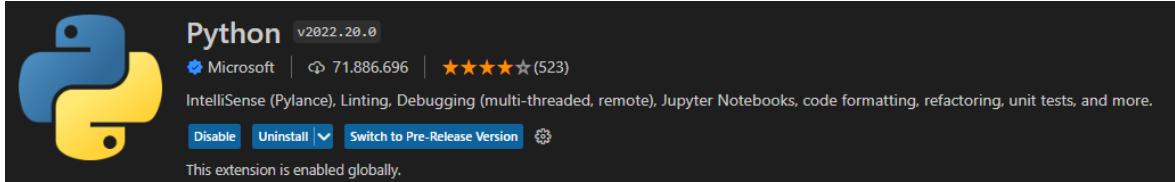


● Seção 02: Python e VSCode

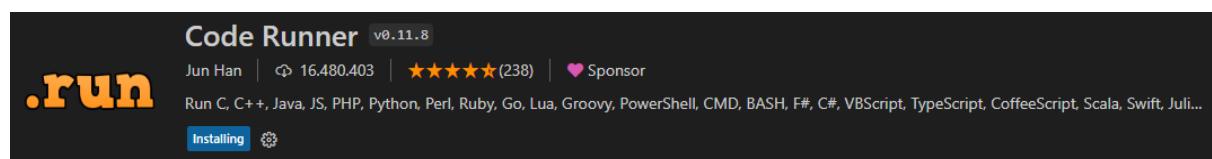


■ Extensões

Python



Code Runner



```
// settings.json
{
    "code-runner.runInTerminal": true,
    "code-runner.clearPreviousOutput": true,
    "code-runner.executorMap": {
        "python": "cls ; python -u",
    },
    "code-runner.ignoreSelection": true
}
```

Settings.json (configurações do VS Code)

```
// settings.json
{
```

```

    "editor.fontSize": 18,
    "explorer.compactFolders": false,
    "workbench.iconTheme": "material-icon-theme",
    "workbench.colorTheme": "Nord",
    "code-runner.runInTerminal": true,
    "code-runner.clearPreviousOutput": true,
    "code-runner.executorMap": {
        "python": "cls ; python -u",
    },
    "code-runner.ignoreSelection": true,
    "python.defaultInterpreterPath": "python",
    "diffEditor.wordWrap": "off",
    "editor.wordWrap": "on"
}

```

● Seção 03: Introdução ao Python e Lógica de programação



Extensão do arquivo: file.py

Comentário: “# comentários”

DocString: utilizado para comentar multi linhas, lembrando que não é um comentário. O interpretador lê, mas não faz nada com essas anotações, apenas guarda na memória.

```

"""
docstrings (aspas duplas)
"""

# ou
'''

docstrings (aspas simples)
'''
```

print(): por padrão, o print já adiciona um separador entre os argumentos, mas ele pode ser alterado utilizando o sep.

```

print(12, 34) # 12 34 (padrão)
print(12, 34, sep = "-") # 12-34
```

Também é possível alterar o que ocorre no final do comando print, com o end.

```

# \r\n -> CRLF (padrão windows)
# \n -> LF (padrão unix)

# \n (padrão)
print(56, 78, sep = "-", end = '\n')
print(78, 1011, sep = "-", end = '\n')

"""

saída
56-78
78-1011
"""

# outros caracteres
print(56, 78, sep = "-", end = '##')
print(78, 1011, sep = "-")

"""

saída
56-78##78-1011
"""
```

■ Tipos de Dados

O python é linguagem de programação **DINÂMICA** (a linguagem já sabe o tipo da informação que está sendo passada.) e **FORTE**.

Strings → str

```
"rafael" # o python reconhece que isso é uma string
```

caractere de escape: `\`

```
print("Rafael \"Oliveira") # Rafael "Oliveira  
""" \ -> diz para não interpretar o próximo caractere ir  
ao próximo. """
```

E se a `\` precisa ser mostrada? Utiliza-se o `r`

```
print(r"Rafael \"Oliveira\"") # Rafael \"Oliveira\"
```

Uma maneira mais simples, é utilizar aspas duplas dentro de aspas simples, ou ao contrário.

```
print('Rafael "Oliveira") # Rafael "Oliveira"
```

Int e Float

int: números inteiros, positivos e negativos → [10](#)

float: números com ponto flutuante (casas decimais), positivos e negativos → [10.1](#)

Para saber o tipo de dado, utiliza-se a função `type`

```
print(type(10)) # <class 'int'>  
print(type(10.1)) # <class 'float'>  
print(type("Rafael")) # <class 'str'>
```

Boolean → Bool

Duas respostas possíveis, `True(sim)` ou `False(não)`

```
print(10 == 10) # Sim -> True  
print(10 == 11) # Não -> False  
print(type(10 == 10)) # <class 'bool'>
```

Coerção/Conversão de tipos

Converter um tipo de dado para outro, também conhecido como: type conversion, typecasting, coercion.

Str, int, float e bool são tipos **primitivos** e **imutáveis** (não podem ser alterados ao longo do programa)

Existem funções que convertem um tipo em outro

```
print(1 + 1) # 2  
print('oi' + ' Rafael') # oi Rafael  
  
print('1' + 1) # erro de tipos, pois não se pode concatenar str com int, somente str com str  
  
# str -> int  
int('1') # -> 1  
print(int('1') + 1) # 2  
  
# str -> float  
float('1') # -> 1.0  
print(float('1') + 1) # 2.0  
  
# str -> bool  
print(bool('')) # -> False -> str vazia é considerada False  
print(bool(' ')) # -> True -> str com conteúdo é considerada True  
  
# int -> str  
str(11) # -> '11'  
print(str(11) + 'b') # 11b
```

Variáveis

São usadas para **salvar algo na memória do computador**. A **PEP8** (guia de estilos do python) indica que o nome das variáveis sejam com letras minúsculas, podendo utilizar números e *underline*.

O sinal de `=` é um **operador de atribuição**, ou seja, atribui um valor à variável.

Ex: `nome_variavel = expressao`

```
nome_completo = "Rafael Oliveira" # str
idade = 18 # num int
maior_de_idade = idade >= 18 # expressao

print('Nome: ', nome_completo, 'Idade: ', idade, 'Maior de Idade: ', maior_de_idade) # Rafael Oliveira 18
```

■ Operadores Aritméticos (matemáticos)

```
adicao = 10 + 12 # 22
subtracao = 22 - 10 # 12
multiplicacao = 10 * 2 # 20
divisao = 10 / 3 # 3.33333 (o retorno sempre é float)
divisao_inteira = 10 // 3 # 3.0
exponenciacao = 2 ** 10 # 1024
modulo = 55 % 2 # 1 (resto da divisão)
```

(+) Concatenação

"Apenas strings" `+` "devem ser concatenadas" `+` "com strings"

```
# concatenação (+)
concatenacao = 'A' + 'B' + 'C'
print(concatenacao) # ABC
```

(*) Repetição

Um inteiro `*` "é a string que deve ser repetida"

```
# repeticao (*)
r_dez_vezes = 'R' * 10
print(r_dez_vezes) # RRRRRRRRRR

rafael_duas_vezes = 'Rafael' * 2
print(rafael_duas_vezes) # RafaelRafael
```

Precedência entre os operadores aritméticos

Cada operador é executado de acordo com sua precedência, ou seja, sua hierarquia.

```
# 1 - parenteses (n + n)
# 2 - exponenciacao **
# 3 - * / // % (com multiplicação e divisão na mesma conta, a ordem será da esquerda pra direita)
# 4 - + -

conta_1 = 1 + 1 ** 5 + 5 # 1 ** 5 = 1 -> +1 = 2 -> +5 = 7
conta_2 = (1 + 1) ** (5 + 5) # 1024

print(conta_1)
print(conta_2)
```

`...` → Ellipsis, usado pra indicar algum código que será escrito

```
altura = 1.85
peso = 90
imc = ...
```

■ Formatação de Strings (f-strings)

utilizando o `f'texto {variavel}'`

```
texto = f'{nome}, tem {altura} de altura\npesa {peso} quilos e seu IMC é {imc}'
```

quantas casas decimais depois da vírgula → `[variavel:.2f]` → duas casas decimais → 1.85

```
texto = f'{nome}, tem {altura:.2f} de altura\npesa {peso} quilos e seu IMC é {imc}'  
print(texto)  
  
# com vírgula  
valor = 10050.4  
print(valor:,.2f) # -> 100,050.4 -> cem mil e 50
```

utilizando o `.format()`

Tudo em python é um objeto, e objeto têm métodos dentro dele, ou seja, algumas ações que podem ser realizadas, por exemplo, as strings tem o método `format()`

```
a = 'A'  
b = 'B'  
c = 1.1  
  
formato = 'a = {} b = {} c = {:.2f}'.format(a, b, c)  
print(formato) # -> a = A b = B c = 1.10  
# a,b e c são argumentos  
# seguindo a ordem, dentro da primeira chave (a = {}) vai vir o  
# primeiro argumento passado em format(), que no caso, é o a.  
  
# ou  
string = 'a = {} b = {} c = {}'  
formato = string.format(a, b, c)  
print(formato) # -> a = A b = B c = 1.1  
  
# utilizando os índices  
string = 'a = {0} a = {0} a = {1} c = {2}'  
formato = string.format(a, b, c)  
print(formato) # -> a = A a = A a = B c = 1.1  
  
# erro  
# string = '\na = {} b = {} c = {} {}'  
# formato = string.format(a, b, c)  
# daria um erro de "index out of range", pois esta buscando algo que já acabou  
# foram passadas 4 chaves (a = {} b = {} c = {} {}), para 3 argumentos (a, b, c), a ultima chave, ficaria sem argumento  
  
# utilizando parametros nomeados  
string = 'a = {nome1} b = {nome2} c = {nome3}'  
formato = string.format(nome1 = a, nome2 = b, nome3 = c)  
print(formato) # -> a = A b = B c = 1.1
```

Input

Para receber dados do usuário, utiliza-se a função `input()`. E os inputs sempre vão retornar uma str, independente do que for digitado.

```
nome = input('Qual o seu nome? ') # usuário digitou 'Rafael'  
  
print(f'O seu nome é {nome}') # O seu nome é Rafael  
print(f'O seu nome é {nome=}') # O seu nome é nome='Rafael'
```

```
# números no input  
num1 = input('Digite um número: ') # 10  
num2 = input('Digite outro número: ') # 5  
print(f'A soma dos números é: {num1 + num2}') # 105 -> como são duas strs, ele vai concatenar, e não somar  
  
int_num1 = int(num1)  
int_num2 = int(num2)  
print(f'A soma dos números é: {int_num1 + int_num2}') # 15
```

E porque não converter direto no input? O código poderia ser quebrado antes mesmo de ser realizada alguma verificação. Pois, se o usuário digita “a” por exemplo, o código já retornaria um erro, pois não tem como converter “a” em inteiro. O mais recomendado é usar uma estrutura condicional (if / elif ... else) depois que o usuário digitar para verificar se ele realmente digitou um número.

Estrutura condicionais (if / elif ... else)

Com as estruturas condicionais, é possível mudar o fluxo do código.

```
entrada = input('Você quer "entrar" ou "sair"? ')

if entrada == 'entrar':
    print('Você entrou no sistema')
elif entrada == 'sair':
    print('Você saiu do sistema')
else:
    print(f'A opção "{entrada}" é inválida. Digite "entrar" ou "sair".')
```

■ Operadores relacionais (comparativos)

Servem pra **fazer comparações**, sempre retornando **True** ou **False**.

```
maior = 2 > 1 # True
maior_ou_igual = 2 >= 2 # True
menor = 4 < 2 # False
menor_ou_igual = 2 <= 2 # True
igual = 'r' == 'r' # True
diferente = 'a' != 'a' # False
```

■ Operadores lógicos

And (e)

Todas as condições precisam ser verdadeiras. Ou seja, se qualquer valor for considerado falso, toda a expressão é dada como falsa.

```
entrada = input('[E]ntrar [S]air: ')
senha_digitada = input('Senha: ')

senha_correta = '1234'

if entrada == 'E' and senha_correta == '1234':
    print('Entrar')
else:
    print('Sair')
```

Valores **considerados falsos** no python (falsys)

```
# (falsys)
0
0.0
'
False
None
```

O python **checa valor por valor, quando ele se depara com um False**, no caso do operador and, **ele para e retorna o False**. Isso gera uma economia de recurso.

```
# Avaliação de curto circuito
print(True and False and True) # retorno -> False
print(True and 0 and True) # retorno -> 0
```

Or (ou)

Qualquer valor verdadeiro, considera a expressão toda verdadeira.

```
entrada = input('[E]ntrar [S]air: ')
senha_digitada = input('Senha: ')

senha_correta = '1234'

if (entrada == 'E' or entrada == 'e') and senha_correta == '1234':
    print('Entrar')
else:
    print('Sair')
```

O python **checa valor por valor, quando ele se depara com um True**, no caso do operador or, **ele para e retorna o True**. Isso gera uma economia de recurso.

```
# Avaliação de curto circuito
print(True or False or 0) # retorno -> True
print(0 or False or 0.0 or 'r' or True) # retorno -> 'r'
```

Not (negação)

Usado para inverter expressões, ou seja, o que for False vira True, e o que for True vira False.

```
# not True -> False
# not False -> True

senha = input('Senha: ')
if not senha:
    print('você não digitou nada')

print(not True)
print(not False)
```

In e Not in

Esses operadores são **utilizados para verificar se aquele valor está entre alguma expressão**.

Strings são iteráveis, ou seja, pode se navegar item por item.

```
# 0 1 2 3 4 5 -> índices positivos
# R a f a e l
#-6 -5 -4 -3 -2 -1 -> índices negativos

nome = 'Rafael'
print(nome[2]) # -> f
print(nome[-2]) # -> e

# Nesse caso, o python vai checar letra pro letra para verificar se o 'R' está entre as letras do nome
print('R' in nome) # True
print('y' in nome) # False
print('afa' in nome) # True
print('ele' not in nome) # True

# ex
nome = input('Digite um nome: ')
encontrar = input('Digite o que deseja encontrar: ')

if encontrar in nome:
    print(f'{encontrar} está em {nome}')
else:
    print(f'{encontrar} não está em {nome}')
```

■ Interpolação básica de strings (%tipo_do_dado)

Formatação de um jeito diferente do format.

```
"""
s - string
d e i - int
f - float
x(gera um hexa minúsculo) e X(gera um hexa maiúsculo) - Hexadecimal (ABCDEF0123456789)
"""

nome = 'Rafael'
preco = 1000.928356334
texto_montado = '%s, o preço é R$%.2f' %(nome, preco)
print(texto_montado) # Rafael, o preço é R$1000.92
print('O HEXADECIMAL de %d é %04X' %(15, 15)) #ou somente %X, colocando %04X ele preenche o que faltar, ou %08X
```

■ Formatação básica de strings

```
"""
Formatação básica de strings utilizando o f strings

```

```

s - string
d - int
f - float
.<número de dígitos>f
x ou X - Hexadecimal
(Caractere)(><^)(quantidade)
> - Esquerda
< - Direita
^ - Centro
= - Força o número a aparecer antes dos zeros
Sinal - + ou -
Ex.: 0>-100,.1f
Conversion flags - !r(__repr__) !s(__str__) !a(__asc__)
"""

variavel = 'ABC'
print(f'{variavel}') # ABC
print(f'{variavel:$>10}') # #####ABC
print(f'{variavel:#<10}') # ABC#####
print(f'.{variavel:^10}.') # . ABC .

print(f'{1000.48342342234:0=>10,.1f}') # +001,000.5
print(f'0 hexadecimal de 1500 é {1500:08X}')

```

■ Fatiamento de strings e função len()

[i: f: p] [: :]

```

# Fatiamento [i: f: p] [ : : ]
# i -> inicio f -> final p -> passo (de quantos em quantos caracteres)

# 012345678
# Ola mundo
#-987654321
variavel = 'Ola mundo'
print(variavel[4:8]) # mund -> nesse caso, o índice final não é incluído, ou seja, o 8, então esta indo do 4 ao 7
print(variavel[4:]) # mundo -> omitindo o final, ele sabe que tem que ir até o final da string
print(variavel[:5]) # Ola m -> omitindo o inicio, ele sabe que tem que começar no inicio da string

# len, retorna a quantidade de caracteres da string
print(len(variavel)) # 9

#
print(0:len(variavel):1) # Ola mundo
print(0:len(variavel):2) # Oamno
print(-1:-10:-1) # odnum alo

```

Exercício

```

# Exercício

# Peça ao usuário para digitar seu nome
nome = input('Digite seu nome: ')
# Peça ao usuário para digitar sua idade
idade = input('Digite sua idade: ')

# se nome e idade forem digitados, exiba:
if nome and idade:
    # seu nome é {nome}
    print(f'Seu nome é {nome}')
    # seu nome invertido é {nome invertido}
    print(f'Seu nome invertido é {nome[::-1]}')
    # se o nome contém ou não espaços
    if ' ' in nome: print(f'Seu nome, {nome}, contém espaços')
    else: print(f'Seu nome, {nome}, não contém espaços')
    # seu nome tem {n} letras
    print(f'Seu nome tem {len(nome)} letras')
    # a primeira letra do seu nome é {letra}
    print(f'A primeira letra do seu nome é {nome[0]}')
    # a última letra do seu nome é {letra}
    print(f'A última letra do seu nome é {nome[len(nome)-1]}') # ou nome[-1]
# se nada for digitado em nome ou idade:
else:
    print('Desculpe, você deixou campos vazios.')

```

■ Introdução ao try...except

O try significa **tentar executar o código** e o except é quando **ocorreu algum**

erro ao tentar executar o código. Erros em python são chamados de exceções.

```
numero_str = input('Vou dobrar o número que você digitar: ')

try:
    print('STR: ', numero_str)
    numero_float = float(numero_str)
    print('FLOAT: ', numero_float)
    print(f'O dobro de {numero_str} é {numero_float * 2}')
except:
    print('Isso não é um número')

# no except também é possível capturar o erro que ocorreu
# isso faz com que o 1º ocorra que ocorra já gere uma exceção, alterando o fluxo do código, fazendo com que o erro seja tratado, ou seja, é possível tratar diferentes tipos de erros
```

■ Variáveis, constantes e complexidade de código, boas práticas

CONSTANTES são variáveis que não vão mudar. No python não tem um comando específico que aponte uma constante, mas existe uma convenção em que **utiliza-se LETRAS MAIUSCULAS para indicar constantes**.

```
velocidade = 61
local_carro = 90

RADAR_1 = 60 # velocidade máxima do radar
LOCAL_1 = 100 # local onde o radar 1 está
RADAR_RANGE = 1 # a distância onde o radar pega

velocidade_carro_passou_radar_1 = velocidade > RADAR_1
carro_passou_radar_1_limite_inferior = local_carro >= (LOCAL_1 - RADAR_RANGE)
carro_passou_radar_1_limite_superior = local_carro >= (LOCAL_1 + RADAR_RANGE)
carro_passou_radar_1 = carro_passou_radar_1_limite_inferior and carro_passou_radar_1_limite_superior

carro_multado_radar_1 = carro_passou_radar_1 and velocidade_carro_passou_radar_1

if velocidade_carro_passou_radar_1:
    print('A velocidade excedeu o limite do radar 1')

if carro_passou_radar_1:
    print('O carro passou no radar 1')

if carro_multado_radar_1:
    print('Carro multado')
```

■ (ID) Identidade de um elemento na memória

```
v1 = 'r'
v2 = 'r'
print(id(v1)) # 4303610608
print(id(v2)) # 4303610608
```

O python tenta ser o mais eficiente possível, ou seja, se duas variáveis diferentes, tem o mesmo valor, ele atribui as duas a mesma identidade, até que as variáveis tenham valores diferentes.

■ Flags, is, is not e None

Por exemplo, **se eu quero saber se meu código passou no if**, utiliza-se um sinalizador, uma **flag**, ou seja, uma bandeira.

```
condicao = True
passou_no_if = None

if condicao:
    passou_no_if = True # bandeira
    print('Faça algo')
else:
    print('Não faça algo')

print('passou_no_if, passou_no_if is None') # is None -> == None
```

Imagina que o if com a condição está em outro módulo separado, seria utilizado a bandeira para sinalizar se passou ou não no if.

```
if passou_no_if is None:  
    print('Não passou no if')  
  
if passou_no_if is not None:  
    print('Passou no if')
```

■ Tipos built-in, documentação, tipos imutáveis, métodos de string

Os tipos **built in** é o nativo do python, ou sejam **já vem inserido nele, sem a necessidade de instalações.**

Built-in Types

The following sections describe the standard types that are built into the interpreter. The principal built-in types are numerics, sequences, mappings, classes, instances and exceptions. Some collection classes are mutable. The methods that add, subtract, or rearrange their members in place, and don't return a specific

 <https://docs.python.org/pt-br/3/library/stdtypes.html>



Os objetos do tipo **int, float, string, bool** são **imutáveis**. Isto significa que objetos deste tipo **não podem ter seus valores alterados.**

```
string = 'Rafael'  
# não teria como por exemplo, atribuir a posição [3] dessa string, um novo valor  
# string[3] = 'ABC' # isso não seria aceito pois a str é um tipo imutável.  
  
# o que daria pra usar, é a concatenação, junto ao fatiamento de string  
outra_string = f'{string[:3]}ABC{4:}'  
  
print(string) # Rafael  
print(outra_string) # RafABCel
```

Todos esses tipos são objetos, e **existem ações que podem ser realizadas dentro desses objetos**, chamadas de **métodos**.

Métodos de string

Os métodos são seguidos de **um ponto, o nome do método e os parenteses que indicam ser uma função**. Todos os métodos estão presentes na [documentação oficial do python](#).

.capitalize(): retorna a string com o primeiro caractere em maiúsculo.

```
string = 'rafael'  
print(string.capitalize()) # Rafael
```

.zfill(): preenche com zeros a esquerda até a quantidade de caracteres desejada

```
print(string.zfill(10)) # 0000rafael
```

■ Estrutura de repetição (while e for)

While (enquanto)

Executa uma ação enquanto uma condição for verdadeira

while + break: o **break** faz sair do laço, independente da condição do while ser verdadeira ou não.

```
condicao = True  
  
while condicao:  
    nome = input('Qual o seu nome: ')  
    print(f'Seu nome é {nome}')  
  
    if nome == 'sair':  
        break # faz sair do laço
```

Operadores aritméticos com operadores de atribuição

```
"""
OPERADORES DE ATRIBUIÇÃO COM OPERADORES ARITMÉTICOS
(funciona com numeros e strings)
+=
-=
*= 
/= 
//=
**=
% =
"""

i = 0
while i < 10:
    i += 1 # i = i + 1
    print(i)

print(f'Acabou o while, {i=}')

```

while + continue: quando o código bate no **continue**, ele ignora o resto do código e volta no começo do laço, ou seja, volta no while para testar a condição. Pode-se ter mais de um continue dentro do laço. As vezes se tem um while dentro do outro, então, tanto o break quanto o continue são aplicados no laço mais próximo deles.

```
i = 0
while i < 100:
    i += 1 # i = i + 1

    if i == 6: continue # 1 2 3 4 5 7 8 9... 40 Acabou o while, i = 40
    print(i)
    if i == 40: break

print(f'Acabou o while, {i=}')

```

while + while: laços internos.

```
qtd_linhas = 5
qtd_colunas = 5

linha = 1
while linha <= qtd_linhas:
    coluna = 1
    while coluna <= qtd_colunas:
        print(f'{linha=} X {coluna=}')
        coluna += 1
    linha += 1

```

Exercício

```
"""
Calculadora com while

- pedir 1º número pro usuário
- pedir 2º número pro usuário
- pedir o operador ao usuário
- somente permitir adição, subtração, multiplicação e divisão
-
"""

while True:
    num1 = input('Digite o 1º número: ')
    num2 = input('Digite o 2º número: ')
    op = input('Digite o operador da conta (+-/*): ')

    numeros_validos = None # flag
    operadores_permitidos = '+-/*'
    resultado = 0

    num1_float = 0.0
    num2_float = 0.0

    ### VERIFICAR OS NÚMEROS ###
    try:
        num1_float = float(num1)
        num2_float = float(num2)
        numeros_validos = True
    except:

```

```

numeros_validos = None

if numeros_validos is None:
    print(f'Um ou ambos os números digitados -> ("{num1}" "{num2}") é inválido.')
    continue # se um dos números for inválido, volta pra pessoa digitar outros números

### VERIFICAR OS OPERADORES ###
if op not in operadores_permitidos or len(op) > 1:
    print(f'{op} -> Operador Inválido')
    continue

### EFETUANDO AS CONTAS ###
if op == '+':
    resultado = num1_float + num2_float
elif op == '-':
    resultado = num1_float - num2_float
elif op == '/':
    resultado = num1_float / num2_float
elif op == '*':
    resultado = num1_float * num2_float
else:
    print('nunca deveria chegar aqui, ta tudo validado')

### EXIBINDO O RESULTADO ###
print(f'{num1_float} {op} {num2_float} = {resultado:.2f}')

### SAIR ###
sair = input('Deseja sair? [s]im ').lower().startswith('s')

if sair: break

```

while + else: recurso específico do python.

```

# while e else, quando o laço é executado completamente, o else é executado

nome = 'Rafael'
i = 0

while i < len(nome):
    letra = nome[i]

    if letra == '': break
    print(letra)
else:
    print('Não encontrei nenhum espaço no nome')

```

Se o código sair do while por algum motivo, por exemplo por conta do break, o else não é executado.

For (para)

Geralmente, quando não sabemos quantas repetições vão ter, utiliza-se o while e para repetições finitas, utiliza-se o for

for + in

```

nome = 'Rafael'

# percorrer cada item da string
for i in nome:
    print(i)

```

for + range: usando intervalos. For e range são funções independentes.

```

"""
range(start, stop, step)
start - digito inicial
stop - digito final (o ultimo digito não é incluído)
step - pula de quantos em quantos

quando se passa apenas um argumento -> range(10), esse 10 é considerado
o 'stop', o start é 0 e o step 1.
"""

numeros = range(0, 10, 2)

for numero in numeros:
    print(numero) # 0 2 4 6 8 -> o ultimo numero não é incluído, 0-9

```

For por debaixo dos panos, como ele funciona

Iterável: elemento que pode entregar uma coisa por vez (str, range, etc). Dentro desse elemento, existe um método chamado `__iter__`. Esse método retorna um objeto que vai saber iterar dentro do elemento, entregando um item por vez.

```
nome = iter('Rafael') # 'Rafael'.__iter__()
print(nome) # <str_iterator object at 0x100957a90>
```

Iterador: quem sabe entregar um valor por vez.

next: quem sabe entregar o próximo valor.

```
nome = iter('Rafael') # 'Rafael'.__iter__()
print(nome) # <str_iterator object at 0x100957a90>

print(next(nome)) # nome.__next__() -> R
print(next(nome)) # a
print(next(nome)) # f ...
```

iter: quem sabe entregar o iterador.

Quando todos os itens são mostrados, um por vez, por meio do `next`, o python retorna um erro `StopIteration`.

```
# for letra in texto
texto = 'Python' # iterável
iterador = iter(texto) # iterador -> __iter__

print("EXEMPLIFICANDO COMO O FOR FUNCIONA, COM MÉTODOS E WHILE")
while True:
    try:
        letra = next(iterador) # __next__
        print(letra)
    except StopIteration: break

print("UTILIZANDO O FOR")
for letra in texto:
    print(letra)
```

for + continue + break + else

```
for i in range(10):
    if i == 2:
        print('i é 2, pulando...')
        continue

    if i == 8:
        print('i é 8, seu else não executará')
        break

    for j in range(1,3)
        print(i, j)

else:
    print('For completo com sucesso')
```

■ Lista mutáveis (list) → array

O tipo `list`, é um **tipo de dado mutável**, são as *listas em python*. Suporta vários valores e de diversos tipos. Alguns dos seus métodos mais úteis são: `append`, `insert`, `pop`, `del`, `clear`, `extend`, `+` entre outros.

```
# list() essa função list() poderia ser utilizada para fazer uma conversão para list
# lista = []
# print(lista, type(lista)) # [] <class 'list'>

# -----0-----1-----2---3---4--
lista = [18, 'Rafael Oliveira', True, [], 1.85]
print(lista) # [18, 'Rafael Oliveira', True, [], 1.85]
print(lista[2]) # Rafael Oliveira

lista[2] = 'Rafa'
print(lista[2]) # Rafa
print(lista) # [18, 'Rafa', True, [], 1.85]
```

Alterando listas pelo índice

del, .append() e .pop()

```
lista = [10, 215, 30, 700]
print(lista) # [10, 215, 30, 700]

# del -> apaga um item da lista
del lista[2]
print(lista[2]) # 700 (a lista é reorganizada)
print(lista) # [10, 215, 700]

# append -> adiciona ao final da lista
lista.append(50)
lista.append(100)
print(lista) # [10, 215, 700, 50, 100]

# pop -> remove o último elemento da lista
lista.pop()
print(lista) # [10, 215, 700, 50]
```

Inserindo itens em qualquer índice da lista com insert

insert

```
lista = [10, 215, 30, 700]
lista.insert(0, 400) # 0 -> o índice que vai ser adicionado, 400 -> o valor que vai ser adicionado

print(lista) # [400, 10, 215, 30, 700]

# se um índice maior que a lista for passado, ele adiciona o valor no final da lista
lista.insert(1000, 5)
print(lista) # [400, 10, 215, 30, 700, 5]

# clear -> limpa a lista
lista.clear()
print(lista) # []
```

Concatenando e estendendo listas

+

```
lista_a = [1, 2, 3]
lista_b = [4, 5, 6]

lista_c = lista_a + lista_b
print(lista_c) # [1, 2, 3, 4, 5, 6]
```

extend

```
"""
lista_d = lista_a.extend(lista_b)
print(lista_d) # None

O método extend() não retorna nada, ou seja, ele trabalha em cima
do próprio elemento que está chamando, nesse caso a lista_a. Ou seja,
não tem como pegar o "lista_a.extend(lista_b)" e jogar em uma variável.
"""

lista_a.extend(lista_b)
print(lista_a) # [1, 2, 3, 4, 5, 6]
```

Cuidados com tipos de dados mutáveis

```
lista_a = ['Rafael', 'Gabriel']
lista_b = lista_a

"""
isso faz com que as duas variáveis apontem para o mesmo
endereço de memória. Ou seja, quando eu alterar algo em uma,
automaticamente irá alterar na outra
"""

lista_a[0] = 'João'
print(lista_b) # ['João', 'Gabriel']
```

[copy](#)

```
lista_c = ['Rafael', 'Gabriel', 'João', 'Maria']
lista_d = lista_c.copy()

lista_c[0] = 'Rafinha'
print(lista_c) # ['Rafinha', 'Gabriel', 'João', 'Maria']
print(lista_d) # ['Rafael', 'Gabriel', 'João', 'Maria']
```

For/in com listas

```
lista = ['Rafael', 'Gabriel', 'João']

for i in lista:
    print(i)
```

Mostrando os índices

```
lista = ['Rafael', 'Gabriel', 'João']
indice = 0

for i in lista:
    print(f'{{indice}} -> {i}') # [0] -> Rafael
    indice += 1

# outro jeito
indices = range(len(lista))
print(indices) # range(0,3)

for indice in indices:
    print(indice, lista[indice]) # 0 Rafael
```

■ Empacotamento e Desempacotamento

Todos os *objetos iteráveis* podem ser desempacotados.

```
nomes = ['Rafael', 'João', 'Luiz']
nome1, nome2, nome3 = nomes

print(nome1) # Rafael
print(nome2) # João
print(nome3) # Luiz

# ou
nome1, nome2, nome3 = ['Rafael', 'João', 'Luiz']
print(nome1) # Rafael

# pegando só o 1º valor, para isso, eu preciso indicar o que fazer com o resto
# para isso, o python empacota o resto, o qual nos indicamos uma variável com um asterisco
# para indicar que ali é pra ser tratado o resto
nome1, *resto = ['Rafael', 'João', 'Luiz']
print(nome1) # Rafael
print(resto) # ['João', 'Luiz']

"""
geralmente essa variável "resto" não vai ser utilizado
em python, quando uma variável não é utilizada, é aplicada uma convenção
para essa situação, que é utilizar um "_". Isso vai indicar que a variável funciona
mas não será utilizada
"""
nome1, *_ = ['Rafael', 'João', 'Luiz']

# se eu quisesse só o 3º valor
_, __, nome3, *_ = ['Rafael', 'João', 'Luiz']
print(nome3) # Luiz
```

■ Tipo tuple (tuplas)

A **tupla** é uma *lista imutável*, ou seja, quando eu **preciso de uma lista, e que não vá alterar seus valores**, nem adicionar, editar ou excluir, **utiliza-se uma tupla**, que é até mais rápida que uma lista.

```
nomes = 'Rafael', 'João', 'Luiz' #ou ('Rafael', 'João', 'Luiz')
print(nomes) # 'Rafael', 'João', 'Luiz'
print(type(nomes)) # <class 'tuple'>

# conversao de lista pra tupla
dias = ['Segunda', 'Terça', 'Quarta', 'Quinta', 'Sexta']
dias = tuple(dias)
# ou o inverso dias = list(dias)
print(dias) # ('Segunda', 'Terça', 'Quarta', 'Quinta', 'Sexta')
```

■ Enumerate

O **enumerate** é utilizado para **enumerar valores iteráveis**, ou seja, pegar seus índices.

```
nomes = ['Rafael', 'João', 'Luiz']
nomes_enumerados = enumerate(nomes)
print(nomes_enumerados) # <enumerate object at 0x1000fb900>

lista_nomes_enumerados = list(enumerate(nomes))
print(lista_nomes_enumerados) # [(0, 'Rafael'), (1, 'João'), (2, 'Luiz')]

for i in nomes_enumerados:
    print(i) # (0, 'Rafael') (1, 'João')...

for item in enumerate(nomes):
    indice, nome = item
    print(f'{indice} -> {nome}') # [0] -> Rafael

# o for acima pode ser simplificado da seguinte forma
for i, nome in enumerate(nomes):
    print(f'{i} -> {nome}') # [0] -> Rafael
```